Genetic selection of parametric scenes

Bruce Merry[*] Gianni Giacchetta[†] Bruce Thwaits[‡] James Gain[§]

Technical Report CS03-12-00
Department of Computer Science
University of Cape Town

Abstract

Using a modelling package such as Alias Maya or SoftImage XSi to create a natural scene is too tedious to be practical. Procedural generation techniques reduce the amount of work involved, but there may still be too many parameters to be selected manually. We propose a new method of generating natural scenes, using a genetic algorithm (GA) to infer the user's preferences from user feedback. In order to allow the goal to be reached in a reasonable time, the GA must converge quickly. The scene generation and display preprocessing must also be efficient. We present techniques that attain these goals while still producing reasonable quality output and interactive frame-rates. We also compare this approach to having a user manually select parameters.

1 Introduction

Traditionally, virtual environments have been modelled by hand, using 3D modelling packages such as Maya. More recently, procedural methods have been used to remove much of the manual labour by automatically generating complex scenes. Nevertheless, creating a procedural scene depends on tuning many, possibly hundreds, of parameters. Performing this tuning is difficult and time-consuming, especially as some parameters may not obviously correspond to a physical aspect of the environment.

We propose using artificial intelligence to guide the selection of parameters. The fitness of scenes will be determined by the user, rather than an automated fitness function. We have created a system for generating virtual forest scenes. These scenes contain trees, terrain, clouds and sky. It is divided into three primary subsystems:

1. Artificial intelligence
2. Scene generator
3. Renderer

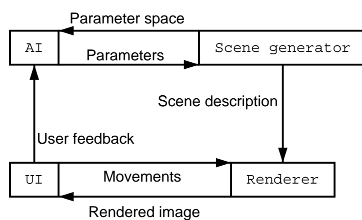

Figure 1: System design. The boxes represent components of the system while arrows represent information flows.

[*]bmerry@cs.uct.ac.za
[†]gianni@cs.uct.ac.za
[‡]bthwaits@cs.uct.ac.za
[§]jgain@cs.uct.ac.za

These subsystems are related as shown in figure 1.

Section 2 surveys previous work in each of these fields. Section 3 explains our overall design, while sections 4–6 describe the three components above. Finally we present results and conclusions.

2 Background

Using user feedback to guide a search has been fairly successful in operations research [Anderson et al. 1999; Scott et al. 2002]. When used to control evolution in an artificial life system, the technique is known as *interactive evolution*. Sims [1991] pioneered the application of interactive evolution in the field of computer graphics. He uses the technique to produce abstract images (by evolving Lisp expressions) and plant models (by evolving parameter vectors).

Rowland and Biocca [2000] use genetic algorithms to produce sculptures (described by parameter vectors). Rather than a user providing fitness values, the user directly selects individuals to be mated or mutated.

Previous work has generally focused on evolving single objects, such as plants or sculptures. In contrast, our system produces entire scenes, with multiple elements (trees, ground, sky and clouds). This poses new challenges, as the system must be efficient enough to handle the complexity.

2.1 Genetic algorithms

Searching through a problem domain for an optimal solution is a common problem in modern scientific and business arenas. Doing so by hand is needlessly inefficient, and with human error built into the equation the probability of correctly finding an optimal solution is very low.

Computational methods are possible for simpler problems (the class P of polynomial problems, for example), and more complex computational solutions to the simple NP problems are also possible.

In the case of NP-complete problems, however, no polynomial-time algorithms are known. In this case, computational searches must be used in order to search a parameter space. Many different approaches to this exist:

2.1.1 Random search

By randomly generating solutions one has the possibility of running into an optimal solution by chance, however unlikely. This approach gives no guarantee that such a solution will be found, and with a large enough parameter space, the likelihood of such a solution being found in a reasonable amount of time becomes vanishingly small.

2.1.2 Biased random search

An improvement to the random search algorithm is to bias the results towards an area of the parameter space more likely to hold

the intended solution. If one knew that a specific portion of a bit-string should always hold specific values, for example, one could lock these bit values, thereby biasing the search results.

2.1.3 Exhaustive search

In an exhaustive search, as the name suggests, all possible combinations of bit values are exhaustively tested. This guarantees a perfect solution, and in fact it is the only approach to parameter space searching which does. However, the number of possible solutions which are tested is 2^x, where x is the number of bits. With a parameter space of 20 bits and with a testing time of half a second per solution, it would require 6 days to test all possible solutions, while another algorithm would be likely to find an acceptable (i.e. about 95% acceptance) solution within a much shorter time. This approach is only used either when the parameter space is small, when massive computational resources are available, or when perfection is required. However, these three are seldom the case.

