Procedural 3D Texture Synthesis Using Genetic Programming

Adam Hewgill Brian J. Ross [1]

Brock University, Dept. of Computer Science
St. Catharines, Ontario, Canada L2S 3A1

Abstract

The automatic synthesis of procedural textures for 3D surfaces using genetic programming is investigated. Genetic algorithms employ a search strategy inspired by Darwinian natural evolution. Genetic programming uses genetic algorithms on tree structures, which are interpretable as computer programs or mathematical formulae. We define a texture generation language in the genetic programming system, which is then used to evolve textures having particular characteristics of interest. The texture generation language used here includes operators useful for texture creation, for example, mathematical operators, colour functions and noise functions. In order to be practical for 3D model rendering, the language includes primitives that access surface information for the point being rendered, such as coordinates values, normal vectors, and surface gradients. A variety of experiments successfully generated procedural textures that displayed visual characteristics similar to the target textures used during training.

Key words: procedural textures, evolution, genetic programming

1 INTRODUCTION

Procedural textures are an integral part of contemporary rendering technology [5] [29]. Procedural textures can convincingly model a variety of natural phenomena, such as wood, stone, and terrain, as well as innumerable unnatural effects. Moreover, procedural textures apply rendered results seamlessly over

Email addresses: ahewgill@hotmail.com (Adam Hewgill), bross@cosc.brocku.ca (Brian J. Ross).
URL: http://www.cosc.brocku.ca/∼bross/ (Brian J. Ross).
[1] Corresponding author. Equal authorship implied.

surfaces, without tiling or other repetitive artifacts. This makes them ideal for photorealistic rendering.

One disadvantage of procedural textures is their technical complexity. Mathematical and procedural models of well-known effects, such as stone, wood, and terrain, have been carefully engineered and hand-crafted. Typically, parameterized versions of these pre-defined effects are included in rendering tools, along with suitable interfaces for user-manipulation of the parameters. The derivation of a new procedural texture can be a challenging task. If the user has a visual effect in mind, the derivation of a corresponding procedural texture that produces this effect will depend upon the user's proficiency and experience in mathematical texture modeling. Even for experts, trial-and-error will be the norm. Hence, the invention of computer-based tools which can automatically derive procedural textures is a worthy goal.

This paper investigates the automatic derivation of procedural textures for 3D surfaces using genetic programming technology. Genetic programming is a machine learning paradigm inspired by Darwinian evolution. Populations of computer programs are bred until a program solving some problem of interest is eventually evolved. We use genetic programming to evolve populations of 3D texture formulae. The approach is similar to previous work in automated 2D procedural texture evolution [1][10][15][22][31]. The main enhancement necessary for evolving effective 3D textures is to incorporate surface information into texture formulae. Our texture language includes provision for surface coordinates, surface orientation (normal direction), and surface gradient (normal deviation). As in 2D texture evolution, a target texture is used for training purposes. Surface points on an example model rendered with the target texture are manually sampled. These sample points are then used as a training set by the genetic programming system. The goal is to evolve a 3D texture whose colour rendering on 3D surface components is as similar as possible to that in the target texture.

Section 2 reviews relevant work in texture evolution. An overview of genetic algorithms and genetic programming is given in Section 3. System and experimental details are discussed in Section 4. Some results are given in Section 5. Section 6 gives some concluding discussion.

2 Texture Evolution

Evolutionary algorithms have been widely used in graphics and design [2] [3]. Examples applications range from texture generation (discussed below), graphic design [6] [8], image processing [7] [20] [16], model morphology [26], animation [25] [27] [28], visualization [11], and ray tracing [13]. A number of

commercial products for texture generation have used evolutionary algorithms, for example, Kai's Power Tools and Alien Skin Textureshop. The field has matured to the point that it has attracted artistic critiques [4] [30].

This paper is primarily concerned with procedural texture evolution. A procedural texture may be defined as being one in which individual pixel RGB values are computed by algorithms and/or mathematical formulae. For 2D textures, texture formulae have the form:

$$f : (X, Y) \rightarrow (R, G, B)$$

where (X,Y) are the pixel coordinates, and (R,G,B) is the computed colour. Evolutionary algorithms such as genetic programming [12] are then used to derive an appropriate formula for f. The formula f is equivalent to the genetic material or *genotype* found in natural biology, while the rendered texture is the *phenotype* or realization of the genotype in physical form. The evaluation of a texture is performed by inspection of the set of (R,G,B) values for all pixels coloured, or in other words, the texture phenotype. The actual composition of formula f is not inspected; it is evaluated only in terms of what texture it produces.

Typically, texture evolution systems are supervised, and rely on the user to interactively participate in the evolution process [14] [21] [23] [24]. A population of candidate textures is displayed to the user. He or she selects the texture(s) of interest, to be mutated or recombined into the next set of textures. The user also controls a mutation rate parameter, which determines the degree to which new textures differ from current textures. The mutation rate is usually large at the beginning of a session, and then is gradually reduced when a texture of interest is discovered. In essence, the user becomes an oracle or "fitness evaluation function" for the duration of the search. Although this procedure is a useful interactive tool for texture exploration, a disadvantage is that it is not automated – the user plays the pivotal role in guiding evolution at all stages. The user will suffer fatigue if he or she must evaluate many hundreds of textures.

Unsupervised, fully automated texture evolution has been investigated [1] [10][15][22][31]. In order to automate the search, genetic algorithms must select candidate solutions based upon their fitness scores, which are indications of how well the textures conform to some set of desired visual characteristics. Derive an effective fitness function to rank candidate textures based on their visual characteristics is challenging, since in-depth analyses cannot be used due to their computational burden on the evolutionary search. Typically, a suite of feature tests are used, which rate rudimentary visual characteristics of a texture, such as colour distribution, luminosity, and shape. Although each feature test by itself is a crude measurement of an image's characteristics, they

often produce useful evaluations of textures when combined together. During a session, the feature test scores are compared to the corresponding feature scores of a target texture image. The goal for evolution is to derive a procedural texture whose visual characteristics closely match those of the target texture image, as indicated by the proximity of feature test scores between the target and candidate textures. It is not realistic, nor necessarily desirable, for the genetic algorithm to derive a texture identical to the target texture, but rather, to evolve one that is similar in flavour to the target.

The majority of work in texture evolution has concentrated on 2D textures. This is largely because 2D textures are easily analysed with well-known image processing techniques. Although 2D textures can be extended onto 3D surfaces, the results are usually unsatisfactory, since such textures do not consider morphological features of the surface, lights, or viewer. This results in surface textures which are uniform across the entire 3D surface, and which do not react to surface characteristics such as location or orientation in 3-space.

Evolutionary computation has been used to evolve textures suitable for 3D graphics. Ibrahim's Genshade system evolves Renderman shaders [10]. High-level Renderman shader expressions are denoted as dataflow graphs. Feature tests such as luminosity (brightness), chromaticity (colour), and wavelet analysis (shape) are performed. Evolution can proceed interactive with user guidance, or automatically with feature test evaluation. Multiple target textures can be used as well. The results show effective evolution of Renderman shaders that match target textures generated by Renderman itself. Automated evaluation of textures is performed on 2D renditions of shader. These shaders may be applied to 3D object renderings, which in turn can be presented to the user during interactive evaluation. Lewis also evolves shaders for the Houdini system [14]. Like Genshade, data flow networks of shader primitives are evolved. Evolution is strictly interactive, and the user is shown 3D renditions of the shaders. Neither of these systems explicitly account for 3D morphology during texture generation and analysis, other than rendering the textures onto 3D objects to display to the user during interactive evaluation.

3 Evolutionary Computation

3.1 Genetic algorithms

Evolutionary algorithms are statistical search techniques inspired by Darwinian evolution. One such evolutionary algorithm is the genetic algorithm (GA), invented by John Holland in the 1970's [9]. The GA has proven very successful in finding good solutions for difficult real-world problems [17]. GA

are often very effective for applications that have appropriate structure (although the exact form of this "structure" is not well understood at present). GA's are relatively general algorithms that often require only modest specialization to new applications. This can be contrasted to algorithms that must be engineered from scratch for particular problem specifications.

Figure 1 outlines a basic GA. In a GA, a population of candidate solutions is maintained. Each individual in the population is represented by a chromosome, which encodes some candidate solution to the problem at hand. Chromosomes in basic GA's take the form of bit strings. Fields in these strings will map to various values and parameters of a candidate solution to a problem that is being solved. Furthermore, each individual is assigned a fitness score, which measures the quality of the solution encoded by the individual. A "perfect" fitness score will signal when a solution has been found. Otherwise, the fitness scores can be used to track the relative performance of individuals in the population, as well as the population collectively. Fitness scores also play a key role in the algorithm during the selection of individuals for modification (steps a and b). Fitness values are used to probabilistically select individuals for reproduction, in the sense that fitter individuals will be more likely to be selected than those less fit. This parallels Darwinian evolution's "survival of the fittest", in that fit individuals survive and procreate, while weak individuals die off and become extinct.

