A GENETIC ENGINEERING APPROACH TO TEXTURE SYNTHESIS

by

Philip Ray Peterson
B.Sc., Simon Fraser University, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Philip Ray Peterson 1997
SIMON FRASER UNIVERSITY
April 1997

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Philip Ray Peterson

Degree: Master of Science

Title of thesis: A Genetic Engineering Approach to Texture Synthesis

Examining Committee:

Dr. Binay K. Bhattacharya
Chair

Dr. Thomas W. Calvert
Senior Supervisor

Dr. F. David Fracchia
Supervisor

Dr. Brian V. Funt
External Examiner

Date Approved: 17 April 97

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Genetic Engineering Approach to Texture Synthesis

Author:

Philip Ray Peterson

April 17, 1997
(date)

Abstract

Much of the richness and detail found in computer graphics imagery is the result of the application of texture maps to three-dimensional geometric models. In many cases, the best approach is to generate a texture procedurally. However, the definition of a texture synthesis procedure can be a complex task. In addition to defining a functional algorithm, often a large parameter space must be explored.

Techniques based on natural systems are increasingly applied as search procedures in a variety of problems. One such technique, genetic programming, has been used to create procedural texture programs. Under human guidance, both program and parameter space are explored simultaneously. This method, however, has several shortcomings when used alone.

In this thesis, a genetic engineering approach is developed. Based on interactive evolution, the genetic programming paradigm is used to evolve procedural texture programs of varying size and shape, a hybrid genetic algorithm is used to explore the parameter space of target programs, and a data-flow representation is used for the direct manipulation of program structure and variables. A prototype texture design tool implementing this approach is described. The effectiveness of this prototype is discussed, and a number of examples are presented showing how it can be used successfully to design a variety of texture synthesis programs.

In memory of my father,

WILLIAM JOHN PETERSON

Acknowledgements

Many thanks to my senior supervisor, Dr. Tom Calvert, for giving me the support and the freedom to complete this thesis in my own way. Also, to my supervisor, Dr. David Fracchia, for all his help and unwavering enthusiasm. Thanks also to Chris Welman, for providing me with a work environment which both enabled and encouraged me to complete this thesis, and to Troy Brooks, for acting as a sounding board and providing many valuable comments and suggestions.

Finally, I would like to thank my family, my mother and brothers, for their encouragement and support, and especially my wife Samantha, for her unflagging confidence and understanding.

Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

List of Figures ix

List of Algorithms x

1 Introduction 1

1.1 Texture Mapping 2

1.2 Texture Definition 3

1.2.1 Texture Generation 3

1.2.2 Procedural Texture 7

1.3 Motivation 10

1.4 Organization 11

2 Genetic Programming 12

2.1 Fundamental Genetic Concepts 13

2.1.1 Structure 13

2.1.2 Reproduction 14

2.1.3 Evolution 15

2.2 Genetic Algorithms 16

2.2.1 The Basic Genetic Algorithm 16

2.2.2 Real Number Encodings 22

2.3 Genetic Programming . . . 23
2.3.1 The Standard Genetic Programming Paradigm . . . 23
2.3.2 Program Structure . . . 24
2.3.3 Genetic Operators . . . 27
2.4 Interactive Evolution . . . 30
3 Evolving Procedural Textures . . . 31
3.1 Applications of Interactive Evolution . . . 31
3.1.1 Summary . . . 34
3.2 Interactive Evolution of Textures . . . 35
3.2.1 Evolving Images and Textures . . . 36
3.2.2 Evolving Solid Textures . . . 38
3.2.3 Limitations . . . 38
3.3 A Genetic Engineering Approach . . . 41
3.3.1 Prototype Implementation . . . 41
4 A Texture Design Tool . . . 43
4.1 Program Structure . . . 43
4.1.1 Program Components . . . 44
4.1.2 Implementation . . . 49
4.2 Interactive Evolution . . . 51
4.2.1 User Interface . . . 51
4.2.2 Evolving Programs . . . 53
4.2.3 Evolving Parameters . . . 55
4.2.4 Control Parameters . . . 58
4.3 Direct Manipulation . . . 58
4.3.1 User Interface . . . 58
4.3.2 Editing Operations . . . 59
4.3.3 Data-flow Display . . . 60
5 Results . . . 61
5.1 Genetic Engineering Approach . . . 61

5.1.1 Exploration 62
5.1.2 Refinement 64
5.1.3 Adjustment 65
5.1.4 Summary 66
5.2 Examples 69
5.2.1 Image Processing 70
6 Conclusion 74
6.1 Future Work 75
6.2 Summary 76
Appendices
A Function Set 78
B Program Syntax 82
C Example Programs 85
References 91

List of Figures

2.1 Crossover and recombination. 15

2.2 Binary string encoding in a genetic algorithm. 18

2.3 Crossover in a genetic algorithm. 20

2.4 Mutation in a genetic algorithm. 21

2.5 Program tree for $1 - e^{0.01d}$. 25

2.6 Crossover in genetic programming. 28

2.7 Mutation in genetic programming. 29

4.1 Hierarchy within the *image* abstract data type. 45

4.2 Program tree for `(blend (noise 0.1) (noise 0.4) 0.67)`. 47

4.3 Simple texture synthesis programs. 48

4.4 Internal structure of population. 51

4.5 User interface for interactive evolution. 52

4.6 A chromosome corresponding to the parameters of a program tree. 56

4.7 User interface for direct manipulation. 59

5.1 Population evolved using GP. 68

5.2 Population evolved using GA. 68

5.3 Textures generated using the texture design tool. 71

5.4 Textures generated using the texture design tool. 72

5.5 Textures applied to 3D geometry. 73

5.6 Image processing programs. 73

List of Algorithms

2.1 The basic genetic algorithm. 17
2.2 The standard genetic programming paradigm. 24
2.3 Generating a random program. 26

Chapter 1

Introduction

The richness and detail that breathes life into three-dimensional computer graphics is an illusion, and *texture mapping* is one of the most important tools at our disposal for creating this illusion. Since its introduction, the use of texture mapping has increased to the point where it is now considered an integral, if not essential, part of most three-dimensional computer graphics systems. Without texture mapping it would be extremely difficult to achieve the level of realism that is routinely found in the computer graphics images of today.

An increasingly common approach to texture mapping is to define synthesis procedures that generate textures either prior to, or during, the rendering of a three-dimensional scene. However, the specification of such procedures can be a difficult task requiring specialized knowledge.

A promising approach to many problems for which a direct solution method is not applicable, is to use an analogue of natural systems to *evolve* a solution. In this thesis, we investigate how such techniques can be effectively applied and augmented to develop a system for the design of texture synthesis procedures.

1.1 Texture Mapping

In 1974 Catmull [11] introduced the concept of texture mapping. He showed how surface coloration could be determined from a two-dimensional pattern definition function such as a picture. For example, a picture can be *mapped* onto a bivariate patch by associating its coordinate space with the parametric space of the patch. Since the patch is defined parametrically, this mapping is straightforward: Given a point (x,y,z) on a bivariate patch, there is an associated parametric coordinate (u,v) which can be easily transformed into its equivalent texture coordinate (s,t) in the 2D pattern space.

For some geometric descriptions of surfaces, polygonal meshes for example, there may not be an inherent 2D parameterization. In these cases, a projection function can be used to provide the mapping from surface coordinate (x,y,z) to texture coordinate (s,t). Although this can be any function $f:\mathbb{R}^3 \mapsto \mathbb{R}^2$, in practice it is typically either a planar, cylindrical, or spherical projection.

A texture can also be defined as a 3D, rather than 2D, function as described in Peachey [59] and Perlin [60]. Using this model, referred to as *solid* texture, the surface coordinate need not be mapped to a 2D space, but rather, can be used directly.

Texture maps have also proven useful in controlling rendering attributes beyond simple surface color determination. Blinn and Newell [7] used texture maps to simulate the surroundings of an object reflected on its surface (reflection or environment mapping). Textures can also be used to modify surface normals prior to the illumination calculation (bump mapping). Blinn [6] showed how this can be used to simulate a more complex surface than is modeled geometrically. The opacity of an object can also be controlled by a texture (transparency mapping) [25], as can surface perturbation (displacement mapping) [12], and myriad other attributes.

A good indication of the number of uses and flexibility of texture mapping can be found in Cook [12], in which a general model for surface shading based on the evaluation of an expression tree is introduced. In this model, a texture map can be introduced at virtually any point in the shade tree. This allows it to not only be used for any attribute, but more interestingly, to act as a control structure. For example, a texture map defining a decorative pattern could be used to

switch between two different shading algorithms corresponding to different regions of the pattern.

Without question, there are many potential applications for texture mapping. However, as with the application of any technique, it is worth considering how it can be applied in a useful and effective manner.

Texture mapping involves two components: first, a suitable texture must be defined; and second, the texture must be mapped onto the surface. The former provides the motivation for this thesis and will be elaborated on in the following sections. The latter will not be discussed further except where it serves to clarify the subject at hand. The interested reader wishing further details will find useful introductions to texture mapping in Heckbert [33] and Magnenat-Thalmann and Thalmann [50].

1.2 Texture Definition

Methods for texture definition can be roughly divided into two categories: those that are generated as a pre-process and indexed during the rendering phase and those that are evaluated as functions during the rendering phase. It is important to note however, that this is not a precise division, and the distinction can be blurred in many cases. An overview of texture definition methods follows.

1.2.1 Texture Generation

Methods for texture definition based on *generation* are distinguished as such by being encoded as a map prior to the rendering phase. During the rendering phase, the map is indexed using texture coordinates in order to produce a value. Many methods have been proposed for texture generation, but only a few are commonly used. What follows is a concise review of the major methods, their advantages and drawbacks.

Image Acquisition

The most commonly used approach to texture generation is to acquire and encode as a texture map an image from some source[1]. Most often, this is a picture digitized from print, film, or video, but could also be a previously rendered computer graphics image.

After it is acquired, an image is often altered using image processing operators prior to being used as a texture map. For example, an operator may be applied to perform color correction, to increase the contrast, or to adjust the image for better registration in texture coordinates.

Image capture is (so far) the best solution for complex, inhomogeneous maps such as the label for a bottle, satellite imagery for terrain, or an advertisement for a billboard. Computer graphics images generated by an earlier rendering pass can be used very effectively as well. For example, in a scene with many trees, the ones closest to the camera can be modeled geometrically, but ones further away can be planar polygons texture mapped with pre-rendered images of the modeled trees. If done correctly, this can reduce the computational complexity of the scene dramatically with minimal loss of resulting quality.

Image acquisition is also frequently used to generate what we are more likely to think of as *textures*. For example, carefully taken photographs [8] of wood, carpet, or stucco, or repeating patterns like brick, cloth, or wallpaper can be digitized for use as texture maps. While it would seem at first that this should provide very accurate results, there are some associated difficulties. For example, in most instances, a texture will need to be *tiled* to provide adequate coverage of the surface while providing the correct texture scale. Such textures typically exhibit discontinuities where they join and may also exhibit patterning when a number of tiles are visible at once. While there are techniques to minimize these effects, they often do not work well in practice. In addition, a picture of some material may contain unwanted lighting effects due to the environment in which it was photographed or filmed. When applied as a texture these may contradict the lighting for the surface to which it is being applied. These effects can be minimized in a controlled situation but in doing so, images often lose some of the characteristics that made them desirable for use as textures in the first place.

[1] For solid textures, a sequence of images may need to be acquired.

Digital Painting

Digital paint systems are often used in the generation of texture maps. Most commonly they are used by an artist to *paint* a picture (using a digital brush metaphor) for use as a map. In addition, they can be used to *touch-up* captured images to help alleviate some of the problems described earlier in this section regarding image acquisition.

By incorporating the 3D geometry to be mapped into the system, digital painting can be used to help overcome one of the fundamental problems in texture mapping: the potential *warping* of the texture due to the non-uniform relationship between the surface parameterization and the surface geometry[2]. For example, to create an image that will represent facial features when texture mapped onto a bicubic patch model of a head requires that the image compensate for the parameterization when mapped. This can be a very difficult task. Hanrahan and Haeberli [30] proposed a method of direct surface painting to overcome this problem. Using this technique, an artist appears to paint directly on the 3D model, but is actually painting a pre-warped texture image. The brush's location in the texture image is determined by the parametric coordinate of the brush on the three-dimensional model. The resulting texture image is applied using conventional techniques.

Texture Synthesis

Earlier in this section, we described how image capture can be used to generate maps of natural textures such as wood, stone, or dirt. We also explained how standard image processing techniques can be used to modify such images. However, none of these techniques can be used to modify the textures in a way that is based on their *content*. For example, given an image of stones, we can scale that image to make the stones appear larger or smaller overall, but we cannot make the stones more widely varied in size. Similarly, in an image of woven cloth, we cannot make the weave coarser without changing the width of each thread.

Partially in an effort to address some of the shortcomings of image acquisition for texture generation, a number of researchers have developed algorithms for the *synthesis* of texture. Typically, a procedure is used to generate texture images with control over the exact content governed by

[2]This does not apply to solid textures.

some number of independent variables. Many of these methods are based on mathematical models derived from the analysis of natural textures such as those found in Brodatz's album [8]. The following examples, while not exhaustive, should provide an indication of the flavor and diversity of synthesis methods. A detailed survey of the majority of texture synthesis methods can be found in Kaufman and Azaria [39].

Mezei, Puzin, and Conroy [53] devised a method for generating cellular patterns like those found on reptile skin or stone walls using a growth model around distributed nuclei. They also generated textures approximating bark and fur by choosing a number of small *signs* (a collection of geometric primitives) and distributing them with a dense uniform random distribution. Schacter and Ahuja [67] describe a similar technique called a *bombing* model. In this method a plane (the image) is *bombed* with shapes as a random point process.

A syntactic approach is taken by Fu and Lu [21]. In their method, a texture image is comprised of a set of fixed-size tiles or windows, each of which has an associated tree grammar. Moreover, within each window's tree, each node corresponds to a pixel. The placement and distribution of different tile types is governed by another stochastic grammar. Using this method, they show how they can (rather rudimentarily) synthesize a set of textures taken from Brodatz [8].

Kaufman and Azaria [39] also developed a syntactic approach to texture synthesis based on a tile generator. However, instead of a tree grammar, a more general programming language (TSL) is used allowing for complex tilings to be expressed procedurally rather than explicitly.

Yessios [84] describes algorithms for the plotting of stones, wood, plants, and ground material for architectural presentation drawings. While the algorithm varies for each texture, the fundamental principle is the same; an initially regular pattern is subjected to regulated random disturbances. For example, wood grain is synthesized by randomly positioning, scaling, and disturbing a series of concentric ellipses.

In Lewis' [48] *spectrum painting*, a complex texture is generated by first computing a texture sample using a Fourier transform of a painted spectrum, then performing an out-of-order convolution of the sample (windowed to remove edge discontinuities) with a sparse noise. This approach was engineered to be suitable for creating brush and medium textures for digital painting. However, due to the inherent periodicity of the Fourier transform, these textures are easily tiled,

making them also useful for large natural textures such as terrain. A similar technique, *spot noise,* is used by van Wijk [80]. It is derived through the filtering (convolution) of a grid of white noise with controlled variations of a prototype *spot.* In both of these methods there is a strong visual correlation between the control (spot or painted spectrum) and the resulting texture.

A novel method for synthesizing variants on a class of textures is proposed by Lewis [49]. Given a set of input patterns, a neural network is trained to give an objective output that recognizes patterns belonging to a certain class. Textures are synthesized by applying a random input to the network which is then adjusted until the output falls within some pre-defined error bound. Unfortunately, the derivation of class recognizers that ensure membership, yet allow for sufficient variation, proves to be rather difficult.

Both Turk [78] and Witkin and Kass [81] use reaction-diffusion systems for texture synthesis. Employing an analogue of biological growth, these methods have been successful in synthesizing patterns found in nature such as the spots on a leopard or the stripes on a zebra.

Despite the abundance of methods proposed for texture synthesis, in practice they are rarely employed. In some cases, this is simply because the techniques are no longer relevant, or are too computationally expensive. But for the most part, it is because they do not provide sufficient usability. In particular, many of the methods are parameterized in a way that has a far too complex relation to the visual result, or they may simply have too many parameters to be comprehensible or adjusted in any intuitive way.

1.2.2 Procedural Texture

In the previous section, we described several approaches to the definition of texture maps, all of which require the map to be generated prior to rendering. For digitally acquired or painted images this seems a perfectly reasonable, if not necessary, step. Many texture synthesis techniques (all those described in Section 1.2.1) require this as well as they are essentially image *construction* methods. However, there are a number of texture and image synthesis methods that are fundamentally point processes. In particular, the value of a texture map at any texture coordinate is solely a function of a single texture coordinate plus any independent parameters of the texture synthesis procedure. Therefore, there is no need for generation of the map prior to rendering. We

will refer to methods belonging to this class as *procedural* textures.

Procedural textures are not a recent extension. In fact, they were introduced simultaneously with the concept of texture mapping. In particular, Catmull [11] noted that any function of texture coordinates could be used, not just a table lookup (mapping of previously generated textures). For example, in both Fournier, Fussell, and Carpenter [20] and Harayuma and Barsky [32] it is shown how an approximation to fractional Brownian motion can be used to generate stochastic textures. Another example of procedural texture can be found in Gardner[24, 25], in which a short sum of sine waves is used to create naturalistic textures for clouds, trees, and ground-cover.

With the introduction of *solid* textures by Peachey [59] and Perlin [60], the value of procedural evaluation became widely recognized. To begin with, the extension of 2D texture maps to 3D would require a marked increase in storage. Moreover, several of the functions they found useful were more conveniently defined in an arbitrary 3D space. Peachey describes the extension of 2D methods such as Fourier synthesis and bombing to 3D. He also argues that 2D projection mapping techniques can be considered a class of solid textures. Perlin provides several noteworthy examples of procedural textures including fire, water, and marble. Most of these examples are based on his implementation of his now widely used *noise* function.

A major hurdle in using procedural textures was the difficulty of integrating them into existing renderers. As most renderers used one or more fixed shading models, this meant that to use procedural textures, one had to be able to modify the rendering algorithm. This tended to restrict their use to either research environments or to specialty applications such as flight simulators. In order to overcome this difficulty, a change in shader [3] access and architecture would be necessary.

Cook [12] proposed a flexible rendering architecture using *shade trees.* Rather than use a set of fixed shaders, Cook showed how a shader could be implemented as a general expression tree. In a shade tree, a lookup into a pre-defined texture map could be introduced at any node. A procedural texture is essentially any subtree that depends on texture coordinates. The only restriction on procedural textures in this context is that they must be constructed as expression trees themselves.

[3] A *shader* is a procedure that computes a resulting color based on the current rendering state plus some set of attributes for the shading algorithm.

Perlin [60] also proposed a more general shading approach with his *Pixel Stream Editing* language (PSE). Unlike Cook's shade tree language, PSE incorporated a full set of procedural constructs including loops, conditionals, and function calls. In addition, PSE also included support for solid textures. However, unlike shade trees, PSE constructs are evaluated after surface visibility determination, thereby preventing certain texture applications. That aside, its power is shown to great effect in Perlin's many examples of procedural textures.

Ideas from both Cook's shade trees and Perlin's PSE are incorporated into an architecture called *RenderMan*. The *RenderMan* standard [61] is a rendering interface specification encompassing scene and surface description as well as an integrated shading language. The *RenderMan* shading language is a small procedural language (based on a subset of C [41]) similar in scope to PSE. However, it uses a model of light, surface, and atmosphere similar to that of Cook. The principles of the shading language and an implementation of *RenderMan* are described in detail in Hanrahan and Lawson [31]. Descriptions of its use with many examples can be found in Upstill [79].

The inclusion of support for user defined shaders is now widely considered a fundamental component of a state-of-the-art general purpose rendering application. The *RenderMan* interface is available in Pixar's *PhotoRealistic RenderMan* [62], a widely used commercial application. A similar model is implemented in the *mental ray* raytracer [52]. However, instead of using a dedicated shading language, *mental ray*'s interface allows for any dynamic object code with C linkage.

A comprehensive treatment on the design and use of procedural textures can be found in Ebert et al. [18]. Included in this text are a number of useful examples of textures implemented using the *RenderMan* shading language.

Generating textures procedurally has several advantages over the *off-line* generation discussed in the previous section. Since a discrete table need not be defined for the texture space, they can easily be computed using different sampling frequencies of the texture space. This helps procedural textures to be less susceptible to problems of scale. Also, for many types of textures, in particular ones that have a stochastic or time-varying component, a generation procedure requires much less storage than would a texture map with a sufficiently high sampling of the texture space to be useful. Furthermore, procedural textures implemented using a shading language are able to

integrate localized information such as surface normal and derivatives into the texture synthesis procedure. This information is generally not available to an external synthesis application.

However, there are also some disadvantages. Since they are defined as point processes, some synthesis techniques are difficult to apply. For example, any operator requiring regional information (convolution), or any algorithm that is based on a construction technique (bombing models) may require complex or redundant computations. Furthermore, if a texture need be re-evaluated many times and requires computationally costly operations, it may be advantageous to use a precomputed version instead.

