

Interactively Evolving Virtual Environment Maps with Continuous Layered Pattern Functions

Matthew Lewis*

Advanced Computing Center for the Arts and Design

The Ohio State University

Richard Parent†

Department of Computer and Information Science

The Ohio State University

Abstract

Height fields are evolved for use in virtual environments. Interactive aesthetic selection is employed as a fitness function for generating successive populations of images with a genetic algorithm. The images are represented using continuous layered pattern functions, which are based on procedural texturing techniques. The design space defined by the representation can be controllably biased toward specific formal qualities.

Keywords: interactive evolutionary design, virtual environments, procedural texturing

1 Introduction

Common desktop computers are currently able to represent extremely complex virtual environments containing thousands of objects and seemingly endless amounts of animation. This capability for complexity opens a bottomless pit of demand for new content design. As high-end 3D graphics make their way onto the web and into every home computer, there is an immense need for software which allows untrained users to design the virtual spaces in which they can now work and play.

One technique which has been used to facilitate non-expert design is called *interactive evolutionary design* (IED). IED allows users to create by selection, rather than construction. The computer first creates a random population of solutions to some design problem. A human designer then judges these solutions for quality. Based on the designer's input, the computer then generates a new population of solutions, using pieces of the selected high fitness individuals to construct the next generation's individuals. This results in increasingly improved designs. As the selection and creation process is iterated, creative solutions can gradually evolve.

IED has successfully been used primarily by specific artists and designers within limited domains. Most implementations suffer from a lack of flexibility as a result of the means used for representing designs. In some cases, the representation is too low-level and it becomes difficult to interactively find the few high-fitness regions in the design space. This problem is often addressed by biasing the

*ACCAD, 1224 Kinnear Road, Columbus, OH 43212 USA, mlewis@cgrg.ohio-state.edu

†CIS, 395 Dreese Lab, 2015 Neal Avenue, Columbus, OH 43210 USA, parent@cis.ohio-state.edu

space toward known high-fitness solutions. This usually results in a distinct signature look, with all solutions generated by the system sharing similar properties [23].

The continuous layered pattern functions discussed here were created as an intermediate-level design space representation primitive which provides a great deal of flexibility and control for a number of domains [11]. This paper demonstrates how these pattern functions can be evolved to produce geometry for use in virtual environments such as are commonly designed for multi-user computer games.

2 Background

For many problem spaces, it is nearly impossible to produce the objective fitness function necessary for automating the design process with genetic techniques. In domains where a human designer *must* judge quality interactively, this is particularly challenging. Following Dawkins' biologically inspired Biomorphs program which evolved 2D branching drawings resembling insects or plants [7], Todd and Latham [27] and Sims [25] were the first to evolve computer graphics images via interactive selection. Since then, interactive evolution has been applied to numerous specific design domains. Takagi recently completed a large survey of systems employing interactive evolutionary computation [26]. Some of the applications most relevant to the work in this paper include the evolution of designs for architecture [5][16][22], images [8][12][21][28] and objects [2][6][9][14].

Sims' original image generation work (and that of the many people who've implemented and extended his techniques) remains the most similar to this work. The primary distinction however is that the approach presented here yields complex images using a fixed-length parametric representation rather than hierarchical expressions. This allows a great deal of control over the mapping genes to specific visual traits. It also promotes greater inheritance of properties in offspring.

3 Interactive Evolutionary Design

Interactive evolutionary interfaces give a user the ability to create designs by simply choosing the best individuals from a computer generated population. New populations are then produced for further selection, by combining the designs selected by the user. The representation of the data used to generate possible solutions is referred to as the *genotype*. The collection of specific traits composing an individual created from a genotype is referred to as the *phenotype*. Different styles of evolutionary algorithms employ different genotype representations. In genetic algorithm-based systems (GAs), a genotype consists of an array of gene values, often referred to as a chromosome. In evolutionary algorithms employing a genetic programming (GP) approach, genotypes are represented with a directed acyclic graph with internal nodes corresponding to functions and leaf nodes containing atomic values.

In an interactive evolutionary design system, after an initial population of models is generated, the resulting individuals are presented to the user for evaluation. Most IED systems present the entire population in a 2D grid, though individuals can also be displayed sequentially if their structure requires it. Users typically specify fitness by selecting those members of the population determined to be the most pleasing (or otherwise "interesting".) Interface components are often provided to influence the progress of evolution. Factors such as the mutation amount and frequency, crossover frequency, population size, and so forth can often be manually adjusted.