A defining characteristic of a genetic algorithm is the means by which new candidate solutions are generated. The most important reproduction operation in a GA is crossover. Crossover is inspired by sexual reproduction in nature, and is the means by which parental traits are inherited by offspring. To perform crossover, two parents are selected based on their fitness, and their chromosomes are randomly split and merged together to form two offspring. For example, consider two individuals with chromosomes $ABCDEFG$ and $tuvwxyz$ respectively. A single-point crossover operation finds some random split point in the 7-gene chromosome. If this point is 3, then the offspring become:

$$\begin{array}{ccc} \underline{ABC}DEFG & & \underline{tuv}DEFG \\ & \Longrightarrow & \\ \underline{tuv}wxyz & & \underline{ABC}wxyz \end{array}$$

The other genetic reproduction operator is mutation. This involves randomly changing some random gene in a chromosome. For example, the individual $ABCDEFG$ may become $ABxDEFG$ by a random mutation of the gene C to x. Typically, mutation plays a lesser role than crossover in GA's, and is used more sparingly.

It is worth reiterating the importance of fitness-based selection in the above discussion. In order for a genetic algorithm to be successful, individuals used for reproduction must be selected based upon their relative fitness or strength

in solving some problem of interest. There are a number of schemes for implementing fitness-based selection. The one used here is the *tournament selection*. Here, a fixed number of individuals are randomly selected from the population. Then the individual with the highest fitness score in this set is retained as a parent for reproduction. This tournament selection is performed once for mutation, and twice for crossover (once for each parent).

3.2 Genetic programming

Genetic programming (GP) is a specialized class of genetic algorithm, in which individuals in the population are interpretable as computer programs [12]. The salient difference between GA's and GP's is that a GP chromosome translates to a computer program, while a GA chromosome translates to a vector of values. One of GP's strengths as a paradigm of evolutionary computation is its applicability to a wide variety of problems. Any application that is best solved via an algorithm or mathematical formula is a good candidate for genetic programming. This differs from a vanilla genetic algorithm that uses bitstrings as chromosomes, which might require sophisticated coding schemes in order to be applied to problems with complex or dynamically changing requirements.

Most GP applications use a tree-based representation of programs. In this representation, internal nodes denote function calls, and nodes below them denote arguments (Figure 2). In order for this representation to be usable, it must be ensured that all trees processed are both syntactically correct, and executable without errors. Executability or *closure* is maintained by ensuring that every function used in the program will execute error-free on any supplied input data. For example, a closed division operator will not invoke an error condition if a division by zero is attempted, but instead will return some predefined value as an answer.

Syntactic correctness of programs is required if programs are to execute in a correct and predictable fashion. Firstly, random trees as created in the initial population must be syntactically sound. To do this, the random tree generator must supply any function in a tree with its required arguments. Reproduction must also maintain syntactic correctness. After two parents are selected for crossover, random nodes are selected in each parents tree (the highlighted subtrees in Figure 3). The subtrees defined at these nodes are then swapped to create the offspring (bottom trees in Figure 3). This maintains syntactic correctness. Mutation involves either replacing a randomly selected subtree with a randomly generated subtree, or replacing a selected terminal node with a random terminal. A result of these tree-based reproduction operators is that chromosomes are variable-sized, and can grow to unbounded proportions if not checked. Therefore, tree-depth limits are typically used in GP runs.

```
1. Initialize the population with random chromosomes.
   Rate the fitness of the individuals in the initial population.
2. Repeat until solution found OR maximum generation reached:
    i) Repeat until new population generated:
       a) $P_1$\% of the time: Select 2 individuals based on their
  fitness, and apply crossover to generate two offspring.
  Add them to new population.
       b) $P_2$\% of the time: Select an individual based on fitness,
  and apply mutation.
  Add result to new population.
    ii) Rate the fitness of individuals in new population.
```

Fig. 1. Example genetic algorithm

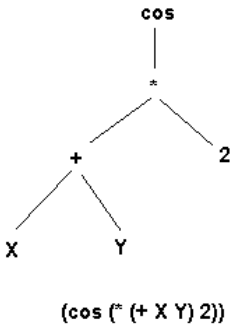

Fig. 2. Example tree and corresponding expression

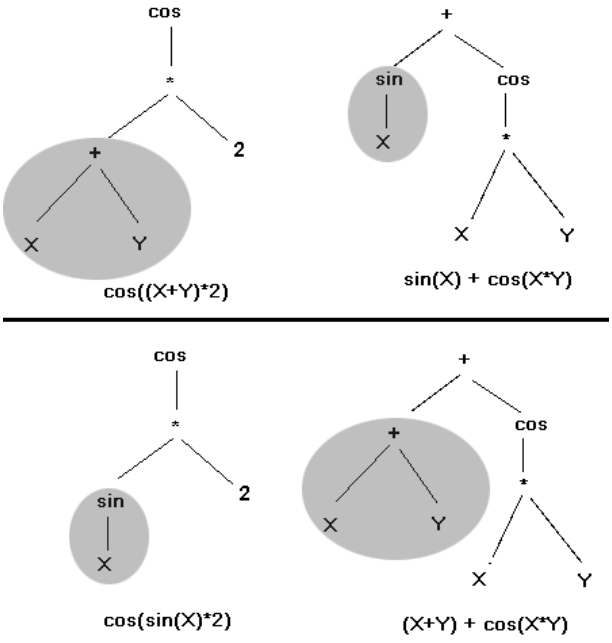

Fig. 3. Tree-based GP crossover

4 Experiment

4.1 Graphics environment

A fundamental characteristic of procedural textures is that they define a colour for all points in the coordinate space. This makes them ideal for ray-tracers, in which textures can be calculated for all visible hit points on model surfaces. On the other hand, they are less useful for rasterized graphics, in which lighting is computed for vertices only, and polygon surface shading is interpolated via Gouraud or Phong shading, mixed perhaps with texture maps. As a consequence, a ray-tracing environment is used in this research. The intention is for the evolved procedural texture formulae to be supplied to a ray-tracer for rendering purposes. However, actual rendering is not performed during texture evolution, but rather, texture formulae will be evaluated for defined sets of example points. Other well-known ray-tracer effects, such as local lighting, reflection, transmission, and shadows, are not considered nor implemented here. They could easily be incorporated with the textures we evolve.

Theoretically, any surface texture is definable by some suitable procedural texture. Naturally, the complexity of the corresponding texture formula or algorithm can vary considerably, depending on the match between texture primitives and the target texture desired. At a minimum, a 3-D texture requires the XYZ coordinates of a surface point in order to calculate that point's RGB value:

$$f : (X, Y, Z) \quad \rightarrow \quad (R, G, B)$$

Such textures are defined entirely by the position of surface points in the coordinate space. For some textures, such as stone and marble, this is satisfactory and adequate. More complex effects, such as the natural rendering of a mountain surface, with a snow-capped peak, rocky cliff, and green base, can also be handled to an extent by these formulae. To do so, the height of the model is influential in producing the appropriate effect as seen in the model's altitude. Often, however, we require textures that conform to additional local shape characteristics of a model. For example, a mountain surface might turn rocky wherever there is a steep cliff, which might occur at various altitudes. The steepness of the cliff might be modeled as a function of the surface normal. In other instances, we might like a texture which changes when there are abrupt discontinuities or creases on a surface. Such features are readily denoted via gradients (the degree to which the surface normal changes).

We used the following model surface information in various experiments:

- X, Y, Z coordinate: This will be the coordinate of the hit point on the model surface as computed during ray-tracing.
- Surface normal: the normal of a hit point.
- Interpolated mesh normal: This is Phong-style normal interpolation on a polygonal mesh. Shared vertex normals are interpolated from surrounding polygons. Then a hit point normal is interpolated from the surrounding vertex normals.
- Surface gradient: This is a scalar value representing the degree to which normals on a local area of a surface are changing. First, the interpolated mesh normal N^i for a hit point is calculated. Then the difference in unsigned magnitude between it and the real surface normal N of the plane on which it resides is found. These differences are then averaged:

$$(|N_x^i - N_x| + |N_y^i - N_y| + |N_z^i - N_z|)/3$$

The above surface parameter definitions are used for generating training data for target textures. An interactive texture sampling editor is used for this purpose. A 3-D model with an applied bitmapped texture is read into the system. The user then manually selects surface points to be used as training data. For all points selected, a record of the above surface information is generated and saved in a table, along with the corresponding RGB colour used to render that sample. This table is saved in a text file, for use by the genetic programming system.