1.3 Motivation

Each method for texture definition discussed in the previous sections has its advantages and disadvantages. Moreover, each is the *best* method in some circumstances. For textures that are not possible (or convenient) to define algorithmically, image acquisition is the preferred method, and for the creation of texture maps correlating to surface features, digital painting may be the best choice. However, as should be evident from the previous sections, there are many applications where algorithmically defined textures are useful. Unfortunately, as is the case with many algorithmic methods, they can be difficult to specify.

Specification can be difficult for two reasons. First, even if given a toolkit of applicable operators (as in *RenderMan*), shader design and modification is essentially a *programming* task. Therefore, some amount of technical knowledge is required. Moreover, to apply the principles on which many texture synthesis methods are based necessitates some reasonable level of mathematical sophistication. Second, merely adjusting the independent parameters of a texture synthesis algorithm can be daunting — their relationship to the result may be obtuse, or they may simply be too numerous to comprehend (Gagalowicz and de Ma [22] describe a statistical model of natural textures that involves between 40 and 2000 parameters).

In his paper *Artificial Evolution for Computer Graphics* [69], Sims describes a method for image and texture generation based on evolutionary programming techniques. In this method, an

image or texture is represented by an expression tree which is created and manipulated programmatically — a user interactively guides the design process solely through subjective judgement of the results. This programming paradigm does not require of the user any knowledge of, or contact with, the underlying structure. Nor does it require any direct manipulation of independent parameters. Therefore, this system addresses both of our concerns with difficulty of specification.

Sims' approach is simple to use and can produce complex and engaging results in a short period of time. However, it is difficult to use this system to achieve a *desired* result. In this thesis, we consider extensions to this approach that allow for a more design-oriented system to be developed.

1.4 Organization

Chapter 2 provides an introduction to genetic algorithms and the genetic programming paradigm. Chapter 3 describes previous work using interactive evolution for generating textures and discusses its limitations from a design point of view. It concludes with a description of the work proposed in this thesis to address these limitations. Chapter 4 describes a prototype texture design tool which implements the genetic engineering approach proposed in Chapter 3. Chapter 5 discusses the effectiveness of the prototype and offers examples of its application. Chapter 6 offers some concluding remarks as well as suggestions for future work.

Chapter 2

Genetic Programming

Perhaps in recognition of the immense adaptive power of nature, there is a growing interest in systems that emulate natural processes. As a result, several problem solving methodologies have been proposed that are either abstractions or generalizations of biological or physical systems. These can be used either as alternatives to, or in conjunction with, conventional algorithmic approaches. Of particular interest here is a method for program induction based on the principles of evolution and survival of the fittest as given by Charles Darwin in his classic monograph: *On the Origin of Species by Means of Natural Selection* [13].

In this chapter we will describe the *genetic programming* paradigm for program induction proposed by Koza [43]. Genetic programming can be considered a reformulation of the standard *genetic algorithm* introduced by Holland [36]. The conventional genetic algorithm has been used successfully in a variety of problem domains as well as considered from a theoretical standpoint. Since the essential concepts of the genetic algorithm have direct analogues in genetic programming, we will first describe the standard genetic algorithm as an introduction to genetic programming. As a further introduction, we will begin with a review of some important concepts and terminology from the field of genetics. We will conclude with an introduction to interactive evolution, a variation applicable to both genetic algorithms and genetic programming, and the methodology employed in this thesis.

2.1 Fundamental Genetic Concepts

Genetics is the science of inheritance and variation in organisms — the fundamental mechanisms responsible for evolution in the Darwinian sense. Genetic algorithms and their extension to genetic programming make liberal use of vocabulary and concepts borrowed from the study of genetics. In particular, the existence of a *genetic code*, and the variation resulting from recombination and mutation during the reproductive process are fundamental to these methodologies. In this section we will briefly review the important aspects of genetic structure, reproduction, and evolution that are commonly used as basic principles in both genetic algorithms and genetic programming. It is important to note however, that an extremely simplified version of human genetics is employed as the basis for genetic algorithms. It is this simplified version that is presented here — less as a proper treatment of genetics than as a pedagogical aid in understanding the basis for genetic algorithms. For a more complete and accurate treatment of the subject, the reader is encouraged to consult one of the many standard texts on genetics [23, 51, 65].

2.1.1 Structure

Genetic structures are the *blueprint* for an organism. In other words, they encode the information that serves as a precise specification for every cell of an individual during development. This specification is sometimes referred to as the *genetic code.*

The fundamental genetic structure is the *chromosome.* A chromosome is comprised of a linear arrangement of *genes*, each of which represents some inheritable characteristic. The position on the chromosome of a specific gene is called its *locus*, with each locus corresponding to a specific genetic trait. For each gene, several variants are possible. Each variant, called an *allele*, corresponds to a different form of the same characteristic.

Each cell of an organism (with the exception of sex cells), contains a complete set of chromosomes (where a set is one or more depending on the specific organism). Human cells, for example, have 23 pairs of chromosomes where each pair contains one chromosome from each parent. For the rest of this section, we will continue to use human genetics as a running example of genetics in general.

The genes contained in the complete set of chromosomes form a *genome*, with the information contained within referred to as the organism's *genotype*. The genotype completely defines the internal genetic makeup of an individual. The external expressed traits of an individual — the manifestation of its genotype — is referred to as its *phenotype*.

The information in a chromosome is encoded in macromolecules of deoxyribonucleic acid (DNA). DNA can be visualized as two strands forming a double helix. Along each strand is a sequence of nucleotides. Each nucleotide consists of a phosphate group, a sugar and one of the four bases: Adenine (A), Thymine (T), Cytosine (C), and Guanine (G). The two strands are connected by hydrogen bonds between pairs of nucleotide bases (*base pairs*). These sequences form the so-called genetic alphabet, and hence, serve as the genetic code.

2.1.2 Reproduction

Genetic traits are passed from one generation to the next during reproduction. This passing of information, called *inheritance*, is a fundamental factor in the evolutionary process. Equally important is how genetic information is recombined and altered during reproduction. The mechanisms responsible for this are often referred to as *genetic operators*.

In normal human cells, each chromosome pair is homologous, meaning that both chromosomes in the pair have the same sequence of genes, but each gene pair may contain different alleles. In the production of sex cells (*gametes*), homologous pairs are potentially recombined. The genetic operation responsible for the recombination of genetic material is called *crossover*, which recombines a chromosome pair through the exchange of DNA (Figure 2.1 on the following page). During meiosis, each chromosome replicates itself but remains joined. A pair of chromosomes now consists of four chromatids arranged in matched pairs (Figure 2.1(a)). Crossover interchanges DNA from two (one from each pair) of the chromatids (Figures 2.1(b) and 2.1(c)). After the chromatids separate, there are two new recombinant and two non-recombinant chromosomes (Figure 2.1(d)). The locus of the crossover is subject to random variation, thereby allowing a very large number of potential recombinants.

Gametes, in contrast to normal cells, contain a single set of chromosomes. During sexual reproduction, male and female gametes fuse to produce a new cell. This new cell contains a set

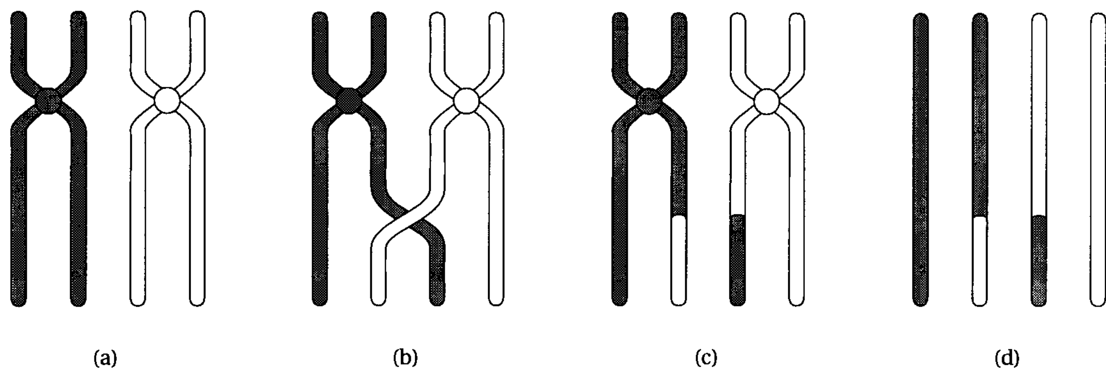


Figure 2.1: Crossover and recombination. (a) Two homologous chromosomes. (b) Crossover between the two central chromatids. (c) Chromosome pairs after crossover is complete. (d) The resulting two recombinant, and two non-recombinant, chromosomes.

of chromosomes comprised of genetic material from both parents, with each pair containing one chromosome from each sex of gamete. Therefore, the offspring's genotype is a combination of both parent's genotypes.

Another source of variability during reproduction is the genetic operation known as *mutation.* Mutation is the random alteration of a gene to a new allele. However, most often the result of mutation is a damaged or nonfunctional gene. While once thought to be the primary force in evolutionary progress, it is now generally considered to be much less significant than crossover.

2.1.3 Evolution

Evolution is the process by which a *population* adapts to its environment. Darwin [13] proposed that evolution is the result of the random variation of inheritable characteristics coupled with natural selection (survival of the fittest). The mechanisms responsible for the random variation are, as described in Section 2.1.2, the genetic operations during reproduction. Natural selection suggests that the more fit the individual, the greater its chance of survival and the more likely it is to reproduce, thereby passing its favorable traits onto the next generation. Over a period of generations, the population as a whole comes to contain more individuals whose chromosomes give them the genetic makeup to be *fitter* individuals. Thus, the composition of individuals in the population changes due to natural selection. However, it is essential that a population maintain

its variability, otherwise it will be unable to adapt to changes in the environment that affect its ability to survive and reproduce.

2.2 Genetic Algorithms

The *genetic algorithm* (GA) is an adaptive multi-dimensional search technique based on Darwinian evolution theory. Using an analogue of natural selection in conjunction with the principles of reproduction and recombination from natural genetics, genetic algorithms use the evolutionary process as a highly parallel search mechanism.

Holland introduced genetic algorithms in his pioneering book *Adaptation in Natural and Artificial Systems* [36]. While this remains the primary theoretical monograph, there are a growing number of treatments on the subject. Goldberg [26], gives a survey of the field within a textbook on genetic algorithms. In addition to expository examples, he also includes an extensive bibliography. A General survey of genetic algorithms and related methodologies can be found in Davis' collection of research papers [14]. More practical issues, including several applications of GA are addressed in his *Handbook of Genetic Algorithms* [15]. Michalewicz [54] gives an overview of genetic algorithms and their application to a variety of optimization problems, as well as their generalization to *evolution programs*. Concise but comprehensive treatments can be found in Srinivas and Patnaik [73] and Holland [37].

Since its introduction, many researchers have modified and extended Holland's original genetic algorithm. In this section we will first describe this basic genetic algorithm (on which Koza based his genetic programming paradigm [43]), then briefly discuss alternatives to binary encoding for certain types of problems.

2.2.1 The Basic Genetic Algorithm

The basic genetic algorithm (Algorithm 2.1 on the next page) uses a population of fixed-length binary strings in which each string is an encoded representation of a potential solution to the problem. New solutions are introduced using an analogue of natural evolutionary processes. The GA selects strings from the current generation to be reproduced into the next based on their *fitness*

(ability to solve the problem). Strings selected for the next generation may be recombined with other strings, as well as subjected to random mutation. This generational cycle is repeated until some termination criteria is met.

Algorithm 2.1 The basic genetic algorithm.

1: $t \leftarrow 0$ {generation number}
2: Initialize population $P(t)$
3: Evaluate population $P(t)$ for fitness
4: **while** Termination criteria not satisfied **do**
5: $t \leftarrow t + 1$
6: Select next generation $P(t)$ from $P(t-1)$ based on fitness
7: Recombine (crossover) selected pairs from $P(t)$
8: Mutate selected individuals from $P(t)$
9: Evaluate population $P(t)$ for fitness
10: **end while**

In using a genetic algorithm to solve a problem, the following components are required:

1. A method of encoding the problem variables or solution as fixed-length binary strings.

2. A *population* of fixed-length binary strings.

3. A *fitness* function.

4. A *selection* and *reproduction* mechanism.

5. Genetic operators.

6. Parameters and control variables for the algorithm.

Encoding

Genetic algorithms do not operate on problem solutions directly. Instead, they use an encoding of the problem variables. In the basic genetic algorithm, the variables are encoded as a fixed-length binary string (Figure 2.2 on the following page). The specific method of encoding depends on the domain of the variable. A Boolean value is simply represented by one position in the string, whereas a real number might be encoded as a bit vector of length l. The encoded versions of

each variable are concatenated to form the complete string of length L. This method of encoding mimics the natural structures described in Section 2.1.1. In particular, an encoded string is the single chromosome for the genotype defined by the variables. Each region (locus) of bits in the string corresponding to a variable is a gene, and a particular combination of bits represents an allele.

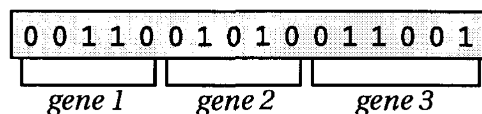


Figure 2.2: Binary string encoding in a genetic algorithm.

Population

Rather than continually refining a single solution, genetic algorithms operate on a *pool* of solutions simultaneously. This approach mimics the *gene pool* of natural systems. Through *survival-of-the-fittest*, the best solutions in a population are reproduced into the next generation. However, reproduction alone does not allow for a solution to evolve. It is the recombination of solutions from the population that generate fitter solutions, and it is the diversity of a large pool of potential recombinants that makes this possible. For the reader interested in an explanation of why this works in the binary string case, a theoretical basis is provided by Holland's *schema theory* [36].

The first step in a GA is to initialize the population for the first generation. In particular, for a population of size N, each string $X_i, i = 1, \ldots, N$, is initialized to a random bit vector of length L, where L is determined by the encoding mechanism, and is the same for each string. The population size, N, is one of the control parameters for the GA (for which a suitable value varies from problem to problem).

Fitness Function

Fundamental to genetic algorithms is the notion of a *fitness function*. The fitness function for a given problem provides a measure of how well an individual string solves the problem and is

used to determine which strings are chosen to form the next generation. For example, the fitness function for a stock selection problem might evaluate the rate of return, whereas for a navigation problem it might return the distance traveled to reach the target.

The fitness function in a GA is an abstraction of Darwinian *natural selection*. In nature, the ability for an individual to survive and mate (be a fitter individual) ensures the transference of that individual's genetic makeup into the gene pool of the next generation. Therefore, favorable traits are passed from generation to generation. Moreover, selection pressure will result in a population becoming more fit over time (assuming a static problem).

Selection and Reproduction

As in natural selection, the more fit the individual, the more likely it is to survive and reproduce. In genetic algorithms, it is the fitness function that is the measure of this likelihood. At each generational step in a GA, each string X_i is evaluated according to the fitness function. The next generation is chosen such that strings with higher fitness values are more likely to reproduce into the next generation. This scheme, *fitness proportionate selection*, can be implemented using a roulette wheel allocation method. Given a fitness function f, the fitness of a string X_i is $f(X_i)$. We define the normalized fitness, F_i, of X_i to be $f(X_i)/\sum_{j=1}^{N} f(X_j)$ for a population of size N. Given that $\sum_{i=1}^{N} F_i = 1$, we can allocate a proportion (a number of *slots*) of the roulette wheel equal to F_i for each X_i. Strings are then selected for reproduction into the next generation by randomly choosing a slot in the wheel, then copying the string associated with that slot into the new population. This process is repeated N times for a population of size N. For a sufficiently large population, this method reproduces strings into the next generation proportional to their fitness.

Selection and reproduction do not alone determine the next generation. More precisely, they determine the *mating pool*, which is then subjected to genetic operators to determine the next generation.

Genetic Operators

In a GA, genetic operators are used to recombine or modify strings from the mating pool to produce the population for the next generation. In the basic genetic algorithm, the primary operator

is crossover, however, mutation is often also used.

Crossover As in genetics, the fundamental evolutionary operator is that of crossover. In the basic genetic algorithm, a proportion of the strings are chosen from the mating pool to be recombined. The recombined strings then become part of the next generation. Recombination is accomplished using a single-point crossover method that simulates the exchange of genetic material between chromosomes in natural genetics (Section 2.1.2).

Pairs of strings, (A,B), are chosen from the mating pool at random. Since each string is of length L, there are $L-1$ interstitial points on each string. A crossover point, c, is chosen between 1 and $L-1$ using a uniform random distribution. Two offspring, (A',B'), are created such that $A' = A_{1,\ldots,c}B_{c+1,\ldots,L-1}$ and $B' = B_{1,\ldots,c}A_{c+1,\ldots,L-1}$ (Figure 2.3). The recombined offspring then become part of the population for the next generation.

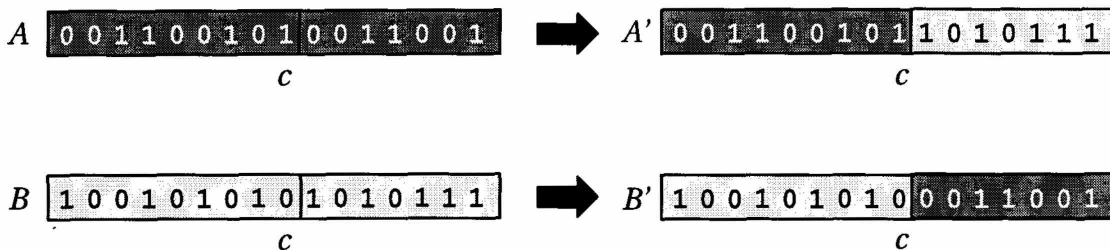


Figure 2.3: Crossover in a genetic algorithm.

Mutation The basic genetic algorithm also uses mutation as a secondary genetic operator. As is believed to be the case in nature, mutation is used not as the primary evolutionary force, but rather, to preserve a level of diversity in the population. For example, if a particular allele for a gene represented as a binary string is needed for the solution, and that allele is not present in the initial population, then that solution cannot be reached through recombination alone. This situation can lead to a premature convergence of the algorithm. In this case, mutation can be effectively used to reintroduce lost genetic material into the population. Moreover, it can be used to introduce a new allele that was not present in the initial population.

is crossover, however, mutation is often also used.

Crossover As in genetics, the fundamental evolutionary operator is that of crossover. In the basic genetic algorithm, a proportion of the strings are chosen from the mating pool to be recombined. The recombined strings then become part of the next generation. Recombination is accomplished using a single-point crossover method that simulates the exchange of genetic material between chromosomes in natural genetics (Section 2.1.2).

Pairs of strings, (A, B), are chosen from the mating pool at random. Since each string is of length L, there are $L-1$ interstitial points on each string. A crossover point, c, is chosen between 1 and $L-1$ using a uniform random distribution. Two offspring, (A', B'), are created such that $A' = A_{1,\ldots,c}B_{c+1,\ldots,L-1}$ and $B' = B_{1,\ldots,c}A_{c+1,\ldots,L-1}$ (Figure 2.3). The recombined offspring then become part of the population for the next generation.

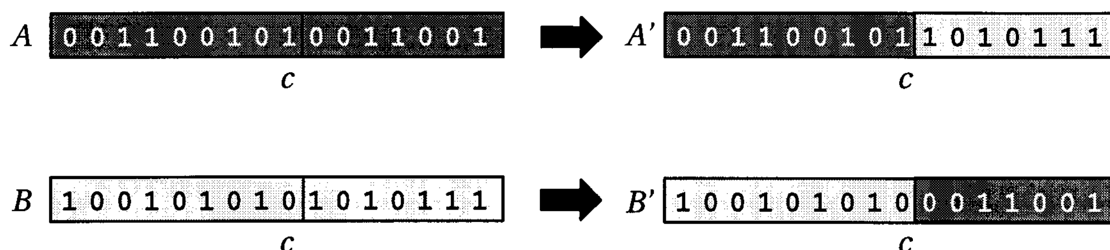


Figure 2.3: Crossover in a genetic algorithm.

Mutation The basic genetic algorithm also uses mutation as a secondary genetic operator. As is believed to be the case in nature, mutation is used not as the primary evolutionary force, but rather, to preserve a level of diversity in the population. For example, if a particular allele for a gene represented as a binary string is needed for the solution, and that allele is not present in the initial population, then that solution cannot be reached through recombination alone. This situation can lead to a premature convergence of the algorithm. In this case, mutation can be effectively used to reintroduce lost genetic material into the population. Moreover, it can be used to introduce a new allele that was not present in the initial population.

In the basic genetic algorithm, a binary string A is mutated (Figure 2.4) by *flipping* (0 becomes 1 and 1 becomes 0) the bit at a location A_m in the string, where m is between 1 and $L-1$. Mutation in the basic genetic algorithm is implemented in one of two ways: either a single position is chosen at random to be mutated, or each position within a string is independently considered for mutation based on a random value. The frequency of mutation is governed by a control parameter, p_m.