In order to mate individuals and produce the next generation, pairs of parents are selected from the current population to be mated. There are numerous methods for selecting parents for mating. In IED it is not uncommon to use the selected individuals as the mating pool. Random pairs are then selected from this pool for recombination. By combining individuals’ genetic material, mating explores regions of the solution space bounded by the individuals. The combination process is usually referred to as *crossover*. Part of one parent’s chromosome is first copied to the offspring, then the copying process “crosses over” to the corresponding position in the other parent’s chromosome to copy the remaining genes. The number of times the process crosses from one parent to the other can be adjusted producing different degrees of property inheritance and space exploration.

After offspring have been generated via crossover, it is often useful to apply some amount of *mutation* to the population’s genes. This can add diversity back into the population. When performing mutation, it is important that small changes in a genotype correspond to small changes in the phenotype [3]. Otherwise it can become difficult to fine-tune solutions.

As subsequent generations are produced, the user often reduces mutation (as well as crossover rates) to slow movement through design space from generation to generation. The search is gradually narrowed into a specific region upon which the user’s interest is focused. Typically the user is either exploring or refining. In order for the user to be able to specify small mutation amounts to refine the best-so-far individuals, it is a necessity that the solution space be continuous. If the space is defined so that phenotypes change abruptly for very small changes in gene values, then it will not be possible for the user to investigate minor variations of a given individual.

4 Continuous Layered Pattern Functions

Solution space design is a critical factor in determining the degree of signature of the individuals likely to be discovered with an interactive evolutionary design system. If the space is too high-level, with too few parameters, the potential diversity of the possible designs will be highly constrained. A representation that is very low-level on the other hand may *theoretically* contain a much more diverse set of individuals. But if highly fit individuals are too rare in the solution space, then they are unlikely to be found with an IED system, given the practical constraints of low population sizes, and few generations. This section describes a representation method that provides a more general building block than most domain-specific parametric component-based methods, while still attempting to be more high-level than uniform discrete atomic-collection style approaches.

The design space presented here is constructed from *layers* of *continuous patterns* made of *features*. The concepts introduced are inspired by procedural texture authoring techniques. Procedural textures are often referred to as *shaders*.

4.1 Features

A pattern function F has a vector of default attribute values A_d associated with it. For example, default image attributes might be the color white and zero displacement height. A sample point p and an array of parameter values G (i.e., genes) are passed to f to determine whether there is a feature at p and what the feature’s attributes are. A feature corresponds to a local region of the function’s domain in which the values vary from A_d in some parametrically controlled way. The attributes of each feature are determined by a subset of G called the *feature genes*:

$$F(G, p) = A_d + f(G, p) \tag{1}$$

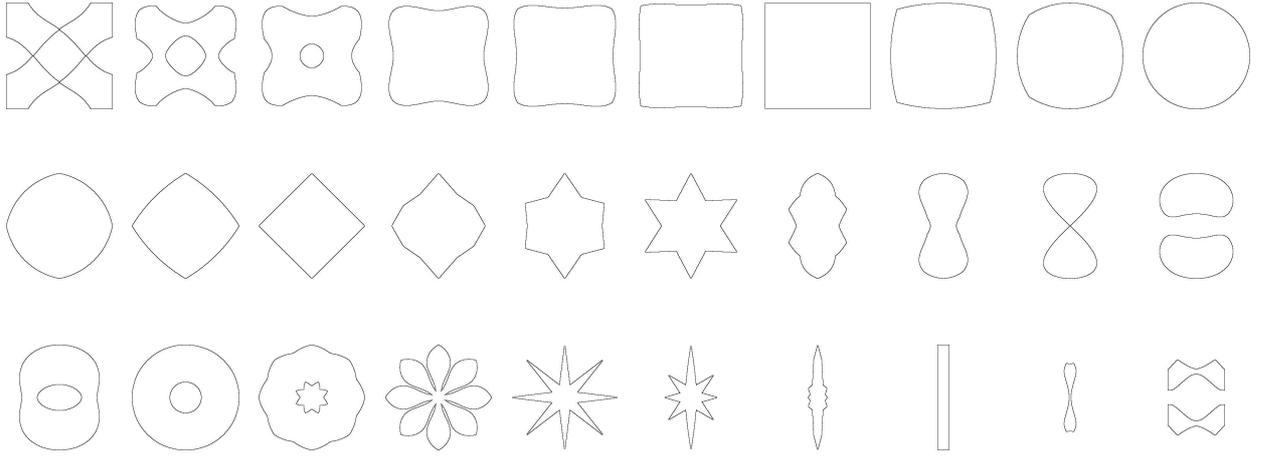


Figure 1: Pairs of implicit shapes can be blended by interpolating the values returned by their implicit functions. The above series shows a blend through a series of ten shapes with two intermediates for each pair. The blend proceeds from the top left, left to right, top to bottom.