The inclusion of normals and gradients in textures results in considerable differences in texture characteristics. Without normals and gradients, surface detail disappears for all but the simplest models (Figure 4a). Adding normals into texture formulae permits surface detail to arise without the use of lighting (Figure 4b), and gradients introduce an additional dimension of surface detail (Figure 4c). Gradient-based textures create shiny, (false) specular highlights, and are often metallic and iridescent in appearance.

4.2 Evaluation strategy

Once a training set of texture information is sampled by the user, we wish to evolve a procedural texture that exhibits the texture characteristics as encoded in the training set. Given a candidate texture formula, a requirement will be to evaluate how well its rendered texture corresponds to the sampled target texture. Such evaluations take the form of fitness scores, in which a high score means a good match, and hence strong fitness.

A simple evaluation strategy will be used to determine the fitness score of a

texture formula. The surface parameters of each sample point in the training file will be made available to the procedural texture. The texture formula is then interpreted on a training sample. It will make free use of the example surface parameters whenever they are required. Eventually, the formula computes an RGB colour for that point. The distance in RGB space between the actual rendered colour RGB_a and the sampled target colour RGB_s is determined. This distance is then tallied for all k samples in the test set:

$$Fitness = \sum_{i=1}^{k} dist(RGB_a, RGB_s)$$

where

$$dist(RGB_a, RGB_b) = \sqrt{(R_a - R_b)^2 + (B_a - B_b)^2 + (G_a - G_b)^2}.$$

An alternative metric is to use discrete hits and miss frequencies as measured against the input example set. Precise hit tallying in this application is not practical, since it is nearly impossible for any texture to precisely generate the RGB colours residing on a target texture. Our RGB distance relaxes the evaluation criteria, by considering colour proximity, rather than precise colour matches. Nevertheless, tallied hit scores are generated during runs, in order to provide a more intuitive indication of a run's performance. To do this, if a texture's generated colour at a particular sample point is within 0.10 RGB distance of the corresponding target colour, it is counted as a hit. These hit scores are not used by the GP system during fitness evaluation.

4.3 Texture language

The genetic programming system used here is based on Koza's Lisp-based genetic programming model, in which programs take the form of symbolic expressions or *s-expressions* [12]. An s-expression is a list or atom in the Lisp language. An example expression in Lisp notation is:

```
(/ (+ 6 6) 4)
```

This denotes the arithmetic expression (6 + 6)/4. The mapping between s-expressions and trees is straight-forward: the first element in a list is the function (subtree root), its arguments are branches, and atoms are leaves.

The strongly-typed lilGP 1.1 system is used as the GP platform for the experiments [32]. This is a C-based GP system, which implements Koza's tree-based

GP paradigm. It is embellished with strong-typing, which means that expressions can be assigned to return designated data types [18].

Figure 5 shows a Backus-Naur Form grammar of the texture language. Note that expressions are denoted in standard algebraic notation, rather than as Lisp s-expressions. There are two data types in the language – rgb vectors (RGB) and floating point values (F). The RGB operators consist of either the current 3-D coordinate of the point being rendered, the current surface normal of the point being rendered, or an RGB vector $rgbvec$ constructed from the values of three floating point expressions. Note that the RGB-space is modeled with clamped floating-point values between 0.0 and 1.0.

The rest of the grammar defines floating point expressions. Floating point terminals can be basic surface information for the point being rendered, such as one of its coordinate values (X, Y, Z), normal directions (N_x, N_y, N_z) or surface gradient (*diff*). The other floating point terminal, *ephem*, is an ephemeral random constant. This is a constant value that is initialized with a random value between 0.0 and 1.0, but then retains its initialized value throughout its lifetime during the evolution process. In other words, it does *not* denote a call to a random number generator.

The floating point functions include sine, cosine, minimum, maximum, and conventional arithmetic operators. The divide operator was omitted, due to its tendency to create extremely large or small values, which in turn nullify the utility of many formula. The *avg* function finds the average between two expressions, while *lum* finds the luminosity (average channel value) of an RGB vector. The if expression interprets its first floating point argument. If that expression is greater than 0.5 in value, then the value of the second argument is computed and returned as a result. Otherwise the value of the third argument is returned. Finally, $noise$ generates Perlin noise computed to the 4^{th} harmonic [19].

Because the texture language is strongly typed, tree generation and reproduction will maintain legal data typing at all times. During random tree generation, trees must adhere to the typing conventions of each function shown in the grammar in Figure 5. For example, if a function requires an RGB argument, then an RGB function or terminal will be randomly selected. When applying crossover, the data type of the root of each subtree to be swapped must also match.

4.4 GP parameters

Table 1 lists the genetic programming parameters common to all the experiments. All the experiments used a population of size 1000 running for at least

200 generations. This was done for 10 runs per experiment, where each run has a different random number seed. Therefore, each run processes at least 200,000 textures, and each experiment at least 2 million textures. This processing was greatly aided by the use of a 16-CPU Silicon Graphics Origin 2000 server, which permits 10 runs to execute concurrently.

A run begins by first generating a new population with randomly-formed trees. This initial randomized population is created using Koza's ramped half&half tree generation strategy from [12], whose intension is to generate a set of random trees having a variety of sizes and shapes. Half the trees are *grow trees*, in which each randomly generated node has an equal chance of being a function (internal node) or terminal (leaf), up to a maximum depth for the tree. The other trees are *full trees*, where nodes are leaf nodes only when the maximum depth of the tree has been reached. Hence grow trees tend to be asymmetric and smaller in size than the larger, bushier full trees. The idea of *ramping* means that maximum tree sizes are iterated between 5 to 10 levels deep for both grow and full trees. Ramped tree generation proceeds until the population is filled.

During evolution, crossover is used to create 90% of a new population, and mutation is used for the remaining 10%. Trees can never exceed a maximum depth of 17 levels. If they do, the reproduction operator is tried again with new parents. Parent(s) are selected using a tournament selection with a tournament size of 5. This tournament size creates a fairly strong selection pressure, compared to weaker tournaments of, say size 2.

5 Results

Details of specific experiments are discussed in the subsections to follow. A summary of performance for the experiments is given in Table 2. Some parameters different from the common ones in Table 1, such as total generations and training set size, are included here. The performance results (fitness and hits) are averaged over the 10 runs done per experiment. In the summary, the population fitness is the mean fitness of the final population. Similarly, the best fitness refers to the best solution in the run. Because raw fitness scores as described in Section 4.2 correlate with the size of the training set, the fitness values reported here are normalized with respect to the training set size, in order to make better comparisons between the experiments. The RGB space ranges from 0.0 to 1.0 on each channel, and the maximum RGB distance possible is between 2 opposite corners of the RGB cube: $\sqrt{3} = 1.73$. The fitness

values in the table are computed with the following:

$$fitness = 1 - \frac{\sum_{i=1}^{N} dist_i}{N\sqrt{3}}$$

where $dist_i$ is the RGB distance for each example, and N is the total number of examples. The result of this is that a texture that precisely matches a target texture will have a fitness score of 100%, while the worst score obtained would be 0%. The hits column refers to the percentage of training points having an RGB distance value less than 0.10. It is important to realize, however, that even after this normalization of scores, experiments will differ widely with respect to the complexity of training samples due to the combined interaction of surface characteristics, colours, and example set size.

Examining Table 2, the primary colour cube and normal gradient experiments were the most successful in terms of performance scores. The clothing experiment's fitness value is close to the hit boundary of 0.10, showing that that experiment was fairly successful in ensuring that lots of computed texture colours were close to their training values. As expected, the best fitness is always better than the mean population fitness. The difference between the best fitness and population fitness is typically small, which implies that there is a good deal of convergence in the population, and hence a multiplicity of similarly performing textures. The tournament selection we used, with its tournament size of 5, is known for causing high convergence. In addition, we permit fairly large texture formula (maximum tree depth of 17), which also promotes convergence. Another indication of convergence will be seen later when looking at a performance graph of a run.

In the remainder of the section, alternate models are rendered with solution textures by rescaling their coordinate extents to be those of the original training model. These alternate models are equivalent to *testing performance* indicators. We did not use a quantitative measurement of testing performance in our experiments. To do so, a texture would be scored against examples which were not used during training. Since procedural textures can be applied to a wide variety of models for which the notion of testing samples does not really apply, we opt to subjectively evaluate the application of textures onto new models by inspecting the rendered result.

5.1 Primary Colour Cube

This first example uses a simple training set (Figure 6). Six sample points are selected from centers of sides of a cube. These points are assigned the colours red, green, blue, grey, white, and yellow. The gradient and noise primitives

were removed from the texture language for this run. As is done in all experiments, the fitness function evaluates how closely a texture formula renders these points with colours near their sampled target colours. This does not imply that the entire side of the cube will necessarily be that colour, but only the single center point sampled for training. This indeed can be seen in the cube images in the figure. All the solutions obtained, including the 3 shown, performed with nearly 100% accuracy on the training set.