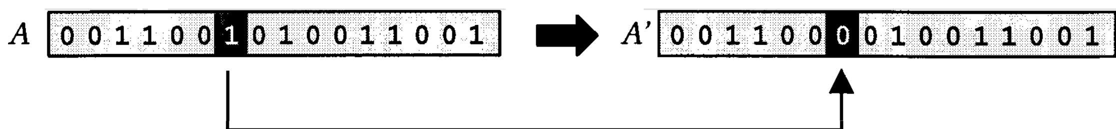


Figure 2.4: Mutation in a genetic algorithm.

Control Parameters

The basic genetic algorithm makes use of several control parameters:

Population size The population size N determines the number of strings in each generation. The larger the size, the more potential diversity in the gene pool.

Termination criteria There must be well-defined criteria for terminating the algorithm. Typically the algorithm is terminated if one of the following becomes true when the population is evaluated: one or more of the solutions has a fitness value exceeding a particular threshold, or a maximum number of generations has been reached.

Probability of crossover (p_c) After the mating pool has been selected according to fitness, p_c is the probability that a randomly chosen pair of strings will be subject to crossover.

Probability of reproduction (p_r) After the mating pool has been selected according to fitness, p_r is the probability that a randomly chosen string will be reproduced directly into the next generation.

Probability of mutation (p_m) The probability that an individual string will be subject to mutation is given by p_m. Alternatively, p_m is the mutation rate — the probability that a single bit is mutated.

2.2.2 Real Number Encodings

In the standard genetic algorithm, the chromosome is a fixed-length binary string. For a problem that is defined using a number of real-valued variables, an encoding scheme is used. An alternative approach is to define a chromosome as a list of real numbers.

Michalewicz [54], in his generalization of genetic algorithms to evolution programs, suggests that the chromosome reflect the native data types of the problem. Therefore, a real-valued problem should use a real-valued chromosome. Davis [15] also promotes the hybridization of GA in this respect. In addition, Goldberg [26] notes that despite strong theory to support binary encoding, an increasing number of applications are using real-valued chromosomes.

Genetic Operators

The predominant operator in genetic algorithms is crossover. Typically, a single-point crossover is used, and this operator can also be used with real number encodings. However, whereas a binary encoded GA uses domain independent operators, a real number encoding can also utilize domain specific operators [15, 54]. For example, a *uniform* or *average* crossover mechanism can be used. In uniform crossover, one or more genes are swapped during recombination. This operator can also be used with binary encodings but tends to be too brittle. Average crossover is only applicable to real numbers. It produces a single offspring where each gene is the average of the two parental alleles.

Mutation operators tend to be more widely used in real-valued models. A variant of binary point mutation can be used which replaces a gene with a new allele from a specified range. Another common mutation technique adds some amount of Gaussian distributed noise to the value of a gene. The variance of the noise can be scaled according to fitness or time-step so that as the population converges on a solution, the mutation is more focussed.

2.3 Genetic Programming

Genetic algorithms have been used successfully in many problem domains. However, if the size and form of the solution procedure is unknown, they can be difficult to apply. Koza's *genetic programming* paradigm [43] proposes to overcome this problem through a change in representation. In genetic algorithms, the genotype (binary string) represents the parameter space of a known procedure, whereas in genetic programming, it represents the entire procedure. In particular, the genotype in genetic programming is a complete hierarchical computer program.

In the following sections we will describe the genetic programming paradigm (GP) proposed by Koza. In particular, we will examine how computer programs can be structured to fit into the paradigm, and how the standard genetic operators can be applied to such programs.

2.3.1 The Standard Genetic Programming Paradigm

As has been the case with genetic algorithms, the genetic programming paradigm has been continually modified and extended. However, the essential flavor has remained unchanged. In this section we will describe the prototypical version described in Koza [43].

Koza's GP paradigm (Algorithm 2.2 on the next page) follows the same basic steps as the basic genetic algorithm (Algorithm 2.1) described in Section 2.2.1. However, the structures that make up the population are radically different. In the basic GA, the genotype for a particular solution is a fixed-length binary string, where the binary string represents one point in the multi-dimensional solution space of a known procedure. In genetic programming, a genotype is a parse tree representing the structure of a complete computer program. Therefore, instead of operating on a population of fixed-length binary strings, genetic programming evolves a population of computer programs of varying shape and size.

Algorithm 2.2 The standard genetic programming paradigm.

1: $t \leftarrow 0$ {generation number}
2: Initialize population $P(t)$
3: Evaluate population $P(t)$ for fitness
4: **while** Termination criteria not satisfied **do**
5: $t \leftarrow t+1$
6: **if** Reproduce **then** {with probability p_r}
7: Select individual programs from $P(t-1)$ based on fitness
8: Copy selected programs into $P(t)$
9: **else if** Recombine **then** {with probability p_c}
10: Select program pairs from $P(t-1)$ based on fitness
11: Perform crossover on selected pairs
12: Copy offspring pair into $P(t)$
13: **end if**
14: **if** Mutate **then** {with probability p_m}
15: Mutate selected individuals from $P(t)$
16: **end if**
17: Evaluate population $P(t)$ for fitness
18: **end while**

2.3.2 Program Structure

An important consideration in genetic programming is the representation of the programs in the population. In particular, the choice of representation must allow for the desired genetic operators to be applied. A convenient representation is that of rooted, point-labeled, ordered trees. This model of program structure is easily constructed and allows for the representation of many common programming constructs. A simple example of such a tree would be a function returning the value of $1-e^{0.01d}$ (Figure 2.5 on the following page).

In Koza's implementation of GP, trees are represented as LISP S-expressions [74]. For example, the tree in Figure 2.5 would be represented by the S-expression:

```
(- 1 (exp (* 0.01 d))).
```

It is important to note that GP does not require the use of LISP or any of its special features. However, unlike many programming languages, there is a simple and direct correspondence between LISP S-expressions and their parse trees. Therefore, we will continue to use S-expressions in this thesis for their syntactic convenience.

For a particular GP problem, a set of program constructs must be defined. In GP program

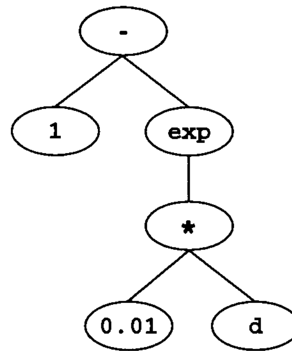


Figure 2.5: Program tree for $1 - e^{0.01d}$.

trees, leaf nodes belong to a set of *terminals*, $\mathcal{T} = \{a_1, a_2, \ldots, a_{\text{Nterminals}}\}$, and non-leaf nodes to a set of *functions*, $\mathcal{F} = \{f_1, f_2, \ldots, f_{\text{Nfunctions}}\}$. Each function, f_i, takes a specific number of arguments $\Phi(f_i)$, from the set $\mathcal{F} \cup \mathcal{T}$.

In selecting the sets of functions and terminals, the conditions of *closure* and *sufficiency* must be satisfied. *Closure* requires that, for every function $f_i \in \mathcal{F}$, that for each of its arguments, be able to accept any value and data type returnable by any function $f_j \in \mathcal{F}$, and any value and data type of any terminal $a_k \in \mathcal{T}$. This guarantees that any program generated randomly, or that is the result of the application of a genetic operator, is a valid program. *Sufficiency* is satisfied if there is a program that can be constructed using only the set $\mathcal{F} \cup \mathcal{T}$ that provides a satisfactory solution to the problem.

To satisfy closure, it may be necessary to provide functions that return default values rather than fail under certain conditions. For example, a *protected divide* operator is often used. It is the same as a regular division operator, except that if the divisor is zero, a default value (such as 0 or 1) is returned instead of the operation being undefined. Functions representing control structures may also need to be recast. For example, the typical conditional operator may be reformulated to make the condition implicit to avoid having multiple data types for its arguments. A common operator of this type is the *if-less-than-zero* operator, which avoids the need for a Boolean argument to represent an arbitrary conditional. In addition to providing tailored functions, syntactic restrictions may be enforced during program construction and during the application of genetic operators to ensure closure. Most often, these are restrictions placed on the allowable data types

for a function's arguments.

Program Generation

The generation of the initial population in genetic programming is considerably more complex than that of genetic algorithms. Instead of generating random binary strings, complete syntactically correct parse trees must be built. In GP, each tree in the initial population is generated separately with a single function as the root, and a tree depth no greater than some pre-defined maximum. Koza [43] defines two variations (*grow* and *full*) of a recursive generation program (Algorithm 2.3). The *grow* method produces trees of random size and shape, while the *full* method produces trees of uniform size and shape.

Algorithm 2.3 Generating a random program.

1: Randomly select a function $root \in \mathcal{F}$
2: **if** $depth = depth_{\max} - 1$ **then**
3: Randomly select a terminal $\in \mathcal{T}$
4: **else if** *root* is a function **then**
5: **for all** $arg_j, j = 1, \ldots, \Phi(root)$ **do** $\{\Phi(f) = \text{the arity of } f\}$
6: **if** use grow method **then**
7: Randomly select $arg_j \in \mathcal{F} \cup \mathcal{T}$
8: **else if** use full method **then**
9: Randomly select $arg_j \in \mathcal{F}$
10: **end if**
11: Recursively generate subtree with arg_j as root (starting at step 2)
12: **end for**
13: **else** {*root* is a terminal}
14: {This branch is complete}
15: **end if**

To help ensure sufficient diversity in choosing an initial population, Koza [43] uses a mixture of *grow* and *full* generation in what he calls a *ramped half and half method*. In this method, half of the population is generated using the grow method and the other half using the full method. Within each subpopulation, the maximum tree depth is *ramped*. In particular, for a population of size N, each subpopulation will be of size $N/2$. With a maximum tree depth of 5, $N/10$ trees will be generated at each depth $1, \ldots, 5$ for each method.

2.3.3 Genetic Operators

The common set of genetic operators used in genetic programming are analogous to those used in a genetic algorithm. However, the more complex structures of GP lend themselves to a wider variety of operators. The standard GP paradigm described in Koza [43] categorizes operators as either primary or secondary. He suggests that the primary operators are sufficient, but that the secondary operators may provide some benefit in certain cases. This categorization, and the reasoning behind it, follows from both Holland's [36] genetic algorithm, and from natural genetics. In particular, that reproduction and recombination through crossover are the fundamental forces behind evolutionary change.

Primary Genetic Operators

As in the basic genetic algorithm, the primary operators in the standard genetic programming paradigm are reproduction and crossover. Many applications forego mutation and use only these two operators.

Reproduction Individual programs are reproduced into the population for the next generation based on their fitness. The number of programs chosen to be reproduced is given by the probability of reproduction (p_r). In the standard GP paradigm, a fitness-proportionate selection scheme is employed using individual programs' normalized fitness. Reselection is allowed; therefore, a program may be reproduced more than once. Programs chosen for reproduction are simply copied into the new population. Reproduction is asexual, since an offspring is the result of only one parent.

Crossover Programs are recombined producing offspring for the next generation using crossover. Two parental programs are chosen using the same selection scheme based on fitness as in asexual reproduction. The number of programs chosen for recombination is given by the probability of crossover (p_c).

In the standard genetic programming paradigm, a subtree crossover mechanism is used. First, a point is chosen randomly in each of the parent trees. Second, the subtrees rooted at the chosen

crossover points in each of the parental trees are detached. Finally, the first offspring is produced by attaching the subtree fragment from the second parent at the crossover point in the first parent. The second offspring is produced in a symmetric manner. In each case, a pair of parents produces a pair of offspring. This process is illustrated in Figure 2.6.

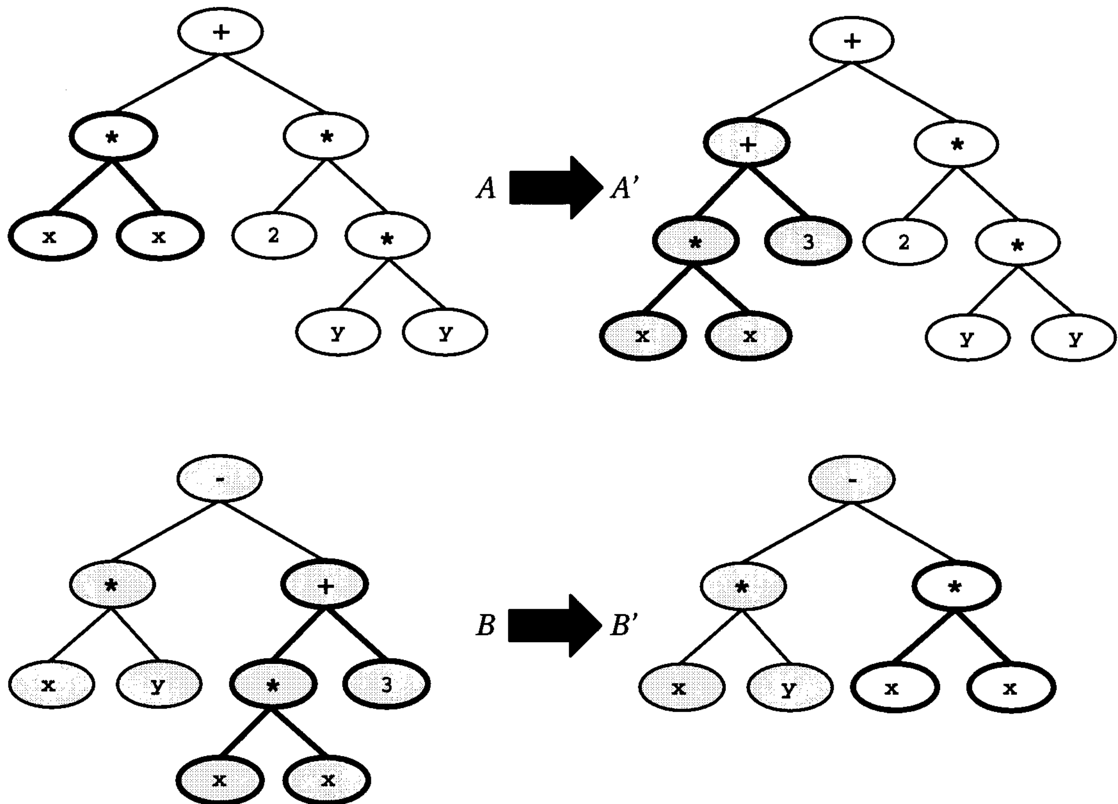


Figure 2.6: Crossover in genetic programming.

There are two important differences between the GP crossover operator and the single-point crossover used in a GA. First, the offspring in GP are most likely of a different size and shape than their parents (depending on the crossover points), in a GA they remain fixed-length binary strings. Second, if reselection is allowed, a program of high fitness may be chosen for both parents. This results in an *incestuous* crossover. In a GA, incestuous crossover degenerates to an instance of asexual reproduction. In GP, the two offspring will differ unless the same crossover point is chosen for both instances of the parent.

Secondary Genetic Operators

There are a number of secondary genetic operators used in GP. Koza [43], in the book *Genetic Programming*, describes five: mutation, permutation, editing, encapsulation, and decimation. In this section we will describe the two most commonly used in the standard GP paradigm. The others provide extensions (encapsulation), or are rarely employed (permutation and decimation).

Mutation Koza's *mutation* operator replaces the subtree rooted at a randomly selected point with a new randomly generated subtree (Figure 2.7). A probability based on tree depth can be used to prevent excessive terminal swapping or to encourage larger or smaller trees as a result. This operator is most often used to introduce lost diversity into a prematurely converging population.

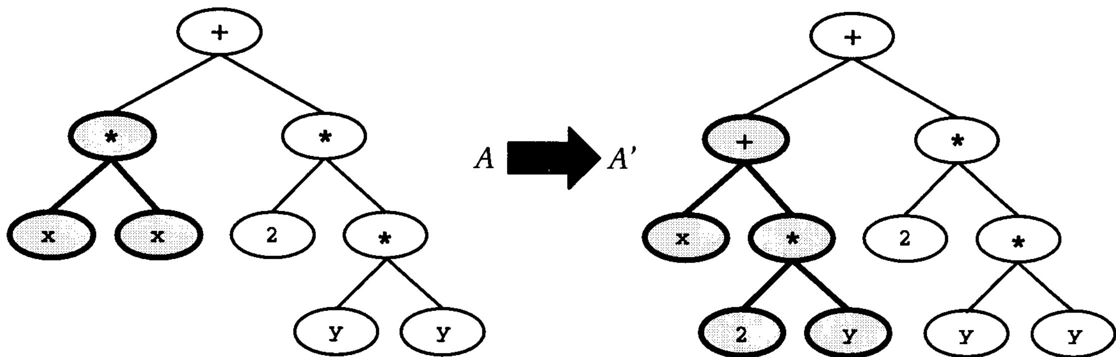


Figure 2.7: Mutation in genetic programming.

Editing The *editing* operation replaces a constant valued subtree with its value. For example, the S-expression `(* 1 (+ 1 1))` can be replaced with the terminal `2`. Editing is used as an asexual operator that for a given program tree, recursively invokes a set of editing rules at each non-terminal node in the tree. This operator can be used after a solution is found to *clean-up* the result, or during the evolution process to encourage parsimony. The latter must be used with caution, however, as it may have an adverse effect on the diversity of the population.

2.4 Interactive Evolution

Genetic algorithms and genetic programming are both based on the evolutionary principle of natural selection. In these methods, survival depends on the fitness of an individual as determined by the fitness function. The definition of an appropriate fitness function is therefore critical to their application. The fitness function is generally an algorithm which given a potential solution, computes an objective measure of how satisfactorily it solves the problem. However, if the problem at hand requires a subjective judgement, it may be impossible to define a useful fitness function. For example, the definition of a function able to universally quantify the melodiousness of a computer-generated musical score is highly unlikely.

Interactive evolution systems [4] replace the objective fitness function (in either genetic algorithms or genetic programming) with subjective human judgement. At each generational step in the algorithm, a set of phenotypes representing the population is presented. Selection of the mating pool for the next generation is controlled by the user. In most applications, the user performs the selection directly. In others, the users assigns each phenotype a relative fitness value which is then used in a fitness-proportionate selection scheme.

Interactive evolution systems typically operate on extremely small populations. Limitations associated with human comprehension restrict the size of a population that can be meaningfully examined to be of magnitude 10 (in practice, generally less than 25). This has the unfortunate drawback of drastically reducing the diversity available in the gene pool. Because of this, interactive evolution systems frequently use very high mutation rates in order to maintain sufficient variability.

Chapter 3

Evolving Procedural Textures

In *Artificial Evolution for Computer Graphics* [69], Sims describes how interactive evolution techniques can be used to produce complex procedural textures quickly and easily. This approach addresses the difficulty in specifying texture synthesis procedures by isolating the designer from the specification — design proceeds only through the subjective judgement of results. However, this approach is not without drawbacks. First, by hiding the specification completely, a degree of control is lost. In particular, there is no facility for a designer to *fine-tune* a result. Second, the difference between parent(s) and offspring tends to vary widely. This can be advantageous when exploring large spaces, but makes it difficult to focus design changes once a close approximation is found.

In this chapter, we will first review related work in using interactive evolution for design-oriented tasks. Then, a detailed description of Sims' system applied to the creation of procedural textures will be given. Finally, we will propose an extended system to address the shortcomings just described.

3.1 Applications of Interactive Evolution

Interactive evolution systems differ from typical GA or GP applications by requiring explicit human intervention at each generational step. In particular, they replace the fitness function used in

selection with subjective human judgement. Currently, published work regarding interactive evolution (IE) systems comprises a very small portion of the body of literature concerning evolution-based computation as a whole. However, in situations where explicit fitness functions are not available, they have been at the forefront. In this section we will briefly review the major applications of interactive evolution.

Dawkins created the first major IE application with *The Blind Watchmaker* [16]. This influential program allowed Dawkins to experiment with artificial evolution using *biomorphs.* Initially, Dawkins' biomorphs were two-dimensional, symmetrical tree structures whose form was determined by a genotype consisting of nine integer valued genes. Later [17], he developed a larger genotype that allowed for more complex 2D creatures to be described. Dawkins' biomorphs evolve by starting with a single random genotype. From a single parent, a small population of offspring are generated by incrementing or decrementing one gene per offspring. At each generational step, the user selects one of the biomorphs from a visual display to act as the parent for the next generation.

Oppenheimer [58] describes a simple interactive evolution approach to create stochastic, self-similar, 3D models of trees. Like Dawkins, he uses a short sequence of genes (in this case real numbers) to form the genotype, and effects evolutionary change through mutation.

Smith [72] describes a system for evolving Fourier series based biomorphs using genetic algorithms. Instead of the *select-and-mutate* approach used by Dawkins and Oppenheimer, Smith's system (using a real valued chromosome), employs recombination using single-point crossover in addition to mutation. Although this system is capable of producing interesting *creatures,* it is worthwhile to note that Smith's intention was to provide a working example of the applicability of IE in design problems lacking explicit fitness criteria.