One of the primary challenges in feature design is to simultaneously maximize flexibility *and* fitness. A feature’s representation should be capable of instantiating a wide range of shapes, but it should also not degenerate into unstructured noise through most of the parametric space.

Common shader writing practices illustrate how to combine one-dimensional functions to produce two-dimensional feature shapes. The implicit method employed in shader writing relies on determining whether a given point in the domain being sampled is in the interior or exterior of a feature. A common method of producing features for shaders is to produce an appropriate *in_out* test for each desired feature shape. Some common 2D feature shapes include discs, stars, rectangles, and lines [13][17].

For the evolutionary design needs presented here, it is important that we are able to smoothly interpolate from one shape to another. There are many possible representations for 2D shapes that allow interpolation. The method given here provides a continuous mapping between a single shape gene and different feature shapes by using implicit equations. This choice of representation allows for smooth interpolation by simply interpolating the equations [4].

These shapes are not intended to provide a definitive set of all possible feature qualities that might be desired, but rather to illustrate the creation of a continuous shape parameterization yielding a wide diversity of forms. The set of 2D shapes used in this work are shown in figure 1. Sample equations for a number of these shapes follow [15]:

$$\text{circle}(x, y) = x^2 + y^2 - r^2 \tag{2}$$

$$\text{diamond}(x, y) = |x| + |y| - r \tag{3}$$

$$\text{hourglass}(x, y) = x^4 - x^2 + y^2 \tag{4}$$

$$\text{line}(x, y) = \max(|10x| - r, |y| - r) \tag{5}$$

$$\text{pillow}(x, y) = x^4 + y^4 - (x^2 + y^2) \tag{6}$$

$$\text{rods}(x, y) = \max(\max(|x| - r, |y| - r), x^4 + y^4 - 2x^2 - 2y^2 - x^2y^2 + 1) \tag{7}$$

$$\text{square}(x, y) = \max(|x| - r, |y| - r) \tag{8}$$

$$\text{torus}(x, y) = (x^2 + y^2 + r_0^2 - r_1^2)^2 - 4r_0(x^2 + y^2) \tag{9}$$

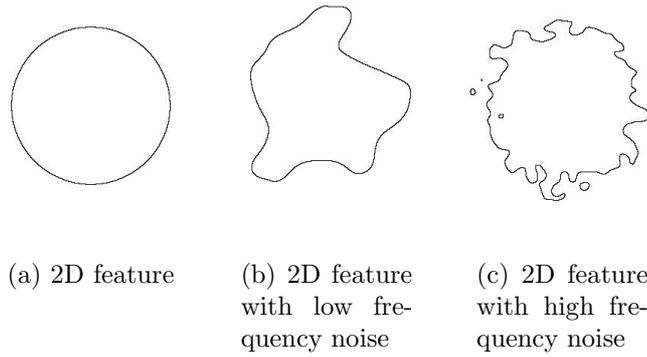


Figure 2: Noise is applied to a feature’s sample positions.

As was mentioned previously, implicitly defined shapes can be smoothly interpolated by interpolating their functions as is shown in figure 1. The rate at which most of the shapes interpolate visually is not linear however. Perlin’s bias and gain functions [19] can be used to empirically construct a blending function with two parameters to control the rate at which shape blending takes place:

$$\text{isoBlend}(s_1, s_2, t, b, g) = \text{lerp}(s_1, s_2, \text{bias}(b, \text{gain}(g, t))) \quad (10)$$

By composing bias and gain in the isoBlend function, the intermediate shape transitions can be intuitively pushed closer to either shape with the bias control. The speed at which the interpolations ease-in and ease-out at either end can also be tuned using the gain parameter.