Figure 7 shows the evolved texture formula used in column 3 of Figure 6. The size and complexity of this formula is a fairly typical result from genetic programming. There is room for simplification in the formula. For example, the term *(min 0.36341 0.13969)* found 15 times in the formula can be replaced with the constant *0.13969*. Such extraneous expressions are called *intron code* or *program bloat*, and have no deleterious effect on the quality of a solution, other than making interpretation slower than necessary.

5.2 Terrain Texture

Figure 8 shows some results of evolved textures suitable for a mountainous landscape. The training model (top row) consists of 97 points sampled from a polygonal mesh mountain. The intension is for the mountain to have a white snowcap, grey cliffs, and a green base. The three solutions shown have training hit scores of approximately 33%. As shown in Figure 2, the terrain textures had some problems getting high training scores. This could be due to the fairly noisy nature of the training set.

5.3 Clothing Texture

Next, a female figure is used as a surface for training (Figure 9). A total of 1327 sampled points are used. The idea is that the procedural texture will colour the woman by giving her blonde hair, a green tank-top shirt, purple pants, black shoes, and pink face and arms. This is a complex task for a mathematical texture formula, given that the training file does not have high-level information about model components (arms, legs,...), but only the coordinate, normal, and gradient information of each sample point. The results after 600 generations are shown in Figure 10. The hit scores for these solutions are (clockwise from top left) 63%, 38%, 74% and 36%. The high training score of 74% for this last solution can be seen in the image (row 2, column 2), as the model is rendered closely to the training specification. The terrain images is rendered with this best texture.

Figure 11 shows the fitness progress of the run that evolved the texture in row

2, column 2 of Figure 10. The fitness of the best individual and population average shows steady progress through the entire 600 generations of the run. This shows that 600 generations is not excessive. In fact, additional generations would likely give further progress, based on the trend in this graph. This performance graph is also somewhat unusual for many applications, as it is odd to see a steady increase in both population and best performances for so long. The reason this is occurring here is due to the high degree of convergence in the population. As is seen in Table 2, the population and best scores are very close, and indicates that the population is highly dominated by textures that are very similar to the best solution. As discussed earlier, this is a product of the high selective pressure of the tournament selection scheme (tournament size 5), and the large texture formula trees permitted (depth 17).

5.4 Normal and Gradient Specialized Texture

Figure 12 shows the evolution of a texture that captures a particular surface orientation and shape for a model. The training model (column 1) consists of 5 steep hills on a flat surface. The tip of each hill is to be coloured red. The tip area has a high gradient, and variable normal. The rest of the hill below the tip is green. The flat base from which the hills rise is white. A total of 115 training points were used. The shown solutions (in order) have hit scores of 91%, 90%, 96% and 97%. Furthermore, the second solution has an average (normalized) distance score of 95.7%, which is less fit than the other solutions. This is evident in the rendered image, as the purple colour is further away than the desired red. The last solution's normalized distance of 97.8% is closer to the training specification.

5.5 Miscellaneous Results

Figure 13 shows an experiment in which a terrain-oriented training set similar to that in Section 5.2 is used. After completing the sampling of the training set, however, we swapped the blue channel and gradient values. It is difficult to intuit the effect of this new interpretation of data. The runs yielded some unexpected and interesting textures, and models tended to appear made of transluscent minerals. This shows how automated evolutionary algorithms can be used to discover interesting effects.

The final experiment uses the following training set. A total of 182 sample points are used upon a cube (Figure 14). The cube is blue, with 3 coloured bands (yellow, purple, tan) wrapping around the cube in the middle of each face on each of the XY, XZ and YZ planes. The 8 vertices have shared normals, interpolated from the faces surrounding each vertex. Hence the normals do not

indicate the planar normals of each face. Since this is a particularly challenging texture to evolve, we let the system run for a total of 600 generations. Some results are shown in Figure 15. They have hit scores of (in order) 47%, 40% and 40%. The results of training (column 1) are unexceptional. This is likely due to descriptive shortcomings in the texture language, which is clearly lacking in primitives that are adaptable to this particular target texture. Nevertheless, the results of applying the solution texture to other models are interesting, especially given that the overall colours used in the training set were prevalent.

6 Conclusion

This paper reports a first investigation into the suitability of using evolutionary computation to automatically synthesize 3-D procedural textures. The results of these experiments are positive and promising. Our scoring method – positive example matching – is very simple. This makes the results even more impressive, since 3-D procedural texture spaces can be complex and high–dimensional. The randomness and chaos that is an inherent part of evolutionary computation is a distinct advantage in texture generation, as it lends an element of invention and surprise to the texture synthesis process. Admittedly, some of the target textures used here are simple enough that handwritten solutions could be derived. The important point, however, is that no manual derivation of procedural textures was necessary – genetic programming obtained results automatically from training data. Obtaining samples for training is a much simpler task than deriving texture formulae.

This paper can be compared to a few other investigations into 3D texture evolution. Ibrahim's Genshade system evolves Renderman shaders [10]. Genshade can be used in either supervised or unsupervised modes. In unsupervised mode, a set of rudimentary image analysis evaluations are performed on a 2D rendition of the texture. Resulting shaders can be applied to 3D surfaces at any time. Unlike this paper, however, 3D surface effects are not considered during evaluation. In supervised mode, the user may be presented with renderings of textures on 3D objects. No automated evaluation of the 3D rendered textures is done. Work by Lewis evolves texture shaders for the Houdini animation system [14]. These shaders are intended for rendering 3D objects. Similar to Genshade's supervised mode, no automated analysis nor effects of surface characteristics are considered.

There are a number of directions for improving the evolution of 3D textures. Firstly, the parameters of the experiments could be altered to promote more optimal performance. For example, most runs resulted in bloated solutions. This expression bloat contributes to convergence. The bloat terms not only have the effect of protecting expressions from harmful alteration, but also al-

terations that might improve the fitness. Secondly, the texture language used in this paper is very rudimentary, having only a basic set of arithmetic and RGB operators, along with a basic noise primitive. More complex textures will arise with a language that has higher-level texture generating primitives. For example, the use of Renderman shaders as done in Genshade would immediately result in more complex texture results [10]. The language could also be supplemented with additional information about the model structure, for example, the hierarchical composition of models. This would permit the texture formulae to incorporate which portion of the model hierarchy a texture is to be rendered upon, and would result in textures that are tuned more intimately to the model's structure.

More work needs to address the issue of fitness evaluation. Although the positive example matching approach used here is simple to implement, and often yields acceptable results, it could use improvement. A fundamental problem with example scoring is that the results are strongly dependent on the fairness of the example set. For example, if the majority of examples are blue, then fitness evaluation will naturally be biased towards formulae yielding blue colours. To overcome this bias, the user needs to balance the example set in a way that different features are adequately populated within the set.

Another problem with example scoring is that positive example matching is unsuitable for evolving noisy textures. The nature of noise is that a target pixel satisfying various surface characteristics may have substantially variable colour depending on the chaotic nature of the noisy texture space itself. Unfortunately, the use of positive example matching is decidedly prejudiced against noise, because of the chaotic colouring that occurs when noise primitives are present in a texture formula. A noisy area of texture space may generate colours within some distribution. Fitness scoring, however, requires a strict match with the colour designated within the example. This means that the existence of a noise function will detract from fitness, and so evolution will prefer non-noisy formulae with more deterministic rendering behaviour. Our experience was that noise primitives almost always disappeared from the population during early generations. An alternative fitness evaluation technique that would not be biased against noise would be to incorporate probabilistic matching of candidate and target examples. For example, one could ascribe to an example point a probabilistic colour distribution. This would presume that a number of examples would be included in this probabilistic colour set, so that the resulting texture's colour distribution would closely align itself to the example set.

Since genetic algorithms are inherently statistical in nature, there is still a strong element of discovery in unsupervised evolution of textures. Different runs may produce quite unique and unexpected solutions. Often, the most pleasing aesthetic results do not necessarily correlate with the strongest fit-

ness scores. Errors inherent in non-optimal solutions are often intriguing from artistic points of view.

The value of unsupervised texture evolution is that acceptable solutions can be obtained with no need for user involvement. This contrasts to supervised evolution, in which the user is solely responsible for the nature of solutions obtained. Perhaps the ideal system will be a compromise between these two extremes. Until research into aesthetic modeling is more rigorously developed, we feel that artistic applications such as this one can always benefit from the aesthetic sensibilities and artistic intervention of a human being. It is interesting to consider the implementation of semi-automated texture synthesis systems that incorporate a dynamic interplay of automated and user-directed evaluation. Such tools might turn out to be the most practical.

