Universidad de Extremadura

Evolucionando Terrenos Artificiales con Programación Genética Automatizada de Terrenos

Evolving Artificial Terrains with Automated Genetic Terrain Programing

Miguel Monteiro de Sousa Frade

PhilisophæDoctor (PhD) Dissertation

July 2012, Spain

Organization:

Dept. de Tecnologías de los Computadores y de las Comunicaciones
Universidad de Extremadura

Title:

Evolucionando Terrenos Artificiales con Programación Genética Automatizada de Terrenos

Evolving Artificial Terrains with Automated Genetic Terrain Programming

Author:

Miguel Monteiro de Sousa Frade
E-mail: miguel.frade@ipleiria.pt
School of Technology and Management, Polytechnic Institute of Leiria, Portugal

Supervisors:

Dr. Francisco Fernández de Vega
E-mail: fcofdez@unex.es
Centro Universitario de Mérida, Universidad de Extremadura, Spain

Dr. Carlos Cotta Porras
E-mail: ccottap@lcc.uma.es
ETSI Informática, Campus de Teatinos, Universidad de Málaga, Spain

Dr. D. Francisco Fernández de Vega, professor titular de la Universidad de Extremadura y Dr. D. Carlos Cotta Porras, professor titular de la Universidad de Málaga,

CERTIFICAN:

que la presente memoria, titulada "Evolucionando Terrenos Artificiales con Programación Genética Automatizada de Terrenos" ha sido realizada por D. Miguel Monteiro de Sousa Frade, bajo la dirección del Dr. Francisco Fernández de Vega y el Dr. Carlos Cotta Porras, en el Departamento de Tecnología de los Computadores y de las Comunicaciones de la Universidad de Extremadura.

Y para que conste, y en cumplimiento d ela legislación vigente, firma la presente,

Dr. Francisco Fernández de Vega. Dr. Carlos Cotta Porras

Resumen

La industria del videojuego afronta en la actualidad un gran reto: mantener el coste del desarrollo de los proyectos bajo control a medida que estos crecen y se hacen más complejos. La creación de los contenidos de los juegos, que incluye el modelado de personajes, mapas y niveles, texturas, efectos sonoros, etc, representa una parte fundamental del costo final de producción. Por eso, la industria está cada vez más interesada en la utilización de métodos procedurales de generación automática de contenidos, para amplificar la efectividad de las inversiones en los procesos de diseño de videojuegos. Sin embargo, crear y afinar los métodos automáticos de generación de contenidos no es una tarea trivial.

En esta memoria, se describe un método procedural basado en Programación Genética, que permite la generación automática de terrenos para videojuegos. Los terrenos presentan características estéticas, y no requieren ningún tipo de parametrización para definir su aspecto. Así, el ahorro de tiempo y la reducción de costes en el proceso de producción es notable. Para conseguir los objetivos, se utiliza Programación Genética de Terrenos (Genetic Terrain Programming, GTP, en inglés).

La primera implementación utilizada de GTP utilizó un método basado en Evolución Interactiva, en la que la presencia del usuario que guía el proceso evolutivo es imprescindible. A pesar de los buenos resultados, el método está limitado por la fatiga del usuario (algo común en los métodos interactivos). Para resolver esta cuestión se desarrolla un nuevo modelo de GTP en el que

el proceso de búsqueda es completamente automático, y dirigido por una función de aptitudo. La función considera accesibilidad de los terrenos y perímetros de los obstáculos. Los resultados obtenidos se incluyeron como parte de un videojuego real.

Abstract

Nowadays video game industry is facing a big challenge: keep costs under control as games become bigger and more complex. Creation of game content, such as character models, maps, levels, textures, sound effects and so on, represent a big slice of total game production cost. Hence, video game industry is increasingly turning to procedural content generation to amplify the cost-effectiveness of the efforts of video game designers. However, creating and fine tunning procedural methods for automated content generation is a time consuming task.

In this thesis we detail a Genetic Programming based procedural content technique to generate procedural terrains. Those terrains present aesthetic appeal and do not require any parametrization to control its look. Thus, allowing to save time and help reducing production costs. To accomplish these features we devised the Genetic Terrain Programming (GTP) technique.

The first implementation of GTP used an Interactive Evolutionary Computation (IEC) approach, were a user guides the evolutionary process. In spite of the good results achieved this way, this approach was limited by user fatigue (a common trait of IEC systems). To address this issue a second version of GTP was developed where the search is automated, being guided by a direct fitness function. That function is composed by the weighted sum of two morphological metrics: terrain accessibility and obstacle edge length. The combination of the two metrics allowed us remove the human factor form the evolutionary process and to find a wide range of aesthetic and fit

terrains. Procedural terrains produced by this technique are already in use in a real video game.

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, suggestions and encouragement during the research period.

I would like to express also my gratitude to the following projects, that directly or indirectly funded this research: TIN2007-68083-C02-01, TIN2008-05941, and TIN2011-28627-C04 from Spanish Ministry of Science and Education; GRU-09105, GR10029 from Gobierno de Extremadura; TIC-6083 from Junta de Andalucia; and the Chapas Video Game project, a partnership between Centro Universitario de Mérida (Universidad de Extremadura), GLOW (an animation studio) and Junta de Extremadura, who made possible the development of Chapas to test our technique in a real video game. Finally a word of appreciation to the Polytechnic Institute of Leiria, Portugal who welcomed me in their laboratories with support from the PROTEC grant SFRH/BD/49610/2009 of Portuguese National Foundation for Science and Technology (FCT).

I am also deeply grateful to the Informatics and Communications Services of Computer Science department from University of Coimbra, Portugal, for giving us a time slot on 18 nodes of their MILIPEIA cluster. Also, a special word of appreciation to Patrício Domingues that made the required arrangements that made possible the use of the cluster. Without their generosity most of our tests would not have been possible.

A big thank to my brother, Nuno Frade, for his helpful assistance to generate some of the 3 dimensional renders of the terrains.

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

Abbreviations

AI	Artificial Intelligence
DEM	Digital Elevation Models
EA	Evolutionary Algorithms
EC	Evolutionary Computation
ERC	Ephemeral Random Constant
GIS	Geographic Information Systems
GP	Genetic Programming
GTP	Genetic Terrain Programming
GTP_a	Automated Genetic Terrain Programming
GTP_i	Interactive Genetic Terrain Programming
GUI	Graphical User Interface
H	Hurst index
IEC	Interactive Evolutionary Computation
NPC	Non-Player Character
PCG	Procedural Content Generation
RPG	Role-Playing Game
SBPCG	Search Based Procedural Content Generation
TIN	Triangular Irregular Network
TP	Terrain Program

Chapter 1

Introduction

Video games constitute a crucial area of the entertainment industry, with impressive financial investments. For the top publishers, gaming businesses is becoming increasingly more similar to Hollywood: each new game is a costly bet that can generate big profits, or big losses. M2 Research estimates the production cost of high quality video games for the 7^{th} generation consoles at $10 million for one platform and $18-$28 million for multiple platforms (Meloni, 2010). Conversely, prior console generations had development costs ranging between $3-5 million per platform. This cost increase is driven mainly by the higher complexity of new and more powerful hardware and the effort required to fully exploit its capacity to present players with richer content (Loftus, 2011). With costs increasing at this pace, video game industry is facing a big challenge: keep costs under control as games become bigger and more complex.

Creation of game content, such as character models, maps, levels, textures, sound effects, animations and so on, represent a big slice of total production costs (Edwards, 2006). On *Gears Of War* [1] these costs represented the largest share with 25%, see Fig. 1.1 (Rosmarin, 2006). Traditionally, the main techniques used in content development for video games have been

[1] video game published by Microsoft Game Studios in 2006

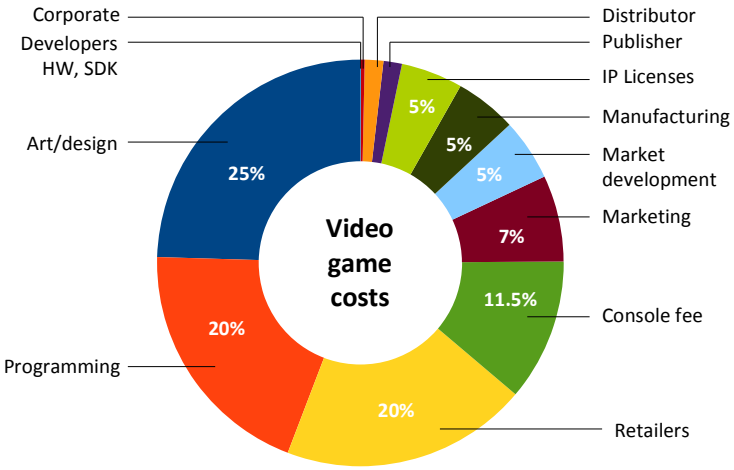


Figure 1.1: *Gears Of War* video game costs

artistry. Generally, all game content has been handcrafted by artists and designers working specifically to that end. This approach ensures game designers full control over their creations. Nevertheless, by delegating most or all of the details up to the designer, manual content production impose high requirements on designer in terms of time and effort, which has a huge impact on production costs. Therefore, game industry is increasingly turning to procedural generation techniques that allow the automation of content creation (Nelson and Mateas, 2007) and this way can save significant expenses.

Procedural content generation allows the automation of game content creation through algorithmic means and parametric control (Ebert et al, 2003). For instance, it is possible to generate a forest where each plant specie is represented by a set of parameters and each tree is slightly different just by changing the seed for the pseudo-random numbers generator (Lane and Prusinkiewicz, 2002; Prusinkiewicz and Lindenmayer, 2004). This allows the amplification of designers inputs: a few parameters yield large amounts of details (Ebert et al, 2003). Therefore, it will require less effort and time than modeling techniques to create complex content. Procedural techniques also allow more dynamic processes during the game development cycle. Designers can change the location of some level elements without having to redraw everything else. The procedural content can have rules built in to automat-

ically adapt to those changes, thus allowing to save precious development resources.

Financial benefits are not the only advantage of procedural content generation techniques. The representation of procedural content is also extremely compact and can be measured in *Kilobytes*, while others require *Megabytes* of storage. One good example of compactness is the classic game *Elite* [2], which succeed to keep 8 galaxies of 256 planets each in a few tens of kilobytes by representing each planet with just a few numbers. When procedural algorithms produce the same content given the same parameters they can also be considered as a form of data compression. Another advantage that some procedural content has is the ability to be computed at any desired resolution. Fractals are a good example of this characteristic (Mandelbrot, 1983). A third advantage is that the quality of procedural generated content is independent of user skills, therefore they can be used by people without the designing skills that are required for handcrafted content. Furthermore, procedural content generation techniques can also allow the emergence of new types of games, where their contents can be created according to some criteria like player satisfaction, challenge, novelty, etc. This might lead to the creation of games that never end, that whatever the player does or whenever he goes, there always will be something new to explore. Although this kind of games is not yet a reality, they would have infinite replay value. Finally, procedural generation algorithms have the potential to generate entirely new content designs, that challenge human imagination. These new designs can serve as inspiration and serve as a base for designers to create their own creations (Togelius et al, 2011).

However, procedural content generation has also its own drawbacks. One disadvantage is its evaluation. This operation requires intense computations which can be very expensive. Another disadvantage is the modeling problem: how to achieve the desired features? Typically, procedural methods offer a set of parametric controls that enable a procedure to generate many

[2] video game published by Acornsoft in 1984

different outputs. To make a procedure more flexible, more controls can be added. While the power of a procedure may be enhanced in this way, the resulting interface can become overly complex. In the case of a human using the interface, coming up with good results from a powerful procedure often degenerates into an authoring processing of trial and error. Besides, procedural algorithms present a certain degree of unpredictability: a small change in one parameter can result in big changes on the outcome, or big changes might not result in any significant modification. Whatever is the case, designers end up performing a lot of tests and simulations until they learn how the procedural system behaves to tune it. The search for the right input parameters and algorithm tune to achieve the desired output is time consuming. For example, the development of "Far Cry 2" [3] video game took as much as 15 times more time to refine and tune procedural tools than the amount of time developing the underlying game engine (Remo, 2008).

1.1 Aims and Contributions

Among the many video game contents that can be generated procedurally, we find artificial terrains, the main focus of our research. Artificial terrains have an important role in video games dynamics (Forbus et al, 2002) and contribute greatly to re-playability (Sampath, 2004). Nowadays there are a wide range of techniques for terrain generation, which are detailed in Section 3, but all of them present some constraints. More elaborated methods depend highly upon designer's skills, time and effort to obtain acceptable results or are not entirely procedural, which prevents them from being used in an automated fashion. The simpler methods allow only a narrow variety of terrains types and are difficult to model. Therefore, the aim of this research was to develop a new technique, employing Evolutionary Algorithms, to overcome the main constraints of current procedural methods, namely:

- the modeling problem, to avoid the time consuming process of proce-

[3] video game published by Ubisoft in 2008

dure tune to achieve the desired terrain features;

- and allow a broad range of terrain types with emphasis on aesthetic. All current procedural techniques are focused on the generation of realistic terrains. Although this is important, it might prevent designers from achieving their goals when they attempt to represent an alien or exotic looking terrain. Terrain novelty might have also a positive impact on video game's target audience and increase players interest.

This thesis presents and discusses a new procedural generation technique of terrains for video games, which was coined as Genetic Terrain Programming (GTP) (Frade et al, 2008c, 2009b). GTP is a search-based procedural content technique that employs Genetic Programing (GP) as an evolutionary search tool for procedural terrains, designated Terrain Programs (TPs). This technique combines the advantages of procedural content generation with the ability to create novel and innovative solutions of Evolutionary Algorithms. This approach allows the generation of new terrain types with novel looks and aesthetic appeal. The evolutionary search of terrains with the desired characteristics will produce TPs that do not require any parameter input to control its looks. Therefore, once evolved, TPs can be integrated in video games without the need for a human performing parameter tuning, thus allowing to save time and money. Another purpose of GTP could be its integration in authoring tools to inspire designer imagination and serve as base of their work.

Two versions of the GTP technique were developed during the research period. The first version, designated GTP_i, is interactive where a human guides the evolution process of TPs. This approach allowed the generation of many aesthetic procedural terrains, but also required expensive human resources for the evaluation process (more details in Chapter 4). Therefore, a second version was devised to eliminate the human intervention during the evolutionary phase. This new version, designated GTP_a, performs an automated search based on geomorphological metrics.

1.2 Thesis Structure

In this thesis we tried to organize the chapters from broader scope to more specific topics. Chapter 2 presents an introduction to Evolutionary Algorithms (EAs) in general, followed by a description of evolutionary design. Then, some examples of EAs applied to video games are shown and at the end of this chapter we present the inner working of genetic programming. Chapter 3 shows the different data structures available to represent terrains followed by a literature survey with existing terrain generation techniques. The work developed during the research period is presented in chapters 4 and 5. The first approach, GTP_i, to the research question is address in Chapter 4. Chapter 5 focus on the limitations of GTP_i, explains GTP_a solution to address them and finally presents a series of tests and discusses its results. To show the viability of our technique some TPs were incorporated in Chapas video game, detailed in Chapter 6. Conclusions and future work are laid out in Chapters 7 and 8 respectively.

Some additional information was added as appendices. Terrain Programs from all presented terrains throughout this thesis are displayed at Appendix A. Appendix B shows more detailed graphics from the tests presented in Chapter 5, which helped to interpret some results. A list of publications achieved during the research period is presented in Appendix C.

Chapter 2

Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a kind of bioinspired algorithms that apply Darwin's theory (Darwin, 1859) of natural evolution of the species, where living organisms are rewarded through their continued survival and the propagation of its own genes to successors. There are four main classes of EAs: genetic algorithms (GA) (Holland, 1975), evolutionary strategies (Rechenberg, 1971; Schwefel, 1977), genetic programming (GP) (Koza, 1992) and evolutionary programming (Fogel et al, 1966). They can be seen as search techniques (Langdon and Qureshi, 1995) and are able to achieve good approximate solutions to a large number of problems, thanks to their flexibility and adaptability to different search scenarios. Evolutionary algorithms do not make any assumption about the underlying search landscape and this characteristic is the key factor of success in such diverse fields as: engineering, art, biology, economics, marketing, genetics, operations research, robotics, social sciences and physics just to name a few.

More recently EAs have also been used to generate video game content - a field where our research fits in. Section 2.1 presents a summary of what has been done in this specific area. Another area where EAs have been employed is to design artifacts, Section 2.2 presents an overview of this topic. Finally, Section 2.3 presents an explanation of GP algorithm.

2.1 Evolutionary Algorithms in Video Games

Much of the early work on computational intelligence and games was directed toward applying Evolutionary Computation (EC) methods to evolve tactical and strategic content common to Non-Player Characters (NPCs) Artificial Intelligence (AI). The work on this area is well documented (Bourg and Seemann, 2004; Lucas and Kendall, 2006; Miikkulainen et al, 2006; Rabin, 2002). Therefore, the following paragraphs will focus instead on techniques to generate procedural content for video games, such as: tracks and levels; weapons; buildings and vegetation. There are also some research in this field regarding terrains, the main topic of this thesis, but they will be described in Chapter 3. The techniques that fit this category were designated Search-Based Procedural Content Generation (SBPCG) by Togelius et al (2011). They propose to search the right input parameters or to generate the procedure itself that will produce content with the desired characteristics.

SBPCG techniques incorporate a generate and test approach. After a candidate content instance is generated, it is tested according to some criteria. A test function (also called fitness function) grades the procedural content instead of simply accepting or rejecting it. Then, new content is produced, that is dependent on the score of previous content, and this way tries to find better scoring content. The process is repeated until the content is considered good enough (Togelius et al, 2011). Evolutionary algorithms are a perfect match for this approach, although not the only search mechanism of SBPCG. An overview of the SBPCG techniques employed to generate video game content is presented below.

Tracks and levels, represent an important aspect of video games as they impose restrictions or difficulties to players progress. Togelius et al (2006, 2007) designed a system for generation of tracks for a simple racing game. A racing track is created from a parameter vector by interpreting it as the parameters for b-spline (a sequence of Bezier curves). The resulting shape forms the mid-line of the racing track. Each candidate track is evaluated by

letting a neural network-based car controller drive on the track. The fitness of the track is dependent on the driving performance of the car: amount of progress, variation in progress and difference between maximum and average speed. Pedersen et al (2009) designed a user study focused on a version of the *Super Mario Bros* platform game to allow the creation of personalized levels. The fitness functions uses a neural network that converts level parameters and information about player's playing behavior to one emotional state predictors, such as: fun, challenge, frustration, predictability, anxiety and boredom. The neural networks were trained by collecting game play metrics and data from player reported emotions through a questionnaire. Sorenson and Pasquier (2010) presented a framework to generate levels for different but related game genres. They employ a Feasible-Infeasible Two-Population (FI-2Pop) genetic algorithm (Kimbrough et al, 2002) that was designed for constraints satisfaction problems. Level designers specify a set of constraints, which determine the basic requirements for a level to be considered playable. The *infeasible population* consists solely of levels which do not yet satisfy all these constraints, and these individuals are evolved towards minimizing the number of constraints violated. When individuals are found to satisfy all the constraints, they are moved to the *feasible population* where they are subjected to a fitness function that rewards levels based on any criteria specified by the level designers. Jennings-Teats et al (2010) also describes a framework, for 2 dimensional platform game, to generate levels at runtime that adapt their difficulty to player skills. Short level segments are used to collect data of player's behavior, level features and inquired ranking. The collected data was used to train a neural network to order segment levels by its difficulty. Then, while playing, the level is generated ahead of the player with a rhythm-based generation mechanism, that is ranked either higher or lower according to player's gaming skills.

Landscapes are another important facet of modern video games, in particular the ones that aim at representing realistic scenarios. They help to increase the immersion feeling on players. Some of the most critical features of a good landscape are: terrains (see Chapter 3), cities and vegetation. Mar-

tin et al (2010) designed an interactive system to generate 3D buildings for the commercial video game Subversion, which is being developed by Introversion, to allow users to procedurally build cities. Buildings are composed by a stack of three-dimensional objects, each described as a two-dimensional shape that is vertically extruded. They applied an IEC approach (Takagi, 2001), which we also applied in our first implementation, GTP_i (more details on Chapter 4). In each generation the user chooses two parent buildings and from them a screen of 16 new offspring are presented to the user. Each object or group of objects can be subject to various transform operations such as translation, rotation and union.

Lindenmayer (1968) devised L-systems with the aim to modulate plants and all kinds of vegetation. Parametric L-systems (Lindenmayer, 1974) are a powerful and flexible technique for plant modeling. However, it is a hard task to specify a PL-system, that generates a plant of a desired species. To address this issue Traxler and Gervautz (1996) used GAs to interactively find the appropriate production rules that produced the desired plant. Jacob (1996a,b) took another approach, using GP to evolve context sensitive PL-systems using plant characteristics, such as the number of blooms blossoming and the number of leaves, in the fitness function. Although none of these works were specifically developed with video games in mind, the content generated by them can be easily incorporated into a video game during its development phase.

The evolutionary approach to generate video game content also allows the appearance of new games types. Hastings et al (2009) propose a new algorithm to automatically generate game content while the game is played, based on the past preferences of the players. They developed Galactic Arms Race (GAR), a online multiplayer gaming platform, that is able to generate and evolve particle system weapons. The fitness of each weapon depends on how often the several users logged onto the same server choose to fire the weapon relative to how long it stays unused. This way players implicitly indicate their preferences and guide evolution without knowing the underlying system.

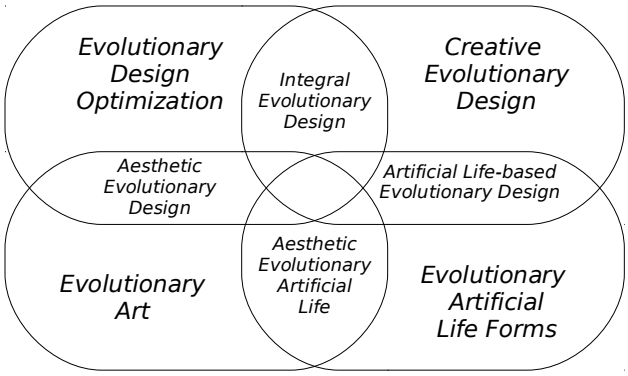


Figure 2.1: Evolutionary design categories

2.2 Evolutionary Design

Evolutionary design might be seen as a particular case of SBPCG techniques, where EAs are the search mechanism. 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 optimized selected parts of their designs using evolution, artists have used evolution to generate aesthetically pleasing forms (Machado et al, 2005), architects have evolved new building plans from scratch (Soddu, 2003), computer scientists have evolved morphologies and control systems of artificial life. Evolutionary design can be divided into four main categories (Bentley, 1998): evolutionary design optimization, 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 2.1. Our first implementation of GTP, GTP_i, fits in Aesthetic Evolutionary Design, a branch of the evolutionary art category. On the other hand, GTP_a fits in the evolutionary design optimization.

2.3 Genetic Programming

Genetic programming is the evolutionary algorithm used in the work presented in this thesis. Therefore, some details about its inner working are presented in this section. However, for the readers who wish to deepen their knowledge, the book *A Field Guide to Genetic Programming* from Poli et al (2008) offers a very good introduction and overview to GP. A thoroughly analysis on this topic is provided by the book *Genetic Programming - On the Programming of Computers by Means of Natural Selection* by Koza (1992), the main proponent of GP who has pioneered the application of genetic programming in various complex optimization and search problems.

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, after several generations a population of programs is stochastically transformed into new, hopefully better, populations of programs (Poli et al, 2008). 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 (Langdon and Qureshi, 1995):

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

There are now 36 instances where GP has automatically produced a result

Algorithm 2.1 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

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 20^{th} century invention; 6 instances where GP has done the same with respect to a 21^{st} century invention and 2 instances where GP has created a patentable new invention (Koza, 2004).

Algorithm 2.1 shows the basic steps of GP. The generated programs are run 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;

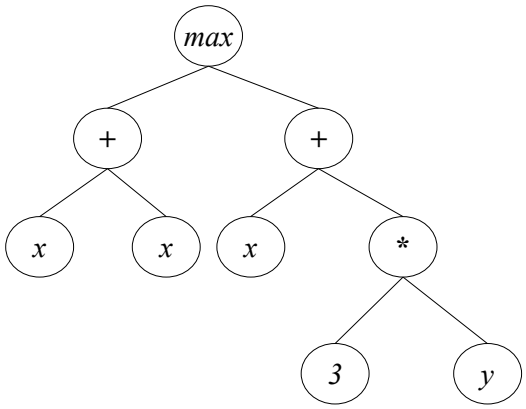


Figure 2.2: GP tree representation of $max(x + x, x + 3 * y)$. Adapted from (Poli et al, 2008).

2.3.1 Representation

In GP, programs are usually expressed as syntax trees rather than as lines of code. For example Figure 2.2 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 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.

2.3.2 Initializing the Population

Like in other evolutionary algorithms, the individuals in the initial GP population are typically randomly generated. There are a number of different approaches to generating this random initial population. In the following paragraphs we present a description of the two simplest methods, the *full*

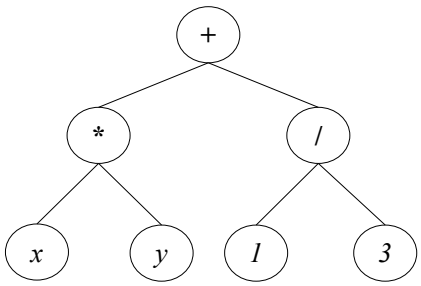


Figure 2.3: Example of a tree having maximum depth 2 created with the *full* initialisation method. Adapted from (Poli et al, 2008).

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 trees' 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 2.2 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 2.3 shows an example of a tree having maximum depth 2 created with the full initialization method, where all leaves are at the same depth. However, 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 2.4 illustrates an example of a tree created with the *grow* initialization method

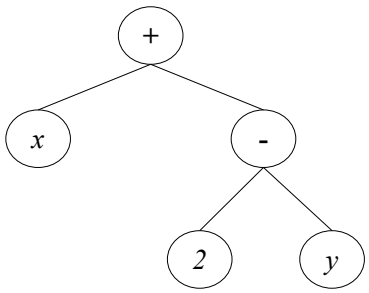


Figure 2.4: Example of a tree having maximum depth 2 created with the *grow* initialization method. Adapted from (Poli et al, 2008).

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 (1992) 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 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 initialization mechanisms check (Poli et al, 2008).

2.3.3 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 the ones proposed by Goldberg (1989); Luke

and Panait (2002):

- Fitness Proportional Selection;
- Lexicographic Parsimony Pressure Tournament;
- Doubletour;

2.3.4 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 $1 - p$ (Poli et al, 2008).

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

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

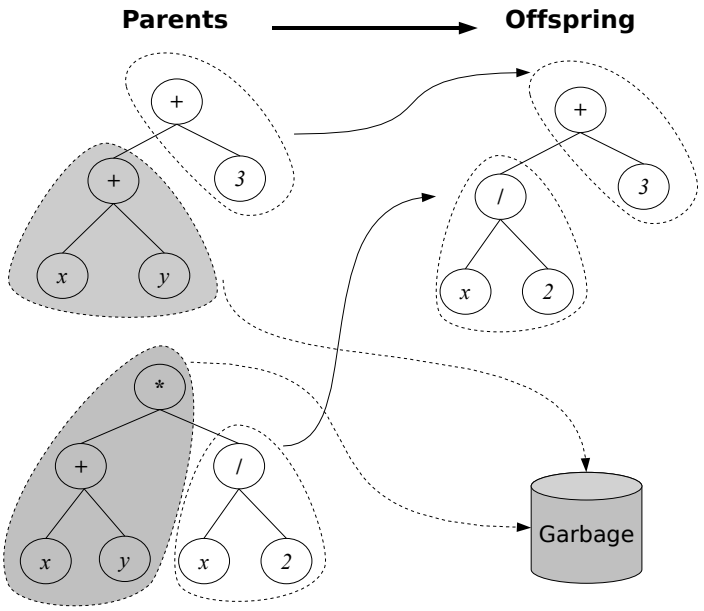


Figure 2.5: Example of subtree crossover (the trees on the left are copies of the parents). Adapted from (Poli et al, 2008).

illustrated in Figure 2.5. 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 (1992) suggested 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 (Poli et al, 2008):

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

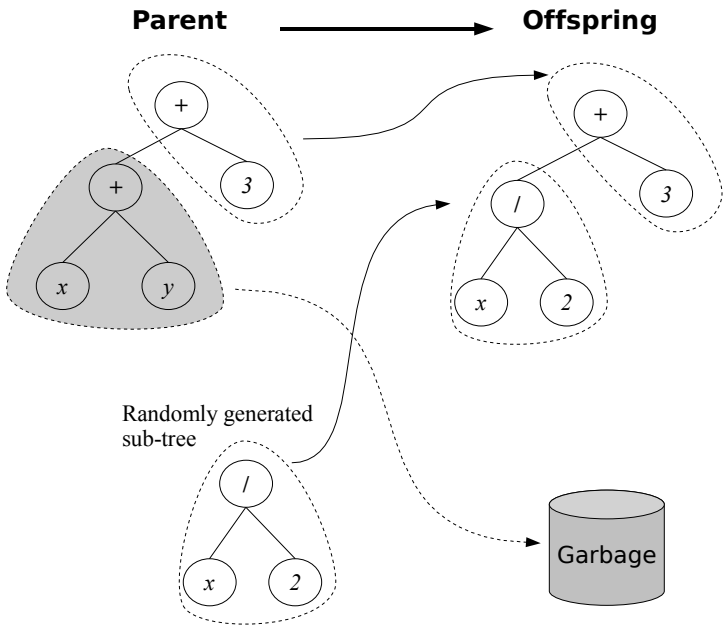


Figure 2.6: Example of subtree mutation. Adapted from (Poli et al, 2008).

- 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 2.6.

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 (Poli et al, 2008):

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

2.3.5 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 behavior 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 initializing 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. (2.1).

$$T = \{x, y, \Re\} \ . \tag{2.1}$$

2.3.6 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* (Koza, 1992). The *closure* property can be divided

into *consistency* and *evaluation safety* properties.

Subtree crossover, as described in Section 2.3.4, 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 (Poli et al, 2008).

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 behavior 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 users 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 (Poli et al, 2008).

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 synthesize analogue electrical circuits, the function set might include components such as transistors, capacitors, resistors, etc.

2.3.7 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 synthesize an amplifier automatically. Then the fitness function is the mechanism which tells GP to synthesize 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 recognizing 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 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 2.7.

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

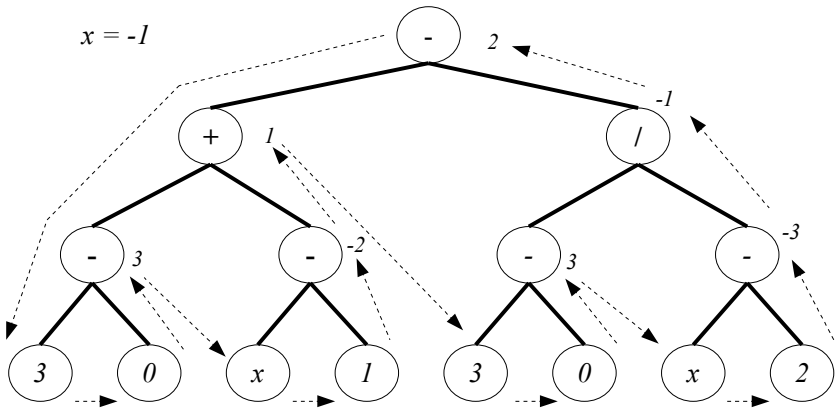


Figure 2.7: Interpretation example of a GP tree (the terminal x is a variable and has a value of -1). The number to the right of each internal node represents the result of evaluating the subtree root at that node. Adapted from (Poli et al, 2008).

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 (Poli et al, 2008).

There are also interactive GP systems were individuals evaluation is performed by a human. The main reason for this approach is the impossibility, or impracticably, 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.

2.3.8 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 recom-

mendations 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 (Poli et al, 2008).

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 larger 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 (Poli et al, 2008).

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 is 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 (Poli et al, 2008).

2.3.9 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 3

Artificial Terrains

Artificial terrain generation has been addressed by several researchers for a long time. They are used for a broad range of applications in many fields, from computer generated art and animation, architecture to virtual reality and video games. A thorough list of application examples can be found in Virtual Terrain website (Virtual Terrain Project, 2009). This chapter summarizes what have been done on this field and starts by analyzing in Section 3.1 the different data structures that exist to represent terrains, their benefits and shortcomings (Frade, 2008). Then Section 3.2 addresses the existing terrain generation techniques.

3.1 Representation

Before we can generate terrains, it is necessary to define how to represent them. The chosen data structure will influence the way the terrain is built, the available tools to manipulate it and might affect also the terrains features that can be represented. Regarding these topics there are several considerations, namely: render scale (arbitrary or limited); ability to represent all real terrain features (like caves) or only simpler ones; need to account for planetary curvature, or only a flat approximation; and finally, is it required

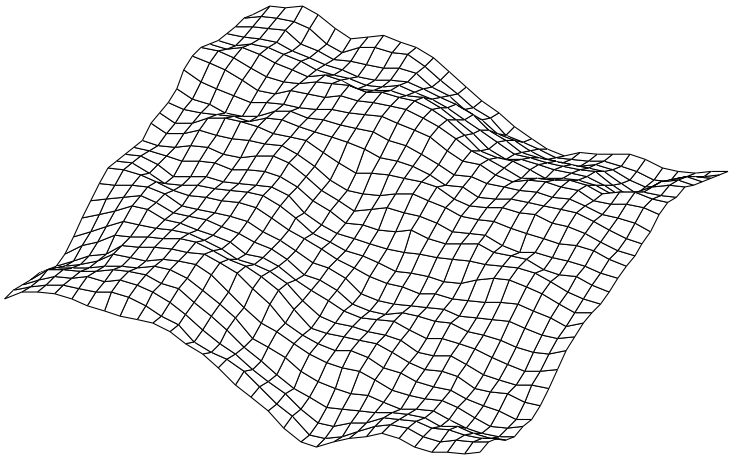


Figure 3.1: A discrete height map example

to perform collision detection in an efficient way. Having this considerations in mind, several alternative data structures to represent terrains will be analyzed:

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

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. (3.1). 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 3.1). The most common data structure to represent them are 2D arrays filled with the elevations values.

$$h = f(x, y) \quad . \tag{3.1}$$

The height maps' regular structure is their main advantage: it allows the optimization 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 (Duchaineau et al, 1997; Li et al, 2003; Losasso and Hoppe, 2004), 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 normalized it becomes the same thing as gray scale image. This means that image processing and computer vision techniques may be used to construct, modify and analyze 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.

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 overcome 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.

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

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) (Pajarola et al, 2002). 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 non-overlapping triangles. TINs are often derived from the elevation data of a rasterized digital elevation model (DEM).

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. 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. 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 the best of our knowledge, there are no methods (other than manual sculpting) to directly generate a non-uniform mesh.

Analytical and Fractal Functions - Another way to represent terrains is through the use of fractal functions or analytic expressions (procedural content). This approach is not used often, being MojoWorld [1] and GTP, our technique (see Chapter 5) two examples.

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 amenable for 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. One problem with this approach 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,

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

but with performance costs. But the main challenge of analytical functions is to model them. 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.

3.2 Generation Techniques

Terrain generation techniques can be divided in three main categories: measuring, modeling, and procedural. Although our interest is on procedural techniques, we briefly review the two other categories, because some techniques present characteristics from more than one category.

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 [2]. 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 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.

Modeling - Modeling is by far the most flexible technique for terrain generation and all kins of handcrafted content. A human artist models or sculpts the terrain morphology manually using a 3D modeling program (e.g. Maya [3], 3D Studio [3], or Blender [4]), or a specialized terrain editor program

[2] http://rockyweb.cr.usgs.gov/nmpstds/demstds.html
[3] http://www.autodesk.com/fo-products
[4] http://www.blender.org

(e.g. the editors that ship with video games like Unreal Tournament 2004 [5], SimCity 4 [6] or SimEarth [7]). 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.