4.2 Noise

While the above blending approach can create a wide range of feature shapes, they are (by definition) geometric and regular. When irregularity is desired in a procedural texture a continuous Perlin noise function is often used [18]. Noise is perfect for the needs of interactive evolutionary design because it can be controllably and continuously added to regular forms to make them smoothly become more irregular. Noise is a function which, in most implementations, outputs a single value based on its inputs. Small changes in the input “seeds” yield a correspondingly small change in the output value. The rate at which the output value changes is controlled by multiplying the input seeds by a desired frequency.

Noise’s continuity is extremely important for its use in IED. One of the requirements of IED is that small changes in the parametric design space result in small changes in the individual designs. If non-continuous random values were used, this would not be the case, as a small mutation might result in an arbitrarily complex change in the individual. Using noise makes it more likely that small steps in parameter space result in small changes in the final design.

Values produced by the noise function are usually used to perturb another set of values. Noise can be added to a single value or a multidimensional coordinate. Typically a varying value such as the position of the sample is passed to noise as the seed value, multiplied by some frequency f and offset by some value o . The value returned by noise (usually either in $[0..1]$ or $[-1..1]$) is then

scaled by some amplitude a :

$$v' = v + a \cdot \text{noise}(p_x * f_x + o_x) \quad (11)$$

$$v' = v + a \cdot \text{noise}(p_x * f_x + o_x, p_y * f_y + o_y) \quad (12)$$

$$v' = v + a \cdot \text{noise}(p_x * f_x + o_x, p_y * f_y + o_y, p_z * f_z + o_z) \quad (13)$$

Figure 2 shows high and low frequency noise deforming a simple feature. A set of genes controls the amplitude, frequency, and offset of noise applied to the position and value of the samples in individual features.

4.3 Patterns

The features function as primitives for building patterns. A real version of a modulus function is used to make other simple functions periodic. Given a function $f(x)$ defined on $[0, p]$, a periodic version of $f(x)$ with period p can be constructed [1]:

$$\text{mod}(a, b) = b \cdot (a/b - \lfloor a/b \rfloor) \quad (14)$$

$$f_p(x) = f(\text{mod}(x, p)) \quad (15)$$

$$\text{pattern}(x, y, p) = \text{feature}(\text{mod}(x, p), \text{mod}(y, p)) \quad (16)$$

Pattern functions can be created using this technique with any of the feature generation functions previously defined. We can use this implementation of `mod` to create an arbitrary number of copies of any feature function. The `mod` function creates p cells (or *tiles*) each of which contains a copy of the feature. Genes are used to control the period, and whether the number of features along each axis is the same. The following subsections describe some of the ways that patterns of simple repeating features can gain visual complexity.

4.4 Bombing

As was done with individual features above, we can use noise to smoothly make the patterns less regular in a number of ways. An important technique for individualizing the features of a pattern is called *bombing*. Bombing involves using the index of the feature to determine a single noise value to be used for modifying a property of that feature [13][17]:

$$\text{whichTile}(x, f) = \lfloor xf \rfloor \quad (17)$$

$$\text{positionBomb}(x, f, a) = x + a \cdot \text{noise}(\text{whichTile}(x, f)) \quad (18)$$

The `whichTile` function returns a unique integer for each tile. A function like `positionBomb` can then use this tile index. In this example, the x position of a feature is modified, shifting the feature to the left or right. Figure 3 shows examples of bombing several attributes.

Since a single noise value is found for all points on the feature, the entire feature is modified uniformly (e.g., scaled, rotated, colored, etc.) The primary benefit is that each feature can be made unique for a given visual property. Depending on the frequency of the noise used, neighboring features may change their properties gradually or erratically, based on their relative spatial location. Genes' values are mapped to the amplitude, frequency, and offset of the noise used for bombing.