Sims has employed interactive evolution to design a variety of computer graphics elements including: 3D plant structures [69], 2D images and textures [69, 71], solid textures [69, 71], image processing operators [69, 71], parametric surfaces [71], and dynamical systems [70, 71]. In each of these applications, the user is presented with a display of some small number of phenotypes from which they choose one or two individuals whose genotype(s) will act as the parent(s) for the next generation. In his earliest system, Sims used a real-valued GA to evolve 3D plant structures. While

the basis of this system is similar to Oppenheimer's [58], it employs a much more complex model of reproduction using several domain-specific recombination operators in addition to mutation. In his later efforts, Sims uses a GP approach — evolving symbolic expressions to generate images, textures and surfaces. In [70, 71], he describes how sets of expressions can be co-evolved to develop dynamical systems. Sims' system for evolving images and textures is discussed in detail in Section 3.2.

Latham [29, 47], in designing sculptures, developed an interactive evolution technique employing both recombination and mutation using hand-drawn evolutionary trees. In collaboration with Todd [76, 77], this idea was integrated with a procedural solid modeling system to create *Mutator. Mutator* incorporates interactive evolution into a GA to evolve virtual sculptures (3D computer models). A population of nine sculptures, each the expression of a real-valued genotype, is judged by the artist. One or two of the sculptures can be selected as the parent(s) for the next generation. For sexual reproduction, either a single-point or uniform crossover is used. For asexual reproduction, chromosomes are mutated using a method which attempts to *steer* the mutation in the direction indicated by past choices. This is accomplished by linearly extrapolating the gene vector prior to mutation in the direction derived from the difference between the current selection and its ancestors . In addition to solid models, *Mutator* can also be used to evolve simple solid textures based on a Fourier model.

Caldwell and Johnston [10] describe an application that uses an interactive genetic algorithm to produce composite drawings of faces. Unlike most IE applications, selection is not explicit. Rather, each member of the population is ranked on a scale, then a standard fitness-proportionate selection scheme is used.

Baker and Seltzer [2] also describe a system for evolving faces. Their system uses a hybrid genetic algorithm in which a chromosome encodes a list of *strokes*. From a panel of choices, one or two individuals can be chosen as the parent(s) for the next generation. The hybrid representation used requires that the genetic operators be domain specific. Recombination is accomplished using a combination of uniform and average crossover that incorporates knowledge of the underlying structure. The mutation operator either randomly adds, deletes, or translates stroke points or entire strokes.

Nguyen and Huang [55] use a hybrid GP/GA approach to evolve 3D models of aircraft. Crossover and mutation can either operate structurely (GP) on the grammar tree that defines an aircraft, or can operate on the parameters (GA) using conventional binary-string encoded methods. In both cases, a number of constraints are incorporated into the genetic operators to ensure workable results.

Graf and Banzhaf [28] describe a system which uses interactive evolution to evolve general bitmap images based on their content. In this method, rather than synthesizing images, they evolve them from a base set, using warping [82] and morphing [5] operators to mutate and recombine parent images.

Baluja, Pomerleau, and Jochem [3] use IE to produce a training set for an artificial neural network. Using a subset of the capabilities for generating images described in Sims [69], the user assigns a fitness rank to each image at each step. Using these images as input, a neural network is trained to match the fitness ranking. This approach, if successful, would enable the creation of fitness functions even in situations with imprecise notions of fitness. However, in this preliminary work, the results do not generalize well due to the abstractness of the criteria.

A concise overview of interactive evolution (including several of the systems discussed above) can be found in Banzhaf [4].

3.1.1 Summary

All of the applications described above use a common interface to evolve solutions. This interface, originating with Dawkins' *Blind Watchmaker* [16], has become the hallmark of interactive evolution systems. In each application, a small population of phenotypes is displayed (typically arranged as a two-dimensional grid). At each generational step, the user of the application interactively assigns a fitness value to each of the phenotypes. Fitness may be a relative or normalized ranking to be used in a standard fitness-proportionate selection scheme, or it may be binary, allowing the user to select the mating pool explicitly. Using the results of this interactive selection process, standard genetic operators are then used to create the next generation. This style of interface places an important restriction on the population size being evolved. Practical display constraints, the limitations of human visual comprehension, and the maintenance of interactivity

require the use of very small populations. This has the side-effect of drastically reducing genetic diversity, hence, the predominant use of mutation in IE applications.

All of the systems discussed here rely solely on the evolutionary process to generate results. Several however, do offer the user some choices in the application of this process. For example, in *Mutator*, Todd and Latham [77] allow the artist to control the mutation rate, as well as choose from several methods of mutation and recombination. They also allow mating to occur with a parent from outside the current population to reduce the effects of inbreeding. In a similar feature, Baluja et al. [3] give access to an extensible genome library, allowing genetic material to be introduced at the user's discretion. An example of more direct manipulation can be found in Caldwell and Johnston [10], who allow a gene to be *locked* to guarantee its preservation from generation to generation.

The interactive evolution approach has demonstrable usefulness particularly in complex search problems lacking explicit criteria (construction of composite faces), and exploring large parameter spaces (biomorphs, dynamical systems). In this thesis, however, we are interested in its usefulness as an artistic design tool. The work of Sims [69, 71], and that of Todd and Latham [76, 77], has without question produced both novel and impressive results. However, the introduction of additional modes of control may lead to a more powerful system for design.

3.2 Interactive Evolution of Textures

Sims [69, 71] describes a system for the synthesis of images and textures using interactive evolution. Beginning with a random set of genotypes, a corresponding set of phenotypes is displayed on a grid. The user interactively selects one or more of these phenotypes based on their aesthetic appeal. The genotypes for the selected set are then used as parents for the next generation. This process is repeated as many times as the user desires. In producing each new generation, a number of mutation and mating operators may be applied.

In this system, the genetic programming paradigm is employed. Genotypes are modeled using symbolic expressions, and the complete system is implemented in LISP [74] on a data parallel Connection Machine [35].

In this section, we will describe the structures and genetic operators used by this system to evolve 2D images and textures. We will also discuss the minor modifications necessary to allow the evolution of solid textures. We will conclude with a discussion of the system's practical limitations.

3.2.1 Evolving Images and Textures

Two-dimensional images and textures are represented by symbolic expressions that return a color for each pixel coordinate (x, y).[1] As in all GP applications, there must be well-defined sets of functions and terminals that together satisfy the property of closure (Section 2.3.2).

Function Set

Each function in this system returns as its result either a color or grayscale image. Function arguments may be either scalar or vector (color) values, or entire color or grayscale images. Depending on the function, different data types for a particular argument may be either handled differently, or coerced into the correct type. In addition, some functions may also restrict certain arguments to a subset of the four types.

The function set consists of standard mathematical functions modified to operate on entire images, procedural noise generators that compute regular and warped versions of both scalar and color noise, a variety of general image processing operators that work on single or multiple images, and fractal pattern generators.

Terminal Set

The terminal set contains the base components of the allowable data types as well as the program variables. Images are constructed as arrays of the base data types. The base data types are scalar and vector constants. A vector constant has three components, and is generally considered a color. However, it may also be used as an arbitrary vector by some functions. The symbolic variables x and y represent the pixel coordinates for the image being generated and serve as program variables.

[1] In this case, no distinction is made between images and textures.

Genetic Operators

Both sexual and asexual reproduction mechanisms are provided for in Sims' implementation. For asexual reproduction, several mutation operators are used. A program expression is mutated by traversing the expression tree. During traversal, each node is subject to mutation based on a probability measure (frequency of mutation). The probability of each type of mutation is further dependent on the type of node, and the overall frequency of mutation is scaled so that it decreases as tree size increases. The mutation operators used are as follows:

1. The subtree rooted at the node is replaced with a new randomly generated subtree.
2. A constant scalar or vector is offset by some random amount.
3. A function is replaced with a new randomly generated function.
4. The subtree rooted at the node becomes an argument to a new randomly generated subtree which takes its place.
5. The subtree rooted at the node is replaced with the subtree of one of its arguments.
6. The subtree rooted at the node is replaced with a randomly chosen subtree from the same symbolic expression. Note that this operator is subsumed by subtree crossover if incestuous mating is allowed.

For sexual reproduction, two different recombination operators are used:

1. A subtree crossover producing a single offspring can occur. One parent is chosen for reproduction, and within that parent, a subtree is selected randomly. The chosen subtree is replaced with a randomly selected subtree from the second parent to produce the offspring.
2. A single offspring can be produced by incorporating material from both parents' genotypes. This is accomplished by recursively traversing both parents' expression trees in parallel. When the parents' structures are the same, they are reproduced in the offspring. When two subtrees differ, one is chosen (with equal probability) to be copied to the offspring.

3.2.2 Evolving Solid Textures

The system described for evolving 2D images and textures can be easily modified to evolve solid (3D) textures by making the following changes:

1. Instead of displaying images, the phenotype display must be modified to display samples of the solid texture applied to 3D geometry. This may be accomplished simply by using default objects such as spheres or planes.

2. The terminal set must be extended to include a z coordinate in addition to x and y.

3. The function set must be modified to handle the new domain. The 2D noise functions must be replaced with 3D ones, and any functions requiring neighboring coordinate information (those using convolution for example) must be reformulated to work in 3D. Furthermore, if the texture is to be evaluated as a point process, this must be taken into consideration when constructing such operators that may use a region in computing a point value.

3.2.3 Limitations

Sims' system for evolving textures is capable of producing unique and complex results. As the examples show in [69], fairly short symbolic expressions can produce results that would require considerable time and effort to craft by hand. However, as with any system, there is room for improvement. In this section we will discuss several limitations in Sims' implementation. In particular, we will address the difficulties in using such an implementation as a *design*, rather than an *exploration* tool. In addition, we will examine some efficiency concerns in applying the results.

Design

Genetic algorithms, and their extension to genetic programming, are both highly efficient techniques for searching large parameter spaces. Due to the parallel way in which they explore the search space, they are particularly effective in generating approximate results quickly. Sims' system exploits this characteristic in generating new and interesting textures. However, an important part of effective design is the ability to *refine* an approximate solution.

The genetic programming paradigm, given sufficient genetic diversity in the population, is capable of evolving any solution covered by the sufficiency of its function and terminal sets. However, given the extremely large program space defined by the set of possible texture generation expressions, the precise result may take many generations to evolve. This situation is exacerbated by the necessarily small population in an interactive evolution system.

Using Sims' system, once an approximate solution is found, it is possible to control evolution to a certain degree by adjusting the probabilities for the different mutation operators. Decreasing the frequency of operators that affect structural change relative to those that adjust constants will result in a more *incremental* approach similar to that of Dawkins [16]. However, this applies only to asexual reproduction. Consider the following design scenario using Sims' system. After a number of generations of exploration, we find an interesting texture. Wanting to examine a set of similar variants, we adjust the mutation rates to favor incremental adjustment, and continue selecting one phenotype at each generation. After a few more generations, we discover two textures that seem to define the boundaries of what we are looking for. Therefore, what we need is something that is intuitively *between* the two. However, we are now faced with two unsatisfactory alternatives. We can either continue to mutate one or both of the textures, or we can recombine them structurally. The first alternative ignores the power of recombination. The second is likely to recombine blindly — offering no control over what is meant by a combination from the viewpoint of the designer. This scenario suggests the need for a method to delimit the search space when appropriate without sacrificing the ability to effectively recombine potential solutions.

In the example just given, we considered the case where further refinement was desired, but it may be difficult (or undesirable) to explicitly identify the characteristics to be changed. However, in many cases, the difference between an approximate and final design can be attributed to a single salient feature. For example, a texture may be visually appealing to the designer with the exception of one parameter, such as a color. From a design perspective, the most expedient way to achieve the desired change would be to allow the designer to make the adjustment directly. However, in a strictly evolution-based system such as Sims', we are relegated to waiting for this change to evolve. It can be argued that *final* adjustments can be considered a post-process. For example, by editing the parameters of the resulting texture procedure. However, this type of adjustment

may also be useful for intermediate results that will be subject to further evolution. Therefore, some facility for adjustment would be useful during the evolution process.

Efficiency

In using a system such as Sims' for designing textures, it is expected that the results are to be applied to geometric models as texture maps. For both 2D and 3D (solid) textures, the symbolic expression can be translated into an appropriate shader. Furthermore, as many rendering systems offer interactive interfaces to adjust shaders, the constant valued terminals in the texture expression could be made available for adjustment using standard graphical user interface controls. This application of the results of a system such as Sims' raises two concerns over efficiency. The first concern relates to the computational efficiency of the resulting symbolic expressions. The second, to the practical efficiency of editing the result.

Symbolic expressions evolved using the genetic programming paradigm often contain subtrees that do not affect the final result. In natural genetics, non-coding regions of DNA are called *introns*. This terminology has been adopted in the GP literature to describe non-effective subtrees. It is important to note that in GP, a subtree classified as an intron may become effective in subsequent generations through mutation and recombination. Recently, there has been some debate as to role of introns in GP [1, 56, 57]. However, we are only concerned here with the *results* after the evolutionary process has completed. Therefore, any non-effective subtrees will remain non-effective.

The presence of introns in a resulting texture procedure will unnecessarily increase the computational resources required to evaluate that texture. This could be significant if the number of texture evaluations is large. Moreover, the nature of many of the functions used by Sims can result in a relatively high percentage of non-effective subtrees.

In addition to containing introns, resulting expressions may be unparsimonious. For example, the computed result of a large subtree may always be a constant color. This type of situation results in a *practical* inefficiency. If a texture procedure such as this is converted to an editable shader, the subtree in question may result in many adjustable parameters, all of which can be subsumed by a single vector parameter.

3.3 A Genetic Engineering Approach

In this thesis we develop a *genetic engineering* model for procedural texture design. This approach extends a typical interactive evolution system to allow both constrained and explicit alteration of genetic code. By incorporating multiple levels of control, it is hoped that a more design oriented tool will result. In Section 3.2, we described an interactive evolution system for generating textures. Using the functionality of this system as a basis, we will develop several extensions to address the practical limitations identified in Section 3.2.3.

3.3.1 Prototype Implementation

A prototype implementation of the genetic engineering model for texture design will be the focus of this thesis. The basis for the prototype will be Sims' system for image and texture design described in Section 3.2. In particular, the prototype will use the genetic programming paradigm to create 2D images and textures using a function and terminal set similar to that described in Sims [69].

The implementation of a system that replicates the essential functionality of Sims' will form the base layer of our application. We will refer to this base as the *GP layer.* We will augment this base with two other levels of control. The first new level, the *GA layer,* will use a hybrid genetic algorithm to evolve textures without altering their programmatic structure. The second new level, the *editing layer,* will allow the user to directly manipulate a program tree using a graphical user interface. In describing these three modes of control below, we indicate how the genetic engineering model proposes to address the limitations discussed in Section 3.2.3.

1. **GP Layer**

   The GP layer will provide equivalent functionality to Sims' system for evolving textures. Using the genetic programming paradigm, interactive evolution will be employed with a variety of mutation and recombination operators.

2. **GA Layer**

   The GA layer will use the same interactive evolution interface as the GP layer. However,

it will employ a hybrid genetic algorithm to evolve the population. By using a real-valued chromosome whose coding reflects the parameters of the GP genotype, genetic operators can be used that do not alter the structure of the program. This has the effect of restricting the search space. By using domain-specific genetic operators with a delimited search space, the user should be able to perform design refinements in a more controlled manner. This addresses our first design limitation.

3. **Editing Layer**

   The editing layer will enable a user to manipulate a texture program directly using a graphical representation of the genotype. Allowing individual parameters to be manipulated directly addresses a second design limitation — the inability to make specific adjustments when desired. In addition to providing controls for manipulating terminals, a number of structural editing operations may be employed. As well as providing direct control over salient features, structural modification may be used to eliminate introns or collapse unparsimonious subtrees. This provides a means to address our efficiency concerns.

Using these three levels of control, a hierarchical approach to the design of a texture is possible. Initial exploration is done using the GP layer. Once a close approximation to a satisfactory result is found, it is further refined using the GA layer. Final adjustments are then made using the editing layer. However, there is nothing enforcing a strictly hierarchical approach. All layers operate on the same program tree representation of genotypes. Therefore, a designer is free to switch between levels of control allowing an iterative approach.

Chapter 4

A Texture Design Tool

In this chapter we describe the implementation of a prototype *texture design tool* — a system for creating texture synthesis programs. The foundation of our program is modeled after an interactive evolution system developed by Sims [69]. We build on this model using a *genetic engineering* approach to create a more design-oriented system (Section 3.3). In Sims' system (Section 3.2), the genetic programming paradigm is used to evolve texture synthesis programs. We augment this model with two additional levels of control: a hybrid genetic algorithm to evolve program parameters, and a graphical user interface for direct manipulation of program structures.

4.1 Program Structure

Our system for genetically engineering texture synthesis programs is based on the genetic programming paradigm. As in a standard GP system, each program can be considered a single, ordered, point-labeled tree. In GP, programs evolve through manipulation by genetic operators. To facilitate this, it must be possible to treat the programs being evolved as data. In this implementation, programs are represented internally as tree structures, allowing for relatively straightforward program manipulation.

An important consideration in GP is that although programs are considered data, there must also be a mechanism to invoke them. This mechanism must be integrated into the GP system as

program results are the criteria by which their fitness is judged, and fitness must be considered at each generational step. In GP terminology, a program is the genotype, and the expression of the genotype, the phenotype, is the program's result. In the system described here, the internal tree structure representing a program (genotype) can be evaluated directly to produce a texture image (phenotype).

4.1.1 Program Components

The components used to construct a texture synthesis program follow Sims' example (Section 3.2.1). A program is structured as a tree in which the internal nodes represent *functions*, and the leaves represent *terminals*. When evaluated, it produces a 2D texture image as its result.

Data Types

The following data types are used in texture programs. There are two base types out of which are constructed two compound types. Furthermore, these four types can be considered together as one abstract type.

Scalar A real number stored in a floating-point representation.

Color A three-component vector of scalars representing red, green, and blue.

Monochrome image A 2D array of scalars representing gray values.

Color image A 2D array of colors.

Image An abstract type which includes the first four (scalar, color, monochrome image, and color image).

A function's type is considered to be the type of its return value or result. In this implementation, all functions return values of the abstract type *image*. Therefore, to satisfy closure, functions (for the most part) also accept arguments of type *image*. Currently, terminals only belong to a subset of the possible types. In particular, they are one of the two base types (scalar and color).

Functions that take images as arguments generally invoke slightly different versions of code for different types of images. For the most part, they also return an image of the same type as their

argument(s). However, additional care must be taken when a function is invoked with multiple arguments that are images, but of different types. In this case, the arguments are coerced into a common type using a promotion scheme. The common type is chosen such that it is the minimal type that subsumes all of the arguments' types. This type is then used for the result. The hierarchy of types within the *image* abstract data type is depicted in Figure 4.1.

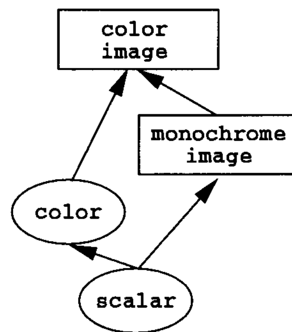


Figure 4.1: Hierarchy within the *image* abstract data type.

Function Set

The function set employed in this prototype implementation is similar to the one described in Sims [69]. Each function uses the abstract data type, *image*, as described on the page before. The majority of the functions used are either straightforward versions of operators commonly used to manipulate images [38, 63, 66], or to implement shaders [18].

The following is a complete list of the functions used. Full descriptions of each function are provided in Appendix A.

- `+`, `-`, `*`, `/`, `mod`, `round`, `min`, `max`, `abs`, `log`, `sin`, `cos`, and `atan` are standard mathematical functions modified to operate on arguments of type *image*.
- `and`, `or`, and `xor` perform bit-wise Boolean operations directly on the machine representation of floating-point numbers.
- `noise`, `cnoise`, `warped_noise`, `warped_cnoise`, `turbulence`, and `vlnoise` generate different types of procedural noise.

- `blur`, `laplace`, `hgrad`, `vgrad`, `combine`, and `blend` provide various image processing operations.
- `fft` and `invfft` perform forward and inverse Fourier transforms.
- `iflte` is a conditional operator with an implicit *if-less-than-or-equal-to* test.
- `hsvtorgb` performs a color space conversion from HSV to RGB.
- `warp` indexes an *image* using arbitrary transformations of the coordinate space.
- `picture` references a precomputed image stored in a file.

Terminal Set

The current implementation uses the following types of terminals. Each type represents a value belonging to the *image* abstract data type.

- `scalar` is a real-valued constant.
- `color` is a vector-valued (RGB) constant.
- `X` and `Y` are variables representing texture coordinates.

Programs

Texture synthesis programs are constructed of elements from the function and terminal sets just described. For example, the program

```
( blend ( noise 0.1 ) ( noise 0.4 ) 0.67 )
```

blends together two different frequencies of procedural noise. The tree corresponding to this program is shown in Figure 4.2 on the following page.