2.1.4 Genetic Algorithms

A different possible approach is to employ genetic selection [Wright 1991]. Genetic operators (similar to those that occur in combining the genes of parents to produce the genetic identity of a child) can be used on these genes (as if they were normal biological genes) to create new potential solutions in the problem domain. In this way, instead of iteratively trying better bit-strings in some kind of mathematical fashion, discarding valuable information contained in each generated combination, the information held within better genes is kept, and thus a more logical search is performed. This allows for faster and more efficient searching of the problem domain for better solutions. An optimal solution is not guaranteed, however, as only by using an exhaustive search can this be confirmed. Genetic selection is more commonly used, however, because of the radically reduced computation time.

Two major parameters can be adjusted in order to affect changes to the way that Genetic Algorithms operate. A crossover parameter specifies how genes are combined, allowing a greater or lesser amount of "mixing" of the genetic data. A mutation parameter specifies how many bits in each gene produces are spontaneously mutated (i.e. their bit value is flipped), allowing the algorithm to explore the parameter space thoroughly.

Many different approaches to genetic selection exist, and each has its merits and downfalls. Conventional approaches use techniques taken directly from genetic science, and obtain new genes by combining older genes which were closest to the required solution. In this way information is retained in the genes which are currently in the "gene pool". However, much information is still discarded with each generation of the algorithm.

Newer (and more efficient) approaches maintain an internal state which is used to produce new genes, and genes which are close to the required solution are used to update this internal data in some fashion.

2.1.5 PBIL/ACO

Baluja and Caruana suggested replacing the standard genetic approach to GAs with more mathematical models [Baluja and Caruana 1995]. The Population Based Incremental Learning algorithm [Baluja 1994] and the Ant Colony Optimisation algorithm [Dorigo and Di Caro 1999] are both newer approaches to Genetic Searches, and are essentially the same in terms of their operation, so we shall only outline the operation of the PBIL algorithm.

The PBIL algorithm keeps an internal structure known as a probability vector (PV). Essentially, it is a vector (of the same length as the genes being generated) of values between 0 and 1. The closer a value is to 1, the more likely a 1 is to appear in that bit position, and similarly for 0. The initial values of this vector are usually set to 0.5.

PBIL operates as follows :

1. Genes are generated based on the current state of the PV. This is done by generating random numbers between 0 and 1, and applying thresholds of the current PV values in order to generate a sequence of 0s and 1s.

2. Genes are rated as for a conventional GA. Generally, a rating of 0 to 10 is used, although more complex methods of feedback can also be used.

3. The PV is updated according to the information held within the genes of the current generation. This is done by taking a weighted sum of the current PV values with the values of what we call an "updating vector". The exact weightings are determined by the convergence rate.

4. Mutation occurs on the PV. This is either done by shifting the PV values a small distance towards 0.5, or by taking a weighted average of the current PV values and random values between 0 and 1. The weightings are determined by the mutation rate.

5. If an optimal solution has been found, then terminate, otherwise return to step 1. Other implementations run for a specific number of generations, rather than for an unspecified amount of time.

As for conventional GAs, two major parameters exist: convergence rate, and mutation rate. Convergence rate dictates to what extent genes in each generation are allowed to affect the values in the PV. Convergence rates that are too high allow for possible destruction of valuable information held within the PV. Convergence rates that are too low, while making the process of finding an optimal solution slower, do not really destroy any valuable information, and thus lower values are usually recommended. Mutation rate dictates how much mutation is allowed to occur on the PV in each generation. The effect of this parameter is the same as the effect of the mutation operator in a conventional GA.

2.2 Procedural generation

Computers are perfect for repetitive and tedious tasks.

Algorithms exist for producing complex images and objects that would be near to impossible for a human to create manually. Examples of complex objects and images include fire, trees, wood and marble textures and mountains. Researchers such as Ebert, Musgrave and Perlin have discovered and invented several techniques such as Multifractals and Perlin Noise for producing some of these natural phenomena [1994]. Lindenmayer and Prusinkiewicz [1990] invented a formal system called Lindenmayer Systems (or L-Systems) for generating flora.

2.2.1 Fractals

Nature often exhibits self-similarity. For example, the leaves of a fern looks like a miniature version of the entire plant. Another example is a ocean wave breaking upon the shore. The wave appears to have little miniature waves and in those waves there are even smaller waves.

Mandelbrot invented *fractals* [Ebert et al. 1994] which are a mathematical technique used to produce self-similar objects such as coastlines. Fractals have infinite detail which implies that one may zoom into a fractal without reduction of detail.

Fractals have been used to generate mountains and even some plants [Prusinkiewicz and Lindenmayer 1990].

2.2.2 L-Systems

With reference to the previous section, many plants exhibit self-similarity. This implies that a plant can be generated according to a set of rules or a formal system. Such a system is called a Lindenmayer-System (or L-System) [Prusinkiewicz and Lindenmayer 1990]. They comprise a formal grammar, an axiom and an alphabet just like any other formal language. The difference between L-Systems and formal grammars such as context-sensitive and context-free grammars is that, instead of the input string (or axiom) being processed from left to right, they are processed in parallel and therefore are called *parallel rewriting systems*. As a result, a context-free L-System is more powerful than any context-sensitive grammar.