Procedural - The desire for providing the player with novel and engaging content without a large investment on designers resources drives the goal of automatic content generation. Terrains are one of the many assets whose generation can be automated. Fractals are the most common procedural method to generate artificial terrains. They offer unlimited extent landscapes and can cover an arbitrarily large area without seams or unwanted pattern repetition. Some provide also mathematical advantages that make them friendly for rendering and allow ray intersections to be calculated in a straightforward way (Ebert et al, 2003). Self-similarity is the key concept behind any fractal technique. This means that when an object is magnified, subsets of the object look like (or identical to) the whole and to each other (Peitgen et al, 2004). This allows the use of fractals to generate surfaces, regardless of the scale in which it is displayed. However, real terrains present this characteristic only on a limited scale (Goodchild, 1980). These algorithms are the favorite ones by game designers, mainly due to their speed and simplicity of implementation.

Mandelbrot (1983) was the first to realize the similarity between the trace of one dimensional fractional Brownian motion and the contours of mountains peaks. This view was later generalized to fractal Brownian motion (fBm) surfaces with a power spectrum of $f^{-\beta}$. Over the years other fractal algorithms were invented that approximate this power spectrum and nowa-

[5] http://www.mobygames.com/game/unreal-tournament-2004
[6] http://simcity.ea.com/about/simcity4/overview.php
[7] http://www.mobygames.com/game/simearth-the-living-planet

days there are five different approaches: Poisson faulting (Mandelbrot, 1983; Voss, 1987); Fourier filtering (Mandelbrot, 1983; Mastin et al, 1987; Sakas, 1993; Voss, 1987); midpoint displacement (Miller, 1986); successive random additions (Voss, 1987); and finally summing band-limited noises (also known as noise synthesis) (Miller, 1986; Musgrave et al, 1989; Perlin, 1985).

The statistical behavior of fractals results in maps that present homogeneous features that are noticeable on large scales, which makes them easily recognizable. To address this issue Musgrave et al (1989) introduced a noise synthesis variant that enables some control over fractal dimension to create eroded fractal terrain, referred as multifractal. This approach is able to generate terrains with different fractal dimension on its features, such as mountains, with a rougher surface (higher fractal dimension) and smoother valleys (lower fractal dimension). To increase the realism Musgrave et al (1989) also resort to physical simulation of two erosion algorithms: hydraulic and thermal weathering. However, erosion simulation is slow and introduces more parameters for the user to control. To alleviate this problem Olsen (2004) proposed several optimizations that sacrifice physical correctness over performance with little visual impact. His approach applies erosion algorithms to a base terrain generated by fractal Brownian motion and perturbed Voronoi diagrams instead of multifractals. Olsen (2004) also provides some metrics for evaluating terrain (such as low average height and a high standard deviation for slope) to compute a game suitability score used to evaluate generated terrains.

Other fractal based terrain generation approaches have been proposed. Pabst and Jense (1995) implemented a multifractal terrain analysis algorithm that captures terrain characteristics of real-world data, into five parameters. These parameters were then put into a multifractal terrain generation algorithm that produced synthetic terrain with similar features to those in the terrain that was analyzed. Pi et al (2006) create fractal landscapes using Perlin noise to generate landscapes with focus on obtaining the desired level of detail.

All terrain synthesis based purely on fractals, control the output by means of parameters, such as the *Holder exponent*, *fractal dimension*, *octaves* and *lacunarity*, just to name a few. These parameters impact the generated terrain as a whole and do not allow the specification of features location or their dimensions. Besides, to grasp the effect of each parameter requires a deep understanding of fractal mathematics and/or trial and error experiments until the desired effect is found. This process is time consuming and there is no guarantee the desired features are discovered. To overcome the modulation issue of fractal algorithms a new set of methodologies have been devised over the years that can be categorized into: (1) synthesis by example of real world data; (2) constrained generation; (3) interactive modification of a base terrain; (4) use of software agents; and finally (5) search based algorithms. These will be successively described below.

(1) *Synthesis of terrain by example of real world data* - Techniques in this category consists on: extracting features from Digital Elevation Models (DEM); classify them; compose a new terrain with the desired characteristics; and finally smooth the transitions between the different terrain features. This concept is applied by Chiang et al (2005) where an interactive environment was created to synthesize terrains based on microscopic terrain features of real world data. Their non-fractal approach uses geometric primitives, such as triangles and trapezoids to build the terrain profile. Then, a matching procedure is applied to replace the geometric primitives by real world data. Later Tu et al (2008) proposed several improvements to this method. Another similar approach is presented by Brosz et al (2006), were they extract the small scale characteristics from one real terrain to apply them to a base terrain and increase its detail and resolution. A distinct method is described by Zhou et al (2007) where a terrain is generated based on example input height-map and a user line drawing that defines the occurrence of large-scale features, such as a mountain ridge. Then a technique, from the geomorphology field, is used to extract features from the example height-map and matched to the sketched curves and seamed together in the resulting height-map. Yet another different approach is presented by Li et al (2006). Their

proposal has four stages: terrain silhouette generation; terrain feature retrieval; region selection and filling; and texture generation. The last phase uses a machine learning algorithm to model the texture for the final height map. The main advantage of all techniques in this category is the realism of produced terrains. However, they require a suitable set of examples to be able to create all desired terrain features. Although nowadays there are many free sources of real world DEM, building the appropriate data set can be tedious and time consuming.

(2) *Constrained generation* - Control of terrain features can also be attained by imposing constraints, where the process takes into account some restrictions during or after the initial generation phase. There are several methods in this area that can be further sub-categorized into: surface approximations and deformation. Surface approximation methods are commonly used to reconstruct sparse DEM data, or to procedurally amplify DEM resolution. Vemuri et al (1997) constraint fractals to pass through a set of pre-defined points. However, there is no guarantee as to the shape of the terrain between points. With the same goal Pouderoux et al (2004) managed to obtain good approximations using radial basis functions. A constrained fractal model based on midpoint displacement algorithm is presented by Belhadj (2007). His main goal is to reconstruct DEM's where the control is provided by specifying the exact locations and height of the DEM points. Creation of ridge and river networks to be used as constraints of fractals to generate a complete heightmap is the proposal of Belhadj and Audibert (2005). The ridge and river networks are created by randomly depositing particles and allowing them to interact with each other and the terrain. As with most fractal-based approaches, their algorithm does not appear to be controllable. A different method is introduced by Szeliski and Terzopoulos (1989). They apply a surface fitting algorithm using splines, then the resulting smooth surface is perturbed by adding fractal detail to the resulting heightmap. Due to the use of a coarse spline mesh, only large scale modifications are possible. The proposal of Kamal and Uddin (2007) resembles the Poisson faulting fractals, but allows some level of control over terrain features. On each iteration

this technique draws straight lines across the base map to create a series of randomly placed polygons. Then it performs random walks that raise the points in these polygons, starting from the polygons inside the pre-specified location of the desired terrain feature. Besides the locations of the terrain feature, three parameters are used, whose impact on the resulting height map is not intuitive.

(3) *Interactive modification of a base terrain* - On the demand for easily and intuitively control of terrain features from the user perspective, several methods have emerged based on interactive modification of a base terrain. These methods have one interactive phase were the user can specify major features of maps, but rely on procedural techniques to add the small details. Schneider et al (2006) introduced a real time editor where the user edits the terrain by interactively modifying the base functions of the noise generator by replacing the Perlin noise grid with a set of user-drawn gray-scale images. This approach has the advantages to break the too homogeneous look of large scale fractal terrains. Carpentier and Bidarra (2009) created an application that allows users to paint height-maps directly in 3D view by applying procedural brushes. These are simple terrain raising brushes or brushes that generate several types of noise in real time. However, this approach shares some disadvantages of other manual editing methods, such as: requires large amount of memory to store the resulting terrain and still requires from the user time and skill to obtain the desired terrain. Smelik et al (2010) proposes another sketch based approach were users compose a digital sketch of the rough terrain layout. They declare the location of important terrain features, such as forests, mountains, cities, and villages. Once they are satisfied with the layout, the framework generates a high-resolution terrain map that complies to the specified features at large, but has, on a small scale, a high level of detail and variations in elevation. Although interactivity can be seen as the main strength of these techniques, it is also its main disadvantage because it prevents terrain generation from being fully automated.

(4) *Software agents* - A new approach, based on software agents, has been proposed by Doran and Parberry (2010). Their generator applies agents in

three phases: coast line, landform and erosion. They execute five different kinds of agents, each one with a set of parameters that describe a terrain feature, such as mountains or rivers. The authors claim their approach to be more intuitive and controllable than fractals. However, the quantity of parameters that need to be defined is huge (12 only for the mountains) and will require a certain amount of trial and error experiments until the desired result is achieved.

(5) *Search based algorithms* - The main challenge of parametric approaches is to find the right values of parameters that produce the desired terrain features. However, these approaches often degenerate into an authoring processing of trial and error, which is time consuming and offers no guarantee that such values are found. To address this problem several proposals have been made that relay on search based algorithms to find the right way to achieve the desired terrain features. For instance, Stachniak and Stuerzlinger (2005) employ a stochastic local search algorithm that finds an acceptable set of deformation operations to apply to a base terrain in order to obtain a map that approximately adheres to the specified constraints. An evolutionary approach to generate terrains was proposed by Ong et al (2005). They use an Evolutionary Design Optimization 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. The 2D terrain silhouette and a database of representative height map samples are the only form of control for their algorithm. Another evolutionary approach was proposed by

Ashlock et al (2008), co-evolving L-systems parameters and grammar to fit a specific terrain shape, which has some resemblance to symbolic regression. A different perspective is proposed by Togelius et al (2010b). They apply multi-objective EAs to evolve height maps that fit some user predicted entertainment metrics to hopefully increase players interest on the game. This concept is further developed and applied to StarCraft video game (Togelius et al, 2010a). None of these approaches addresses aesthetic appeal or creativity of the generated terrains.

The proposal presented in this thesis fits the search-based category, where the evolutionary search mechanism in use is genetic programing. GTP uses analytic expressions, designated Terrain Programs (TPs), to represent terrains. However the molding problem of analytical functions is addressed by letting the evolutionary systems find the right expression that fits our goals. In GTP_i the evolution is interactive, where a human guides TP evolution accordingly to its desired features and aesthetic appeal. Conversely, in GTP_a TPs are evolved automatically, without human intervention in the evolutionary process, to fit two morphological criteria. Once found, TPs can be easily integrated in video games and will not require any input parameter to control its look. To the best of our knowledge, this area has not been address in previous procedural content generation research.

Chapter 4

Interactive Genetic Terrain Programming

This Chapter presents the first implementation of GTP, which was interactive (Frade et al, 2008c, 2009b) and whose technique was developed during the research for the Master Thesis (Diploma de Estudios Avanzados) (Frade, 2008). The GTP_i overview is included here because it is part of the whole research line and to allow the reader to understand all the path that our research followed.

The GTP_i technique uses an Aesthetic Evolutionary Design approach and was the first attempt to address the weaknesses of existing terrain generation methods, allowing also the generation of aesthetic terrains. This technique lies in the combination of interactive evolutionary art systems with GP to evolve mathematical expressions, designated TPs, to generate artificial terrains as height maps.

Interactive 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 minimize 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 (Bentley, 1999). 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 present an overview of the most prominent works on evolutionary art systems with GP. Bentley (1999) and Takagi (2001) present a good literature survey on this topic.

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 (Sims, 1991), or create graphic movies by adding a time variable to the dynamic differential equations (Sims, 1992). 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.

Unemi (1998, 1999) developed *SBART* (Simulated Breeding ART), 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 normalized to values in $[-1, 1]$ using a saw-like function. It allows the creation of movies by replacing $(x, y, 0)$ with

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

(x, y, t), where t is a time variable. The *SBART*'s functions were expanded to create a collage (Unemi, 2000). A human user selects preferred 2D images from 20 displayed images at each generation and the system creates the 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) of Machado and Cardoso (2000), 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 color images, *NEvAr* resorts to a special kind of terminal that returns a different value depending on the color 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 (Machado et al, 2005).

4.1 Method

GTP_i relies on GP as evolutionary algorithm where the initial population is created randomly, with trees depth size limited initially to 6 and a fixed population size of 12 (see Table 4.1). 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 (Koza, 1992) 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. The use of crossover operator should be avoided, because the Evolutionary Algorithm is used as continuous novelty generators.

Table 4.1: Parameters for a GTP run (Frade, 2008)

Objective:	Generate realistic or aesthetic terrains
Function set:	Functions from Table 4.2, all operating on matrices with float numbers
Terminal set:	Terminals from Table 4.3 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, after there are no tree size or depth 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)
Operators:	Three mutation operators are used with equal probability: (1) *Replace mutation* where a random node is replaced with a new random tree generated by the grow method; (2) *Shrink mutation* where a random subtree (S) is chosen from the parent tree and replaced by a random subtree of S; (3) *Swap mutation* where two random subtrees are chosen from the parent tree and swapped, whenever possible the two subtrees do not intersect. One crossover operator is used: *subtree crossover* where random nodes are chosen from both parent trees, and the respective branches are swapped creating two offspring.
Termination:	Can be stopped at any time by the designer, the "best" individual is chosen by the designer

Consequently, non-convergence of the EA is a requirement. The extensive use of the crossover operator will make the population converge to a few solutions, thus leading to the loss of diversity of individuals and limiting the designer to explore further terrains. This is also the reason for the high rate of the mutation operator when compared with usual rates of optimizations problems with EAs.

Each GP individual is a tree composed by functions, listed in Table 4.2, and height maps as terminals, see Table 4.3. Ephemeral Random Constant (ERC) is a special terminal that creates values randomly which remain constant until it disappears from the GP tree due to the use of a genetic operators. Except for *rand* all the terminals depend upon a ERC to define some characteristics, such as the spectrum value of *fftGen*. All terminals have also some form of randomness, which means that consecutive calls of the same terminal will always generate a slightly different height map. This characteristic allows us to create different terrains, but with the same morphological features, for each time a TP is executed. All terminals generate surfaces that are proportional to the side size of the height map. This ensures that the terrain features of a TP are scale invariant. Figure 4.1 shows height maps of size 30×30 generated by terminals *fftGen*, *gauss*, *step* and *sphere*.

While in Unemi (1999, 2000) the mathematical equations are used to calculate both the pixel value and its coordinates, in GTP_i only the height will be calculated. The (x, y) coordinates will be dictated by the matrix position occupied by the height value.

4.2 GenTP Tool

To implement this new technique we developed *GenTP* (Generator of Terrain Programs) (Frade et al, 2008b), an application developed with GPLAB [2], an open source GP toolbox for Matlab [3]. *GenTP* has three functional modules

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

Table 4.2: GP Function Set (Frade, 2008)

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 Fast Fourier Transform
$smooth(h)$	circular averaging filter with $r = 5$
$gradientX(h)$ $gradientY(h)$	returns the gradient (dh/dx or dh/dy) of a height map h. Spacing between points is assumed to be 1

Table 4.3: GP Terminal Set (Frade, 2008)

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

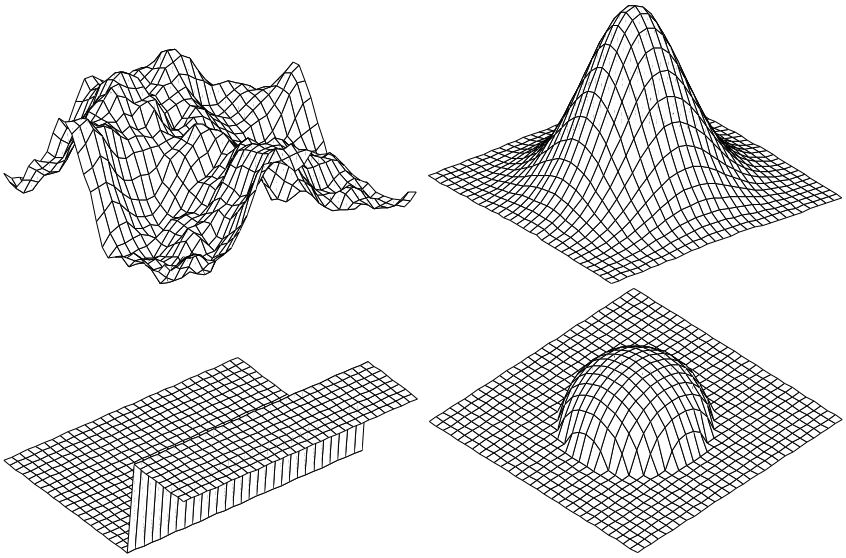


Figure 4.1: Example of height maps terminals *fftGen*, *gauss*, *step* and *sphere* (Frade, 2008)

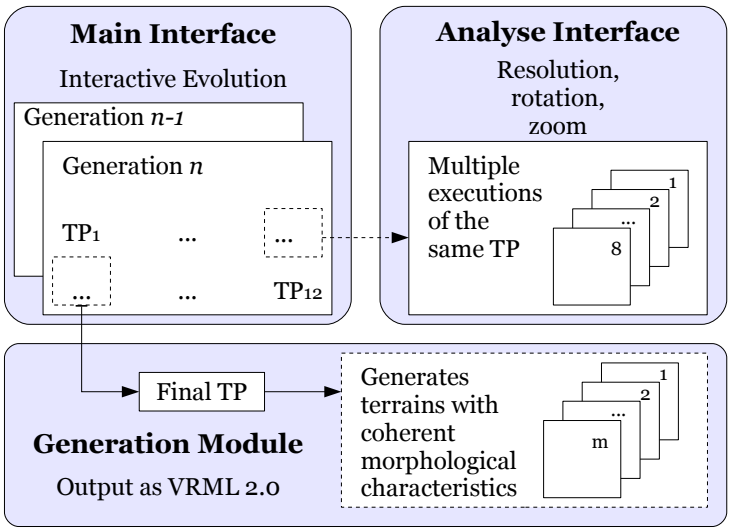


Figure 4.2: *GenTP*'s functional modules (Frade, 2008)

(depicted in Figure 4.2):

- Interactive evolution;
- Analyze;
- Generation;

The interactive evolution module is where the GP is implemented and the designer chooses the desired terrains for the next generation, for the analyze or generation modules. Figure 4.3 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 a negative impact in the tool responsiveness.

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 more diversity and

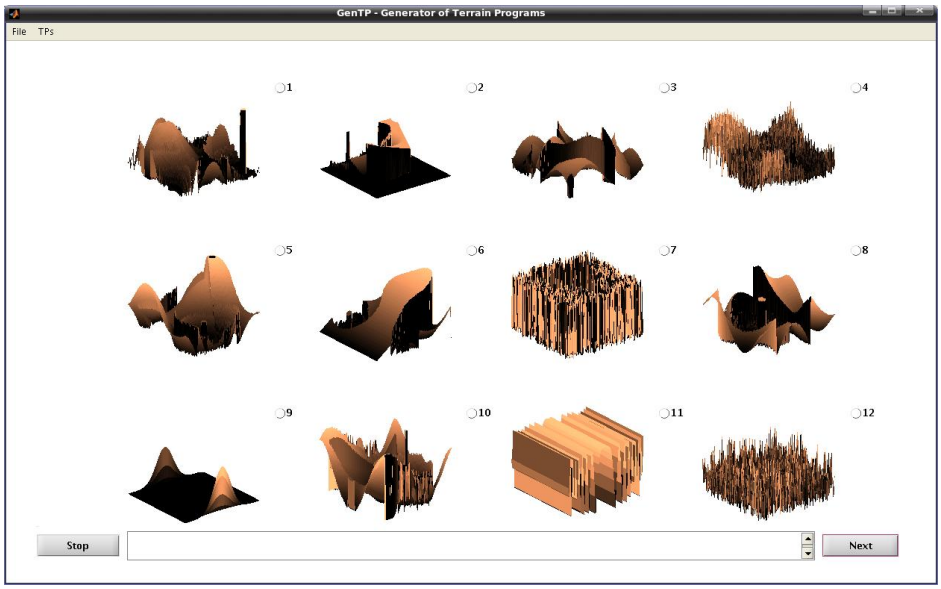


Figure 4.3: *GenTP* main user's interface (Frade, 2008)

the evolved TPs can change their look more dramatically. On the other hand, if the designer opts to select two individuals, the next generation will present few variations from the selected individuals and the TP will evolve slowly. On the bottom 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 analyze module was added to our application. This new functionality opens a new windows, see Figure 4.4, and performs 8 consecutive executions of the TP selected from the main interface. To allow a more detailed analysis of the TP characteristics this interface allows the designer to rotate, zoom and change the terrains resolution. This way the designer has more information about a

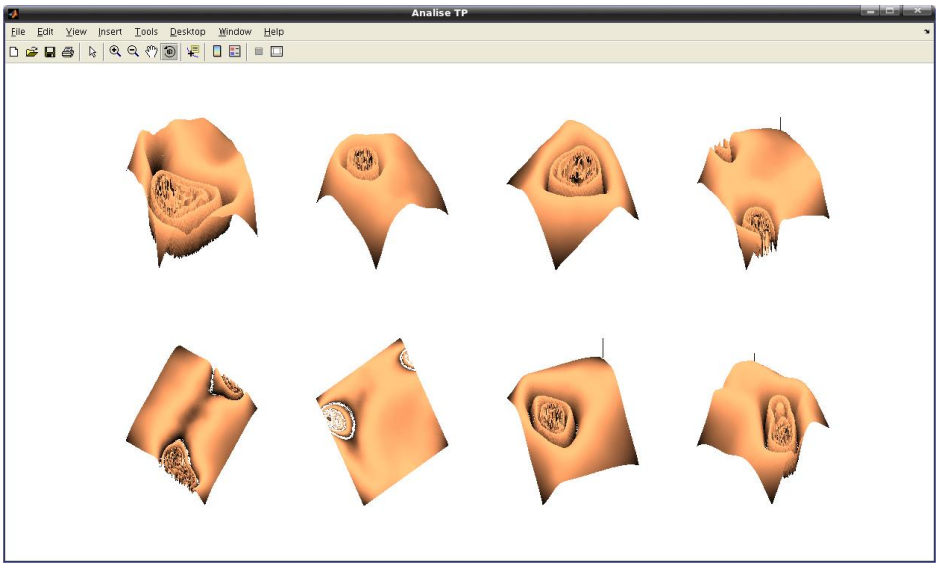


Figure 4.4: *GenTP* analyse user's interface (Frade, 2008)

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 Tests and Results

Two kind of experiments were conducted with GTP_i, the first one consisted on obtaining aesthetic appealing terrains (regardless of their realism) and the second one to achieve a realistic terrain with a specific feature in mind. 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. Must of them are difficult to

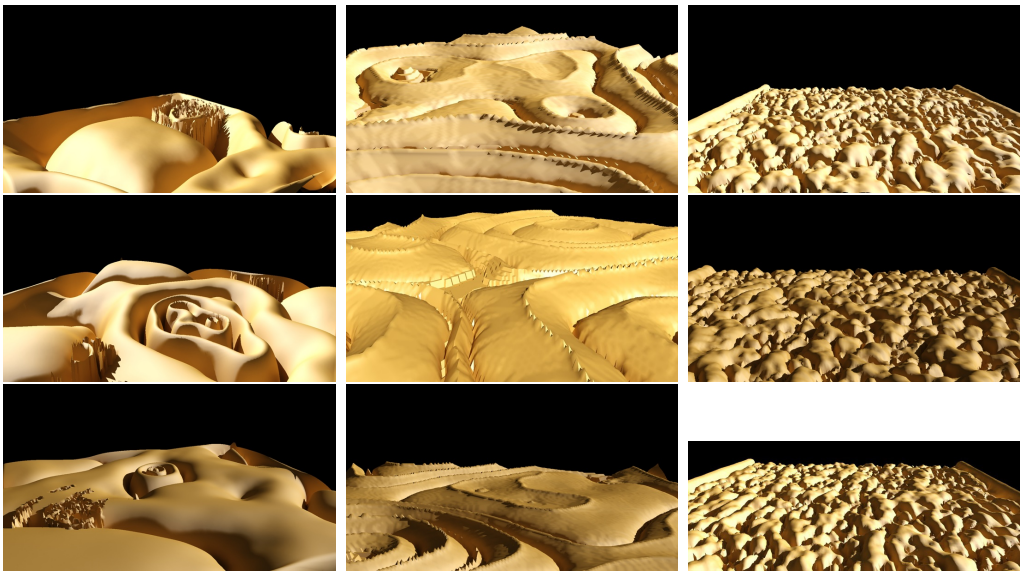


Figure 4.5: Exotic terrains generated by three different TPs (rendered with 3DS Max). The pictures of the third column were generated by Eq. (4.1) (Frade, 2008).

describe due to their exotic look (see Fig. 4.5).

$$H = \quad myLog(Incline(mySin(mySqrt(Smooth(fftGen(1.25)))))) \ . \quad (4.1)$$

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 Fig. 4.5). Fig. 4.5 has examples of terrains generated from three different TPs. Each column has pictures of terrains generated by three consecutive executions of the same TP. In this set of pictures is visible that each TP is capable of generate different terrains, but with the same features.

On the second kind of experiments we tried to obtain TPs to generate terrains with a 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, avoid-

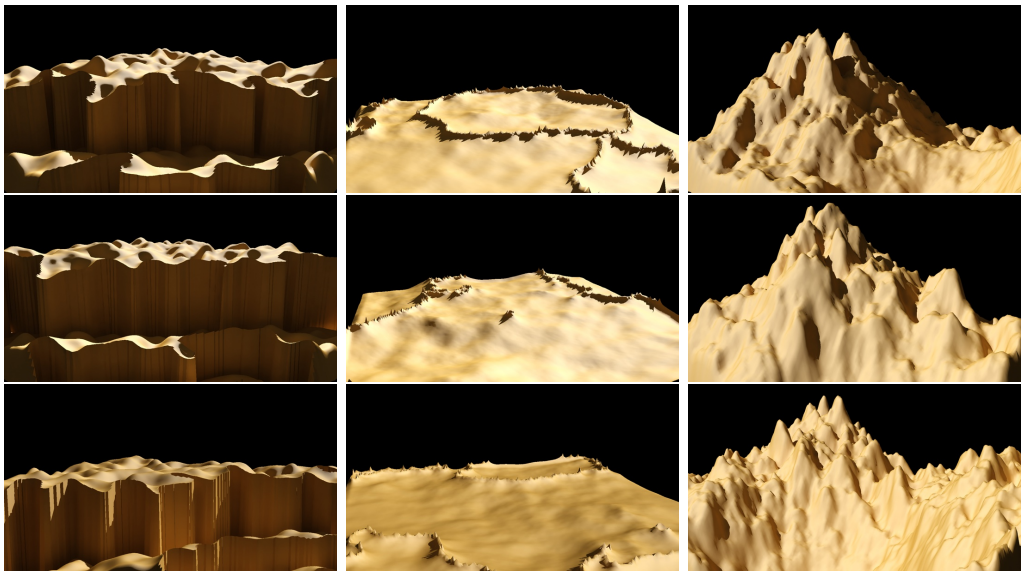


Figure 4.6: TPs evolved with specific features in mind (rendered with 3DS Max). From left to right column: cliffs, corals (Eq. (4.2)) and mountains (Frade, 2008).

ing this way a long run. We also verified that, for realistic landscapes, the range of terrains types were narrower than in the first experiment. Equation (4.2) has an example of a TP that was evolved having in mind to achieve a coral looking terrain. In the set of pictures on Fig. 4.6 it is visible that the terrains generated by each TP are always different, but still present the same features.

$$\begin{aligned} H = \quad & myLog(minus(\mathit{fftGen}(2.75), myLog(minus(\\ & Smooth(\mathit{fftGen}(1.50)), \mathit{fftGen}(2.50))))) \ . \end{aligned} \tag{4.2}$$

The evolution is influenced by the number of selected TPs, if two TPa are selected - crossover operator applied - the next generation will present few variations of the selected individual and the TP will evolve slowly. On the other hand, if the designer opts to select only one individual - only mutation operator applied -, the next generation will present more diversity and the evolved TPs can change their look more dramatically. Some robustness tests, on a few TPs, showed that the functions $myLog$, $myPower$, $myTan$ and

myAtan are the ones that have more influence in the terrain look, followed by the *Smooth*. Changes on the ERC also influence the terrain look, but that change is not always noticeable.

In spite of the good results obtained with GTP_i, the evolutionary process depends on a human to perform the evaluation and classification of each individual in the GP population, which is known to cause user fatigue (Bentley, 1999). Consequently, individuals are scored in a highly inconsistent way, which makes the GP runs hardly repeatable. This approach also requires expensive human resources for the evaluation process. GTP_i presented also another limitation: although TPs were procedural, they were unable to perform zoom over a certain terrain area. This limitation was due to the implementation of GP terminals, which depended directly from a random function instead of x, y coordinates. To address these limitations, a second version of GTP was developed. This new version, designated GTP_a, fits the SBPCG category and is fully detailed in Chapter 5.

Chapter 5

Automated Genetic Terrain Programming

GTP_a was implemented to address GTP_i limitations: user fatigue, lack of zoom and to automatically classify each GP individual. This automated version of GTP removes the human factor from the evolutionary process. It also allows designers to search terrains offline (during game development phase) and incorporate them as procedures into video games. With GTP_a the terrain generation for a given TP is deterministic, that is, it will always generate the same terrain, while in GTP_i it was stochastic (different terrains each time the generation process took place). In spite the differences of GTP_a over GTP_i the goal of generating aesthetic terrains remains. However, this time the evolutionary process is guided by a direct fitness function.

In Section 5.1 we detail GTP_i limitations. The proposed solution, its implementation and impacts on terminal and function sets are explained in Section 5.2. The reasoning for our fitness function and its formula are laid out in Section 5.3. Following, used tools are described in Section 5.4 and the explanation of the devised tests and consequent results in Section 5.5. Section 5.6 presents some sample terrains rendered in three dimensions. Finally, a preliminary analysis about the creativity of GTP_a is presented in Section 5.7.

5.1 Adding Zoom Feature to Terrain Programs

An important characteristic of some procedural techniques is their ability to generate a scene with the required *resolution* and *zoom* level, in fact this is probably the main advantage of terrain procedural techniques over the other types of techniques.

Due to computers' digital nature they cannot truly represent continuous data. So, all continuous data must be sampled to discrete values. The amount of samples per unit determines the resolution, which is perceived by the user as the quality of the digitalized function. Figure 5.1 represent a continuous function with two different samples rates and the correspondent result of that sampling. The higher the sampling rate, the better is the quality of the digitalized function. However, after a certain point there is no use to increase the amount of sampling because of the display medium limitation, or ultimately, due to the biological limitations of the human eyes. For instance, many LCD monitors can only display images up to 72 dpi (dots per inch). So, increasing the sampling rate beyond this limit will require more storage space but does not improve user's perceived quality.

If it is required to view more details than the ones allowed by the display medium limitations it is possible to resort to zoom. The zoom feature consists of narrowing the apparent angle of view of a scene, giving the sense of approximation. This feature can be achieved in procedural techniques by scaling and increasing the sampling rate. For example, the top image on Fig. 5.2 represents continuous function where the sampling rate is 2 (shown by the grid lines). To enlarge three times the area bounded by the zoom box, the sampling rate must be increased three times and the output must be scaled for the same resolution, as shown on the bottom image of Fig. 5.2. Notice the distance between each sample before and after the zoom, 0.5 units and 0.166 units respectively. However, they are represented in the output medium with the same distance between them due to the scaling process. For easier illustration the examples used a function with just one input variable.

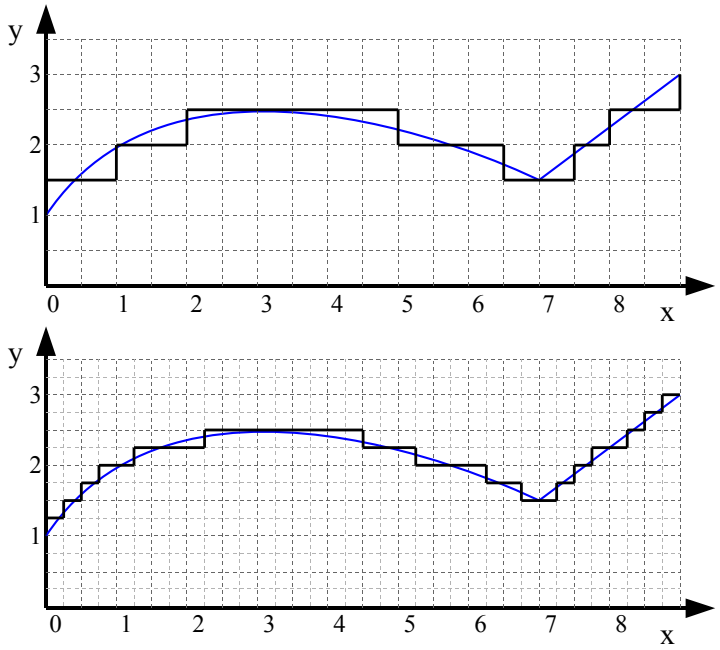


Figure 5.1: Example of a continuous function sampling, where the grid lines represent the sampling points. Both images present the same continuous function, but the sampling rate of the bottom image is twice from the one on the top.

Nonetheless, the same principles apply to any function independently of how many input variables it has.

It is thanks to the resolution and zoom properties present in procedural techniques, that computers can better simulate and represent continuous data.

In spite of the procedural nature of the TPs, the GTP_i implementation only allows to choose terrain resolution, but not zoom level (Frade et al, 2008a). This is a limitation that runs against procedural terrains advantages which we wanted to eliminate. To implement the zoom feature the continuous surface that a TP can generate must be delimited and sampled with fixed increments of x and y to obtain the corresponding altitude h, where $h = f(x, y)$, $h, x, y \in \mathbb{R}$. The altitude values are stored in matrix $H = \{h_{r,c}\}_{c \leqslant n_c}^{r \leqslant n_r}$, whose size $n_r \times n_c$ depends on the amount of samples and therefore define the height map resolution. Equation (5.1) shows the relationship between

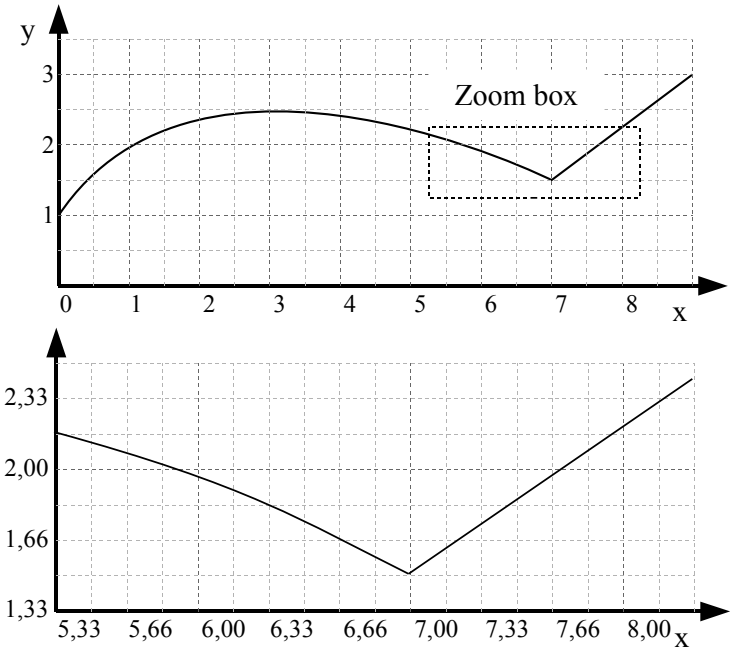


Figure 5.2: Sampling grid of a continuous functions before zoom (top) and after a 3 times zoom of the zoom box (bottom)

the height map matrix H and the TPs continuous functions. The value $h_{r,c}$ represents the elevation value for row r and column c, and D_x, D_y are the terrain dimensions. S_x, S_y allow the control of the zoom level and L_x, L_y allow us to localize the origin of the terrain view area (see Fig. 5.3). This equation alone do not solve the problem of zoom feature, there are also some requirements to be fulfilled by the terminal and function sets, which are detailed in the next section.

$$
\begin{aligned}
h_{r,c} = & f\left(\frac{c \times \frac{D_x}{n_c - 1}}{S_x} + L_x, \frac{r \times \frac{D_y}{n_r - 1}}{S_y} + L_y\right) \\
& r \in \{1, \cdots, n_r\}, c \in \{1, \cdots, n_c\}, \\
& D_x, D_y, S_x, S_y \in \mathbb{R}^+ \text{ and } L_x, L_y \in \mathbb{R}
\end{aligned}
\tag{5.1}
$$

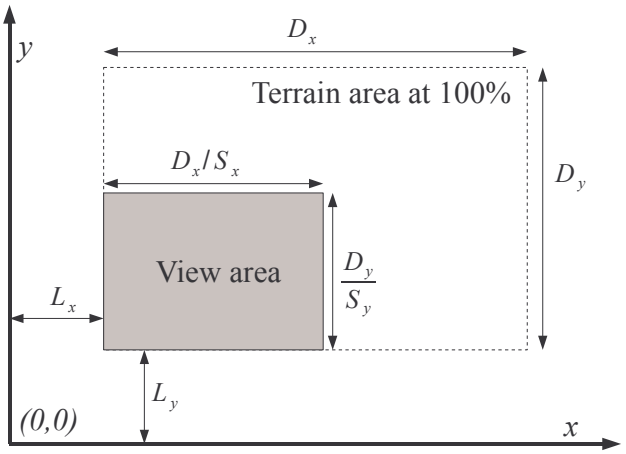


Figure 5.3: Terrain view area

5.2 Terminal and Function Sets

Terminal set used in GTP_i, shown in Table 4.3, presents functions that do not depend directly on (x, y) input variables. The result of that functions must depend only on the (x, y) coordinates, without this characteristic it is not possible to implement the zoom feature. Terminals *gauss*, *plane*, *step* and *sphere* can be easily rewritten to be directly dependent on (x, y). But the same does not hold for the *rand* and *fftGen* terminals.

