Diploma de Estudios Avanzados

Genetic Terrain Programming

Miguel Monteiro de Sousa Frade

Universidad de Extremadura

June 2008, Spain

Organization :
Dept. de Tecnologías, Computadores y Comunicaciones
Escuela Politécnica, Universidad de Extremadura

Title :
Genetic Terrain Programming

Author :
Miguel Monteiro de Sousa Frade
School of Technology and Management, Polytechnic Institute of Leiria, Portugal

Supervising Teacher :
Francisco Fernandéz de Vega
Centro Universitario de Merida, Universidad de Extremadura, Spain

Co-supervising Teacher :
Carlos Cotta
ETSI Informática, Campus de Teatinos, Universidad de Málaga, Spain

Line of Investigation :
Techniques for Planning the Improvement of the Efficiency of Evolutionary Algorithms
Técnicas de Planificación para la Mejora de la Eficiencia de Algoritmos Evolutivos

Period 2007/2008

Abstract

Nowadays there are a wide range of techniques for terrain generation, but all of them are focused on providing realistic terrains, often neglecting other aspects (e.g., aesthetic appeal or presence of desired features). This thesis presents a new technique, GTP (Genetic Terrain Programming), based on evolutionary design with Genetic Programming. The GTP technique consists of a guided evolution, by means of Interactive Evolution, accordingly to a specific desired terrain feature or aesthetic appeal. This technique can yield both aesthetic and real TPs (Terrain Programmes) which are capable of generating different terrains, but consistently with the same features. TPs are also scale invariant, meaning that terrain features will be preserved across different LODs (Levels Of Details), which allows the use of low LODs during the evolutionary phase without compromising results. Additionally, the resulting TPs can be incorporated in video games, like any other procedural technique, to generate terrains. Furthermore, by way of resorting to several TPs to compose the full landscape, it is possible to control some localised terrain features, thus eliminating the main drawback of traditional procedural techniques. The combination of GP with evolutionary art systems also diminish the effort and time required to create complex terrains when compared to modeling techniques. Moreover, the results are not dependent on the designer's skills.

List of Publications

- Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Modelling video games' landscapes by means of genetic terrain programming - a new approach for improving users' experience. In *EvoWorkshops 2008*, volume 4974, pages 485-490, Napoli, Italy, 2008. Springer [1]

- Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Genetic Terrain Programming - An Aesthetic Approach to Terrain Generation . In *Computer Games and Allied Technology 08*, pages 1-8, Singapore, 2008. (to appear) [2]

- Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. GenTP – Uma Ferramenta Interactiva para a Geração Artificial de Terrenos. In *3^{rd} Iberian Conference in Systems and Information Technologies (CISTI 2008)*, pages 1-12, Ourense, Spain, 2008 (to appear) [3]

Acknowledgements

I wish to express my sincere thanks and appreciation to my supervisors, Dr. Francisco Fernandez de Vega and Dr. Carlos Cotta, for their attention, guidance, insight, support, constructive comments and suggestions during the development of this research.

I cannot end without thanking my son Manuel and my daughter Carolina for their patience and understanding. Lastly, and most importantly, I wish to deeply thank my wife Céu for her constant encouragement and love throughout this journey. To them I dedicate this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

EA	Evolutionary Algorithms
EC	Evolutionary Computation
EP	Evolutionary Programming
Eq.	Equation
ES	Evolutionary Strategies
GA	Genetic Algorithm
GenTP	Generator of Terrain Programmes
GP	Genetic Programming
GTP	Genetic Terrain Programming
GUI	Graphical User Interface
IEC	Interactive Evolutionary Computation
NEvAr	Neuro Evolutionary Art
NPCs	Non-Player Characters
REC	Random Ephemeral Constant
SBART	Simulated Breeding ART
TP	Terrain Program

Chapter 1

Introduction

Video games constitute a crucial area of the entertainment industry, with impressive financial investments to make them more appealing and interesting. Game players seek continually for more enjoyable games as they spent 3.7 days per week playing an average of 2.01 hours per day [4]. Entertainment industry wants to maintain or increase this user's interest. In order to achieve richer human-machine interaction video games must be dynamic, they need to present game players with novel plots, intelligent artificial opponents, different goals and even scenario changes. Artificial terrain generation algorithms have an important role in the video games' dynamics. They help keep the user interested in playing by providing new landscapes. Video games are the most common application for artificial terrain generation algorithms, but they are also used in other fields such as movies, simulations and training environments (see Section 2.1).

Nowadays there are a wide range of techniques for terrain generation, which are presented in Sections 2.4 and 2.5, but all of them have key constrains. More elaborated methods depend highly upon designer's skills, time and effort to obtain acceptable results, and can not be used to automatically generate terrains in real time. The simpler methods allow only a narrow variety of terrain types and usually require a second algorithm (e.g. erosion algorithms) to provide a more natural looking result. Another key limitation of most techniques is the lack of control over localised terrain features.

A common feature with current techniques is that they all try to generate realistic terrains. Although this is an important aspect, imagine the possibility of a game designer to evolve their own terrain accordingly to his aesthetic feelings. This can lead to the generation of more exotic terrains at the cost of realism, but might also give players a reaction of awe and increase their interest in playing. This report presents a new technique, that we designated as Genetic Terrain Programming, that allows the evolution of TPs (Terrain Programmes) based on aesthetic evolutionary design with Genetic Programming (GP).

A TP is a mathematical expression that can generate height maps. Due to the incorporated randomness, each execution will generate a different terrain, but that will always present the same features. So, a TP can be executed like other procedural techniques and as many times as desired.

The use of evolutionary art systems with GP will allow the creation of both aesthetic and real terrains (without requiring a database of real terrain data). Additionally some control over localised terrain features will be possible trough the use of several TPs to compose the full landscape, which is the main drawback of the current procedural techniques. It will also require less effort and time than modeling techniques to create complex terrains and the result is not solely dependent on the designer's skills.

This document includes two background chapters, Chapter 2 and 3. Chapter 2 presents what have been done in the field of terrain generation and a terrain generator taxonomy. Chapter 3 introduces the evolutionary algorithms paradigm, their characteristics and applications, with emphasis on video games and evolutionary design. An introduction to GP is also included. Our technique, Genetic Terrain Programming, its implementation details and results are exhibited on Chapter 4. Finally, conclusions and future work are presented on Chapters 5 and 6.

Chapter 2

Artificial Terrains

Artificial terrain generation has been addressed by several researchers for a long time. This chapter presents what have been done in this area and starts to present some examples of terrain generation applications in Section 2.1. Section 2.2 introduces a number of traits that an ideal terrain generation algorithm would have to establish a base line of comparison with real algorithms. Then different options of data structure to represent terrains models, their benefits and short comings are presented in Section 2.3. The existing generation techniques are addressed in two sections, Section 2.4 deals with the traditional generation techniques, which are the ones most commonly used, and Section 2.5 presents the newer methods for terrain generation based on evolutionary algorithms. Finally, some considerations are made regarding how different generation techniques approaches the LOD (Level Of Detail) problematic in Section 2.6.

2.1 Applications

Terrain generation is used for a broad range of applications in many fields, from computer generated art and animation, to video games , virtual reality and architecture. The following list presents some examples:

- virtual tourism and travel planning;
- geographical and general reference education (e.g. the Virtual Field Course[1]);

[1]http://www.geog.le.ac.uk/vfc/

- land planning, usage and urban planning;
- civil engineering, utilities, and other infrastructure;
- visualisation of weather and other environmental attributes;
- real estate (e.g. Sense8 virtual real estate demo [2]);
- diplomacy (e.g. pilots study virtual terrain in Bosnia[3]);
- radio, tv and cellular transmitter placement and signal analysis (e.g. Radio Mobile[4]);
- public participation in land management (e.g. Virtual reality solutions to coastal erosion [5]);
- media and advertising;
- security and defence industry;
- games and entertainment;

2.2 Ideal Terrain Generator

In order to compare existing generation techniques R. Saunders [5] proposed a list with the desired properties of a perfect terrain generator:

- low requirements of human input;
- allow a high degree of human control;
- to be completely intuitive to control;
- produce models at arbitrary levels of detail;
- fast enough for real-time applications;
- to be able to generate a wide variety of recognisable terrain types and features;
- to be extensible to support new types of terrain;

2.2.1 Low Requirements of Input

Digital terrain models typically involves large amounts of data. For example, to represent a terrain with a grid of points with 10 meters of distance between points requires more than 10.000 data points per square kilometer. The manual placement of all this data points would be a tedious and lengthy task. Therefore, the automated construction of details, without compromising the

[2]http://www.sense8.com/gallery/data_visual.html
[3]http://www2.cnn.com/TECH/9512/virtual_bosnia/index.html
[4]http://www.cplus.org/rmw/english1.html
[5]http://www.edp24.co.uk/content/news/2002/06/asp/020605coasterode.asp

designer control, should result in a productivity increase. An ideal terrain generation methodology would accept a command like this “give me a 10 km by 5 km area of desert-like terrain with a resolution of 10m per sample” and produce a believable result. A low input requirement would also be useful for generating unique terrains automatically , e.g. random terrain generator for a game.

2.2.2 High Degree of Control

To give designers the full creative freedom requires the ability to control the behaviour of the terrain across all scales. From the macro-scale features (e.g. hills, valleys, mountains) to the micro-scale details (e.g. cracks, crevices) with the desired localisation. An ideal algorithm would permit the artist to exert whatever amount of control he wishes, where he wishes, while intelligently filling in the details he does not want to specify directly. Obviously this goal conflits with the previous one of low requirements of human input.

2.2.3 Intuitive to Control

An ideal terrain generation algorithm would be perfectly intuitive to use, without requiring knowledge of the algorithm’s operation. All of the inputs available to the user would be easily understood: even an inexperienced user would have a reasonable idea of how tweaking an input would affect the generated terrain.

2.2.4 Arbitrary Levels of Detail

An ideal terrain generation algorithm would permit the creation of terrain at multiple levels of detail, and the segmentation of large terrains into regions of differing levels of detail. Full detail terrain models are not always needed, the required level of detail is determined by their intended use. For example, a computer animated film needs a highly realistic detail of the terrain model only in the areas near to which the action is to take place, and can use relatively simple geometry for the distant terrain. Similarly, the level of detail required for the terrain in a flight simulation is significantly smaller than what is needed for ground level action.

2.2.5 Fast Enough for Real Time Applications

Many terrain-related algorithms are very slow, or else require a lengthy preprocessing phase before the terrain can be used in a real-time application. Only the simplistic terrain generation algorithms are able to generate terrains in real time. However, this feature can provide several important advantages, including:

- instantaneous feedback regarding a particular modification;
- the ability to modify dynamically the terrain, such as transforming a mountain into a crater, or rapidly eroding a riverbed into a deep gorge;
- if the full terrain can be (re)generated from a compact set of parameters, the required memory and disk storage will be significantly less demanding;
- the ability to create seamless, infinite worlds, with arbitrarily fine, dynamic level of detail;

2.2.6 Capable of Diverse Features and Terrain Types

An ideal algorithm would be able to create a wide diversity of terrain types (e.g., desert, mountains, plains) and features (e.g. ravines, riverbeds, volcanoes), both with realistic look and from the imagination or aesthetic appeal of the artist. The transitions between different types of terrain (e.g., from mountains to foothills) would be credible. Additionally, the placement of semantic features would make sense, e.g. waterfalls would pour into pools, rivers would always flow downhill.

2.2.7 Extensible

The digital creations of a designer or artist are only limited by their imagination. Therefore, regardless of how many types of terrain a terrain generation tool can create, it is not possible to cover all the possibilities. Thus, an ideal terrain generation algorithm would be extensible in some way, allowing new types of terrain to be introduced easily.

2.3 Terrain Representation

Before a terrain generator is developed, it is necessary to define how to represent a terrain. The chosen data structure will influence the way the terrain is build, the available tools to manipulate it and might affect also the terrains features that can be represented. Regarding these topics there are several considerations:

- the terrain must be rendered at any arbitrary scale, or is it acceptable to have a finite maximal resolution?
- is it necessary to represent all terrain features where there are multiple surfaces for the same horizontal coordinates, like caves and overhangs?
- is it necessary to account for planetary curvature, or a flat approximation is enough?
- is it necessary to perform collision detection in an efficient way?

Having this considerations in mind, several alternative data structures to represent terrains are analysed:

- height maps;
- voxel grids;
- non-uniform meshes;
- analytical and fractal functions;

2.3.1 Height Maps

Height maps are probably the most common method used to represent terrains. Formally, a height map is a scalar function of two variables, such that for every coordinate pair (x, y) corresponds an elevation value h, as shown in Eq. (2.1). In practice a height map is a two-dimensional, rectangular grid of height values, where the axis values are spaced with regular intervals valid over a finite domain (see Figure 2.1). The most common data structure to represent them are 2D arrays filled with the elevations values.

$$h = f(x, y) \quad . \tag{2.1}$$

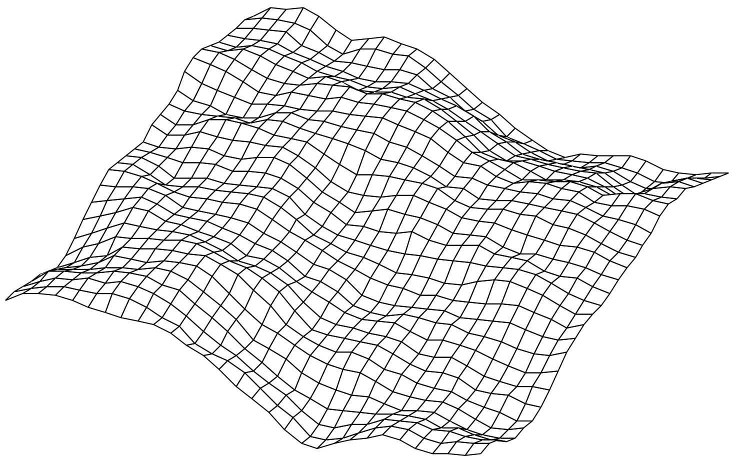


Figure 2.1: A discrete height map example

Advantages

The height maps' regular structure is their main advantage, it allows the optimisation of operations such as rendering, collision detection and path finding. The render of huge height maps in real-time is now possible due to the creation of several continuous level of detail (CLOD) algorithms [6–8], which render highly visible areas of the terrain with detailed geometry, using progressively simpler geometry for more distant parts of the terrain. Collision detection is greatly simplified if one of the objects is a height map, because only a few surrounding triangles need to be checked for collision.

If the values of a height map are normalised it becomes the same thing as grey scale image. This means that image processing and computer vision techniques may be used to construct, modify and analyse terrain models represented as height maps. For example, a height map can be stored, imported or exported using an image file format, or a filter can be applied to smooth a rough terrain.

Finally, Geographic Information Systems (GIS) use height maps to represent real world terrain, which are commonly built using remote sensing techniques such as satellite imagery and land surveys. This is another advantage due to the significant amount of real world terrain models available to work with.

Disadvantages

The main limitation of height maps is the inability to represent structures where multiple heights exist for the same pair of coordinates. So height maps are inherently unable to represent caves, overhangs, vertical surfaces, and other terrain structures in which multiple surfaces have the same horizontal coordinates. Fortunately, only a small percentage of natural terrain fall into this category and this limitation can be overcame by using separate objects placed on top of the terrain model.

A second disadvantage of height maps is that it has a finite uniform resolution, which means there is no simple way to handle a terrain with different local levels of details. If the resolution is chosen to match the average scale of the features in the terrain, then any finer-scale features will be simplified or eliminated. Conversely, if the resolution is chosen to be high enough to capture the fine-scale features, areas containing only coarse features will also be captured at this same high resolution, an undesirable waste of space and processing time. Ideally, a terrain representation for terrain generation would either be infinite in resolution, or else would adaptively increase its resolution to accommodate the addition of fine scale details, rather than requiring an a priori decision about resolution.

A third disadvantage of height maps is its inadequacy to represent terrain on a planetary scale. Rectangular height maps do not map directly to spheroid objects, usually a two-pole spherical projection is used. In those cases the density of height field points will be substantially greater in areas near the poles than at those near the equator.

2.3.2 Voxel Grids

A voxel grid is a discrete three-dimensional grid of volumetric pixels (voxels) where each voxel is filled or not. This structure allow the representation of arbitrary 3D shapes.

Advantages

The voxel grids' advantage over height maps is their ability to represent any terrain structures like caves, overhangs and vertical surfaces.

Disadvantages

Voxel grids share the same disadvantages of height maps, such as finite resolution and inability to gracefully handle planetary curvature. Additionally, operations like rendering and collision detection consumes more processor power and memory than height maps.

2.3.3 Non-Uniform Meshes

The terrain surface can be represented as an arbitrary mesh of 2D primitives, usually polygons, in the 3D space. This is a more general representation of 3D objects and there are several tools to work with this representation. A special case of a non-uniform mesh is TIN (triangular irregular network) [9]. A TIN is a vector based representation of a physical surface made up of irregularly distributed nodes and lines with three dimensional coordinates (x, y, z) that are arranged in a network of nonoverlapping triangles. TINs are often derived from the elevation data of a rasterized digital elevation model (DEM).

Advantages

The main benefit of using non-uniform meshes to model the terrain surface is that they are extremely general. The surface may have arbitrary geometry (overhangs, caves, etc.). This is the most common paradigm used in 3D tools and allows an artist to freely model any arbitrary 3D object using a single modeling paradigm. Furthermore, there are a significant amount of available tools to work with, see Section 2.4.

A second advantage of using a TIN over a height map is that the points of a TIN are distributed variably based on an algorithm that determines which points are most necessary to an accurate representation of the terrain. So, they naturally support variable level of detail, allowing more vertices in areas of sharp change and relatively few vertices in flat areas. As a result, a mesh structure can store some terrain models much more efficiently than regular grid methods, since it does not require a globally high resolution in order to achieve fine-scale features in a few places.

Disadvantages

The main problem when using meshes for terrain generation is that it is not clear how to generate them automatically. Although a terrain is always tessellated into polygons before rendering, to best of our knowledge, there are no methods (other than manual sculpting) to directly generate a non-uniform mesh.

2.3.4 Analytical and Fractal Functions

Another way to represent terrains is through the use of fractal functions or analytic expressions. This approach is not used often, being MojoWorld [6] and GenTP (see Chapter 4) two examples.

Advantages

The main advantage of analytical and fractal functions is the ability of being displayed at any scale without losing resolution. Due to the continuous nature analytical functions it is possible to recalculate the terrain so it does not look faceted when viewed close-up, like height maps tend to do. Some analytical functions are render friendly and/or collision detection, others, such as polynomial surfaces of low degree (quadratic and below), allows ray/surface intersections to be calculated in a straightforward way. However, terrains produced by analytic functions tend to become more and more linear when enlarged. On the other hand, fractals functions continue to produce new details as they are evaluated progressively at finer scales.