Common feature properties for bombing include size, position, color, and existence. All of the feature parameters defined (e.g., the 2D shape parameter) can be bombed as well. Note that

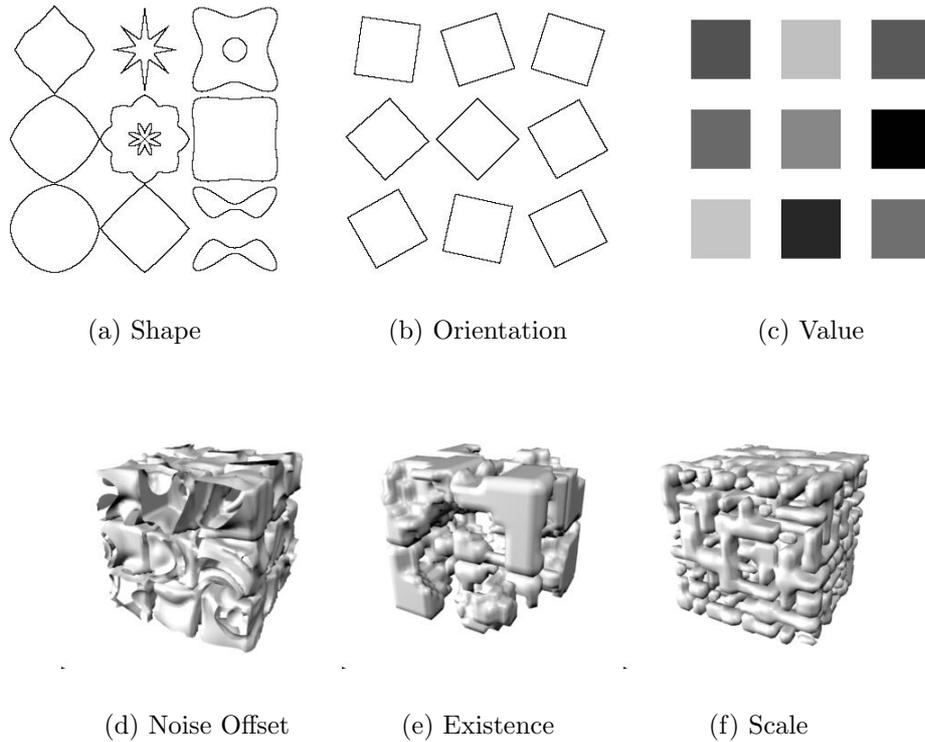


Figure 3: Bombing the values of different attributes.

changing some traits like the size or position can actually move part of a feature (if not all) into a neighboring cell. This results in a clipping of the feature at the cell boundary unless neighboring cells are checked for features as well [17].

4.5 Layers

Once individual patterns of features have been created using the techniques discussed, patterns can be composited for additional complexity. Figure 4 shows a few examples of layers being combined. The method used to combine patterns depends largely on the visual attributes needed for the target design domain. Compositing options include summing, averaging, or taking the maximum value of the individual layers.

The maximum number of layers possible determines the length of the individuals' chromosomes. If a maximum of L layers is to be possible, and the properties of a single layer can be described using N genes, then for a given design space, all individuals will have a chromosome of length $c + (N \cdot L)$. The constant c represents a small number of genes describing layer independent properties of an individual. Note that one of the c genes can determine how many of the L layers are actually used for a given individual, while the rest of the layers remain recessive.

5 Results

The previous section introduced techniques for constructing image design spaces from layered continuous pattern functions. Because of the methods used to construct these solution spaces, they