As the name suggests, the *rand* terminal generates random numbers. Although we can fixate the random number seed to ensure the same values can be obtained as many times as desired, that behavior is not enough to allow the implementations of the zoom feature. This happens for two reasons, first the random number function is not continuous. Second, the output of that function depends on the number of times it is called and not on an input variable. Figure 5.4 shows the consequences of random numbers in TPs produced by GTP_i. When we increase the terrain resolution the resulting terrain is different, in spite of the similarities in their features.

The *fftGen* terminal is more complex, it is based on the Fourier transform. The theory of Fourier states that any function can be represented as a sum of sinusoidal terms. The Fourier transform takes a function from the

Figure 5.4: Two terrains generated by the same TP with different resolutions (150×150 on the left and 450×450 on the right) that show the problem of random numbers with the same seed in GTP_i implementation

spatial or time domain into the frequency domain, where it is represented by the amplitude and phase of a series of sinusoidal waves. Summing together the series of sinusoidal waves reproduces the original function - this is called the inverse Fourier transform (Bracewell, 1999). The *fftGen* terminal starts by generating random frequency components (amplitude values), then a low band filter is applied to eliminate high frequency components. Finally the inverse Fast Fourier Transform (FFT) - an efficient algorithm to compute the discrete Fourier transform - is computed to convert the frequency components into altitudes (Olsen, 2004). The outcome of this terminal is a height map whose surface roughness can be controlled by the low band filter. The lower the filter value, the smoothest the surface is. This terminal presents two problems for the zoom implementation: first it is based on a random number generator thus suffering from the same problems of the *rand* terminal. Additionally, even if we solve the random number issue, this terminal would still be an obstacle to the zoom feature due to the fact of working initially with components in the frequency domain. The inverse FFT algorithm requires a large set of points to convert them to the space domain, it does not allow the computation of a single point which is required to implement the zoom feature.

The main obstacle to implement the zoom feature are the *rand* and *fftGen* terminals. They must be replaced by terminals that depend directly from the (x, y) input coordinates, or simply eliminated. For the *rand* terminal we

believe that it should be replaced, otherwise we would lost one important characteristic of a single TP to be able to produce a family of terrains - different terrains that share the same morphological characteristics. To replace the *rand* terminal we propose the use of *noise* functions, these kind of functions have been widely used in procedural textures for several years. Noise functions are stochastic functions whose ideal properties are (Ebert et al, 2003):

- the noise function must have a repeatable pseudorandom output, based on its inputs variables;
- the output range is known, namely from -1 to 1;
- the output is band-limited, with a maximum frequency of about 1;
- the noise function should not exhibit obvious periodicities or regular patterns. Pseudorandom functions are always periodic, but the period can be made very long and therefore the periodicity is not noticeable.
- the noise function is stationary, that is, its statistical character should be translationally invariant.
- the noise function is isotropic, that is, its statistical character should be rotationally invariant.

There are several noise functions available, such as: *Voronoi*, *Cell Noise*, *Perlin*, *Blender Noise*, among others. Although these functions share the statistical behavior previously described, they produce different outputs. So, the question that arises is: which one should replace our *rand* terminal? Our preliminary analysis showed that *Voronoi* noise presents structured random patterns, for this reason we believe it is not the best candidate to replace the *rand* terminal. On the other hand, *Cell Noise*, *Perlin* and *Blender Noise* are able to produce pseudo-random outputs. Therefore, they seem to be more adequate to replace our *rand* terminal. Given that the output from these noise functions is very similar between each other, we choose the *Blender Noise* function designated $orgBlenderNoise$[1] to replace *rand*. The output range of $orgBlenderNoise$ is from 0 to 1, so to conform the properties listed

[1]Source code available under GNU General Public License at http://www.blender.org/

earlier we created $myNoise$ as listed in Eq. 5.2. Because $orgBlenderNoise$ function depends on (x, y, z), but our TPs will need only two input coordinates we fixated the z value to zero. The terminal $myNoise(x, y)$ is a lattice noise function. Lattice noise functions, commonly used as fractals primitives, use one or more set of uniformly distributed pseudo random numbers at every integer coordinate point. The intermediate values are calculated using spline interpolation. Further implementation and mathematical details are presented by Perlin (1985, 2002).

$$myNoise(x, y) = 2 \times blender\_noise(x, y, 0) - 1 \tag{5.2}$$

Regarding the *fftGen* terminal its simple elimination is not desirable because most of the interesting TPs produced by GTP_i have this terminal present at least once. The problem is to know which new terminal, or terminals, should replace it. We started our search by comparing the output of the *fftGen* terminal (see Fig. 4.1) with possible candidates. The best candidates we found to replace *fftGen* were terrain fractals (Ebert et al, 2003). Like the noise, there are also several procedural fractals:

- distorted noise
- hetero-terrain
- multifractal
- hybrid multifractal
- ridged multifractal
- fBm (fractal Brownian motion)
- turbulence
- voronoi

None of the analyzed fractals produced the same output of our *fftGen* terminal, but *fBm* was one that showed more similar results. However, their output depends not only from input coordinates, but also from other variables such as *Hurst* index (H), *lacunarity* (*lac*) and *octaves* (*oct*). Equation (5.3) shows the relation between a generic noise function and the other variables to

calculate the value of *fBm*. This opens the question if those variables should be added to the terminal set with implicit default values, or if they should evolve as well. However, most of those parameters are valid or usable only on a limited range. The *Hurst* index is valid only on the range $0 < H < 1$, *octaves* can take any real positive value, but after the value 8 its visual effect is not notable but will have a big impact on computation time. Finally, *lacunarity* works as displacement factor and like *octaves* it can take any real positive value, but Ebert et al (2003) states that best results are achieved if its value is not an integer and are around the value 2. Other fractal functions have additional parameters, like *gain* and *offset*, which are also valid or usable on limited ranges. So, if we allow those parameters to evolve the values must be normalized, which raises the question of what normalization function should be used. Due to this problem we opted to use a smaller terminal set than in GTP_i. To compensate this simplification we propose a rich function set that will enable the GP system to generate mathematical expressions like *fBm* in Eq. (5.3).

$$\begin{aligned} &fBm(x,y) = noise(x,y) + \\ &\sum_{i=0}^{\lfloor oct \rfloor - 1} noise\left((x \cdot lac)^{-2^i H}, (y \cdot lac)^{-2^i H}\right) \cdot lac^{-2^i H} + \\ &(oct - \lfloor oct \rfloor) \cdot noise\left((x \cdot lac)^{-2^{\lfloor oct \rfloor} H}, (y \cdot lac)^{-2^{\lfloor oct \rfloor} H}\right) \cdot lac^{-2^{\lfloor oct \rfloor} H} \end{aligned} \tag{5.3}$$

In GTP_i we opted by having a very rich terminal set, which allowed us to get many different terrains types with few GP generations. Obtaining interesting results with relative few generations and small populations is very important to minimize user fatigue in IEC system. However, with automated classification of individuals the evaluation is no longer dependent on a user, so very large populations can be used as well as many generation as desired. Therefore, it is possible to have a smaller terminal set as long as the function set is rich enough for the system to evolve freely to build a wide range of

Table 5.1: GP function set

Name	Description
$plus(a,b)$, $minus(a,b)$, $multiply(a,b)$	arithmetical functions
$sin(a)$, $cos(a)$, $tan(a)$, $atan(a)$	trigonometric functions
$exp(a)$	returns e^a
$myLog(a)$	returns 0 if $a = 0$ and $log(\|a\|)$ otherwise
$myPower(a,b)$	returns 1 if $b = 0$, 0 if $a = 0$ and $\|a\|^b$ otherwise
$myDivide(a,b)$	returns a if $b = 0$ and $a \div b$ otherwise
$mySqrt(a)$	returns $\sqrt{\|a\|}$
$negative(a)$	returns $-a$

terrain types.

The chosen function set for GTP_a is listed on Table 5.1 and instead of one we decided to test three different terminal sets to evaluate its influence on the resulting terrains: $T_1 = \{myNoise(x,y),\ ERC\}$, $T_2 = \{X,\ Y,\ ERC\}$ and $T_3 = \{myNoise(x,y),\ X,\ Y,\ ERC\}$. ERC stands for *ephemeral random constant* (Koza, 1992) and $ERC \in [0, 10]$. $myNoise(x,y)$ is a stochastic function that is commonly used in fractals (see Eq. 5.2), but its input parameters are implicit, so to differentiate them from the explicit parameters, X and Y, small case is used. Given the way we implemented terminal $myNoise$, it is possible to produce a terrain with only this terminal, Fig. 5.5 shows a three-dimensional render of such terrain.

We have chosen three different terminal sets because TPs generated with them will have different properties. Terrain Programs generated with terminal set T_1 will have only implicit functions. Therefore, it will be possible for a single TP to generate many view areas (see Fig. 5.3) that share the same

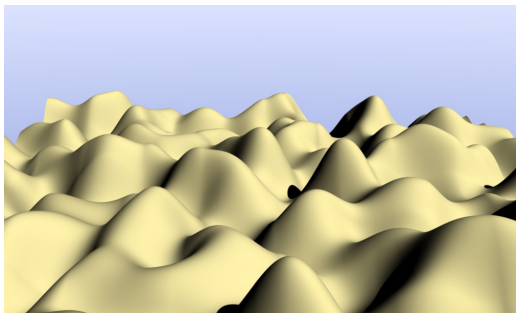


Figure 5.5: Terrain generated by $myNoise(x, y)$ with height map parameters specified in Table 5.5

morphological look - a trait presented by TPs produced with GTP_i. This property, different view areas with same morphological characteristics, can be used to simulate randomness, were L_x and L_y would work as seeds, as long as the different values for L_x and L_y are big enough to avoid overlapping of viewing areas. This feature opens the possibility to game developers to offer players with novel, but similar, terrains each time they play and that way increase game replayability value. However, the small amount of terminals in T_1 will probably confine terrain types diversity. Still, we want to test how many different terrain types our fitness function will be able to find. Two more terminal sets were created: T_2 and T_3. Terminal set T_2 presents a terminals with only the basic ingredients to build a two variables (X and Y) function, in a similar way to Koza (1992) on its symbolic regression tests. Finally, T_3 is the union of the previous terminal sets. Although TPs from both T_2 and T_3 lack the possibility of generating different view areas with the same morphological look, we want to study their behavior regarding terrain aesthetic appeal, diversity and if they are more fit for our fitness function.

5.3 Terrain Programs Evaluation

The interactive evolutionary process, implemented in GTP_i, not only depends on expensive human resources, but it is also prone to user fatigue (Bentley, 1999). To overcome these limitations a fitness function must be devised to

z_1	z_2	z_3
z_4	z_5	z_6
z_7	z_8	z_9

Figure 5.6: Neighbor positions

replace the human in the evolutionary process. To this end two morphological metrics were developed: accessibility score (Frade et al, 2010a) and obstacles edge length score (Frade et al, 2010b). The accessibility score aims to generate terrains were a certain percentage of the terrain area is accessible. A part of a terrain is accessible if its slope is under a defined threshold. Slope is an important terrain characteristic, because movement and structure placing is often restricted to low slopes. So, we create the slope map $S = \{s_{r,c}\}_{c \leqslant n_c}^{r \leqslant n_r}$ to store the declination for each cell r, c of the height map H. The slope values are calculated as the magnitude of the gradient vector (tangent vector of the surface pointing in the direction of steepest slope (Horn, 1981)). With this approach, the slope is computed at a grid point with Eq. (5.4), which depends on the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ of the height map function $z = f(x, y)$.

$$Slope(\%) = 100 \times \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{5.4}$$

The most common approximation for partial derivatives is a weighted average of the elevation differences between the given point and all points within its 3×3 neighborhood (Horn, 1981). The estimate of partial derivatives for cell z_5 (see Fig. 5.6) are given by Eq. (5.5) and Eq. (5.6), were Δx and Δy are the height map distances between each cell.

$$\frac{\partial f}{\partial x} \approx \frac{(z_3 + 2z_6 + z_9) - (z_1 + 2z_4 + z_7)}{8\Delta x} \tag{5.5}$$

$$\frac{\partial f}{\partial y} \approx \frac{(z_7 + 2z_8 + z_9) - (z_1 + 2z_2 + z_3)}{8\Delta y} \tag{5.6}$$

Once the slope map S is calculated it is necessary to determine the cells that are accessible. To that end an accessibility map $A = \{a_{r,c}\}_{c \leqslant n_c}^{r \leqslant n_r}$ is created with the same size of the height map. A is a binary map, with either 0 or 1 value on each cell depending on the slope threshold S_t as defined in Eq. (5.7).

$$a_{r,c} = \begin{cases} 1 & \text{if} \quad s_{r,c} < S_t \\ 0 & \text{if} \quad s_{r,c} \geqslant S_t \end{cases} \tag{5.7}$$

The accessible cells of a terrain (with slope below S_t) should be connected in an large area to allow player units to move around and for building placement. Therefore, we search the biggest connected accessible area in A, recurring to a component labeling algorithm. From this search we identify the amount of accessible areas and its size. The smaller accessible areas are not connected with the biggest one and are also considered inaccessible areas. Therefore, those smaller accessible areas are removed from the accessibility map. Then the terrain is evaluated by Eq. (5.8), where A_+ is the amount of cells that belong to the main accessible area.

The accessibility criteria alone would make a completely flat terrain the best fit. However, such terrain does not add realism or interest to the terrain and does not provide obstacles to units movement, which is undesirable. To prevent this, the accessibility score υ_s, is defined in Eq. (5.9). The biggest accessible area is limited by the threshold υ_t, where $p_a \in [0, 1]$ is the percentage of desired accessible area. The *ceil* function is used to ensure that the amount of desired cells for the main accessible area is round up to the nearest integer value. This way it will be possible for υ_s to achieve the exact value of zero and stop the evolutionary process. Otherwise we would have to stipulate a tolerance value within which υ_s would be considered close enough to zero and stop the evolutionary process. However, the tolerance value would be dependent from the chosen resolution for the height map,

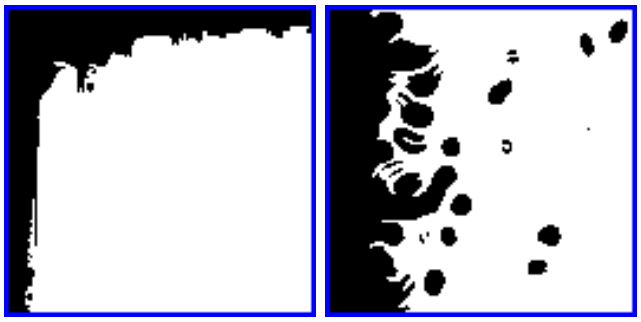


Figure 5.7: Example of two accessibility maps using only accessibility score with terminal set T_2 (left) and T_3 (right). The black areas represent terrain obstacles.

which is undesirable.

$$\upsilon = \frac{n_r n_c}{A_+}, \quad \text{where} \quad A_+ = \sum_{r=1}^{n_r} \sum_{c=1}^{n_c} a_{r,c}, \quad A_+ \neq 0 \tag{5.8}$$

$$\upsilon_s = |\upsilon - \upsilon_t|, \quad \text{where} \quad \upsilon_t = \frac{n_r n_c}{\lceil p_a n_r n_c \rceil}, \quad p_a \neq 0 \tag{5.9}$$

However, this metric alone tends to produce terrains with a single or very few obstacles with a simple edge on the accessibility map, see Fig. 5.7 (Frade et al, 2010a). This problem is specially obvious with terminals T_2 and T_3. These terrains are less suitable and interesting for video game usage.

To address the problem of simple edges we decided to measure the edge length of the obstacles on the accessibility maps and use it also to calculate individuals' fitness. With this metric we wanted to increase the complexity of obstacle edges (Frade et al, 2010b). To measure edge length, the edge map $E = \{e_{r,c}\}_{c \leqslant n_c}^{r \leqslant n_r}$ is created from the accessibility map A. For images without noise, as is the case of accessibility maps, the edge line can be determined through the Laplacian operator (Gonzalez and Woods, 2002). The Laplacian of $f(x, y)$ is a second order derivative defined by Eq. (5.10).

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{5.10}$$

The numerical estimation of the Laplacian for cell z_5 (see Fig. 5.6) is

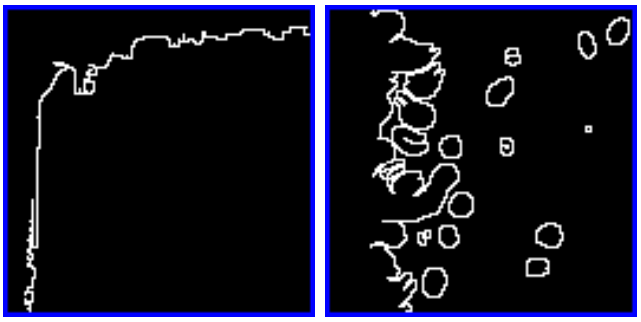


Figure 5.8: Edge maps built from the accessibility maps on Fig. 5.7

given by Eq. (5.11) and whenever it returns a positive value means that z_5 belongs to the edge line. E is a binary map, with either 0 or 1 value on each cell $e_{r,c}$ according to Eq. (5.12). Fig. 5.8 shows two examples of edge maps.

$$\nabla^2 f \approx 8z_5 - (z_1 + z_2 + z_3 + z_4 + z_6 + z_7 + z_8 + z_9) \tag{5.11}$$

$$e_{r,c} = \begin{cases} 1 & \text{if} \quad \nabla^2 f > 0 \\ 0 & \text{if} \quad \nabla^2 f \leqslant 0 \end{cases} \tag{5.12}$$

Based on the amount of cells that belong to the edge, we classify the terrain by the edge value ε defined in Eq. (5.13), where E_+ is the sum of all cells with $e_{r,c} = 1$. The formula in (5.13) was built to be minimized, so the smaller value of ε the bigger the edge length is.

$$\varepsilon = \frac{n_r n_c}{E_+}, \quad \text{where} \quad E_+ = \sum_{r=1}^{n_r} \sum_{c=1}^{n_c} e_{r,c}, \quad E_+ \neq 0 \tag{5.13}$$

$$\varepsilon_s = |\varepsilon - \varepsilon_t|, \quad \text{where} \quad \varepsilon_t = \frac{n_r n_c}{\lceil p_e n_r n_c \rceil}, \quad p_e \neq 0 \tag{5.14}$$

Without any threshold the edge value ε used as fitness function, would produce terrains without large accessible areas. To prevent this, we defined the edge score ε_s in Eq. (5.14). This way the edge length is limited by the threshold ε_t, where $p_e \in [0, 1]$ is the desired percentage of edge length in relation to the total terrain area. The *ceil* function is used for the same

purposes as in the accessibility metric showed in Eq. (5.9). We have built the fitness function as a weighted sum of these two metrics, see Equation (5.15), to analyze the impact of each metric in terrains aesthetic and influence the GP search performance. Algorithm 5.1 summarizes all the steps required to evaluate a given TP and reach its fitness value.

$$fitness = w_a \upsilon_s + w_e \varepsilon_s \tag{5.15}$$

Algorithm 5.1 GTP_a evaluation steps of TPs

Require: TP, w_a, w_e and height map parameters
1: Form TP generate the height map H
2: From H calculate the slope map S
3: From S calculate the accessibility map A
4: Determine the largest accessible area of A
5: Eliminate the smaller accessible areas of A
6: Calculate the accessibility score υ_s
7: From A calculate the edge map E
8: Calculate the edge length score ε_s
9: **return** fitness $w_a \upsilon_s + w_e \varepsilon_s$

The fitness function was built with a similar reasoning as the one used by Olsen (2004). However, instead of obstacles edge length he uses slope map standard deviation. We have also made some tests with this metric to find out that it was not adequate to our purposes. Olsen (2004) uses a base terrain and then applies erosion algorithms to help the appearance of both flat areas and obstacles. Due to its nature, these transformations are limited by the base terrain. On the other hand, GTP creates terrains from scratch without constraints regarding their initial form. Therefore, the GP system was able to easily generate them with the desired standard deviation values by producing stair forms. This was undesired, because it was limiting the appearance of more diverse terrain types.

5.4 Used Tools

One of the advantages of automated evaluation is the ability to use very large populations, when compared with IEC systems. On the other hand very large populations will also require a lot of computation power. Therefore, for GTP_a we choose an evolutionary tool that could provide us the best performance possible. Given that for automated evolution a graphical user interface is not required we choose *Lil-gp* [2] as our evolutionary frame work. *Lil-gp* is a C language system for developing genetic programming applications based on the LISP work of John Koza at Stanford University (Koza, 1992). *Lil-gp* evolves trees whose nodes are C function pointers, so tree evaluation is done entirely with complied code, which allows speed increase and to handle much large problems with bigger populations and more generations. To help us run all the envisioned tests, which are detailed in Section 5.5, we created a template file with all the required input parameters. Then we used scripts to: generate all input files (one for each test); to run *Lil-gp* with our code and input files; and finally to extract and process the results.

5.5 Tests and Results

As detailed in previous section, TPs evaluation depends on several parameters: slope threshold (from now on represented by s), percentage of accessibility area p_a, percentage of the edge length p_e and weights w_a and w_e. All parameters and terminal sets will impact both GP performance and resulting terrains. Therefore, to understand the behavior of GTP_a with weighted sum of accessibility and edge length scores, we devised a series of tests (Frade et al, 2012b).

We grouped in Table 5.2 a set of parameters, which we designate as *Test Parameters*, whose influence we want to study. T_i where $i = 1, .., 3$ represent terminal sets whose propose was detailed on Section 5.2. Slope is another

[2] http://www.genetic-programming.com/c2003lilgpwebpagedarren.html

Table 5.2: Test parameters and their values

Par.	Value	Par.	Value
T_1	$\{ERC, myNoise\}$	s_1	18%
T_2	$\{ERC, X, Y\}$	s_2	27%
T_3	$\{ERC, X, Y, myNoise\}$	s_3	36%
pa_1	70%	pe_1	20%
pa_2	80%	pe_2	25%
pa_3	90%	pe_3	30%
w_a	$0.0, 0.1, ..., 1.0$	$seed$	$1, 2, ..., 20$

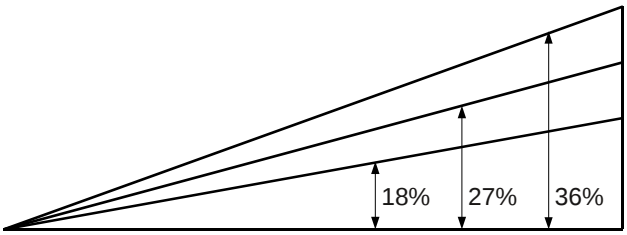


Figure 5.9: Visualization of the different slope values chosen for our tests.

important parameter, it will affect the construction of the accessibility map A, see Eq. (5.7), and that way will also influence the fitness value of a given TP. Three different slopes s_j, $j = 1, .., 3$, were tested, whose values are in Table 5.2 and also represented in Fig 5.9. These slope values were chosen because they are big enough to affect the movement of common motorized vehicles and to see how flexible our system is to generate terrains with different slopes and its impact on GTP_a performance. We also want to verify if this test parameter can indirectly influence terrain smoothness.

$$pe(\%) = 100 \times \frac{E_+}{n_r n_c} \tag{5.16}$$

The percentage of accessible terrain and edge length are controlled by p_a and p_e respectively. We performed tests with three different values for both parameters. We analyzed the accessibility maps produced with only

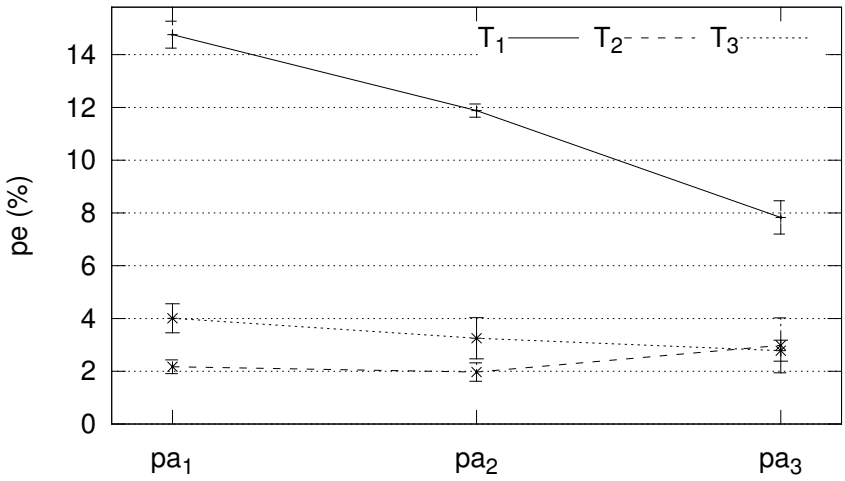


Figure 5.10: Mean percentage of p_e values calculated from the results obtained with Accessibility Score function (Frade et al, 2010a). Error bars show the standard error of the mean for 20 runs.

the accessibility constraint (see Fig. 5.7) and measured the edge length from all maps, and calculated the correspondent p_e values with Eq. (5.16), where $n_r, n_c = 128$ (Frade et al, 2010a). Figure 5.10 shows the results from that analysis, where it is noticeable that terminals T_2 and T_3 produce terrains with significantly smaller p_e values than T_1. This observation was reinforced by a Mann-Whitney U-test of each parameter combination for T_i, $i > 1$ with respect to the corresponding parameter combination in T_1. All tests returned *p-values* lower to 0.05 (see Table 5.3), which means that T_1 edge values are different from the ones obtained with T_2 and T_3 with statistical significance. In face of these values and considering that the maximum p_e obtained was 22.25% we decided to perform tests with the following values: $pe_1 = 20\%$, $pe_2 = 25\%$ and $pe_3 = 30\%$.

Finally, for these series of tests we established a linear relation between w_a and w_e as shown in Eq. (5.17). Due to this relation, from now on, we

Table 5.3: Mann-Whitney U-test for edge values calculated when only accessibility was in use.

Test parameters		T_1		
		pa_1	pa_2	pa_3
T_2	pa_1	$6.302e^{-08}$	$6.302e^{-08}$	$6.302e^{-08}$
	pa_2	$6.302e^{-08}$	$6.302e^{-08}$	$4.871e^{-07}$
	pa_3	$1.122e^{-06}$	$1.122e^{-06}$	$1.122e^{-06}$
T_3	pa_1	$6.302e^{-08}$	$6.302e^{-08}$	$3.929e^{-05}$
	pa_2	$7.415e^{-07}$	$1.122e^{-06}$	$1.175e^{-05}$
	pa_3	$6.302e^{-08}$	$6.302e^{-08}$	$7.415e^{-07}$

will refer only to w_a on results' discussion.

$$w_a + w_e = 1 \tag{5.17}$$

Our tests included all the combinations between all the test parameters T_i, s_j, pa_k, pe_l and w_m. For each combination 20 runs ($r = 1, 2, .., 20$) were performed with different seeds, which sums to 17 820 different executions. The experiments were performed on a cluster with 18 virtual machines on heterogeneous computers, 8 of them on 32 bits OS and the remaining on 64 bits OS, all running GNU Linux.

Besides the Test Parameters, there are two more sets whose values where fixed for all runs. *GP Parameters* is one of them, whose maximum and initial values, as well as operators, are defined in Table 5.4. The search stops whenever the fitness reaches the value of zero or the amount of generations reaches the value of 50, whichever comes first. Both crossover and mutation operators are the same as the ones used by Koza (1992). The crossover operator uses tournament selection to chose two individuals and swap between them two randomly selected subtrees. Our tests were performed with a tournament size of 7, however preliminary tests were made with different sizes, but they all presented similar results. The mutation operator is subtree mutation and

Table 5.4: GP Parameters

GP	Value
maximum generations	50
population size	500
initialization method	half and half
ramped	from 2 to 6
max. depth	17
selection operator	tournament, size 7
crossover operator	rate 70%
mutation operator	rate 30%

Table 5.5: Height map parameters

Height map	Value
n_r and n_c	128
L_x and L_y	0
S_x and S_y	1
D_x and D_y	10

is applied to randomly chosen individuals, where a randomly selected subtree is replaced by another randomly created subtree. The mutation rate might be considered too high for most GP applications, specially if one considers optimization problems. However, our goal is not optimization, but to use the GP system as a tool to explore many different solutions. Therefore, a high mutation rate will help to avoid equal solutions for different runs.

The other parameter set is the *Height Map Parameters*, whose values are presented in Table 5.5. They are necessary because the evaluation of the GP individuals is made after converting them to high maps. These parameters were also fixed across all the runs we made.

Sub-section 5.5.1 presents the results of the test parameters over fitness, number of generations, tree size and tree depth. Terminals and functions

frequency analysis is presented on sub-section 5.5.2 followed by a terrain overlap study on section 5.5.3. Finally, the render of some TPs are shown on section 5.6.

5.5.1 GP System

The amount of time the search phase will take is influenced by the complexity of the fitness function and test parameters values. So, in order to analyze how our GP system performed we plotted the average number of generations (Fig. 5.11), tree sizes (Fig. 5.12), tree depths (Fig. 5.13) and fitness values (Fig. 5.15).

Figure 5.11 shows the average number of generations that our system had to perform until a solution was found. The smaller the number of generations the better (less computations to find a solution). A fitness value of zero means that the TP fulfills the accessibility and edge length restrictions imposed by our fitness function in Eq. (5.15).

Figure 5.11 presents five graphics. On top is plotted the mean number of generations regarding all performed experiments (global mean m_g) for each w_a. Bellow, four additional plots are presented regarding the difference between the mean number of generations for a given test parameter $m_{<parameter>}$ and the global mean. Those graphics, with difference values, are sorted by test parameters: *terminals* ($m_{T_i} - m_g$), *slopes* ($m_{s_j} - m_g$), accessibility ($m_{pa_k} - m_g$) and edge length ($m_{pe_l} - m_g$). This approach, of one global plot followed by four plots of differences, is also applied to Fig. 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.20 and 5.40.

The first thing to stand out from Fig. 5.11 (and also on Fig. 5.12 and Fig. 5.13) is that $w_a = 0$ and $w_a = 1$ are special cases. The amount of required generations on both situations is considerably lower than for $0.1 \leqslant w_a \leqslant 0.9$. Nevertheless, the average amount of generations is slightly lower for $w_a = 1$ than for $w_a = 0$. For $0.1 \leqslant w_a \leqslant 0.9$ the number of generations present a small tendency to decrease as w_a increases. Regarding the influence of each

terminal, it is clear that on average T_2 requires more generations than T_1 and T_3. The accessibility parameter pa_3 also requires more generations than pa_1 and pa_2 before achieving a solution. On the other hand, both slope and edge length parameters present a small influence on the number of generations.

Average tree sizes and their relation to the test parameters are represented on Fig. 5.12. Again $w_a = 0$ and $w_a = 1$ are special cases, but average tree sizes for $w_a = 1$ are a bit smaller than for $w_a = 0$. For the remaining values of w_a, tree sizes present a small trend to increase with the increase of w_a. Terminal T_2 generate trees whose size is consistently higher than T_1 and T_3. T_3 presents the smaller tree sizes, but with a very small difference to T_1. Parameter pe_1 displays smaller tree sizes than the others edge length parameters, but that advantage decreases as w_a increases and vanish after $w_a = 0.7$. The test parameters for slope and accessibility have a very small influence on tree sizes.

Tree depth is limited to 17 (which is a relative low value), so we were expecting completely flat plots at the maximum allowed depth for $0.1 \leqslant w_a \leqslant 0.9$. Figure 5.13 shows the average tree depths, where it is possible to see a very small increase in depth as w_a increases (for $0.1 \leqslant w_a \leqslant 0.9$). Tree size and tree depth are related, so the effect of the test parameters over tree depths is very similar to the effect on tree sizes. T_2 generates deeper trees than T_1 and T_3. Terminal T_1 has trees with smaller depth, but T_3 follows closely. Slope and accessibility parameters have no significant influence on tree depths, except for s_1 and pa_3 at $w_a = [0.1, 0.2]$. Parameter pe_1 displays smaller tree depths, but that advantage decreases as w_a increases and vanish after $w_a = 0.7$.

Although we collected the time each GP run took, we do not present them. As stated previously, our experiments were performed on a cluster with heterogeneous computers, therefore those values would be misleading. However, since TPs can be used to generate terrains dynamically, their execution time is of most importance. Therefore, we opted to present on Fig. 5.14 the average execution time of the best TPs. For this task we measured

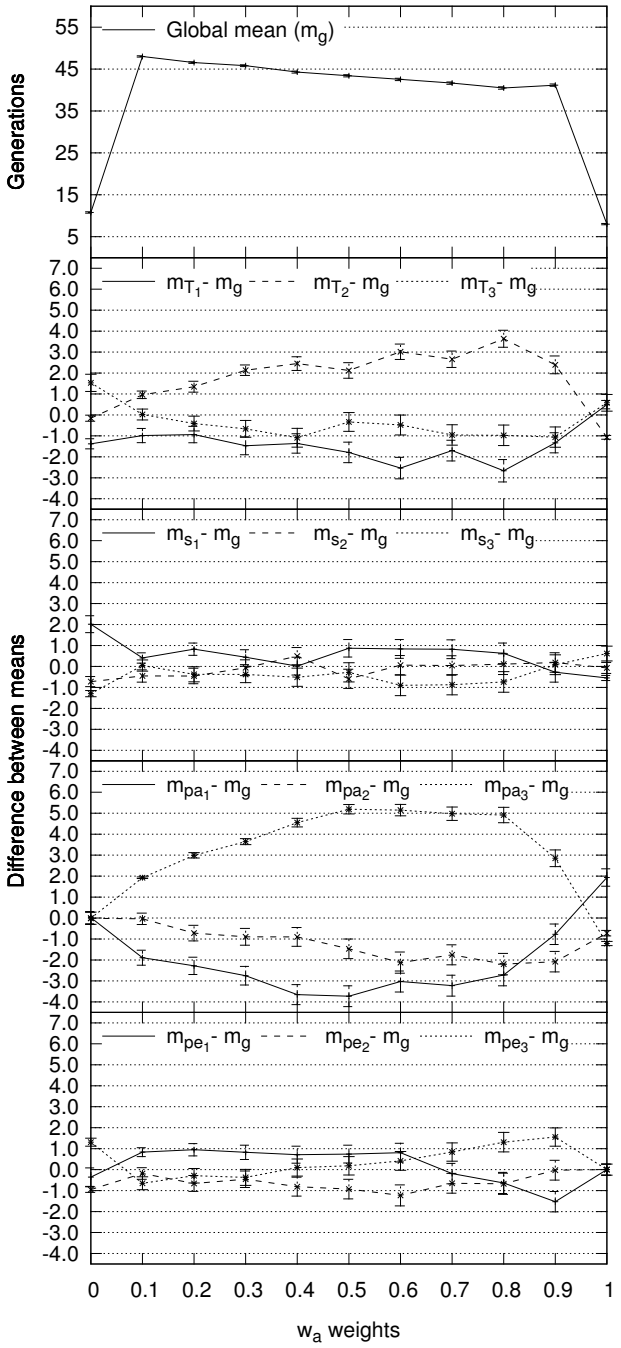


Figure 5.11: Mean number of generations versus w_a. Error bars represent the standard error of the mean.