Acknowledgement: This research is supported by NSERC Operating Grant 138467-1998.

References

[1] S. Baluja, D. Pomerleau, and T. Jochem. Towards Automated Artificial Evolution for Computer-generated Images. *Connection Science*, 6(2/3):325–354, 1994.

[2] P. Bentley. *Evolutionary Design by Computers*. Morgan Kaufmann, 1999.

[3] P. Bentley and D.W. Corne. *Creative Evolutionary Systems*. Morgan Kaufmann, 2002.

[4] A. Dorin. Aesthetic Fitness and Artificial Evolution for the Selection of Imagery fgrom the Mythical Infinite Library. In *Advances in Artificial Life – Proc. 6th European Conference on Artificial Life*. Springer-Verlag, 2001.

[5] D.S. Ebert, F.K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: a Procedural Approach*. Academic Press, 2 edition, 1998.

[6] R. Gatarski. Evolutionary Banners: An Experiment With Automated Advertising Design. In *Proc. COTIM-99*, 1999.

[7] J. Graf and W. Banzhaf. Interactive Evolution of Images. In *Proc. Intl. Conf. on Evolutionary Programming*, pages 53–65, 1995.

[8] J.I. Hemert and A.E. Eiben. Mondrian Art by Evolution. In *Proc. Dutch/Belgian Conf. on Artificial Intelligence (BNAIC '99)*, 1999.

[9] J.H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[10] A.E.M. Ibrahim. *GenShade: an Evolutionary Approach to Automatic and Interactive Procedural Texture Generation*. PhD thesis, Texas A&M University, December 1998.

[11] C. Jacob. *Illustrating evolutionary computation with Mathematica*. Morgan Kaufmann, 2001.

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

[13] B. Lange and M. Beyer. Rayvolution: An Evolutionary Ray Tracing Algorithm. In G. Sakes, P.Shirley, and S. Muller, editors, *Photorealistic Rendering Techniques*, pages 136–144. Springer-Verlag, 1995.

[14] M. Lewis. Aesthetic Evolutionary Design with Data Flow Networks. In *Proc. Generative Art 2000*, 2000.

[15] P. Machado and A. Cardoso. All the Truth About NEvAr. *Applied Intelligence*, 16(2):101–118, 2002.

[16] P. Machado, A. Dias, N. Duarte, and A. Cardoso. Giving Colour to Images. In *Proc. AISB 2002 Symposium on AI and Creativity in the Arts*, 2002.

[17] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[18] D.J. Montana. Strongly Typed Genetic Programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[19] K. Perlin. An Image Synthesizer. *Computer Graphics*, 19(3), 1985.

[20] R. Poli. Evolution of Graph-like Programs with Parallel Distributed Genetic Programming. In Thomas Back, editor, *Proc. 7th Intl. Conf. on Genetic Algorithms*, pages 346–353. Morgan Kaufmann, 1997.

[21] S. Rooke. Eons of Genetically Evolved Algorithmic Images. In P.J. Bentley and D.W. Corne, editors, *Creative Evolutionary Systems*, pages 330–365. Morgan Kaufmann, 2002.

[22] B.J. Ross and H. Zhu. Procedural Texture Evolution Using Multiobjective Optimization. Technical Report CS-02-18, Brock University, Dept. of Computer Science, July 2002.

[23] A. Rowbottom. Evolutionary Art and Form. In P.J. Bentley, editor, *Evolutionary Design by Computers*, pages 330–365. Morgan Kaufmann, 1999.

[24] K. Sims. Interactive evolution of equations for procedural models. *The Visual Computer*, 9:466–476, 1993.

[25] K. Sims. Evolving Virtual Creatures. In *SIGGRAPH 94*, pages 15–22, 1994.

[26] S. Todd and W. Latham. *Evolutionary Art and Computers*. Academic Press, 1992.

[27] J. Ventrella. Explorations in the Emergence of Morphology and Locomotion Behavior in Animated Characters. In R. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 436–441. MIT Press, 1994.

[28] J. Ventrella. Disney meets Darwin - the evolution of funny animated figures. In *Computer Animation 95*, pages 35–43. IEEE Press, 1995.

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

[30] M. Whitelaw. Breeding Aesthetic Objects: Art and Artificial Evolution. In P. Bentley and D.W. Corne, editors, *Creative Evolutionary Systems*, pages 129–145. Morgan Kaufmann, 2002.

[31] A.L. Wiens and B.J. Ross. Gentropy: Evolutionary 2D Texture Generation. *Computers and Graphics Journal*, 26(1):75–88, February 2002.

[32] D. Zongker and B. Punch. *lil-gp 1.0 User's Manual*. Dept. of Computer Science, Michigan State University, 1995.

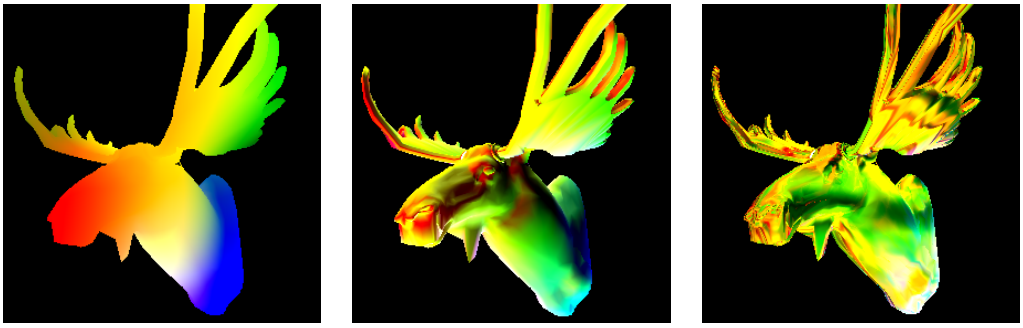

Fig. 4. Texture language effects. (a) Basic. (b) Normals. (c) Normals and gradients.

$$
\begin{aligned}
RGB &::= RGB_{term} \mid RGB_{func} \\
RGB_{term} &::= v_{xyz} \mid n_{xyz} \\
RGB_{func} &::= rgbvec(F, F, F) \\
F &::= F_{term} \mid F_{func} \\
F_{term} &::= x \mid y \mid z \mid n_x \mid n_y \mid n_z \mid diff \mid ephem \\
F_{func} &::= sin(F) \mid cos(F) \mid F + F \mid F - F \mid F * F \mid avg(F, F) \mid lum(RGB) \\
&\quad \mid max(F, F) \mid min(F, F) \mid if(F, F, F) \mid noise(RGB)
\end{aligned}
$$

Fig. 5. Texture Language Definition

Parameter	*Value*
Population size	1000
Generations	200 (sometimes more)
Runs/experiment	10
Initialization	ramped half&half
Initial ramped tree depth	5 to 10
Max. tree depth	17
Crossover rate	0.9
Mutation rate	0.1
Selection scheme	tournament (size 5)

Table 1
Genetic Programming Parameters

Experiment	Gen.	Training set size	Population fitness	Best fitness	Best hits
Primary cube (Fig. 6)	300	6	99.3%	100.0%	100%
Terrain (Fig. 8)	200	97	84.0%	84.3%	29%
Clothing (Fig. 10)	600	1327	92.5%	92.9%	46%
Normal gradient (Fig. 12)	200	115	96.7%	97.3%	88%
Channel swap (Fig. 13)	200	97	90.4%	90.7%	37%
Banded cube (Fig. 15)	600	182	87.6%	88.0%	43%

Table 2
Experiment summary. Results averaged over 10 runs.

