Gentropy: Evolving 2D Textures

Andrea L. Wiens
50 Lakeshore Road, Unit #94
St. Catharines, Ontario,
Canada L2N 6P8
janet.wiens@sympatico.ca

Brian J. Ross[*]
Department of Computer Science
Brock University
St.Catharines, Ontario,
Canada L2S 3A1
bross@cosc.brocku.ca

Abstract

Gentropy is a genetic programming system that evolves two-dimensional procedural textures. It synthesizes textures by combining mathematical and image manipulation functions into formulas. A formula can be re-evaluated with arbitrary texture-space coordinates, to generate a new portion of the texture in texture space. Most evolutionary art programs are interactive, and require the user to repeatedly choose the best images from a displayed generation. *Gentropy* uses an unsupervised approach, where one or more target texture image are supplied to the system, and represent the desired texture features, such as colour, shape and smoothness (contrast). Then, *Gentropy* evolves textures independent of any further user involvement. The evolved texture will not be identical to the target texture, but rather, will exhibit characteristics similar to it. When more than one texture is supplied as a target, multiobjective feature analysis is performed. These feature tests may be combined and given different priorities during evaluation. It is therefore possible to use several target images, each with its own fitness function measuring particular visual characteristics. *Gentropy* also permits the use of multiple subpopulations, each of which may use its own texture evaluation criteria and target texture.

Keywords: procedural textures, texture synthesis, evolution, genetic programming

[*] Corresponding author (email: bross@cosc.brocku.ca; phone 905-688-5550 ext 4284; fax 905-688-3255).
Preprint from *Computers & Graphics*, v.26, n.1, Feb 2002, pp. 75-88.

1 INTRODUCTION

Computers have established themselves as indispensable tools in artistic applications. One such application is the generation of interesting images and textures for desktops, web pages, documents and animation programs. While a computer program has no concept of creativity, it is still a powerful tool to be used by an artist for generating fascinating textures. Given the mathematical nature of procedural (or algorithmic) textures, computer software is well suited to their generation [1, 2]. On the other hand, random texture generation is not a practical strategy. Of the virtually infinite number of procedural textures possible, a relatively small proportion of them will be aesthetically interesting and useful to the artist.

One technique found to be useful for constraining the random generation of textures is evolutionary computation [3, 4, 5]. Evolution is a practical search paradigm for searching a vast space of candidate solutions. This applies to texture generation, given the enormous number of texture formulas possible. Since computer software is currently unable to make aesthetic and artistic judgments about textures, the most practical solution up to now has been to incorporate the user in the judgment task during texture creation [6, 7, 8, 9]. Interactive evolutionary texture generation applications take a given texture formula, and perform varying degrees of mutation upon it. This population of textures is displayed to the user. The user interactively selects the mutated texture of most interest, thereby determining the direction of the texture generation process. The user also determines the amount of mutation performed on it. Typically, early stages in the synthesis process will use high levels of mutation. As the user focuses on a texture of interest, the mutation rate will be reduced. Interactive texture evolution therefore is an iterative process, in which the user drives the search towards a texture of interest. Of special note is the Genshade system, which evolves Renderman shaders [10]. Genshade takes an unsupervised approach, in which automatic image analysis is performed during genetic evolution.

This paper presents the *Gentropy* system. Like other evolutionary art applications, *Gentropy* uses evolutionary computation to explore the space of texture formulas. However, *Gentropy* is an unsupervised texture synthesis system. Its major contribution is that it uses a robust set of automated image analyses to evaluate the suitability of candidate textures, while at the same time uses a fairly rudimentary set of texture generation operators. The user supplies *Gentropy* with one or more target texture images. The user also indicates how each texture is to be evaluated, based on a suite of image analysis tests. A wide variety of subpopulation architectures are definable by the user. Each subpopulation uses one or more textures and feature tests, and feeds its results into other similarly specialized subpopulations. *Gentropy* takes this information, and at the end of a run generates a solution texture, which will exhibit characteristics of the target textures. Note that it is not the goal to find a procedural texture that generates a texture identical to the target. Rather, the target texture is used for synthesizing a texture that is similar to it, according to the specified image analyses criteria.

An outline of the paper is as follows. Section 2 discusses evolutionary computation and genetic programming. Section 3 reviews the concept of procedural texture generation, and discusses the procedural texture formulas used in *Gentropy*. Section 4 discusses the types of image analyses performed in *Gentropy*. Other details about the *Gentropy* system are given in Section 5. Some example results are presented in Section 6. Comparisons to related work are given in Section 7. A discussion concludes the paper in Section 8. Some example texture formulas as obtained by *Gentropy* for the results in Section 6 are given in Appendix A.

2 EVOLUTIONARY COMPUTATION AND GENETIC PROGRAMMING

Evolutionary computation is a search strategy inspired by natural evolution. The most popular evolutionary search technique is the genetic algorithm (GA) [3, 4]. An example GA is in Fig. 1. To use a GA to solve a problem, an encoding must be derived which maps candidate problem solutions into a binary chromosome or string. These strings are considered “individuals” to be tested and reproduced during the course of the GA run. A population of random chromosomes is initially created. Each individual is tested and given a fitness measurement, which indicates its relative merit towards solving the problem of interest. To create a new population of individuals, fitness proportional selection of individuals is performed. This models the concept of “Darwinian survival of the fittest”, since the stronger individuals (those with better fitness scores) will be more likely to be selected for reproduction than the weaker individuals. The most common types of reproduction operators are crossover (genetic recombination) and mutation. Crossover takes two selected parent individuals, and randomly mixes their chromosomes to create two offspring. The power of crossover stems from the idea that, as in nature, offspring may inherit the favourable characteristics of each parent. In other words, crossover is the means in which desirable genetic characteristics can be combined and inherited in subsequent generations. Mutation is used to maintain population diversity during evolution. The power of GA comes from the fact that, using fitness-proportional selection and crossover reproduction, a vast number of problem prototypes can be explored during a run. This has been called “implicit parallelism” in the literature.