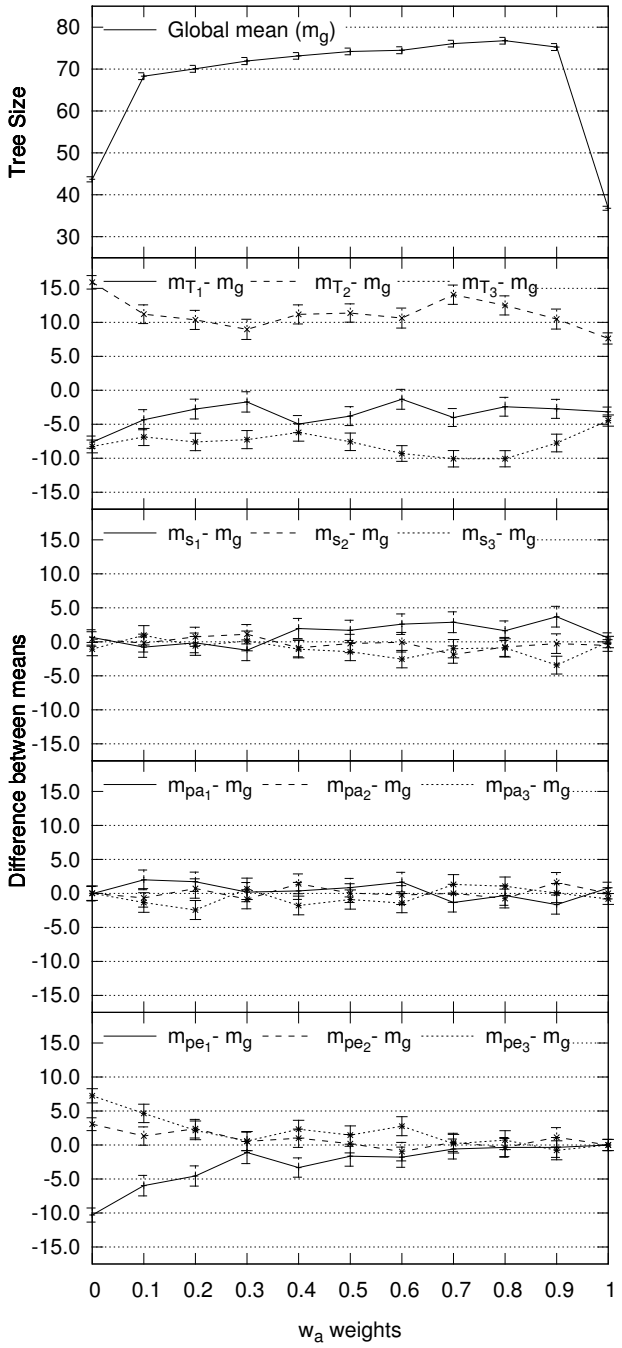


Figure 5.12: Mean of GP tree sizes versus w_a. Error bars represent the standard error of the mean.

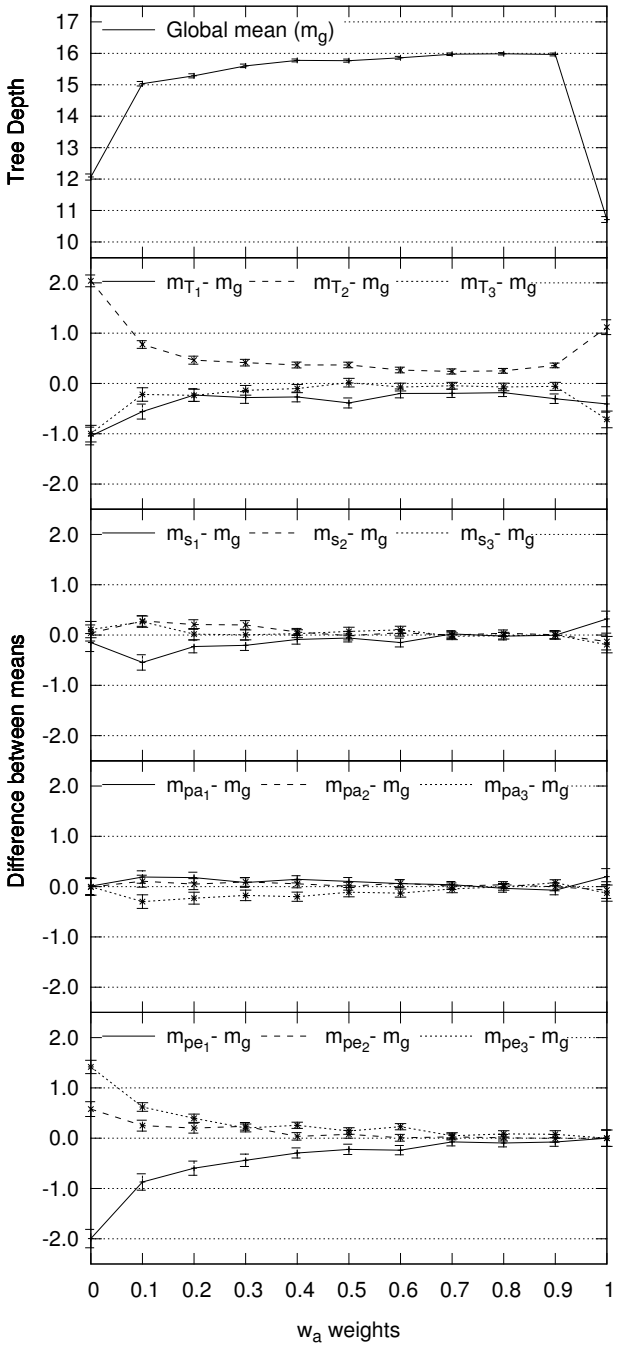


Figure 5.13: Mean of tree depths versus w_a. Error bars represent the standard error of the mean.

only the time each TP took to generate a height map of size 1024×1024 with *double* precision. The time required to render those height maps was not taken into account as it would depend on many variables, such as textures, lights, render engine and so on. Presented times were measured on a single computer with a Core 2 Duo CPU running at 2.4GHz with 1GB of RAM, running 64 bits GNU Linux natively.

The lower execution times of TPs were found for $w_a = 1.0$, followed by $w_a = 0.0$ and in the range $0.1 \leqslant w_a \leqslant 0.9$ they slightly increase as w_a increases. Execution time depends on tree sizes: the bigger the tree is, the more time it will take to execute. So we expected a global behavior similar to tree sizes shown in Fig. 5.12. Therefore, slope, *pa* and *pe* impact mimics the one found for average tree sizes. Although it presents bigger tree sizes, terminal T_2 presents execution times much lower than T_1 and T_3. This is explained by the fact that terminal $myNoise$ is very complex and time consuming function, which penalizes execution times of T_1 and T_3.

The time required to generate a map is an important aspect for video games, specially if the maps must be rendered in real time. Our results show that TPs can generate big maps with times in the same order of magnitude of the ones obtained by Belhadj (2007). However, TPs execution times can be greatly improved, since each cell value of the height map is independent of the others cells and therefore do not require any interprocess communication. On the other hand the technique presented by Belhadj (2007) the value of a cell depends on several cells. For these reasons, TPs present very good scalability and can take advantage of modern multi core CPUs or GPUs to speed up its generation.

Fitness is the most relevant value regarding GP systems. As stated previously, our fitness function was built to be minimized, therefore the closer the fitness values are to zero, the better, see Fig. 5.15. Globally, the higher w_a is (except for $w_a = 0$) the better the fitness values are (closer to zero). It is clear that as *pa* increases the fitness values get worse, which was expected. However, we did not anticipated such a huge difference between pa_3 and the

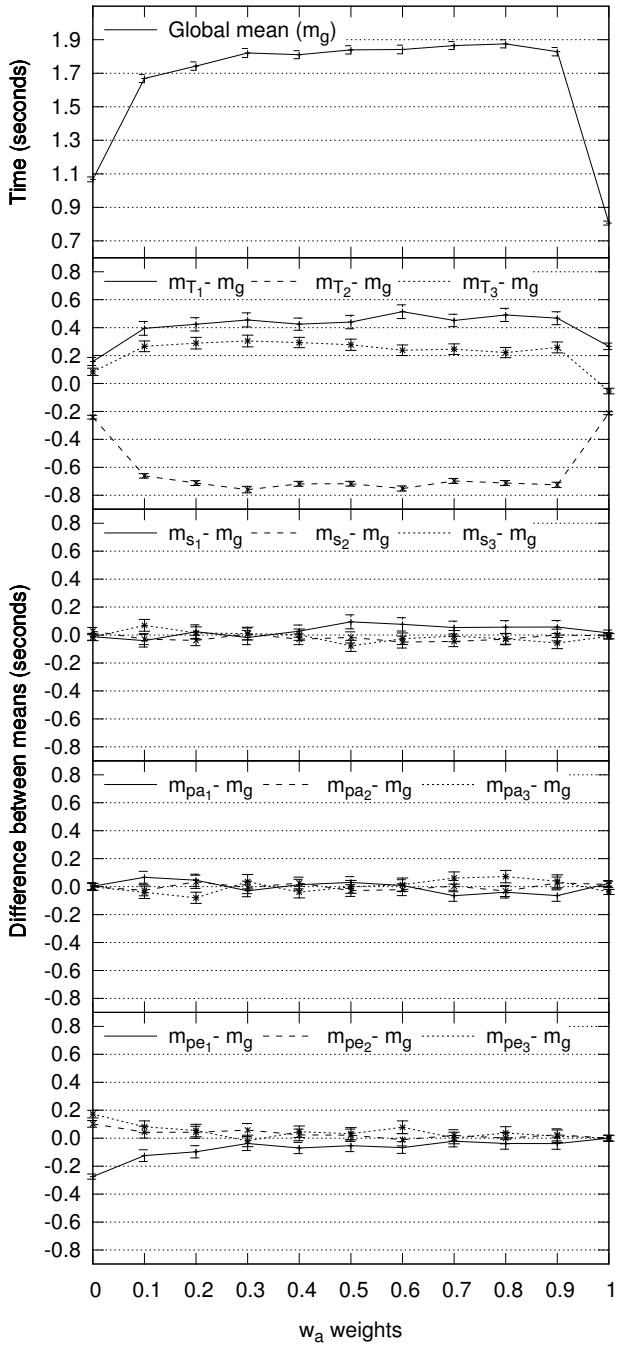


Figure 5.14: Mean of TP execution times versus w_a. Error bars represent the standard error of the mean.

others pa values. The slope impact on fitness shows that s_1 has worse performance than s_2 and s_3. We were expecting that smaller slope values would have better fitness. A more detailed analysis was conducted and we noticed that runs with the combination of s_1 with pa_3 was the main cause for the globally bad performance of s_1. Slopes s_2 and s_3 have a similar behavior.

Considering the average edge length values on Fig. 5.10 (obtained using only the accessibility score function) we expected that the higher the pe value was, the worse the fitness would be. However, pe_1 presents worse values than the others, pe_2 has the better fitness values, followed closely by pe_3. This might mean that our system does not behave linearly with the edge length parameter, further tests with lower pe values are required to better understand this parameter.

On Fig 5.15 we expected the fitness of T_3 to be similar or better than the other two terminals sets, given that T_3 is the union of T_1 and T_2. To find out why this was happening we decided to plot also the percentage of solutions (TPs) that reached fitness zero, see Fig. 5.16. Slope has a very small impact on the percentage of TPs with fitness zero and both pa and pe present a behavior consistent with the one shown for the fitness on Fig. 5.15. Concerning terminal sets, T_1 is the one that presented more TPs with fitness zero, followed by T_3 and T_2 is the terminal with lower percentage of TPs with fitness zero. Terminal set T_3 is the biggest one with 4 elements, so the search space is also bigger than the others. We believe this is the reason why T_3 had worse fitness values, but they can be improved with a higher limit of generations.

5.5.2 Occurrence Analysis

To see which functions and terminals contributed the most to achieve the best solutions, we decided to calculate how often each of them occurred (Frade et al, 2012b). Therefore, we calculated the percentage of occurrence for each terminal and function according to Eq. (5.18). The terminal or function

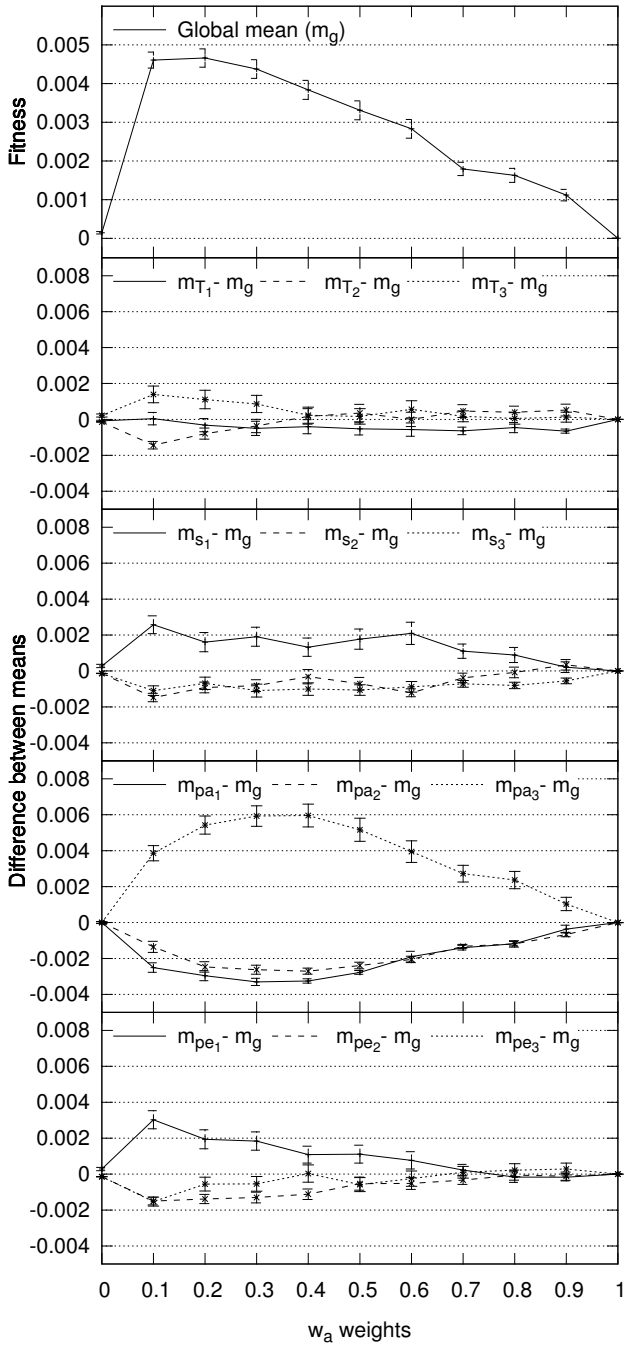


Figure 5.15: Mean fitness values versus w_a. Error bars represent the standard error of the mean.

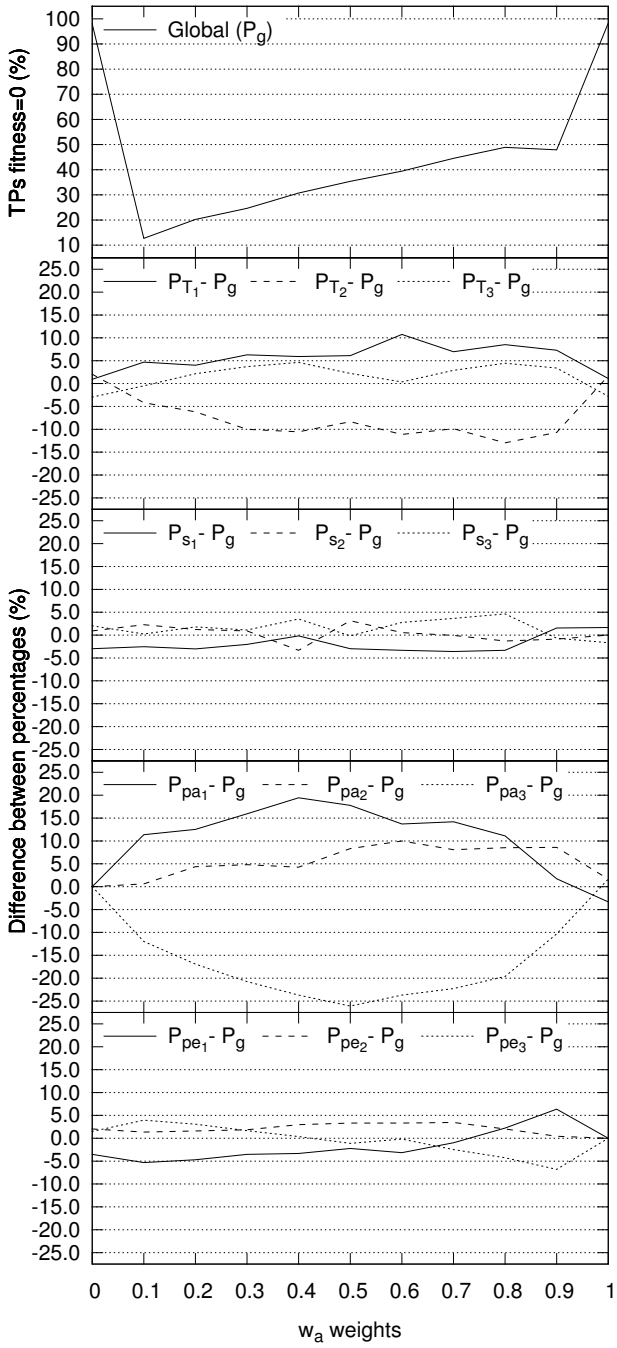


Figure 5.16: Percentage of TPs that reached fitness 0 versus w_a.

which we want to calculate is represented by fun_h, TP_r is the solution for the seed r and N is the sum of tree sizes for all seeds r and is calculated by Eq. (5.19).

$$Oc(fun_h) = \frac{1}{N} \sum_{r=1}^{r_n} count(fun_h, TP_r) \tag{5.18}$$

$$N = \sum_{h=1}^{fun_n} \sum_{r=1}^{r_n} count(fun_h, TP_r) \tag{5.19}$$

Figure 5.17 shows the occurrence of each terminal and function and their variation imposed by test parameters. As expected, terminals have a great impact. For T_1 and T_3 the terminal *myNoise* is quite predominant when compared with the remaining functions. Terminal *myNoise* seems to be main responsible for the good results of T_1 and T_3 depicted in Fig. 5.16, although X and Y seem to be better at finding solutions for the edge length score function for $0 < w_a < 0.3$. Terminal *ERC* occurrence is impacted by terminals and, with less significance by slope, accessibility and edge length parameters. The third most common function is *cos* which is affected by chosen terminal, slope or edge length parameters. It is also noticeable that *pa* has almost no influence on functions occurrence, *pe* only influences *cos* and *mySqrt* significantly. Finally, *slope* influences mainly *cos*, *multiply*, *myDivide* and *mySqrt*.

Figure 5.18 shows the influence of w_a over the average occurrence of each function. However, only $w_a = 1$ has a considerable impact on functions occurrence.

5.5.3 Overlap

As shown on Fig. 5.16, there was a relative large number of TPs to reach the perfect fitness value of zero for different test parameters combinations. We want to investigate if this happens due to the existence of several different solutions, or due to convergence of the solutions. We already know that

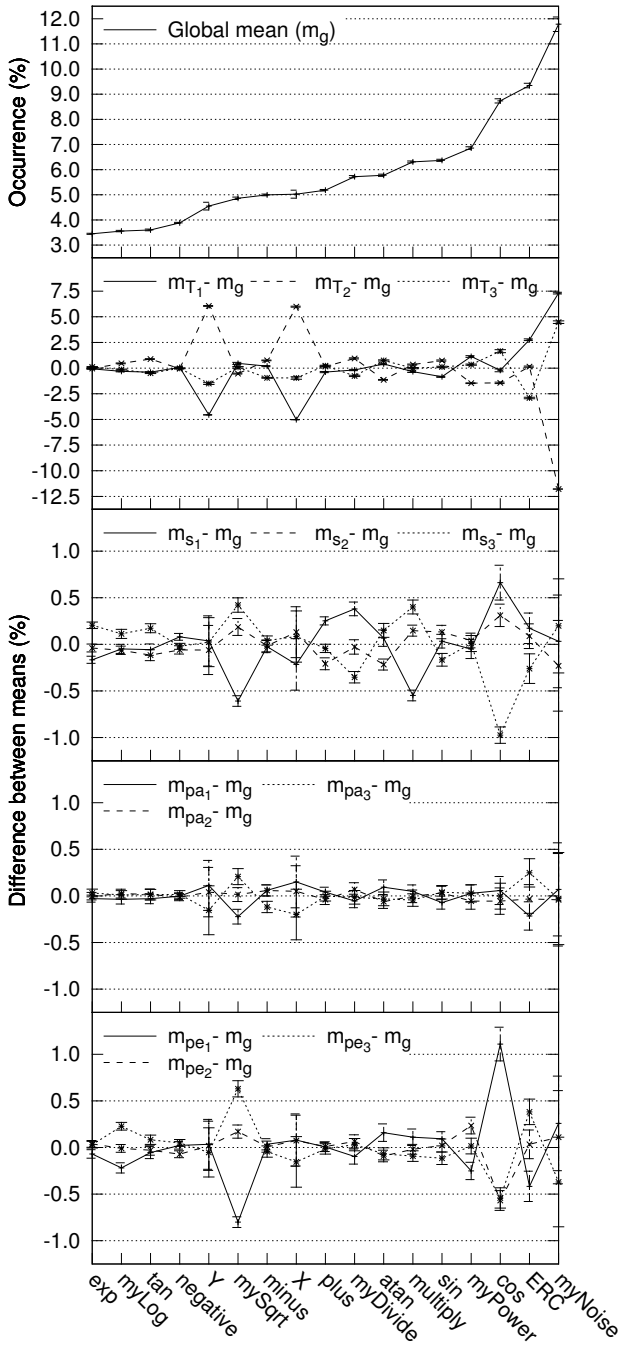


Figure 5.17: Mean occurrence of functions and terminals versus w_a. Error bars represent the standard error of the mean.

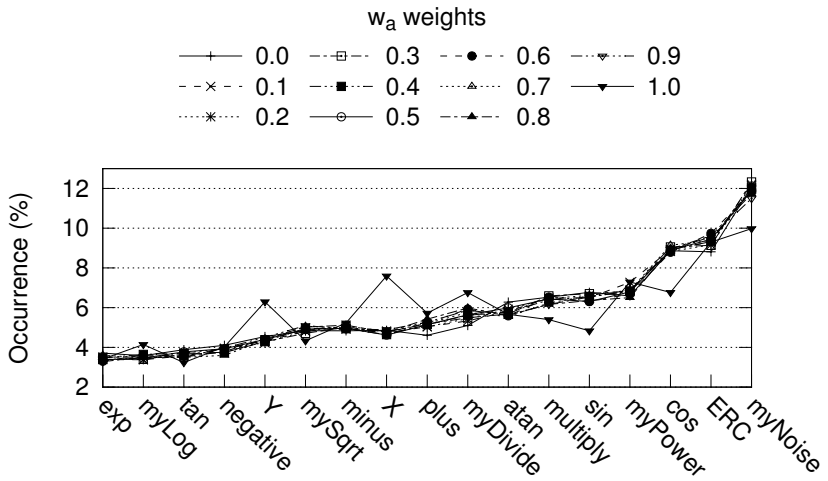


Figure 5.18: Mean occurrence of functions and terminals for a given w_a weight.

there are some repeated TPs, but these solutions have not reached a fitness value of zero, besides there might exist different TPs that are mathematically equivalent and render the same terrain. Therefore, we compared each accessibility map with the other 19 from the 20 runs of each test (changing only the seed). The comparison consists in counting how much inaccessible (black, $a_{r,c} = 0$) area overlaps between two accessibility maps, see example on Fig 5.19. To compute the overlap value $o_{p,q}$ between two maps A_p and A_q we used Eq. (5.20), were an overlap value of 100% means that maps A_p and A_q are equal. Accessibility maps are binary, as shown by Eq. (5.7), therefore Boolean operations can be performed with them. In Eq. (5.20) we negate the accessibility maps to count the inaccessible areas and to find the intersection between them. Then we defined the overlap value of each map o_p as the average of all $o_{p,q}$, as shown in (5.21).

$$o_{p,q}(\%) = 100 \times \frac{\sum_{r=1}^{n_r}\sum_{c=1}^{n_c}(\neg A_p \wedge \neg A_q)}{max\left(\sum_{r=1}^{n_r}\sum_{c=1}^{n_c}\neg A_p, \sum_{r=1}^{n_r}\sum_{c=1}^{n_c}\neg A_q\right)} \tag{5.20}$$

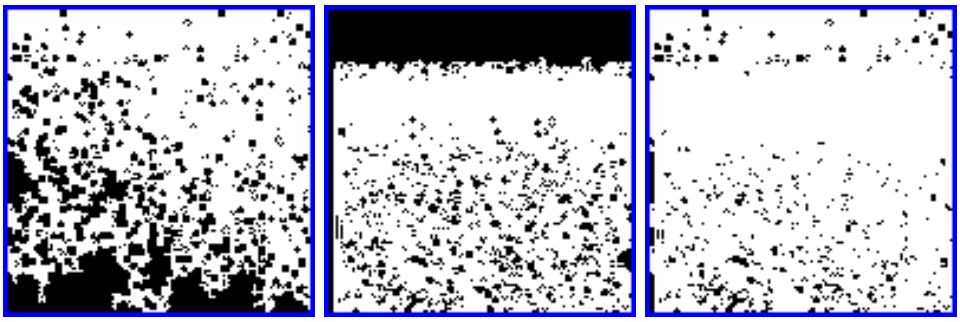


Figure 5.19: Overlap of inaccessible areas between two maps. These maps are from T_2, s_2, pa_1, pe_3, r_8 on the left (1) and r_{16} at center (2). On the right is the resulting overlap with $o_{1,2} = 21.67\%$

$$o_p = \frac{1}{r_n - 1} \sum_{q=1}^{r_n} o_{p,q}, \quad q \neq p \tag{5.21}$$

Figure 5.20 shows the average of overlap values o_p and the correspondent influence of the test parameters. Overall, the overlap value for $w_a = 0$ and $w_a = 1$ are higher than for the remaining range of w_a. This is in part explained by the amount of terrains that presented an overlap of 100%, which was 1.30% and 1.11% respectively. For $w_a = 0.1$ and $w_a = 0.2$ the amount of maps with an overlap of 100% was 0.19% and 0.06%, while for the remaining w_a values was 0.00%. Overall, the weighted combination of the accessibility score and edge length score is beneficial to reduce the average overlap values. Which is good, because we want to be able to generate as much diverse terrains as possible, as opposed to regular optimization problems where the goal is to have convergent solutions.

Terminal T_2 is the one that provides lower overlap values, followed by T_3 and then T_1. Slope has no significant impact on overlap and pe only makes difference for pe_1 and $w_a = 0$, for the remaining values it also has no influence. Finally, the accessibility parameter presents an expected behavior, the higher pa is the lower the overlap values are.

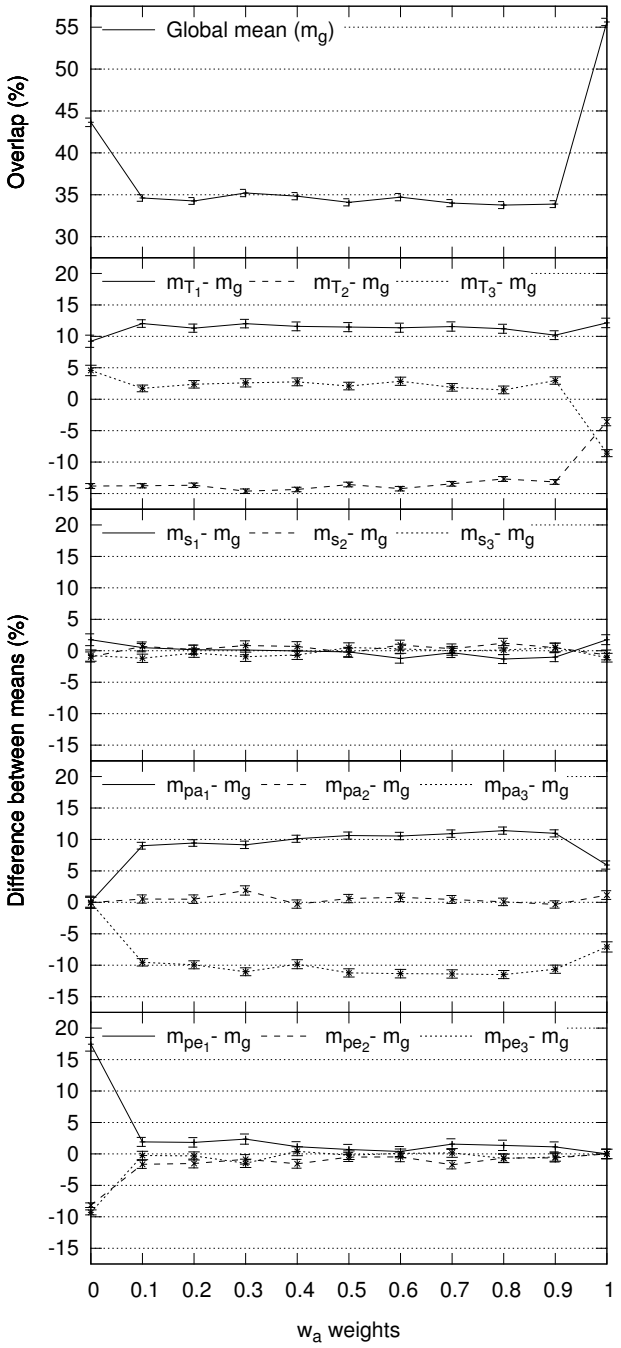


Figure 5.20: Overlap of inaccessible areas versus w_a. Error bars represent the standard error of the mean.

5.6 Sample Terrains

Given the huge amount of results, we only performed a visual inspection of 100 terrains for each terminal set. To illustrate them we present 8 different TPs for each terminal set, which are displayed in Fig 5.27 to 5.38. Both the visual inspected and presented terrains were randomly selected. TPs expressions from all presented images are listed in Appendix A.

For all depicted terrains in subsections 5.6.1 and 5.6.2, we present on top the H map displayed as gray scale image and its correspondent accessibility map A. On the bottom, we show a rendered image of a three dimension view point from the terrain. Those renders were performed on Blender 3D[3] with a single point of light and without textures to emphasize terrains surface shape. Each figure has the identification of the TP that generated it with the following syntax: *terminal*, *slope*, *pa*, *pe*, w_a and *seed*. For abbreviation proposes we replaced w_a by w_m and *seed* by r_u, where m can take values in the range $m = 0, .., 10$ and $u = 1, .., 20$.

5.6.1 Terrains with a single metric

Terrains obtained with $w_a = 0$ (only edge length metric) and $w_a = 1$ (only accessibility metric) are special cases. As our results showed in this two cases GP system performance was globally better than when both metrics were in use. Therefore, we present sample terrains from these two cases separately from the other ones.