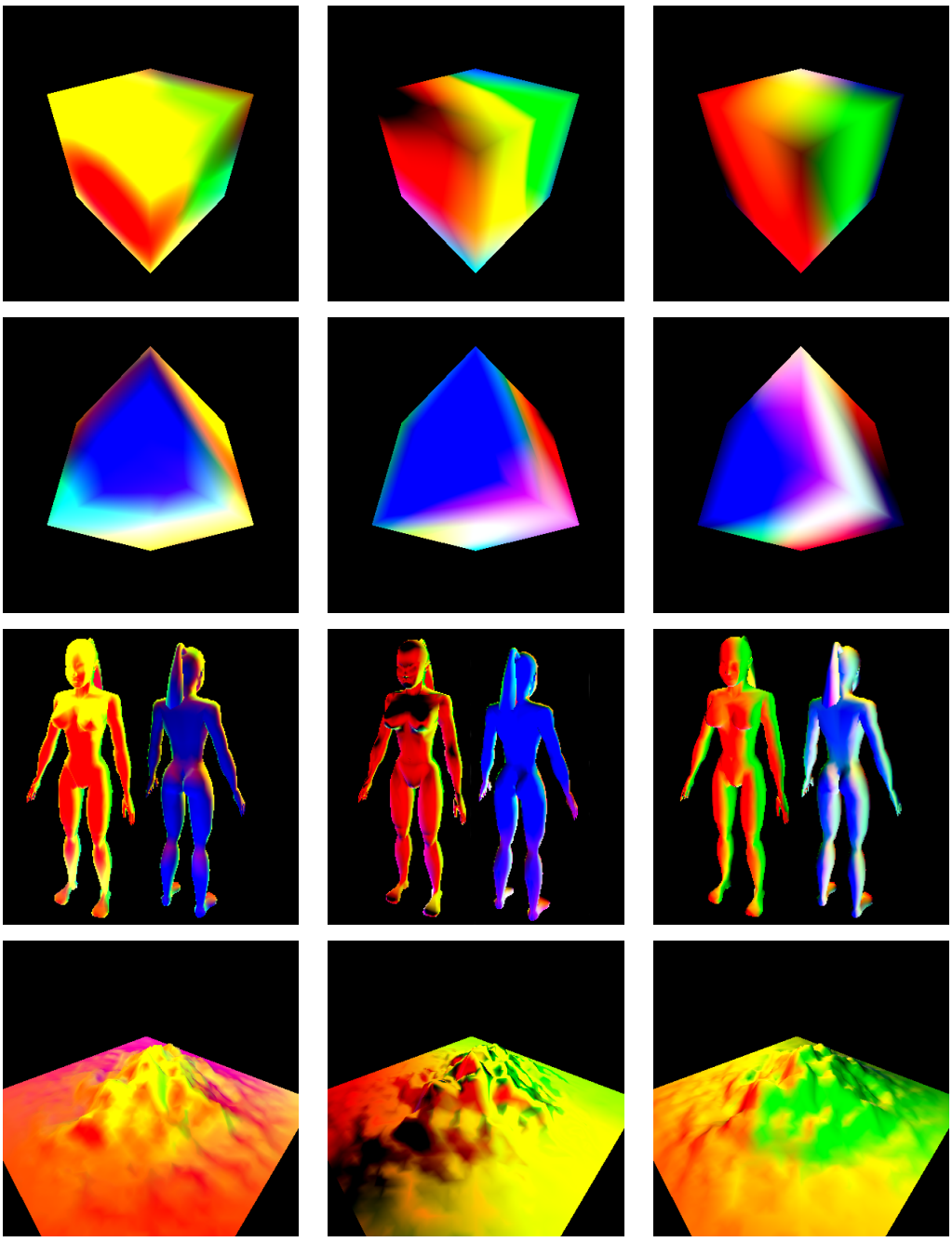

Fig. 6. Primary colour cube. Each column is a separate solution. The rows show training front view, training back view, woman front and back views, and terrain.

```
(rgb (+ (cos (+ (avg Z NZ) (- (* (- (cos (if (cos (- Z Y)) 0.28686 Y)) (min 0.36341 0.13969)) (min (+ (- Z NZ)
(* NX X)) (- Z NX))) NX))) (* (- (cos (if (- (cos (if (min 0.36341 0.13969) (- (min 0.36341 0.13969) (avg Z NZ))
(if (min NY 0.13969) (min (+ (avg Z NZ) (lum NXYZ)) (+ Z (- (min NY 0.13969) NZ))) Y))) 0.41939) (+ NZ
Z) Y)) (- Z NX)) (+ (* (avg (+ (- Z NX) (+ NY X)) (avg (+ (+ NZ (+ NY X)) (avg (+ (+ NZ NX) (+ NY X))
(- (cos (avg (+ (+ Z (* NX X)) X) NX)) (- (cos (min NY 0.13969)) (- Z NX))))) (- (cos (+ (avg Z NZ) (+ Y (-
Z NX)))) (min 0.36341 0.13969)))) (- Z NZ)) (- NZ NX)))) (cos (+ (min (avg (+ (min (+ Z (min (+ (avg Z NZ)
(lum NXYZ)) (cos (+ (* (avg (+ (+ NY X) (avg Z NZ)) (- Z NZ)) (- (cos NZ) NY)) (- NZ NX))))) (avg (+ (+ Z
(* NX X)) X) NX)) (+ NY X)) (- (cos (min (avg (+ NY (* NX X)) (avg (+ NY X) (- (cos (avg (+ NY (+ (+ Z
(* NX X)) X)) (- (cos (- Z Y)) (min 0.36341 0.13969)))) (- (cos (min 0.36341 0.13969)) (min 0.36341 0.13969)))))
(* NX X))) (min 0.36341 0.13969))) (min (avg (+ (+ NZ NX) (+ NY X)) (- (cos (min (avg (+ (+ NZ NX) (-
(- (cos NZ) NY) 0.41939)) (- (cos (avg (- Z NX) NZ)) (- (cos (if (lum NXYZ) 0.28686 Y)) (- Z NX)))) (- (- (*
(- (cos (avg (- NZ NZ) NZ)) (- (cos (min 0.36341 0.13969)) (min 0.36341 0.13969))) (- Z NZ)) NX) NZ))) (min
0.36341 0.13969))) (* (avg (+ (* NX X) X) (avg (+ (+ Z (* NX X)) X) NX)) X))) (+ Z (- (+ NZ (+ Z (* NX
X))) NX)))) (* (- (avg (- (cos (+ (* (avg (+ (+ NZ NX) (+ NY X)) (- (cos (- Z Y)) (min 0.36341 0.13969))) (-
(- NZ NX) NZ)) (- NZ NX))) (+ NY X)) (sin X)) (min (+ (avg Z NZ) (lum NXYZ)) (cos (cos (- (cos (cos (+ (*
(avg (+ (+ NY X) (avg Z NZ)) (- Z NZ)) (+ Z (* (- Z NX) X))) (- NZ (+ Z (min (+ (avg Z NZ) (lum NXYZ)) (-
(cos (* NX X)) (min 0.36341 0.13969)))))))) (cos (- (+ (+ NZ NZ) (avg (+ (+ NZ NX) (+ NY X)) (- (cos (avg
(+ (+ Z (* NX X)) X) NX)) (- (cos (min NY 0.13969)) (sin X))))) (min 0.36341 0.13969))))))))) (cos (avg (- (+
(- Z NZ) (* NX X)) (+ (avg Z NZ) (lum NXYZ))) (avg (max (+ (avg Z NZ) (+ Z (- NZ NX))) (min (+ (* (avg
(+ (+ NZ NX) (+ NY X)) (- (cos (- Z Y)) (min 0.36341 0.13969))) (- (- NZ NX) NZ)) (- NZ NX)) (- (cos NZ)
NY))) (- (* (- (cos NZ) (- Z NX)) (min (sin X) (- Z NZ))) NX))))))
```

Fig. 7. Example solution for primary colour cube (col. 3 of Fig. 6).