A powerful variation of GA is genetic programming (GP) [5]. Rather than using fixed linear chromosomes of bits, GP denotes individuals as executable computer programs, typically denoted by parse trees. Crossover is performed by selecting branches in two parse trees, and swapping the branches. Mutation is done by replacing a branch with a randomly generated one. Both these operations are performed in a way that preserves syntactic correctness of programs. In addition, it is important that a *closure property* be maintained: the primitive functions and statements of the programming language being used must be able to execute any possible data arguments without producing an error. For example, an arithmetic division operator must not crash if a zero-valued denominator is given to it. So long as syntactic correctness and closure are maintained, GP essentially works in the same manner as a conventional GA. When a program is to be evaluated for its fitness, it is executed upon the problem of interest, its ability to solve the problem is measured, and the resulting fitness score is assigned to the program.

GA and GP are widely used in scientific, engineering, and artistic applications. Their implicit parallelism makes them both very effective in finding solutions in large search spaces. Naturally, GP is particularly adept at solving problems that are more algorithmic in nature, since GP evolves programs. A representative selection of applications of GA and GP can be found in [11].

3 PROCEDURAL TEXTURES

Procedural or algorithmic textures are patterns that are produced using an assortment of mathematical and algorithmic operators. The input to a texture formula is normally the coordinates of the pixel being rendered. The operators in a texture formula return greyscale or colour data. The rest of this section describes the texture elements used in *Gentropy*.

3.1 TEXTURE FUNCTION SET

The texture function set consists of various mathematical and image processing functions that take an RGB colour (vector) or channel (scalar) as arguments and return either a vector or scalar. A vector is represented by three floating-point values, or *channels*, while a scalar is a single floating-point channel, defining a shade of grey. *Gentropy* uses the following function set:

```
X Y ERC VERC COLGRAD + - diff * / sin cos mod log pow not avg max min if
lum rgb noise turb turbflow cloud marble warprel warpabs kaleid tile
tilerad forv chn wchn
```

The above operations are combined into a *texture formula* in such a way that the output is always an RGB colour vector. The `X` and `Y` parameters denote the coordinates of the pixel being rendered. Some example texture formulas and their rendered results are given in Table 1. Since the functions that make up the texture formula may return any floating-point value, the colour channels are truncated to a range of -1 to 1, before the conversion to an integer range of 0 to 255 (required by the RGB image file format).

Terminals (constants, and functions that take no arguments) have upper-case names. `ERC` returns a random scalar and `VERC`, a random vector. They appear in a texture formula as either a floating-point number or a set of three floating-point numbers within parentheses. In `COLGRAD`, which creates a colour gradient, the red and green values are the current x and y positions and the blue value is the distance to the origin. The `lum` operation finds the luminance, average of all channels, which is a grey colour. Also, there is the `rgb` function, which takes three arguments and returns an RGB colour. The other functions are discussed below.

3.2 NOISE FUNCTIONS

The `noise`, `turb`, `turbflow`, `cloud` and `marble` functions are more complex, generating shaded bumps, curves, gases and other natural-looking effects. See [1] for in-depth discussions of these functions. Table 2 shows a few examples. These functions are capable of producing a wide variety of textures. The `noise` function is the basis for the others. It creates a grid of random grey points and then interpolates between them. `X` and `Y` arguments specify the position within the noise texture and the grid size is determined by a `frequency` argument. Noise values are divided by a varying amplitude value and added together in the `turb` (turbulence) operation.

The following formula is for `turb`, where `col` holds the grey colour value returned, and `x` and `y` define the current position in texture space. `Start`, `end` and `frequency` are all arguments to the function. Example values for these variables are 1, 256 and any positive floating-point number, respectively.

$$col = \sum_{i=0}^{\infty} \frac{noise(2^i \cdot x, 2^i \cdot y, freq)}{2^i \cdot start}$$

where the maximum i is such that $(2^i \cdot start) \le end$.

Amplitude and frequency are both varied in the `turbflow` (turbulent flow) function. In the following pseudo-code for `turbflow`, the `num` iteration variable is positive and small, for example, less than 6. `Persistence` is also positive and less than or equal to 1. The `turboflow` formula is:

$$col = \sum_{i=1}^{num} \frac{noise(x, y, 2^{-i})}{persistence^{i}}$$

`Cloud` is based on turbulence [12]. See [1] for details on `marble` and other noise operations.

3.3 WARPS AND TILING

In the function set, there are several warp and tiling operations, which alter the positions of or duplicate pixel values. The warps take the current x and y positions in texture space and alter them by adding an increment value to each (relative) or assigning them new values (absolute). In addition to the relative (`warprel`) and absolute (`warpabs`) warps, there are three tiling operations (`tile`, `tilerad` and `kaleid`), which create rectangular, circular and kaleidoscopic repeating patterns.

3.4 AUTOMATICALLY DEFINED ITERATION

The final set of three operations performs iteration over a finite colour vector. This iterative control is termed automatically defined iteration or ADI ([13], p.121). The function `forv` performs the actual iteration, like a `for`-loop in C or `dotimes` in the LISP programming language, with the current channel as the looping variable.