L-Systems are used to generate natural objects such as trees and plants as well as artificial structures such as road networks and buildings [Parish and Müller 2001].

2.2.3 Noise functions

Nature is full of randomness. However, this randomness is not the same as other randomness; for example, the output produced by Pseudo-Random Number Generators (PRNGs) is not smooth but a series of discrete points instead. Therefore, a function that takes the output of a PRNG and produces a smoothed random sequence instead is necessary. One such function is the Perlin Noise generator [Ebert et al. 1994]. It uses several types of interpolation to produce a random sequence of numbers suitable for generating natural scenes on a computer. Noise generators are different from fractals in that they are differentiable where as the latter is not. This is important if the goal is to replicate rolling hills and valleys in a computer-generated natural scene.

An improvement to the Perlin Noise generator is fractal Brownian motion (fBm), which is simply a weighted sum of the output from individual Perlin Noise generators. The benefit of using fBm is that it produces localised effects, whereas Perlin Noise produces that same randomness globally. A real life example is a plateau with relief features.

2.3 Rendering

Despite the rapid advances in graphics hardware in the last ten years, a highly detailed forest scene places a heavy demand on the rendering system. When four of these must be displayed simultaneously, the frame rate would be unacceptable without some optimisation. A high-level way to optimise rendering is to only render objects with the amount of detail required, depending on the distance from the viewer. This approach is known as "level of detail". This is a large area of research; Garland [1999] provides a summary of the various approaches.

A fairly successful approach to automatically creating the various representations is the progressive mesh [Hoppe 1996]. A progressive mesh simplifies a complex mesh by applying a sequence of edge collapses. An edge collapse replaces two vertices with one, and destroys two faces (see figure 2). Different implementations of progressive meshes differ in the way they choose which edges to collapse, the order to collapse them and where to place the new vertex. If the choices are good, then the resulting meshes will have roughly the same appearance as the original but with many fewer vertices and faces.

There is a range of ways to make the choices that trade off processing time against mesh quality. The original progressive mesh paper [Hoppe 1996] performs an expensive non-linear optimisation to get very good results. Garland and Heckbert [1997] provide a simpler scheme using quadric functions. Associated with each vertex V is a quadric function Q_V. The value of $Q_V(p)$ (where p is a position in space) estimates the sum of the squares of the distances of p from each face incident on V. If an edge joining V_1 and V_2 is collapsed to a new vertex at position p, then the cost associated with this collapse is $Q_{V_1}(p)+Q_{V_2}(p)$. Finding an optimal placement for p given V_1 and V_2 requires solving a linear system. This quadric scheme does not take into account attributes such as texture. Garland and Heckbert [1998] propose an extension of the scheme that treats position and attributes as part of an n-element vector, and applies the same technique in n dimensions. Hoppe [1999] further extends this scheme to account for infinitely sharp creases. Other schemes include [Lindstrom and Turk 2000; Fei and Wu 1999; Garland and Shaffer 2002; Lindstrom and Turk 1998].

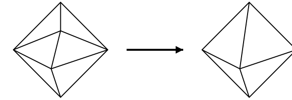

Figure 2: The edge collapse operation for a progressive mesh

Applying the level-of-detail concept to terrain has slightly different requirements. In a walk-through of a virtual environment, there will always be some parts of the terrain that are very close to the camera, and hence must be rendered at the highest level of detail. However, areas that are further away may be rendered at lower detail. The display of several levels of detail within a single mesh is known as selective or view-dependent refinement. This is more complicated than rendering a mesh at a single level of detail, since one must preserve geometric continuity across levels of detail to prevent from cracks appearing [Lindstrom et al. 1996]. However, terrain has the advantage of having very simple topology and connectivity, and there are numerous algorithms that take advantage of this [Lindstrom et al. 1996; Duchaineau et al. 1997; Vlietinck 2003].

2.3.1 Continuous level of detail

Lindstrom et al [1996] define several continuity properties in a LOD scheme that are desirable (the names are our own):

Temporal continuity: The rendered geometry should be a continuous function of time as the viewpoint changes. Alternatively, the rendered geometry should be a continuous function of the position and direction of the viewpoint. Temporal continuity is usually achieved by morphing between discrete levels of detail.

Geometric continuity: If a LOD scheme renders different parts of a continuous model at different levels of detail, the rendered geometry should nevertheless be continuous (as otherwise cracks will appear).

Polygon continuity: For a sufficiently small change in viewpoint, at most one polygon should be added or removed from any given region.