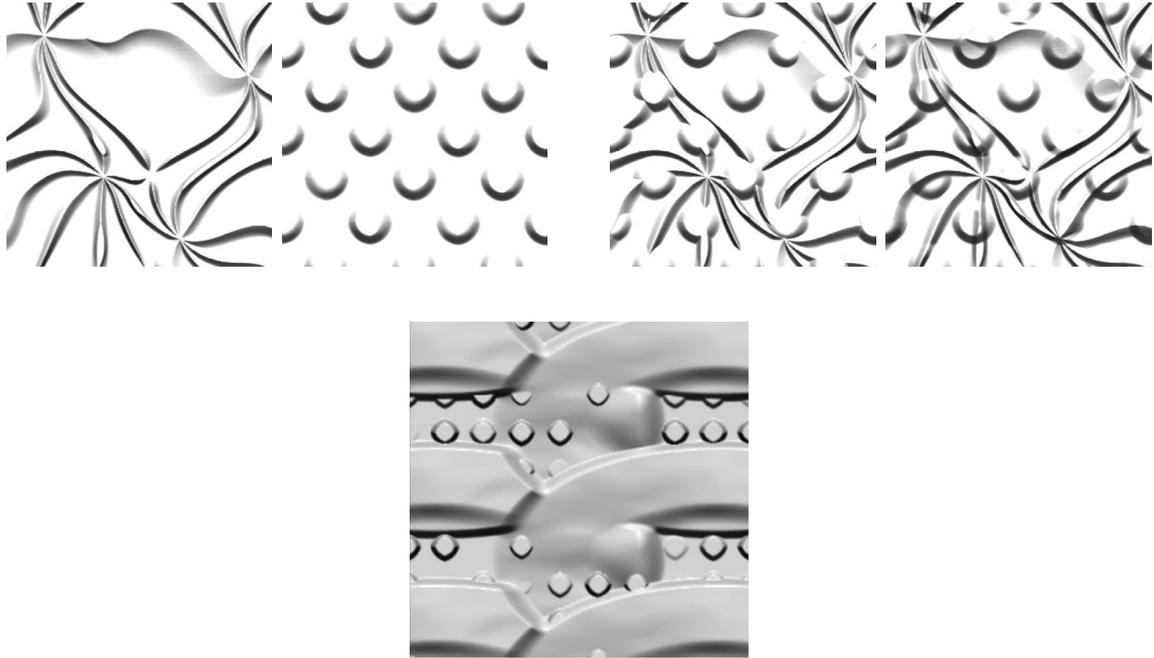


Figure 4: Combining two layers: the two upper-left layers are shown combined in the two upper-right images. The left uses the maximum value while the right uses the sum. The bottom image shows an additional example of two layers.



Figure 5: Two populations of images evolved to be used as height fields are shown.

are interactively searchable using simple, standard genetic algorithms. This section presents a few examples of the results obtained from this approach.

The images were produced within the interface framework of Houdini [24] and RenderMan [20] using a system called Metavolve [10]. For each individual, the geometry and genotype are written to a RenderMan RIB file. A render job is started on a processor for each individual¹. The samples for each pixel in each individual are calculated by passing the UV coordinates at the sample location to a RenderMan shader along with the individual's genes. Once each individual is rendered, they are tiled together into a single mosaic grid image. This image is then texture-mapped onto a grid of selectable objects² for the next round of interactive selection.

By using images like the ones shown in figure 5 to offset the heights of a flat grid of vertices, *height fields* can be created. The ranges and activation thresholds of different properties can be adjusted to bias the system to produce different properties. For example, feature frequency and noise levels can be reduced to make the formation of larger continuous structures more likely. This also makes it more probable that the terrain will be navigable. The domain author is free to adjust the gene value remapping biases and boundaries as he or she sees fit. Figure 5 shows two populations of designs. Two designs were selected and used to generate the game environments shown in figure 6.

When an individual 2D image design is selected in the evolution interface, a 3D representation of the geometry is instantly displayed in an adjoining OpenGL window (see the right images in the first and third rows in figure 6.) This 3D view can be rotated and zoomed for viewing from any direction. This allows the user to preview the translation from 2D to 3D, perhaps revealing hidden advantages or disadvantages that were not as obvious from the overhead view (e.g., the “ramps” from low ground to high ground in the bottom map in figure 6.)

The second and fourth rows in figure 6 show views of the environment after conversion to a game format. Once an environment geometry is evolved, selected, and exported, a script can be run which converts the polygonal geometry into *brushes*. These brushes can then be imported into a game environment authoring tool where lights and player starting positions are added. The environment is then “compiled” which involves computing visibility and lighting information for the different regions of the space. The environment can then be loaded into the game and explored interactively from a first person, ground following perspective.

The top four images in figure 6 show an individual that was chosen for its gentle rolling hills and the unique “spine” that divides the top of one hill with a high wall. By contrast, the map in the bottom four images of figure 6 was chosen to illustrate the potential for generating flat overlapping and adjoining surfaces at different heights. This is a common trait of many environments in the very common “death-match” and “platform” gaming genres.

6 Conclusions and Future Work

A new method for designing real-time virtual environments requiring minimal knowledge of 3D modeling techniques was presented. Interactive evolutionary design interfaces can be used to allow non-expert users to design complex, original computer graphics content. Even expert users can benefit from tools which can aid in brainstorming by suggesting creative new design solutions.

¹Alternatively the entire population can be rendered as a single image using multiple processors.

²The image of each individual is texture mapped onto a four-sided pyramid with its point facing the camera. When an individual is interactively selected, the edges of the selected object are highlighted causing an “X” to be formed over the selected individual.