Operations `chn` and `wchn` either return the appropriate scalar value (one channel of a colour vector) based on the current channel or the current channel plus some increment value, respectively. If there is a `chn` or `wchn` in the subtree, different values will likely be returned for each iteration, as the channel varies, and `forv` returns a (non-grey) colour. Otherwise, an iteration will have the same result and a grey colour will be produced. For example, the last texture in Table 2 has several non-grey colours due to its use of the `wchn` operator.

4 IMAGE ANALYSIS

Selection of individual textures for reproduction depends on their fitness with respect to the rest of the population. *Gentropy*'s fitness function assigns every individual texture a fitness measure between 0 and 1, with 1 indicating a perfect score. This fitness measure is determined by feature tests, which compare the generated image to the target image. This section discusses the suite of image analyses possible within *Gentropy*.

4.1 IMAGE COMPARISON

Colour quantization is used to ease matching, particularly the shape aspect. It categorizes the colours so that very similar colours are represented by the same colour. This stops analyses of minute colour differences that the human eye normally cannot judge anyway. However, this does introduce some error to the matching process. For instance, the fitness function may deem two images identical, even though their colouring or shape is slightly different. The colour space — or number of colours involved — is reduced so that histograms are smaller.

Histograms represent the colour distribution of an image by storing the frequency of each quantized colour [14]. They are used by many of the feature tests in *Gentropy*, including those that match directly and those that match histograms. *Direct matching* involves the comparison of two images, pixel by pixel, while *histogram matching* calculates a histogram for the images and then measures the distance between the histograms. Position is irrelevant in the latter but not the former. In other words, direct matching includes shape information in its analysis.

4.2 FEATURE TESTS

Gentropy uses image comparison methods, mainly adapted from *query by image content* (QBIC) applications [14, 15]. These programs take a target image and search a database for similar images. They often use histograms and wavelet analysis to match images. Table 3 shows several feature tests the user may choose from as part of the fitness function.

It is extremely difficult to do an accurate automatic evaluation of an image. Precise image comparisons require a host of complex, computation-intensive computer vision concepts, which are impractical to apply in a genetic algorithm. *Gentropy* uses a selection of efficient image analyses, which evaluate basic image attributes, such as colour, shape and smoothness. Complex analyses, from computer vision research for instance, would result in more accurate evaluations of images, but a serious price would be paid in run-time speed.

4.2.1 Colour Direct Matching

Colour direct (`CDIR`) is the simplest matching operation. `CDIR` adds the colour similarities of each pixel in the two images and then divides by the number of pixels per image. It uses colour similarity, a measure of the distance between two colours in RGB space. The distance between two RGB colours `C1` and `C2`, where `C1 = (r1,g1,b1)` and `C2 = (r2,g2,b2)`, is given by:

$$dist(C1,C2) = \sqrt{(r1-r2)^2 + (g1-g2)^2 + (b1-b2)^2}$$

The distance is divided by the square root of 3 (the maximum possible distance within the cube) and subtracted from 1 so that the similarity value, `sim(C1,C2)`, ranges between 0 and 1, with 1 representing a colour match. Then, the value is squared in order to exaggerate small differences in similar colours.

$$sim(C1,C2) = (1-(dist(C1,C3)/\sqrt{3}))^2$$

Table 4 shows some example colour similarities.

4.2.2 Colour Histogram Matching

Unlike `CDIR`, colour histogram (`CHIST`) is not concerned with the colour's position within the image. It creates colour histograms from the two quantized images being compared and then finds the histogram distance.

Let `i` be an index into the two histograms (`hist1` and `hist2`), representing a quantized colour. The values of $\mathtt{hist1}_\mathtt{i}$and $\mathtt{hist2}_\mathtt{i}$ hold the frequency of quantized colour `i` within their respective images. The distance for index `i` is given by:

$$\mathtt{dist}_\mathtt{i} = |\mathtt{hist1}_\mathtt{i}-\mathtt{hist2}_\mathtt{i}|$$

`CHIST` returns the histogram distance, which is the sum of all $\mathtt{dist}_\mathtt{i}$ divided by 2.

4.2.3 Colour Histogram Quadratic Matching

`CHISTQ` adds colour similarity to the calculation above in order to find the quadratic histogram distance. This method is used in QBIC systems like VisualSEEk [15].

Let `i` and `j` be indexes into the two histograms (`hist1` and `hist2`), representing quantized colours whose frequencies the histograms measure. The similarity between the quantized colours, represented by the histogram indexes `i` and `j` is given by `sim(i,j)`. The distance between these two indexes is given by:

$$\mathtt{dist(i,j)} = |\mathtt{hist1}_\mathtt{i}-\mathtt{hist2}_\mathtt{i}| * \mathtt{sim(i,j)} * |\mathtt{hist1}_\mathtt{j}-\mathtt{hist2}_\mathtt{j}|$$

The colour histogram quadratic distance is summed for all of the possible combinations of `i` and `j` (depending on the histogram size), in order to obtain an overall measure of histogram difference. Although this computation is somewhat expensive, it allows for the testing of colour closeness, as opposed to merely considering exact colour matches.

4.2.4 Wavelet Analysis

The wavelet (`WAV`) test compares the shape of two images. It creates a frequency image using the colour histogram of the quantized image. Colours with low frequency, as determined by the histogram, are replaced with dark greys, while frequent colours are assigned lighter greys. The result is a grey image with the same shape as the original colour image, but the specific colours are removed. This frequency image can then be used to match shape, without regard to the actual colours involved.

The details of wavelet processing are beyond the scope of this paper; see [16] for an in-depth discussion. WAV uses a basic Haar wavelet decomposition, where each row and then each column in the image is recursively compressed using an averaging and differencing operation. Next, a similar decomposition is performed on each column in the image. Finally, all but the most positive and the most negative values (coefficients) in `imag` are truncated (set to zero). Wavelet analysis is very useful because its shape matching precision is adjustable by specifying the number of coefficients to be kept after truncation. The result of this procedure is a wavelet image of the most significant coefficients, either positive or negative, represented as white or black pixels, respectively. The final step is the comparison of two wavelet images by matching the positions of the white and black pixels.