5.6.2 Terrains with both metrics

This subsection presents some sample terrains obtained when both the accessibility and edge length metric are in use. From our visual inspection, it is clear that terminal sets have a great impact on both terrains look and

[3]Available at http://www.blender.org/

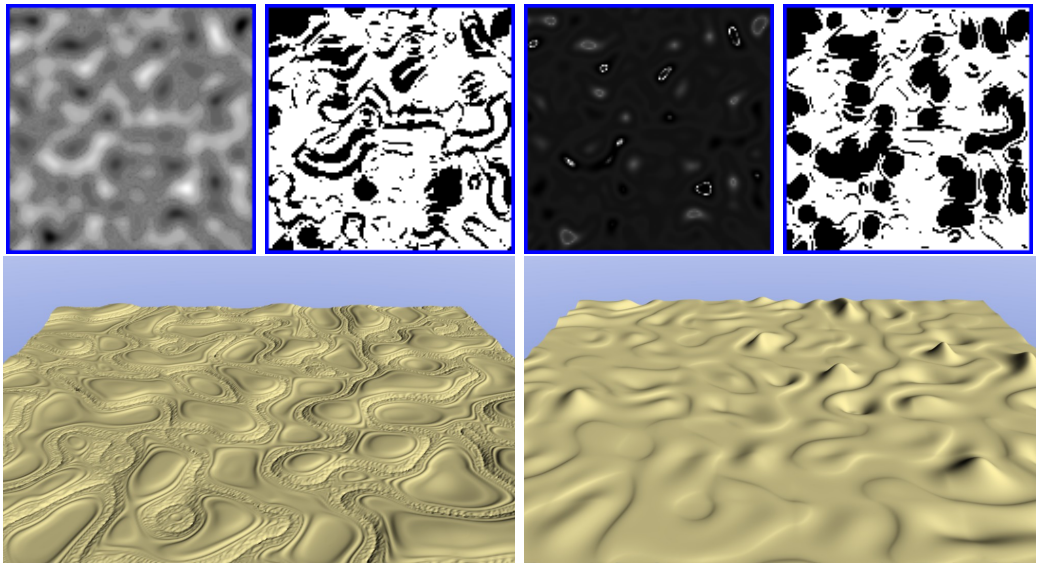


Figure 5.21: Terrains generated by TP $T_1, s_1, pa_1, pe_3, w_0, r_4$ with *fitness*= 0.000000 on the left, and $T_1, s_1, pa_3, pe_2, w_0, r_1$ with *fitness*= 0.000000 on the right

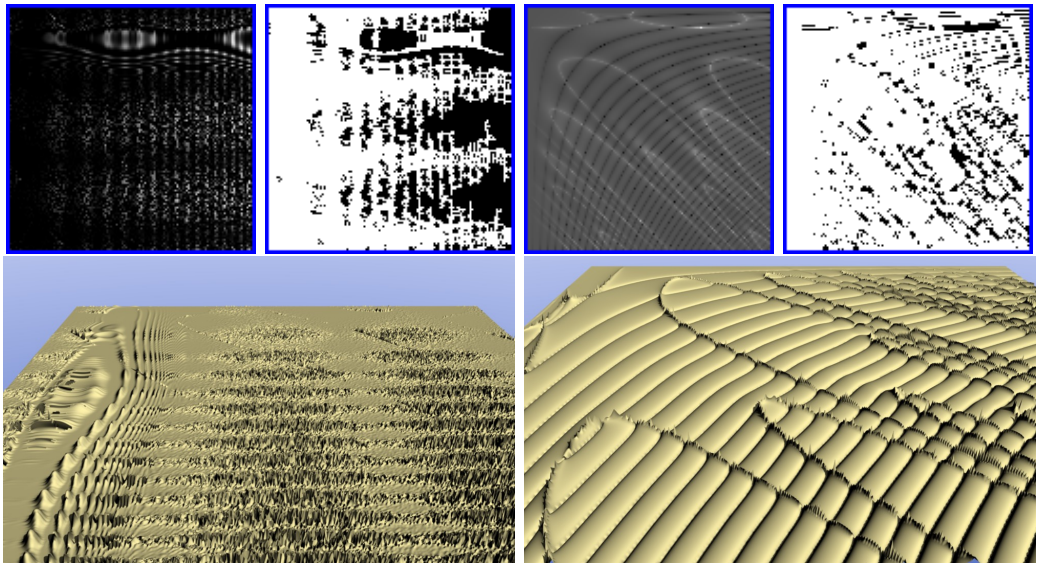


Figure 5.22: Terrains generated by TP $T_2, s_1, pa_1, pe_2, w_0, r_3$ with *fitness*= 0.000000 on the left, and $T_2, s_3, pa_2, pe_3, w_0, r_2$ with *fitness*= 0.000000 on the right

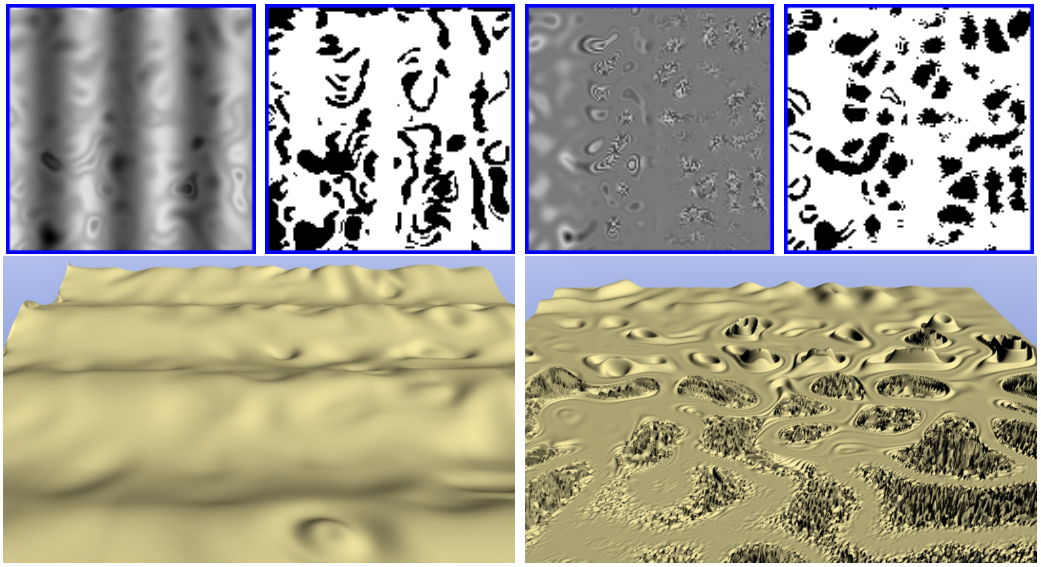


Figure 5.23: Terrains generated by TP $T_3, s_2, pa_1, pe_1, w_0, r_2$ with *fitness*= 0.001523 on the left, and $T_3, s_3, pa_2, pe_1, w_0, r_{10}$ with *fitness*= 0.000000 on the right

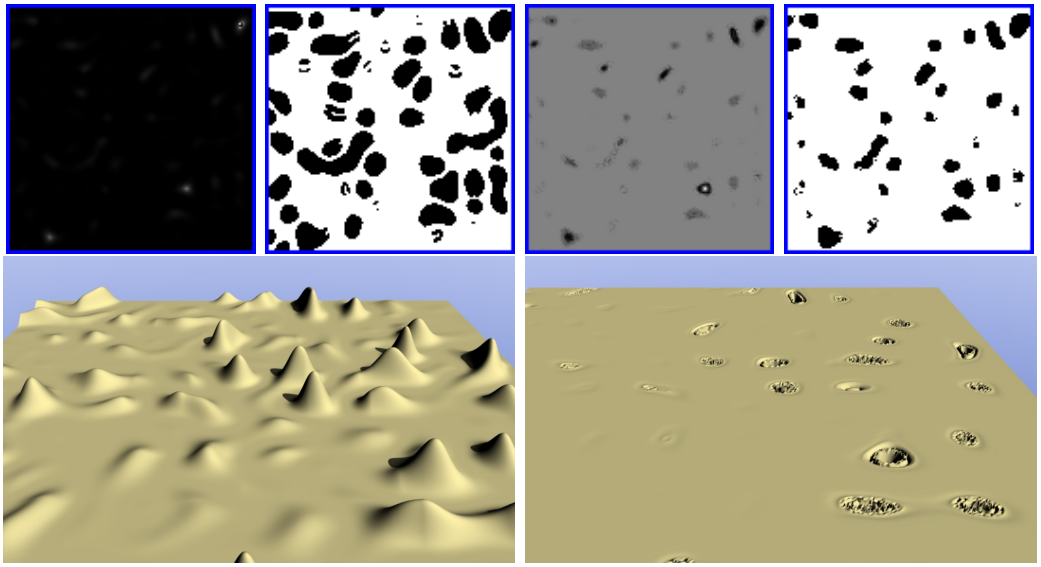


Figure 5.24: Terrains generated by TP $T_1, s_3, pa_1, pe_2, w_{10}, r_{10}$ with *fitness*= 0.000000 on the left, and $T_1, s_3, pa_3, pe_2, w_{10}, r_8$ with *fitness*= 0.000000 on the right

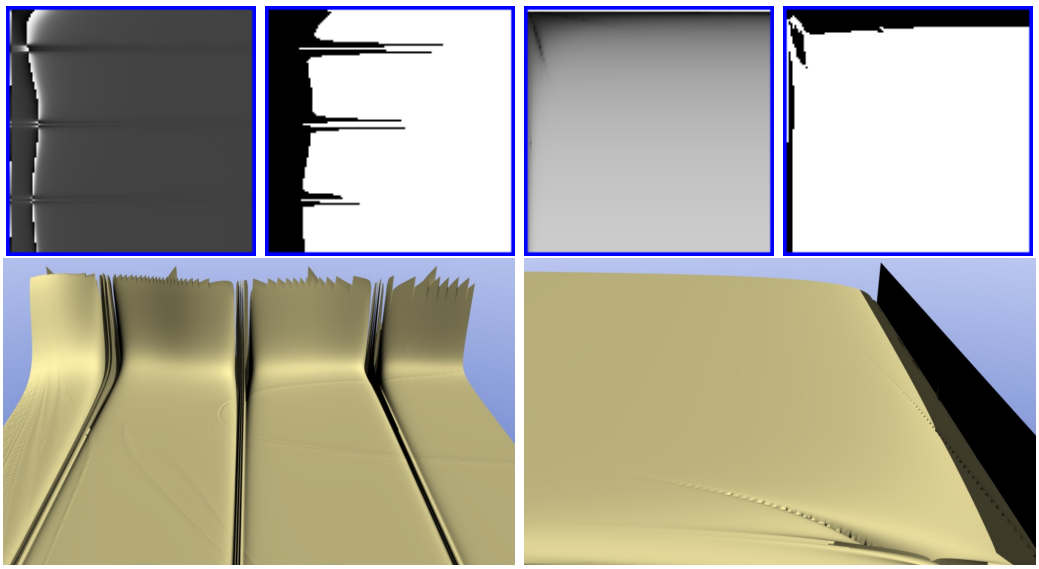


Figure 5.25: Terrains generated by TP $T_2, s_2, pa_2, pe_1, w_{10}, r_{19}$ with *fitness*= 0.000000 on the left, and $T_2, s_2, pa_3, pe_1, w_{10}, r_{11}$ with *fitness*= 0.000000 on the right

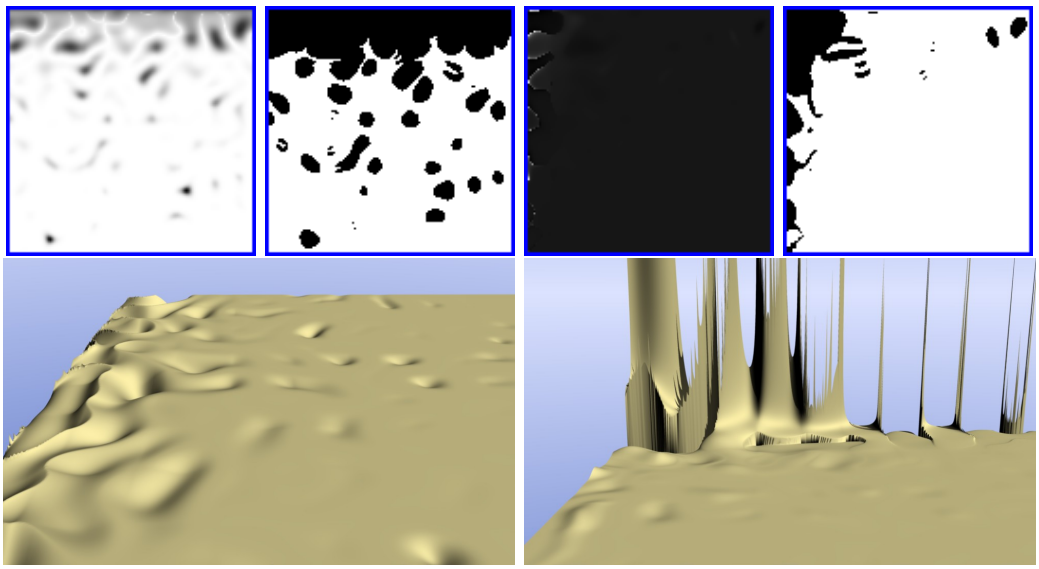


Figure 5.26: Terrains generated by TP $T_3, s_1, pa_1, pe_2, w_{10}, r_{15}$ with *fitness*= 0.000000 on the left, and $T_3, s_3, pa_3, pe_3, w_{10}, r_1$ with *fitness*= 0.000000 on the right

diversity. Terminal set T_1 is the one that has the lowest diversity. We found several terrains that were quite similar, one example is the right terrain from Fig. 5.27 and the left one from Fig 5.29. This similarity is due the small number of terminals in T_1 and the high frequency value of $myNoise$. T_3 presents more diversity than T_1, but the influence of terminal $myNoise$ is quite noticeable, which was expected given its high rate of occurrence shown on Fig. 5.17. The impact of terminals X and Y is also perceptible, but much more subtle. For instance, on left terrain of Fig. 5.35 it is possible to see the wave shape of the terrain (this feature is easier to perceive on the gray scale image), although with a very small amplitude. On the right terrain of the same figure the height values steadily increase along the Y axis (see also correspondent gray scale image). Another good example of X and Y terminals influence on T_3 are both terrains shown on Fig. 5.38, where the terrains change their look at a given point, abruptly on the left terrain and smoothly on the right one. Terminal set T_2 is the one that presents more diverse terrains. From the analyzed samples we have not found terrains with a high degree of similarity as the example mentioned previously. However, terrains from T_2 tend to present geometric patterns and symmetry, which give them a very strange look.

Results regarding diversity were somehow expected, given our experience on previous work. Still, we had hope that the combination of the accessibility and edge length metrics would have a positive impact on diversity. Our hopes increased when the overlap values (presented on Fig. 5.20), showed smaller overlap values when both metrics were used. However, after performing our visual inspection we can not state that the diversity of terrains has increased. We believe that the increase of diversity can be better addressed by fine-tuning the terminal set.

In spite of the overall differences between terrains, some of them present the same feature contours, see for example terrains from Fig. 5.35 (left), 5.36 (right) and 5.37 (left). The terminal *myNoise(x,y)* is the responsible for these contour similarities because it depends only on x and y implicitly. To minimize or even eliminate these similarities *myNoise(x,y)* could be transfered

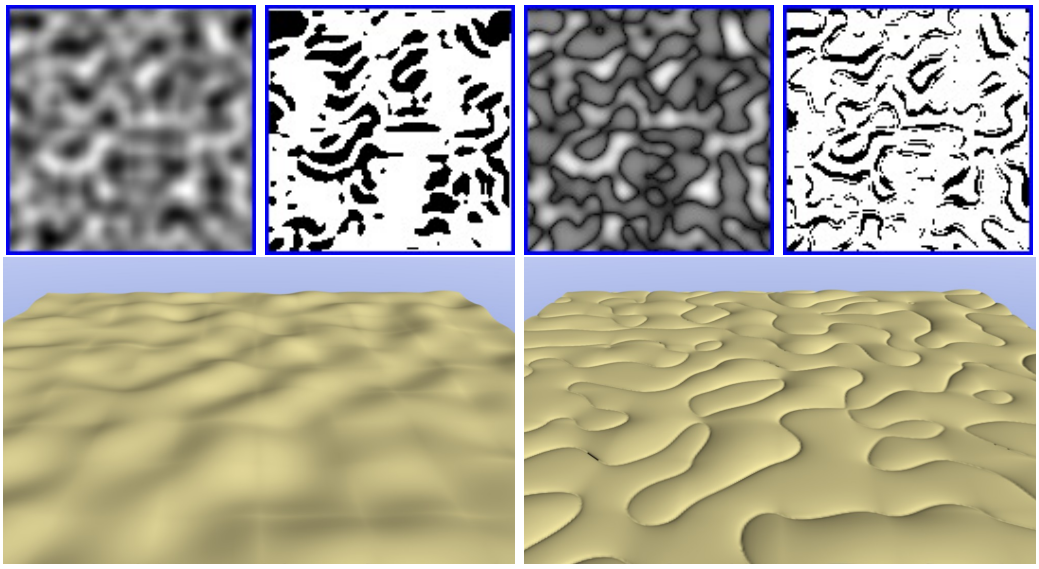


Figure 5.27: Terrains generated by TP $T_1, s_1, pa_1, pe_1, w_2, r_5$ with *fitness*= 0.000000 on the left, and $T_1, s_1, pa_2, pe_3, w_4, r_{16}$ with *fitness*= 0.000445 on the right

to the function set. This way expressions could be evolved as input parameters which would have impact in the frequency, amplitude and phase of the underlying noise function and consequently the terrain shape and contours.

We also noticed an unexpected side effect of using both metrics to generate terrains. Generally, the amplitude of terrains (the difference between the lowest and highest height values) was very small. The left terrain from Fig. 5.30 is one of the few exceptions, but even that one does not present high amplitudes as some terrains obtained for $w_a = 1$. This was strange, because we do not impose any restriction to height values. Our function set (see Table 5.1) is composed by continuous functions, with only three exceptions: $myLog(a)$ when $a = 0$, $myDivide(a, b)$ when $b = 0$ and $myPower(a, b)$ when $a = 0$ and $b < 0$. We thought those exceptions were enough to create sudden changes in terrain and create height obstacles this way. However, to accomplish the required edge length terrain height values must change often. Therefore, we believe the edge length metric is the main responsible for small amplitude terrains, specially with the chosen *pe* values. Frequency results in

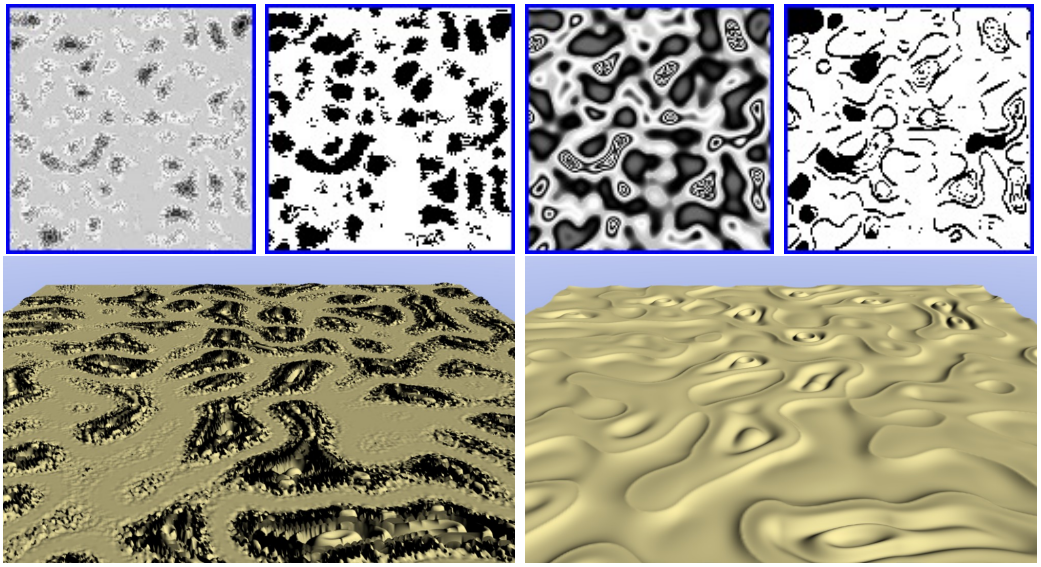


Figure 5.28: Terrains generated by TP $T_1, s_2, pa_1, pe_2, w_9, r_9$ with *fitness*= 0.000098 on the left, and $T_1, s_2, pa_2, pe_3, w_8, r_1$ with *fitness*= 0.000000 on the right

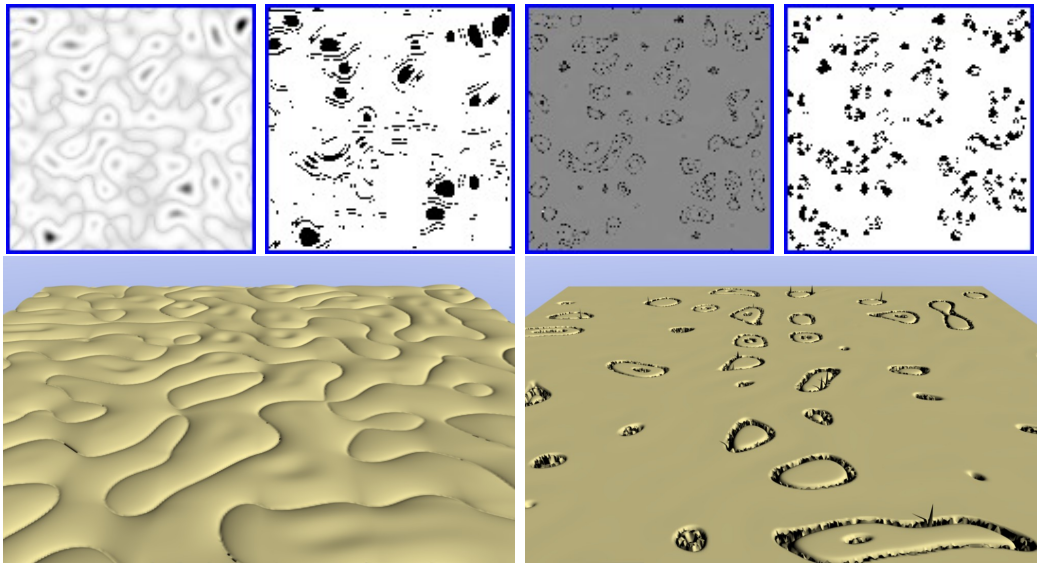


Figure 5.29: Terrains generated by TP $T_1, s_2, pa_3, pe_1, w_1, r_{14}$ with *fitness*= 0.000053 on the left, and $T_1, s_2, pa_3, pe_1, w_5, r_2$ with *fitness*= 0.000000 on the right

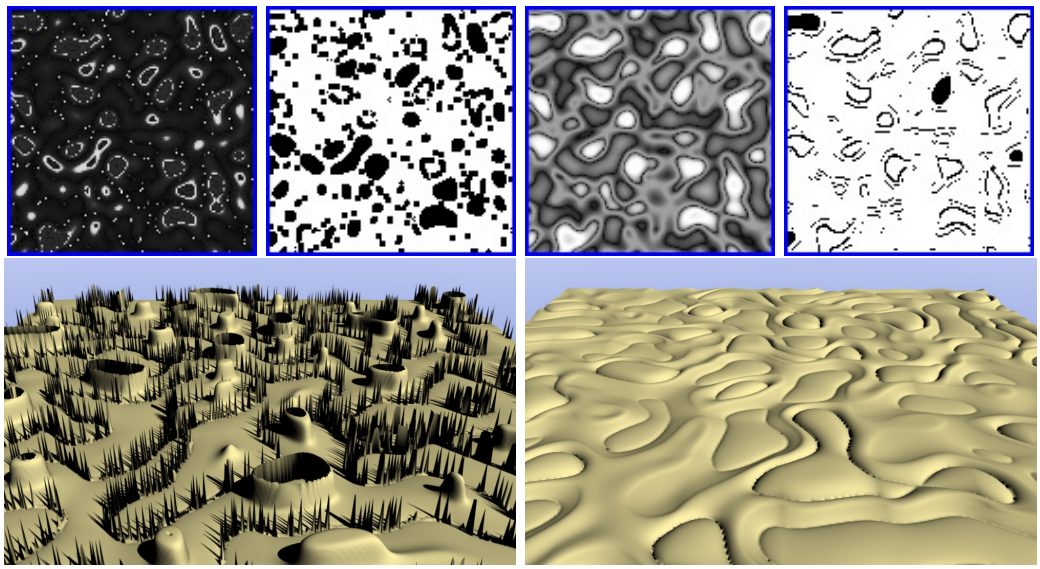


Figure 5.30: Terrains generated by TP $T_1, s_3, pa_1, pe_2, w_4, r_{18}$ with *fitness*= 0.000000 on the left, and $T_1, s_3, pa_3, pe_2, w_5, r_{10}$ with *fitness*= 0.000151 on the right

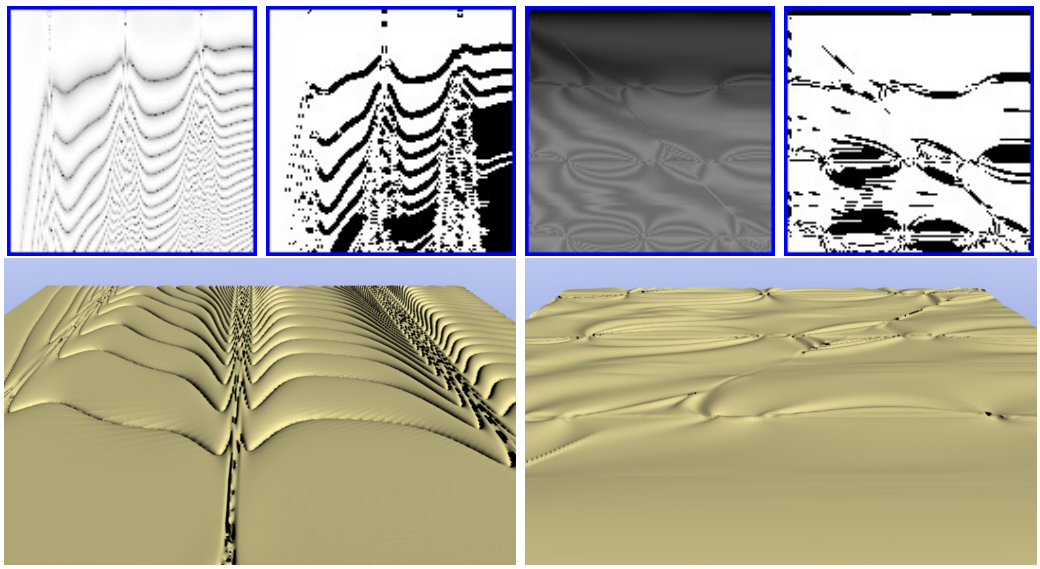


Figure 5.31: Terrains generated by TP $T_2, s_1, pa_1, pe_2, w_7, r_2$ with *fitness*= 0.000000 on the left, and $T_2, s_1, pa_2, pe_1, w_6, r_4$ with *fitness*= 0.000400 on the right

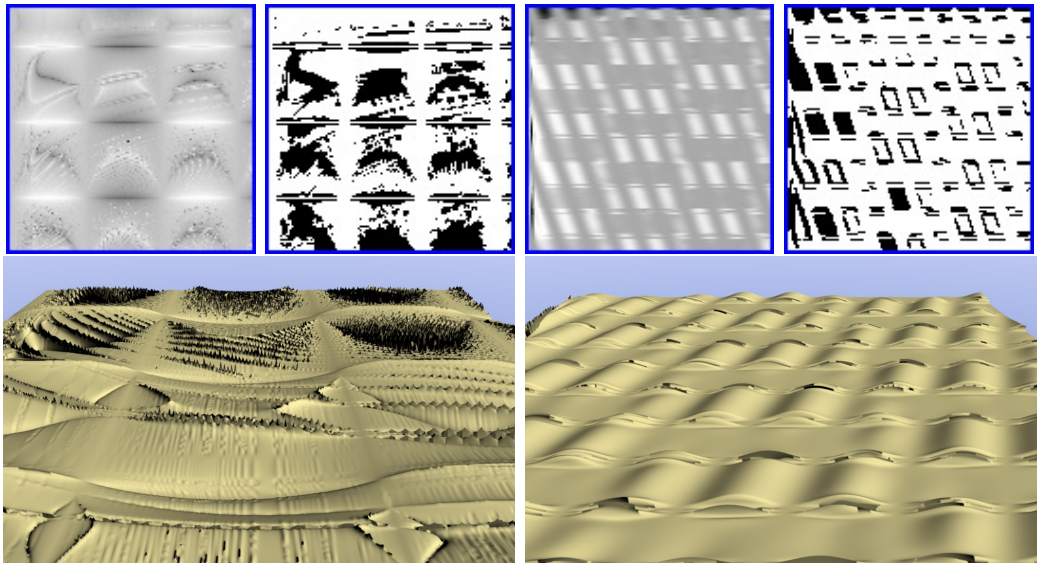


Figure 5.32: Terrains generated by TP $T_2, s_2, pa_1, pe_2, w_9, r_9$ with *fitness*= 0.000000 on the left, and $T_2, s_2, pa_2, pe_2, w_1, r_{18}$ with *fitness*= 0.001144 on the right

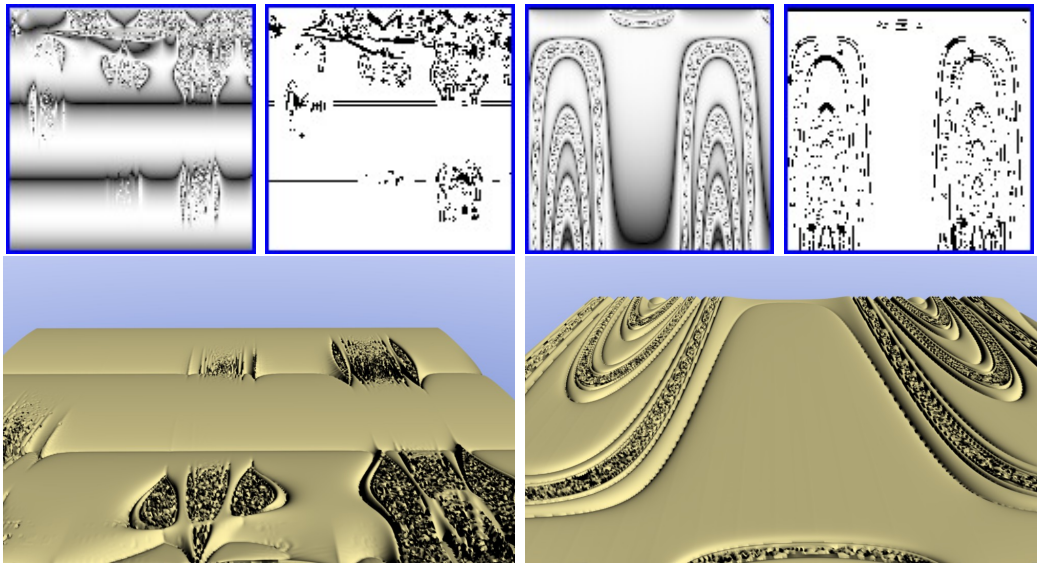


Figure 5.33: Terrains generated by TP $T_2, s_2, pa_3, pe_1, w_8, r_3$ with *fitness*= 0.000181 on the left, and $T_2, s_2, pa_3, pe_2, w_9, r_8$ with *fitness*= 0.000068 on the right

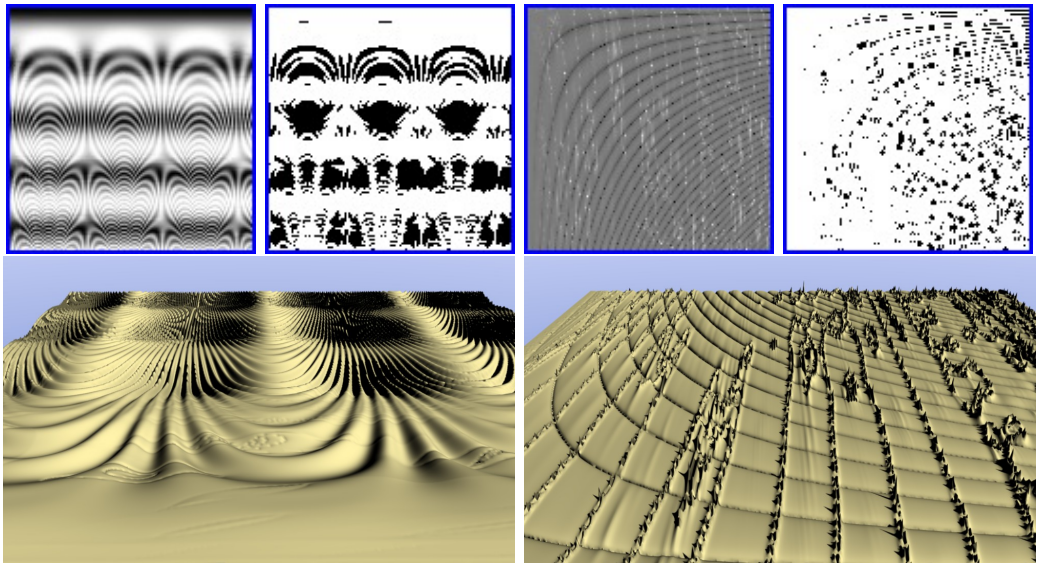


Figure 5.34: Terrains generated by TP $T_2, s_3, pa_1, pe_2, w_2, r_{13}$ with *fitness*= 0.000199 on the left, and $T_2, s_3, pa_3, pe_3, w_1, r_2$ with *fitness*= 0.000015 on the right

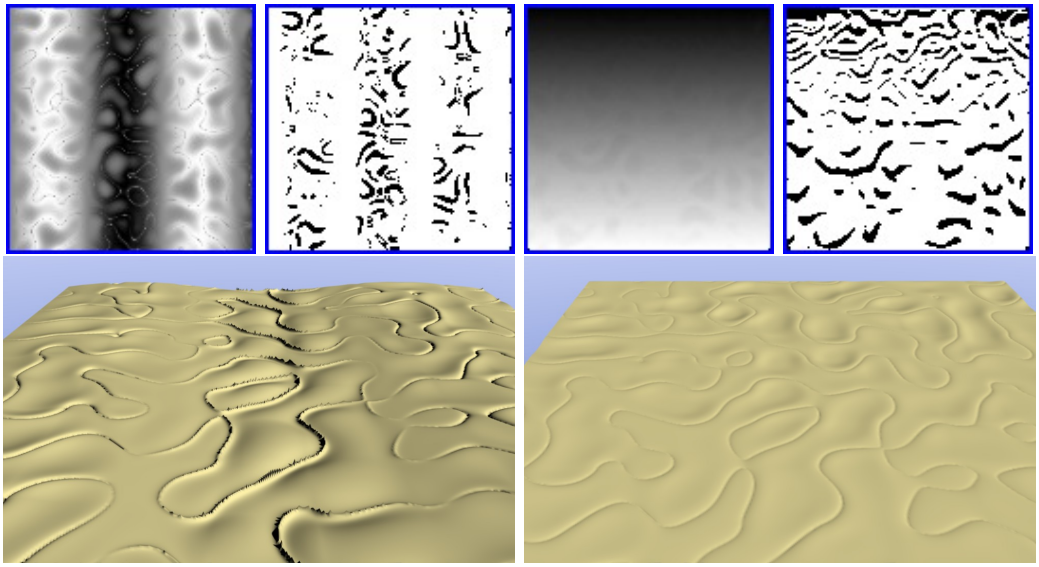


Figure 5.35: Terrains generated by TP $T_3, s_1, pa_3, pe_1, w_2, r_6$ with *fitness*= 0.008510 on the left, and $T_3, s_1, pa_3, pe_2, w_4, r_{11}$ with *fitness*= 0.042003 on the right

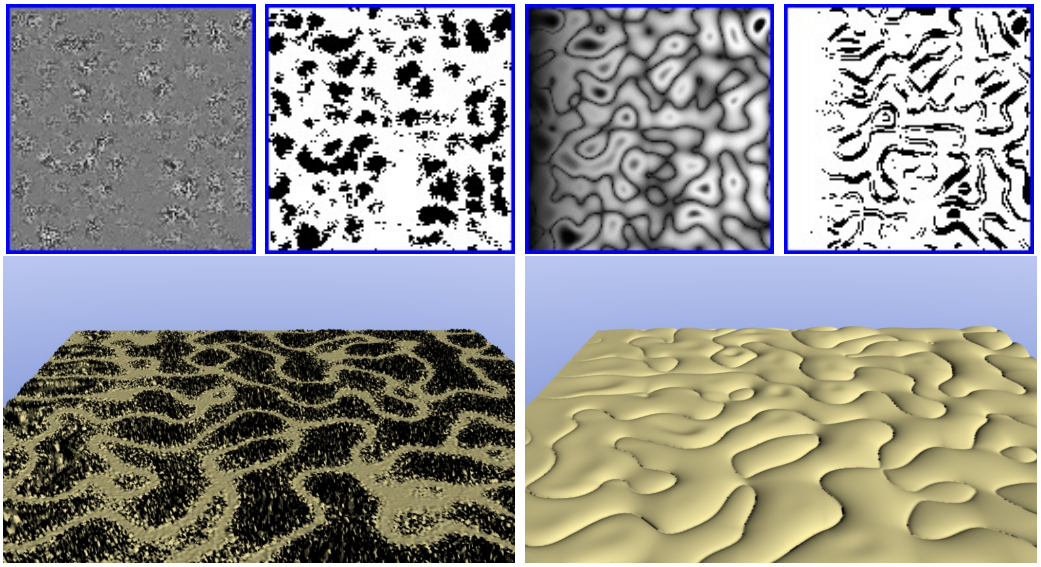


Figure 5.36: Terrains generated by TP $T_3, s_2, pa_1, pe_3, w_8, r_{17}$ with *fitness*= 0.000470 on the left, and $T_3, s_2, pa_2, pe_3, w_4, r_{16}$ with *fitness*= 0.000000 on the right

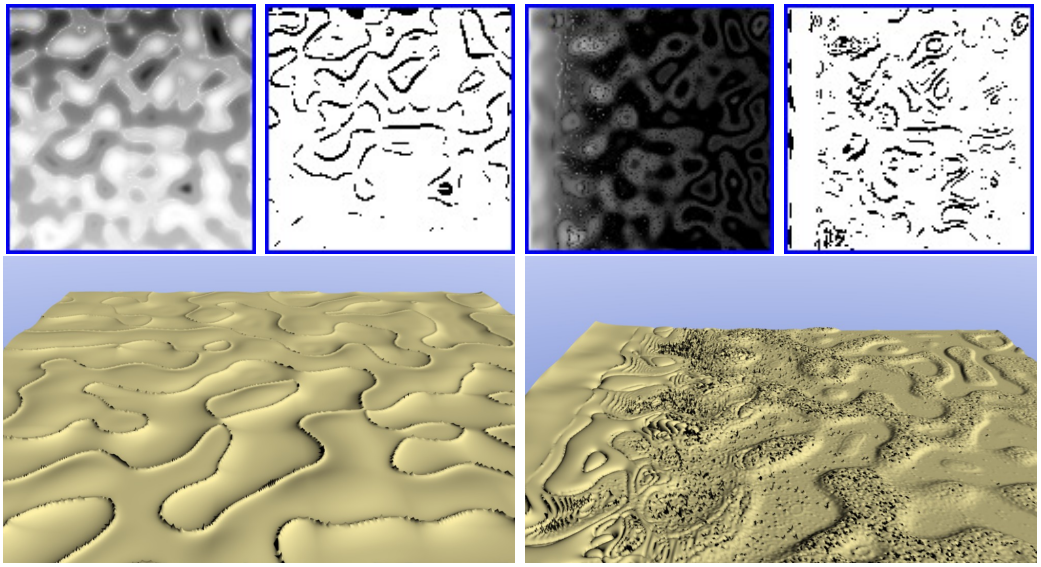


Figure 5.37: Terrains generated by TP $T_3, s_2, pa_3, pe_1, w_8, r_{16}$ with *fitness*= 0.000060 on the left, and $T_3, s_2, pa_3, pe_2, w_7, r_8$ with *fitness*= 0.002825 on the right

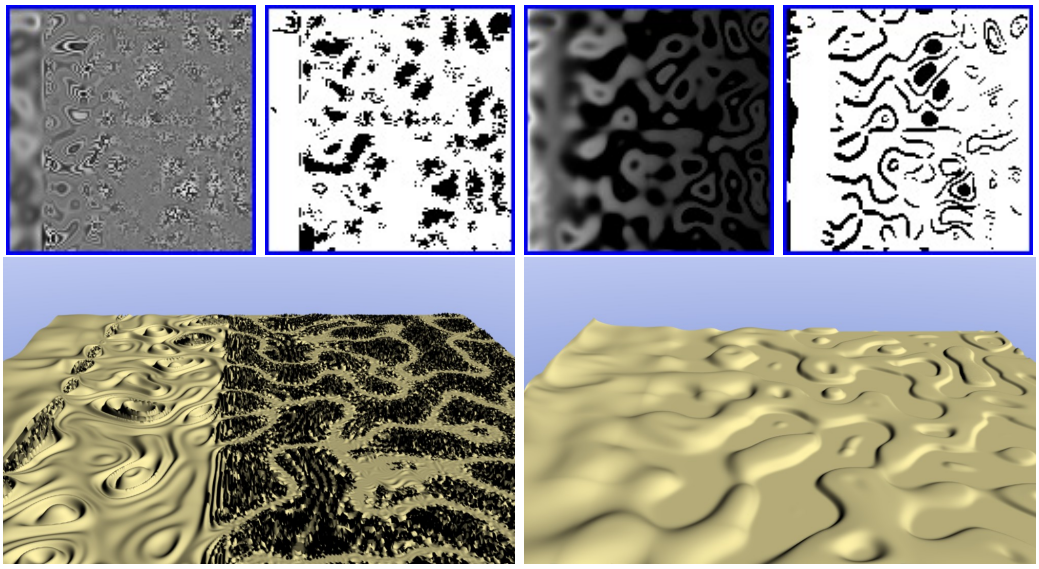


Figure 5.38: Terrains generated by TP $T_3, s_3, pa_2, pe_2, w_8, r_{10}$ with *fitness*= 0.000000 on the left, and $T_3, s_3, pa_3, pe_2, w_2, r_8$ with *fitness*= 0.020270 on the right

Fig. 5.18 corroborate this reasoning, because only for $w_a = 1$ the frequency values change significantly, besides the presence of periodic functions, like *cos* and *sin*, decrease. Still, we think further tests with smaller *pe* values should be performed to confirm whether they allow terrains with bigger amplitudes.