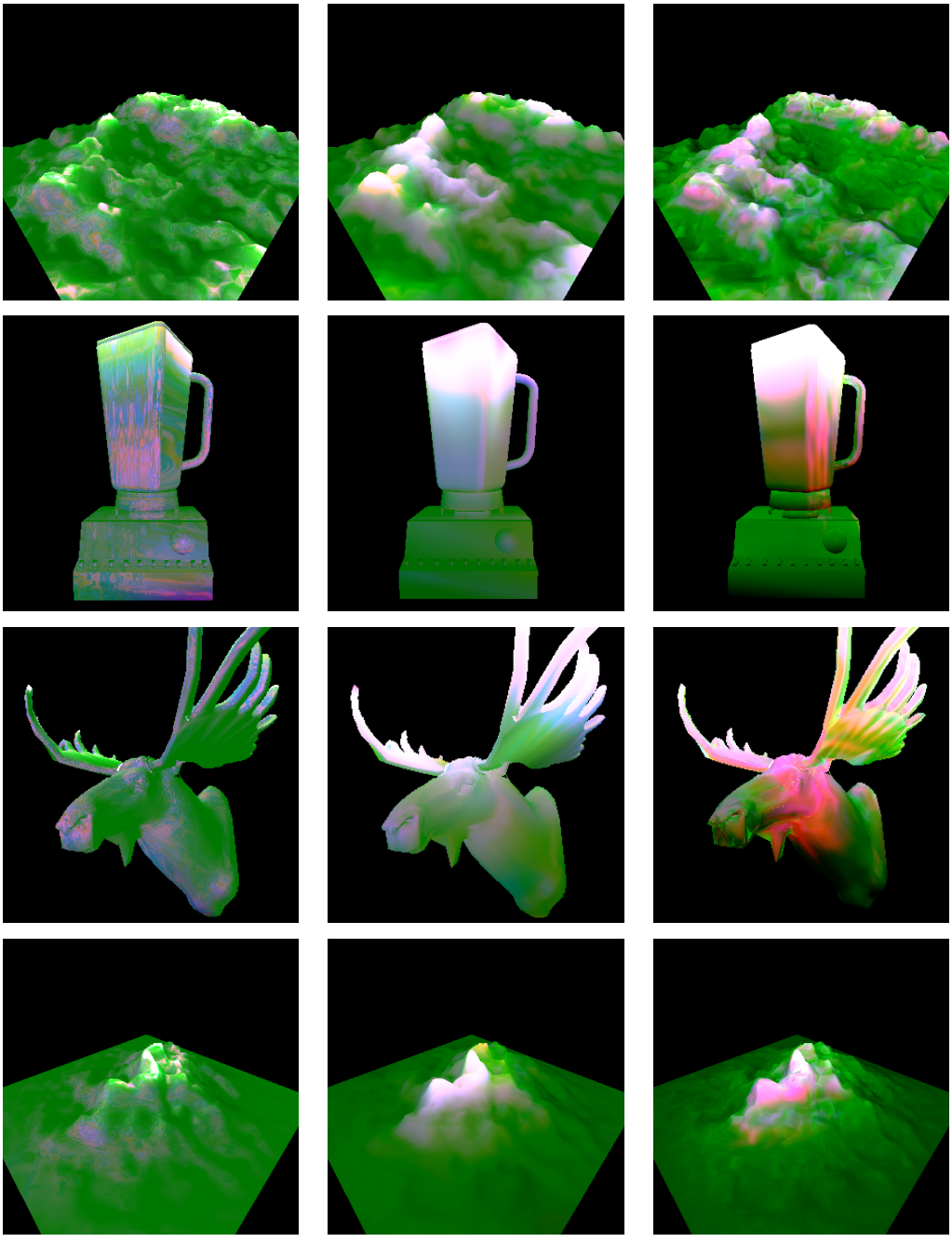

Fig. 8. Terrain texture. Each column is a solution.

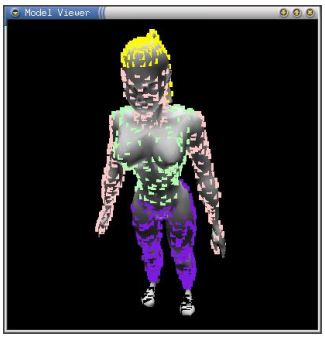

Fig. 9. Woman's clothing training points.

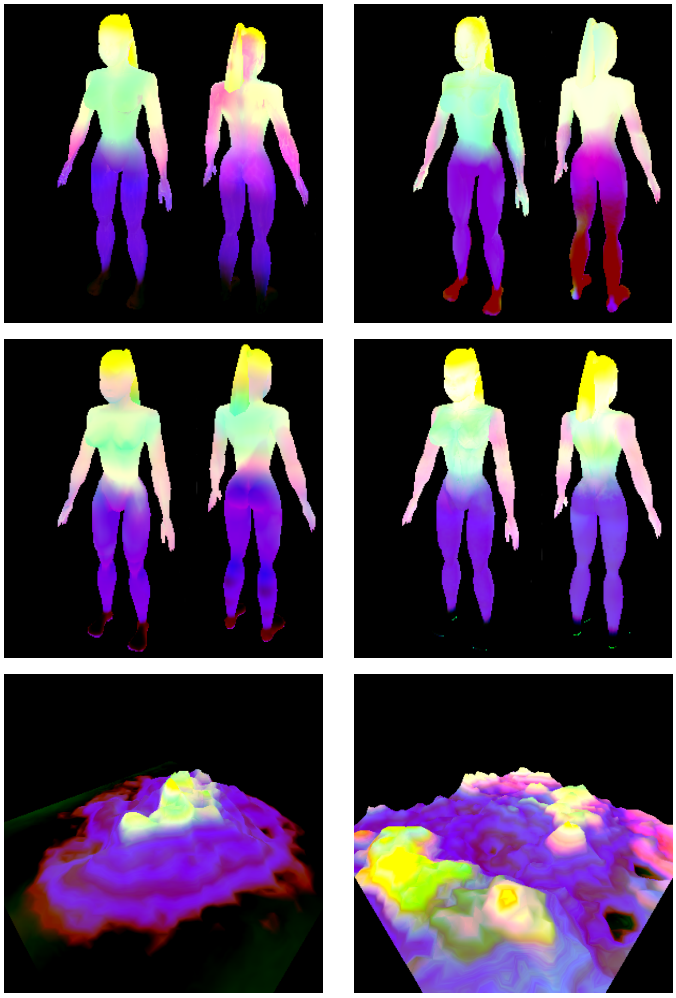

Fig. 10. Woman's clothing texture. Four solutions. Terrain uses bottom right clothing texture.

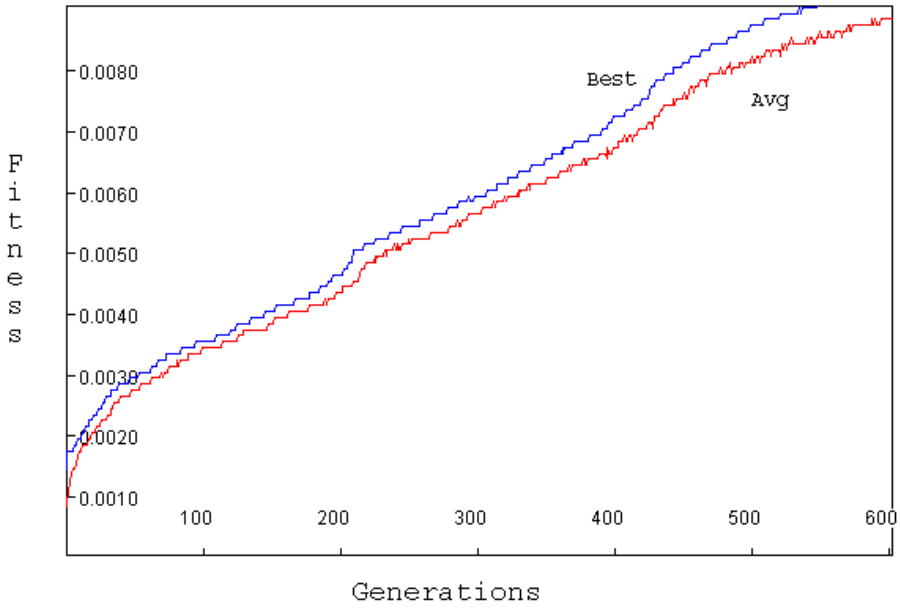

Fig. 11. Fitness progress for clothing texture run.