Table 5 shows some example wavelet representations. Notice that the examples share a few coefficients (white or black dots in the same position) but they are quite different.

4.2.5 Smoothness Histogram Matching

Smoothness histogram (`SHIST`) matches smoothness or contrast by generating a smoothness image, consisting of grey colours indicating the deviation of each pixel from its eight neighbours. Colour similarities between the particular pixel and its surrounding pixels are averaged to find a local deviation measure. If a pixel is very similar to its neighbours, it will have a dark grey in the smoothness image, whereas light colours indicate high contrast. The darker the grey, the smoother the area around the pixel. These grey values make up a smoothness image (see Table 6). Then, `SHIST` creates a histogram and the distance between two such histograms provides the smoothness measure. Review `CDIR` and `CHIST` for detailed descriptions of colour similarity and histogram distance.

Note that this technique may be used to compare edges by matching two smoothness images directly. It may be useful for target images with many edges.

5 GENETIC PROGRAMMING SYSTEM

Gentropy uses the strongly-typed *lil-gp* kernel, a GP system written in the C programming language [17]. Table 7 shows several lil-gp parameters. The maximum nodes parameter refers to the approximate maximum number of operators permitted in a texture formula, while maximum depth indicates the maximum level of nesting within a formula. The initial random population is constructed using a half-and-half algorithm [5], in which the formulas have depths evenly distributed between 2 and 6. There is a 90% chance that crossover will be used during reproduction, and 10% chance that mutation is applied. Fitness-proportional selection is done using tournament selection: 5 individuals are selected from the population at random, and the one with the best fitness is the ‘selected’ individual to be used for reproduction. Evolution stops when the best fitness of this subpopulation is greater than some threshold value (set extremely high so unlikely to happen), or 100 generations has been reached.

Island model GA’s are those which use separate subpopulations or *demes* during evolution. Subpopulations are often useful in GA evolution, as they prevent premature convergence by retaining genetic diversity within each separate subpopulation. In addition, each subpopulation can use its own fitness criteria, thereby contributing specific desirable characteristics to the overall evolutionary process. Subpopulations are particularly useful in multi-objective search problems as performed with *Gentropy*. Without subpopulations, problems arise when combining several feature tests together. For example, a texture with a high fitness score for colour matching may not necessarily have a good shape score. Evolution may try to find the highest possible colour score, without improving the shape fitness. Adding these fitness scores together equally — trying to search in various directions at once — may result in the cancelling of their effects. Hence subpopulations are useful in permitting different populations to concentrate on various features, which are then combined elsewhere. This attempts to find as many good examples of different features as possible. One subpopulation can be specialized to search for textures with good colour characteristics, while another subpopulation searches for textures with good shape. These individuals are then exchanged with the main subpopulation, which measures both attributes. Then, some of the exchanged textures should be useful to the search for a texture with good colour and shape.

Subpopulations in *Gentropy* may be given their own particular fitness functions with unique feature tests and weights. Complex topologies may be designed where several subpopulations, each with different but related fitness functions, exchange individuals with each other. Figs. 2, 3, and 4 show subpopulation architectures used in the experiments in Section 6. Each node in the graph denotes a subpopulation, and the arrows indicate the paths of migration for shared individuals. The highlighted subpopulation at the top of the graph is the "main" subpopulation from which solution textures are obtained. The values in each node denote the weight of that particular feature test used in deriving an overall fitness score within that population. For example, in Fig. 2, the subpopulation with "WAV=1.0" indicates that 100% of the fitness score is composed of the wavelet analysis. Each subpopulation evolves separately, and exchanges 5 individuals with another subpopulation at an interval of 10 generations. This provides each subpopulation with a regular injection of fresh, hopefully useful, textures.

Table 8 lists some image processing parameters specific to *Gentropy*. Colour and shape matching precision is adjustable by specifying how many greys, colours and wavelet coefficients are to be kept after quantization or truncation. Recognize that setting these values too low will likely cause very different images to be treated as similar. On the other hand, setting these values too high retains too much colour and shape information, and this increases evaluation time because the fitness function looks for a near exact match to the target image. Although the evaluation size of the generated images is equal to the target size, the final image size may be much larger. The target images are 50 by 50 pixels, which means that 2500 pixels are evaluated by the fitness function for each individual. Of course, the user is always free to use different coordinate ranges when applying evolved textures.

Gentropy allows the use of multiple target images. It is up to the user to specify which features are desirable in each target and the weight they are given. This removes the need for the user to find one target image that contains all desired criteria in the correct proportions. Furthermore, this allows the results of previous runs of the program to be used together as targets, seeding subsequent runs with appealing textures. This should result in a combination of features from the interesting textures in the final image.

6 RESULTS

Some example results of *Gentropy* experiments are in Tables 9, 10, and 11. Table 9 uses a four-subpopulation arrangement shown in Fig. 2, Table 10 uses the seven-subpopulation architecture in Fig. 3, and Table 11 uses the three subpopulations in Fig. 4. Three separate runs are shown for each experiment. The "solution image" is the texture generated by best texture formula discovered in the run, and its output is shown as a 50 by 50 pixel image used during its evaluation. By default, *Gentropy* uses the coordinate range of -1 to 1 about the origin for the x and y positions when evaluating the solution image. Note that evaluating about the origin like this lends bias towards texture formulae which generate patterns symmetric about the origin and x and y axes. The “expanded coordinates” texture is the image produced by the same solution texture formula, but for a coordinate range of –10 to 10 for x and y. This expanded texture was not subjected to feature analyses during evolution.