Another option to address the amplitude issue would be to include one or more functions with discontinuous behavior on the function set, for example *mod* (remainder for the modulo operation) or the if statement. However, in this case we think that some additional measures should be taken to prevent those discontinuous functions to dominate the solutions, which would prevent the appearance of smooth terrains. One of these measures could be different probability values for a given function to be chosen from the function set.

Although the picked slope values would have a severe impact on the mobility of vehicles, their differences were not big enough to impact terrains on a visible way. In fact, considering the results regarding the GP system, overall *slope* has a very limited influence, being only significant on fitness values. Therefore, we think further tests must be performed with *slope* values

covering a bigger range to access if they can influence terrains smoothness.

As stated previously, TPs generate a continuous surface that needs to be sampled and limited to generate the height map. This is achieved by Eq. (5.1). which also allow us to control the zoom level (through S_x and S_y) and resolution (through n_r and n_c). Both zoom level and resolution control are important features. The zoom level allows video games to compute only a small portion of the terrain that needs to be displayed. This can be used to simulate a player approaching or getting away from a particular point in the terrain, see Fig. 5.39. On the other hand, resolution will allow video game developers to control the amount of processing required to generate the terrain at the expense of terrain details.

5.7 Creativity

Although we have successfully tested our technique, the question about how creative GTP_a remains open. One of the most influential research on how to assess software creativity comes from Ritchie (2007). He proposes a set of criteria to assess programs' creativity based on the artifacts they produce. Pereira et al (2005) apply Ritchie's criteria to a set of systems and suggest also that if a program repeats itself later on it is a sign of less creativity. On the other hand, Colton (2008) argues that creativity assessment based only on produced artifacts is not enough. He suggests that creativity assessment should account also for the process the software performs and assess its functionality. While software creativity assessment is still contentious, we decided to release a TPs database to establish a comparison base for future research regarding creativity of GTP_a as well as aesthetic terrains diversity (Frade et al, 2012a).

The database, formated as comma separated values (CSV) file, contains the results from our 17 820 different executions, with the following fields: *terminal*; *slope*(%); *pa*(%); *pe*(%); w_a; *run*; *fitness*; TP;. The *fitness* value is standardized, so lower values are better. Although the fitness value does

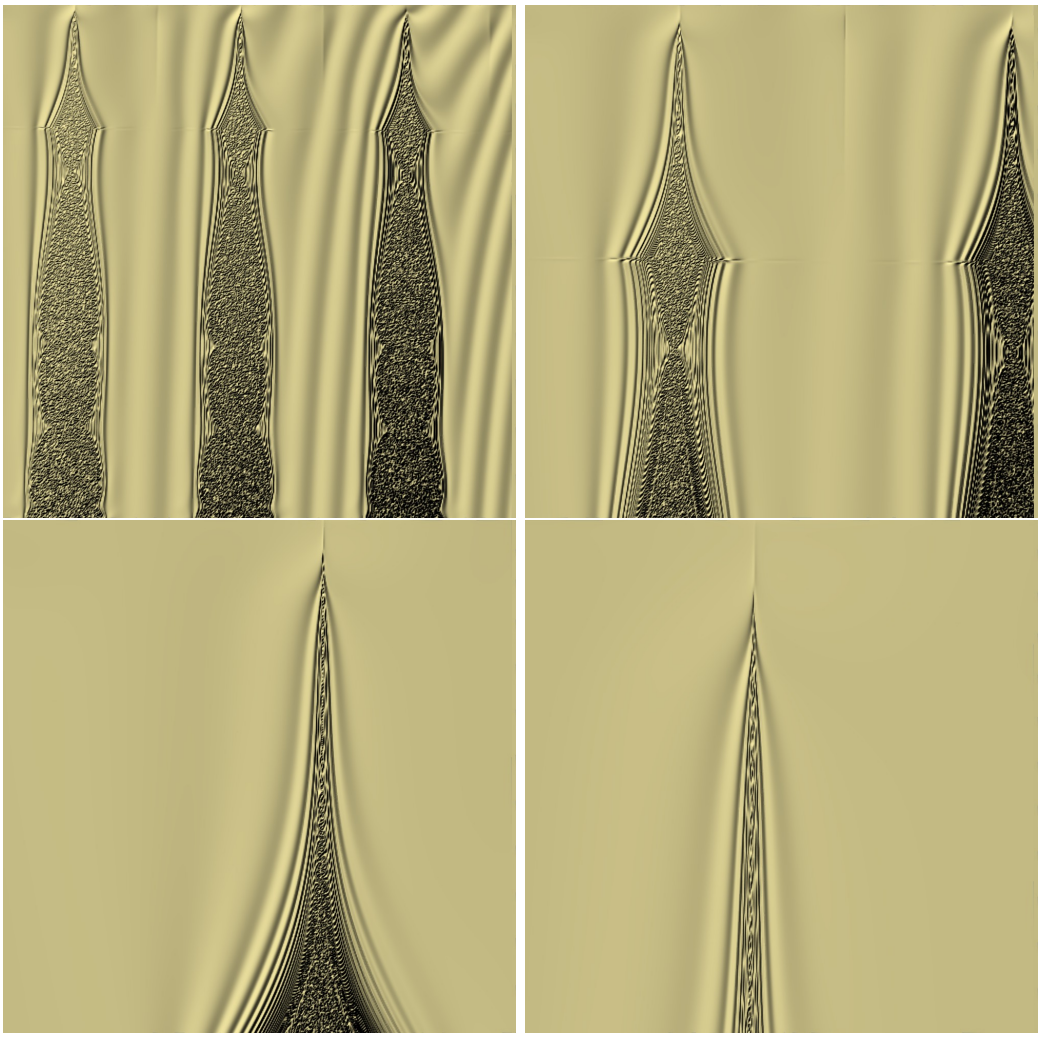


Figure 5.39: Top view of TP $T_2, s_1, pa_3, pe_3, w_5, r_1$ with 4 different zoom levels: $S_x = S_y = 1$, 2, 4 and 8 (see Eq. 5.1)

not give any information regarding the creativity of our system, it is included to indicate how feasible/unfeasible a given TP is regarding our metric. The amount of TPs that reached the perfect fitness value (zero) was 45.22%. Bellow is an example line of the CSV file:

```
T3; 18; 70; 20; 0.0; 09; 0.00000000; myPower(cos(myNoise(X,Y)),exp(myNoise(X,Y)));
```

Due to the large amount of results, we decided to split our database by terminal set: `TPs_T1.csv`, `TPs_T2.csv` and `TPs_T3.csv`. The database is available to the public in the Sourceforge repository http://sourceforge.net/p/tps-db/ under the Creative Commons Attribution-ShareAlike 3.0 Unported License[4]. We have also added to the repository some C code to show how to calculate the height values from TPs.

One of our goals is to find diverse solutions, which can also be considered a way to assess the creativity of GTP_a (on a limited sense) (Pereira et al, 2005). So, in our preliminary assessment of creativity we looked for any repeated TPs in our database. We found 98.61% of unique genotypes and a total of 106 TPs that appeared more than once, relative to 248 runs (1.39%). Figure 5.40 shows how the repeated TPs are distributed in relation to w_a. The higher concentration of repeated TPs is where w_a values have worse fitness values (see Fig. 5.15), specially for $w_a = 0.1$. For $w_a = 0.7$ there are no repeated TPs. On average, the repeated TPs appear as solution of 2.34 runs. Equation (5.22) shows the worse case, it appeared 8 times, Table 5.6 identifies the runs where this particular TP appeared and the correspondent fitness values. We believe that a larger limit of maximum allowed generations can drastically reduce, or even eliminate, the amount of repeated TPs, but more tests are needed to confirm it.

$$TP = \quad cos(cos(atan(atan(atan(atan(myNoise(x, y))))))) \tag{5.22}$$

[4]License available at http://creativecommons.org/licenses/by-sa/3.0/

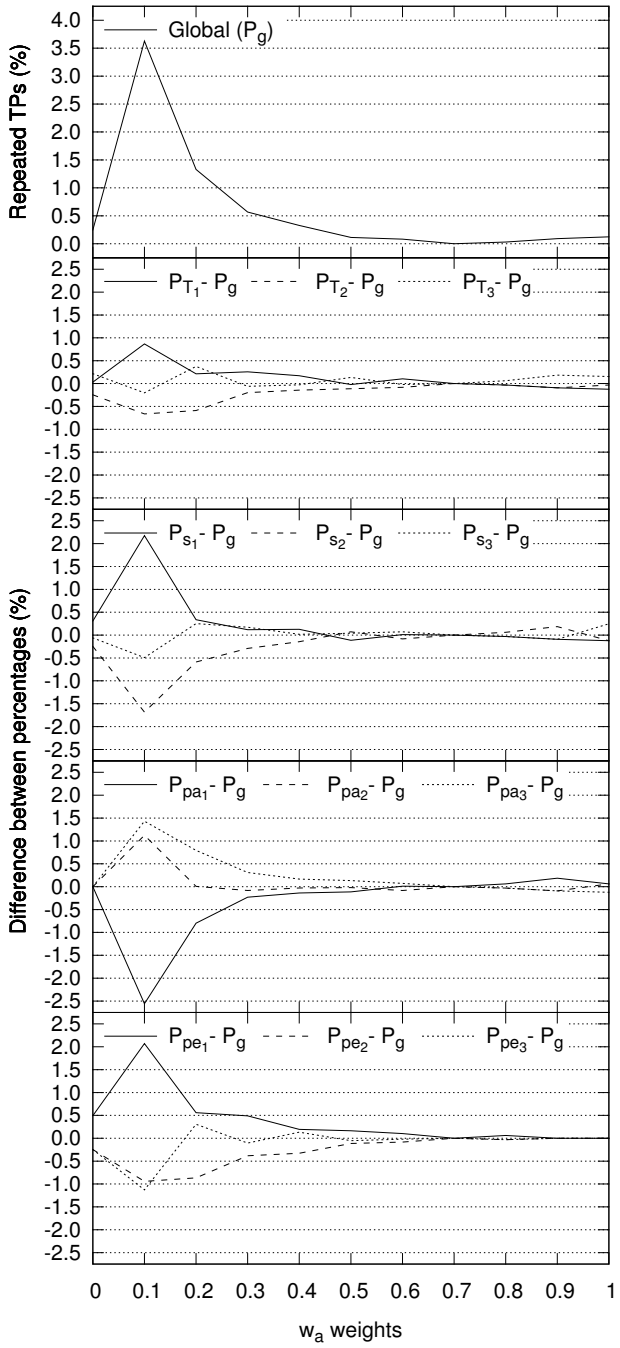


Figure 5.40: Percentage of repeated TPs versus w_a.

Table 5.6: Runs where TP shown in Eq. (5.22) was the best solution and their correspondent fitness values.

terminal	*slope*	*pa*	*pe*	w_a	*run*	*fitness*
T_1	$s18$	$a80$	$e20$	0.1	18	0.03484
T_1	$s18$	$a90$	$e20$	0.1	18	0.04143
T_3	$s18$	$a80$	$e20$	0.1	6	0.03484
T_3	$s18$	$a80$	$e20$	0.1	7	0.03484
T_3	$s18$	$a80$	$e20$	0.3	7	0.03438
T_3	$s18$	$a90$	$e20$	0.1	6	0.04143
T_3	$s18$	$a90$	$e20$	0.1	7	0.04143
T_3	$s18$	$a90$	$e20$	0.3	7	0.05390

Our repetition analysis only accounts for different terrains genotypes. There might exist also different TPs that are mathematically equivalent and render the same terrain, which was partially addressed by the overlap analysis (see Fig. 5.20). However, there are other sources of similarity that cannot be accounted for using the overlap, e.g., terrains that are rotated, shifted, and/or scaled, or that just differ in fine details, but share the same global structure. From the creativity point of view, it is important to inspect also how similar (or diverse) are the phenotypes. To answer the question of how many diverse terrains types GTP_a is able to generate further analysis must be conducted.

Chapter 6

Chapas Video Games

Many educators have taken an interest in the effects that video games have on players, and how some of the motivating aspects of video games might be harnessed to facilitate learning (Squire, 2003). Initial studies comparing video game teaching effectiveness to the classic lecture show positive improvements, typically 30% or more (Mayo, 2007). These results point out the important role video games can play on education. Games are also ideal test beds for computational intelligence theories, architectures and algorithms (Lucas and Kendall, 2006). For these reasons we wanted to test the suitability of GTP_a technique to generate terrains on a real game. This was the genesis of the Chapas video game project, a partnership between Centro Universitario de Mérida (Universidad de Extremadura), GLOW[1] and Junta de Extremadura, Spain.

Chapas[2] is an open source turn-based bottle-cap racing game, with 3D graphics, where the players strategically control the racers with cards. A typical round of Chapas starts by dealing 9 random cards to each player (see Fig. 6.1). Each of these cards represents movement points for the different bottle-caps. Afterwards, an auction takes place, where the players use their money (an initially set amount, that accumulates over different rounds), to

[1] An animation studio http://www.theglow.es

[2] Available for download at https://sourceforge.net/projects/chapas/

Figure 6.1: Screenshots of *Chapas* video game were the terrain was generated online by a TP.

buy the bottle-caps better suited to each player's cards. Finally the race phase starts, where the players use their cards, on their respective turn, to move the bottle-caps across the field, passing through each checkpoint (where they can restock their cards for money). The round ends when every player has reached the finish line, or ran out of cards.

To show the usability of TPs as a procedural technique to generate terrains dynamically on a real video game, a few selected TPs were embedded in Chapas video game. The height maps are computed from the TP expression using the equation define in (5.1). Details regarding Chapas video game and physics engines can be found in Rodrigues et al (2010). Figure 6.2 presents a couple of screenshots of Chapas video game in the running phase.

Figure 6.2: Screenshots of *Chapas* video game in the running phase, were the terrain was generated online by a TP.

Chapter 7

Conclusions

A new approach, designated GTP, was developed to address some limitations of existing procedural techniques, namely: the modeling problem of analytical functions; and the lack of procedures able to generate a wide range of terrains types with focus on aesthetic. Genetic programing was used as evolutionary tool with these two goals in mind. The first implementation, GTP_i, applied an interactive approach similar to evolutionary art systems. Although the results showed aesthetic appealing terrains and a broad range of terrain types, TPs generated this way presented two limitations. First, some terminals prevented the implementation of a scaling (zoom) function. The lack of this feature makes TPs less viable for video games. Second, the interactive evolutionary process not only depends on expensive human resources, but is also prone to user fatigue.

To overcome these limitations an automated version was developed. GTP_a searches procedures that are able to generate terrains according to the weighted sum of two metrics: accessibility score and edge length score. The parameters allow us to control the slope threshold (that differentiate the accessible from the inaccessible terrain areas), how much area should be made accessible, and the edge length of the inaccessible areas. Throughout a series of experiments we have shown that our system is able to find many different so-

lutions that fit our fitness function. Both accessibility score and edge length metrics perform the desired function, but our results show that GTP_a can achieve better fitness values for accessibility score than for the edge length metric. The combination of the two metrics also helps to decrease terrain similarities, as shown by the overlap metric, this characteristic is desirable as it means more diverse solutions. However, it will not increase terrain types diversity when compared with the use of a single metric. This combination also presents the side effect of generating terrains with small amplitudes (the difference between the lowest and highest height values), whose main responsible is the edge length score function. We believe this problem can be addressed using lower *pe* values and by introducing more discontinuous functions in the function set.

Repeated TPs represent 1.39% of the solutions and appear when the fitness values are worse, but overall 45.22% of the solutions reached the perfect score of zero. Both situations, increasing the amount of solutions reaching fitness value of zero and reducing the amount of repeated TPS, can be addressed by increasing the maximum allowed generations. This will also help terminal set T_3 to have better fitness values, because the more elements the terminal set has, the bigger the search space is, and more generations will be required to find a good solution. Chosen slope threshold values also influence fitness values, were $s_1 = 20\%$ presented the worse results, however its impact is negligible for the remaining GP system performance.

The search for the right TP can be long, depending mainly on how many generations are allowed, population size and used metrics, but this is a common characteristic of search-based techniques. However, once found TPs execution times are short and in the same order of magnitude of other procedural techniques. TPs have also the advantages of offering room for execution time improvements as they are easily parallelized, a feature that many procedural techniques do not present. To create a terrain from a TP only height map parameters are needed, but these only concern terrain resolution, zoom level and origin, not its look. Therefore, TPs do not require any input parameter to model terrain shape and there is no need for a time consuming

and expensive phase of parameter tunning. To prove the viability of our technique some TPs are already in use on a real video game, were the terrain generation occurs online.

As expected, terminal sets have a big impact on terrain diversity, look and aesthetic appeal. Terminal set T_1 has few diversity, but showed us the potentiality of implicit functions as terminal by providing appealing surfaces without pattern repetition. On the other hand, T_2 has many diverse terrain types, but exhibits many geometric patterns. Finally, T_3, which is the union of the other two terminal sets, reinforces the importance of $myNoise$ terminal to achieve fit solutions. Therefore, most terrains produced with T_3 present a heavy influence of $myNoise$ terminal on its looks. In regard to slope parameter, it did not present a significant change in terrains looks, which we believe to be a direct consequence of a narrow range of the used values.

Chapter 8

Future Work

In spite of the interesting results, this work opens many challenges for future research. Logically the next step would be to test our system under a multi-objective approach, given that we used two different metrics and more could be added this way. Nevertheless, there are many topics that can also be addressed in future work. For instance, the prevalence of $myNoise$ on T_3 showed us that fractal based function are important to find fitter solutions with an interesting aesthetic appeal. However, the diversity of terrain types is not big enough, so it could be augmented by adding more discontinuous functions to the function set and by adding new fractal based functions. Given the current used metrics discontinuous functions can overtake the predominant role and that way avoiding the appearance of smooth terrains. So, a new line of work can be the study of different probabilities for the functions to be selected from their set. With new fractal based functions we could obtain changes in frequency and amplitude on terrain features, but this approach introduces new questions. An open question is whether the new fractal functions should be introduced as terminals with implicit parameters, or whether those parameters should evolve as well. Some fractal based functions present parameters like octaves and lacunarity whose values are valid or interesting only on a limited range. So, if we let those parameters to evolve the values must be normalized, which raises the question of what

normalization function to use.

Another possible research line could be to try new metrics from the geomorphology field to see if this way it would be possible to obtain more realistic terrains. Some researchers claim it is possible to classify all real terrains with only 3 parameters designated as geometric signatures (Iwahashi and Pike, 2007), using them as a search criteria can be of interest. Other types of search criteria can also be studied, for instance level curves to define desired terrain shape, instead of parameters.

So far our creativity analysis on GTP_a is on a preliminary stage and further studies must be conducted, like applying Ritchie's criteria Ritchie (2007). Another interesting research would be the use of classification system to aggregate terrains by their morphological similarity and this way assess phenotype diversity. This approach poses some challenges on which metric should be used to classify morphological similarity. A different possibility would be to perform a user study to classify terrains creativity characteristics, like novelty or quality, and its impact on video games replayability. GTP_a evaluates TPs after converting them to height maps, however with this approach if we change the resolution n_r and n_c, their fitness value will likely change. This dependence on the chosen resolution is not desirable, so other approach could be devised to evaluate TPs based on their equations rather than on their phenotype.

Bibliography

Ashlock D, Gent S, Bryden K (2008) Embryogenesis of artificial landscapes. In: Hingston PF, Barone LC, Michalewicz Z (eds) Design by Evolution, Natural Computing Series, Springer Berlin Heidelberg, pp 203–221, URL http://dx.doi.org/10.1007/978-3-540-74111-4_12

Belhadj F (2007) Terrain modeling: a constrained fractal model. In: 5th International conference on CG, virtual reality, visualisation and interaction in Africa, ACM, Grahamstown, South Africa, pp 197–204, DOI 10.1145/1294685.1294717

Belhadj F, Audibert P (2005) Modeling landscapes with ridges and rivers: bottom up approach. In: GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia, ACM, New York, NY, USA, pp 447–450, DOI http://doi.acm.org/10.1145/1101389.1101479

Bentley P (1998) Aspects of evolutionary design by computers. In Advances in Soft Computing - Engineering Design and Manufacturing, Springer-Verlag

Bentley P (1999) Evolutionary Design by Computers. Morgan Kaufmann Publishers, Inc., CA, USA

Bourg D, Seemann G (2004) AI for game developers. O'Reilly Media

Bracewell RN (1999) The Fourier Transform & Its Applications, 3rd edn. McGraw-Hill Science/Engineering/Math

Brosz J, Samavati FF, Sousa MC (2006) Terrain Synthesis By-Example. In: First International Conference on Computer Graphics Theory and Applications

Carpentier G, Bidarra R (2009) Interactive GPU-based procedural heightfield brushes. In: Proceedings of the 4th International Conference on Foundations of Digital Games, ACM New York, NY, USA, pp 55–62

Chiang M, Huang J, Tai W, Liu C, Chang C (2005) Terrain synthesis: An interactive approach. In: International Workshop on Advanced Image Tech

Colton S (2008) Creativity versus the perception of creativity in computational systems. In: Creative Intelligent Systems: Papers from the AAAI Spring Symposium, pp 14–20

Darwin C (1859) On the Origin of Species by Means of Natural Selection. John Murray

Doran J, Parberry I (2010) Controlled Procedural Terrain Generation Using Software Agents. IEEE Transactions on Computational Intelligence and AI in Games 2(2)

Duchaineau M, Wolinsky M, Sigeti D, Millery M, Aldrich C, Mineev-Weinstein M (1997) ROAMing terrain: Real-time optimally adapting meshes. In: VIS '97: Proceedings of the 8th conference on Visualization '97, IEEE Computer Society Press, Los Alamitos, CA, USA, pp 81–88

Ebert D, Musgrave K, Peachey D, Perlin K, Worley S (2003) Texturing and Modeling: A Procedural Approach, 3rd edn. Morgan Kaufmann

Edwards R (2006) The Economics of Game Publishing. Website (accessed on Sep. 2011), http://uk.games.ign.com/articles/708/708972p1.html

Fogel L, Owens A, Walsh M (1966) Artificial intelligence through simulated evolution. Wiley

Forbus KD, Mahoney JV, Dill K (2002) How Qualitative Spatial Reasoning Can Improve Strategy Game AIs. IEEE Intelligent Systems 17(4):25–30, DOI http://dx.doi.org/10.1109/MIS.2002.1024748

Frade M (2008) Genetic terrain programming. Master's thesis, University of Extremadura, Cáceres, Spain

Frade M, Fernández de Vega F, Cotta C (2008a) Genetic terrain programming - an aesthetic approach to terrain generation. In: Computer Games and Allied Technology 08, Singapore, pp 1–8

Frade M, Fernández de Vega F, Cotta C (2008b) Gentp – uma ferramenta interactiva para a geração artificial de terrenos. In: Cota MP (ed) Third Iberian Conference in Systems and Information Technologies (CISTI 2008), LibroTeX, Ourense, Spain, vol 2, pp 655–666, iSBN 978-84-612-4840-7

Frade M, Fernández de Vega F, Cotta C (2008c) Modelling Video Games' Landscapes by Means of Genetic Terrain Programming - A New Approach for Improving Users' Experience. In: Giacobini M, et al (eds) Applications of Evolutionary Computing, Springer, Napoli, Italy, Lecture Notes in Computer Science, vol 4974, pp 485–490

Frade M, Fernández de Vega F, Cotta C (2009a) Adding Zoom Feature to Terrain Programmes. In: VI Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB'09), Málaga, Spain, pp 293–300

Frade M, Fernández de Vega F, Cotta C (2009b) Breeding Terrains with Genetic Terrain Programming - The Evolution of Terrain Generators. International Journal for Computer Games Technology 2009(Article ID 125714):13, DOI 10.1155/2009/125714

Frade M, Fernández de Vega F, Cotta C (2010a) Evolution of Artificial Terrains for Video Games Based on Accessibility . In: Chio CD, et al (eds) Applications of Evolutionary Computation, Springer, Lecture Notes in Computer Science, vol 6024, pp 90–99

Frade M, Fernández de Vega F, Cotta C (2010b) Evolution of Artificial Terrains for Video Games Based on Obstacles Edge Length. In: IEEE Congress on Evolutionary Computation 2010, pp 1–8, DOI 10.1109/CEC.2010.5586032

Frade M, Fernández de Vega F, Cotta C (2012a) Aesthetic terrain programs database for creativity assessment. In: IEEE Conference on Computational Intelligence and Games, p 5

Frade M, Fernández de Vega F, Cotta C (2012b) Automatic evolution of programs for procedural generation of terrains for video games. Soft Computing - A Fusion of Foundations, Methodologies and Applications pp 1–22, 10.1007/s00500-012-0863-z

Goldberg DE (1989) Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Gonzalez RC, Woods RE (2002) Digital Image Processing, 2nd edn. Prentice Hall

Goodchild M (1980) Fractals and the accuracy of geographical measures. Mathematical Geology 12:85—-98

Hastings E, Guha R, Stanley K (2009) Automatic content generation in the galactic arms race video game. Computational Intelligence and AI in Games, IEEE Transactions on 1(4):245 –263, DOI 10.1109/TCIAIG.2009.2038365

Holland J (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor

Horn B (1981) Hill shading and the reflectance map. Proceedings of the IEEE 69(1):14–47

Iwahashi J, Pike RJ (2007) Automated classifications of topography from DEMs by an unsupervised nested-means algorithm and a three-part geometric signature. Geomorphology 86(3-4):409 – 440, DOI DOI:10.1016/j.

geomorph.2006.09.012, URL http://www.sciencedirect.com/science/article/B6V93-4M6SB5Y-3/2/510ca957d542d84fba3e3af50968fdf8

Jacob C (1996a) Evolution programs evolved. PPSN-IV, Parallel Problem Solving from Nature IV Berlin, Springer-Verlag pp 42–51

Jacob C (1996b) Evolving evolution programs: Genetic programming and L-systems. In: Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press

Jennings-Teats M, Smith G, Wardrip-Fruin N (2010) Polymorph: A model for dynamic level generation. In: Sixth Artificial Intelligence and Interactive Digital Entertainment Conference

Kamal KR, Uddin YS (2007) Parametrically controlled terrain generation. In: GRAPHITE '07: Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia, ACM, New York, NY, USA, pp 17–23, DOI http://doi.acm.org/10.1145/1321261.1321264

Kimbrough SO, Lu M, Wood DH, Wu DJ (2002) Exploring a two-market genetic algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, GECCO '02, pp 415–422, URL http://dl.acm.org/citation.cfm?id=646205.682628

Koza JR (1992) Genetic Programming. On the programming of computers by means of natural selection. Cambridge MA: The MIT Press

Koza JR (2004) Human-competitive results produced by genetic programming. Website, http://www.genetic-programming.com/humancompetitive.html

Lane B, Prusinkiewicz P (2002) Generating Spatial Distributions for Multilevel Models of Plant Communities. In: Proceedings of Graphics Interface 2002

Langdon WB, Qureshi A (1995) Genetic programming – computers using "natural selection" to generate programs. Tech. Rep. RN/95/76, University College London, Gower Street, London WC1E 6BT, UK, URL citeseer.ist.psu.edu/langdon95genetic.html

Li Q, Wang G, Zhou F, Tang X, Yang K (2006) Example-Based Realistic Terrain Generation. In: Pan Z, Cheok A, Haller M, Lau R, Saito H, Liang R (eds) Advances in Artificial Reality and Tele-Existence, Lecture Notes in Computer Science, vol 4282, Springer Berlin / Heidelberg, pp 811–818, URL http://dx.doi.org/10.1007/11941354_84, 10.1007/11941354_84

Li S, Liu X, Wu E (2003) Feature-based visibility-driven CLOD for terrain. In: Pacific Graphics, IEEE Computer Society Press, Los Alamitos, CA, p 313–322

Lindenmayer A (1968) Mathematical models for cellular interaction in development. Journal of Theoretical Biology

Lindenmayer A (1974) Adding continous components to L-systems. L-systems Lecture Notes in Computer Science, Springer-Verlag

Loftus T (2011) Top video games may soon cost more. Website (accessed on Oct. 2011), http://www.msnbc.msn.com/id/3078404/ns/technology_and_science-games/t/top-video-games-may-soon-cost-more/

Losasso F, Hoppe H (2004) Geometry clipmaps: terrain rendering using nested regular grids. ACM Trans Graph 23(3):769–776, DOI http://doi.acm.org/10.1145/1015706.1015799

Lucas SM, Kendall G (2006) Evolutionary computation and games. IEEE Computational Intelligence Magazine

Luke S, Panait L (2002) Lexicographic parsimony pressure. In: et al WBL (ed) GECCO-2002: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kauffman, pp 829–836

Machado P, Cardoso A (2000) NEvAr - the assessment of an evolutionary art tool. In: Wiggins G (ed) Proceedings of the AISB'00 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science 2000, Birmingham, UK, URL citeseer.ist.psu.edu/machado00nevar.html

Machado P, Romero J, Cardoso A, Santos A (2005) Partially interactive evolutionary artists. New Gen Comput 23

Mandelbrot BB (1983) The Fractal Geometry of Nature. W. H. Freeman

Martin A, Lim A, Colton S, Browne C (2010) Evolving 3D Buildings for the Prototype Video Game Subversion. In: EvoGames Workshop, pp 111–120, URL http://pubs.doc.ic.ac.uk/evo-3d-buildings/

Mastin G, Watterberg P, Mareda J (1987) Fourier synthesis of ocean scenes. IEEE Computer Graphics and Applications 7:16–23, DOI http://doi.ieeecomputersociety.org/10.1109/MCG.1987.276961

Mayo MJ (2007) Games for science and engineering education. Commun ACM 50(7):30–35, DOI http://doi.acm.org/10.1145/1272516.1272536

Meloni W (2010) THE BRIEF - 2009 Ups and Downs. M2 Research website (accessed Oct. 2011), http://www.m2research.com/the-brief-2009-ups-and-downs.htm

Miikkulainen R, Bryant B, Cornelius R, Karpov I, Stanley K, Yong C (2006) Computational intelligence in games. In: Computational intelligence: Principles and practice, Piscataway, NJ: IEEE Computational Intelligence Society, pp 155–191

Miller GSP (1986) The definition and rendering of terrain maps. In: SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, pp 39–48, DOI \url{http://doi.acm.org/10.1145/15922.15890}

Musgrave FK, Kolb CE, Mace RS (1989) The synthesis and rendering of eroded fractal terrains. In: SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques, ACM, NY, USA, pp 41–50, DOI http://doi.acm.org/10.1145/74333.74337

Nelson MJ, Mateas M (2007) Towards automated game design. In: AI*IA '07: Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence on AI*IA 2007, Springer-Verlag, Berlin, Heidelberg, pp 626–637, DOI http://dx.doi.org/10.1007/978-3-540-74782-6_54

Olsen J (2004) 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

Ong TJ, Saunders R, Keyser J, Leggett JJ (2005) Terrain generation using genetic algorithms. In: GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, NY, USA, pp 1463–1470, DOI http://doi.acm.org/10.1145/1068009.1068241

Pabst J, Jense H (1995) Dynamic terrain generation based on multifractal techniques. In: Proceedings of the International Workshop on High Performance Computing for Computer Graphics and Visualisation, Swansea

Pajarola R, Antonijuan M, Lario R (2002) Quadtin: Quadtree based triangulated irregular networks. In: Proceedings IEEE Visualization, IEEE Computer Society Press, p 395–402, URL citeseer.ist.psu.edu/pajarola02quadtin.html

Pedersen C, Togelius J, Yannakakis G (2009) Modeling player experience in super mario bros. In: Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on, IEEE, pp 132–139

Peitgen HO, Jürgens H, Saupe D (2004) Chaos and Fractals - New Frontiers of Science, 2nd edn. Springer

Pereira F, Mendes M, Gervás P, Cardoso A (2005) Experiments with assessment of creative systems: an application of ritchie's criteria. In: Proceedings of the Workshop on Computational Creativity, 19th International Joint Conference on Artificial Intelligence, vol 5, p 05

Perlin K (1985) An image synthesizer. SIGGRAPH Comput Graph 19(3):287–296, DOI http://doi.acm.org/10.1145/325165.325247

Perlin K (2002) Improving noise. In: SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM, New York, NY, USA, pp 681–682, DOI http://doi.acm.org/10.1145/566570.566636

Pi X, Song J, Zeng L, Li S (2006) Procedural terrain detail based on patch-LOD algorithm. In: Pan Z, Aylett R, Diener H, Jin X, Göbel S, Li L (eds) Technologies for E-Learning and Digital Entertainment, Lecture Notes in Computer Science, vol 3942, Springer Berlin / Heidelberg, pp 913–920, URL http://dx.doi.org/10.1007/11736639_111, 10.1007/11736639_111

Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Lulu.com, URL http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)

Pouderoux J, Gonzato JC, Tobor I, Guitton P (2004) Adaptive hierarchical RBF interpolation for creating smooth digital elevation models. In: GIS '04 - 12th annual ACM international workshop on Geographic information systems, ACM, New York, NY, USA, pp 232–240, DOI http://doi.acm.org/10.1145/1032222.1032256

Prusinkiewicz P, Lindenmayer A (2004) The Algorithmic Beauty of Plants. Springer-Verlag

Rabin S (2002) AI game programming wisdom. Charles River Media

Rechenberg I (1971) Evolutionsstrategie: Optimierung technischer systeme nach prinzipien der biologischen evolution. PhD thesis, Technical University of Berlin, Department of Process Engineering

Remo C (2008) MIGS: Far Cry 2's Guay On The Importance Of Procedural Content. Website (accessed on Sep. 2011), http://www.gamasutra.com/php-bin/news_index.php?story=21165

Ritchie G (2007) Some empirical criteria for attributing creativity to a computer program. Minds Mach 17(1):67–99, DOI 10.1007/s11023-007-9066-2, URL http://dx.doi.org/10.1007/s11023-007-9066-2

Rodrigues N, Frade M, Fernández de Vega F (2010) Development of chapas an open source video game with genetic terrain programming. In: VII Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB), Valencia, Spain,, pp 1–8

Rosmarin R (2006) Why gears of war costs $60. Website (accessed on Oct. 2011), http://www.forbes.com/2006/12/19/ps3-xbox360-costs-tech-cx_rr_game06%_1219expensivegames.html

Sakas G (1993) Modeling and animating turbulent gaseous phenomena using spectral synthesis. The Visual Computer: International Journal of Computer Graphics 9(4):200–212, DOI \url{http://dx.doi.org/10.1007/BF01901724}

Sampath D (2004) ABRCon, Adaptive oBject Re-CONfiguration: an approach to enhance, repeat playability of games and repeat watchability of movies. In: ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology, ACM, New York, NY, USA, pp 313–316, DOI http://doi.acm.org/10.1145/1067343.1067388

Schneider J, Boldte T, Westermann R (2006) Real-time editing, synthesis, and rendering of infinite landscapes on GPUs. In: Vision, modeling, and

visualization 2006: proceedings, November 22-24, 2006, Aachen, Germany, IOS Press, p 145

Schwefel HP (1977) Numerische Optimierung von Computer-Modellen. Birkhäuser Basel

Sims K (1991) Artificial evolution for computer graphics. In: SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques, ACM, NY, USA, pp 319–328, DOI http://doi.acm.org/10.1145/122718.122752

Sims K (1992) Interactive evolution of dynamical systems. In: Varela F, Bourgine P (eds) Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life, MIT Press, Paris, FR, pp 171–178

Smelik RM, Tutenel T, de Kraker KJ, Bidarra R (2010) Declarative terrain modeling for military training games. International Journal of Computer Games Technology 2010:11, DOI 10.1155/2010/360458, article ID 360458

Soddu C (2003) Visionary aesthetics and architecture variations. In: Generative Art

Sorenson N, Pasquier P (2010) Towards a generic framework for automated video game level creation. Applications of Evolutionary Computation 6024:131–140

Squire K (2003) Video games in education. Int J Intell Games & Simulation 2(1):49–62

Stachniak S, Stuerzlinger W (2005) An algorithm for automated fractal terrain deformation. Computer Graphics and Artificial Intelligence 1:64–76

Szeliski R, Terzopoulos D (1989) From splines to fractals. SIGGRAPH Comput Graph 23(3):51–60, DOI http://doi.acm.org/10.1145/74334.74338

Takagi H (2001) Interactive evolutionary computation: Fusion of the capabilities of ec optimization and human evaluation. in Proceedings of the IEEE 89(9):1275–1296

Togelius J, De Nardi R, Lucas S (2006) Making racing fun through player modeling and track evolution. In: in Proceedings of the SAB06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games, Citeseer

Togelius J, Nardi RD, Lucas SM (2007) Towards automatic personalised content creation in racing games. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games

Togelius J, Preuss M, Beume N, Wessing S, Hagelback J, Yannakakis G (2010a) Multiobjective exploration of the starcraft map space. In: Computational Intelligence and Games (CIG), 2010 IEEE Symposium on, IEEE, pp 265–272

Togelius J, Preuss M, Yannakakis GN (2010b) Towards multiobjective procedural map generation. In: PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, ACM, New York, NY, USA, pp 1–8, DOI http://doi.acm.org/10.1145/1814256.1814259

Togelius J, Yannakakis GN, Stanley KO, Browne C (2011) Search-based procedural content generation: A taxonomy and survey. Computational Intelligence and AI in Games, IEEE Transactions on 3(3):172 –186, DOI 10.1109/TCIAIG.2011.2148116

Traxler C, Gervautz M (1996) Using genetic algorithms to improve the visual quality of fractal plants generated with csg-pl-systems. In: Proceedings of the Fourth International Conference in Central Europe on Computer Graphics and Virtual Worlds, vol 2, pp 367–376

Tu SC, Huang CY, Tai WK (2008) Terrain synthesis based on microscopic terrain feature. In: Pan Z, Zhang X, El Rhalibi A, Woo W, Li Y (eds) Technologies for E-Learning and Digital Entertainment, Lecture Notes in

Computer Science, vol 5093, Springer Berlin / Heidelberg, pp 644–655, URL http://dx.doi.org/10.1007/978-3-540-69736-7_69

Unemi T (1998) A design of multi-field user interface for simulated breeding. In: Proceedings of the Third Asian Fuzzy and Intelligent System Symposium, Masan, Korea, pp 489–494

Unemi T (1999) 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, IEEE, Adelaide, Australia, pp 288–291, URL ftp://ftp.t.soka.ac.jp/users/unemi/papers/KES99.pdf

Unemi T (2000) SBART 2.4: an IEC tool for creating 2D images, movies, and collage. In: Proceedings of 2000 Genetic and Evolutionary Computational Conference, NV, USA, p 153, URL http://citeseer.ist.psu.edu/369752.html

Vemuri B, Mandal C, Lai SH (1997) A fast gibbs sampler for synthesizing constrained fractals. Visualization and Computer Graphics, IEEE Transactions on 3(4):337–351, DOI 10.1109/2945.646237

Virtual Terrain Project (2009) Why virtual terrain? Website (accessed on Oct. 2011), http://www.vterrain.org/Misc/Why.html

Voss R (1987) Fractals in nature: characterization, measurement, and simulation. SIGGRAPH

Zhou H, Sun J, Turk G, Rehg JM (2007) Terrain synthesis from digital elevation models. IEEE Transactions on Visualization and Computer Graphics 13(4):834–848, DOI http://dx.doi.org/10.1109/TVCG.2007.1027, member-Turk, Greg and Member-Rehg, James M.