We considered geometric continuity to be absolutely required, as cracks in a mesh are very unsightly. Polygon continuity was not considered, but in practice the algorithms used will change only a few polygons for a sufficiently small change in viewpoint. We use "continuous level of detail" to refer to temporal continuity. Lack of temporal continuity is seen as "popping" — pieces of geometry suddenly appearing or disappearing from one frame to the next.

3 Our approach

For generality, we use a bit vector to encode genetic information. Other work has sometimes used more structured genes (cf. [Sims

1991; Rowland and Biocca 2000]), but such structures are usually very specific to some problem domain. Since the genes are used to control a range of different things (trees, clouds etc), a bit vector is a more flexible approach.

Our user interface displays four candidate scenes, and asks the user to select the best one. This is used to update the state of the AI, which then produces four more scenes to be evaluated. This is approach is similar to that of Rowland and Biocca [2000], in that the user selects a best candidate rather than supplying explicit fitness values to all the candidates. However, we are not using a classical genetic algorithm (see section 4), so this candidate is not simply used for mutation.

We have found that selecting a single "best" candidate is difficult, as one must weigh up different elements of the scene. It also hampers the AI; for example, if the best scene has the wrong type of trees but is otherwise very good, the AI will use the genetic material controlling trees even though it is poor. To solve this problem, we ask the user to select the best candidate in each of three areas: trees, terrain and sky (see figure 5). The genes controlling each of these areas undergo parallel but entirely separate evolution.

4 Genetic Algorithms

The Genetic Algorithm employed in this system is Population Based Incremental Learning (PBIL). As such, convergence to an optimal solution occurs much faster than with a conventional Genetic Algorithm.

Initially, the algorithm treated the bit-strings it dealt with as "black boxes", not giving any regard to any internal structure of the genes on which it operated. This was so that emphasis could be placed on the internal structure of the PBIL algorithm implemented in the system.

However, once this internal structure was optimised as much as possible, regard was finally given to the structure of the genes used. The scene generation software utilised specific bit positions to specify specific parameters which defined the make-up of generated scenes, and as such treating portions of the genes as separate entities allowed the PBIL algorithm to better converge towards a solution.

4.1 Bit locking

The initial attempt to speed up the operation of the GA consisted of locking specific portions of the Probability Vector (i.e. trees, terrain or sky) in order to allow the user to concentrate on specific areas of the scene.

Tests with this new method of searching through the parameter space to find an intended solution provided some acceleration of the process, and also some new problems.

A user that used this system tends to concentrate only on a specific aspect of the system (e.g. trees, sky, or terrain), and ignores the other aspects of the scene. When they locked part of the Probability Vector, and progressed to another part of the scene, the other bit positions did not start from their initialised position of 0.5. Instead, they had already moved from that position due to the algorithm's natural progression from its starting point, and had performed a partial accidental convergence.

An obvious solution to this problem is to lock all bits that are not intended to move (e.g. if working on trees, lock sky and terrain bits), and lock and unlock bits as needed. This overcomes the problem of accidental convergence, and allows for the bit-locking algorithm to perform as it was intended.

4.2 Simultaneous scene aspect evaluation

In the algorithm described above, the bits that are locked are a wasted opportunity. Instead of working on only one aspect of a scene at a time, it is possible to work on all aspects simultaneously.

The algorithm until this point in the development of the system consisted of feedback which led to one of the four scenes produced being used as a vector to update the Probability Vector, and any information contained in the other three scenes was wasted.

The solution to this wastage was to allow the user to label one of the four scenes as the best in the categories of terrain, trees and sky. From this feedback the GA constructed its own feedback vector based on the portions of the scene vectors corresponding to the specific aspects of a scene.

This effectively allowed for a simultaneous evaluation and convergence in the three major aspects of the scenes, providing a three to one advantage in speed over the bit-locking algorithm.

5 Scene generation

5.1 Tree Generation

5.1.1 The L-System for generating the skeleton of the tree

Trees generation involved the use of a parametric context-free L-Grammar based on the grammar described in [Prusinkiewicz and Lindenmayer 1990, pp 55-57]. The aforementioned grammar produced trees that were too flat in appearance. The grammar was modified to produce trees that were more natural in appearance.

5.1.2 The generation of the tree mesh

Using the L-System mentioned in 5.1.1, the mesh for the tree was generated using polygonal cylinders. The cylinders were tapered to mimic branches on real life trees. To make the join between branches smoother, the end of the cylinder connecting with the end of another cylinder was conical, rather than flat.

The use of cylinders with conical ends still does not produce a smooth, continuous join between the tree limbs. Several approaches for producing smooth joins were investigated, such as Constructive Solid Geometry (CSG), meta-objects (otherwise known as Blobby objects) [Bloomenthal 2003] and the method described in [Bloomenthal 1985]. It was decided to make use of meta-objects to create the tree mesh for the following reasons:

- The other approaches were not general enough to apply to any type of tree.
- Meta Objects were easier to implement than the other approaches, given the time constraints.
- A 2-manifold mesh is quite easily produced by meta-objects when the Marching Cubes algorithm is used. This is advantageous (although not required) for level of detail algorithms (see Section 6.2) and subdivision surfaces.