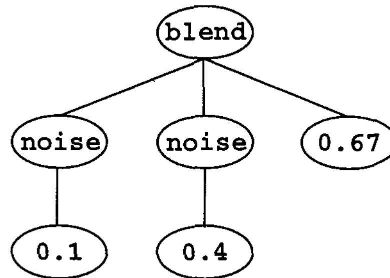


Figure 4.2: Program tree for `(blend (noise 0.1) (noise 0.4) 0.67)`.

The functions used in the current implementation are capable of producing a wide variety of textures. Figure 4.3 on the next page shows the results obtained using the following simple texture synthesis programs.

(a) `( turbulence 0.065 0.291 )`

(b) `( combine Y ( abs Y ) ( / Y -1 ) )`

(c) `( / 1 ( * X Y ) )`

(d) `( warped_cnoise ( * X 3 ) ( * 0.7 Y ) 0.05 )`

(e) `( fft ( and X Y ) )`

(f) `( blend [ 1 0 0 ] [ 0 0 1 ] ( noise 0.05 ) )`

(g) `( warped_cnoise 0.69 ( noise 0.04 ) 1.2 )`

(h) `( vgrad ( hsvtorgb ( turbulence 0.19 0.45 ) ) )`

(i) `( blend 0 [ 0 1 0 ] ( laplace ( iflte ( warped_noise ( 0.012 X 1.2 ) ( + 0.5 Y ) -1 0 ) ) )`

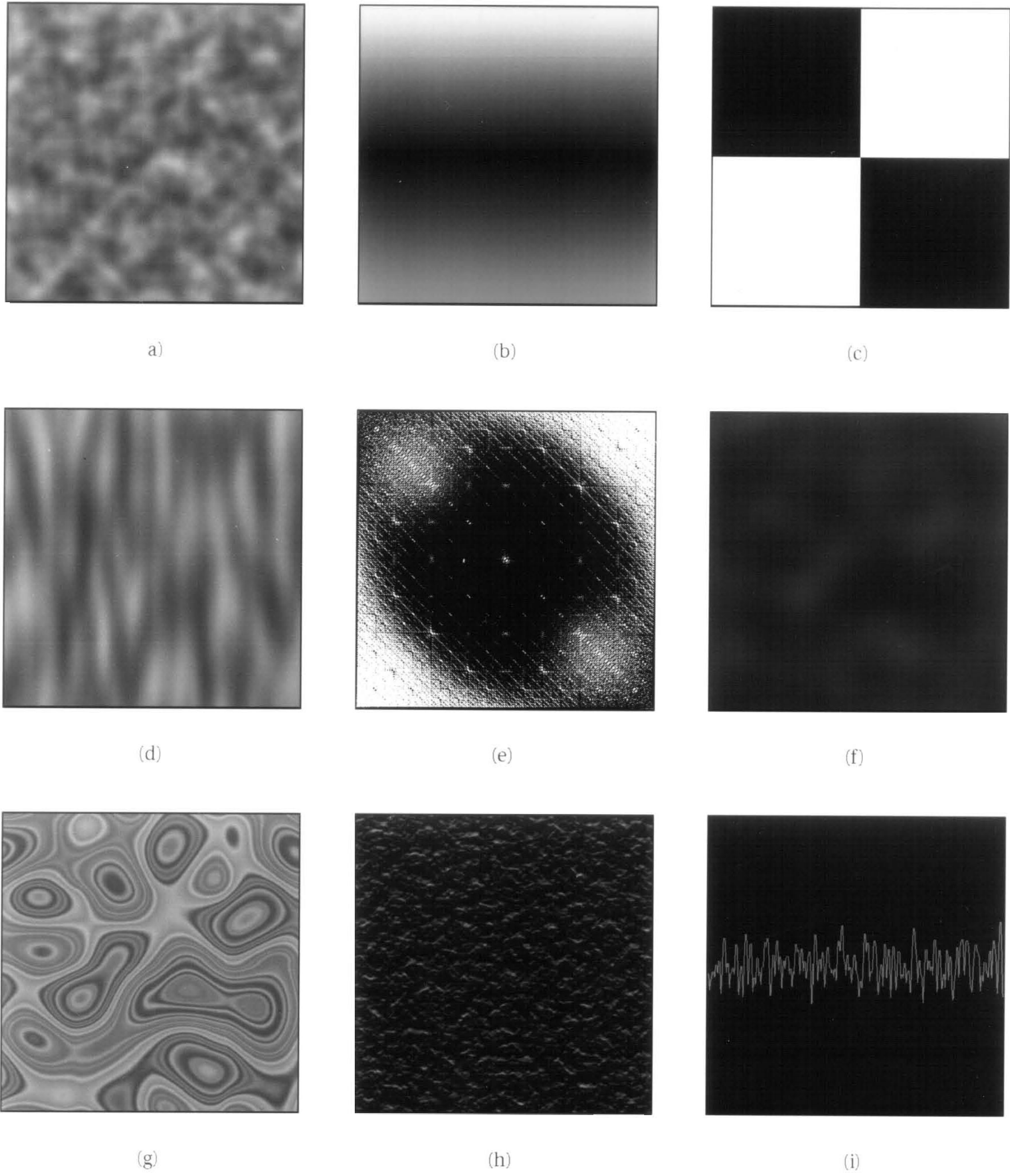


Figure 4.3: Simple texture synthesis programs.

4.1.2 Implementation

The texture design tool is implemented using the C++ programming language [75] on a Silicon Graphics Indigo2 Impact workstation [68].[1] The current implementation utilizes several vendor-specific user interface libraries, but does not make use of any available hardware acceleration in evaluating the texture programs.

Class Hierarchy

The current implementation involves a number of C++ classes divided roughly into three categories: core classes, user interface classes, and utility classes. The core classes are responsible for the representation, manipulation, and evaluation of the texture synthesis programs; the user interface classes implement and manage the graphical user interface; and the utility classes implement random distribution generators and other low-level functions. The user interface and utility classes are typical implementations of standard program features. Therefore, we will forego any further description on their part. We will, however, briefly discuss the role of each of the core classes in implementing a population of programs. How the core classes are combined to provide this implementation is shown in Figure 4.4 on page 51.

Population is a container class for the current generation of programs. It is responsible for maintaining selection lists, the current mode of operation, and various control parameters. This class is also responsible for creating the next generation from the mating pool through the application of genetic operators.

Genome instances represent the programs in the population. A `Genome` contains an instance of the `Dna` class (used to generate new genetic material), and references the `Gene` that is the root of its program tree.

Gene implements a node in a program tree. Furthermore, each node type in the function and terminal sets is represented by a different subclass of `Gene`. Each `Gene` maintains references to its parent in the program tree (if it is not the root) and to its arguments (if it is a function).

[1] Configured with a 195MHz MIPS R10000 RISC CPU and 128 megabytes of RAM.

A number of operations on a `Genome` (mutation, for example) are performed by traversing the program tree and invoking a procedure at each node. For many operations, different versions of the procedure must be used depending on the subclass. This is accomplished using the virtual method facility in C++. This scheme for structuring and operating on the program tree follows the pointer based implementation for GP described in Keith and Martin [40].

A program tree is evaluated to produce a texture (phenotype) by invoking a virtual method (`eval`) at the root node. The `eval` method is called recursively for any children of the node, and the results of evaluating each argument are used in computing the result. The `eval` method for each subclass is responsible for returning an instance of the `Image` class as its result.

Dna maintains the current function and terminal sets, and is responsible for randomly generating elements from these sets. It also acts as a constructor for the subclasses of `Gene`. In particular, it is used to instantiate the appropriate subclass of `Gene` corresponding to a function or terminal.

Image implements the program components described in Section 4.1.1. In particular, each of the functions and terminals is implemented by a method of this class. Therefore, for each subclass of `Gene`, there is a corresponding method in this class.

This class is the implementation of the *image* abstract data type and its associated type promotion mechanism (described in Section 4.1.1).

External Format

Externally, programs are described using a symbolic expression notation based on LISP S-expressions. This format is used when storing and retrieving programs and can be used as an intermediate format when implementing results in a shader language. The complete syntax for this notation is given in Appendix B.

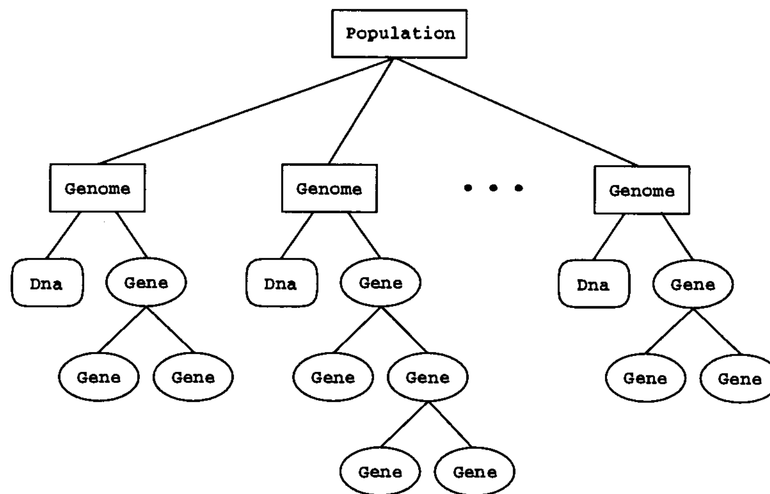


Figure 4.4: Internal structure of population.

4.2 Interactive Evolution

The texture design tool uses interactive evolution to create texture synthesis programs. As is typical of the interactive evolution systems described in Section 3.1, it maintains a small population of genotypes (programs) that evolve using the subjective judgement of the user as a fitness measure. The texture design tool employs a common user interface in implementing both the GP and GA layers described in Section 3.3.

4.2.1 User Interface

The main user interface element in the texture design tool follows the style of the applications based on interactive evolution described in Section 3.1. The main window displays the current generation of textures (phenotypes) in a grid arrangement. The current implementation uses a population of size twenty, arranged in four rows of five each. Programs are added to the selection set by choosing their phenotypes using a mouse. Figure 4.5 on the next page shows a typical window with the second and fourth phenotypes in the top row selected.

Operations involving either the entire population or the selection set are invoked from pull-down menus. The following operations are available in the prototype texture design tool:

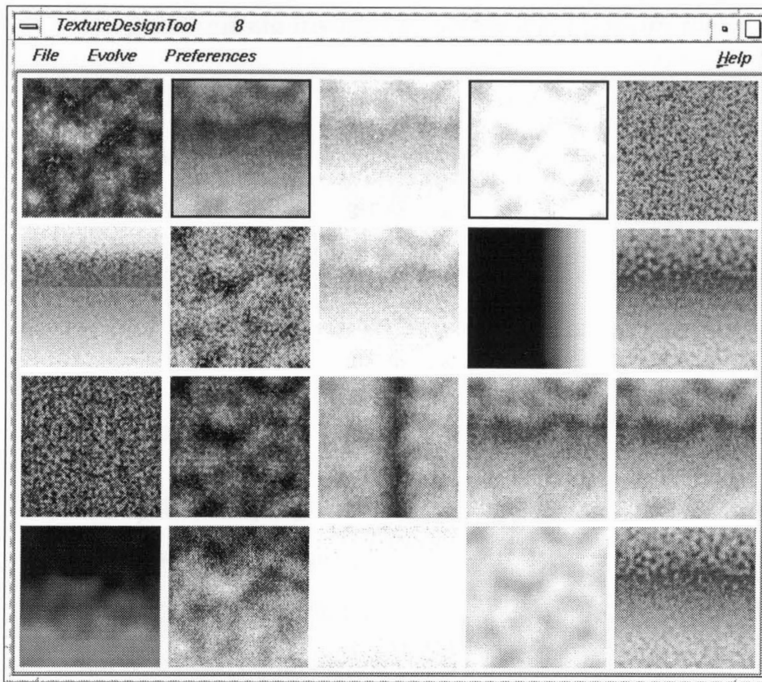


Figure 4.5: User interface for interactive evolution.

New population initializes the population with randomly generated programs. All programs that are not selected are replaced with new programs generated using the *grow* method described in Section 2.3.2.

Previous generation reverts the population to a previous generation within the session.

Save selected writes the selected program(s) to a file.

Load selected reads from a file replacing the selected program(s).

Evolve programs creates the next generation using GP operators with the selected program(s) used as the mating pool. The details of this operation are described in Section 4.2.2.

Evolve parameters creates the next generation using GA operators with the selected program(s) used as the mating pool. The details of this operation are described in Section 4.2.3.

The following operations can be performed on individual programs without adding them to

the selection set. These operations are invoked by activating a pop-up menu over a phenotype (texture image).

Edit opens an editing window for manipulating the program directly. This interface is described in Section 4.3.

Render generates a texture at the desired resolution and displays it in a separate window. From the separate window, the texture image can be saved to a file if desired.

4.2.2 Evolving Programs

Programs are evolved using genetic programming techniques for program manipulation. Each new generation is created based on the mating pool selected interactively by the user. This selection method implies a binary fitness metric (selected or rejected). Therefore, all members of the mating pool are considered to have equal fitness. A new generation is created by first copying the mating pool into the population for the next generation unaltered. The rest of the population is generated using selections from the mating pool. A slot in the population is filled in one of the three following ways:

Random A new, randomly generated individual is introduced into the population. The new program is generated using the same method that is used to initialize new populations.

Asexual A parent selected randomly from the mating pool is copied into the population. The offspring is then subjected to possible mutation.

Sexual Using two parents selected randomly from the mating pool, a single offspring is generated by recombining the two parents and copying the offspring into the population. Since each parent is selected independently, the same program may be selected for both parents resulting in an incestuous crossover.

In filling each open slot in the population for the next generation, the entire mating pool is used. Allowing reselection is necessary as the mating pool is typically a small subset of the population.

Before being copied into the population, each new program must meet the following criteria:

1. The depth of the tree structure for the new program must not exceed some reasonable maximum depth. This is to ensure that time is not wasted on overly complex programs.
2. The program's phenotype must not be equivalent to either a constant scalar or color value. This criterion culls programs that are likely to be unfit.

If either of these criteria is not met, the program is rejected and the slot is refilled.

Genetic Operators

In evolving programs, both sexual and asexual reproduction methods are employed. Sexual recombination (crossover) provides a means for *mating* two programs, and a variety of mutation operators are used when generating offspring asexually. The crossover operator employed is a single-offspring version of standard subtree crossover (Section 2.3.3). Two parents are chosen at random from the mating pool[2], then a crossover point is chosen at random within each parent. An offspring is produced by replacing the subtree rooted at the crossover point in one parent with the subtree rooted at the crossover point in the other parent. In order to encourage the exchange of significant genetic material, the selection of crossover points is biased towards choosing internal (function) nodes over leaf (terminal) nodes.

When a program is reproduced asexually, the offspring is subject to mutation before being copied into the population for the next generation. A program is mutated by traversing its expression tree and applying mutation operators to individual nodes. The number of nodes subject to mutation is governed by the *frequency of mutation*, and the nodes chosen for mutation are selected probabilistically.

Several different mutation operators are used. The operator to be applied to a node is chosen randomly, with each operator having an associated probability of being selected. Generally, each operator is only applicable to a subset of the nodes in a tree. Mutation is handled for each different function and terminal using virtual methods. The virtual method for each function or terminal restricts the set of operators to those that are applicable. The mutation operators included in the current implementation are as follows:

[2]Due to reselection, these may be the same parent.

1. The subtree rooted at the node is replaced with a new randomly generated subtree (subtree mutation).

2. A constant scalar or vector is offset by some random amount using Gaussian distributed noise (adjustment).

3. A node is replaced with a new randomly generated function or terminal (point mutation). Any arguments to the original node are retained as much as is possible, and any new arguments needed are generated randomly.

4. The subtree rooted at the node becomes an argument to a new randomly generated function which replaces the original (expansion). Any additional arguments needed are generated randomly.

5. The node is replaced with one of its arguments (reduction).

4.2.3 Evolving Parameters

In the genetic programming paradigm, both the structure and parameters of programs are evolved. In genetic algorithms, the structure of the solution is known, and the parameter space is searched using genetic operators. In our texture design tool we can evolve a program using a genetic algorithm approach by fixing the structure of a program in the population and evolving only its parameters.

To use a GA approach, we must construct a chromosome using an encoding of the parameters. In particular, we use a hybrid GA with a real-valued chromosome. We consider each terminal in the program tree that represents a constant value (either a scalar or color) to be a parameter of the program. Each of these parameters corresponds to a gene in the chromosome with its locus determined by its location in an ordering of the tree. Therefore, for a given program tree, we can programmatically create a corresponding real-valued chromosome. The relationship between a program tree and its chromosome is shown in Figure 4.6 on the following page.

It is important to note that creating a chromosome representation of a program neither alters, nor permanently fixes, the structure of its program tree. The internal program representation is

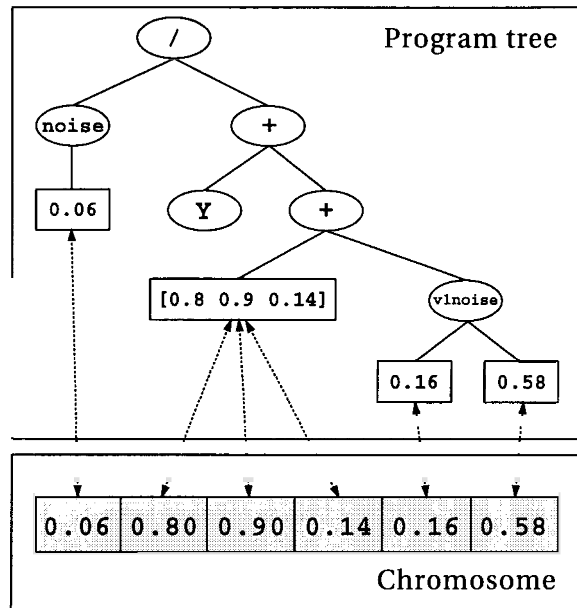


Figure 4.6: A chromosome corresponding to the parameters of a program tree.

retained, and can continue to be evolved using GP operators. However, with the chromosome representation, we can use an alternate method to evolve texture synthesis programs.

As when evolving programs, the mating pool selected interactively by the user is used to create the next generation. However, when a program is reproduced into the population for the new generation, its structure is copied unaltered. The genetic operators replace the parameters (chromosome) of the copied structure with those of the recombined or mutated offspring. The reproduction methods used are also analogous to those used when evolving programs. A new generation is created by first copying the mating pool into the population for the next generation unaltered. The remainder of the population is generated using the following methods:

Random A new, randomly generated individual is introduced into the population. A program structure from the mating pool is copied into the next generation. The parameters for the offspring are determined by a randomly generated chromosome.

Asexual A parent selected randomly from the mating pool is copied into the population. The chromosome for the offspring is then subject to possible mutation.

Sexual Using two parents selected randomly from the mating pool, a single offspring is generated. First, the structure of one of the parents is copied into the next generation. Then, the chromosomes of the two parents are recombined to become the chromosome of the offspring. For sexual recombination to have the desired result, the parent programs should have identical structures. However, if two parents are selected that have different program forms, crossover is still performed by aligning the chromosomes and using the maximum number of matching genes.

Genetic Operators

The method used for encoding a program's parameters in a chromosome allows for a number of domain-specific genetic operators to be used. In particular, since we know that each gene is a real number, we can use operators that are designed specifically for combining and perturbing real numbers. Moreover, since the members of the function set have been designed to work most effectively in a normalized space of $[0, 1]$, the operations can use this to their advantage.

The current implementation of the texture design tool recombines chromosomes in three ways:

1. Each gene is copied from one parent or the other into the offspring with equal probability (uniform crossover).

2. A random value is generated using a Gaussian distribution centered at 0.5. The value of each gene in the offspring is a linear combination of the two parental genes weighted by this value (blended crossover).

3. Each gene in the offspring is a uniformly distributed random value between the values of each parental gene (weighted uniform crossover). Note that uniform crossover is a special case of this operator in which the random value is always zero or one.

A chromosome is mutated by mutating its genes individually. The probability of a gene being subject to mutation is governed by the *frequency of mutation*. Since all of the genes are of the same type, the same operators can apply to all genes in the chromosome. Individual genes are mutated in one of two ways:

1. The current value of the gene is replaced with a new randomly generated value (point mutation).

2. The current value of the gene is offset using Gaussian distributed noise (adjustment).

4.2.4 Control Parameters

Other than the selection of the mating pool, interactive evolution is a stochastic process. In particular, both reproduction and the application of genetic operators use random variates. A number of control parameters are used to define the relative probabilities associated with each of these. For example, in creating a new generation, different methods of reproduction can be used. The relative probability of using each different method is given by a control parameter. Similarly, there are control parameters defining the relative probabilities of applying different crossover and mutation operators. Moreover, there may be different sets of probabilities depending on the context in which the operator(s) are to be applied. In addition to defining relative probabilities, there are also control parameters governing the frequency of mutation, selection preferences for crossover points, and tree depth.

4.3 Direct Manipulation

In addition to using interactive evolution to create texture synthesis programs, individual programs selected from the population can be manipulated directly using a graphical user interface. Programs edited using this interface remain in the population and can continue to be used as part of the gene pool in evolving subsequent generations. The editing interface is launched by invoking the *edit* operation from a pop-up menu over a program's phenotype display.