The fitness scores denote the results of feature tests. The best fitness scores fall within the 60% to 78% range. A near perfect score is not necessary in order to gain the visual effects desired. If there is more than one feature test for the subpopulation, they are given equal weight in this experiment. Although shape matching is the most difficult analysis to undertake, it is possible to compensate for this by increasing its weight in the fitness function: by giving `WAV` a higher priority, evolution will concentrate on shape and thereby allow more colour variation in comparison to the target. Note that the experiment in Table 10 using target image *prim* uses `CDIR` (direct colour matching) in place of the SHIST (smoothness) feature test.

The runs in Tables 10 and 11 use single target images. Their subpopulation architectures are designed to effectively search for various features of the targets, hopefully resulting in solutions that effectively encapsulate all of the features to varying degrees. The four-subpopulation arrangement (Fig. 2) has three subpopulations, each of which pass their best five textures to the main subpopulation every ten generations. These immigrant textures are replaced with five random textures. The main subpopulation has feature tests for colour (`CHISTQ`), shape (`WAV`) and smoothness (`SHIST`), while each of the lower three subpopulations are concerned with only one feature. The seven-subpopulation strategy (Fig. 3) is more complicated. It has three intermediate subpopulations. They have two feature tests and exchange their five best textures with the main subpopulation every ten generations and receive five random ones in return. The lower three subpopulations are concerned with a single feature and exchange textures with two of the intermediate subpopulations.

The resulting evolved textures in Tables 10 and 11 are usually faithful to the target textures images. Note that, in all these results, the intention is not to derive an identical texture as the target, but rather, a texture similar in characteristics. The runs using target image “stripe” almost always evolve textures that are monochromatic stripes. Note that the wavelet analyses does not account for vertical or horizontal aspects of shape, so a few solutions have vertical stripes, unlike the target image. The “splot” runs also generate textures with similar colour and shape features of the target. The results for “prim” are especially impressive, as the solution images often reproduce the same quadrant of primary colours as found in the target. Note that the expanded coordinate view of the solution textures is often very different than the evaluated image. This occurs because the fitness evaluation is only performed on the constrained coordinate system shown in the solution image column. When the texture formula is applied elsewhere in the coordinate system, as in the expanded images, new areas of the texture space are being explored, which were not subject to evaluation during evolution.

Table 11 shows some results from runs in which multiple target textures are used. The subpopulation architecture used for these runs is the 3-subpopulation architecture in Fig. 4. Here, two subpopulations are specialized to evaluate each target image separately. The subpopulation labeled “splot” uses that target image for colour analyses, while the other labeled “stripes” performs a wavelet (shape) analysis on the stripe target. Hence the stripe image will determine shape features, while splot is used for colour characteristics. The main subpopulation applies colour and shape analyses in equal weights, but again applying the colour score relative to splot, and wavelet score to stripes. As a result, it is clear that the solution textures tend to have colours similar to those in the splot image, and stripe shapes akin to the stripes image.

All the experiments were run on a 16-CPU Silicon Graphics Origin server. Each CPU is a 250 MHz MIPS R10000. Typically, all runs for a given experiment were executed concurrently on separate processors. The average processing time for a typical run is about 52 hours.

7 RELATED WORK

Genetic algorithms have been applied elsewhere to the problem of procedural texture generation [6, 7, 8, 9, 10]. Most of these applications use supervised evaluation. The user is presented with a small population of images using a real-time display and asked to choose several for breeding. Thus the user is the fitness function. Karl Sims’ LISP-based system is the inspiration for *Gentropy*’s image representation and function set [7]. Most of the operations in *Gentropy*’s function set are based on the mathematical and image-processing functions described in Sims’ paper. The *Gaia* system has similar mathematical operators to *Gentropy*, as well as the concept of an adjustable coordinate domain in the image evaluation process [8]. *Gaia* stores both a formula and a domain, which determines where the formula is evaluated, for each individual in order to generate an image. Steven Rooke’s system contains functions from the real and complex planes [6]. Evolution is stopped when the user sees a desirable image, and the formula is saved in "digital amber" for seeding later runs of the program. In this way, the artist adds his/her creativity to the process.

The evolutionary art application closest in functionality to *Gentropy* is Aladdin Ibrahim's *Genshade* [10]. Both *Gentropy* and *Genshade* use unsupervised evolution, and both use such analyses as wavelet comparison. *Genshade*'s target language is high-level Renderman shaders, denoted as connected graphs. This is essentially a GP denotation that uses a high-level texture generation library as operators. *Gentropy's* target language is considerably more rudimentary. This points to a philosophical difference between the two research projects. A goal of *Gentropy* is to investigate how effectively a texture can be evolved for arbitrary complex target textures, using a basic set of mathematical and noise operators. On the other hand, it appears as though *Genshade* is most effective at evolving Renderman shaders using target textures generated by Renderman itself. It is unclear whether *Genshade* can effectively evolve Renderman shaders for arbitrary texture targets. Likewise, *Gentropy* would have difficulty reproducing the same quality of solutions for Renderman-generated target textures that *Genshade* enjoys with its Renderman shader language.

Another technical difference between *Gentropy* and *Genshade* is that *Gentropy* supports more robust, customizable user control over texture evaluation. Both systems accommodate multiple target images. However, *Gentropy* permits the user to assign specific suites of feature tests and weights to different target images. The use of user-defined subpopulation topologies, and corresponding evaluation criteria, is also unique to *Gentropy*. This makes *Gentropy* a more robust environment for multi-objective feature-based evolution than *Genshade*.