Disadvantages

Modeling with analytical functions is a great challenge, if a single, global function is used, it is difficult to know how to modify it to achieve a certain local effect. A more common approach is the use of several functions to compose a full landscape, were B-spline patches are an example. Fractals present the same disadvantage due to the few input variables to control the output.

[6] http://www.pandromeda.com/products/

Another problem is the complexity to render the terrain directly from the functions, because ray tracing systems and hardware were built to work with polygon-based rendering. One way to address this issue is the introduction of an additional stage were the function is converted to another form of terrain representation, like height maps, before rendering, but with performance costs.

2.4 Traditional Generation Techniques

Traditional terrain generation techniques can be divided in three main categories:

- Measuring;
- Modeling;
- Procedural;

2.4.1 Measuring

In the measuring techniques elevation data is derived from real-world measurements to produce Digital Elevation Models (DEM), commonly built using remote sensing techniques such as satellite imagery and land surveys [7]. This is the most common basis for digitally-produced relief maps. Measuring has the advantage of producing highly realistic terrains with very little human effort, but at the expense of designer control. If the designer has specific goals for the the terrain's design and features (e.g. mountains, valleys, lakes) this approach may be very time-consuming, as the designer might have to search extensively to find real-world data that meets his specific criteria.

2.4.2 Modeling

Modeling is by far the most flexible technique for terrain generation. A human artist models or sculpts the terrain morphology manually using a 3D modeling program (e.g. Maya [8], 3D Studio [8], or Blender [9]), or a specialised

[7] http://rockyweb.cr.usgs.gov/nmpstds/demstds.html
[8] http://www.autodesk.com/fo-products
[9] http://www.blender.org

terrain editor program (e.g. the editors that ship with video games like Unreal Tournament 2004 [10], SimCity 4 [11] or SimEarth [12]). The way the terrain is built is different depending on the features provided by the chosen editor, but the general principle is the same. With this approach the designer has unlimited control over the terrain design and features, but this might be also a disadvantage. By delegating most or all of the detail up to the designer, these technique imposes high requirements on the designer in terms of time and effort. Also the realism of the resulting terrain is fully dependent on the designer's skills.

2.4.3 Procedural

Procedural techniques are those in which the terrains are generated programmatically. These category can be divided into the following techniques:

- Physical;
- Spectral synthesis techniques;
- Fractals;

Physical

Physically-based techniques simulate the real phenomena of terrain evolution trough effects of physical processes such as erosion by wind [13], water [10], thermal [11], or plate tectonics. These techniques generate highly realistic terrains, but require an in-depth knowledge of the physical laws to implement and use them effectively. Physically-based techniques are also very demanding in terms of processing power.

Spectral synthesis techniques

Another procedural approach is the spectral synthesis. This technique is based on the observation that fractional Brownian motion (fBm) noise has a well defined power spectrum. So random frequency components can be easily calculated and then the inverse Fast Fourier Transform (FFT) can

[10]http://www.mobygames.com/game/unreal-tournament-2004
[11]http://simcity.ea.com/about/simcity4/overview.php
[12]http://www.mobygames.com/game/simearth-the-living-planet
[13]http://www.weru.ksu.edu/weps.html

be computed to convert the frequency components into altitudes [11]. This technique does not allow the designer much control on the outcome of terrains features.

Fractals

Self-similarity is the key concept behind any fractal technique. An object is said to be self-similar when magnified subsets of the object look like (or identical to) the whole and to each other [12]. Terrain falls into this category (to a limited extent). The jagged edge of a broken rock has the same irregularities as a ridgeline on a distant horizon. This allows the use of fractals to generate terrain which still looks like terrain, regardless of the scale in which it is displayed [13]. Every time these algorithms are executed they generate a different terrain due to the incorporated randomness. This class of algorithms is the favourite one by game's designers, mainly due to their speed and simplicity of implementation. There are several tools available that are predominantly based on fractal algorithms, such as Terragen [14] (which is a hybrid fractal/modeling tool) and GenSurf [15] (a mapping tool for Quake 3 Arena video game). However, generated terrains by this techniques are easily recognised because of the self-similarity characteristic of fractal algorithms. Although these algorithms present some parameters that can be tweaked to control, e.g the roughness, the designer does not have control on the resulting terrain features.

Due to the importance of these techniques in entertainment industry, we will explain the *Midpoint Displacement Algorithm* and the *Diamond-Square Fractal Algorithm*.

Midpoint Displacement Algorithm

One of the most popular method for automatic terrain generation is the diamond-square algorithm. This fractal algorithm is a variation of the random midpoint displacement algorithm [14] applied to two dimensions. The roughness constant, H, is the value which will determine the roughness of the resulting fractal. This constant can take values from 0 to 1 and is used to calculate the scaling factor $f = 2^{-H}$. The resulting fractal will be very smooth with $H = 1$ and very jagged with $H = 0$ [13].

To explain how the random midpoint displacement algorithm works, lets

[14]http://www.planetside.co.uk/terragen
[15]http://tarot.telefragged.com/gensurf

consider its use on just one dimension. If $X(t)$ is to be computed for times t between 0 and 1, then it may be started by setting $X(0) = 0$ and $X(1)$ as a sample of a Gaussian random number. Next $X(\frac{1}{2})$ is computed as $\frac{1}{2}(X(0)+X(1))+D_1$, where D_1 is a Gaussian random number which should be multiplied by a scaling factor of $\frac{1}{2}$. On the next iteration the scaling factor is reduced by $\frac{1}{\sqrt{8}}$ $(H = 0.5)$ and the two intervals $\left[0, \frac{1}{2}\right]$ and $\left[\frac{1}{2}, 1\right]$ are divided again. $X(\frac{1}{4})$ will be set to $\frac{1}{2}(X(0)+X(\frac{1}{2}))+D_{2,1}$, where $D_{2,1}$ is a Gaussian random number which should be multiplied by $\frac{1}{\sqrt{8}}$ (the current scaling factor). The same reasoning is applied to $X(\frac{3}{4})$ where $D_{2,2}$ is computed the same way as $D_{2,1}$. This process can be repeated as many times as desired [12]. Figure 2.2 shows three runs of the random midpoint displacement algorithm with different roughness (H) values.

The same reasoning is applied to $X(\frac{3}{4})$ as showed in Formula 2.2, where $D_{2,2}$ is computed the same way as $D_{2,1}$ [12].

$$X\left(\frac{3}{4}\right) = \frac{1}{2}\left(X\left(\frac{1}{2}\right) + X\left(1\right)\right) + D_{2,2} \ . \tag{2.2}$$

The third stage proceeds in the same way: reduce the scaling factor by $\sqrt{2}$, which will be $\frac{1}{\sqrt{16}}$, and set the values of $X(\frac{1}{8})$, $X(\frac{3}{8})$, $X(\frac{5}{8})$ and $X(\frac{7}{8})$ in a similar fashion to Formula 2.2.

$$\begin{aligned}
X\left(\frac{1}{8}\right) &= \frac{1}{2}\left(X\left(0\right) + X\left(\frac{1}{4}\right)\right) + D_{3,1} \\
X\left(\frac{3}{8}\right) &= \frac{1}{2}\left(X\left(\frac{1}{4}\right) + X\left(\frac{1}{2}\right)\right) + D_{3,2} \\
X\left(\frac{5}{8}\right) &= \frac{1}{2}\left(X\left(\frac{1}{2}\right) + X\left(\frac{3}{4}\right)\right) + D_{3,2} \\
X\left(\frac{7}{8}\right) &= \frac{1}{2}\left(X\left(\frac{3}{4}\right) + X\left(1\right)\right) + D_{3,2} \ .
\end{aligned} \tag{2.3}$$

Diamond-Square Fractal Algorithm

The midpoint displacement algorithm, described in previous section, can be extrapolated into 3D space, one of those extrapolations is called diamond-square algorithm. A two-dimensional array of height values is needed, they will map indices (x and y values) into height values (h). Although the end goal is to generate three-dimensional coordinates, the array only needs to store the height (h) values. The algorithm starts with a large empty 2D

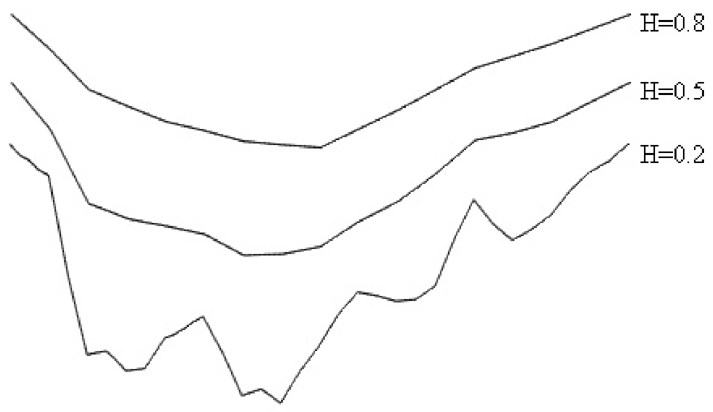


Figure 2.2: Execution examples of the random midpoint displacement algorithm

square array of points. The dimension of this array should be $d = 2^n + 1$, where n is the number of desired interactions and the scaling factor should now be $f = \sqrt{2^{-H}}$ [12]. The four corner points should be set to the height value and form a square. This is the starting-point for the two steps iterative subdivision routine [14] (see Figure 2.3):

- *The diamond step*: Taking a square of four points, generate a random value at the square midpoint, where the two diagonals meet. The midpoint value is calculated by averaging the four corner values, plus a random amount. This gives diamonds when it has multiple squares arranged in a grid;
- *The square step*: Taking each diamond of four points, generate a random value at the centre of the diamond. Calculate the midpoint value by averaging the corner values, plus a random amount generated in the same range as used for the diamond step. This will generate squares again;

2.5 Evolutionary Generation Techniques

All traditional generation techniques present some kind of limitation, in an attempt to overcome them new techniques were developed based on evolutionary algorithms. Evolutionary algorithms (EA) are a kind of bio-inspired algorithms that apply the Darwin's theory [15] (see Chapter 3).

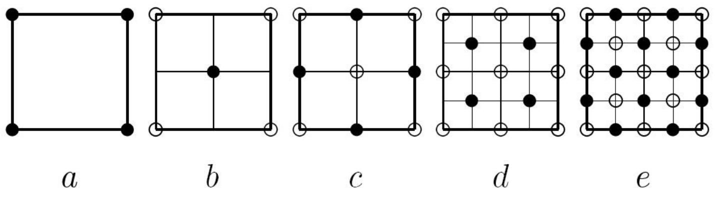


Figure 2.3: Two iterations of the diamond-square algorithm. Pseudo random number are used for initial values in step a. In step b (the "diamond" step) a new value is found by offsetting the average of the four values of step a. Step c (the "square" step) fills in the rest of the midpoint values also by offsetting the average of the four neighbours of each new point. Steps d and e show the next iteration [11].

To the best of our knowledge, Teong Ong et al. [16] were the first authors to propose an evolutionary approach to generate terrains. They proposed an evolutionary design optimisation technique to generate terrains by applying genetic algorithms to transform height maps in order to conform them to the required features. Their approach breaks down the terrain generation process into two stages: the terrain silhouette generation phase, and the terrain height map generation phase. The input to the first phase is a rough, 2D map laying out the geography of the desired terrain that can be randomly generated or specified by the designer. This map is processed by the first phase to remove any unnaturally straight edges and then fed to the second phase, along with a database of pre-selected height map samples representative of the different terrain types. The second phase searches for an optimal arrangement of elevation data from the database that approximates the map generated in the first phase. Since the height map generation algorithm is inherently random, the terrains generated from two separate runs of the algorithm will not be the same, even if they use the same map. While this has the benefit of allowing an infinite number of variations to be created, it could also inhibit the designer's creation process, as each successive run would produce an entirely new terrain, even if the designer had made only a small tweak to the map between runs. To control this, the seed for the random number generator can be kept the same across separate runs of the algorithm, allowing the same terrain to be regenerated as many times as desired.

We proposed a new evolutionary approach designated GTP (Genetic Terrain Programming) [1], which is detailed in Chapter 4. Our approach consists on the combination of evolutionary art systems with GP to evolve mathematical expressions, designated TPs (Terrain Programmes), to generate artificial terrains.

2.6 Level of Detail

A detailed terrain model involves a huge amount of polygons to be represented, even when considering only the portion of the scene that is visible. Clark [17] suggested using simpler versions of the geometry for objects that had lesser visual importance, such as those far away from the viewer. These simplifications are called Levels of Detail (LODs) and allows the adaptation of structures, such as terrains, to the processing power requirements. However, when generating terrains LOD acquires a higher importance. On the rendering phase LOD can be considered as the upper limit on the amount of detailed to be shown. However, on the generation phase LOD will have a huge effect on the shape and features of the generated terrain. As the LOD of a terrain is reduced, the finer details are lost, leaving only the larger features. Due to this result the details that disappeared are considered as belonging to the finer LOD. Regarding LOD we can divide generations methods in three classes:

- LOD-agnostic;
- Superposition;
- Progressive refinements;
    - Fractals;

The following subsections describe those generation methods.

2.6.1 LOD-agnostic

Some generation methods' approach to LOD by considering it only initially, by determining the resolution of the terrain model to be generated. These LOD-agnostic methods generate the terrain at once with the maximum detail.

Collaging/faulting methods [16] are an example of LOD-agnostic methods. In this method, a flat landscape is used as starting point and repeatedly add

[16] http://www.geocities.com/Area51/6902/t_fault.html

or subtract height from some portion of the bitmap. Usually a straight line is drawn across the bitmap and raise the left or the right part by one unit. This is repeated from 1.000 to 10.000 times with randomly placed lines of random slope to get some reasonably realistic mountains. The resolution of the height field is fixed at the start of the algorithm, and the collaging process runs to completion without explicit consideration for the LOD of generated features. This approach was used by Mandelbrot [18] in building early fractal landscapes. This is generally an expensive approach, although there are some methods that can be used to speed it up in some cases.

Another class of methods that is typically LOD-agnostic is the class of simulation methods. Simulation is usually done using numerical integration techniques on a regular grid structure, the initial choices of grid resolution and simulation time-step are normally the only points at which LOD considerations are consciously applied. Thereafter, the physical features of the terrain emerge through the approximation of physical processes, without further reference to LOD (though the initial choice of LOD will certainly affect the accuracy of the simulation).

2.6.2 Superposition

Superposition methods construct several layers with different scales and then add them together until the desired LOD is achieved. This approach is used in spectral synthesis methods (see Section 2.4) where each layer is composed by a finite slice of the frequency domain. The higher the frequency, the smaller are the details on the space domain. The recombination of these different scales into a spatial-domain height field is done by applying the appropriate inverse transform (Fourier or wavelet) from the frequency-domain representation [19].

Another example of a superposition method is Perlin Noise [20, 21] , in which random noise layers of different frequencies and amplitudes are added together; the choice of frequency and amplitude for each of the layers gives some measure of control over the overall characteristics of the resulting height map. Perlin Noise (and similar superposition methods) allow a user to achieve a greater variety of effects and allows him some degree of predictable control over the behaviour of the surface at different scales.

2.6.3 Progressive refinements

A third approach to LOD in terrain generation is to refine the terrain surface progressively, starting from a coarse LOD and adding finer and finer details until the desired LOD is reached. These methods differ from the superposition ones in that they generate new fine scale details as modifications to the coarser LODs. The use of recursive subdivision is common on these methods. They start with a coarse LOD of the terrain model and the next finer LOD is constructed by subdividing it and by adding small scale modifications. The Midpoint Displacement Method (MPD) is an example of this approach. The advantage of progressive refinement methods is that they can make use of the previously constructed and provide better guarantee of perseverance of macro terrain features without the need of any additional algorithm.

Fractals

Fractals are a special case of the progressive refinements category. Mandelbrot [18] observed that many objects in nature display the characteristic of self-similarity, having essentially the same structure when viewed at different scales. Saunders [5] summarises this behaviour quite well:

> Fractal algorithms answer the question, "What should the terrain look like at this scale?", with, "Just like it did at the larger scale, but smaller."

Fractals have been used in computer graphics to represent natural structures both regular (e.g plants and trees) and irregular (e.g terrains). Randomised fractal algorithms are a convenient method of creating irregular shapes across an entire range of LODs. Because each successively finer LOD has a well defined relationship to the one preceding it (except for the effects of the random numbers), fractal algorithms offer unlimited amount of detail.

However, not all terrain types exhibit the self-similar property across all scales. For example, both photos from Figure 2.4 are from Death Valley (CA, USA), but seen at very different scales. On top is a close-up of cracked dried mud in a creek from the Death Valley and at the bottom is a satellite image of the same region. As is easily verified, in this case there is no self-similarity between the two scales of these terrain photos.

Therefore, fractals cannot be used as general functions to generate terrains at different LODs, because there will be types of terrains that they

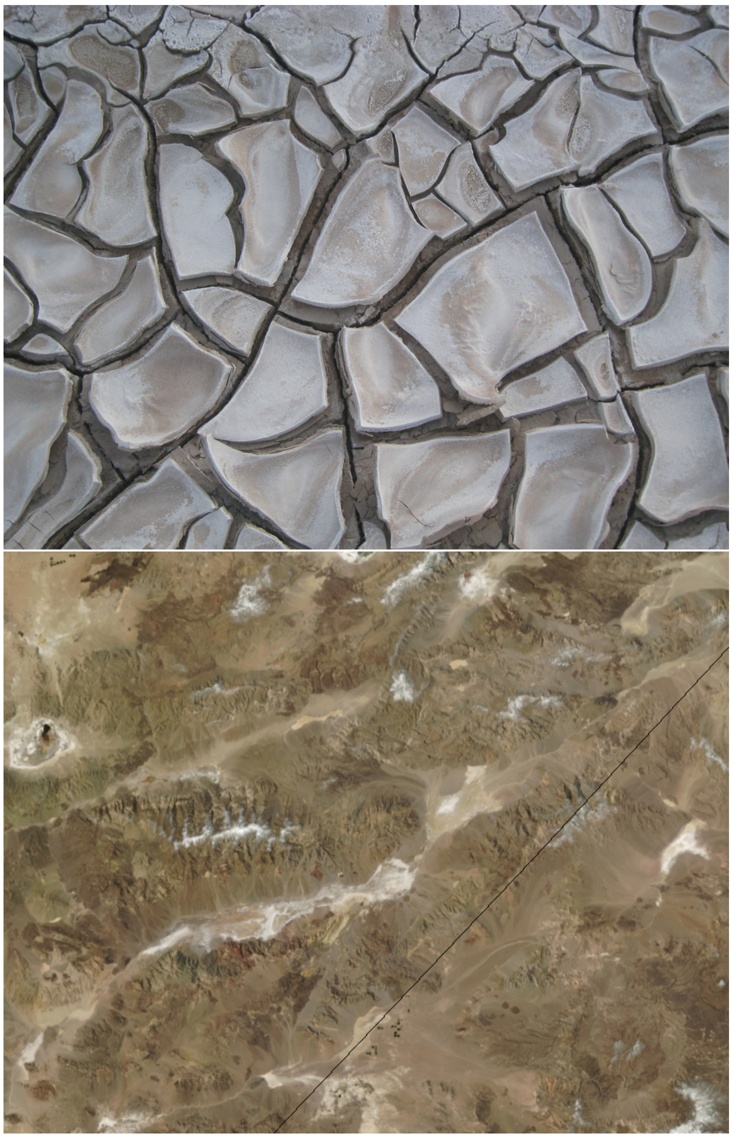


Figure 2.4: Images of Death Valley, on top "Cracked mud on the way to the borax haystacks", by *redteam* (Creative Commons license), bellow a satellite image from NASA (public domain). On this example there is no self-similarity between the two scales of this region

will be unable to reproduce. Any terrain generation algorithm that aims to support a broad range of terrain types at various scales must allow for the possibility of non-self-similar terrain.