4.3.1 User Interface

The editing interface uses a visual representation of the program tree for interactive control (Figure 4.7). In particular, each node in the tree is an active control. Using the mouse, the user can select an operation from a pop-up menu over a node. From the pop-up menu, the user can choose

from a set of editing operations applicable to the node, or can launch a *thumbnail* display (described in Section 4.3.3).

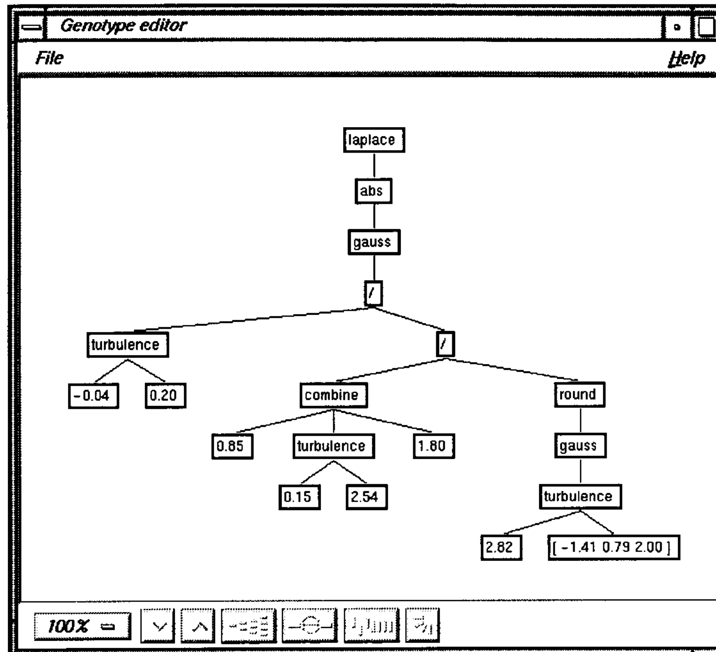


Figure 4.7: User interface for direct manipulation.

4.3.2 Editing Operations

In principle, it is possible to interactively apply any of the mutation operators described previously. It may also be useful to transform a program using operators that may not be applicable during evolution. In the texture design tool, we have implemented a selection of editing operators that allow for common types of adjustment to program structure and parameters. It is important to note that each operator is applicable only in certain contexts. The edit menu for a node only includes those that may be invoked on that node. The following editing operations are included:

Adjust allows a program parameter (a `scalar` or `color` terminal) to be adjusted directly. Values are adjusted using interactive controls appropriate for the type of parameter.

Change replaces a node with a new function or terminal chosen from a selection dialogue. Applying this operation may require that extraneous arguments be deleted, or new, randomly generated arguments be added. This operation is equivalent to the *point mutation* operator described in Section 4.2.2.

Expand makes the node an argument to a new function chosen from a selection dialogue. The new function assumes the original node's position in the program tree. If the new function requires more than one argument, additional arguments are generated randomly. This operation is equivalent to the *expansion* operator described in Section 4.2.2.

Reduce replaces the node's parent with itself. This causes any siblings of the node to be deleted and is equivalent to the *reduction* operator described in Section 4.2.2. This operation may not be applied to the root of the tree.

Permute re-orders a node's arguments. The new ordering is selected from a dialogue box. For some functions, certain orderings may be invalid, and for terminals and unary functions, it is a null operation. This operation is similar to the *permutation* operator described in Koza [43].

Collapse reduces the subtree rooted at the node to a terminal. First, the subtree rooted at the node is evaluated, then the average value of the result (either a `scalar` or `color`) is used to replace the subtree.

4.3.3 Data-flow Display

For any node, a *thumbnail* display window can be launched from its pop-up menu. A *thumbnail* is attached to a particular node and displays the texture computed as a result of the subtree rooted at that node. Any number of *thumbnails* may be open at one time, and each one is updated automatically to reflect any changes in the tree at, or below, its attachment point. This is similar in approach to many data-flow style displays used in image processing and scientific visualization.

Chapter 5

Results

In this thesis, we formulate a *genetic engineering* approach to texture synthesis and develop a prototype *texture design tool* implementing this model. Our motivation for this work is the difficulty associated with the specification of texture synthesis programs. In this chapter, we will discuss how the texture design tool described in this thesis provides an effective means to develop texture synthesis programs. In doing so, we will indicate how it addresses the specific limitations of previous work outlined in Section 3.2.3.

5.1 Genetic Engineering Approach

Typically, GA or GP systems rely solely on the evolutionary process to arrive at solutions. However, as Holland [37] points out, these techniques are best used to explore for potential solutions and that approximate results are most easily refined by augmenting the genetic method. In adopting a *genetic engineering* approach, we provide means to manipulate this process. In particular, we introduce more control over the rate of evolution as well as mechanisms for the specific alteration of genetic material.

We can consider the process of designing a texture synthesis program as consisting of a hierarchical arrangement of tasks. At the top level, a set of potential solutions is explored in search of a candidate. At the middle level, the candidate solution is refined to produce a more satisfactory

result. At the lowest level, specific adjustments are made to arrive at a final solution. It is important to note, however, that although these tasks form a hierarchy, the process may be iterative. For example, it may be useful to refine a program further after making a specific adjustment, or to evolve a new program after explicitly introducing genetic material.

In this section, we will discuss how the genetic engineering approach implemented in the texture design tool supports each of these three levels.

5.1.1 Exploration

The texture design tool's interactive evolution interface provides a means to explore the large space of texture programs using the genetic programming paradigm. Evolving texture programs using GP allows the user to generate complex results without explicitly specifying the underlying program. In particular, the process is guided purely by subjective aesthetic judgement. This method clearly addresses a fundamental problem associated with texture program design — difficulty of specification. In the genetic programming paradigm, programs are not specified, but rather, are evolved programmatically. Therefore, the user is not directly responsible for program specification.

In general, we have found that by using GP to evolve programs, the texture design tool is capable of generating a variety of complex and visually interesting textures. This confirms that the interactive evolution method described in Sims [69] shows promise as a viable method for creating texture synthesis programs. Figure 5.1 on page 68 shows the results of a population of programs evolved using only GP. The programs in the population shown are all descendants of the first three programs whose phenotypes are displayed in the top row. In this set, it can be seen how the genetic operators in GP produce offspring that tend to exhibit inherited characteristics, but can also differ substantially from their parents.

In a standard GP system, an initial random population is evolved until some termination criteria is reached (in this case, until a visually satisfying texture is found). Due to the necessarily small population size used in the texture design tool, the number of generations required to produce an interesting result can vary widely. In particular, the initial random population may be highly unfit, and therefore, more evolutionary steps may be necessary before fit individuals emerge. This

situation can be partially alleviated by allowing the population to occasionally receive an influx of new random genetic material. The texture design tool facilitates this by allowing the user to replace any number of programs in the population with new randomly generated ones. Effectively, this introduces a level of parallel random search for base genetic material.

For the most part, GP systems develop solutions from a *primordial soup* of programs. In particular, each session attempts to evolve a fit individual from a set of potentially primitive ancestors. In the texture design tool, we have found it useful to *seed* the population with individuals that are known to have been considered fit in the past.

Genetic Library

In developing the texture design tool we implemented a method to save program genotypes in an external format. We could then retrieve saved programs at any time. As more and more programs were saved, a sort of *genetic library* started to develop. This genetic library has proven useful beyond being merely an archive. When designing a new texture, the user may have some general criteria in mind. If a texture in the genetic library approximates these criteria, it can be used to *bootstrap* the population. As the library grows, the likelihood of a useful seed increases.

We have also found that many interesting textures contain useful *building-blocks* for different synthesis structures. For example, structures that separate texture coordinates into discrete tiles, or directional scale functions for warping coordinates are often useful. Introducing appropriate individuals from the library into an evolving population can provide needed genetic material that may otherwise take many generations to evolve.

A sufficiently diverse and well chosen genetic library could be used to increase the effective population size. Genetic operators could use the library as a source of material for recombination and mutation as well as introducing random individuals into the main population. Because the library contains only individuals that have achieved some level of fitness, the overall fitness of the effective population should increase.

5.1.2 Refinement

In the previous section, we describe how genetic programming can be used effectively to generate a variety of texture synthesis procedures. However, due to the granularity of the genetic operators, it can be difficult to refine an approximate solution using GP. In GP, the size and shape of the programs change as a result of mutation and recombination. As a result, the evolutionary steps tend to be large, making it difficult to focus design changes. In the texture design tool, this is exacerbated due to the high mutation rates introduced to compensate for the small population required for an interactive interface. To refine a solution, a more incremental approach similar to the one used by Dawkins [16, 17] would be useful.

In the texture design tool, we address the problem of refinement by constraining the amount of evolutionary change. Assuming that some characteristics of a texture synthesis program are controlled by independent variables (as is typical of most procedural textures), we can refine solutions by fixing the program structure and exploring only the parameter space of the program. This has the effect of delimiting the search space to the results of a particular parameterized procedure. Using this approach, the results of one evolutionary step in refining a candidate program are shown in Figure 5.2 on page 68. In this figure, each program represented is an offspring of the program corresponding to the phenotype shown in the upper left corner. This figure demonstrates that by eliminating structural change, the population appears more homogeneous than is typically the case when evolved using GP (Figure 5.1).

This type of refinement is supported in the texture design tool by using a hybrid genetic algorithm as an alternative to GP for evolving programs. In addition to constraining the search space, the GA layer allows us to take advantage of our knowledge of the parameter space to provide domain-specific genetic operators. In particular, the function and terminal sets employed in the programs use real-valued, normalized parameters. Therefore, we are able to use genetic operators that combine or perturb these parameters in useful ways with generally predictable results. For example, in the design scenario given in Section 3.2.3, we considered a situation in which the desired result was something *between* two different parameterizations of the same procedure. The crossover operators used in the GA layer of the texture design tool (described in Section 4.2.3) provide this functionality.

The texture design tool, using the GA layer, introduces a mechanism for more constrained control over the evolution process. This level of control provides a means for design refinement not easily obtained using a strictly GP system. Furthermore, it successfully addresses the specific limitations in previous work (described in Section 3.2.3) with respect to the refinement of approximate results.

5.1.3 Adjustment

Interactive evolution can be used effectively to generate texture synthesis procedures. Moreover, it accomplishes this without the user needing to directly specify or manipulate program code. However, in some cases, direct manipulation may be the most expedient way to achieve the desired result.

Using the IE interface of the texture design tool, potential textures can be explored and refined. However, often a texture program is evolved that would benefit from the adjustment of a small number of salient features. For example, a change in one of the contributing colors or a different frequency of procedural noise might be desired. To facilitate this, the texture design tool incorporates a visual interface for editing the programs directly (described in Section 4.3).

We have argued that the primary advantage of using an interactive evolution system is that the user is not required to specify the procedure. Moreover, the designer need not understand the potentially complex workings of the result. Given this, a facility for directly manipulating programs seems contradictory. However, it is possible to provide direct editing facilities that offer functionality without sacrificing abstractness.

For a designer to adjust some identifiable feature of a texture, they must be able to identify the program component responsible. The graph-based editor used in the texture design tool aids in this task. In particular, the display of program structure is analogous to the data flow within the program. In this form, the result corresponds to the root of the tree, and subtrees define components of the program. By displaying the partial results at different nodes, the user can quickly find the subtree responsible for a desired feature. The characteristics of that feature are then determined by the nodes in that subtree. Most often, a feature can be modified by simply adjusting the terminals of that subtree.

In addition to adjusting program parameters, we have found it useful to perform other more complex operations directly. For example, changing a procedural noise generator to one of a different type, or introducing additional control by adding a multiplier. These types of operations involve slightly more sophistication than merely adjusting parameters, but still a negligible amount. Of course, for the user inclined to do so, complex program manipulation can also be accomplished.

The operations supported by the graph-based editor in the texture design tool provide a facility for simple and direct design adjustment (our third stage of design). Furthermore, they address the second design limitation associated with strictly IE systems described in Section 3.2.3.

In addition to supporting editing operations, the graph-based editor can also be used to address the efficiency concerns raised in Section 3.2.3. A program's complexity can be reduced through the judicious application of the *collapse* operation. Using the display of partial results, it is possible for the user to identify introns in the program tree (subtrees whose result has no impact on the final result), and reduce them to a terminal. Similarly, the user may be able to identify unparsimonious subtrees. For example, a complex subtree may evaluate to a constant color. By collapsing this subtree to a `color` terminal, not only is the complexity reduced, but there is now a simple control available for that feature.

It may be possible to automate the process of reducing program complexity, but establishing sufficient criteria is not straightforward. A simple rule might be to collapse all subtrees that evaluate to a constant. However, one might also wish to collapse complex subtrees that have a minimal visual impact on the final result. Unfortunately, the definition of minimal is best left to subjective aesthetic judgement. Also, it may be desirable to collapse a subtree in order to introduce a simple control that when adjusted converts an enclosing subtree from an intron to a coding region. Again, this would require user intervention.

5.1.4 Summary

It is worthwhile noting that the additional controls provided by the GA and editing layers used for refinement and adjustment are only capable of producing structures that could evolve using the GP layer. Therefore, the GP layer subsumes the other two. However, using GP alone can require

an indefinite amount of time to reach the desired solution. Clearly, from a design point of view, this is impractical. Therefore, although the additional controls are theoretically redundant, they are eminently useful.

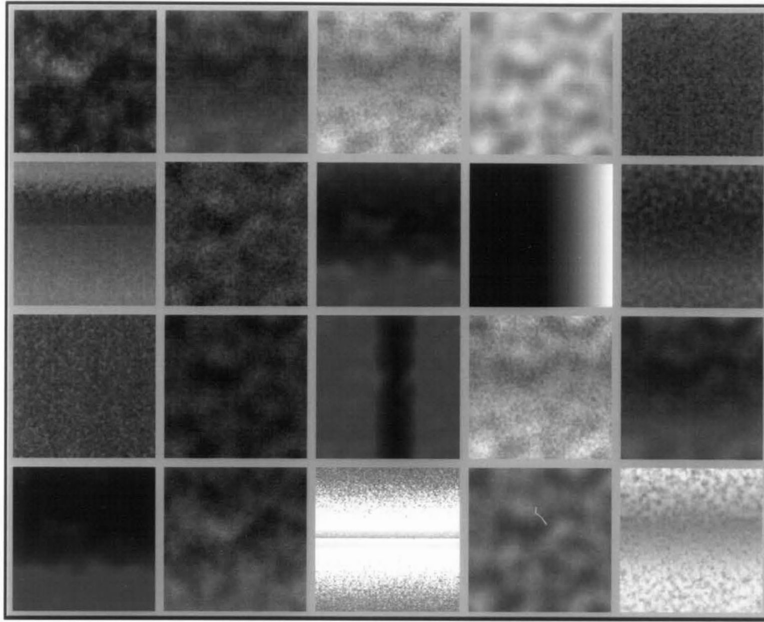


Figure 5.1: Population evolved using GP.

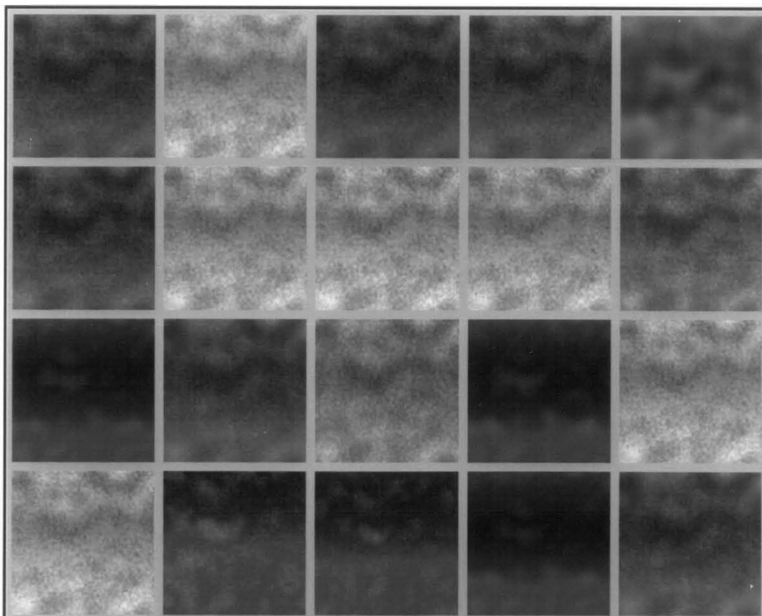


Figure 5.2: Population evolved using GA.

5.2 Examples

Using the texture design tool, a wide variety of results can be genetically engineered. In this section, we present a number of examples to show the diversity and complexity possible. In Figures 5.3 on page 71 and 5.4 on page 72, a selection of textures created using the texture design tool are shown. Each of these textures was derived from a random population using genetic programming. In many cases, additional random material was introduced to be recombined with evolving programs. Several of the textures were further refined using the GA layer in the texture design tool. Almost all of the textures shown were adjusted in some way using the direct manipulation interface.

It is worthwhile to note that none of the textures shown involved significant structural editing (except to reduce complexity). Direct editing, when employed, was primarily used to adjust colors, noise frequencies and blending parameters. In addition, multipliers were added to act as controls in several of the examples.

Most of the examples have been edited to remove obvious introns and to collapse constant-valued subtrees. In some cases, this has resulted in a considerable reduction in both program size and evaluation time. For example, the *turf* texture shown in Figure 5.3(c), is the result of the program evolved as:

```
( gauss ( warp ( warped_noise ( max ( turbulence 0.553487 0.125729 ) (
and ( turbulence 0.95874 -0.425775 ) ( and -0.117874 [ 0.309164 0.20771
0.512453 ] ) ) ) ( turbulence 1.06279 1.02221 ) 0.39901 ) -0.273012 (
combine ( * ( min ( / X [ 0.830889 0.418106 0.154696 ] ) ( - [ 0.205259
0.794183 0.993309 ] 0.831197 ) ) ( + ( and -1.31821 [ 0.849741 0.177322
0.431386 ] ) ( combine 1.42009 X 0.150112 ) ) ) ( combine ( / ( vlnoise
-0.228824 -0.0730438 ) ( warp 1.36315 [ 0.145681 0.50392 0.641761 ] [
0.357369 0.660575 0.600341 ] ) ) ( - ( gauss [ 0.618005 0.00859624
0.288083 ] ) ( max [ 0.040908 0.143739 0.735133 ] X ) ) ( warp ( gauss
X ) ( warp [ 0.62784 0.974512 0.518238 ] Y X ) ( cnoise 0.257875 ) ) )
```

```
( / ( and ( max X -0.156993 ) ( round Y ) ) ( log ( gauss -0.02709 ) )
) ) ) )
```

This program was easily reduced using the *collapse* operator to:

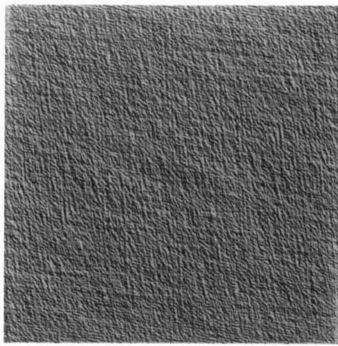
```
( gauss ( warp ( warped_noise ( turbulence 0.553487 0.125729 ) (
turbulence 1.06279 1.02221 ) 0.39901 ) -0.273012 ( combine 0 ( combine
( / ( vlnoise -0.228824 -0.0730438 ) [ 0.357369 0.660575 0.600341 ] ) (
- [ 0.618005 0.00859624 0.288083 ] ( max [ 0.040908 0.143739 0.735133 ]
X ) ) ( warped_cnoise X [ 0.637718 0.984328 0.527458 ] 0.26 ) ) 0 ) ) )
```

This reduction not only made it easier to adjust parameters (reducing the total number to approximately half), but also resulted in a 45% reduction in computation time.

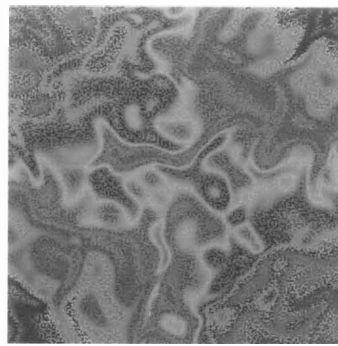
Each of the examples presented in this section required between 5 and 40 minutes (of real time) to evolve, refine, and edit starting with a random population. The programs used to generate the textures shown in Figures 5.3 and 5.4 are given in Appendix C. A selection of these textures are shown applied to 3D geometry and rendered using the *mental ray* raytracer in Figure 5.5 on page 73.