8 CONCLUSIONS

This research successfully used genetic programming to perform automatic texture generation. Given the difficulty of automating "aesthetic sensibilities", *Gentropy*'s use of a user-selectable set of image analysis fitness functions was shown to be effective. *Gentropy* takes an unsupervised approach, where a fitness function judges the similarity of generated images to one or more targets. *Gentropy* allows any image to be used as a target, including both images generated by the program and images from other sources (scanned, drawn or generated by another program). In some cases, the simplest strategy, with a single subpopulation and feature test, is useful. However, more complex strategies, such as those that use several weighted feature tests, allow the user to better specify his/her preferences in the final texture.

Although several feature tests and subpopulation topologies were tried, many more are possible. For example, more sophisticated edge and texture analysis tests may prove useful. However, more complex analyses will result in computational overhead, which is detrimental to GA efficiency. Alternatives to the use of fitness weights should be investigated, such as the use of dynamic weights that change during evolution, and the averaging of different feature test scores. One method of increasing *Gentropy*'s speed is through the use of multi-threaded GP. Each subpopulation could run on one or more processors and truly evolve in parallel. Should the program be able to run in real-time, more user interaction would be possible.

At the present time, *Gentropy* evaluates a small portion of the texture. Often the rest of the texture shares features with this portion. The fitness function may operate by adjusting the domain, evaluating the texture at different places in two-dimensional space or at different resolutions (dimensions) to see how it matches the target. *Gentropy* could be enhanced by expanding its set of operators, for example, with fractals, complex arithmetic, and perhaps high-level shaders such as used in *Genshade* [10].

Acknowledgments

This research is supported through NSERC grant 138467-1998.

References

[1] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin and S. Worley. *Texturing and Modeling: a Procedural Approach*. Toronto: Academic Press, 1994.

[2] A. Watt and M. Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. New York: ACM Press, 1992.

[3] J. H. Holland. *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press, 1992.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison Wesley, 1989.

[5] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.

[6] S. Rooke. The Genetic-Evolutionary Art Process of Steven Rooke, 1993. <http://www.azstarnet.com/~srooke/process.html> (April 11 2001).

[7] K. Sims. Interactive Evolution of Equations for Procedural Models. *The Visual Computer*, 1993; 9:466-476.

[8] J. L. Abadia. Gaia: Generating New Images using Genetic Algorithms, 1997. <http://www.iua.upf.es/~jlabadia/gaia/> (April 11 2001).

[9] A. Rowbottom. Evolutionary Art and Form. In *Evolutionary Design By Computers*. Ed. Peter Bentley. San Francisco: Morgan Kaufmann, 1999.

[10] A. E. Ibrahim. *Genshade: An Evolutionary Approach to Automatic and Interactive Procedural Texture Generation*. Doctoral thesis, College of Architecture, Texas A&M University, 1998.

[11] D. Whitley. *Proceedings of the Genetic Evolution and Computation Conference* (GECCO 2000). San Francisco: Morgan Kaufmann, 2000.

[12] H. Elias. Cloud Cover, 2000. <http://freespace.virgin.net/hugo.elias/models/m_clouds.htm> (April 11 2001).

[13] J. Koza. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco: Morgan Kaufmann, 1999.

[14] J. R. Smith. *Integrated Spatial and Feature Image Systems: Retrieval, Analysis and Compression*. Doctoral thesis (JRS97), Center for Telecommunications Research, Graduate School of Arts and Sciences, Columbia University, 1997.

[15] J. R. Smith and S.-F. Chang. VisualSEEk: A Fully Automated Content-based Image Query System, 1996. <http://www.ctr.columbia.edu/~jrsmith/html/pubs/acmmm96/acmfin.html> (April 11 2001).

[16] E. Stollnitz, T. Derose, and D. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. San Francisco: Morgan Kaufmann, 1996.

[17] S. Luke. *Patched lil-gp Kernel*. Computer Software. University of Maryland at College Park, 1997. <http://www.cs.umd.edu/users/seanl/gp/patched-gp/> (April 11 2001).

APPENDIX A: SOME EVOLVED TEXTURE FORMULAS

Example 1:

```
RESULT #           TARGET      FITNESS
3 (Table 9)        stripes     0.5738

FORMULA
kaleid(max(max(log(Y),
               turbflow(-0.95,Y,0.86)),
           lum(COLGRAD)),
       forv(cloud(cloud(log(max(sin(X),
                                cos(turbflow(-0.95,Y,0.86)))), X, Y,
                        avg(X, 0.05)),
                  lum(COLGRAD),
                  max(lum(marble(X, X, (-0.68,-0.13,0.83))),
                      diff(Y,
                           avg(min(Y, Y),
                               avg(min(Y, Y),
                                   turbflow(turbflow(-0.95,Y 0.86),Y,0.86))))),
                  pow(noise(pow(X,
                                avg(sin(X)-X, 0.86)-X)-X, -0.36),
                      lum(COLGRAD)))))
```

Example 2:

```
RESULT #           TARGET      FITNESS
1 (Table 9)        splot       0.6735

FORMULA
rgb(turbflow(mod(X, X), X,
             if(cos(sin(sin(sin(Y)))), X, 0.82))
    + not(turbflow(if(0.08,
                      Y/Y,
                      max(0.76, sin(Y))),
                      lum(warpabs(0.69, 0.76, (0.58,-0.04,-0.79))), X)),
    sin(sin(Y))
    + turb(cos(sin(sin(sin(Y)))))
           * noise(not(turbflow(mod(X, sin(Y)),
                                lum(warpabs(0.09, X, (0.58,-0.04,-0.79)))
                                cos(0.52))), 0.69),
           not(0.08)),
    turbflow(mod(X, mod(X, log(Y))),
             min(cos(mod(X,
                         mod(X, Y/Y))), 0.82), 0.64))
```