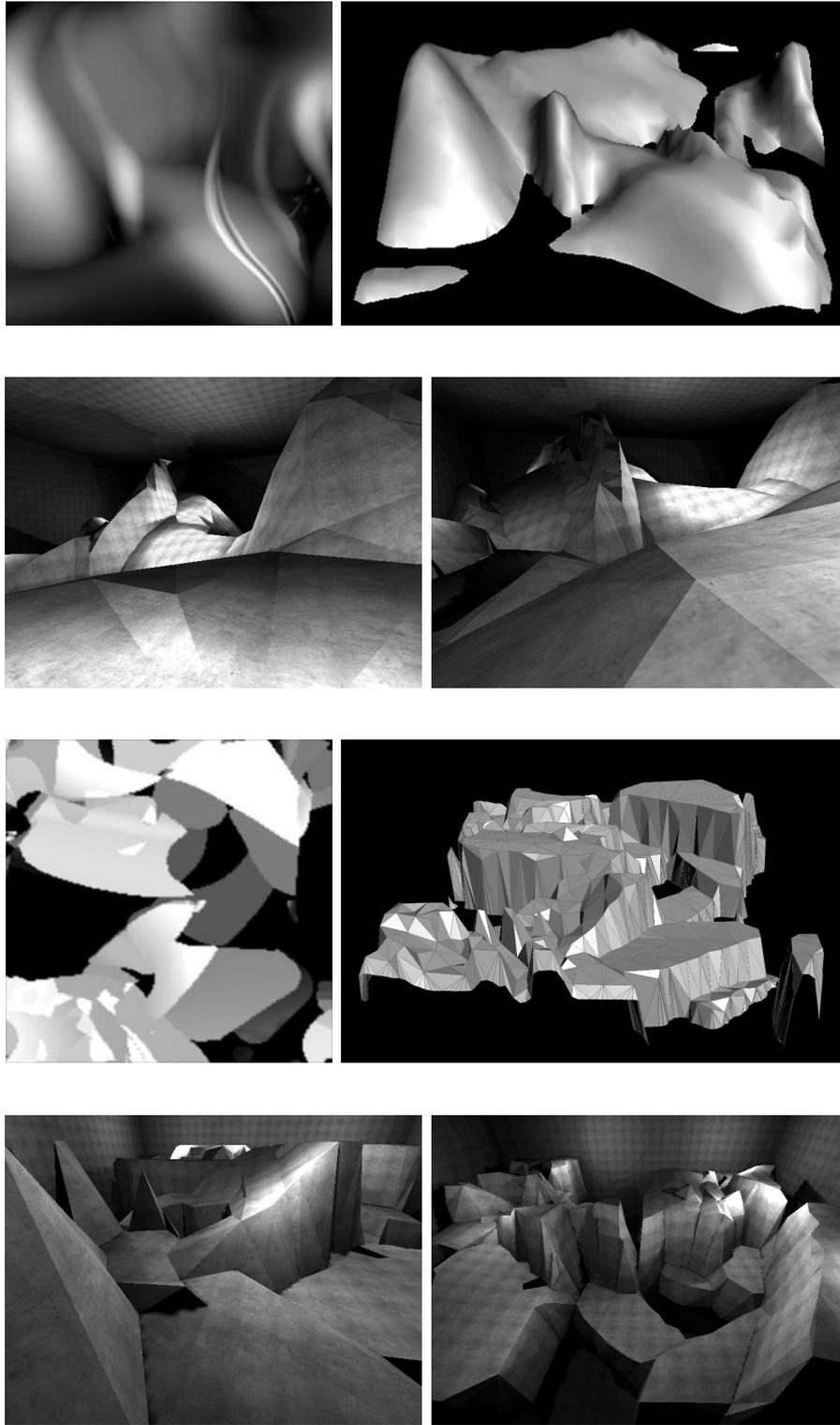


Figure 6: The first and third rows show an evolved image (left) reinterpreted as a height field (right). The second and fourth rows show different views of the height fields imported into Quake III ArenaTM (a product of id Software, Inc.)

A few directions for future work include the generation and inclusion of texture and lighting in the environment, implementing learning algorithms to correlate the user's preferences with genetic values, and improved interfaces to let the user manipulate and constrain individual gene values.