5.2.1 Image Processing

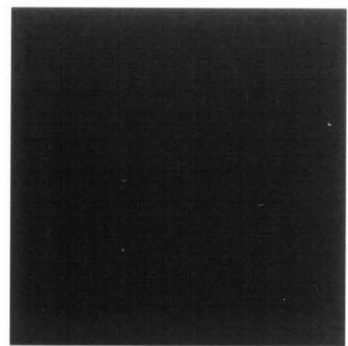
In addition to synthesizing textures from primitive elements, the texture design tool can also be used to modify existing images. By using the `picture` function to include images as terminals, general image processing programs can be designed. Since an external image is referenced by name, evolved programs can be applied to other source images, or an image reference can be replaced with a procedural result. Examples of image processing operators created with the texture design tool are shown in Figure 5.6 on page 73. In these examples, two different photographs are processed. In both cases, the original photograph is shown on the left, with two different processed versions shown to its right. The programs used to process the images are given in Appendix C.



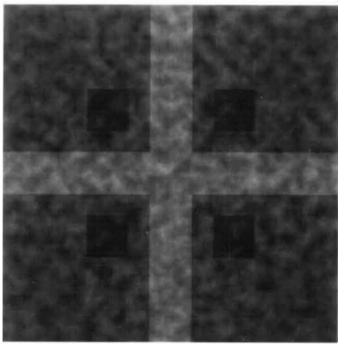
(a)



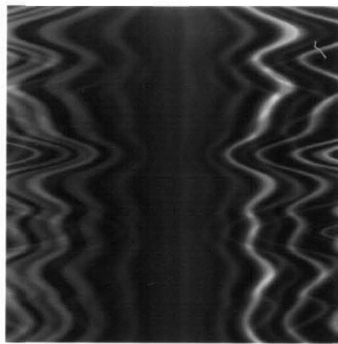
(b)



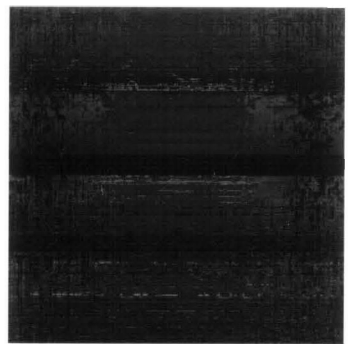
(c)



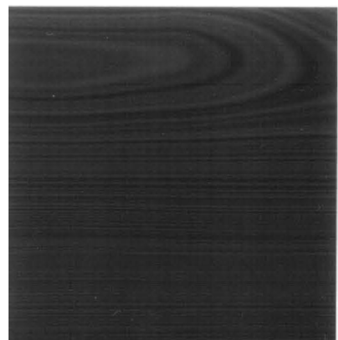
(d)



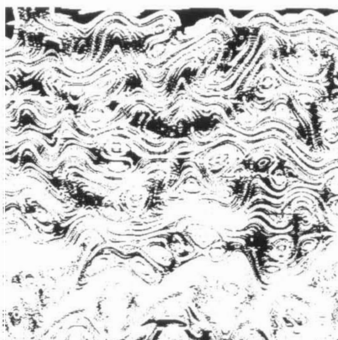
(e)



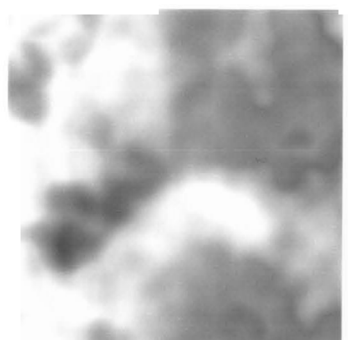
(f)



(g)

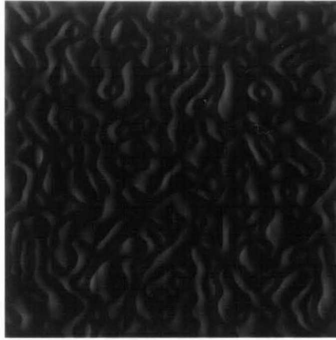


(h)

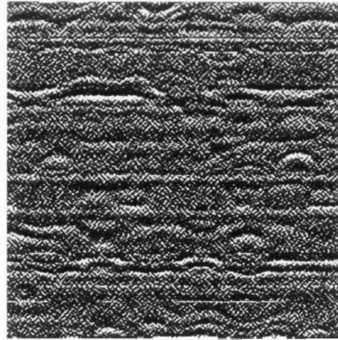


(i)

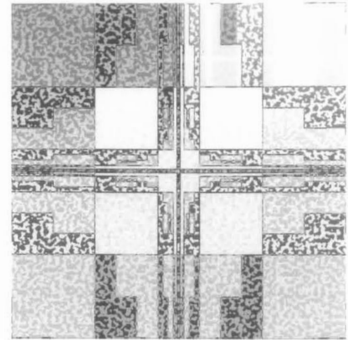
Figure 5.3: Textures generated using the texture design tool.



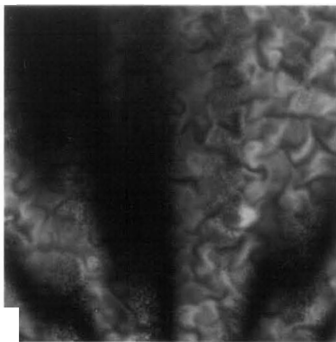
(a)



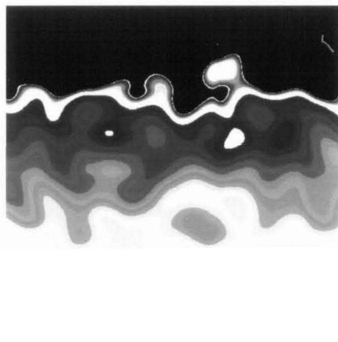
(b)



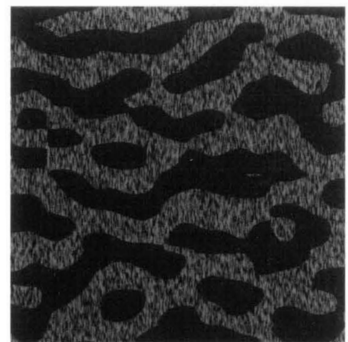
(c)



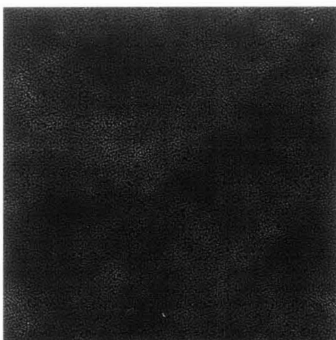
(d)



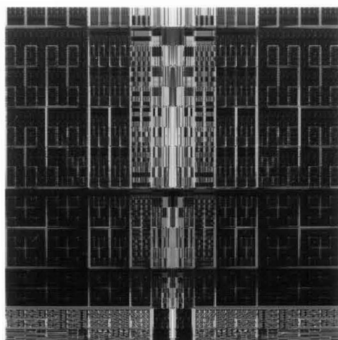
(e)



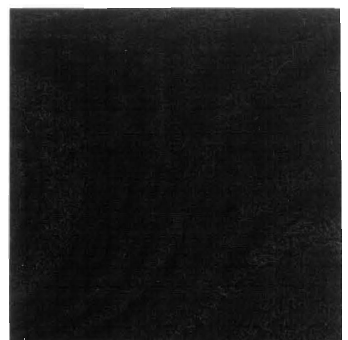
(f)



(g)



(h)



(i)

Figure 5.4: Textures generated using the texture design tool.

Figure 5.5: Textures applied to 3D geometry.

(a)

(b)

(c)

(d)

(e)

(f)

Figure 5.6: Image processing programs.

Chapter 6

Conclusion

In this thesis, a *genetic engineering* approach to texture synthesis is developed. This approach uses the genetic programming paradigm within an interactive evolution framework to evolve texture synthesis programs. In order to facilitate a more controllable design process, the genetic programming model is augmented with two additional layers of control: a hybrid genetic algorithm to evolve program parameters, and a graphical user interface to directly manipulate program genotypes. This approach is used in the implementation of a prototype texture design tool.

The genetic engineering model implemented in the texture design tool supports a hierarchical approach to the design of texture synthesis programs. A user is able to explore a large number of possible textures using the genetic programming paradigm. The designer interactively guides this process using subjective aesthetic judgement. Programs of interest can be further refined using a hybrid genetic algorithm. Again, this process is driven using an interactive evolution interface. Specific design modifications, such as the fine-tuning of parameters, are carried out using a graphical representation of a program's genotype.

A number of examples are provided showing the diversity of results obtainable using the texture design tool. A representative set of two-dimensional texture images designed entirely using the texture design tool is presented. In addition, several of these textures are shown applied procedurally in a rendered three-dimensional scene. Additional examples are included showing how the texture design tool can be used for the design of image processing programs.

6.1 Future Work

The texture design tool developed in this thesis serves as a useful prototype. However, as with any prototype, there are several extensions which may be beneficial. In addition, the development of the texture design tool has indicated a number of potential areas for future work.

- The existing application could be easily, and usefully, extended to include a more diverse function set. For example, more complex synthesis and processing operators could be included such as Worley's cellular basis function [83], or line integral convolution [9].

- The graphical user interface for directly manipulating programs could be refined and expanded. *Thumbnails* might better represent the data flow if integrated into the program tree display. In addition, some editing operations might be more easily performed by directly interacting with the tree. It may also be useful to allow programs to be constructed explicitly, then be evolved (a bottom-up approach). As well, the editing operations could be extended to allow recombination of genotypes. For example, to directly apply an analogue to crossover.

- In order to maintain interactivity, the complexity of the programs is limited in practice by processor power. The texture design tool could be easily extended to take advantage of specific hardware acceleration for image-processing operations, or converted to a coarse-grained distributed processing architecture.

- The current implementation uses a standard genetic programming model where programs are single, point-labeled trees. In order to take better advantage of functional *building-blocks*, a more elaborate model could be used. In particular, an architecture could be employed utilizing either automatically defined functions [45, 46], or an alternative method of evolving the function set [42]. Similarly, more complex program structures could be introduced. Program trees could incorporate higher level programming constructs such as iteration and recursion. Another possibility would be to allow the program architecture itself to be evolved [44].

- It might be interesting to experiment with predictive mutation operators like those used by Todd and Latham [77] and Graf and Banzhaf [27]. These methods should be relatively straightforward to add to the texture design tool. However, due to the nature of the function set, their usefulness is not guaranteed.

- Using the thumbnail displays, the task of isolating significant parameters within the program tree is simplified. However, as program size and complexity increases, this becomes more complicated. An interesting problem would be to see if a set of *macro* controls could be automatically derived that can control the parameters of a program using a standard set of definitions. For example, Rao and Lohse [64] propose a set of orthogonal dimensions describing fundamental texture characteristics. It might be possible to define controls for a given texture using this parameter space.

- An extremely challenging problem would entail reverse-engineering existing texture samples to create programs that could then be further evolved. Heeger and Bergen [34] have developed a method for synthesizing textures from digitized samples that performs well for a certain class of input. However, it does not apply well to inhomogeneous textures, such as tilings or those that contain discernible features. A general solution is unlikely in the near future. However, it may be useful to employ multiple methods to reverse engineer different classes of textures.

- A potentially practical application would be to define a function and terminal set that corresponds to the operators and variables associated with shading algorithms in a renderer. This would allow the interface to be used for designing general shaders.

6.2 Summary

The texture design tool implemented in this thesis shows considerable promise as a viable method for the design of complex texture synthesis procedures. It is capable of producing both novel and useful results within reasonable times. Furthermore, it accomplishes this without requiring a designer to directly specify the often complex and sophisticated program code commonly needed.

The texture design tool provides enhanced practical usability over previous methods employing only genetic programming. The addition of a hybrid genetic algorithm layer allows a designer to constrain and control the evolution process in order to refine a design. The graphical editing interface enables a user to quickly identify significant control parameters in order to fine-tune the result. In addition, it can be used effectively to isolate non-productive regions and eliminate them to increase the efficiency of the program.

The genetic engineering model implemented in the texture design tool also has potential application in other areas involving visual design. In many situations, it may be valuable to allow a user to explore design alternatives without being required to explicitly specify each variation. However, to complete a design, it must be possible to specifically direct design changes. The genetic engineering approach employed in this thesis addresses this need.

Although only a prototype, the texture design tool provides a combination of features not otherwise available to a designer. As a result, it provides superior functionality over existing applications for the design of texture synthesis procedures.

Appendix A

Function Set

The functions used by the texture design tool to construct programs are designed to compute *images* as their result, and, for the most part, take *images* as arguments. Each function performs its operation on a discrete two-dimensional image on a pixel by pixel basis. Each image, regardless of size, is indexed using normalized texture coordinates. The following functions comprise the set used in this thesis:

`( + a b )` Adds image `a` to image `b`.

`( - a b )` Subtracts image `b` from image `a`.

`( * a b )` Multiplies image `a` with image `b`.

`( / a b )` If `b` is 0, returns 0. Otherwise, divides image `a` by image `b`.

`( min a b )` Computes the minimum of image `a` and image `b`.

`( max a b )` Computes the maximum of image `a` and image `b`.

`( mod a b )` If `b` is 0, returns 0. Otherwise, computes image `a` *modulo* image `b`.

`( abs a )` Computes the absolute value of image `a`.

`( round a )` Rounds image `a` to the nearest integer.

`( and a b )` Computes the bit-wise *and* of image `a` and image `b`. The computation is performed on the floating-point representation of the image.

`( or a b )` Computes the bit-wise *or* of image `a` and image `b`. The computation is performed on the floating-point representation of the image.

`( xor a b )` Computes the bit-wise *exclusive-or* of image `a` and image `b`. The computation is performed on the floating-point representation of the image.

`( sin a )` Computes the sine of image `a`. The result is scaled to the range [0,1].

`( cos a )` Computes the cosine of image `a`. The result is scaled to the range [0,1].

`( atan a )` Computes the arctangent of image `a`. The result is scaled to the range [0,1].

`( log a )` Computes the natural logarithm of image `a`. The result is scaled to the range [0,1].

`( gauss a )` Convolves image `a` with the kernel

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

to perform a Gaussian blur.

`( laplace a )` Convolves image `a` with the kernel

$$\frac{1}{6}\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$

to perform a simple edge detection operation.

`( hgrad a )` Convolves image `a` with the kernel

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

to compute a discrete horizontal gradient.

(`vgrad a`) Convolves image a with the kernel

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

to compute a discrete vertical gradient.

(`fft a`) Computes a Fourier transform of image a.

(`ivfft a`) Computes an inverse Fourier transform of image a.

(`noise f`) Computes a Perlin gradient noise [60] with frequency f. The texture coordinates are used as implicit input coordinates and f is restricted to a *scalar* type.

(`colornoise f`) Computes a Perlin gradient noise [60] with frequency f independently for each color component. The texture coordinates are used as implicit input coordinates and f is restricted to a *scalar* type.

(`warped_noise u v f`) Computes a Perlin gradient noise [60] with frequency f. The input coordinates are given by the images u and v, and f is restricted to a *scalar* type.

(`warped_colornoise u v f`) Computes a Perlin gradient noise [60] with frequency f independently for each color component. The input coordinates are given by the images u and v, and f is restricted to a *scalar* type.

(`turbulence f m`) Computes a Perlin turbulence function [60] with frequency f and frequency multiplier m. The texture coordinates are used as implicit input coordinates and both f and m are restricted to a *scalar* type.

(`vlnoise f d`) Computes a variable lacunarity noise described by Musgrave [18] with frequency f and distortion factor d. The texture coordinates are used as implicit input coordinates and both f and d are restricted to a *scalar* type.

(`hsvtorgb a`) Transforms image a from HSV color space to RGB color-space using an algorithm from Foley et al. [19].

`( combine r g b )` Produces an image using the three images `r`, `g`, and `b` as the three color components.

`( blend a b x )` Computes an image by blending image `a` with image `b` using image `x` as the blend factor.

`( iflte a b c d )` Computes an image using an implicit comparison operator. If image `a` is less than or equal to image `b` the result is image `c`. Otherwise, it is image `d`.

`( warp u v a )` Computes an image by using images `u` and `v` as indices into image `a`.

Appendix B

Program Syntax

The following grammar defines the syntax for the external format used by the texture design tool to describe programs. This syntax also corresponds to the internal tree structure used in the texture design tool given in prefix notation.

⟨genome⟩ ⟶ (⟨gene⟩)

⟨gene⟩ ⟶ ⟨add⟩ | ⟨subtract⟩ | ⟨multiply⟩ | ⟨divide⟩ | ⟨min⟩ | ⟨max⟩ |
⟨mod⟩ | ⟨abs⟩ | ⟨round⟩ | ⟨and⟩ | ⟨or⟩ | ⟨xor⟩ | ⟨gaussian⟩ |
⟨laplacian⟩ | ⟨h-gradient⟩ | ⟨v-gradient⟩ | ⟨fft⟩ |
⟨inverse-fft⟩ | ⟨sine⟩ | ⟨cosine⟩ | ⟨arctangent⟩ | ⟨log⟩ |
⟨noise⟩ | ⟨color-noise⟩ | ⟨warped-noise⟩ | ⟨warped-color-noise⟩ |
⟨turbulence⟩ | ⟨vl-noise⟩ | ⟨hsv-to-rgb⟩ | ⟨combine⟩ | ⟨blend⟩ |
⟨if-less-than-or-equal-to⟩ | ⟨warp⟩ | ⟨picture⟩ |
⟨pixel⟩ | ⟨color⟩ | ⟨scalar⟩

⟨add⟩ ⟶ (+ ⟨gene⟩ ⟨gene⟩)

⟨subtract⟩ ⟶ (- ⟨gene⟩ ⟨gene⟩)

⟨multiply⟩ ⟶ (* ⟨gene⟩ ⟨gene⟩)

⟨divide⟩ ⟶ **(/ ⟨gene⟩ ⟨gene⟩)**

⟨min⟩ ⟶ **(`min` ⟨gene⟩ ⟨gene⟩)**

⟨max⟩ ⟶ **(`max` ⟨gene⟩ ⟨gene⟩)**

⟨mod⟩ ⟶ **(`mod` ⟨gene⟩ ⟨gene⟩)**

⟨abs⟩ ⟶ **(`abs` ⟨gene⟩)**

⟨round⟩ ⟶ **(`round` ⟨gene⟩)**

⟨and⟩ ⟶ **(`and` ⟨gene⟩ ⟨gene⟩)**

⟨or⟩ ⟶ **(`or` ⟨gene⟩ ⟨gene⟩)**

⟨xor⟩ ⟶ **(`xor` ⟨gene⟩ ⟨gene⟩)**

⟨gaussian⟩ ⟶ **(`gauss` ⟨gene⟩)**

⟨laplacian⟩ ⟶ **(`laplace` ⟨gene⟩)**

⟨h-gradient⟩ ⟶ **(`hgrad` ⟨gene⟩)**

⟨v-gradient⟩ ⟶ **(`vgrad` ⟨gene⟩)**

⟨fft⟩ ⟶ **(`fft` ⟨gene⟩)**

⟨inverse-fft⟩ ⟶ **(`invfft` ⟨gene⟩)**

⟨sine⟩ ⟶ **(`sin` ⟨gene⟩)**

⟨cosine⟩ ⟶ **(`cos` ⟨gene⟩)**

⟨arctangent⟩ ⟶ **(`atan` ⟨gene⟩)**

⟨log⟩ ⟶ **(`log` ⟨gene⟩)**

⟨noise⟩ ⟶ **(`noise` ⟨scalar⟩)**

⟨color-noise⟩ ⟶ **(`cnoise` ⟨scalar⟩)**

⟨warped-noise⟩ ⟶ **(`warped_noise` ⟨gene⟩ ⟨gene⟩ ⟨scalar⟩)**

⟨warped-color-noise⟩ ⟶ **(`warped_cnoise` ⟨gene⟩ ⟨gene⟩ ⟨scalar⟩)**

⟨turbulence⟩ ⟶ **(`turbulence` ⟨scalar⟩ ⟨scalar⟩)**

⟨vl-noise⟩ ⟶ **(`vlnoise` ⟨scalar⟩ ⟨scalar⟩)**

⟨hsv-to-rgb⟩ ⟶ **(hsvtorgb** ⟨gene⟩ **)**

⟨combine⟩ ⟶ **(combine** ⟨gene⟩ ⟨gene⟩ ⟨gene⟩ **)**

⟨blend⟩ ⟶ **(blend** ⟨gene⟩ ⟨gene⟩ ⟨gene⟩ **)**

⟨if-less-than-or-equal-to⟩ ⟶ **(iflte** ⟨gene⟩ ⟨gene⟩ ⟨gene⟩ ⟨gene⟩ **)**

⟨warp⟩ ⟶ **(warp** ⟨gene⟩ ⟨gene⟩ ⟨gene⟩ **)**

⟨picture⟩ ⟶ **{** *imagename* **}**

⟨pixel⟩ ⟶ **X** | **Y**

⟨color⟩ ⟶ **[** ⟨red⟩ ⟨green⟩ ⟨blue⟩ **]**

⟨red⟩ ⟶ ⟨scalar⟩

⟨green⟩ ⟶ ⟨scalar⟩

⟨blue⟩ ⟶ ⟨scalar⟩

⟨scalar⟩ ⟶ *real number*

Appendix C

Example Programs

The programs corresponding to the examples shown in Figures 5.3 on page 71, 5.4 on page 72, and 5.6 on page 73 are as follows:

Figure 5.3(a)

```
( * 0.793 ( atan ( fft ( warped_noise X ( and ( abs ( + Y ( + ( noise
0.0894939 ) ( + X 0.517532 ) ) ) ) ( fft ( / ( turbulence -0.284625
-0.843481 ) Y ) ) ) -0.0773555 ) ) ) )
```

Figure 5.3(b)

```
( hsvtorgb ( min ( blend -0.299623 ( warped_noise ( + ( vlnoise
-0.0773153 -1.63315 ) ( * X Y ) ) ( and ( turbulence -1.0031 0.707749 )
( / Y Y ) ) 0.137458 ) ( cos ( combine ( abs 1.57762 ) ( combine
[ 0.498877 0.818949 0.424351 ] -0.864037 Y ) ( laplace X ) ) ) )
[ 0.132977 0.208504 0.742424 ] ) )
```

Figure 5.3(c)

```
( gauss ( warp ( warped_noise ( turbulence 0.553487 0.125729 ) (
turbulence 1.06279 1.02221 ) 0.39901 ) -0.273012 ( combine 0 ( combine
```

```
( / ( vlnoise -0.228824 -0.0730438 ) [ 0.357369 0.660575 0.600341 ] )
( - [ 0.618005 0.00859624 0.288083 ] ( max [ 0.040908 0.143739 0.735133 ]
X ) ) ( warped_cnoise X [ 0.637718 0.984328 0.527458 ] 0.26 ) ) 0 ) ) )
```

Figure 5.3(d)

```
( - ( - ( turbulence -0.12706 0.715664 ) ( and ( and Y [ 0.875942
0.253831 0.195372 ] ) X ) ) ( and ( and ( and Y [ 0.875942 0.253831
0.195372 ] ) [ 0.875942 0.253831 0.195372 ] ) X ) )
```

Figure 5.3(e)

```
( - ( - ( / X ( warped_cnoise ( / X ( warped_cnoise Y -1.25985
-0.0823928 ) ) -1.25985 -0.0823928 ) ) X ) ( - X 0.194661 ) )
```

Figure 5.3(f)

```
( hsvtorgb ( and ( mod Y ( warped_cnoise ( iflte Y ( laplace ( - ( min
( min Y X ) ( round -0.759688 ) ) ( round ( noise 0.102823 ) ) ) ) )
0.000362914 ( atan ( + ( combine ( mod 0.397413 Y ) ( + Y [ 0.0786573
0.787291 0.172843 ] ) Y ) [ 0.991102 0.910452 0.9536 ] ) ) ) ( or (
warped_cnoise Y -0.642404 1.33972 ) ( atan ( vlnoise -0.145881 -1.51107
) ) ) -0.550043 ) ) ( gauss ( - [ 0.344016 0.55521 0.714202 ] ( / X (
blend ( / ( turbulence -1.32591 -0.727532 ) ( blend X ( laplace [
0.0513501 0.625347 0.708985 ] ) ( vlnoise -0.430886 1.14337 ) ) ) ( max
( noise -0.201062 ) ( vgrad ( mod X -0.348112 ) ) ) ( hgrad ( / ( mod
-0.324329 Y ) ( warped_noise X X 0.811769 ) ) ) ) ) ) ) ) )
```

Figure 5.3(g)

```
( blend [ 0.070588 0.0392157 0.00392157 ] [ 0.454902 0.403922 0.337255 ]
( warped_noise ( blend Y ( hsvtorgb ( turbulence -0.00706075 -0.199053 )
)( gauss Y ) ) 1.55009 -1.12015 ) )
```

Figure 5.3(h)

```
( or ( fft ( fft ( * -0.853484 Y ) ) ) ( gauss ( vgrad ( atan ( vgrad
( + ( and ( atan -0.121768 ) ( + ( noise 0.0894939 ) ( + Y 0.517532 ) )
) ( + ( noise 0.0894939 ) ( + Y 0.517532 ) ) ) ) ) ) ) )
```

Figure 5.3(i)

```
( blend ( turbulence 0.027308 0.467137 ) ( gauss ( hsvtorgb
( turbulence 0.026 0.517 ) ) ) X )
```

Figure 5.4(a)

```
( * 2 ( hgrad ( gauss ( hsvtorgb ( noise 0.134074 ) ) ) ) )
```

Figure 5.4(b)

```
( vgrad ( + ( hgrad ( log ( turbulence 1.07898 -0.255015 ) ) ) ( /
( warped_noise ( and ( laplace -1.59036 ) ( combine [ 0.756225 0.933451
0.0971272 ] 0.147115 -1.403 ) ) ( log ( warped_noise Y [ 0.00779032
0.439196 0.994108 ] 0.985297 ) ) 0.0250437 ) ( warped_noise ( hsvtorgb
( mod Y [ 0.697445 0.788174 0.499839 ] ) ) ) ( abs ( cnoise -0.0687338 )
) 0.265385 ) ) ) )
```

Figure 5.4(c)

```
( cos ( + ( log ( combine ( laplace ( log ( log ( cnoise 0.604988 ) ) )
) Y ( gauss ( hsvtorgb ( and 0.825707 ( - 1.17772 Y ) ) ) ) ) ) ) ( or (
iflte [ 0.148981 0.493705 0.815772 ] [ 0.766372 0.482488 0.149179 ] (
min ( warped_cnoise ( log ( and 1.42816 [ 0.350527 0.942373 0.844572 ] )
) ( log ( combine X Y -0.0305977 ) ) 0.133791 ) Y ) ( vlnoise -0.892233
0.53369 ) ) ( iflte ( combine ( mod ( turbulence 0 0.644947 ) ( hsvtorgb
( / 0.140727 [ 0.711297 0.848649 0.996823 ] ) ) ) -0.790699 ( combine (
```

```
/ ( noise -0.537927 ) ( hsvtorgb [ 0.483102 0.281036 0.54312 ] ) ) (
warped_noise ( cos [ 0.0580386 0.274143 0.530461 ] ) ( laplace [ 0.34912
0.0950274 0.471114 ] ) -1.09461 ) ( hsvtorgb Y ) ) ) ( atan Y ) ( / (
laplace ( round ( or X Y ) ) ) -1.13976 ) ( vgrad -0.421917 ) ) ) ) )
```

Figure 5.4(d)

```
( blend ( sin ( - [ 0.296061 0.0605026 0.815732 ] ( / ( - ( blend ( sin
( - [ 0.296061 0.060502 0.815732 ] ( / ( * X [ 0.830596 0.83634 0.136028
] ) [ 0.296061 0.060502 0.815732 ] ) ) ) ( vgrad [ 0.695407 0.870632
0.838287 ] ) ( vlnoise -1.29758 -1.31393 ) ) ( / ( * X [ 0.83059 0.83634
0.136028 ] ) ( atan Y ) ) ) ( atan Y ) ) ) ) ( vgrad ( fft ( hsvtorgb (
gauss [ 0.695407 0.870632 0.838287 ] ) ) ) )( vlnoise 0.206406 -1.31393
) )
```

Figure 5.4(e)

```
( or ( - ( noise 0.079 ) Y ) [ 0.172549 0.168627 1 ] )
```

Figure 5.4(f)

```
( blend [ 0.0196078 0.0156863 0.00392157 ] [ 0.780392 0.584314 0 ] (
atan ( or ( * ( fft ( warped_noise X ( fft ( / ( turbulence 0.057
-0.843481 ) X ) ) -0.395 ) ) 0.037 ) ( gauss ( vgrad ( + ( noise
0.0894939 ) ( + X 0.517532 ) ) ) ) ) ) )
```

Figure 5.4(g)

```
( laplace ( abs ( gauss ( / ( turbulence -0.0384502 0.201 ) ( / (
combine 0.8538 ( turbulence 0.152 2.53948 ) 1.8 ) ( round ( gauss
( turbulence 2.82247 [ -1.40754 0.78592 2.00362 ] ) ) ) ) ) ) ) )
```

Figure 5.4(h)

```
( laplace ( laplace ( laplace ( hsvtorgb ( or ( and [ 0.496834 0.215828
0.000936845 ] X ) ( + 0.536 ( / Y 1.918 ) ) ) ) ) ) )
```

Figure 5.4(i)

```
( laplace ( hsvtorgb ( log ( warped_cnoise ( blend Y -1.21864 ( blend
X [ 0.364093 1.71072 0.393419 ] ( blend Y [ 0.573389 0.362213 0.75348 ]
X ) ) ) ( vlnoise -1.03953 1.10975 ) -0.10725 ) ) ) )
```

Figure 5.6(a)

```
( { monument } )
```

Figure 5.6(b)

```
( - ( + ( - [ 0.243137 0.243137 0.243137 ] { monument } ) ( iflte
{ monument } [ 0.737255 0.737255 0.737255 ] { monument } [ 1 1 1 ]
) ) ( hgrad { monument } ) )
```

Figure 5.6(c)

```
( iflte { monument } 0.745 ( blend { monument } ( - ( / { monument }
( cnoise 0.178149 ) ) -0.114815 ) ( * 0.174 ( blend ( - ( / { monument }
( cnoise 0.178149 ) ) -0.114815 ) { monument } [ 0.0230223 0.0370927
0.0176135 ] ) ) ) ( * ( turbulence 0.049 3 ) [ 0.407843 0.439216
0.592157 ] ) )
```

Figure 5.6(d)

```
( { jazz } )
```

Figure 5.6(e)

```
( blend ( * ( vgrad { jazz } ) ( * 3 [ 1 0.654902 0.643137 ] ) )
{ jazz } 0.5 )
```

Figure 5.6(f)

```
( gauss ( warp ( + X ( * 0.098 ( noise 0.14 ) ) ) ( + Y ( * 0.061
( noise 0.25145 ) ) ) { jazz } ) )
```

References

[1] ANDRE, D., AND TELLER, A. A study in program response and the negative effects of introns in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference* (1996), MIT Press, pp. 12–20.

[2] BAKER, E., AND SELTZER, M. Evolving line drawings. In *Proceedings of Graphics Interface '94* (1994), pp. 91–100.

[3] BALUJA, S., POMERLEAU, D., AND JOCHEM, T. Towards automated artificial evolution for computer-generated images. *Connection Sciences 6*, 2–3 (1994), 325–354.

[4] BANZHAF, W. Interactive evolution. In *Handbook of Evolutionary Computation.* IOP Publishing Ltd. and Oxford University Press, 1997, ch. 2. To appear.

[5] BEIER, T., AND NEELY, S. Feature-based image metamorphosis. In *Computer Graphics (SIGGRAPH '92 Proceedings)* (1992), vol. 26, pp. 35–42.

[6] BLINN, J. F. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (1978), vol. 12, pp. 286–292.

[7] BLINN, J. F., AND NEWELL, M. E. Texture and reflection in computer generated images. *Communications of the ACM 19*, 10 (Oct. 1976), 542–547.

[8] BRODATZ, P. *Textures: A Photographic Album for Artists and Designers.* Dover Publications, New York, 1966.

[9] CABRAL, B., AND LEEDOM, L. Imaging vector fields using line integral convolution. In *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), vol. 27, pp. 263–270.

[10] CALDWELL, C., AND JOHNSTON, V. S. Tracking a criminal suspect through "face-space" with a genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (1991), pp. 416–421.

[11] CATMULL, E. *A Subdivision Algorithm for the Computer Display of Curved Surfaces.* PhD thesis, University of Utah, 1974.

[12] COOK, R. L. Shade trees. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (1984), vol. 18, pp. 223–231.

[13] DARWIN, C. *On the Origin of Species by Means of Natural Selection.* Murray, London, 1859.

[14] DAVIS, L. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence. Morgan Kaufmann, 1987.

[15] DAVIS, L. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[16] DAWKINS, R. *The Blind Watchmaker.* Longman, 1986.

[17] DAWKINS, R. The evolution of evolvability. In *Artificial Life: Proceedings of the First Artificial Life Workshop* (1988), C. G. Langton, Ed., Santa Fe Institute Studies in the Science of Complexity, Addison-Wesley, pp. 201–220.

[18] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. *Texturing and Modeling: A Procedural Approach.* AP Professional, 1994.

[19] FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. *Computer Graphics: Principles and Practices,* second ed. Addison-Wesley, 1990.

[20] FOURNIER, A., FUSSELL, D., AND CARPENTER, L. Computer rendering of stochastic models. *Communications of the ACM 25,* 6 (June 1982), 371–384.

[21] FU, K. S., AND LU, S. Y. Computer generation of texture using a syntactic approach. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (1978), vol. 12, pp. 147–152.

[22] GAGALOWICZ, A., AND DE MA, S. Model driven synthesis of natural textures for 3-D scenes. *Computers and Graphics 10,* 2 (1986), 161–170.

[23] GARDNER, E. J., AND SNUSTAD, D. P. *Principles of Genetics*, sixth ed. John Wiley and Sons, 1981.

[24] GARDNER, G. Y. Simulation of natural scenes using textured quadric surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (1984), vol. 18, pp. 11–20.

[25] GARDNER, G. Y. Visual simulation of clouds. In *Computer Graphics (SIGGRAPH '85 Proceedings)* (1985), vol. 19, pp. 297–303.

[26] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[27] GRAF, J., AND BANZHAF, W. An expansion operator for interactive evolution. In *Proceedings of the IEEE International Conference on Evolutionary Computation* (Perth, Australia, 1995), IEEE Press, pp. 798–802.

[28] GRAF, J., AND BANZHAF, W. Interactive evolution of images. In *Proceedings of the Fourth Conference on Evolutionary Programming (EP95)* (1995), D. Fogel, Ed.

[29] HAGGERTY, M. Evolution by esthetics. *IEEE Computer Graphics and Applications 11*, 2 (Mar. 1991), 5–9.

[30] HANRAHAN, P., AND HAEBERLI, P. E. Direct WYSIWYG painting and texturing on 3D shapes. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (1990), vol. 24, pp. 215–223.

[31] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)* (1990), vol. 24, pp. 289–298.

[32] HARUYAMA, S., AND BARSKY, B. A. Using stochastic modeling for texture generation. *IEEE Computer Graphics and Applications 4*, 3 (Mar. 1984), 7–21.

[33] HECKBERT, P. S. Survey of texture mapping. *IEEE Computer Graphics and Applications 6*, 11 (Nov. 1986), 56–67.

[34] HEEGER, D. J., AND BERGEN, J. R. Pyramid-based texture analysis/synthesis. In *Computer Graphics (SIGGRAPH '95 Proceedings)* (1995), pp. 229–238.

[35] HILLIS, W. D. The connection machine. *Scientific American 255*, 6 (June 1987).

[36] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

[37] HOLLAND, J. H. Genetic algorithms. *Scientific American 267*, 1 (July 1992), 66–72.

[38] HOLZMANN, G. J. *Beyond Photography: The Digital Darkroom.* Prentice Hall, 1988.

[39] KAUFMAN, A., AND AZARIA, S. Texture synthesis techniques for computer graphics. *Computers and Graphics* 9, 2 (1985), 139–145.

[40] KEITH, M. J., AND MARTIN, M. C. Genetic programming in C++: Implementation issues. In *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, pp. 285–310.

[41] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice-Hall, 1978.

[42] KINNEAR, JR., K. E. Alternatives in automatic function definition: A comparison of performance. In *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, pp. 119–142.

[43] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[44] KOZA, J. R. Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming. Tech. Rep. STAN-CS-94-1528, Stanford University, Stanford, California, Oct. 1994.

[45] KOZA, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[46] KOZA, J. R. Scalable learning in genetic programming using automatic function definition. In *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, pp. 99–118.

[47] LATHAM, W. Sculptures in the void. *New Scientist 125*, 1701 (Jan. 1990), 40–45.

[48] LEWIS, J. P. Texture synthesis for digital painting. In *Computer Graphics (SIGGRAPH '84 Proceedings)* (1984), vol. 18, pp. 245–252.

[49] LEWIS, J. P. Shape and texture generation by neural network creation paradigm. In *Proceedings of Graphics Interface '91* (1991), pp. 129–133.

[50] MAGNENAT-THALMANN, N., AND THALMANN, D. *Image Synthesis: Theory and Practice.* Springer-Verlag, 1987.

[51] MCCONKEY, E. H. *Human Genetics: The Molecular Revolution.* Jones and Bartlett, 1993.

[52] MENTAL IMAGES GESSELSCHAFT FÜR COMPUTERFILM UND MASCHINENINTELLIGENZ MBH & CO. KG. *mental ray Programmer's Reference Guide.* Berlin, 1995.

[53] MEZEI, L., PUZIN, M., AND CONROY, P. Simulation of patterns of nature by computer graphics. In *Information Processing '74 (Proceedings of IFIP Congress '74)* (1974), North-Holland, pp. 861–865.

[54] MICHALEWICZ, Z. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, 1992.

[55] NGUYEN, T., AND HUANG, T. Evolvable 3D modeling for model-based object recognition systems. In *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, pp. 459–476.

[56] NORDIN, P., AND BANZHAF, W. Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, L. Eshelman, Ed. Morgan Kaufmann, 1995, pp. 310–317.

[57] NORDIN, P., FRANCONE, F., AND BANZHAF, W. Explicitly defined introns and destructive crossover in genetic programming. In *Advances in Genetic Programming II*, P. Angeline and K. E. Kinnear, Jr., Eds. MIT Press, 1996, ch. 17.

[58] OPPENHEIMER, P. The artificial menagerie. In *Artificial Life: Proceedings of the First Artificial Life Workshop* (1988), C. G. Langton, Ed., Santa Fe Institute Studies in the Science of Complexity, Addison-Wesley, pp. 251–274.

[59] PEACHEY, D. R. Solid texturing of complex surfaces. In *Computer Graphics (SIGGRAPH '85 Proceedings)* (1985), vol. 19, pp. 279–286.

[60] PERLIN, K. An image synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)* (1985), vol. 19, pp. 287–296.

[61] PIXAR. *The RenderMan Interface*, September 1989. Version 3.1.

[62] PIXAR. *PhotoRealistic RenderMan User's Manual*, January 1993. UNIX Version.

[63] PRATT, W. K. *Digital Image Processing*, second ed. John Wiley & Sons, 1991.

[64] RAO, A. R., AND LOHSE, G. L. Towards a texture naming system: Identifying relevant dimensions of texture. In *Proceedings of Visualization '93* (1993), IEEE, pp. 220–227.

[65] ROTHWELL, N. V. *Understanding Genetics*, fourth ed. Oxford University Press, 1988.

[66] RUSS, J. C. *The Image Processing Handbook*. CRC Press, 1992.

[67] SCHACTER, B., AND AHUJA, N. Random pattern generation processes. *Computer Graphics and Image Processing 10*, 2 (1979), 95–114.

[68] SILICON GRAPHICS INC. *Indigo2 IMPACT Workstation Owner's Guide*. Mountainview, CA, 1996.

[69] SIMS, K. Artificial evolution for computer graphics. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (1991), vol. 25, pp. 319–328.

[70] SIMS, K. Interactive evolution of dynamical systems. In *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life* (1992), F. J. Varela and P. Bourgine, Eds., MIT Press, pp. 171–178.

[71] SIMS, K. Interactive evolution of equations for procedural models. *The Visual Computer 9*, 8 (1993), 466–476.

[72] SMITH, J. R. Designing biomorphs with an interactive genetic algorithm. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (1991), pp. 535–538.

[73] SRINIVAS, M., AND PATNAIK, L. M. Genetic algorithms: A survey. *Computer 27*, 6 (1994), 17–26.

[74] STEELE, G. L. *Common LISP: the language.* Digital Press, Burlington, Mass., 1984.

[75] STROUSTRUP, B. *The C++ Programming Language*, second ed. Addison-Wesley, 1991.

[76] TODD, S., AND LATHAM, W. Artificial life or surreal art? In *Towards a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life* (1992), F. J. Varela and P. Bourgine, Eds., MIT Press, pp. 504–513.

[77] TODD, S., AND LATHAM, W. *Evolutionary Art and Computers.* Academic Press, 1992.

[78] TURK, G. Generating textures on arbitrary surfaces using reaction-diffusion. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (1991), vol. 25, pp. 289–298.

[79] UPSTILL, S. *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics.* Addison-Wesley, 1990.

[80] VAN WIJK, J. J. Spot noise: Texture synthesis for data visualization. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (1991), vol. 25, pp. 309–318.

[81] WITKIN, A., AND KASS, M. Reaction-diffusion textures. In *Computer Graphics (SIGGRAPH '91 Proceedings)* (1991), vol. 25, pp. 299–308.

[82] WOLBERG, G. *Digital Image Warping.* IEEE Computer Society Press, 1990.

[83] WORLEY, S. A cellular texture basis function. In *Computer Graphics (SIGGRAPH '96 Proceedings)* (1996), pp. 291–294.

[84] YESSIOS, C. I. Computer drafting of stones, wood, plant and ground materials. In *Computer Graphics (SIGGRAPH '79 Proceedings)* (1979), vol. 13, pp. 190–198.