Chapter 3

Evolutionary Algorithms

Evolutionary algorithms can be seen as search techniques [22]. They are able to achieve good approximate solutions to all types of problems, because they do not make any assumption about the underlying search landscape. This characteristic is the key factor of success in such diverse fields as engineering, art, biology, economics, marketing, genetics, operations research, robotics, social sciences, physics, and chemistry. Apart from their use as optimisers, evolutionary algorithms have also been used as an experimental framework to validate theories about biological evolution and natural selection, particularly through work in the field of artificial life [23].

Another filed were evolutionary algorithms have been used as an artificial intelligence technique is video games. They provide competitive, dynamic environments that make them ideal test beds for computational evolutionary theories, architectures, and algorithms [24–26].

Given the increasing importance of the this entertainment industry, Section 3.1 provides an overview of the use of evolutionary algorithms on video games. Section 3.2 presents the concept of evolutionary design followed by related work on this field on Section 3.3. Finally, Section 3.4 explains how GP is implemented, which is used in our technique.

3.1 Video Games

Much of the early work on computational intelligence and games was directed toward classic board games, such as noughts and crosses [27], chess, and checkers [28]. Board games can now, in most cases, be played by a computer at a higher level than the best humans. For example the best computer checkers player is considered to be better than the world champion. The best computer chess players, despite Deep Blue beating Kasparov, are still not rated as highly as the world champion human players. But the day cannot be too far away when the best computer players can consistently beat the best human players [26]. Board games usually capitulate to brute force methods of search to produce the very best players. Go is an exception, and has so far resisted machine attack. The best Go computer players now play at the level of a good novice [29]. In recent years, researchers have been applying EC methods to evolve all kinds of game-players, including real-time arcade and console games (e.g., Quake, Pac-Man [30]). There are many goals of this research, and one emerging theme is using EC to generate opponents (NPCs - non-player characters) that are more interesting and fun to play against, rather than being necessarily superior.

3.1.1 AI agents

Human-centred games are limited by human mental capacity and dexterity. Video games, on the other hand, operate under no such constraints and typically have a more complex state space than even the most complex of human-centred games. This complexity encourages the development or evolution of more general purpose AI agents than the required for playing board or card games. John Laird [31] emphasised the challenge posed by real-time video games to the AI community, and this is now a rapidly growing field of research [26].

Currently, the majority of Game-AI agents are controlled through scripts, developed by game programmers. The use of computational intelligence techniques offers an alternative to scripted approaches, whereby the agent behaviour can be controlled using an evolved neural network, for example, rather than being programmed. This has two implications: firstly, programming the scripts can be laborious and time consuming and therefore expensive. Evolving the agent behaviours is financially advantageous. Second, an evolved agent may create novel strategies that game players may find especially entertaining to compete with.

Evolved agents tend to be excellent at exploiting loopholes in a game. Identifying and removing these loopholes is an important part of the game development life cycle, and one in which evolutionary computation is just starting to be used [32]. These are aspects of the game, whereby a player finds a fast way to win. The effect of this is that the player tends to lose respect for the game and game developers devote many hours of game playing to find these loopholes.

A major challenge for researchers is to develop effective learning algorithms that can adapt in real time within complex environments. To have game characters that exhibit such characteristic would make games much more interesting. For example, if a NPC learns from playing against the human player, then the human player may benefit in the long term by making deliberate mistakes early on. So far it is not known any game that use this type of learning [26]. Yannakakis [4] has created and used an objective measure framework of interestingness to evolve more engaging ghost behaviours for a simplified version of Pac-Man and other preypredator games.

With computational intelligence techniques the possibility of opening new genres arises. That has happened already in the work of Stanley et al. [33] in their NERO video game. In NERO, the object of the game is to train an army unit to fight as a team and, thereby, achieve the game goals. Each soldier has a neural network that is able to learn through experience. Soldiers who make more progress through the maze have more offspring than those who make less progress, and eventually good maze-solving behaviours can be evolved.

While the major recent trend has been toward designing NPCs for complex 3D games such as Quake, the challenges of older 2D arcade classics such as Pac-Man should not be underestimated. The advantage of working with older-style arcade games is that they offer a sufficiently interesting challenge, while being much easier to implement and faster to simulate. While there exists open-source first-person shooters, these can be quite technical to interface to, and present lengthy simulation times [26].

Researchers at DEMO (the Dynamical & Evolutionary Machine Organisation) have developed a web-based game designated Tron [34]. Tron uses GAs to evolve a better game-playing AI. Tron is a GA-based system that evolves strategies to play the light cycles game as depicted in the movie Tron. The evolving players are set in opposition against human players over the Web. Written entirely in Java, the project offers a variety of statistics over how well its playing style has improved over time. Later Funes and Pollack describe a statistics-based approach to compare players with each other developed in the context of the Tron system [35]. They used a co-evolutionary experiment that matches humans against agents and were able to show that the complex interactions led the artificial agents to evolve towards higher proficiency. At the same time, individual humans learned as they gained experience interacting with the system.

There are research areas, such as robotics, were measuring progress is a significant problem. This happens because of the complex multi-task and multi-objective goals they have to deal with. One way to measuring progress is through real world competitive games. The robotic football tournaments are a great success in this area in the form of RoboCup [36]. This tournaments allowed many research groups to work with a common focus, and progress can now be measured by the performance of current teams against previous teams. Real world robotic games are much harder than they first appear and involve dealing with noisy sensors and imperfect actuators, together with all the strategic issues of how to play the game in question.

Another example of real world games are the computer-controlled car racing. By using real-time video feed from a Web cam, the aim of the controller is to race the car around the track as fast as possible. This is hard enough in itself, but much harder and more interesting when racing two or more cars against each other at the same time. Ivan Tanev [37] hand designed a car control algorithm, then optimised the parameters with an EA. One the other hand, Togelius and Lucas [38] evolved a high-performance neural-network controller for a simulated version of the track, but performed poorly on the real-world track. Floreano et al. [39] evolved an active vision system for driving a car in a 3D racing game. Maybe one day this method can be applied to drive a car in the real world.

3.1.2 ALife Games

Some video games try to go beyond the game playing strategy and controll and apply the concepts of ALife to the whole game. Nerve Garden is a biologically inspired multi-user collaborative 3D virtual world. It is available to a general Internet audience through low speed dial-up connections and standard Internet protocols running on all major hardware platforms. The project combines a number of methods and technologies, including L-systems, Java, cellular automata, and VRML. Nerve Garden is a work designed to provide a compelling experience of a virtual terrarium that exhibits properties of growth, decay and energy transfer reminiscent of a simple ecosystem. The goals of the Nerve Garden project are to create an on-line “collaborative A-Life laboratory” which can be extended by a large number of users for purposes of education and research [40].

The first version of the system allowed users to operate a Java client, the Germinator to extrude 3D plant models generated from L-systems. The 3D interface in the Java client provided an immediate 3D experience of various L-system plant and arthropod forms. Users employed a slider bar to extrude the models in real time and a mutator to randomise production rules in the L-systems and generate variants on the plant models. After germinating several plants, the user would select one, name it and submit it into to a common VRML97 scenegraph called the Seeder Garden.

Mindscape Creatures is a product of the Creature Labs division of Cyber-Life Technology, Ltd. This is a ALife computer game created in the mid-1990s by English computer scientist Steve Grand whilst working for the Cambridge computer games developer Millennium Interactive. The program is regarded as an important breakthrough in the advancement of ALife research. The creatures in this computer game are called Norns, and the world's population of them at one stage hovered around the five million mark, making them more common than many familiar natural species. Each Norn is composed of thousands of tiny simulated biological components, such as neurons, biochemicals, chemoreceptors, chemoemitters and genes. The genes dictate how these components are assembled to make the complete organisms, and the creatures behaviour then emerges from the interactions of those parts, rather than being explicitly programmed in. The Norns are capable of learning about their environment, either by being shown things by their owners or through learning by their own mistakes. They must learn for themselves how to find food and how to interact with the many objects in their environment. They can interact with their owners, using simple language, and also with each other. They can form relationships and produce offspring, which inherit their neural and biochemical structure from their parents and are capable of open-ended evolution over time. They can fall prey to a variety of diseases (as well as genetic defects) and can be treated with appropriate medicines [41]. The player has also the possibility to directly manipulate the Norns genes, through a special editor, in order to obtain the desired physical or behaviour characteristics [42]. The commercial success of this title, with more than a million copies sold worldwide, reflects the relationships many users are willing to form with believable artificial life characters [24].

Spore is another ALife video game by Maxis and designed by Will Wright and it is still under development. The game has drawn wide attention for its promise to simulate the development of a species through open-ended, on-the-fly, user-guided evolution. Spore is a game where the player molds and guides

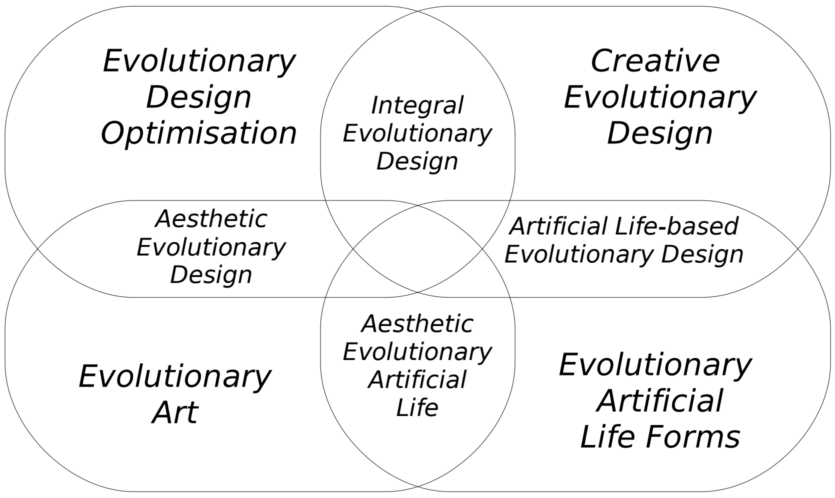

Evolutionary Design Optimisation
Integral Evolutionary Design
Creative Evolutionary Design
Aesthetic Evolutionary Design
Artificial Life-based Evolutionary Design
Aesthetic Evolutionary Artificial Life
Evolutionary Art
Evolutionary Artificial Life Forms

Figure 3.1: Evolutionary design categories

a species across many generations, growing it from a single-celled organism into a more complex animal. Eventually, the species becomes sapient. At this point the player begins molding and guiding this species' society, progressing it towards a space-faring civilisation. Spore's main innovation is the use of procedural generation for many of the components of the game, providing vast scope and open-endedness [43].

3.2 Evolutionary Design

Evolutionary design is a branch of evolutionary computation which has its roots in three different disciplines: computer science, evolutionary biology and design. Evolutionary design has taken place in many different areas over the last decade. Designers have optimised selected parts of their designs using evolution, artists have used evolution to generate aesthetically pleasing forms, architects have evolved new building plans from scratch, computer scientists have evolved morphologies and control systems of artificial life.

Evolutionary design can be divided into four main categories [44]: evolutionary design optimisation, creative evolutionary design, evolutionary art and evolutionary artificial life forms. However, some author's work may be included in two or more categories creating four overlapping sub-categories shown in Figure 3.1. Our technique fits in the subcategory of Aesthetic Evolutionary Design.

3.3 Related Work

Evolutionary art systems are similar in many ways: they all generate new forms or images from the ground up (random initial populations); they rely upon a human evaluator to set the fitness value of an individual based on subjective evaluation, such as aesthetic appeal; the population sizes are very small to avoid user's fatigue and allow a quick evaluation; and user interfaces usually present a grid on the screen with the current population individuals, allowing the user to rank them. However, they differ on their phenotype representations [45]. If we use a terrain surface as phenotype, instead of an image, it is possible to apply the same principle of evolutionary art to terrain generation. The following paragraphs review the most proeminent works of evolutionary art systems with GP.

GP has been the most fruitful evolutionary algorithm applied to evolve images interactively. Karl Sims used GP to create and evolve computer graphics by mathematical equations. The equations are used to calculate each pixel [46], or create graphic movies by adding a time variable to the dynamic differential equations [47]. He created several graphic art pieces including *Panspermia* and *Primordial Dance* and also allowed visitors interact with his interactive art system at art shows and exhibitions. His *Galapagos*[1] is an L-system based Interactive Evolutionary Computation (IEC) system that allows visitors to create their own graphic art through their interaction.

Tatsuo Unemi developed *SBART* (Simulated Breeding ART) [48, 49], an IEC graphics system open to public. *SBART* uses GP to create mathematical equations for calculating each pixel value and its (x, y) coordinates. As GP nodes *SBART* assigns the four arithmetic fundamental operators ($+$, $-$, $\times$ and $\div$), *power*, *sqrt*, *sin*, *cos*, *log*, *exp*, *min* and *max*. The terminal nodes are constants and variables. Three values at each pixel are calculated using one generated mathematical equation by assuming that the constants are 3D vectors consisting of three real numbers and the variables are 3D tuples consisting of $(x, y, 0)$. The three calculated values are regarded as members of a vector (hue, lightness and saturation) and are transformed to RGB values for each pixel. These three values are normalised to values in $[-1, 1]$ using a saw-like function. It allows the creation of movies by replacing $(x, y, 0)$ with (x, y, t), where t is a time variable. The *SBART*'s functions were expanded to create a collage [50]. A human user selects preferred 2D images from 20 displayed images at each generation and the system creates the

[1]http://www.genarts.com/galapagos

next 20 offspring. Sometimes exporting/importing parents among multiple *SBART* instances is allowed. This operation is iterated until the user obtains a satisfactory image.

In *NEvAr* (Neuro Evolutionary Art) [51], of Penousal Machado et al., the function set is composed mainly of simple functions such as arithmetic, trigonometric and logic operations. The terminal set is composed of a set of variables x, y and random constants. The phenotype (image) is generated by evaluating the genotype for each (x, y) pair belonging to the image. In order to produce colour images, *NEvAr* resorts to a special kind of terminal that returns a different value depending on the colour channel – Red, Green or Blue – that is being processed. This tool focus on the reuse of useful individuals, which are stored in an image database and led to the development of automatic seeding procedures.

Despite the interesting results achieved by the above evolutionary art systems, to the best of our knowledge, none of them has applied this technique to evolve terrain landscaps.

3.4 Genetic Programming

Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. At the most abstract level GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level statement of what needs to be done. In GP a population of computer programs is evolved, generation by generation a population of programs is stochastically transformed into new, hopefully better, populations of programs [52]. Due to its random nature GP can never guarantee results, however it has been used successfully in many areas. A brief list of GP applications follows [22]:

- Artificial life;
- Robots and autonomous agents;
- Financial trading;
- Neural networks;
- Art;
- Image and signal processing;
- Prediction and classification;
- Optimisation;

Algorithm 3.1 Genetic programming basic algorithm

```
1: Randomly create an initial population of programs from the available
   primitives
2: repeat
3:    Execute each program and ascertain its fitness
4:    Select one or two program(s) from the population with a probability
      based on fitness to participate in genetic operations
5:    Create new individual program(s) by applying genetic operations with
      specified probabilities
6: until an acceptable solution is found or some other stopping condition
   is met (e.g., a maximum number of generations is reached)
7: return the best-so-far individual
```

There are now 36 instances where GP has automatically produced a result that is competitive with human performance: 15 instances where GP has created an entity that either infringes or duplicates the functionality of a previously patented 20th century invention; 6 instances where GP has done the same with respect to a 21st century invention and 2 instances where GP has created a patentable new invention [53].

Algorithm 3.1 shows the basic steps of GP. The generated programmes are runned for evaluation (line 3) and compared with some ideal. This comparison is quantified to give a numeric value called fitness. The best programs are chosen to breed (line 4) and produce new programs for the next generation (line 5). The primary genetic operators used to create new programs from existing ones are:

- **Crossover** - The creation of a child program by combining randomly chosen parts from two selected parent programs;
- **Mutation** - The creation of a new child program by randomly altering a randomly chosen part of a selected parent program;

3.4.1 Representation

In GP, programs are usually expressed as syntax trees rather than as lines of code. For example Figure 3.2 shows the tree representation of the program $max(x + x, x + 3 * y)$. The variables and constants in the program (x, y and 3) are leaves of the tree, or terminals in GP terminology. The arithmetic operations (+, * and max) are internal nodes called functions. The sets

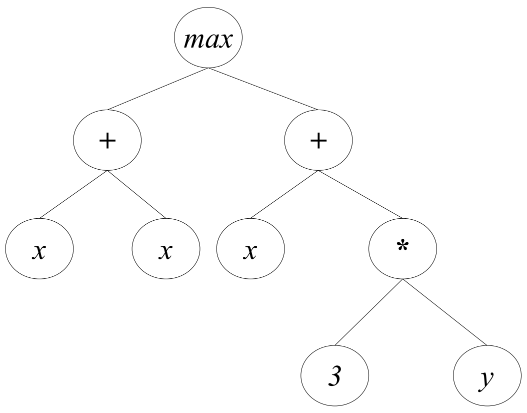


Figure 3.2: GP tree representation of $max(x + x, x + 3 * y)$

of allowed functions and terminals together form the primitive set of a GP system. It is common in the GP literature to represent expressions in a prefix notation similar to that used in Lisp. For example, $max(x + x, x + 3 * y)$ becomes $(max(+xx)(+x(*3y)))$. This notation often makes it easier to see the relationship between (sub)expressions and their corresponding (sub)trees. From now on it will be used trees and their corresponding prefix-notation expressions interchangeably to represent GP programs.