Appendix A

Terrain Programmes

List of all Terrain Programmes (TPs) from the images presented in this thesis.

A.1 Interactive GTP

Figure 4.5 (left)
TP = 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)))))))

Figure 4.5 (center)
TP = myLog(myLog(myLog(myLog(myLog(myLog(fftGen(3.00)))))))

Figure 4.5 (right)
TP = myLog(sin(mySqrt(smooth(fftGen(1.25)))))

Figure 4.6 (left)
TP = myMod(smooth(smooth(fftGen(0.50))) , smooth(plane(5)))

Figure 4.6 (center)
TP = myLog(minus(fftGen(2.75) , myLog(minus(smooth(fftGen(1.50)) , fftGen(2.50)))))

Figure 4.6 (right)
TP = times(sin(fftGen(3.00)) , smooth(times(sin(cos(sin(cos(times(fftGen(1.75) , fftGen(0.75))))))) , fftGen(0.50))))

A.2 Automated GTP

Terrains with a single metric

Figure 5.21 (left)

TP = sin(plus(myDivide(tan(myNoise(X, Y)) , plus(8.52490, myNoise(X, Y))) , myDivide(mySqrt(mySqrt(mySqrt(myDivide(myNoise(X, Y) , plus(exp(sin(myPower(negative(tan(exp(mySqrt(minus(myNoise(X, Y) , 8.70770))))) , myLog(0.94715)))) , minus(myNoise(X, Y) , 9.69240))))) , minus(8.37586, sin(sin(sin(myDivide(2.81278, myNoise(X, Y)))))))))

Figure 5.21 (right)

TP = plus(cos(multiply(myDivide(cos(tan(7.30094)) , myPower(exp(sin(myNoise(X, Y))) , cos(myNoise(X, Y)))) , minus(mySqrt(myNoise(X, Y)) , minus(mySqrt(myNoise(X, Y)) , cos(multiply(cos(myNoise(X, Y)) , myDivide(myNoise(X, Y) , cos(myLog(minus(myNoise(X, Y) , 9.59242)))))))))) , negative(sin(myPower(myLog(tan(myNoise(X, Y))) , multiply(atan(0.42628) , negative(myNoise(X, Y)))))))

Figure 5.22 (left)

TP = myPower(myDivide(minus(atan(myDivide(exp(myPower(0.51531, Y)) , minus(tan(minus(X, Y)) , myDivide(plus(myDivide(mySqrt(X) , myLog(X)) , multiply(exp(myPower(0.51531, Y)) , X)) , cos(Y))))) , negative(3.83396)) , minus(atan(sin(Y)) , sin(myDivide(plus(negative(Y) , plus(minus(myLog(X) , minus(5.09638, Y)) , sin(minus(6.12560, X)))) , myDivide(sin(myPower(sin(mySqrt(Y)) , minus(6.23737, sin(myPower(negative(sin(exp(X))) , X))))) , myPower(Y, Y)))))) , plus(negative(Y) , plus(minus(myLog(X) , minus(5.09638, Y)) , sin(cos(plus(multiply(X, X) , negative(3.83396)))))))

Figure 5.22 (right)

TP = myPower(myDivide(cos(multiply(Y, X)) , cos(minus(multiply(atan(myPower(tan(mySqrt(myPower(Y, sin(cos(minus(X, Y))))))) , mySqrt(Y))) , exp(myPower(atan(Y) , atan(5.90479)))) , multiply(myPower(X, mySqrt(Y)) , exp(minus(myLog(sin(cos(exp(X))))) , multiply(8.29732, X))))))) , sin(multiply(atan(X) , myDivide(0.09512, 5.29683))))

Figure 5.23 (left)

TP = myPower(myDivide(exp(cos(cos(cos(myNoise(X, Y))))) , exp(cos(cos(X)))) , cos(sin(myPower(Y, myNoise(X, Y)))))

Figure 5.23 (right)

TP = mySqrt(myPower(atan(tan(exp(multiply(myNoise(X, Y) , myNoise(X, Y)))))) , cos(minus(myDivide(atan(myNoise(X, Y)) , plus(X, Y)) , multiply(multiply(minus(tan(cos(X)) , exp(multiply(minus(negative(4.97263) , myPower(atan(tan(exp(multiply(myNoise(X, Y) , myNoise(X, Y))))) , cos(myDivide(myPower(myNoise(X, Y) , Y) , sin(multiply(myNoise(X, Y) , myNoise(X, Y))))))) , atan(sin(multiply(myNoise(X, Y) , myNoise(X, Y)))))) , multiply(myNoise(X, Y) , myNoise(X, Y))) , exp(X))))))

Figure 5.24 (left)

TP = mySqrt(myDivide(myDivide(negative(myPower(myNoise(X, Y) , 6.31165)) , sin(sin(1.22076))) , myDivide(myLog(atan(tan(myNoise(X, Y)))) , myPower(mySqrt(exp(myNoise(X, Y))) , multiply(negative(negative(myPower(myNoise(X, Y) , 6.31165))) , cos(myNoise(X, Y)))))))

Figure 5.24 (right)

TP = plus(myDivide(myPower(cos(myNoise(X, Y)) , cos(4.40613)) , minus(atan(myNoise(X, Y)) , multiply(5.92375, 7.96716))) , myPower(multiply(myLog(myPower(cos(tan(multiply(myDivide(cos(myLog(minus(0.87295, cos(multiply(cos(myNoise(X, Y)) , myNoise(X, Y))))))) , cos(cos(myPower(myNoise(X, Y) , 8.64329)))) , exp(atan(myNoise(X, Y)))))) , multiply(exp(atan(myNoise(X, Y))) , cos(multiply(myPower(myNoise(X, Y) , 8.64329) , exp(2.72279)))))) , myNoise(X, Y)) , myPower(myNoise(X, Y) , 8.64329)))

Figure 5.25 (left)

TP = atan(plus(sin(atan(plus(cos(myLog(atan(tan(minus(mySqrt(myLog(myDivide(myPower(cos(minus(atan(5.23822) , mySqrt(X))) , mySqrt(cos(myDivide(X, Y)))) , plus(sin(myDivide(X, Y)) , atan(X))))) , atan(myLog(X))))))) , negative(5.75913)))) , myDivide(cos(myDivide(tan(Y) , minus(5.42187, 1.27538))) , plus(atan(sin(Y)) , multiply(myLog(X) , plus(Y, X))))))

Figure 5.25 (right)

TP = atan(atan(myDivide(mySqrt(Y) , multiply(Y, atan(multiply(myPower(atan(tan(cos(myLog(myDivide(plus(6.76558, X) , multiply(7.67766, myDivide(1.41844, myDivide(X, Y))))))))))) , minus(myDivide(4.92778, 5.87255) , multiply(cos(sin(myLog(sin(myLog(myDivide(8.23575, X))))))) , 0.71917))) , negative(exp(multiply(Y, X)))))))))

Figure 5.26 (left)

TP = cos(myPower(myPower(exp(myPower(myNoise(X, Y) , Y)) , sin(cos(cos(myPower(myPower(exp(myPower(myNoise(X, Y) , Y)) , atan(cos(myNoise(X, Y)))) , cos(myDivide(atan(exp(cos(4.23642))) , atan(exp(atan(cos(myNoise(X, Y)))))))))))) , cos(myDivide(mySqrt(cos(myNoise(X, Y))) , plus(Y, myPower(plus(plus(Y, myNoise(X, Y)) , exp(myPower(myNoise(X, Y) , Y))) , minus(minus(myNoise(X, Y) , Y) , plus(Y, myNoise(X, Y)))))))))

Figure 5.26 (right)

TP = exp(tan(multiply(sin(sin(myPower(myPower(mySqrt(exp(myLog(mySqrt(3.20202)))) , atan(sin(multiply(myLog(myPower(X, X)) , Y)))) , myPower(mySqrt(plus(plus(7.60085, 9.00599) , minus(Y, atan(sin(multiply(myLog(myPower(X, X)) , Y))))))) , atan(negative(myPower(X, X))))))) , atan(myDivide(myPower(myNoise(X, Y) , X) , multiply(myNoise(X, Y) , mySqrt(exp(Y))))))))

Terrains with both metrics

Figure 5.27 (left)

TP = myDivide(myNoise(X, Y) , plus(cos(plus(myDivide(myNoise(X, Y) , plus(cos(plus(plus(minus(sin(myPower(cos(tan(myNoise(X, Y)))) , myDivide(myPower(myNoise(X, Y) , 6.02195) , 0.50166))) , plus(myDivide(myNoise(X, Y) , myNoise(X, Y)) , plus(cos(myNoise(X, Y)) , cos(myNoise(X, Y)))))) , tan(sin(tan(myNoise(X, Y))))) , 6.02195)) , plus(cos(plus(minus(sin(myNoise(X, Y)) , plus(myDivide(myNoise(X, Y) , myNoise(X, Y)) , cos(cos(myNoise(X, Y)))))) , tan(sin(tan(myNoise(X, Y)))))) , 6.02195))) , cos(myPower(cos(myNoise(X, Y)) , cos(myNoise(X, Y)))))) , 6.02195))

Figure 5.27 (right)

TP = atan(myDivide(sin(multiply(mySqrt(myNoise(X, Y)) , plus(myDivide(sin(myNoise(X,

Y)) , 6.16952) , plus(myDivide(sin(myNoise(X, Y)) , 6.16952) , plus(mySqrt(1.69693) , cos(2.12672))))) , 6.16952))

Figure 5.28 (left)

TP = myPower(myPower(cos(myNoise(X, Y)) , cos(myNoise(X, Y))) , cos(myDivide(cos(myDivide(negative(myDivide(mySqrt(5.00842) , myPower(cos(sin(myNoise(X, Y))) , cos(myDivide(myDivide(cos(myDivide(negative(exp(myNoise(X, Y))) , sin(myPower(cos(sin(myNoise(X, Y))) , 2.86638)))) , 4.41543) , plus(cos(myNoise(X, Y)) , exp(cos(myDivide(tan(cos(sin(myNoise(X, Y)))) , multiply(myNoise(X, Y) , myNoise(X, Y)))))))))) , myDivide(myDivide(myNoise(X, Y) , 4.41543) , mySqrt(5.00842))) , sin(myNoise(X, Y)))))

Figure 5.28 (right)

TP = cos(multiply(sin(plus(myPower(myDivide(myNoise(X, Y) , atan(cos(cos(cos(7.68913))))) , atan(atan(myPower(sin(tan(sin(1.15769))) , myDivide(multiply(negative(cos(multiply(myNoise(X, Y) , 0.98480))) , cos(myDivide(myNoise(X, Y) , atan(cos(cos(cos(7.68913)))))))) , myDivide(cos(myNoise(X, Y)) , cos(atan(cos(sin(myNoise(X, Y)))))))))))) , myDivide(exp(exp(myNoise(X, Y))) , negative(cos(multiply(myNoise(X, Y) , 0.98480)))))) , mySqrt(cos(7.68913))))

Figure 5.29 (left)

TP = minus(atan(exp(atan(multiply(0.28298, mySqrt(myNoise(X, Y)))))) , multiply(myPower(myNoise(X, Y) , 5.21597) , atan(sin(minus(atan(sin(atan(cos(mySqrt(myNoise(X, Y)))))))) , multiply(myPower(atan(sin(cos(atan(cos(myPower(myNoise(X, Y) , 5.21597))))))) , 5.21597) , atan(myNoise(X, Y))))))))

Figure 5.29 (right)

TP = myPower(cos(cos(sin(myLog(cos(plus(negative(myLog(sin(sin(atan(plus(negative(myLog(atan(5.23015))) , 5.56839)))))) , multiply(5.56839, myDivide(myLog(sin(7.92654)) , mySqrt(plus(sin(myLog(cos(cos(myNoise(X, Y))))) , myNoise(X, Y))))))))))) , myPower(cos(cos(sin(myLog(cos(plus(negative(myLog(atan(5.23015))) , multiply(tan(myDivide(3.74998, myLog(cos(cos(myNoise(X, Y)))))) , myDivide(myLog(plus(negative(myLog(atan(5.23015))) , 5.56839)) , tan(myLog(5.56839))))))))) , cos(plus(negative(myLog(atan(cos(plus(negative(myLog(atan(5.23015))) , multiply(cos(plus(negative(myLog(atan(cos(sin(myNoise(X, Y))))))) , multiply(sin(atan(minus(myNoise(X, Y) , 6.60566))) , atan(myLog(atan(5.23015))))))) , myLog(mySqrt(cos(cos(myNoise(X, Y))))))))))) , multiply(sin(cos(mySqrt(plus(sin(myLog(cos(cos(negative(myLog(atan(5.23015))))))))) , 4.07902))) , negative(myLog(atan(cos(myNoise(X, Y))))))))))

Figure 5.30 (left)

TP = myDivide(atan(plus(minus(5.48054, tan(myLog(myDivide(myNoise(X, Y) , plus(multiply(3.27641, 0.82816) , myNoise(X, Y)))))) , cos(myNoise(X, Y)))) , plus(cos(plus(minus(myPower(myLog(myPower(myNoise(X, Y) , myDivide(minus(myPower(myLog(plus(2.86537, multiply(myNoise(X, Y) , myNoise(X, Y)))) , minus(myPower(myLog(exp(1.57031)) , mySqrt(negative(mySqrt(8.53152)))) , cos(multiply(myLog(6.21534) , exp(1.57031))))) , mySqrt(negative(sin(mySqrt(8.53152))))) , plus(cos(plus(negative(myDivide(9.71608, 5.10391)) , exp(1.57031))) , minus(myLog(myNoise(X, Y)) , 7.61175)))) , cos(multiply(myPower(2.25783, tan(mySqrt(myNoise(X, Y)))) , myNoise(X, Y)))) , cos(multiply(myPower(2.25783, myNoise(X, Y)) , exp(1.57031)))) , mySqrt(myNoise(X, Y)))) , minus(multiply(myLog(5.91455) , cos(plus(myLog(6.21534) , mySqrt(myNoise(X, Y))))) , 7.61175)))

Figure 5.30 (right)

TP = myDivide(mySqrt(minus(sin(mySqrt(1.01158)) , tan(negative(multiply(myPower(minus(minus(atan(5.70012) , atan(sin(myLog(mySqrt(9.83085))))) , myPower(myDivide(multiply(atan(myNoise(X, Y)) , cos(7.63158)) , myPower(minus(atan(myPower(plus(myLog(3.30988) , 8.27336) , tan(7.85065))) , sin(exp(myDivide(myLog(3.30988) , minus(7.43497, 5.63637))))) , myLog(mySqrt(9.83085)))) , myPower(negative(mySqrt(2.59476)) , myNoise(X, Y)))) , negative(negative(multiply(myNoise(X, Y) , myNoise(X, Y))))) , myLog(sin(exp(cos(myNoise(X, Y)))))))))) , minus(myDivide(mySqrt(plus(5.70012, myNoise(X, Y))) , multiply(myPower(myNoise(X, Y) , myNoise(X, Y)) , cos(7.53382))) , plus(cos(atan(negative(tan(negative(mySqrt(myNoise(X, Y)))))))) , tan(sin(9.97728)))))

Figure 5.31 (left)
TP = myPower(sin(minus(myDivide(plus(Y, minus(minus(Y, X) , plus(Y, minus(minus(Y, X) , multiply(Y, X))))) , myDivide(plus(8.00660, cos(cos(Y))) , multiply(minus(plus(negative(Y) , cos(X)) , myPower(myDivide(myDivide(tan(X) , Y) , X) , mySqrt(atan(8.00660)))) , cos(cos(myLog(cos(X))))))) , mySqrt(myDivide(cos(X) , myPower(sin(minus(myDivide(plus(Y, minus(minus(Y, X) , myDivide(Y, X))) , myDivide(plus(myDivide(myPower(atan(cos(X)) , myDivide(0.09512, 5.29683)) , multiply(minus(plus(negative(Y) , cos(X)) , myPower(minus(minus(Y, X) , multiply(Y, X)) , mySqrt(atan(8.00660)))) , cos(cos(myLog(atan(8.00660)))))) , cos(atan(X))) , minus(myLog(Y) , cos(2.65267)))) , mySqrt(minus(Y, X)))) , multiply(atan(X) , myDivide(0.09512, 5.29683)))))) , multiply(atan(X) , myDivide(0.09512, 5.29683)))

Figure 5.31 (right)
TP = minus(negative(myDivide(sin(myDivide(Y, negative(exp(atan(negative(myDivide(multiply(myPower(mySqrt(negative(myPower(minus(Y, X) , cos(X)))) , myDivide(cos(tan(Y)) , tan(Y))) , Y) , X))))))) , plus(exp(3.61854) , cos(tan(Y))))) , exp(atan(negative(plus(cos(Y) , Y)))))

Figure 5.32 (left)
TP = plus(atan(atan(sin(myPower(plus(minus(plus(mySqrt(Y) , cos(cos(mySqrt(mySqrt(myPower(negative(X) , negative(Y))))))) , cos(minus(minus(myLog(cos(mySqrt(myPower(negative(myPower(plus(2.93753, X) , sin(3.70603))) , negative(Y))))) , mySqrt(myPower(negative(X) , negative(Y)))) , myPower(sin(plus(negative(exp(X)) , myPower(negative(X) , negative(Y)))) , negative(cos(tan(Y))))))) , myLog(myPower(3.70603, 0.54553))) , multiply(sin(X) , atan(myLog(1.12916))))))) , sin(myPower(plus(minus(tan(Y) , myLog(sin(cos(minus(myLog(cos(mySqrt(myPower(negative(myPower(plus(2.93753, X) , sin(3.70603))) , negative(Y))))) , mySqrt(myPower(negative(X) , negative(Y))))))))) , myLog(sin(X))) , multiply(sin(X) , atan(myLog(1.12916))))))

Figure 5.32 (right)
TP = sin(myPower(cos(sin(minus(negative(multiply(1.73798, X)) , minus(X, Y)))) , cos(plus(cos(sin(minus(sin(myDivide(cos(atan(sin(cos(atan(sin(minus(X, Y)))))))) , sin(cos(negative(plus(minus(Y, minus(minus(X, Y) , plus(Y, X))) , atan(Y)))))))) , sin(negative(X))))) , myPower(atan(Y) , myDivide(sin(tan(7.80716)) , X))))))

Figure 5.33 (left)
TP = mySqrt(multiply(sin(minus(myPower(myPower(cos(myPower(tan(X) , myLog(negative(Y))))) , sin(myPower(myDivide(minus(plus(X, negative(X)) , negative(myLog(negative(Y)))) , minus(atan(sin(Y)) , cos(mySqrt(X)))) , plus(negative(Y) , sin(myPower(X, atan(minus(X, 7.03932)))))))))) , minus(cos(minus(sin(myPower(myPower(minus(1.79686, Y) , sin(tan(exp(minus(7.09360, X)))))) , sin(tan(exp(Y))))) , myDivide(Y, X))) , X)) , minus(1.79686,

Y))) , myLog(cos(sin(9.61643)))))

Figure 5.33 (right)

TP = cos(minus(sin(multiply(mySqrt(atan(cos(mySqrt(mySqrt(sin(atan(sin(sin(myDivide(mySqrt(Y) , cos(Y))))))))))) , multiply(mySqrt(sin(atan(sin(sin(myDivide(mySqrt(multiply(tan(multiply(X, 8.22930)) , myDivide(minus(X, 6.88531) , exp(8.46101)))) , exp(8.46101))))))) , atan(myPower(negative(mySqrt(sin(atan(sin(sin(myDivide(mySqrt(Y) , cos(myPower(Y, sin(X)))))))))) , X))))) , sin(atan(cos(mySqrt(mySqrt(sin(atan(sin(sin(myDivide(mySqrt(Y) , cos(myPower(Y, sin(X)))))))))))))))

Figure 5.34 (left)

TP = cos(minus(plus(myPower(cos(cos(cos(multiply(multiply(Y, Y) , sin(X))))) , mySqrt(cos(myPower(mySqrt(cos(myPower(mySqrt(negative(multiply(Y, Y))) , mySqrt(atan(multiply(cos(myDivide(minus(myDivide(Y, X) , 6.71752) , atan(Y))) , sin(Y)))))))) , mySqrt(myPower(cos(myPower(negative(tan(tan(sin(myDivide(X, Y)))))) , multiply(cos(myDivide(minus(exp(Y) , 6.71752) , atan(Y))) , myDivide(Y, 4.05453)))) , mySqrt(Y)))))))) , tan(tan(sin(6.63816)))) , cos(negative(cos(multiply(Y, atan(exp(1.56833))))))))

Figure 5.34 (right)

TP = myPower(myDivide(cos(multiply(Y, X)) , cos(myLog(minus(minus(negative(myDivide(0.09512, multiply(Y, X)))) , atan(sin(myDivide(multiply(sin(minus(mySqrt(Y) , multiply(5.21306, X)))) , minus(plus(5.86645, 8.48695) , multiply(minus(plus(negative(Y) , cos(X)) , myPower(atan(sin(9.17727)) , sin(tan(exp(2.83181))))) , cos(cos(exp(X)))))) , myDivide(minus(9.10819, 2.63591) , cos(9.67310)))))) , multiply(minus(plus(sin(X) , cos(myDivide(Y, X))) , myPower(cos(X) , myLog(4.59794))) , cos(myPower(X, cos(multiply(mySqrt(minus(tan(cos(myDivide(0.09512, multiply(Y, X)))) , myPower(myDivide(Y, X) , myDivide(sin(negative(Y)) , myDivide(minus(9.10819, 2.63591) , myLog(Y)))))) , sin(cos(minus(X, Y))))))))))) , sin(multiply(atan(X) , myDivide(0.09512, 5.29683))))

Figure 5.35 (left)

TP = cos(atan(myPower(minus(multiply(atan(negative(negative(atan(atan(myNoise(X, Y)))))))) , sin(atan(atan(negative(atan(atan(atan(myNoise(X, Y))))))))) , mySqrt(atan(atan(atan(negative(atan(myNoise(X, Y))))))) , multiply(atan(mySqrt(atan(atan(atan(negative(negative(atan(atan(myNoise(X, Y)))))))))) , sin(X)))))

Figure 5.35 (right)

TP = mySqrt(plus(myDivide(multiply(atan(myNoise(X, Y)) , plus(7.27156, Y)) , sin(myNoise(X, Y))) , cos(atan(myLog(myPower(sin(myPower(sin(atan(myLog(mySqrt(myDivide(myDivide(myNoise(X, Y) , myNoise(X, Y)) , cos(sin(myNoise(X, Y))))))))) , cos(atan(myNoise(X, Y))))) , cos(myNoise(X, Y)))))))

Figure 5.36 (left)

TP = multiply(multiply(multiply(0.21532, cos(cos(multiply(Y, myDivide(X, myDivide(multiply(multiply(multiply(0.21532, myNoise(X, Y)) , multiply(multiply(multiply(0.21532, myNoise(X, Y)) , sin(myDivide(X, myDivide(multiply(0.39632, myNoise(X, Y)) , 5.81002)))) , sin(myDivide(X, myDivide(myDivide(0.21532, 5.81002) , cos(myNoise(X, Y))))))) , myDivide(X, myDivide(myDivide(multiply(0.21532, plus(sin(X) , multiply(0.39632, multiply(0.21532, myNoise(X, Y))))) , 5.81002) , sin(X)))) , 5.81002))))) , myNoise(X, Y)) , sin(myDivide(myDivide(X, myDivide(myDivide(multiply(0.21532, plus(sin(X) , multiply(0.39632, multiply(0.21532, myNoise(X, Y))))) , 5.81002) , cos(myNoise(X, Y)))) , myNoise(X, Y))))

Figure 5.36 (right)

TP = myDivide(sin(multiply(mySqrt(myNoise(X, Y)) , atan(myPower(multiply(sin(myDivide(multiply(negative(cos(plus(multiply(mySqrt(mySqrt(myNoise(X, Y))) , mySqrt(myNoise(X, Y))) , 6.31979))) , cos(cos(sin(multiply(mySqrt(myNoise(X, Y)) , mySqrt(myNoise(X, Y))))))) , 3.77733)) , cos(myDivide(mySqrt(exp(myNoise(X, Y))) , negative(exp(myNoise(X, Y))))))) , myPower(myNoise(X, Y) , X)))) , 3.77733)

Figure 5.37 (left)

TP = mySqrt(cos(atan(multiply(sin(mySqrt(mySqrt(multiply(mySqrt(myNoise(X, Y)) , atan(myPower(myPower(myNoise(X, Y) , plus(Y, 6.31979)) , myPower(myNoise(X, Y) , exp(myPower(cos(myNoise(X, Y)) , myNoise(X, Y)))))))))) , minus(myNoise(X, Y) , atan(myDivide(myDivide(7.17759, Y) , atan(minus(atan(minus(negative(exp(myPower(cos(myDivide(myNoise(X, Y) , 3.77733)) , mySqrt(myNoise(X, Y))))) , myPower(myPower(myNoise(X, Y) , plus(Y, cos(myNoise(X, Y)))) , myPower(myNoise(X, Y) , exp(myPower(myPower(myNoise(X, Y) , myNoise(X, Y)) , myNoise(X, Y))))))) , myPower(myNoise(X, Y) , myNoise(X, Y))))))))))))

Figure 5.37 (right)

TP = myDivide(sin(mySqrt(mySqrt(sin(exp(minus(plus(myPower(multiply(cos(tan(myNoise(X, Y))) , mySqrt(X)) , sin(myPower(multiply(multiply(mySqrt(multiply(cos(myNoise(X, Y)) , X)) , X) , multiply(cos(5.94226) , cos(5.94226))) , multiply(cos(myNoise(X, Y)) , mySqrt(multiply(cos(myNoise(X, Y)) , X))))))) , multiply(cos(exp(myNoise(X, Y))) , X)) , exp(myNoise(X, Y)))))))) , exp(plus(myPower(multiply(multiply(mySqrt(sin(exp(minus(plus(myPower(multiply(multiply(X, mySqrt(X)) , mySqrt(X)) , sin(plus(myPower(myNoise(X, Y) , sin(multiply(myNoise(X, Y) , X))) , mySqrt(X)))) , multiply(cos(tan(myNoise(X, Y))) , mySqrt(multiply(cos(myNoise(X, Y)) , X)))) , exp(myNoise(X, Y)))))) , X) , mySqrt(X)) , sin(multiply(myNoise(X, Y) , X))) , multiply(cos(5.94226) , mySqrt(multiply(cos(myNoise(X, Y)) , X)))))))

Figure 5.38 (left)

TP = mySqrt(myPower(atan(tan(exp(multiply(myLog(cos(myDivide(6.93162, 8.67883))) , myNoise(X, Y)))))) , cos(minus(myDivide(atan(myNoise(X, Y)) , minus(minus(atan(multiply(Y, multiply(Y, myNoise(X, Y))))) , mySqrt(cos(minus(myDivide(atan(negative(myNoise(X, Y))) , plus(minus(multiply(minus(2.56689, tan(7.02797)) , minus(myDivide(X, X) , mySqrt(X))) , exp(mySqrt(myNoise(X, Y)))) , Y)) , multiply(myLog(cos(cos(tan(exp(multiply(myNoise(X, Y) , myNoise(X, Y))))))) , exp(X)))))) , myDivide(myPower(myLog(tan(myPower(sin(myNoise(X, Y)) , mySqrt(Y)))) , exp(negative(9.64544))) , myLog(atan(myLog(6.35588)))))) , multiply(multiply(minus(tan(X) , exp(mySqrt(myNoise(X, Y)))) , multiply(atan(multiply(myLog(cos(myDivide(6.93162, 8.67883))) , myNoise(X, Y))) , exp(cos(mySqrt(myNoise(X, Y)))))) , exp(X))))))

Figure 5.38 (right)

TP = myDivide(myPower(mySqrt(mySqrt(myPower(cos(multiply(myNoise(X, Y) , myNoise(X, Y))) , exp(2.88776)))) , cos(multiply(tan(myPower(cos(multiply(myNoise(X, Y) , myNoise(X, Y))) , exp(2.88776))) , sin(minus(minus(multiply(cos(tan(myNoise(X, Y)))) , mySqrt(multiply(mySqrt(myPower(cos(multiply(myNoise(X, Y) , myNoise(X, Y))) , exp(2.88776))) , X))) , exp(plus(multiply(myNoise(X, Y) , X) , multiply(cos(tan(myNoise(X, Y))) , mySqrt(myPower(cos(multiply(myNoise(X, Y) , myNoise(X, Y))) , exp(2.88776))))))) , mySqrt(mySqrt(mySqrt(atan(exp(plus(myPower(multiply(multiply(X, X) , cos(tan(myNoise(X, Y)))) , multiply(cos(tan(myNoise(X, Y))) , cos(X))) , multiply(cos(myNoise(X, Y)) , cos(X)

)))))))))))) , exp(plus(myPower(multiply(multiply(X, X) , mySqrt(X)) , sin(multiply(myNoise(X, Y) , X))) , multiply(cos(tan(myNoise(X, Y))) , mySqrt(X)))))