The resulting string from the L-System is used to build the skeleton of the tree using Turtle commands [Prusinkiewicz and Lindenmayer 1990, pp 6-8]. The meta-tubes are then aligned with the starting points and the direction with each of the limb axes of the tree skeleton. Next, a 3D regular grid (or lattice) is placed over the meta-tube assembly which comprises the tree. We loop though all the points in the lattice, calculating the sum of all the field strengths of each meta-tube. The lattice, now with an associated scalar value for each point, becomes a scalar field. Finally, we need to produce the triangle mesh of the tree using this scalar field. We do this is by using the Marching Cubes Algorithm [Lorensen and Cline 1987].

The overall process of generating a tree is illustrated in Figure 3 and a close-up screenshot of a smooth join can be seen in Figure 9.

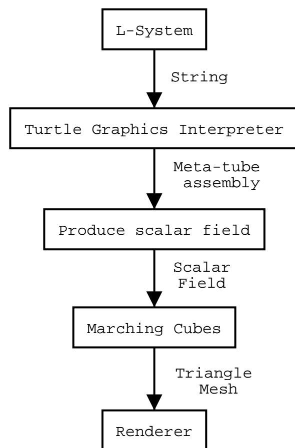

Figure 3: A flow-chart depicting the process in which a tree is generated and rendered.

5.2 Terrain

As mentioned in Section 6.1, the terrain is represented as a height-field. The height values are the output from a 2D dimensional fractal Brownian motion noise generator.

5.3 Clouds

The clouds were implemented as two very large square polygons (quads), each with a cloud texture map. The cloud texture map was generated in a very similar manner to the height-field values for the terrain but interpreted as cloud density rather than terrain height. However, this is not enough as the result is a plasma-like effect. In order to obtain the blotch-like effect that real cloudy skies have, the texture map needs to be filtered [Elias n. d.].

6 Rendering

Efficient rendering of a complex forest scene is clearly required for the system to be usable. However, scenes are generated on the fly and so any preprocessing must be kept to a minimum. Our rendering system aims to balance these requirements, producing interactive frame rates with a minimum of preprocessing. Our goal is to be able to render four scenes simultaneously with a minimum of 15 frames per second, and to be able to preprocess each scene within a few seconds.

We also experimented with both continuous- and non-continuous level of detail schemes. We used hardware with a programmable vertex engine to accelerate morphing between levels of detail (see [Lindholm et al. 2001] for a low-level explanation of the programmable vertex engine).

6.1 Terrain

The terrain is implemented as a regular heightfield. We implemented an algorithm similar to that of Vlietinck [2003]. This algorithm is designed specifically for programmable vertex hardware, such as is found in the GeForce 3 [Lindholm et al. 2001]. It uses regular triangulations, but morphs between coarse triangulations in the distance and fine triangulations close up. Figure 4 shows the interpolations performed by this algorithm.

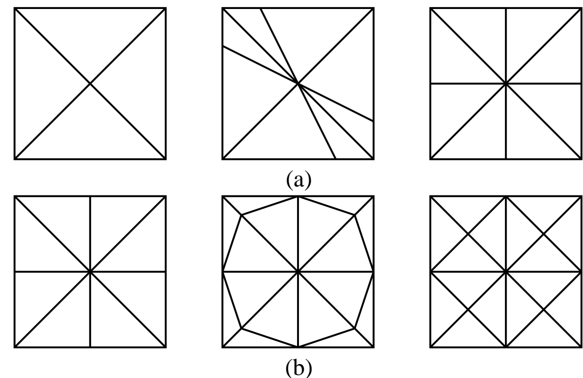

Figure 4: Interpolations used by Vlietinck's algorithm

Every point in the terrain has an associated level of detail, determined as a logarithmic function of the depth. Integer level of detail numbers (LOD numbers) correspond to uniform tilings, such as those in the left and right of figure 4. Non-integer LOD numbers correspond to geometric interpolations between the adjacent uniform tilings. The vertex program used by the algorithm is able to render a block of terrain whose LOD numbers lie between $i-1$ and $i+1$, for some integer i. The algorithm operates recursively, starting with the entire terrain itself. The recursive procedure operates as follows:

1. If the current block lies entirely outside the view frustum, do nothing and return.

2. Find a bound on the range of LOD numbers in the current block. This is done using a bounding box for the block. If the range does not fit into an interval $[i-1, i+1]$ for some integer i, then split the block into four sub-blocks and render these recursively.

3. Otherwise, render the current block using the vertex program.

We modify the algorithm to skip step 1 for blocks over a certain size. This allows a larger portion of the terrain to be visibility culled. The results can be seen in figure 8. Note the increased detail close to the lower left corner, where the camera in located.