3.4.2 Initialising the Population

Like in other evolutionary algorithms, in GP the individuals in the initial population are typically randomly generated. There are a number of different approaches to generating this random initial population. Following we present a description of the two simplest methods, the *full* and *grow* methods, and a widely used combination of the two known as *ramped half-and-half.*

In both the *full* and *grow* methods, the initial individuals are generated so that they do not exceed a user specified maximum depth. The depth of a node is the number of edges that need to be traversed to reach the node starting from the tree's root node (which is assumed to be at depth 0). The depth of a tree is the depth of its deepest leaf (e.g., the tree in Figure 3.2 has a depth of 3). In the *full* method, where all leaves are at the same depth, nodes are taken at random from the function set until the maximum tree depth is reached. Beyond that depth, only terminals can be chosen. Figure 3.3 shows an example of a tree having maximum depth 2 created with the full initialisation method, where all leaves are at the same depth. However,

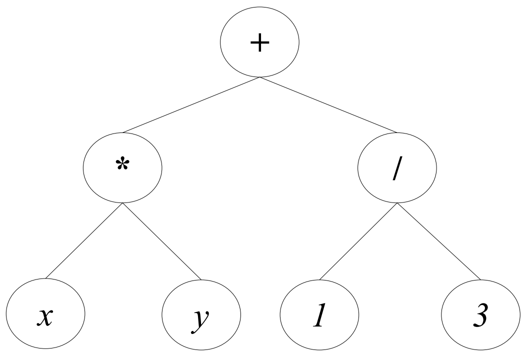


Figure 3.3: Example of a tree having maximum depth 2 created with the *full* initialisation method

this does not necessarily mean that all initial trees will have an identical number of nodes (often referred to as the size of a tree) or the same shape. In fact, this only happens when all the functions in the primitive set have the same number of input values, also known as arity. Nonetheless, even when mixed-arity primitive sets are used, the range of program sizes and shapes produced by the *full* method may be rather limited.

On the contrary the *grow* method allows the creation of trees of more varied sizes and shapes. Nodes are selected from both the primitive function and terminals set until the depth limit is reached. Once the depth limit is reached only terminals may be chosen (like the *full* method). Figure 3.4 illustrates an example of a tree created with the *grow* initialisation method with depth limit 2. Here the first argument of the $+$ root node happens to be a terminal. This closes off that branch preventing it from growing any more before it reached the depth limit. The other argument is the function $-$, but its arguments are forced to be terminals to ensure that the resulting tree does not exceed the depth limit.

Because neither the *grow* or *full* method provide a very wide array of sizes or shapes on their own, Koza [54] proposed a combination called *ramped half-and-half*. Half the initial population is constructed using *full* and half is constructed using *grow*. This is done using a range of depth limits (hence the term "ramped") to help ensure that it generates trees having a variety of sizes and shapes.

These methods are easy to implement and use, but are difficult to control regarding the statistical distributions of important properties such as the sizes and shapes of the generated trees. For example, the sizes and shapes of the trees generated via the *grow* method are highly sensitive to the sizes

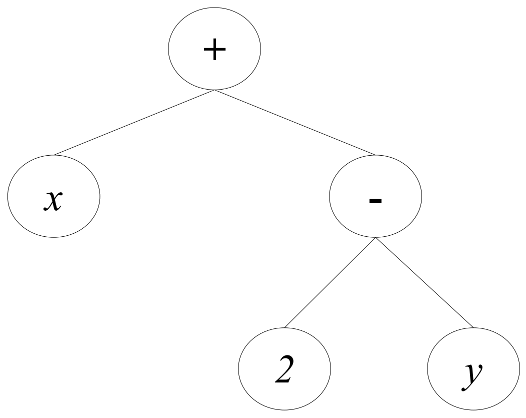


Figure 3.4: Example of a tree having maximum depth 2 created with the *grow* initialisation method

of the function and terminal sets. If, for example, one has significantly more terminals than functions, the grow method will almost always generate very short trees regardless of the depth limit. Similarly, if the number of functions is considerably greater than the number of terminals, then the *grow* method will behave quite similarly to the *full* method. The arities of the functions in the primitive set also influence the size and shape of the trees produced by *grow*. While these are particular problems for the *grow* method, they illustrate a general issue where small (and often apparently inconsequential) changes such as the addition or removal of a few functions from the function set can in fact have significant implications for the GP system, and potentially introduce important but unintended biases. For more information about this and other initialisation mechanisms check [52].

3.4.3 Selection

In GP the genetic operators are applied to individuals that are probabilistically selected based on their fitness. Which means that better individuals are more likely to have more child programs than inferior individuals. The most commonly employed method for selecting individuals in GP is *tournament* selection.

In *tournament* selection a number of individuals are chosen at random from the population. These are compared with each other and the best of them is chosen to be the parent. For crossover two parents are needed, so, two selection *tournaments* are made. *Tournament* selection only looks at which program is better than another, it does not need to know how much better. This automatically rescales fitness, so that the selection pressure on the population remains constant. Hence, a single extraordinarily good program cannot immediately flood the next generation with its children. If that happened, it would lead to a rapid loss of diversity with potentially undesirable consequences. In reverse, *tournament* selection amplifies small differences in fitness to prefer the better program even if it is only marginally superior to the other individuals in a tournament. An element of noise is inherent in *tournament* selection due to the random selection of candidates for tournaments. So, while preferring the best, *tournament* selection does ensure that even average quality programs have some chance of having children.

On interactive systems the selection is performed by a human, usually based on a visual representation of the individuals. Many other selection methods are possible, such as [55, 56]:

- Fitness Proportional Selection (also known as roulette wheel);
- Stochastic Universal Sampling (SUS);
- Lexicographic Parsimony Pressure Tournament;
- Doubletour;

3.4.4 Genetic Operators

The GP implementation of the genetic operators *crossover* and *mutation* are significantly different from other evolutionary algorithms. The choice of which operator, mutation or crossover, should be used to create an offspring is probabilistic. Operators in GP are normally mutually exclusive (unlike other evolutionary algorithms where offspring are sometimes obtained via a composition of operators). Their probability of application are called operator rates. Typically, crossover is applied with the highest probability often being 90% or higher. On the contrary, the mutation rate is much smaller, typically being near 1%. When the sum of crossover and mutation rates are equal to p which is less than 100%, an operator called reproduction is also used. Reproduction is the selection of an individual based on fitness and the insertion of a copy of it in the next generation, with a rate of $1 - p$ [52].

The next two sections provide a brief description of the most common GP operators.

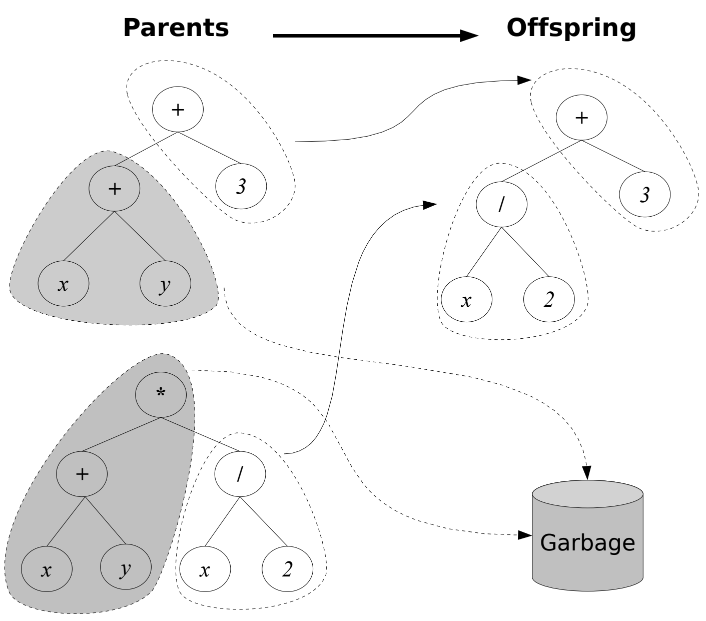


Figure 3.5: Example of subtree crossover (the trees on the left are copies of the parents)

Crossover

The most commonly used form of crossover is subtree crossover. Given two parents, subtree crossover randomly (and independently) selects a crossover point (a node) in each parent tree. Then, it creates the offspring by replacing the subtree rooted at the crossover point in a copy of the first parent with a copy of the subtree rooted at the crossover point in the second parent, as illustrated in Figure 3.5. Copies are used to avoid disrupting the original individuals. This way, if selected multiple times, they can take part in the creation of multiple offspring programs. It is also possible to define a version of crossover that returns two offspring, but this is not commonly used.

Often crossover points are not selected with uniform probability. Typical GP primitive sets lead to trees with an average branching factor (the number of children of each node) of at least two, so the majority of the nodes will be leaves. Consequently the uniform selection of crossover points leads to crossover operations frequently exchanging only very small amounts of genetic material. Some times the operation is reduced to simply swapping two leaves. To counter this, Koza suggested [54] the widely used approach of choosing functions 90% of the time and leaves 10% of the time. Many other types of crossover are possible, such as [52]:

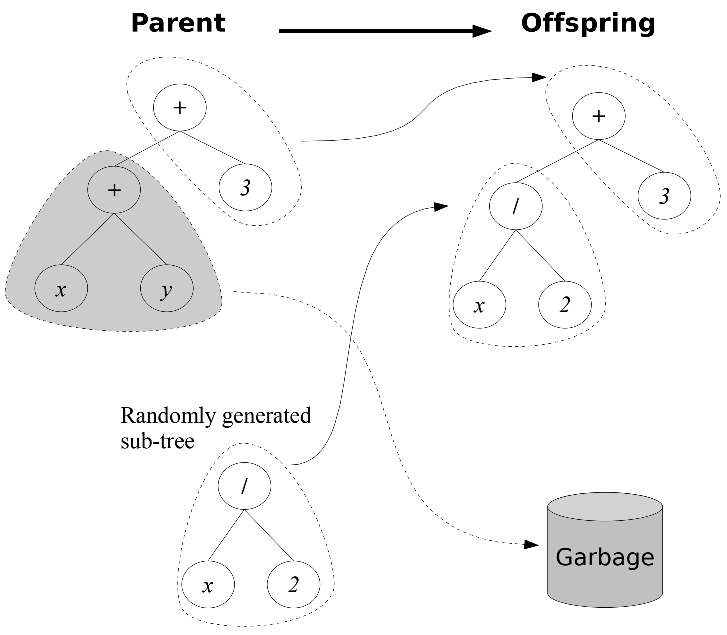


Figure 3.6: Example of subtree mutation

- One-point crossover;
- Uniform crossover;
- Context-preserving crossover;
- Size-fair crossover;

Mutation

The most common type of mutation in GP is called subtree mutation and randomly selects a mutation point in a tree and substitutes the subtree rooted there with a randomly generated subtree. This is illustrated in Figure 3.6.

Another kind of mutation implementation is the point mutation, which is the GP's equivalent of the bit-flip mutation used in genetic algorithms. Point mutation, on the other hand, is typically applied on a per-node basis. A random node is selected and the primitive stored there is replaced with a different random primitive of the same arity taken from the primitive set. If no other primitives with that arity exist, nothing happens to that node, but other nodes may still be mutated. Each node is considered in turn and, with a certain probability, it is altered as explained above. This allows multiple nodes to be mutated independently in one application of point mutation. Many other types of mutation are possible, such as [52]:

- Size-fair subtree mutation;
- Hoist mutation;
- Shrink mutation;
- Permutation mutation;
- Mutating constants at random;
- Mutating constants systematically;

3.4.5 Terminal Set

GP is commonly described as evolving programs, but is not typically used to evolve programs the same way humans do for software development. Instead, it is more common to evolve programs (or expressions, or formulae) in a more domain-specific language. The definition of the terminal and function sets specify such a language. That is, together they define the ingredients that are available to GP to compose computer programs.

The terminal set may consist of:

- the program's external inputs - these typically take the form of named variables (e.g., x, y);
- functions with no arguments - these may be included because they return different values each time they are used, such as the function *rand()* which returns random numbers, or a function distance to *wall()* that returns the distance to an obstacle from a robot that GP is controlling. Another possible reason is because the function produces side effects. Functions with side effects do more than just return a value: they may change some global data structures, print or draw something on the screen, control the motors of a robot, etc;
- constants - these can be pre-specified, randomly generated as part of the tree creation process, or created by mutation;

Using *rand()* as terminal can cause the behaviour of an individual program to vary every time it is called, even if it is given the same inputs. This is desirable in some applications. However, it is more common to want a set of fixed random constants that are generated as part of the process of initialising the population. This is typically accomplished by introducing a terminal that represents an *ephemeral random constant*. Every time this terminal is chosen in the construction of an initial tree (or a new subtree to use in an operation like mutation), a different random value is generated which is then used for that particular terminal, and which will remain fixed for the rest of the run. The use of ephemeral random constants is typically denoted by including the symbol $\Re$ in the terminal set, see example in Eq. (3.1).

$$T = \{x, y, \Re\} \ . \tag{3.1}$$

3.4.6 Function Set

The function set used in GP is based on the nature of the problem domain. For example, in a simple numeric problem the function set can be only the arithmetic functions $(+, -, *, /)$. However, all sorts of other functions typically encountered in computer programs can be used, such as:

- Arithmetic;
- Mathematical;
- Boolean;
- Conditional;
- Looping;
- Signal processing functions;
- etc;

Closure

For GP to work effectively, function sets are required to have an important property known as *closure* [54]. The *closure* property can be divided into *consistency* and *evaluation safety* properties.

Subtree crossover, as described in Section 3.4.4, can mix and join nodes arbitrarily, thus the need for type *consistency*. As a result, it is necessary that any subtree can be used in any of the argument positions for every function in the function set, because it is always possible that subtree crossover will generate that combination. So, all the functions must return values of the same type, and that each of their arguments also have this type. For example $+$, $-$, $*$, and $/$ can be defined so that they each take two integer arguments and return an integer. Sometimes type consistency can be weakened somewhat by providing an automatic conversion mechanism between types. It is possible, for example, convert numbers to Booleans by treating all negative values as false, and non-negative values as true. However, conversion mechanisms can introduce unexpected biases into the search process, so they should be used with care [52].

The other component of closure is *evaluation safety*, this property is required because many used functions can fail at run time. An evolved expression might, for example, divide by 0. This is typically dealt with by modifying the normal behaviour of primitives. It is common to use protected versions of numeric functions that can otherwise throw exceptions, such as *division*, *logarithm*, *exponential* and *sqrt*. The protected version of a function first tests for potential problems with its input(s) before executing the corresponding instruction. If a problem is spotted then some default value is returned. It is common to use the prefix *my* to denote protected functions, for example *mySqrt*.

An alternative way to protect functions is to trap run-time exceptions and strongly reduce the fitness of programs that generate such errors. However, if the likelihood of generating invalid expressions is very high, this can lead to too many individuals in the population having nearly the same (very poor) fitness. This makes it hard for selection to choose which individuals might make good parents.

Sufficiency

Sufficiency is another property that primitives sets should have. *Sufficiency* means it is possible to express a solution to the problem being solved using the elements of the primitive set. Unfortunately, *sufficiency* can be guaranteed only for those problems where theory, or experience with other methods, tells that a solution can be obtained by combining the elements of the primitive set.

An example of an *insufficient* set is $+, -, *, /, x, 0, 1, 2$, which is unable to represent the function $exp(x)$. This function cannot be expressed as a ratio of polynomials, so, it cannot be represented exactly by any combination of $+, -, *, /, x, 0, 1, 2$. When a primitive set is *insufficient*, GP can only generate programs that approximate the desired one. However, in many cases such an approximation can be very close and good enough for the user's purpose. Adding a few unnecessary primitives in an attempt to ensure sufficiency does not tend to slow down GP overmuch, although there are cases where it can bias the system in unexpected ways [52].

Evolving Structures

There are many problems where solutions cannot be directly generated as computer programs. This is common in many design problems were the solution is an artifact of some type: a bridge, a circuit, an antenna, a lens, a terrain, etc. To address this issue the primitive set is set up so that the evolved programs construct solutions to the problem. For example, if the goal is the automatic creation of an electronic controller for a plant, the function set might include common components such as integrator, differentiator, lead, lag, and gain, and the terminal set might contain reference, signal, and plant output. Each of these primitives, when executed, inserts the corresponding device into the controller being built. If, on the other hand, the goal is to synthesise analogue electrical circuits, the function set might include components such as transistors, capacitors, resistors, etc.

3.4.7 Fitness Function

The fitness function has the task to measure how good programs are and rank them. It is through the fitness function that a high-level statement of the problem's requirements is given to the GP system. For example, suppose the goal is to get GP to synthesise an amplifier automatically. Then the fitness function is the mechanism which tells GP to synthesise a circuit that amplifies an incoming signal.

Fitness can be measured in many ways. For example, in terms of: the amount of error between its output and the desired output; the amount of time (fuel, money, etc.) required to bring a system to a desired target state; the accuracy of the program in recognising patterns or classifying objects; the payoff that a game-playing program produces; the compliance of a structure with user-specified design criteria.

Fitness functions used in GP are different from those used in other evolutionary algorithms. This happens because the structures being evolved in GP are computer programs, were fitness evaluation normally requires executing all the programs in the population and typically multiple times. While one can compile the GP programs, the overhead of building a compiler is substantial, so it is much more common to use an interpreter to evaluate the GP programs.

Interpreting a program tree means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of

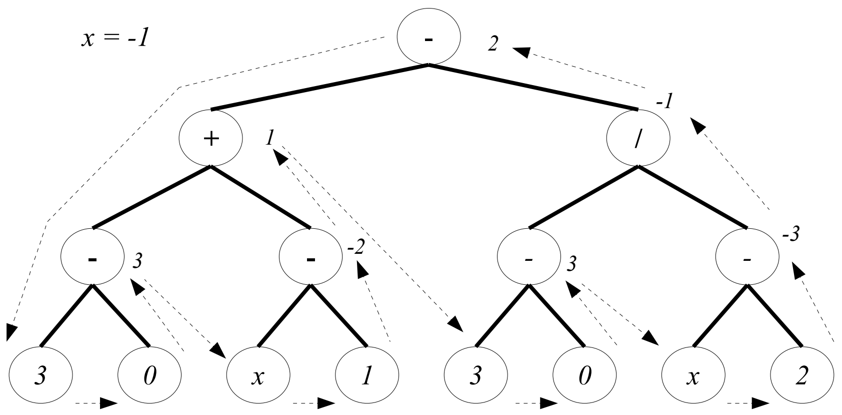


Figure 3.7: Interpretation example of a GP tree (the terminal x is a variable and has a value of -1). The number to the right of each internal node represents the result of evaluating the subtree root at that node.

their arguments is known. This is done by traversing the tree recursively starting from the root node, and postponing the evaluation of each node until the values of its children (arguments) are known. This depth-first recursive process is illustrated in Figure 3.7.