Example 3:

```
RESULT #           TARGET      FITNESS
1 (Table 10)       prim        0.7036

FORMULA
rgb (diff(X, X)
      - pow((diff(X, Y) * Y),
             max(max(diff(X, X), log(Y)),
                 log(X))
                + diff(X, Y) * Y))
             * (not(pow(diff(X, Y) * Y,
                        log(X)))
                + not(pow(max(diff(X, X),
                              log(Y)) * Y,
                          log(X)))),
      sin(min(pow(diff(min(Y, X), X),
                  wchn(Y, (-0.10,-0.01,0.45))),
              log(avg(0.41, Y)))),
      pow(if(max(log(0.12), log(min(Y, X))),
             mod(min(Y, X),
                 not(wchn(X, (-0.60,-0.20,0.30))
                     * X * diff(X, X))),
             diff(X,Y) * Y + X),
          log(0.12)))
```

```
Initialize population with random individuals. Assign fitness values to all individuals.
•  Repeat until solution found or maximum number of generations processed:
    o  Repeat until new population created:
        ▪  Use fitness proportional selection to select 2 individuals.
        ▪  Apply crossover to generate their offspring.
        ▪  Apply mutation to offspring.
        ▪  Assign fitnesses to offspring and insert into new population.
        ▪  Update generation counter.
```

Fig. 1: Genetic algorithm

Table 1: Texture examples

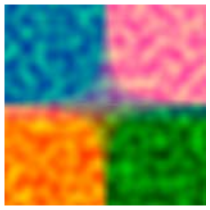
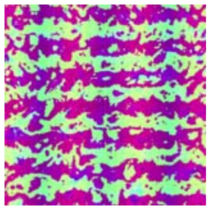
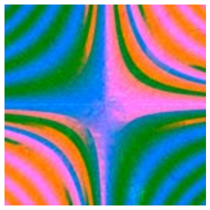
IMAGE	TEXTURE FORMULA
	`rgb(Y*X, noise(Y, -0.71), min(Y, 0.25))`
	`rgb(mod(turbflow(Y,X,X), sin(X)), lum(marble(0.94, -0.78, (-0.46,0.50,-0.63))), turb(chn(COLGRAD), cos(-0.24)))`
	`rgb((wchn(Y*X,COLGRAD)+Y*X)/turbflow(noise(noise(Y, X-Y), cloud(turbflow(X-Y, mod(X,Y), Y/(X-Y)), Y/X, diff(sin(X), cos(noise(Y,Y))), noise(Y,Y))), lum((-0.10,0.34,0.98)), Y/wchn(X,COLGRAD)), noise(noise(Y,Y), cloud(diff(sin(X), cos(noise(Y,Y))), diff(0.94,0.76), mod(X,Y), noise(cos((X-Y)/X), cloud(Y/X, Y/X, sin(X), noise(Y,Y))))), cos(Y*X))`

Table 2: Noise examples

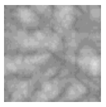
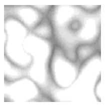
Noise	**Turb**	**Turb Flow**	**Cloud**	**Marble**
				

Table 3: Feature tests

TEST	**DESCRIPTION**
Colour Direct (`CDIR`)	Matches colour similarities pixel by pixel
Colour Histogram (`CHIST`)	Matches colours, position irrelevant
Colour Histogram Quadratic (`CHISTQ`)	Matches similar colours, position irrelevant
Wavelet (`WAV`)	Matches shape using wavelet theory
Smoothness Histogram (`SHIST`)	Matches colour smoothness (contrast), position irrelevant

Table 4: Similarity measure examples

C1	C2	SIM(C1,C2)
Black (0,0,0)	White (1,1,1)	0
Black (0,0,0)	Black (0,0,0)	1
White (1,1,1)	White (1,1,1)	1
Red (1,0,0)	Blue (0,0,1)	0.03
Blue (0,0,1)	Grey (0.5,0.5,0.5)	0.25
Red (1,0,0)	Darker Red (0.8,0,0)	0.78

Table 5: Example quantized, frequency and wavelet images

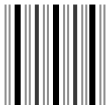
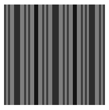
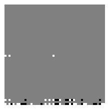
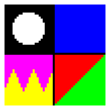
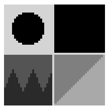
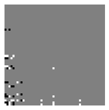
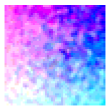
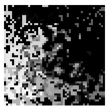
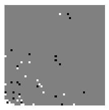
QUANTIZED	**FREQUENCY**	**WAVELET**
		
		
		

Table 6: Sample quantized and smoothness images

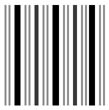
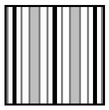
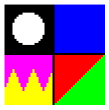
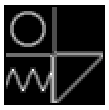
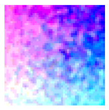
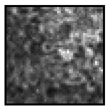
QUANTIZED	**SMOOTHNESS**
	
	
	

Table 7: GP parameters

PARAMETER	VALUE
Maximum nodes	100
Maximum depth	50
Initialization method	half and half
Initial depth	2 to 6
Crossover selection	tournament, size=5
Crossover rate	0.9
Mutation selection	tournament, size=5
Mutation rate	0.1
Population size	5600
Number of subpopulations	1, 3, 4 or 7
Number of exchanges	0, 2, 3 or 9
Subpopulation exchange interval (in generations)	10
Exchange count (number of individuals)	5
Exchange selection methods	best and random
Maximum generations	100