For comparison, we also implemented an algorithm based on Real-time Optimally Adapting Meshes (ROAM) [Duchaineau et al. 1997]. This algorithm does not depend on specific hardware features, but does not lend itself to continuous level of detail. We found that the vertex program algorithm significantly out-performed the ROAM-based algorithm, with similar quality results. Our implementation uses the vertex program algorithm when hardware support is available, and falls back to ROAM otherwise.

6.2 Trees

Trees are implemented as general meshes, and level-of-detail is provided with progressive meshes. The simplification metric used by Hoppe [1996] is designed for off-line preprocessing and hence is too slow for our purposes. Fei and Wu [1999] achieve very fast simplification by considering only edge length and local curvature. However, in our own experiments with metrics based on edge length we found that trees would lose volume very quickly. Instead, we use the error metric of Garland and Heckbert [Garland and Heckbert 1997] to generate the mesh. We further improve the efficiency by constraining collapsed points to lie on the line joining

the original points. This was found to make very little difference in the model quality, and actually reduced texture sliding.

We follow Hoppe [1996] in using a small subset of the progressive mesh sequence and geomorphing between them. Specifically, we use a set in which each mesh has half the number of vertices of the previous mesh. The interpolation is performed using a hardware vertex program. This approach is similar to that of Southern and Gain [2003], but is not constrained by their batched hierarchy.

Hoppe [1996] does not address the problem of selecting a level of detail based on distance. We found that significantly better results could be achieved by basing the decision on the error metric itself. Each mesh is assigned a cost, which is the cost of the most expensive edge collapse used to create it. From this we can compute an ideal distance for the tree, of the form $D = \alpha E^{\beta}$. α determines the trade-off between speed and quality, and β is chosen to make the equation scale-invariant (for example, if E is a volume measure then $\beta = \frac{1}{3}$). The actual distance is used to determine a linear interpolation between the meshes whose ideal distances bound the actual distance.

6.3 Sky

The background of the sky is implemented simply as a textured box. We also considered using a tessellated dome, with colours assigned to the vertices. This would allow the sky to be easy modified, but this is not something we had a need for. We rejected this approach as it had no advantages for our application, and would be been more complex to implement.

Using a textured box for the sky allows other features to be "painted on". Clouds could have been implemented this way, but this would have posed difficulties for the scene generator (painting across seams, for example). Instead, clouds are painted onto horizontal rectangles that are placed inside the sky box. They are implemented as billboards (i.e. with a transparency channel) so that the sky and higher clouds will show through gaps.

7 Results

We tested our system with ten users by asking them to produce five specific scenes. We also implemented an interface that allowed parameters to be selected manually (using sliders), and gave users the same tasks. To avoid a learning bias, some users used the AI interface first while others used the manual interface first. Users were asked to indicate which interface they preferred and to rate each interface for how close it came to producing the target scenes. For each combination of scene and interface, they were also asked to rate the difficulty.

The summarised results are shown in table 1. The difference figures are the difference between the ratings given to the manual and AI interfaces (AI − manual). "Task average" is the average difficulty of the five tasks, as rated by the user. "Closeness" is the users' ratings of how close they got to what they wanted. Raw ratings are on a scale of 1 to 5, so the difference ratings range from -4 to 4. The preference field is 1 if the AI approach is preferred and -1 if the manual approach was preferred (so that the average will be 0 if the approach are equal). The mean of -0.2 indicates that four users preferred the AI interface while six preferred the manual interface. The t-test column is the significance level at which the alternate hypothesis of $\mu \neq 0$ is accepted, using a two-sided t-test.

The negative means show that on all criteria, the manual interface performed better than the AI, which is very disappointing. Possible reasons for this and improvements are discussed in the conclusions.

Result	Mean	σ(mean)	t-test
Task average difference	-0.22	0.11	83%
Closeness difference	-0.75	0.21	98.5%
Preference	-0.2	1.03	N/A

Table 1: Average results

7.1 Genetic Algorithms

Our goals for the genetic algorithm were that it should:

1. Make noticeable improvements (however slight in the first few generations) with each generation of solutions.

2. Converge to a solution with a very high acceptance rating (e.g. with a percentage score, 90% or more).

3. Converge within 40 to 50 iterations (40 to 50 minutes of user time) of the walk-through engine to an acceptable solution.

The results in these criteria were as follows:

1. Due to a convergence rate of 0.2, the algorithm was allowed to make noticeable changes to the scene on each iteration. However, since a range of scenes is produced on each view (i.e. 4), these changes are not immediately apparent. This is intentional, however, as the user must be given as much choice as possible as to which scenes they wish to use as a guide for the scenes to follow.

2. Since the algorithm in the final system is set to run until an acceptable solution is found, and not for a set number of generations, an acceptable solution is always found.