In some problems the solution that is being looked for is the output produced by a program, such as the returned value of the evaluated tree. On other problems the solution that is being looked for is the actions performed by a program composed of functions with side effects. In either case the fitness of a program typically depends on the results produced by its execution on many different inputs or under a variety of different conditions. For example the program might be tested on all possible combinations of inputs $x1$, $x2$, …, xN. Alternatively, a robot control program might be tested with the robot in a number of starting locations. These different test cases typically contribute to the fitness value of a program incrementally, and for this reason are called fitness cases [52].

There are also interactive GP systems were the individuls' evaluation is performed by a human. The main reason for this approach is the impossibility, or impracticability, to define a fitness function to represents the desired solution. This type of evaluation is commonly used to evolve aesthetic designs or other forms of art work.

3.4.8 GP Parameters

There are several parameters that need to be specified before running the GP system. The most important control parameter is the population size. Other control parameters include the probabilities of performing the genetic operations and the maximum size for programs. There are no general recommendations for setting optimal parameter values, as these depend too much on the details of the application. However, genetic programming is in practice robust, and it is likely that many different parameter values will work. As a consequence, one need not typically spend a long time tuning GP for it to work adequately. It is common to create the initial population randomly using ramped half-and-half with a depth range from 2 to 6. The initial tree sizes will depend upon the number of the functions, the number of terminals and the arities of the functions. However, evolution will quickly move the population away from its initial distribution [52].

The main limitation on the population size is the time taken to evaluate the fitnesses. So, it is preferable to have the largest population size that the system can handle gracefully. Normally, the population size should be at least 500, but lager populations are often used. GP runtime can be estimated by the product of: the number of runs R, the number of generations G, the size of the population P, the average size of the programs s and the number of fitness cases F [52].

Typically, the number of generations is limited to between 10 and 15. The most productive search is usually performed in those early generations, and if a solution has not been found then, it's unlikely to be found in a reasonable amount of time. A common wisdom on population size is to make it as large as possible. It is also common to impose either a size or a depth limit or both on tree's sizes to prevent bloat - the uncontrolled growth of program sizes during GP runs [52].

3.4.9 Termination

The last step of GP algorithm is the specification of the termination criterion and the method of designating the result. The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate. Typically, the single best-so-far individual is designated as the result of the run, although additional data might be returned. In case of interactive systems the user decides when to stop the run.

Chapter 4

Genetic Terrain Programming

Current terrain generation techniques have their own advantages and disadvantages, which are detailed in Section 2.4. Notwithstanding the importance of real looking terrains, none of the existing methods focused on generating terrains accordingly to designers' aesthetic appeal. The main goals of Genetic Terrain Programming are to address the weaknesses of existing methods and also to allow the generation of aesthetic terrains. Thus, providing a *better* way of generating virtual terrains for a broad range of applications, with a special emphasis on video games.

In light of the idealised terrain generator (Section 2.2), the goals of Genetic Terrain Programming are (in order of decreasing importance):

1. Capable of generating diverse features and terrain types, both aesthetic and realistic;
2. Extensibility;
3. Intuitive to control;
4. Automated generation with arbitrary LOD;
5. Low requirements of human input;

To achieve these goals we use aesthetic evolutionary design with GP, where the phenotypes are terrains represented as height maps. This approach consists of a guided evolution, through Interactive Evolution, accordingly to a specific desired terrain feature or aesthetic appeal. The extensibility and ability to generate diverse features and terrain types are assured by the GP. The diversity of solutions is directly dependent on the GP terminal and function sets. So, the extensibility feature can be easily achieved by adding new functions and terminals. The designer will guide the terrains evolution, preforming this way the control of the outcome, by selecting which ones he preferres for his specific goals. Consequently, the software tool will be easy and intuitive to use with low input input requirements. The outcome of the interactive evolution will be TPs (Terrain Programmes) which are mathematical expressions with incorporated randomness. Those TPs can be used, like a procedural technique, to automatically generate different terrains but consistently with the same features and with different LODs.

Section 4.1 describes the GP implementation details (parameters, function and terminal sets) of our technique and the differences with regard to other GP evolutionary systems. To demonstrate the GTP feasibility the *GenTP* tool was developed whose functionalities are presented in Section 4.2. Finally, Section 4.3 shows the obtained results in three sets of experiments.

4.1 Method

The initial population is created randomly, with trees depth size limited initially to 6 and a fixed population size of 12. The number of generations is decided by the designer, who can stop the algorithm at any time. The designer can select one or two individuals to create the next population and the genetic operators used depend upon the number of selected individuals. If one individual is selected only the mutation operator will be used. In case the designer chooses to select two individuals both the standard crossover and mutation operators [54] will be applied. Like in others IEC systems, the fitness function relies exclusively on designers' decision, either based on his aesthetic appeal or on desired features.

Accordingly to Bentley [45] the designer is likely to score individuals highly inconsistently as he might adapt his requirements along with the evolved results. So, the continuous generation of new forms based on the fittest from the previous generation is essential. Consequently, non-convergence of the EA is a requirement. Evolutionary art systems do not usually use crossover operators on their algorithms, because EAs are used as a continuous novelty generators, not as optimisers. Therefore, in our algorithm, the use of two individuals for breeding the next generation should be limited.

Table 4.1: GP Functions

Name	**Description**
$plus(h_1, h_2)$ $minus(h_1, h_2)$ $multiply(h_1, h_2)$	arithmetical functions
$sin(h)$ $cos(h)$ $tan(h)$ $atan(h)$	trigonometric functions
$myLog(h)$	returns 0 if $h = 0$ and $log(abs(h))$ otherwise
$myPower(h_1, h_2)$	returns 0 if $h_1^{h_2}$ is NaN or Inf, or has imaginary part, otherwise returns $h_1^{h_2}$
$myDivide(h_1, h_2)$	returns h_1 if $h_2 = 0$ and $h_1 \div h_2$ otherwise
$myMod(h_1, h_2)$	returns 0 if $h_2 = 0$ and $mod(h_1, h_2)$ otherwise
$mySqrt(h)$	returns $sqrt(abs(h))$
$negative(h)$	returns $-h$
$FFT(h)$	2-D discrete Fourier Transform
$smooth(h)$	circular averaging filter with $r = 5$
$gradientX(h)$ $gradientY(h)$	returns the gradient (dh/dx or dh/dy) of a hight map h. Spacing between points is assumed to be 1

The extensive use of crossover operator will converge the population to a single solution, leading to the loos of diversity and limiting the designer to explore further forms.

Each GP individual is a tree composed by functions, listed in Table 4.1, and height maps as terminals (see Table 4.2).

Most terminals depend upon a Random Ephemeral Constant (REC) to define some characteristics, such as the spectrum value of *fftGen*. All terminals have some form of randomness, which means that consecutive calls of the same terminal will always generate a slightly different height map. This is a desired characteristic because we want to be able to create different terrains by each TP, but that will share the same features. All terminals generate forms that are proportional to the side size of the height map. This ensures that the terrain features of a TP are scale invariant. Figures 4.1, 4.2,

Table 4.2: GP Terminals

Name	**Description**
rand	map with random heights between 0 and 1
fftGen	spectral synthesis based hight map, whose spectrum depends on a REC: $1/(f^{REC})$
gauss	gaussian bell shape hight map, whose wideness depends on a REC
plane	flat inclined plane hight map whose orientation depends on a REC within 8 values
step	step shape hight map whose orientation depends on a REC within 4 values
sphere	semi-sphere hight map whose centre location is random and the radius depends on a REC

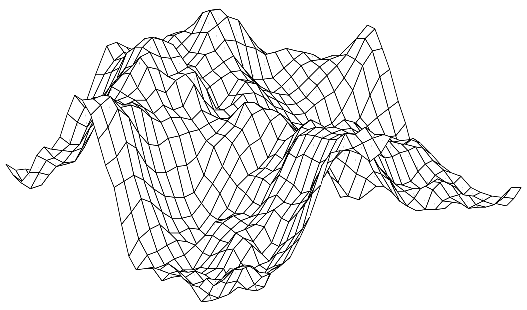


Figure 4.1: Example of hight map terminal *fftGen*

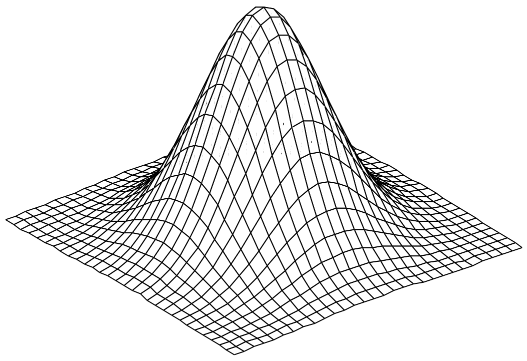


Figure 4.2: Example of hight map terminal *gauss*

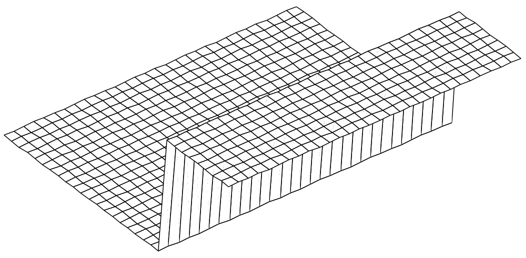


Figure 4.3: Example of hight map terminal *step*

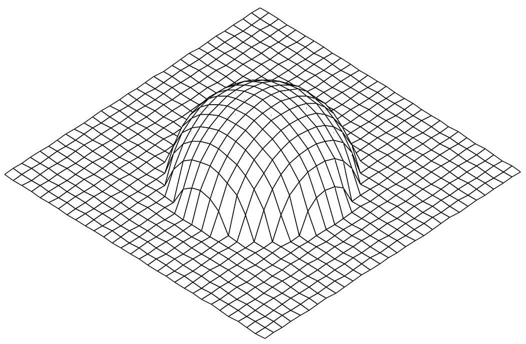


Figure 4.4: Example of hight map terminal *sphere*

4.3 and 4.4 show height maps of size 30×30 generated by terminals *fftGen*, *gauss*, *step* and *sphere*.

While in [49, 50] the mathematical equations are used to calculate both the pixel value and its coordinates, in *GenTP* only the height will be calculated. The (x, y) coordinates will be dictated by the matrix position occupied by the height value.

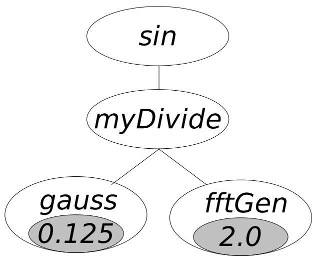


Figure 4.5: Example of a GTP tree individual with two RECs (in grey ellipses)

Table 4.3: Parameters for a GTP run

Objective:	Generate realistic or aesthetic terrains
Function set:	Functions from Table 4.1, all operating on matrices with float numbers
Terminal set:	Terminals from Table 4.2 chosen randomly
Selection and Fitness:	Decided by the designer accordingly to desired terrain features or aesthetic appeal
Population:	Fixed size with 12 individuals; initial depth limit 6; no tree size limits; random initialisation
Parameters:	If 2 individuals are selected: 90% subtree crossover and 10% mutation; if just one individual is selected: 50% mutation (without crossover)
Termination:	Can be stopped at any time by the designer, the "best" individual is chosen by the designer

4.2 GenTP

To implement this new technique we developed *GenTP* (Generator of Terrain Programs) [3], an application developed with GPLAB [1], an open source GP toolbox for Matlab [2]. *GenTP* has three functional modules (depicted in Figure 4.6):

- Interactive evolution;
- Analyse;
- Generation;

The interactive evolution module is where the GP is implemented and the designer chooses the desired terrains for the next generation, for the analyse or generation modules. Figure 4.7 shows the GUI (Graphical User Interface) of *GenTP*'s main interface, which is the visible part of the interactive evolution module. The 12 individuals of current population are represented as 3D surfaces and displayed in a 3×4 grid. Each TP is evaluated to produce a height map of size 100×100 to be displayed to the designer. The height map size can be changed, but should be kept small otherwise it might have

[1] http://gplab.sourceforge.net/
[2] http://www.mathworks.com/

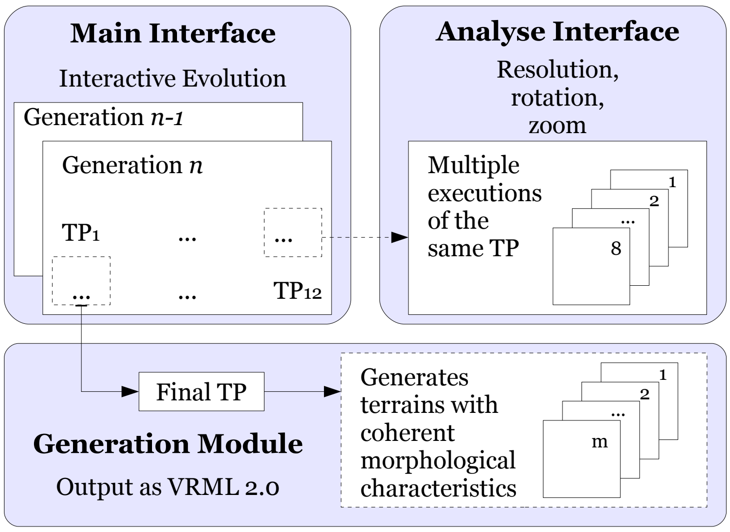


Figure 4.6: *GenTP*'s functional modules

a negative impact in the tool responsiveness. For more details about this see Section 4.3.3.

The *GenTP* main GUI allows a designer to select one or two individuals to create the next population generation. The number of selected TPs will influence their evolution. If just one TP is selected - only the mutation operator will be applied - the next generation will present few variations from the selected individual and the TP will evolve slowly. On the other hand, if the designer opts to select two individuals, the next generation will present more diversity and the evolved TPs can change their look more dramatically.

On the buttom of the main GUI the designer can see the TP mathematical expression that generated the selected terrain and save it on a text file or database. This option will allow the integration of TPs, as a procedural technique, to produce terrains for example on a video game.

Although the main interface serves its purpose, some times it is difficult to see all TP features due the display angle used to show the generated terrain. It is also difficult to inspect small details of a generated terrain and it is not possible to test the TP's features perseverance across multiple executions. For these reasons it might be difficult for the designer to chose the TPs for the next generation. To solve these limitations the analyse module was added to our application. This new functionality opens a new windows, see Figure 4.8, and performs 8 consecutive executions of the TP selected from the main interface. To allow a more detailed analysis of the TP

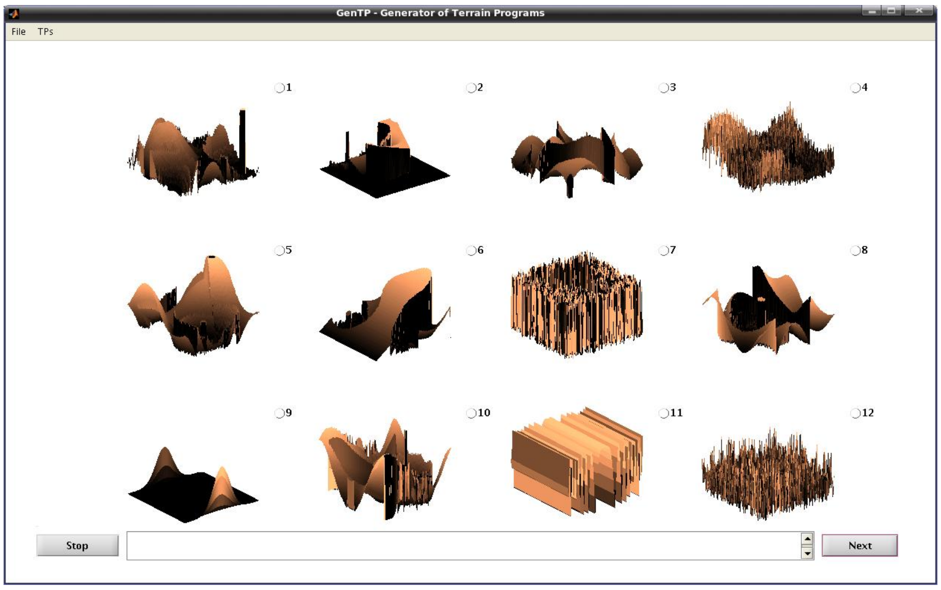


Figure 4.7: *GenTP* main user's interface

characteristics this interface allows the designer to rotate, zoom and change the terrains resolution. This way the designer has more information about a TP to decide if it will be selected, or not, for the next generation.

When the designer achieves the desired TP can save it in a file, or can pass it to the generator module. This module is responsible for the generation of height maps, as many as desired, from the selected TP. Those height maps can be saved as VRML 2.0 permitting its import from other applications, such as 3D modeling and render tools.

4.3 Results

Three kind of experiments were conducted:

1. obtaining aesthetic appealing terrains (regardless of their realism);
2. achieve terrains with a specific feature in mind;
3. test the perseverance of terrain features across several resolutions and generation time impact;

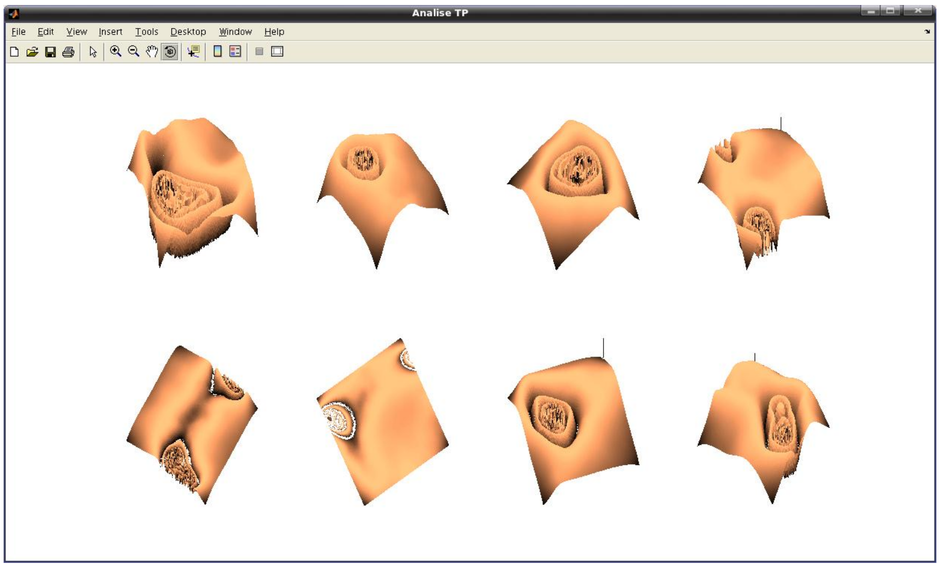


Figure 4.8: *GenTP* analyse user's interface

4.3.1 Aesthetic Terrains

On the first kind of experiments we were able to get aesthetic appealing terrains after about 30 to 70 generations. On those experiments we were able to obtain very different kinds of terrains types, see Figures 4.9, 4.10, 4.11, 4.12 and 4.13. Most of them are difficult to describe due to their exotic look. For example, the TP represented in Eq. 4.1 creates terrains with a bank of knolls with two ridges that give them an alien look (see Figure 4.9).