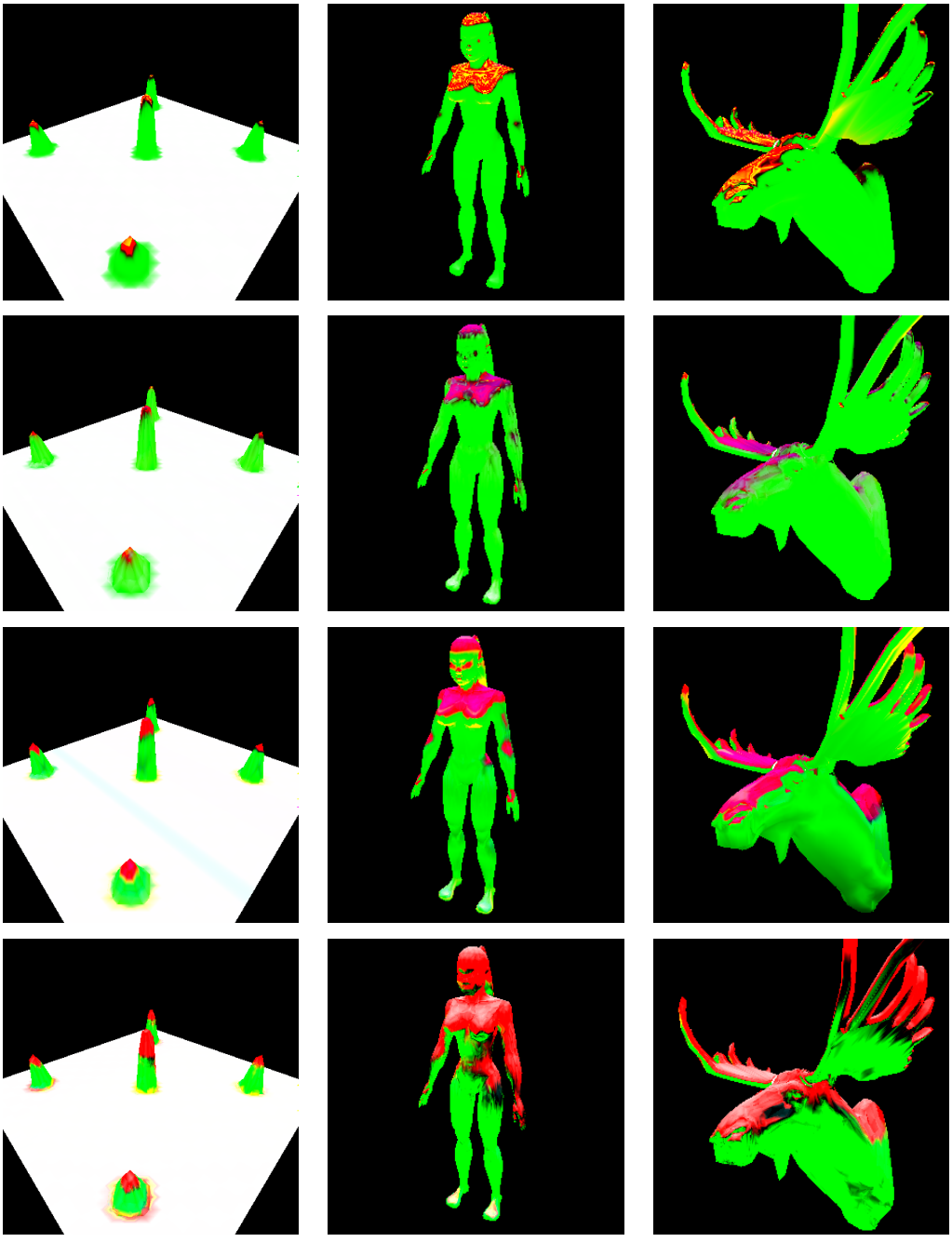

Fig. 12. Normal and gradient specialized texture. Each row is a solution.

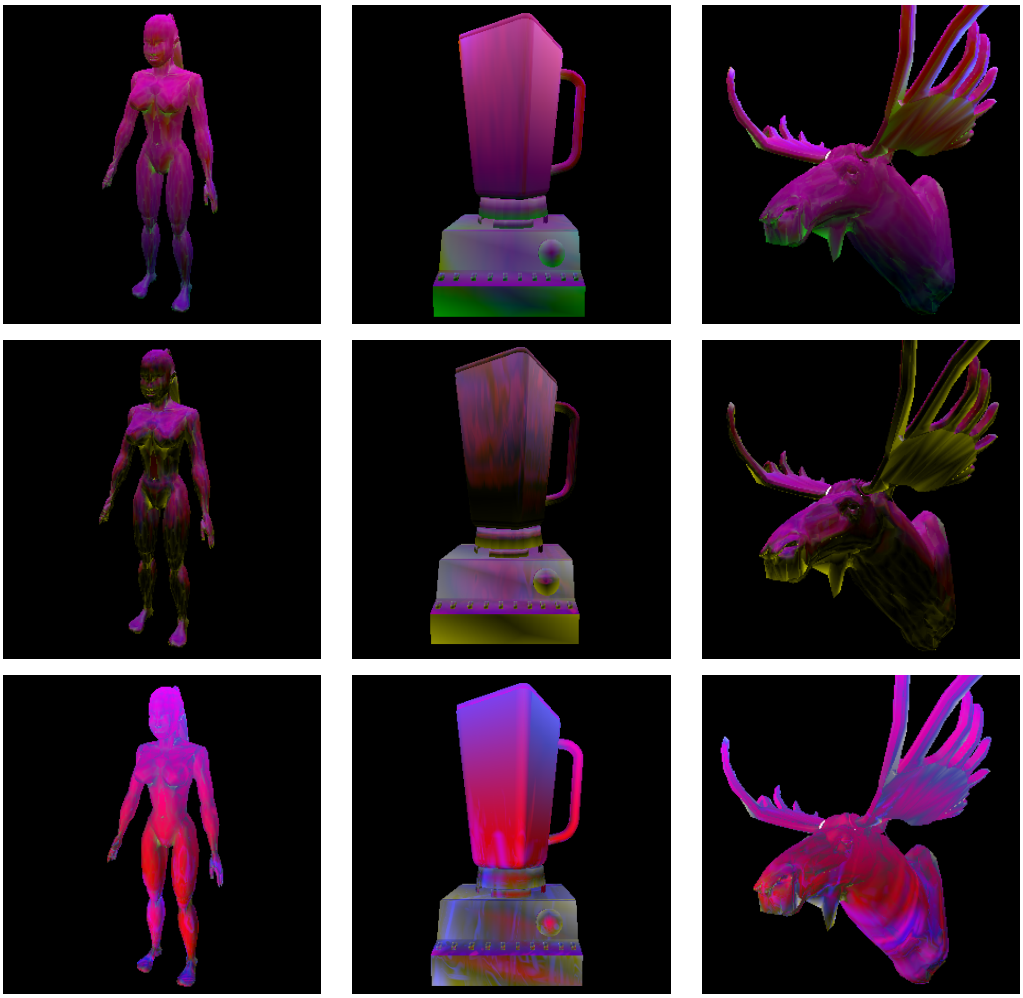

Fig. 13. Swapping blue channel with gradient. Each row is a solution.

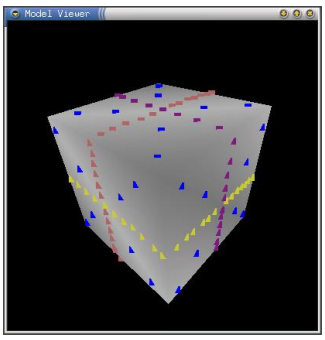

Fig. 14. Banded cube training points.

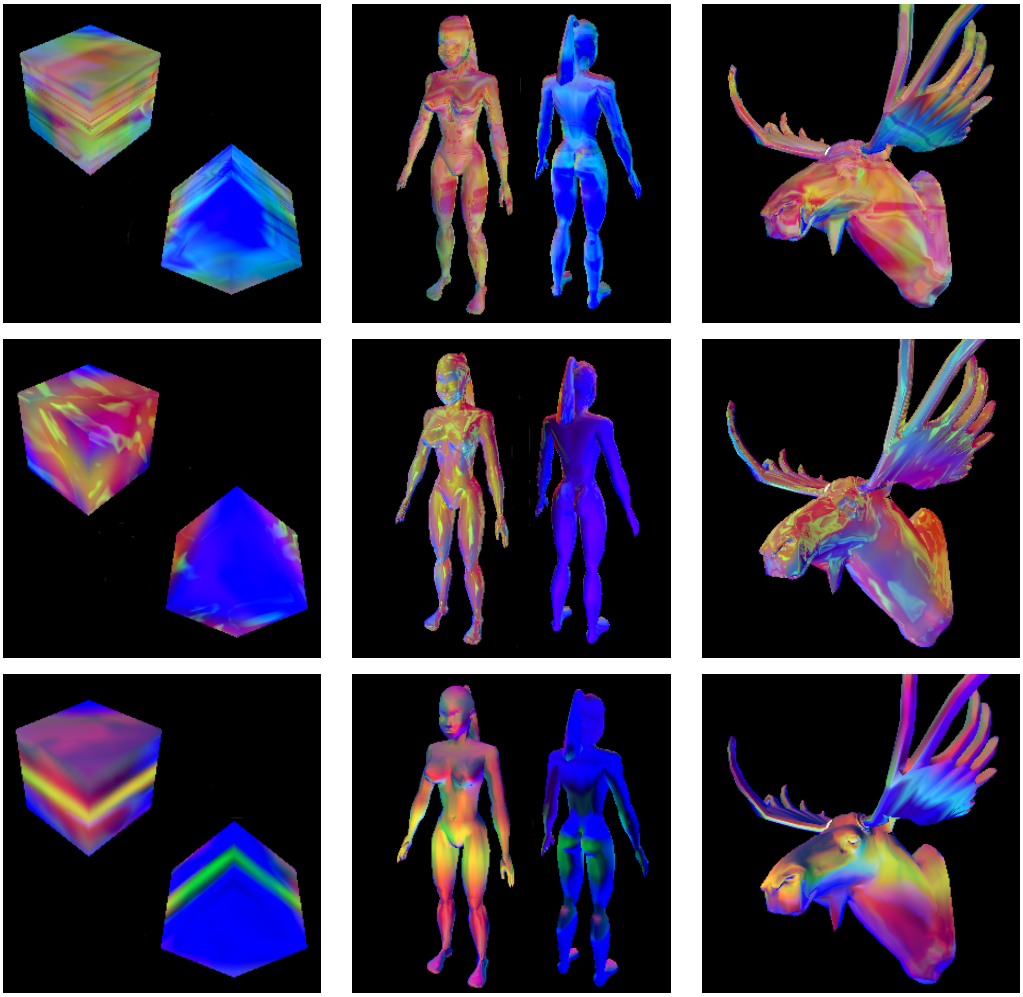

Fig. 15. Banded cube texture. Each row is a solution.