3. Depending on the user's attention to detail, an acceptable solution is usually found within 10 to 15 generations. In our initial projections, we imagined a user taking a minute to view one scene at a time, and thus the estimate of 40 iterations means that with 4 scenes viewed at a time we have met our goal in this regard.

   However, our initial estimate assumed that a user would take a minute to view each scene. However, user testing showed that users took a maximum of a minute to view each set of 4 scenes, and thus took only 10 minutes to find an intended solution, exceeding our goal of 40 minutes by a factor of four to one.

The results therefore show that the Genetic Algorithm performed well beyond our initial expectations.

7.2 Scene generation

The goal of the Scene Generator was to be able to produce a scene in about 2 seconds. A balance had to be struck between visual quality and generation time. The attempt to produce better looking trees using meta-tubes did yield good results where the joins between branches were smooth and the method was general enough to be applied to any type of tree. Unfortunately, the generation time was in the order of minutes and therefore unacceptable for an interactive system. The bottleneck was discovered to be the generation of the scalar field from the meta-tube assembly (see Figure 3). Hence, the trees were constructed from cylinders instead and the generation time was in the order 2-3 seconds.

7.3 Rendering

Despite the limitations on preprocessing and having to render four scenes simultaneously, the renderer was able to achieve well over 15 frames per second on our test system[1], and generally over 60 frames per second. Even in the worst case, the frame rate exceeded 15 frames per second.

The processing took longer than we would have liked, with an average of 3 seconds to preprocess each scene. Further work needs to reduce the preprocessing time.

8 Conclusions

Our current implementation does not produce the results we expected. Given the success of interactive evolution in similar of computer graphics [Sims 1991; Rowland and Biocca 2000], it seems likely that the problem is with our implementation and not the concept. Further studies need be done to determine the problem.

As noted by Sims [1991], interactive evolution depends on the system being able to create individuals quickly enough. Our current preprocessing time is rather high, possibly as a result of the scene generator and the renderer being built as independent modular systems. While this is good practice from an engineering point of view, a tighter coupling of the scene generator and the renderer may allow preprocessing times to be reduced. One approach is to have the scene generator assist in creating level-of-detail meshes, using its knowledge of the underlying structure.

One possible problem area is the number of parameters. Rowland and Biocca [2000] use 104 degrees of freedom and 1625 bits to control a single sculpture, and Sims [1991] uses structured genes to create an unbounded genetic space. In contrast, our implementation uses only 21 parameters and 70 bits to control all the aspects of a scene, and 9 parameters are devoted purely to lighting. This makes it possible for users to easily determine the effects of individual parameters, and hence assemble the scene by hand with little experimentation. If the number of parameters was made an order of magnitude larger, manual tuning would become far less feasible. Further testing would be required to determine whether the AI will cope with such a large number of parameters. Since the AI treats the bit string as a number of small independent strings, it is expected to be able to handle the added complexity. Adding parameters would have real value, as there are many aspects of the scene that currently cannot be controlled (such as cloud density, tree size, tree tilt, bark, grass etc).

References

ANDERSON, D., ANDERSON, E., LESH, N., MARKS, J., PERLIN, K., RATAJCZAK, D., AND RYALL, K. 1999. Human-guided simple search: combining information visualization and heuristic search. In *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation in conjunction with the eighth ACM internation conference on Information and knowledge management*, ACM Press, 21–25.

BALUJA, S., AND CARUANA, R. 1995. Removing the genetics from the standard genetic algorithm. In *The Int. Conf. on Machine Learning 1995*, Morgan Kaufmann Publishers, San Mateo, CA, A. Prieditis and S. Russel, Eds., 38–46.

BALUJA, S. 1994. Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning,. Tech. Rep. CMU-CS-94-163, Pittsburgh, PA.

[1]Pentium 4 2GHz, 512MB of RAM, GeForce 4 Ti 4200

BLOOMENTHAL, J. 1985. Modeling the mighty maple. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, 305–311.

BLOOMENTHAL, J., 2003. Implicits: Intro and tour. Online, July. http://www.unchainedgeometry.com/jbloom/pdf/sig03tut.pdf.

DORIGO, M., AND DI CARO, G. 1999. The ant colony optimization meta-heuristic. In *New Ideas in Optimization*, D. Corne, M. Dorigo, and F. Glover, Eds. McGraw-Hill, London, 11–32.

DUCHAINEAU, M. A., WOLINSKY, M., SIGETI, D. E., MILLER, M. C., ALDRICH, C., AND MINEEV-WEINSTEIN, M. B. 1997. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, 81–88.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1994. *Texturing and modeling: a procedural approach*. Academic Press.

ELIAS, H. Cloud cover. Online. http://freespace.virgin.net/hugo.elias/models/m_clouds.htm.