4.3.2 Terrains with specific features

On the second kind of experiments we tried to obtain TPs to generate terrains with specific features, such as mountains, cliffs or corals. In this case the number of necessary generations varies widely until we are able to get acceptable results. These number is highly dependent on the initial population and could vary between 10 to more than 100 generations. When running the experiments, if after a number of generations an interesting result is not obtained, we have preferred to cancel the experiment and begin again, avoiding this way a long run. We also verified that, for realistic landscapes, the range of terrains types were narrower than in the first experiment.

For example, the Eq. 4.7 has a TP that was evolved having in mind to

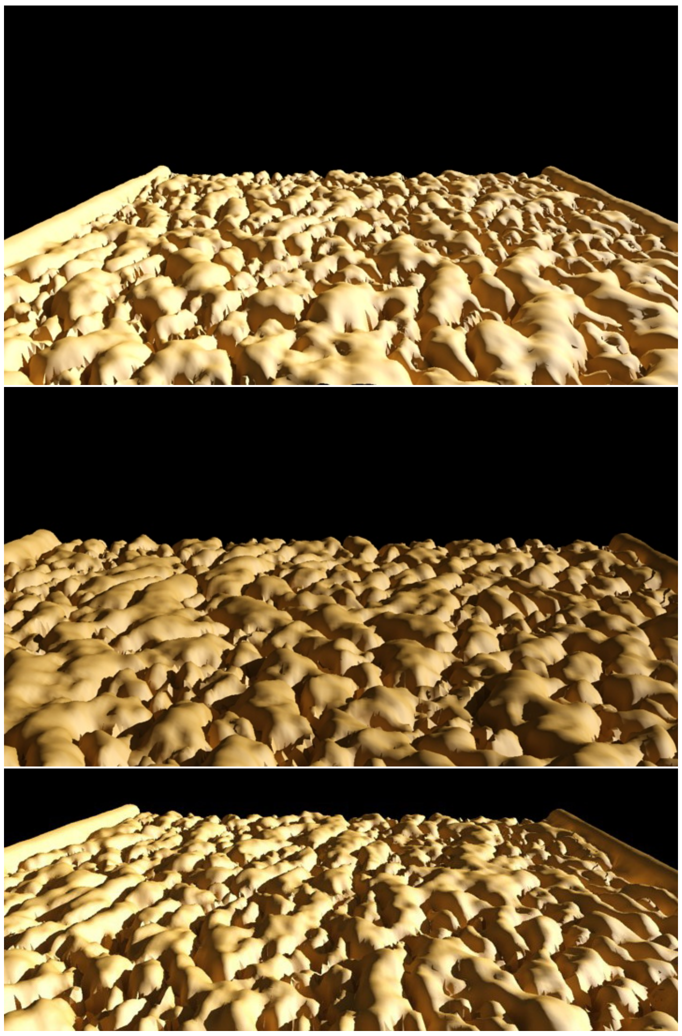


Figure 4.9: Exotic terrains generated by TP 4.1

$$TP = \quad myLog(sin(mySqrt(smooth(fftGen(1.25))))) \ . \tag{4.1}$$

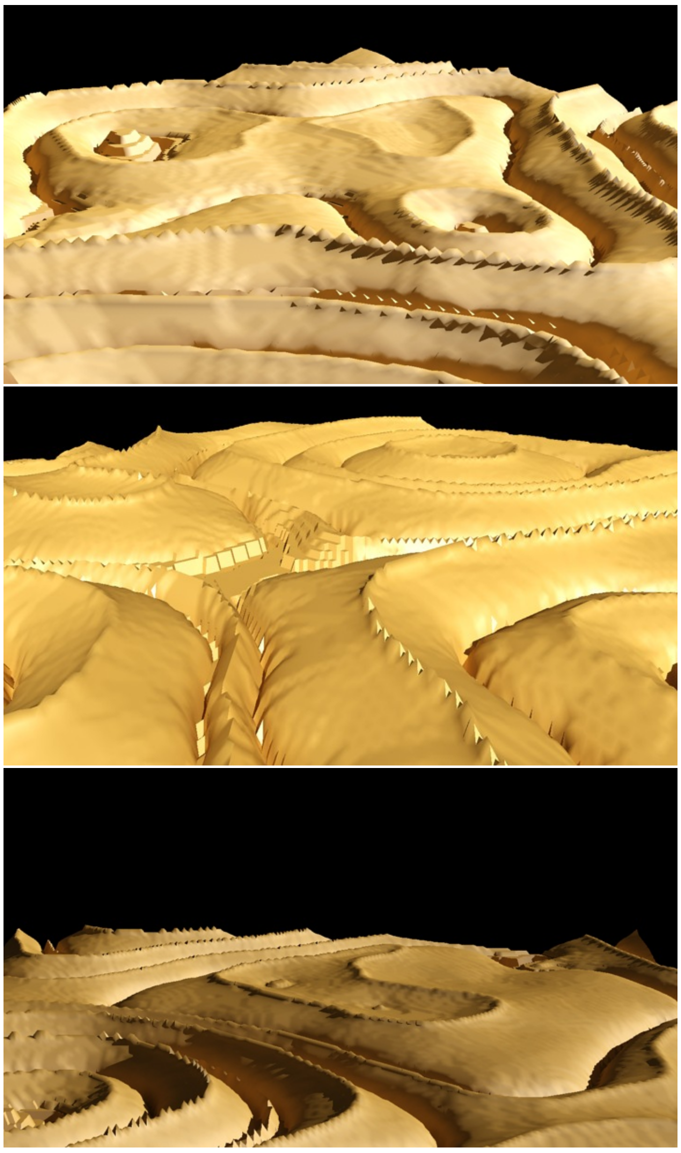


Figure 4.10: Exotic terrains generated by TP 4.2

$$TP = \quad myLog(myLog(myLog(myLog(myLog(myLog(fftGen(3.00))))))) \ . \tag{4.2}$$

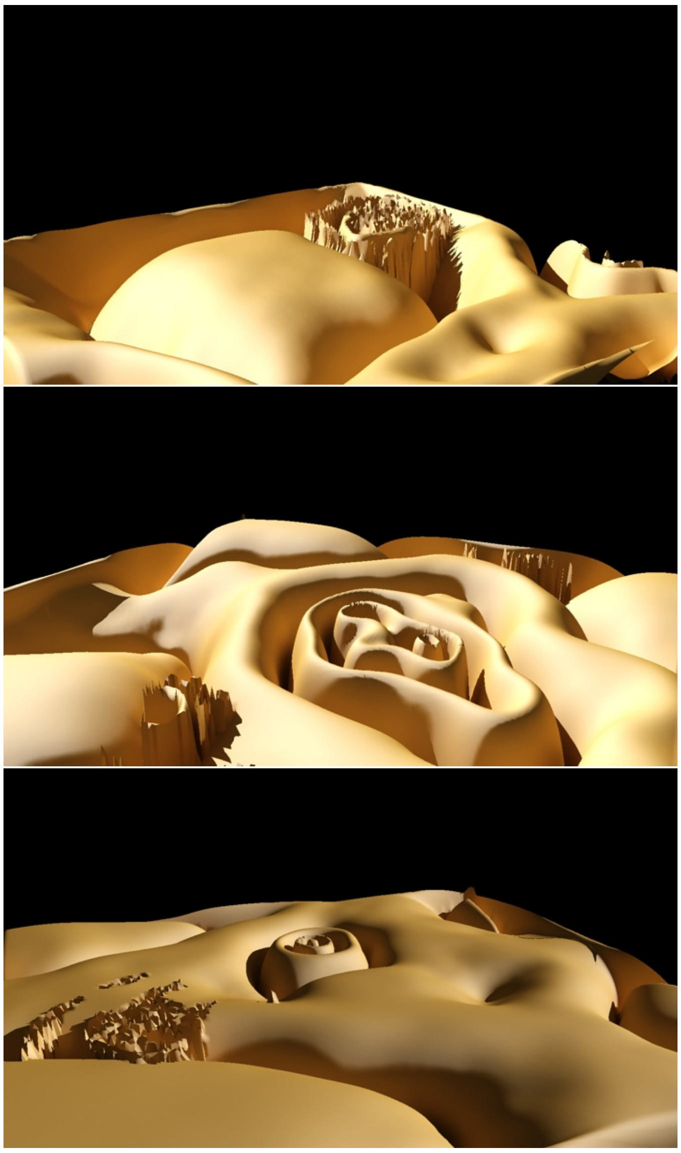


Figure 4.11: Exotic terrains generated by TP 4.3

$$\begin{aligned} TP = \quad & myPower(cos(myDivide(myLog(smooth(fftGen(2.75))), myMod(\\ & sin(fftGen(0.50)), myDivide(myLog(smooth(fftGen(2.75))), \\ & myMod((sin(fftGen(0.50))), fftGen(2.25)))))) \ . \end{aligned} \tag{4.3}$$

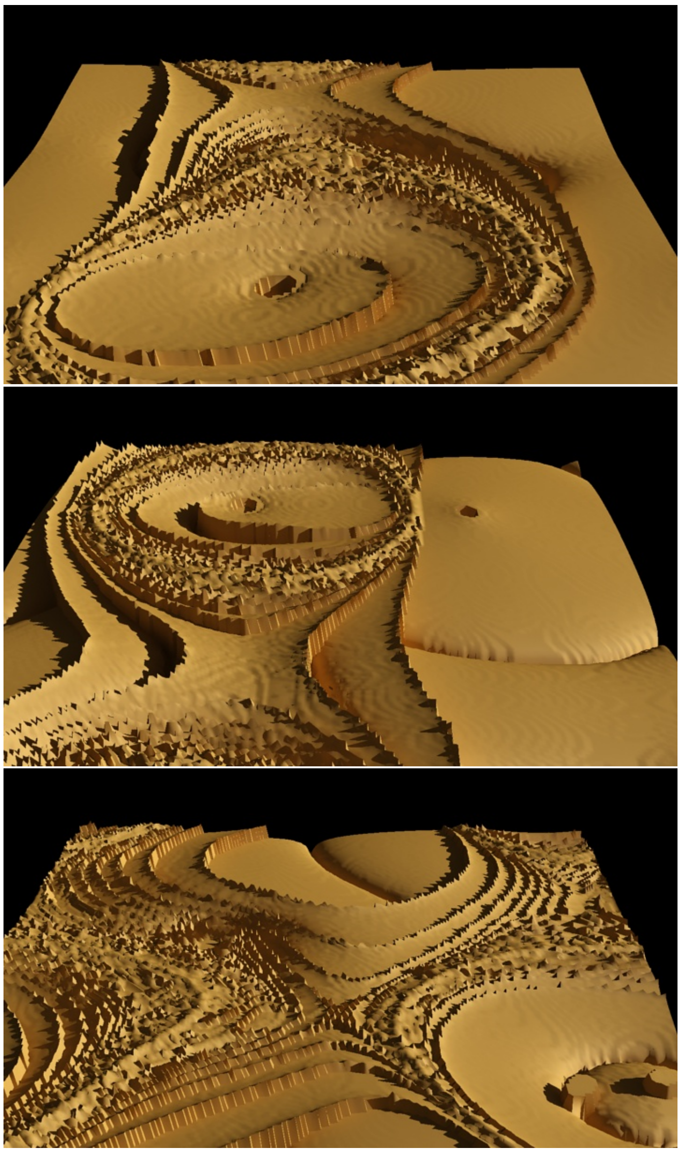


Figure 4.12: Exotic terrains generated by TP 4.4

$$TP = myLog(myLog(myMod(myLog(fftGen(3.75)), myLog(myLog(fftGen(4.25)))))) \ . \tag{4.4}$$

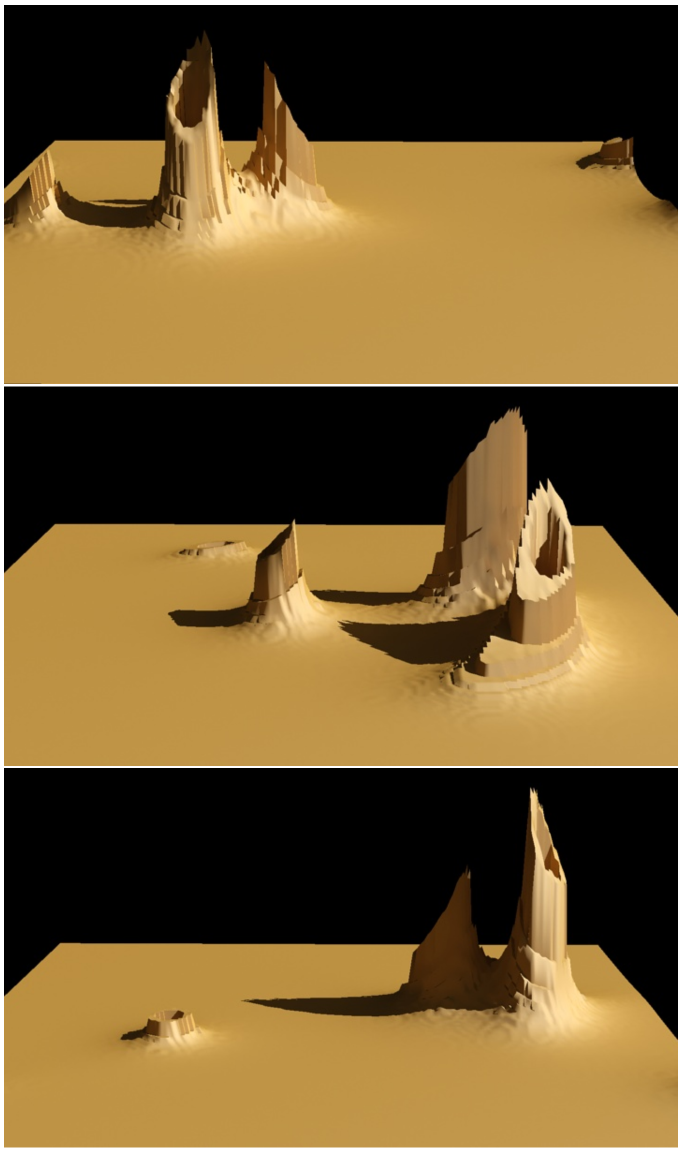


Figure 4.13: Exotic terrains generated by TP 4.5

$$TP = myDivide(myDivide(myDivide(myDivide(plane(1), fftGen(4.00)), fftGen(4.00)), fftGen(4.00)), fftGen(4.00)) \quad . \tag{4.5}$$

achieve a coral looking terrain.

In the set of pictures from Figures 4.14, 4.15, 4.16, 4.17 and 4.18, it is visible that the terrains generated by each TP are allways different, but still present the same features.

4.3.3 Level of Detail

A detailed terrain model involves a huge amount of polygons to be represented, even when considering only the portion of the scene that is visible. Clark suggested [17] using simpler versions of the geometry for objects that had lesser visual importance, such as those far away from the viewer. These simplifications are called Levels of Detail (LODs) and allow adapting structures, such as terrains, to the processing power requirements. The third experiment was conducted to test the perseverance of terrain features across several LODs (or resolutions) and the consequent impact in generation time on our evolutionary tool [2].

A set of TPs was chosen to generate terrains with grid sizes from 50 to 450 with increments of 50. To perform these tests it was necessary to modify the terminals in order to include the variable s to specify the resulting height map size. However, this variable is not evolved, it is like it does not exists from the GP point of view.

The selected TPs are shown in Eq. 4.11, 4.12, 4.13 and 4.14. Figures 4.19, 4.20, 4.21 and 4.22 present the results at three different LODs with grid sizes of 50×50, 150×150 and 450×450.

In this experience all TPs have preserved their main features independently of the chosen grid size. Due to terminals' randomness consecutive calls of the same terminal will always generate a slightly different height map. This is a desired characteristic, that can be controlled by fixating the random number seed. However, this approach does not work for generating terrains at different LODs, because the amount of necessary random numbers will vary accordingly with the chosen LOD. This explains the differences from terrains at different LODs generated by the same TP.