References

- [1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000.
- [2] Edward J. Bedwell and David S. Ebert. Artificial evolution of algebraic surfaces. *Proceedings Implicit Surfaces '99*, 1999.
- [3] Peter J. Bentley. *Evolutionary Design by Computers*. Morgan Kaufmann, 1999.
- [4] Jules Bloomenthal, editor. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.
- [5] Paul Coates. Using Genetic Programming and L-Systems to Explore 3D Design Worlds. In R. Junge, editor, *CAAD Futures '97*. Kluwer Academic, Munich, 1997.
- [6] Sumit Das, Terry Franguidakis, Michael Papka, Thomas A. DeFanti, and Daniel J. Sandin. A genetic programming application in virtual reality. In *Proceedings of the first IEEE Conference on Evolutionary Computation*, volume 1, pages 480–484, Orlando, Florida, USA, 27-29 1994. IEEE Press.
- [7] Richard Dawkins. *The Blind Watchmaker*. Penguin Books, 1986.
- [8] Janine Graf and Wolfgang Banzhaf. Interactive evolution of images. In D. B. Fogel, editor, *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 53–65, 1995.
- [9] Mark W. Jones. Direct Surface Rendering of General and Genetically Bred Implicit Surfaces. In *Proceedings of the 17th Annual Conference of Eurographics (UK Chapter)*, Cambridge, pages 37–46, 1999.
- [10] Matthew Lewis. Aesthetic evolutionary design with data flow networks. In *Proceedings of Generative Art 2000, Milan, Italy*, 2000.
- [11] Matthew Lewis. *Creating Continuous Design Spaces for Interactive Genetic Algorithms with Layered, Correlated, Pattern Functions*. PhD thesis, Ohio State University, 2001.
- [12] Henrik Hautop Lund, Luigi Pagliarini, and Orazio Miglino. Artistic design with genetic algorithms and neural networks. In J. T. Alander, editor, *Proceedings of 1NWGA, University of Vaasa, Vaasa*, 1995.
- [13] Stephen F. May. RManNotes. <http://www.cgrg.ohio-state.edu/~smay/RManNotes>, 2001.
- [14] Hiroaki Nishino, Hideyuki Takagi, and Kouichi Utsumiya. A Digital Prototyping System for Designing Novel 3D Geometries. In *6th International conference on virtual systems and multimedia (VSMM2000)*, Ogaki, Gifu, Japan, pages 473–482, 2000.
- [15] Tore Nordstrand. Surfaces. <http://www.uib.no/People/nfytn/surfaces.htm>, 2001.

- [16] Una-May O'Reilly and Girish Ramachandran. A preliminary investigation of evolution as a form design strategy. In C. Adami, R. Belew, H. Kitano, and C. Taylor, editors, *Artificial Life VI, Los Angeles, June 26-29*. MIT Press, 1998.
- [17] Darwyn Peachey. Building procedural textures. In David Ebert, editor, *Texturing and Modeling: a Procedural Approach*, chapter 2, pages 7–96. Academic Press, 1998.
- [18] Ken Perlin. An image synthesizer. *ACM Computer Graphics*, 19(3), 1985.
- [19] Ken Perlin. Noise, hypertexture, antialiasing, and gestures. In David Ebert, editor, *Texturing and Modeling: a Procedural Approach*, chapter 9, pages 209–274. Academic Press, 1998.
- [20] Pixar Animation Studios. Photorealistic renderman. <http://www.pixar.com>, 2001.
- [21] QBeo, Inc. PhotoGenetics. <http://www.qbeo.com>, 2001.
- [22] M. A. Rosenman. An exploration into evolutionary models for non-routine design. In D. Dasgupta and Z. Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 69–86. Springer-Verlag, 1997.
- [23] Andrew Rowbottom. Evolutionary art and form. In Peter J. Bentley, editor, *Evolutionary Design by Computers*, chapter 11, pages 261–277. Morgan Kaufmann, 1999.
- [24] Side Effects Software, Inc. Houdini. <http://www.sidefx.com>, 2001.
- [25] Karl Sims. Artificial evolution for computer graphics. *ACM Computer Graphics*, 25(4):319–328, 1991.
- [26] Hideyuki Takagi. Interactive Evolutionary Computation: Fusion of the Capabilities of EC Optimization and Human Evaluation. *Proceedings of the IEEE*, 89(9):1275–1296, September 2001.
- [27] Stephen Todd and William Latham. *Evolutionary Art and Computers*. Academic Press, 1992.
- [28] Jeffrey Ventrella. Tweaks. <http://www.ventrella.com>, 2000.