FEI, G., AND WU, E. 1999. A real-time generation algorithm for progressive meshes in dynamic environments. In *Proceedings of the ACM symposium on Virtual reality software and technology*, ACM Press, 178–179.

GARLAND, M., AND HECKBERT, P. S. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 209–216.

GARLAND, M., AND HECKBERT, P. S. 1998. Simplifying surfaces with color and texture using quadric error metrics. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, 263–269.

GARLAND, M., AND SHAFFER, E. 2002. A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization '02*, 117–124.

GARLAND, M. 1999. Multiresolution modeling: Survey and future opportunities. In *Eurographics '99 – State of the Art Reports*, 111–131.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 99–108.

HOPPE, H. 1999. New quadric metric for simplifiying meshes with appearance attributes. In *Proceedings of the conference on Visualization '99*, IEEE Computer Society Press, 59–66.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 149–158.

LINDSTROM, P., AND TURK, G. 1998. Fast and memory efficient polygonal simplification. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, 279–286.

LINDSTROM, P., AND TURK, G. 2000. Image-driven simplification. *ACM Transactions on Graphics (TOG) 19*, 3, 204–241.

LINDSTROM, P., KOLLER, D., RIBARSKY, W., HODGES, L. F., FAUST, N., AND TURNER, G. A. 1996. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 109–118.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM Press, 163–169.

PARISH, Y. I. H., AND MÜLLER, P. 2001. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 301–308.

PRUSINKIEWICZ, P., AND LINDENMAYER, A. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag.

ROWLAND, D., AND BIOCCA, F. 2000. Evolutionary co-operative design between human and computer: implementation of .the genetic sculpture park. In *Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML)*, ACM Press, 75–79.

SCOTT, S. D., LESH, N., AND KLAU, G. W. 2002. Investigating human-computer optimization. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, 155–162.

SIMS, K. 1991. Artificial evolution for computer graphics. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, ACM Press, 319–328.

SOUTHERN, R., AND GAIN, J. 2003. Creation and control of real-time continuous level of detail on programmable graphics hardware. *Computer Graphics Forum Vol. 22*, No. 1 (March), 35–48.

VLIETINCK, J., 2003. Trilinear displacement mapping of a flat surface with a v1.1 vertex shader. `http://users.belgacom.net/xvox/`.

WRIGHT, A. H. 1991. Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms*, G. J. Rawlins, Ed. Morgan Kaufmann, San Mateo, CA, 205–218.

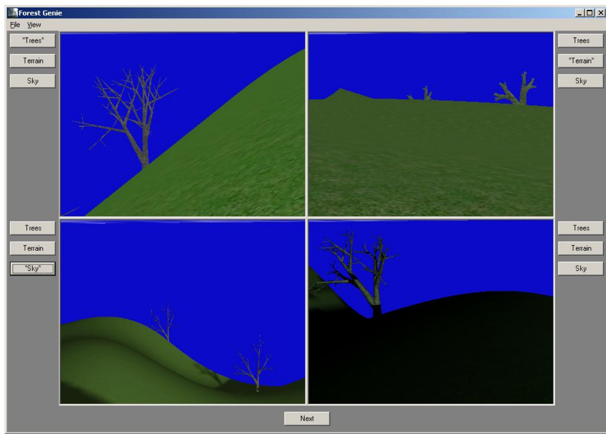

Figure 5: The user interface

Figure 6: A screenshot from the implemented system.

Figure 7: Another screenshot.

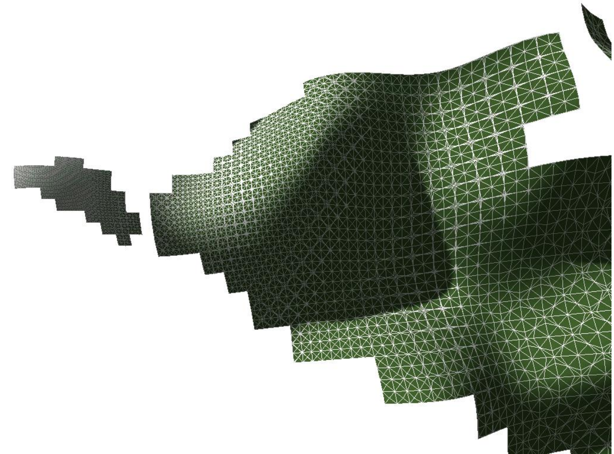

Figure 8: The level-of-detail for terrain (the camera is on the left of the image).

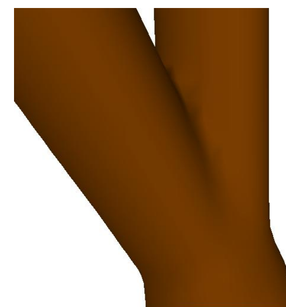

Figure 9: A close-up shot of the smooth join between limbs of a tree generated using meta-tubes and the Marching Cubes algorithm.