Figure 4.23 shows the average time of 10 execution of each TP at each grid size on a Pentium Core 2 Duo at 1,66 GHz with 2 GB of RAM. As expected, the generation time increases at a quadratic pace with the increase of the number of grid points, e.g. for TP 4.14 from $18,4$ ms at 50×50 to $1066,0$ ms at 450×450. The generation time also increased, as anticipated,

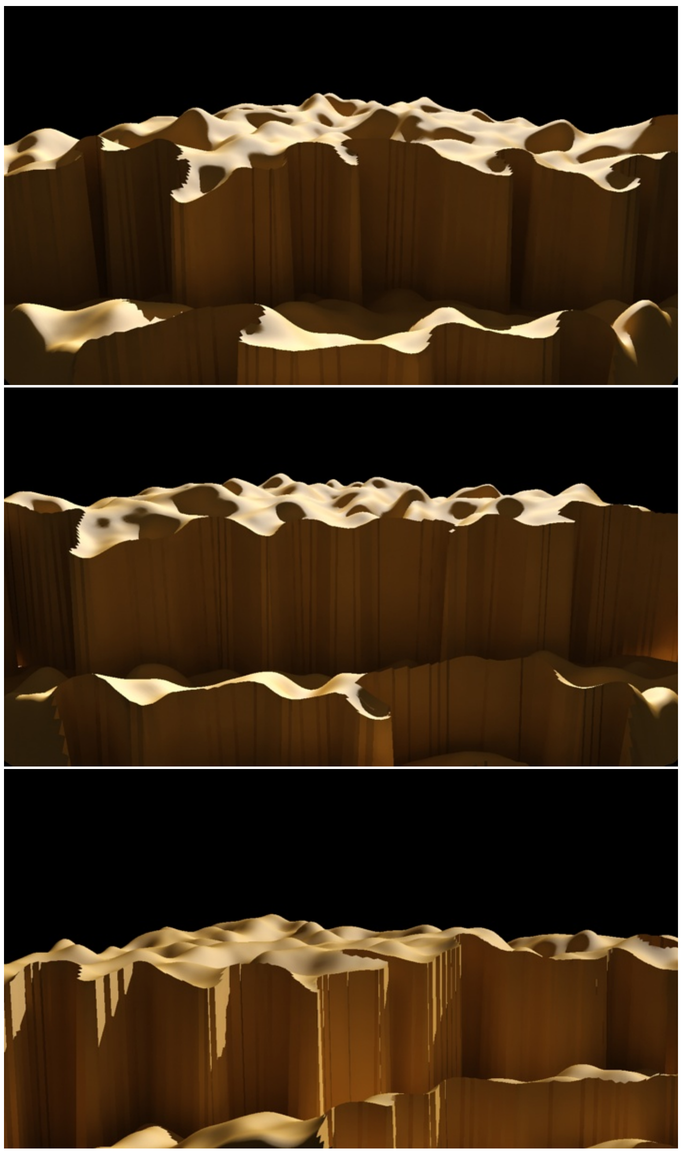


Figure 4.14: Terrain evolved with cliffs in mind, generated by TP 4.6

$$TP = \quad myMod(smooth(smooth(fftGen(0.50))), smooth(plane(5))) \ . \tag{4.6}$$

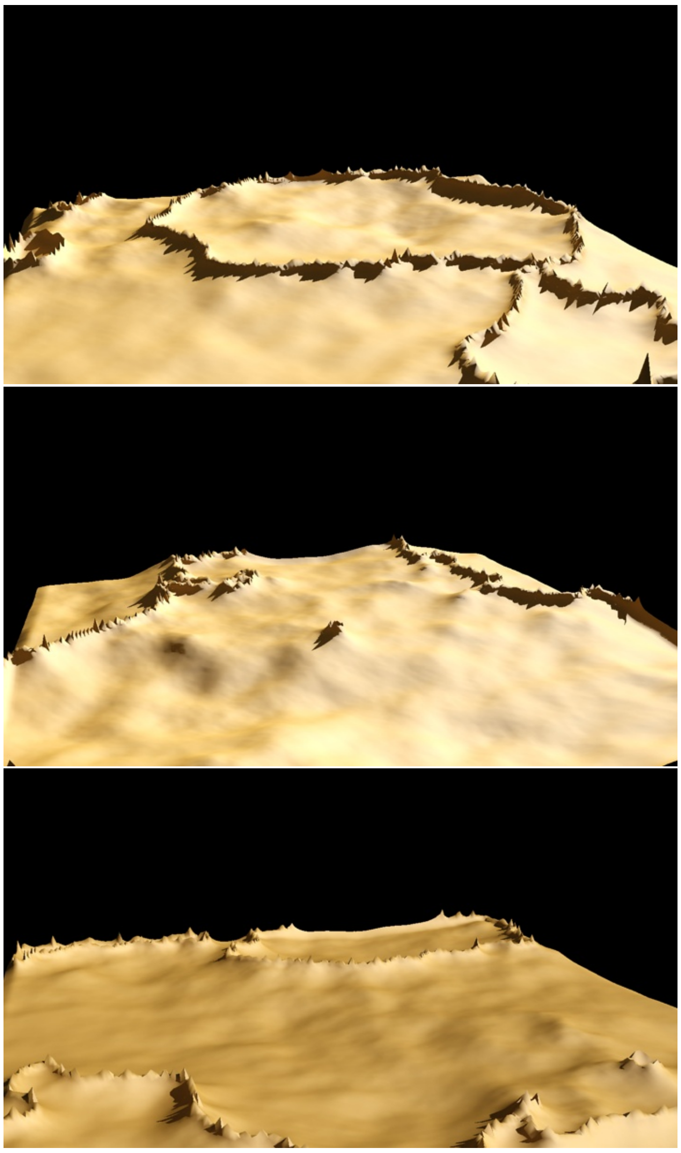


Figure 4.15: Terrain evolved with corals in mind, generated by TP 4.7

$$\begin{aligned} TP = \quad & myLog(minus(fftGen(2.75), myLog(minus(smooth(\\ & fftGen(1.50)), fftGen(2.50))))) \ . \end{aligned} \tag{4.7}$$

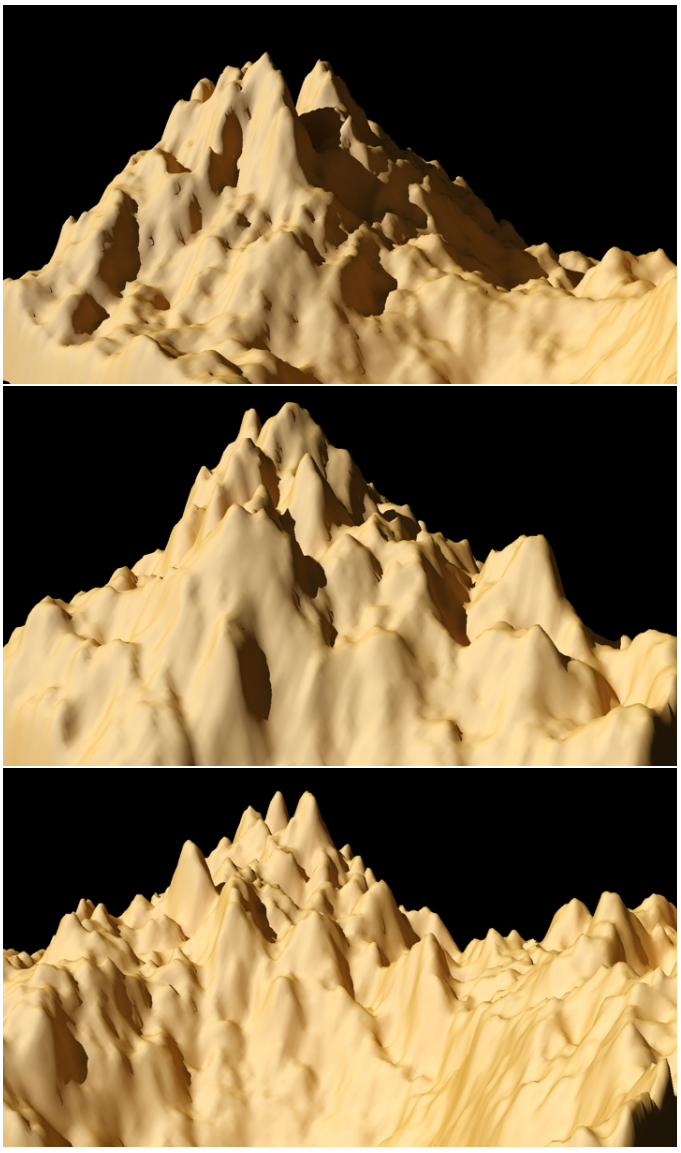


Figure 4.16: Terrain evolved with mountains in mind, generated by TP 4.8

$$TP = \quad times(sin(fftGen(3.00)), smooth(times(sin(cos(sin(cos(times(fftGen(1.75), fftGen(0.75)))))), fftGen(0.50)))) \quad . \tag{4.8}$$

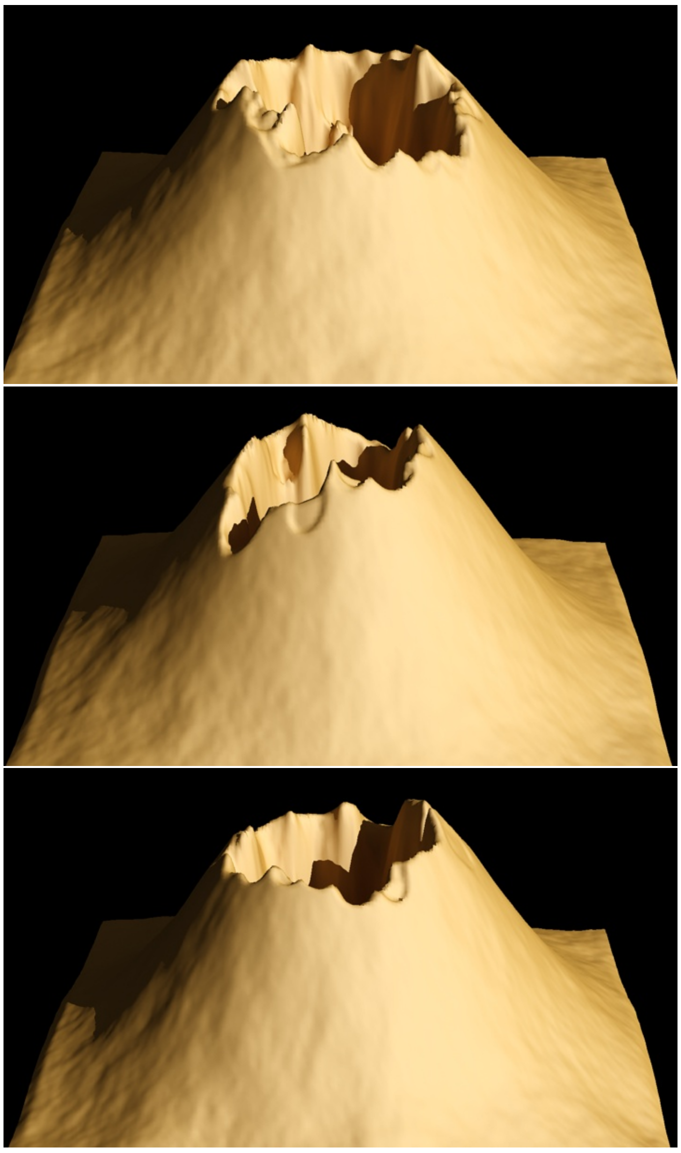


Figure 4.17: Terrain evolved with volcanoes in mind, generated by TP 4.9

$$TP = plus(fftGen(2.00), smooth(myMod(gauss(0.75), cos(fftGen(1.00))))) \ . \tag{4.9}$$

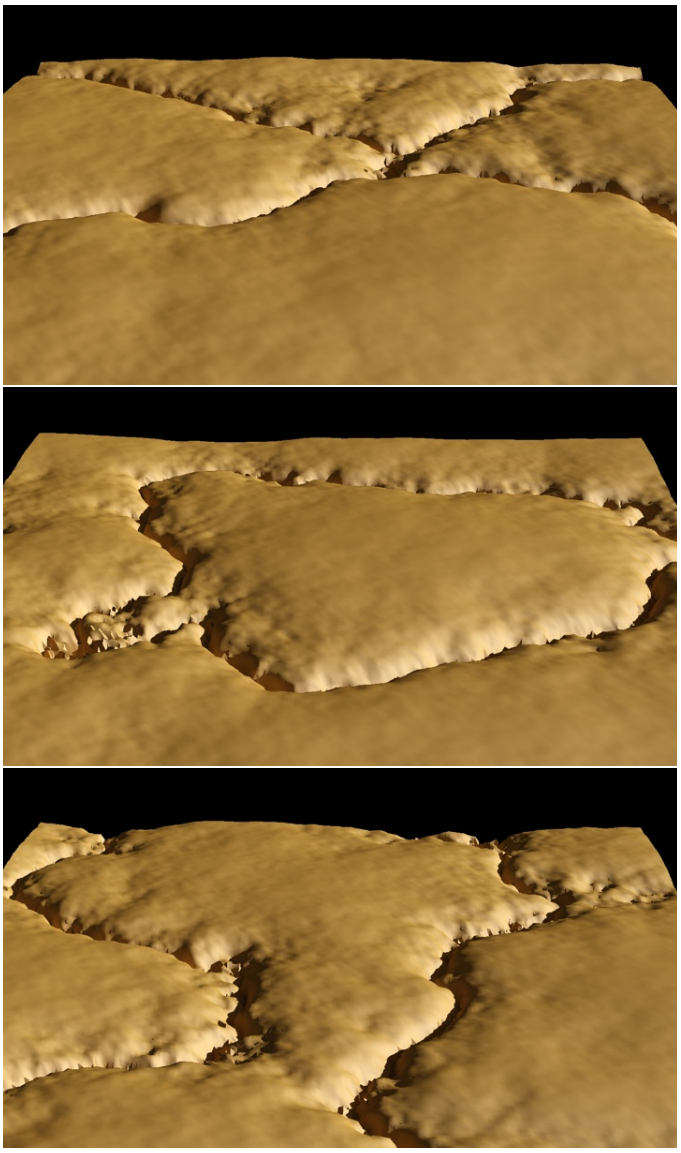


Figure 4.18: Terrain evolved with rivers in mind, generated by TP 4.10

$$TP = \quad myLog(myLog(cos(minus(fftGen(2.00), fftGen(3.75))))) \ . \quad (4.10)$$

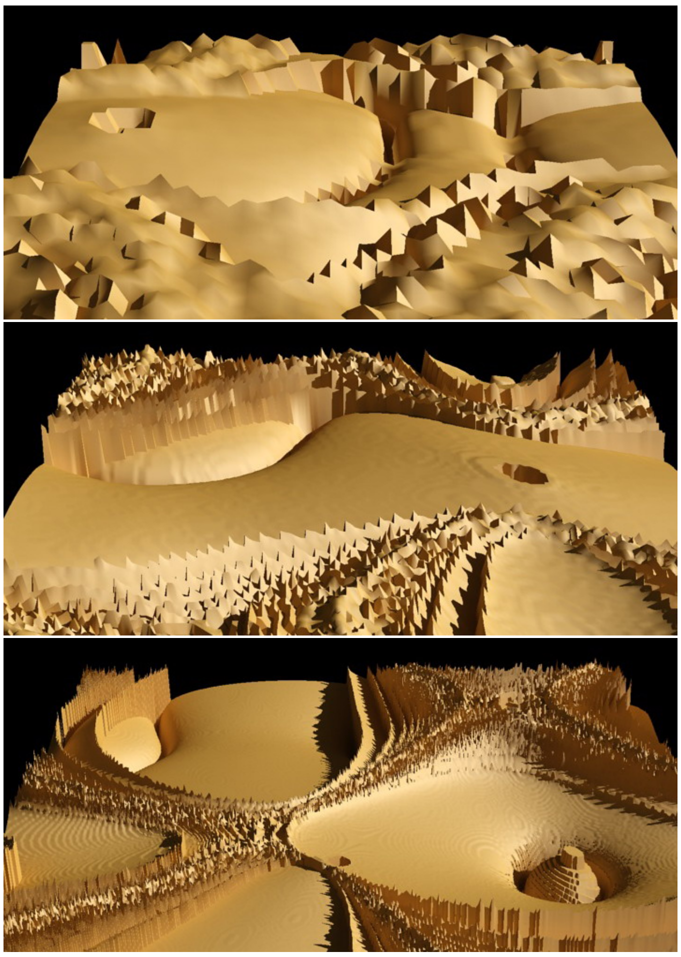


Figure 4.19: Exotic terrain generated by TP 4.11, with resolutions 50*50, 150*150 and 450*450

$$TP = myLog(myLog(myMod(myLog(fftGen(s, 3.75)), myLog(myLog(fftGen(s, 4.25)))))) \ . \tag{4.11}$$

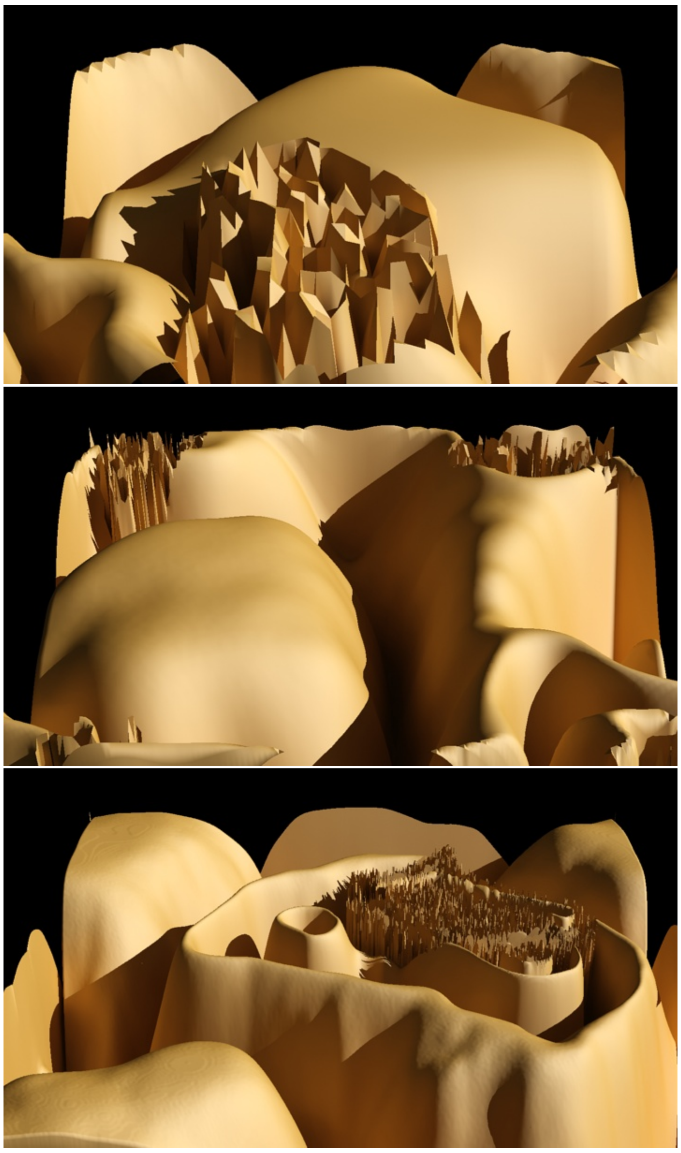


Figure 4.20: Exotic terrains generated by TP 4.12, with resolutions 50*50, 150*150 and 450*450

$$\begin{aligned} TP = \quad & myPower(cos(myDivide(myLog(smooth(fftGen(s, 2.75))), myMod(\\ & sin(fftGen(s, 0.50)), myDivide(myLog(smooth(fftGen(s, 2.75))), \\ & myMod((sin(fftGen(s, 0.50))), fftGen(s, 2.25)))))) \ . \end{aligned} \tag{4.12}$$