Table 8: *Gentropy*-specific parameters

PARAMETER	VALUE
Threshold (stop when fitness >)	0.99
Number of quantized greys	25
Number of quantized colours	1000
Number of wavelet coefficients used	50
Evaluation/Target width and height	50
Final image width and height	500
Number of target images	1 or 2

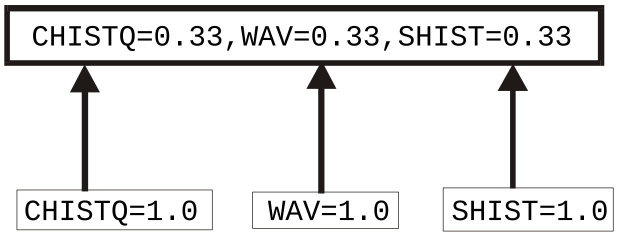


Figure 2: Four subpopulation topology

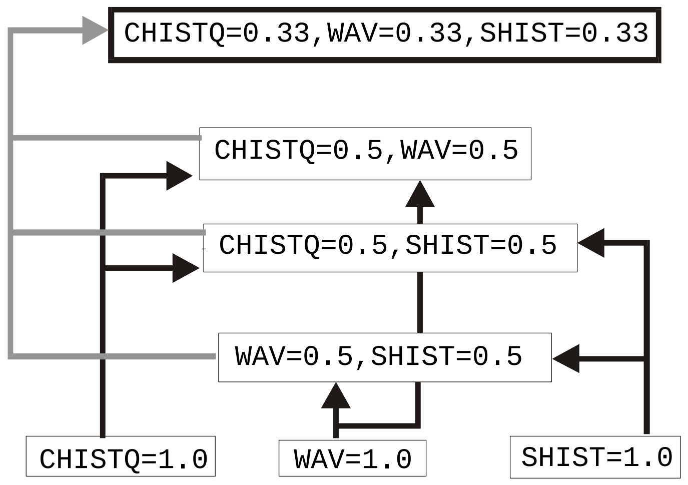


Figure 3: Seven subpopulation topology

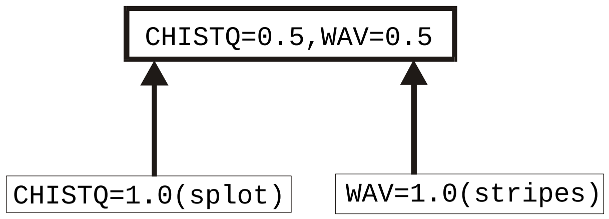


Figure 4: Three subpopulation topology

Table 9: Runs with 4 subpopulations

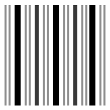
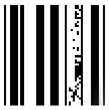
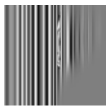
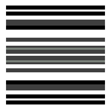
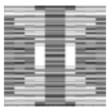
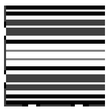
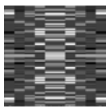
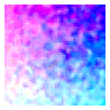
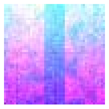
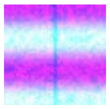
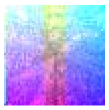
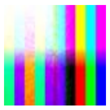
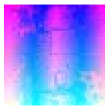
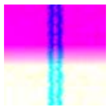
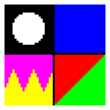
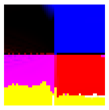
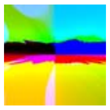
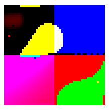
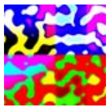
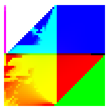
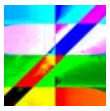
Target	Fitness Score	Solution Image	Expanded Coordinates
	0.6576		
stripes	0.5374		
	0.5738		
	0.6735		
splot	0.6706		
	0.6772		
	0.7260		
prim	0.7304		
	0.7226		

Table 10: Runs with 7 subpopulations

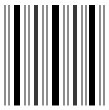
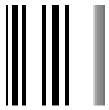
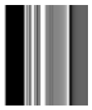
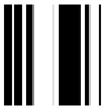
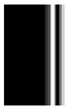
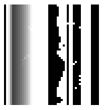
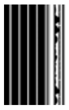
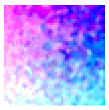
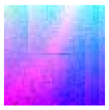
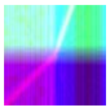
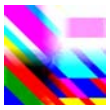
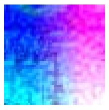
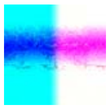
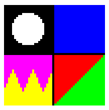
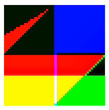
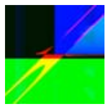
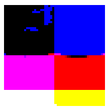
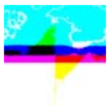
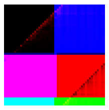
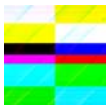
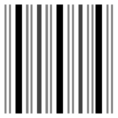
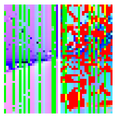
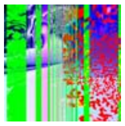
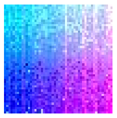
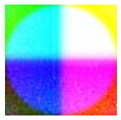
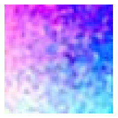
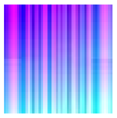
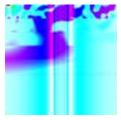
Target	**Fitness Score**	**Solution Image**	**Expanded Coordinates**
	0.5967		
stripes	0.5827		
	0.6241		
	0.6889		
splot	0.6582		
	0.6550		
	0.7036		
prim	0.7033		
	0.7110		

Table 11: Runs with multiple targets

Targets	Fitness Score	Solution Image	Expanded coordinates
	0.7341		
(i) stripes	0.7730		
	0.7286		
(ii) splot			