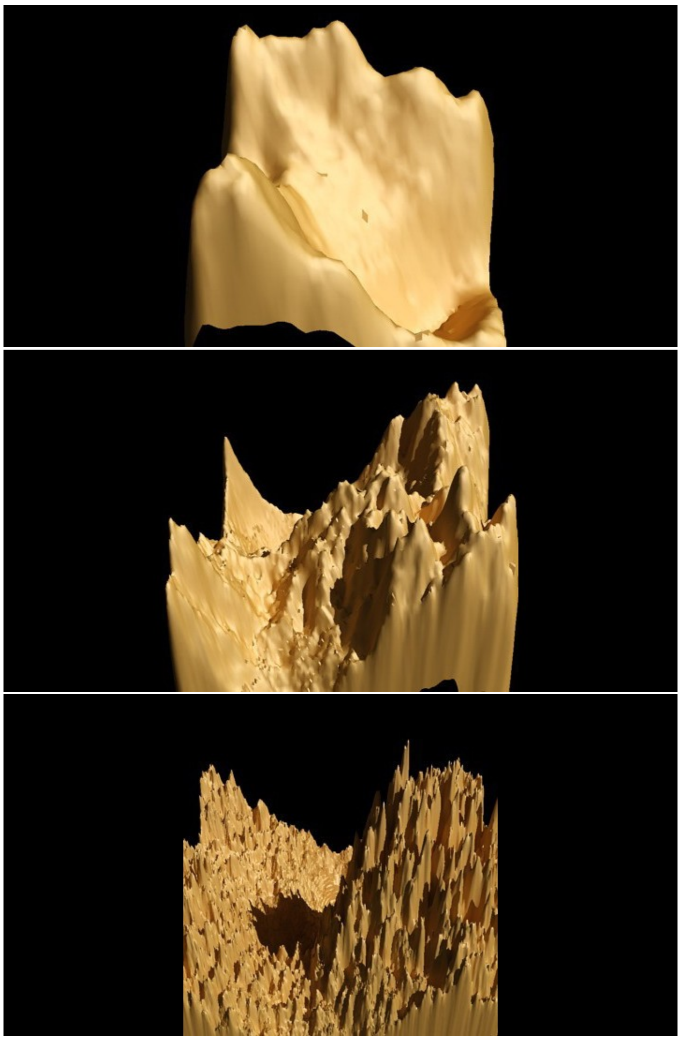


Figure 4.21: Mountains generated by TP 4.13, with resolutions 50*50, 150*150 and 450*450

$$\begin{aligned} TP = \quad & times(sin(fftGen(s,3.00)), smooth(times(sin(cos(sin(cos(times(\\ & fftGen(s,1.75), fftGen(s,0.75)))))), fftGen(s,0.50)))) \ . \end{aligned} \tag{4.13}$$

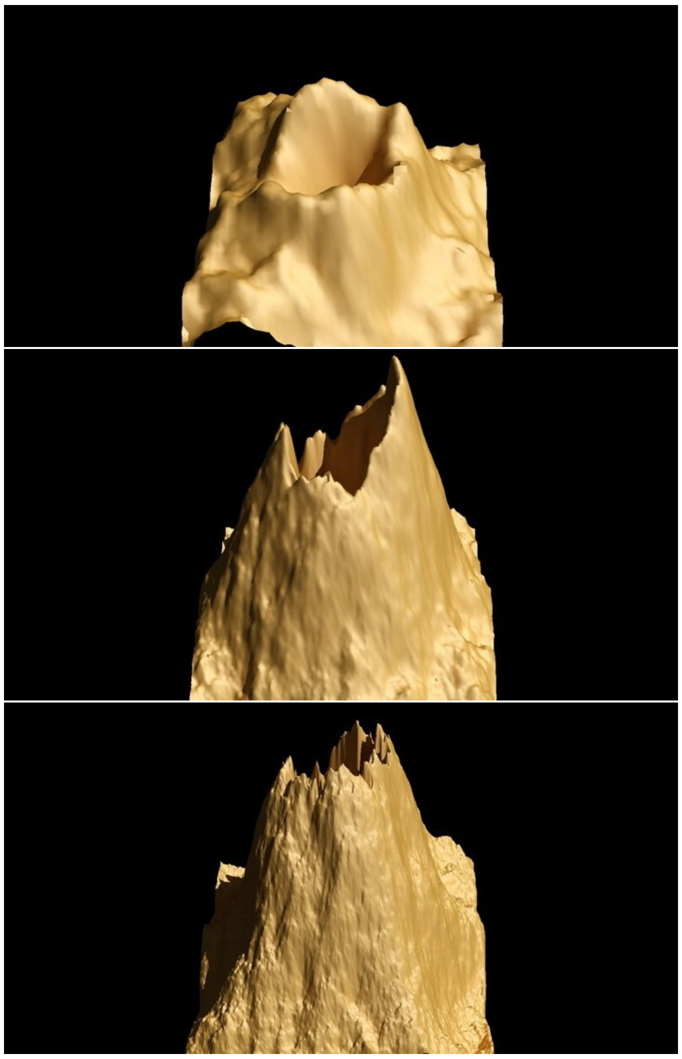


Figure 4.22: Volcanoes generated by TP 4.14, with resolutions 50*50, 150*150 and 450*450

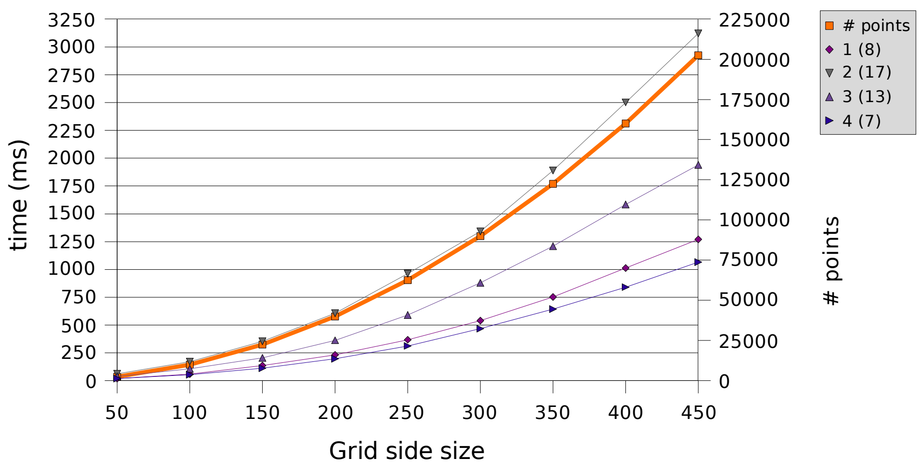
$$\begin{aligned} TP = \quad & plus(fftGen(s, 2.00), smooth(myMod(gauss(s, 0.75), cos(\\ & fftGen(s, 1.00))))) \ . \end{aligned} \tag{4.14}$$

with the number of TP's nodes.

The time values presented on Figure 4.23 are the times for generating each individual. The time to generate the entire population must be multiplied by the population size. For TP 4.12 (with 17 nodes) with a resolution of 450×450, each individual takes $3,122$ seconds. So, to generate an entire population of 12 individuals it will be necessary $37,464$ seconds. A delay of this magnitude is not negligible and will have a negative impact on the response time of interactive application such as our tool. Accordingly to Card et al. [57] and Testa et al. [58]:

- $0,1$ second is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result;
- $1,0$ second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data;
- 10 seconds is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect;

The response times should be as fast as possible to keep user's attention focused on the application. Our goal is to keep generation times for the entire population arround 1 second and never exceding 10 seconds. The generation time depends on the chosen resolution and on each individual's number of nodes, which tends to increase with the number of GP generations. So, from the responsiveness point of view, the use of the lowest resolutions for the evolutionary phase is better. However, if the used resolution is too low the output might not represent all the terrain features, specially small details, and force the designer to use the analyse window more often. This will increase the time needed by the designer to chose the best terrain at each generation and consequently the overall time to acheive the desired terrain. A compromise must be made between the terrain resolution for the evolutionary phase and the application responsiveness. From our set of experiences we



grid size	**Time (ms)** 1 (TP 4.11)	2 (TP 4.12)	3 (TP 4.13)	4 (TP 4.14)
50	18.5	61.4	37.7	18.4
100	59.6	171.9	106.9	53.9
150	136.4	355.1	205.3	113.1
200	233.2	605.7	362.9	198.6
250	368.3	962.5	590.1	312.6
300	539.8	1342.3	878.7	466.8
350	753.5	1889.8	1210.3	643.4
400	1012.7	2501.1	1583.6	841.3
450	1269.8	3122.1	1939.1	1066.0
GP nodes	8	17	13	7

Figure 4.23: Terrain generation times versus grid sizes

found the grid size of 100×100 to be the best settlement. Short generation times will be also advantageous for the future implementation of automated terrain evolution.

Chapter 5

Conclusions

On this report we present the Genetic Terrain Programming technique for terrain generation. The idea behind this new approach is to use interactive evolution with GP to generate TPs. To employ this technique a first implementation of *GenTP* has been carried out with GPLAB and Matlab. Through a series of experiments we have shown that multiple execution of the same TP will allways generate differrent terrains, because of the randomness present on its terminals, but with the same features (e.g. mountains, valleys). With a single technique designers will be able to evolve very different kinds of TPs, from real looking terrains to more exotic ones with an alien semblance. Those TPs can be inserted in video games like any other procedural technique, to generate terrains, with the same features. Furthermore, through a series of experiments we have shown that the feature perseverance is independent - to a limited extent - of the chosen LOD. This means that during the evolutionary phase low LODs can be used without compromising the result. Consequently less time will be required for our evolutionary tool enabling it to be more responsive, which is an important characteristic of interactive tools.

Some game publishers require that all players have the same game "experience" if they make the same choices. They want to measure or obtain quality control both on the user experience side as well as on the development and testing end. This requirement seems to contradict our goal of achieving different terrains with the same TP. However, if a TP is incorporated on a video game as a procedural technique, our technique can deliver two levels of control regarding randomness. First, a specific TP will always generate terrains with the same features, this means that in spite of the present randomness those terrains are similar and not completely random. Second, if full control over the final terrain is required the seed for the random number generator can be kept the same across separate runs of the TP, allowing the same terrain to be regenerated as many times as desired.

Chapter 6

Future work

The potential shown by *GenTP* suggests several lines for future developments:

- Augment the GP terminal set in order to try obtain a wider range of realistic terrain types with fewer generations;
- Compose a terrain through the use of several TPs, where the generated terrains will have to be joined on a credibly and smooth way;
- Create a database of TPs and allow users to classify them and their resulting features. Upon such database, game programmers will be able to generate new terrains based on one TP for each localisation and build a landscape with the desired features composed by several TPs;
- Incorporate the Genetic Terrain Programming technique on a video game to automatically generate terrains;
- Implement the Genetic Terrain Programming technique as a Blender plugin (a 3D modeling tool), to increase the flexibility of our technique;
- Add more features to our technique so that whole scenarios, including vegetation and buildings, can be generated;
- Develop fitness functions to allow the automatic evolution of TP's and avoid designers fatigue (a common problem of interactive evolutionary applications [45]) to achieve the desired terrain;

On a later stage, we will try to involve a large number of users, by means of volunteer computing, to create a whole set of different TPs.

Bibliography

[1] Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Modelling video games' landscapes by means of genetic terrain programming - a new approach for improving users' experience. In Mario Giacobini et al., editor, *Applications of Evolutionary Computing*, volume 4974 of *LNCS*, pages 485–490, Napoli, Italy, 2008. Springer.

[2] Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Genetic terrain programming - an aesthetic approach to terrain generation. In *Computer Games and Allied Technology 08*, pages 1–8, Singapore, 2008.

[3] Miguel Frade, F. Fernandez de Vega, and Carlos Cotta. Gentp – uma ferramenta interactiva para a geração artificial de terrenos. In M. Perez Cota, editor, *Third Iberian Conference in Systems and Information Technologies (CISTI 2008)*, volume 2, pages 655–666, Ourense, Spain, 2008. LibroTeX. ISBN 978-84-612-4840-7.

[4] Georgios N. Yannakakis and John Hallam. A scheme for creating digital entertainment with substance. In David W. Aha, Héctor Muñoz-Avila, and Michael van Lent, editors, *Proceedings of the Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 119–124, Edinburgh, UK, 2005. IJCAI.

[5] Ryan L. Saunders. Terrainosaurus - realistic terrain synthesis using genetic algorithms. Master's thesis, Texas A&M University, 2006.

[6] Mark Duchaineau, Murray Wolinsky, David Sigeti, Mark Millery, Charles Aldrich, and Mark Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[7] Sheng Li, Xuehui Liu, and Enhua Wu. Feature-based visibility-driven

CLOD for terrain. In *Pacific Graphics*, page 313–322, Los Alamitos, CA, 2003. IEEE Computer Society Press.

[8] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, 2004.

[9] R. Pajarola, M. Antonijuan, and R. Lario. Quadtin: Quadtree based triangulated irregular networks. In *Proceedings IEEE Visualization*, page 395–402. IEEE Computer Society Press, 2002.

[10] Alex Kelley, Michael Malin, and Gregory Nielson. Terrain simulation using a model of stream erosion. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 263–268, NY, USA, 1988. ACM.

[11] Jacob Olsen. Realtime procedural terrain generation - realtime synthesis of eroded fractal terrain for use in computer games. *Department of Mathematics And Computer Science (IMADA), University of Southern Denmark*, 2004.

[12] Heinz-Otto Peitgen, Hartmut Jürgens, and Dietmar Saupe. *Chaos and Fractals - New Frontiers of Science.* Springer, 2nd edition, 2004.

[13] Richard Voss. Fractals in nature: characterization, measurement, and simulation. *SIGGRAPH*, 1987.

[14] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, 1982.

[15] C. Darwin. *On the Origin of Species by Means of Natural Selection.* John Murray, 1859.

[16] Teong Joo Ong, Ryan Saunders, John Keyser, and John J. Leggett. Terrain generation using genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1463–1470, NY, USA, 2005. ACM.

[17] J.H. Clark. Hierarchical geometric models for visible-surface algorithms. *Proceedings of the 3rd annual conference on Computer graphics and interactive techniques*, pages 267–267, 1976.

[18] Benoit B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman, 1983.

[19] B. Pelton and D.C. Atkinson. Flexible Generation and Lightweight View-Dependent Rendering of Terrain. *School of Engineering Technical Report COEN-2003-01*, 22, 2003.

[20] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[21] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM.

[22] William B. Langdon and Adil Qureshi. Genetic programming – computers using "natural selection" to generate programs. Technical Report RN/95/76, University College London, Gower Street, London WC1E 6BT, UK, 1995.

[23] Thomas S. Ray. Evolution, ecology and optimization of digital organisms. 1992.

[24] Demetri Terzopoulos. Artificial life for computer graphics. *Communications of the ACM*, 42(8), 1999.

[25] David B. Fogel, Alan D. Blair, and Risto Miikkulainen. Special issue: Evolutionary computation and games. *IEEE Transactioon on Evolutionary Computation*, 9(6), 2005.

[26] Simon M. Lucas and Graham Kendall. Evolutionary computation and games. *IEEE Computational Intelligence Magazine*, 2006.

[27] David B. Fogel. Using evolutionary programming to create neural networks that are capable of playing Tic-Tac-Toe. *Proceedings of IEEE International Conference on Neural Networks IEEE*, 1993.

[28] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 1959.

[29] Wikipedia. Computer go. Website, 2007. http://en.wikipedia.org/wiki/Computer_Go.

[30] Simon M. Lucas. Evolving a neural network location evaluator to play ms. pac-man. *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, pages 203–210, 2005.

[31] John E. Laird. Using a computer game to develop advanced AI. *Computer*, 2001.

[32] J. Denzinger, Kevin Loose, D. Gates, and J. Buchanan. Dealing with parameterized actions in behavior testing of commercial computer games. *Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2005.

[33] Kenneth O. Stanley, Bobby D. Bryant, Igor Karpov, and Risto Miikkulainen. Real-time evolution of neural networks in the nero video game. *The Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI06)*, 2006.

[34] Dynamical & Evolutionary Machine Organization. Tron. Website, 2000. http://helen.cs-i.brandeis.edu/tron/.

[35] Pablo Funes and Jordan B. Pollack. Measuring progress in coevolutionary competition. In et.al. Meyer, editor, *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior.*, pages 450–459. MIT Press, 2000.

[36] The RoboCup Federation. RoboCup. Website, 2007. http://www.robocup.org/.

[37] Ivan Tanev, Michal Joachimczak, and Katsunori Shimohara. Evolution of driving agent, remotely operating a scale model of a car with obstacle avoidance capabilities. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM Press, 2006.

[38] J. Togelius and M. Simon. Evolving controllers for simulated car racing. *Proceedings of the Congress on Evolutionary Computation 2005*, 2005.

[39] Dario Floreano, Toshifumi Kato, Davide Marocco, and Eric Sauser. Coevolution of active vision and feature selection. *Biol. Cybern.*, 2004.

[40] Bruce Damer, Karen Marcelo, and Frank Revi. Nerve garden: A virtual terrarium in cyberspace. *American Association for Artificial Intelligence*, 1998.

[41] Wikipedia. Creatures: artificial life program. Website, 2007. http://en.wikipedia.org/wiki/Creatures_%28artificial_life_program%29.

[42] Gameware Development. Introduction to the genetics in Creatures 2. Website, 2007. http://www.gamewaredevelopment.co.uk/creatures_more.php?id=463_0_6_0_M27.

[43] Wikipedia. Spore video game. Website, 2007. `http://en.wikipedia.org/wiki/Spore_(video_game)`.

[44] Peter Bentley. Aspects of evolutionary design by computers. *In Advances in Soft Computing - Engineering Design and Manufacturing, Springer-Verlag*, 1998.

[45] Peter Bentley. *Evolutionary Design by Computers*. Morgan Kaufmann Publishers, Inc., CA, USA, 1999.

[46] Karl Sims. Artificial evolution for computer graphics. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, pages 319–328, NY, USA, 1991. ACM.

[47] Karl Sims. Interactive evolution of dynamical systems. In F. Varela and P. Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 171–178, Paris, FR, 1992. MIT Press.

[48] Tatsuo Unemi. A design of multi-field user interface for simulated breeding. In *Proceedings of the Third Asian Fuzzy and Intelligent System Symposium*, pages 489–494, Masan, Korea, 1998.

[49] Tatsuo Unemi. SBART 2.4: breeding 2D CG images and movies and creating a type of collage. In *The Third International Conference on Knowledge-based Intelligent Information Engineering Systems*, pages 288–291, Adelaide, Australia, 1999. IEEE.

[50] Tatsuo Unemi. SBART 2.4: an IEC tool for creating 2D images, movies, and collage. In *Proceedings of 2000 Genetic and Evolutionary Computational Conference*, page 153, NV, USA, 2000.

[51] P. Machado and A. Cardoso. NEvAr - the assessment of an evolutionary art tool. In G. Wiggins, editor, *Proceedings of the AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science 2000*, Birmingham, UK, 2000.

[52] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Lulu.com and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[53] John R. Koza. Human-competitive results produced by genetic programming. Website, 2004. `http://www.genetic-programming.com/humancompetitive.html`.

[54] J. R. Koza. Genetic programming. on the programming of computers by means of natural selection. *Cambridge MA: The MIT Press.*, 1992.

[55] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[56] Sean Luke and Liviu Panait. Lexicographic parsimony pressure. In W. B. Langdon et al., editor, *GECCO-2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 829–836. Morgan Kauffman, 2002.

[57] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 181–186, New York, NY, USA, 1991. ACM.

[58] Charles J. Testa and Douglas B. Dearie. Human factors design criteria in man-computer interaction. In *ACM 74: Proceedings of the 1974 annual conference*, pages 61–65, New York, NY, USA, 1974. ACM.