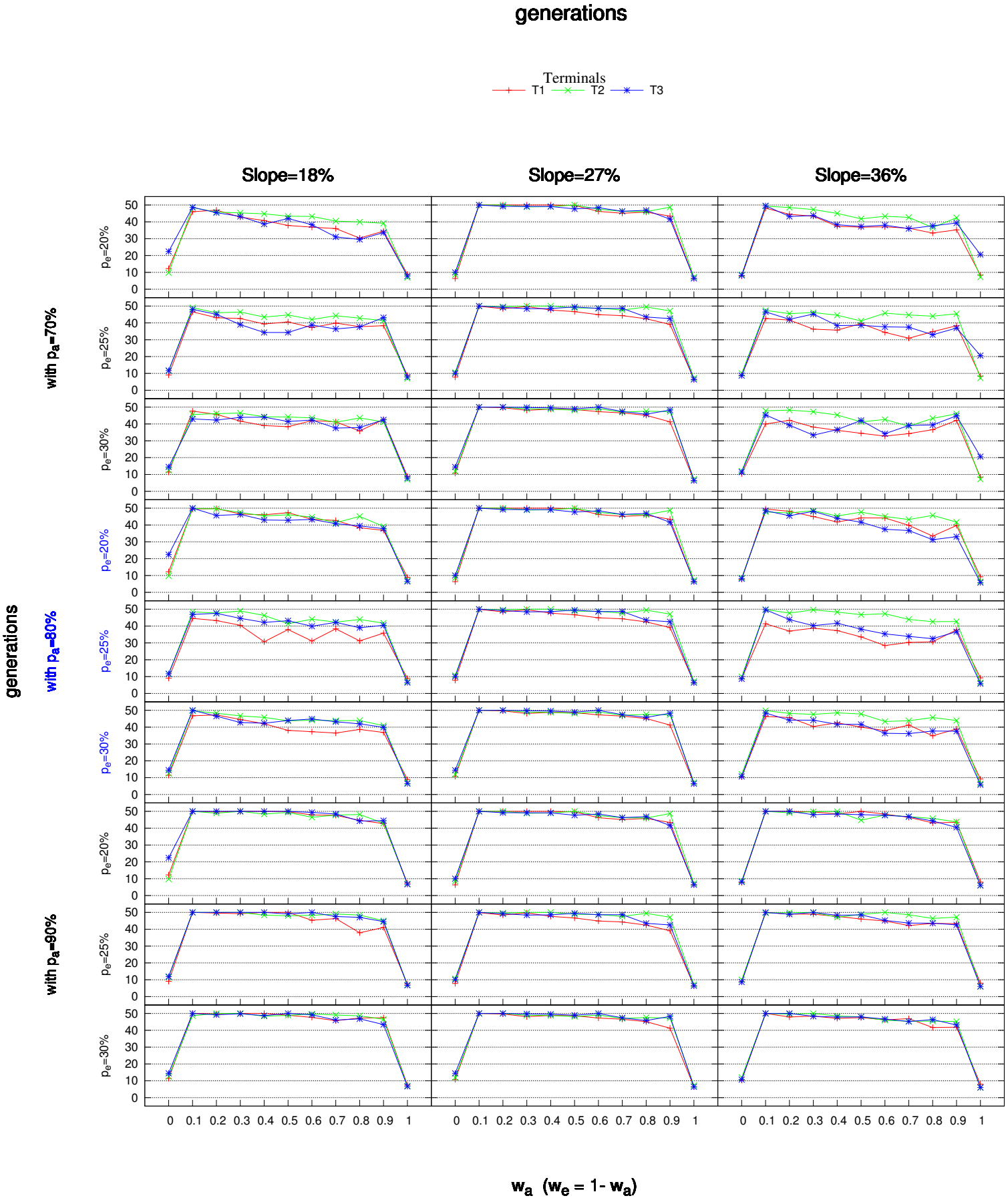
Terrain with Zoom

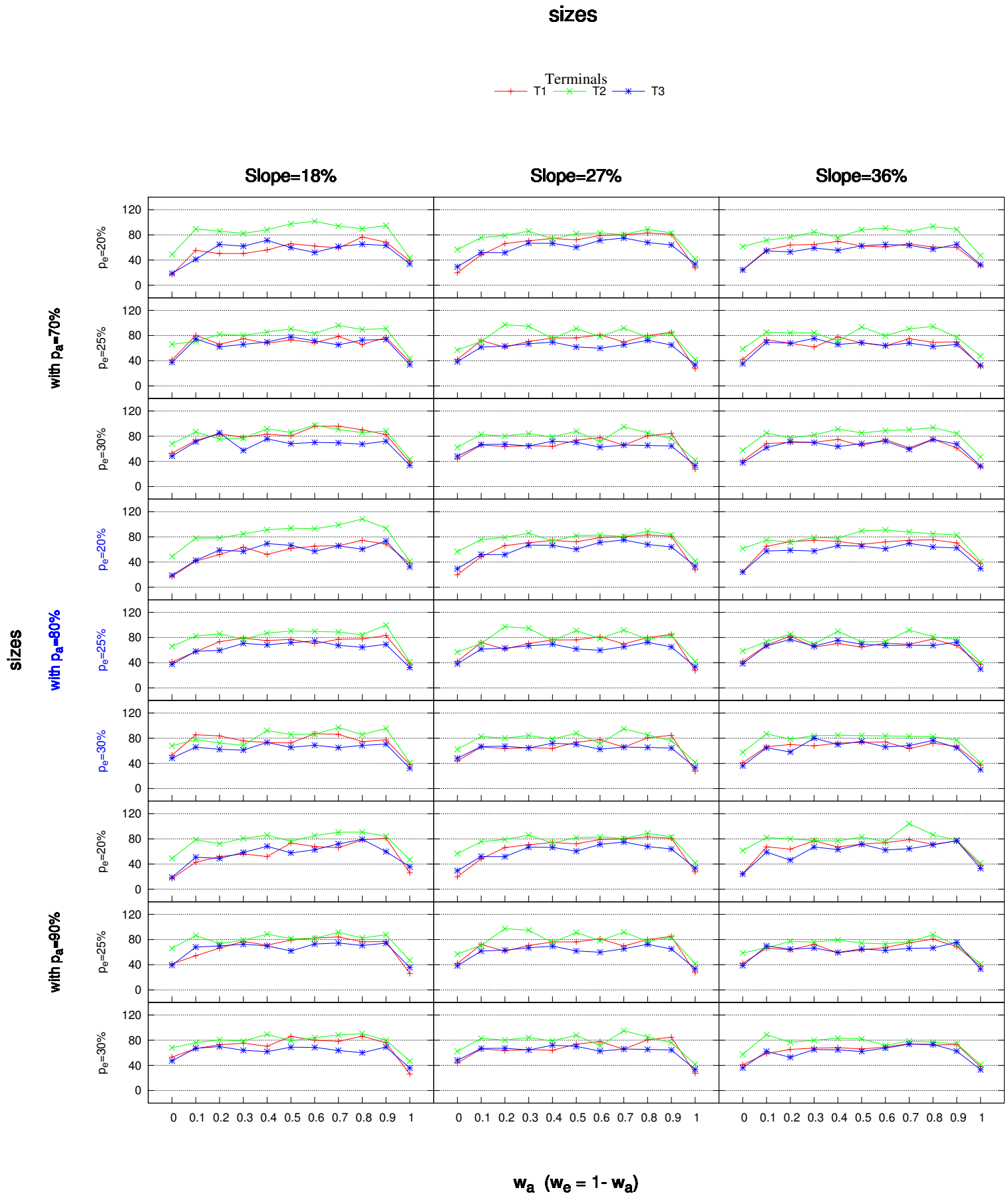
Figure 5.39

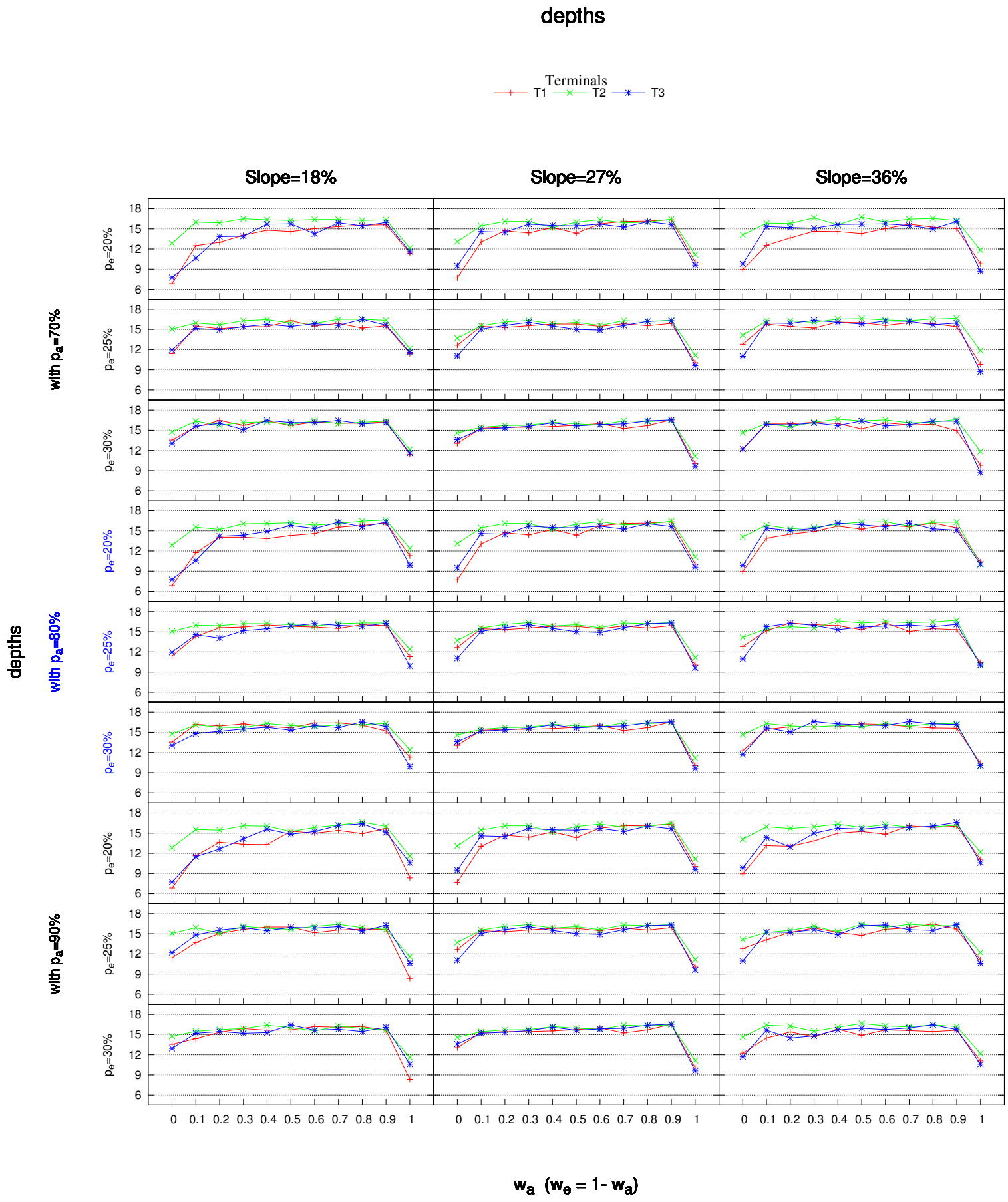
TP = plus(myDivide(tan(cos(multiply(sin(plus(myPower(myDivide(X, 0.78181) , atan(plus(Y, 4.45522))) , myPower(tan(X) , multiply(Y, cos(myPower(myLog(9.50991) , myLog(cos(myLog(myLog(cos(mySqrt(Y)))))))))))) , cos(myPower(mySqrt(myLog(2.96447)) , myLog(cos(cos(multiply(sin(plus(myDivide(X, 0.78181) , cos(myPower(mySqrt(myLog(2.96447)) , myLog(9.50991))))) , myLog(myLog(9.50991)))))))))) , negative(8.66469)) , mySqrt(myLog(mySqrt(myLog(2.96447)))))

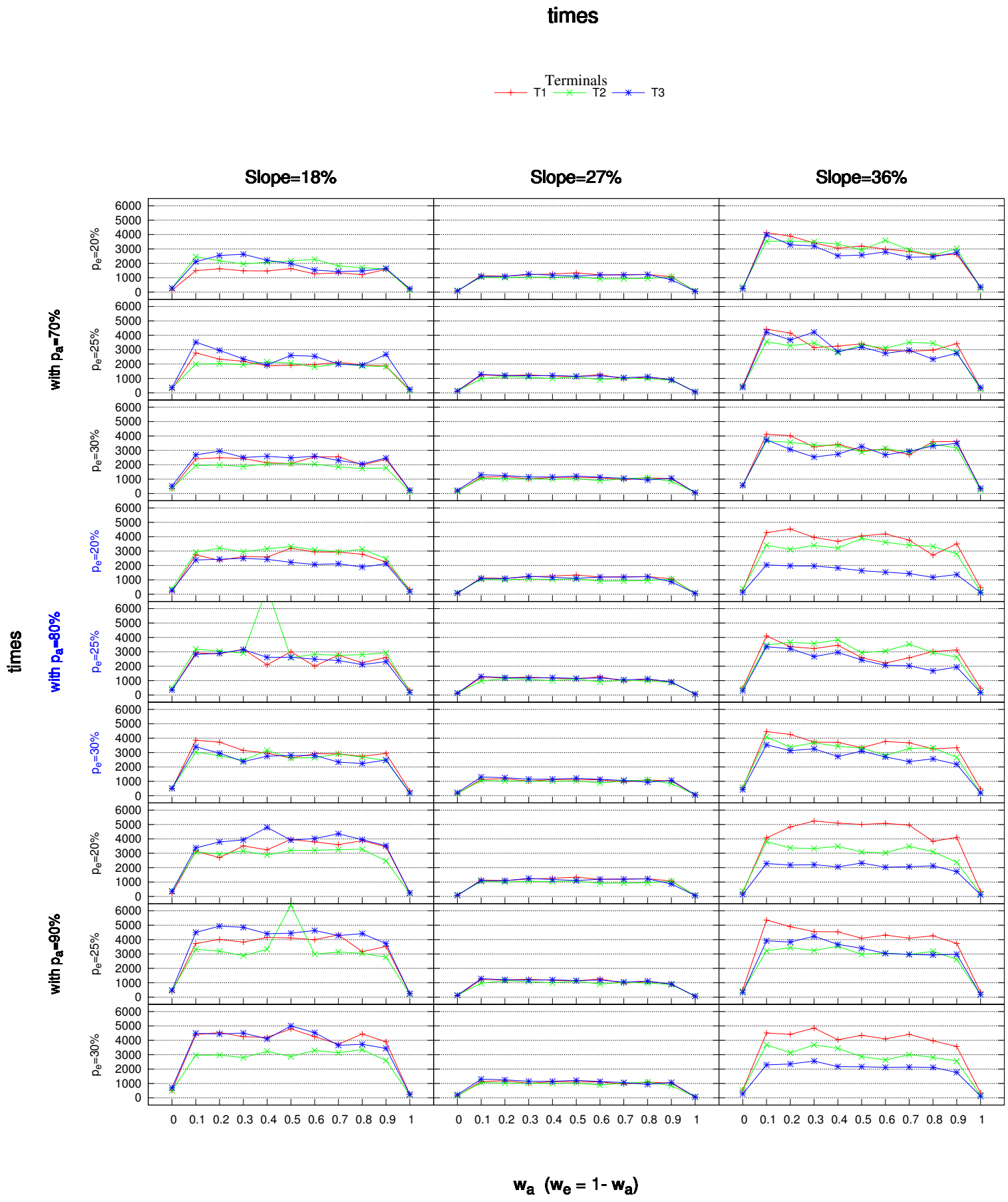
Appendix B

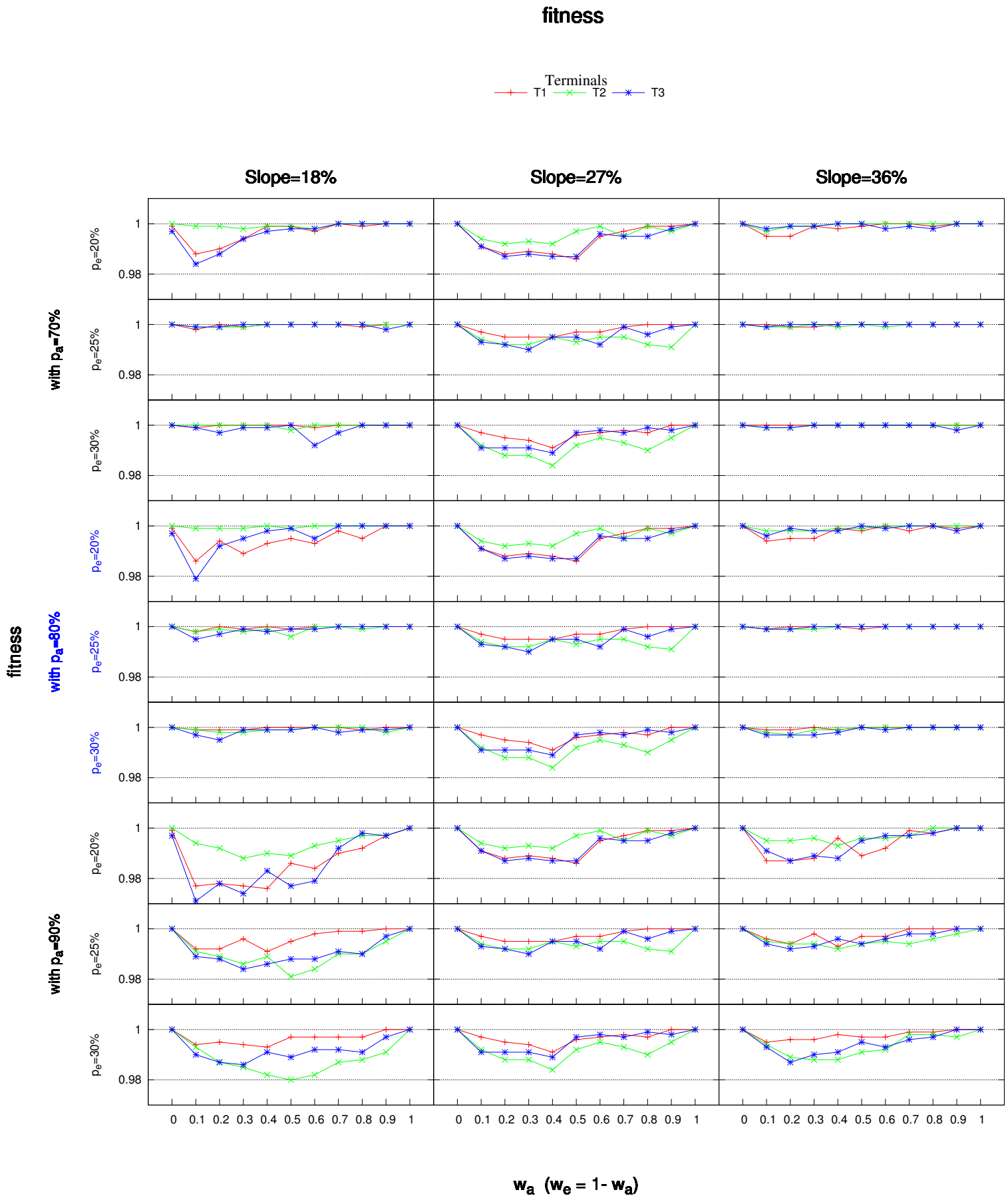
Additional Graphics

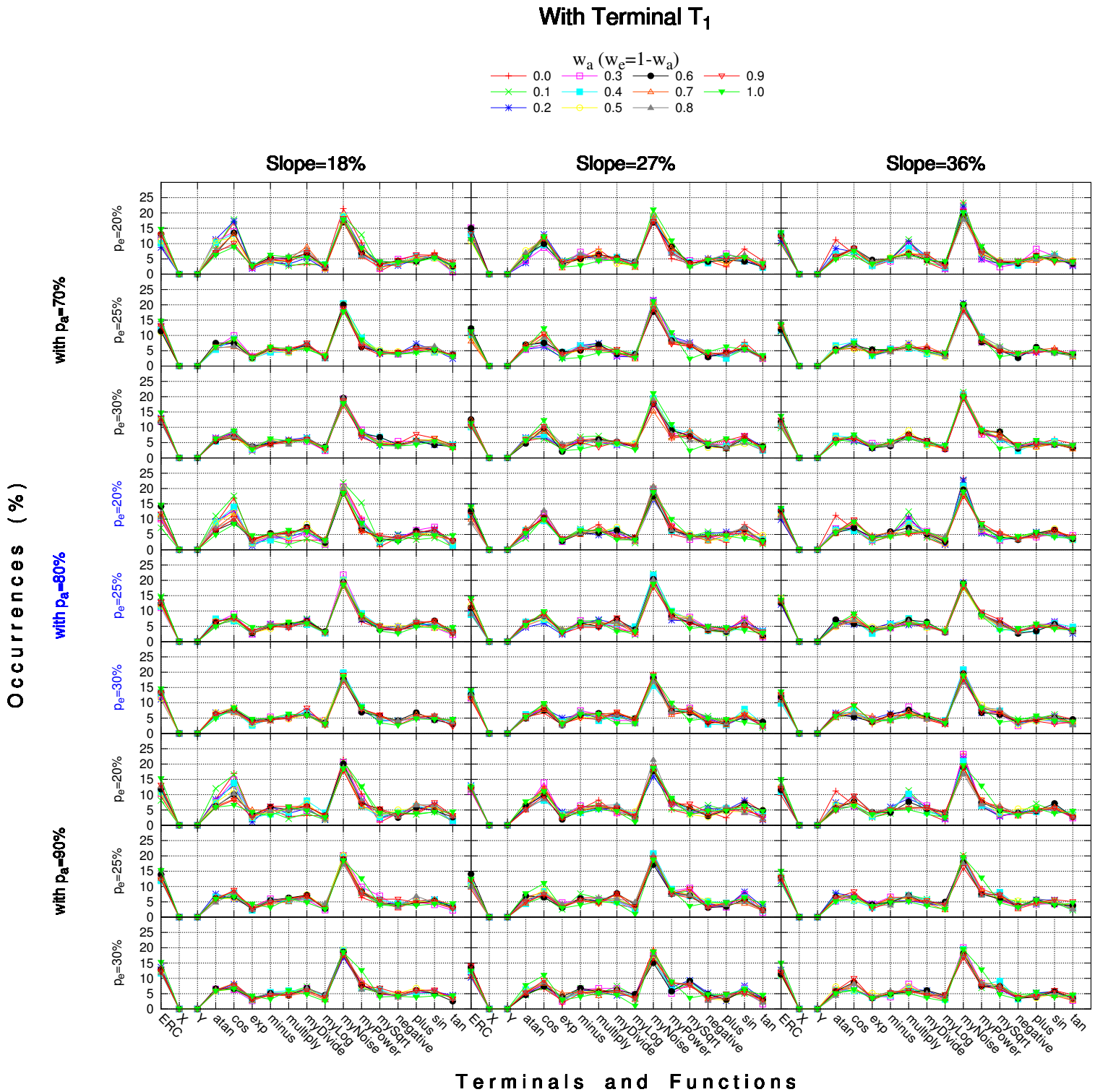

generations
Terminals
T1
T2
T3
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
generations
w_a (w_e = 1- w_a)

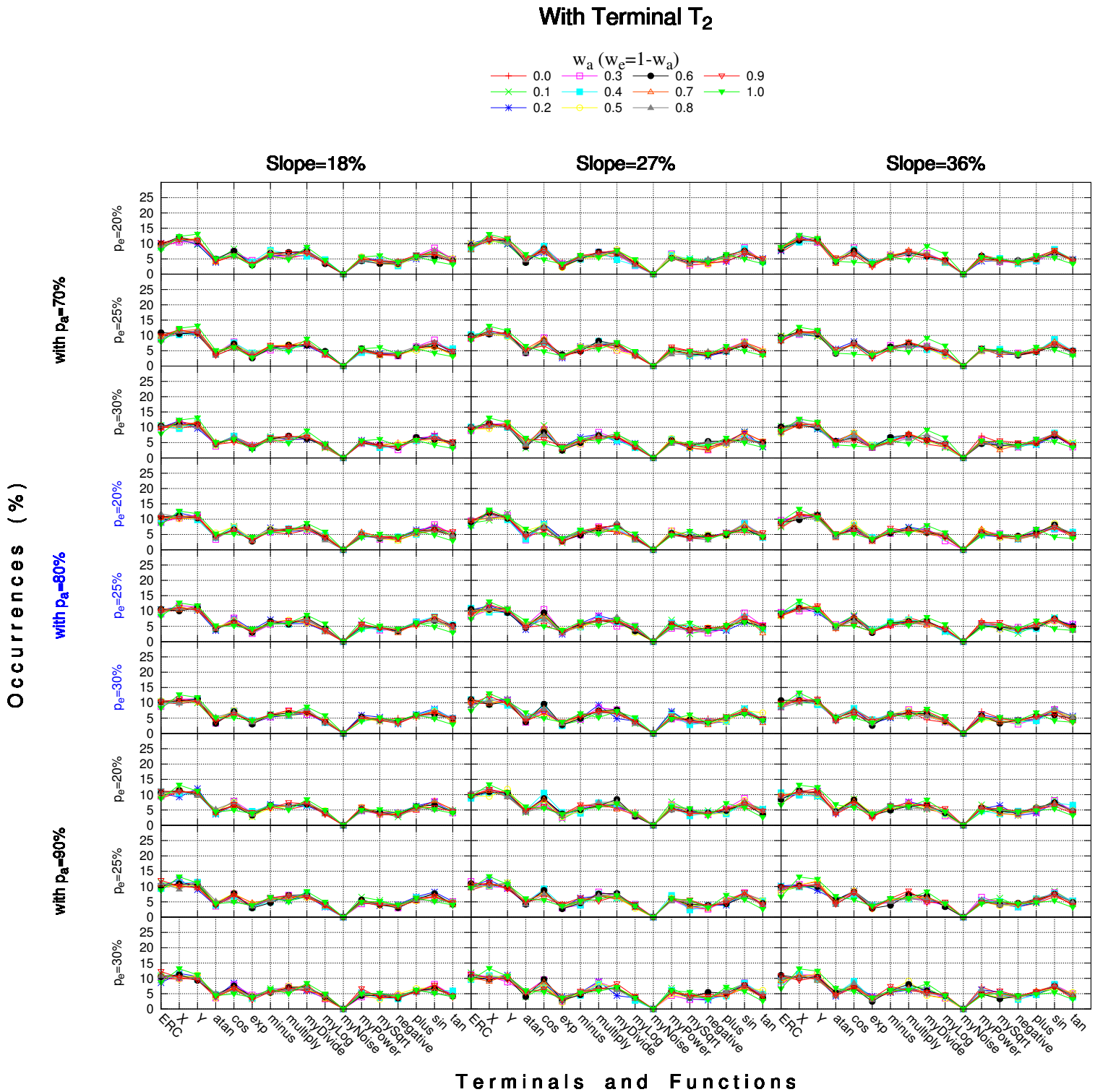

sizes
Terminals
T1
T2
T3
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
sizes
p_e=20%
p_e=25%
p_e=30%
0
40
80
120
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
w_a (w_e = 1- w_a)

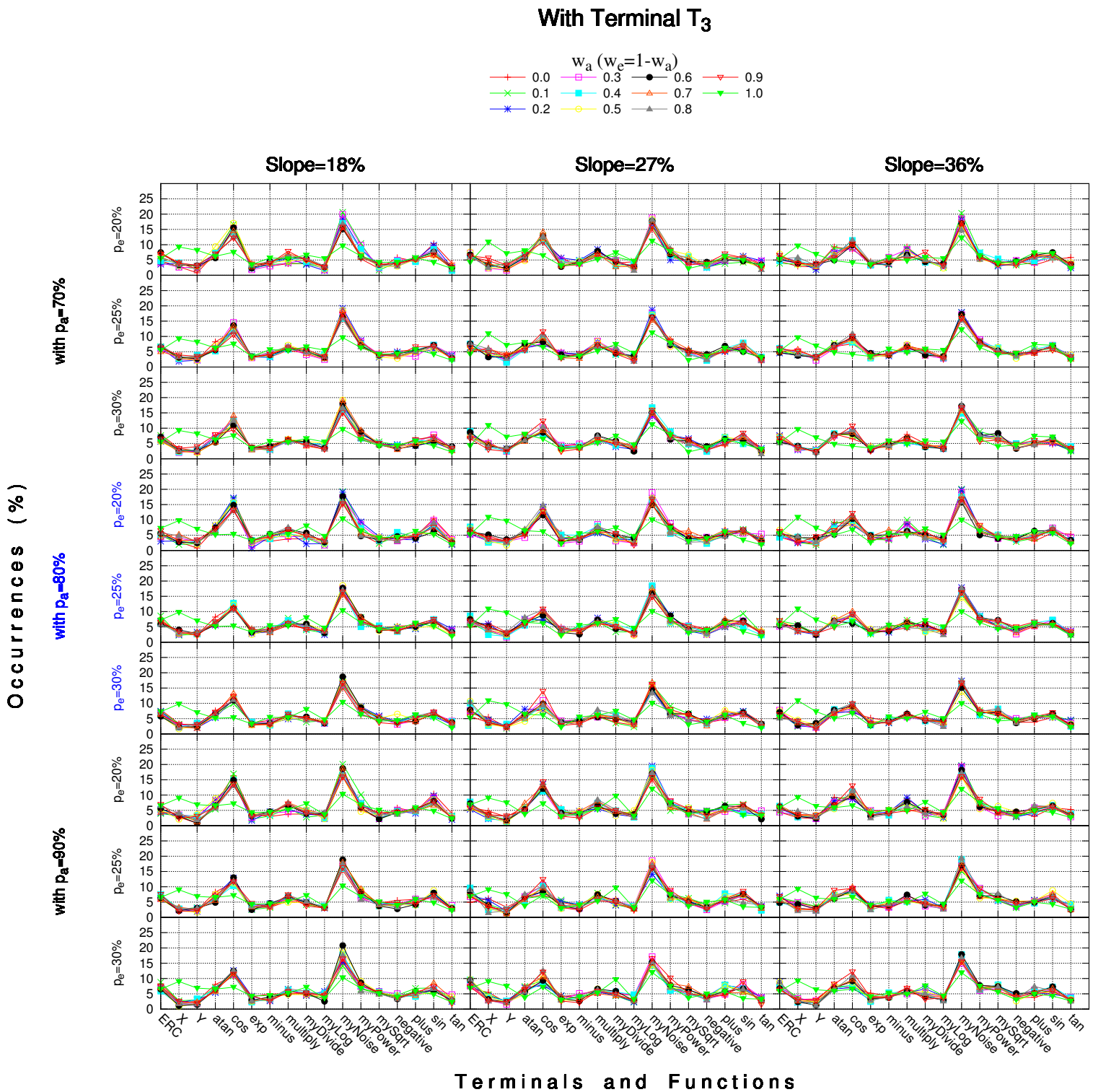

depths
Terminals
T1
T2
T3
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
depths
w_a (w_e = 1- w_a)

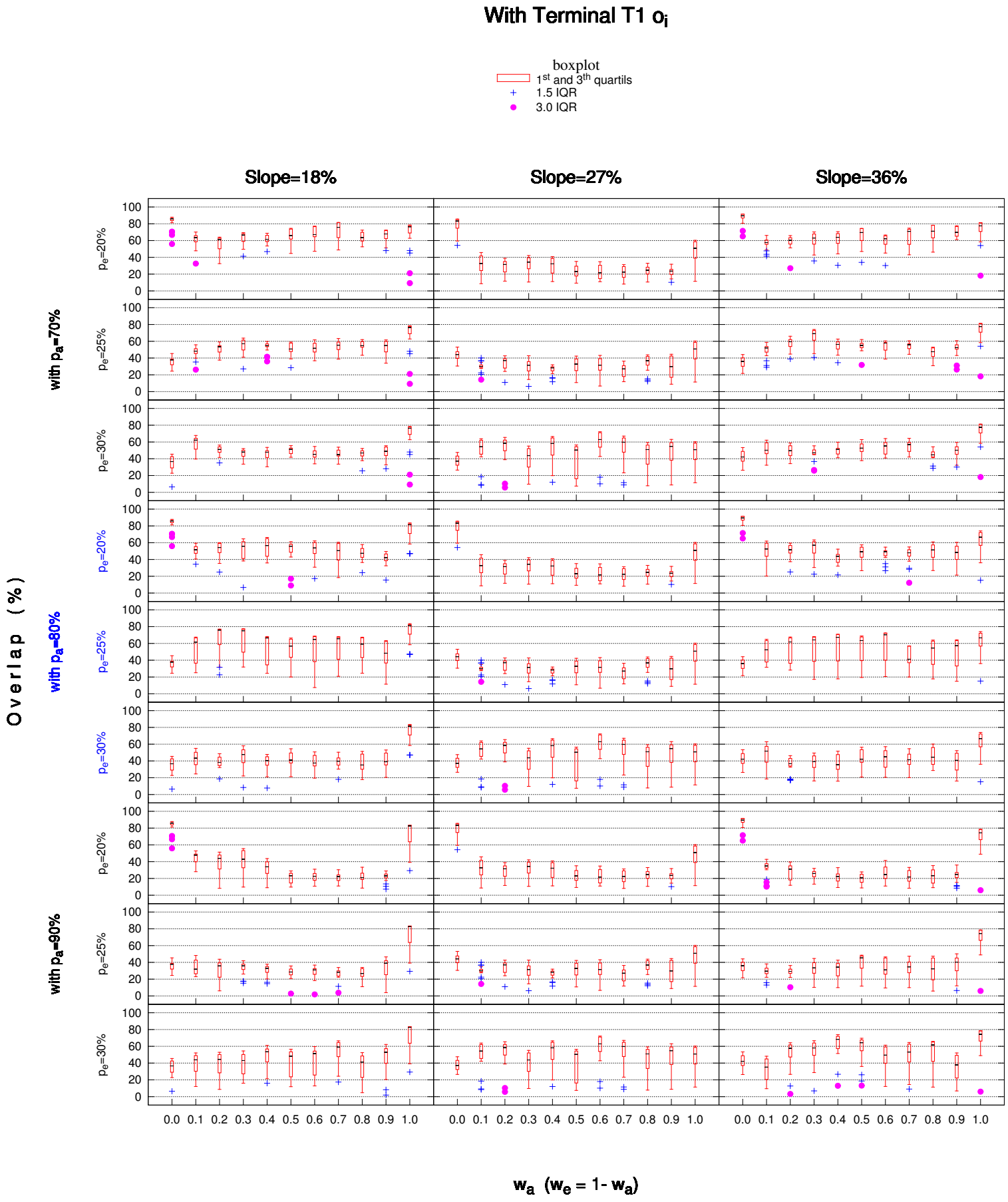

times
Terminals
T1
T2
T3
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
0
1000
2000
3000
4000
5000
6000
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
w_a (w_e = 1- w_a)

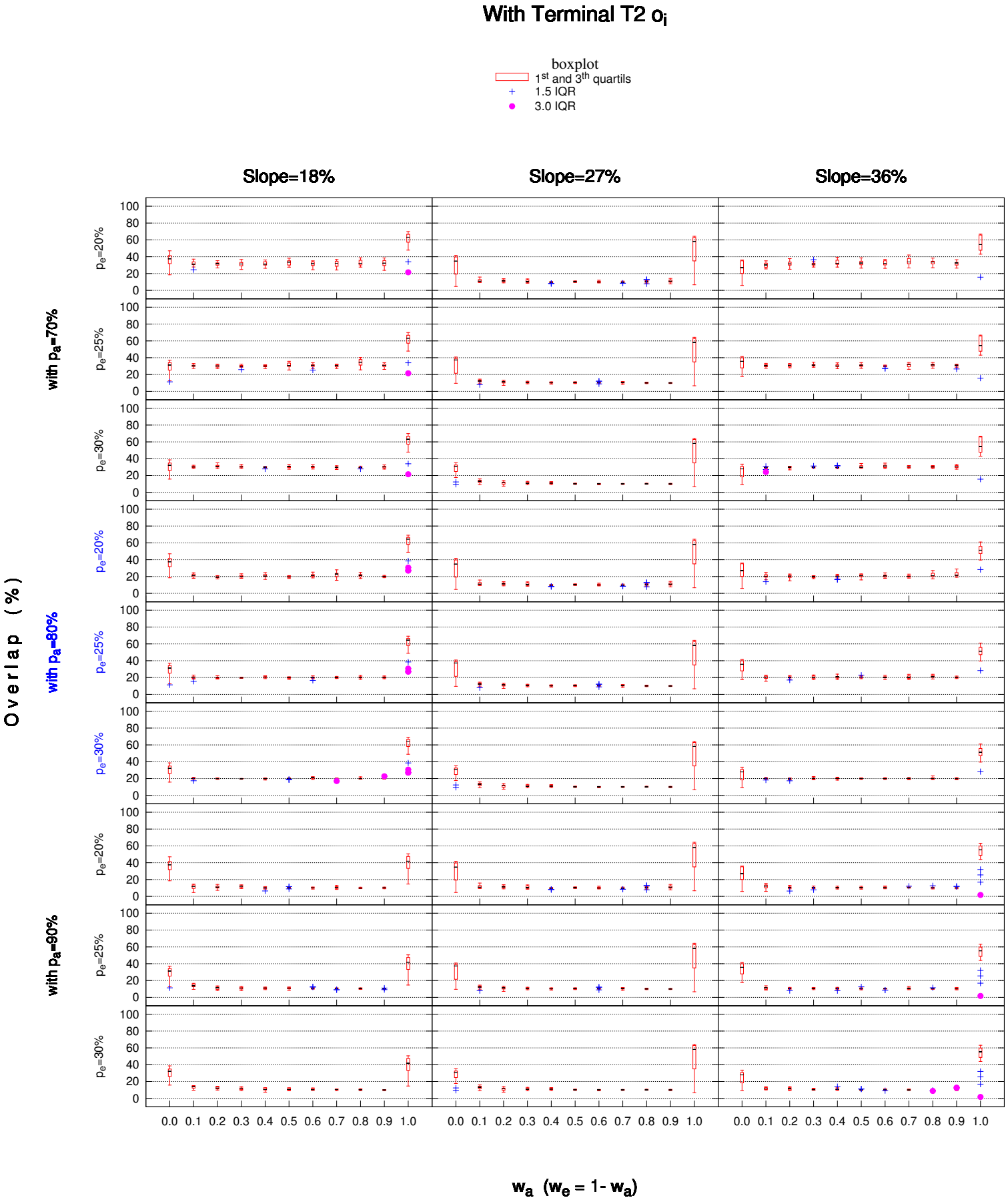

fitness
Terminals
T1
T2
T3
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
fitness
1
0.98
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
w_a (w_e = 1- w_a)

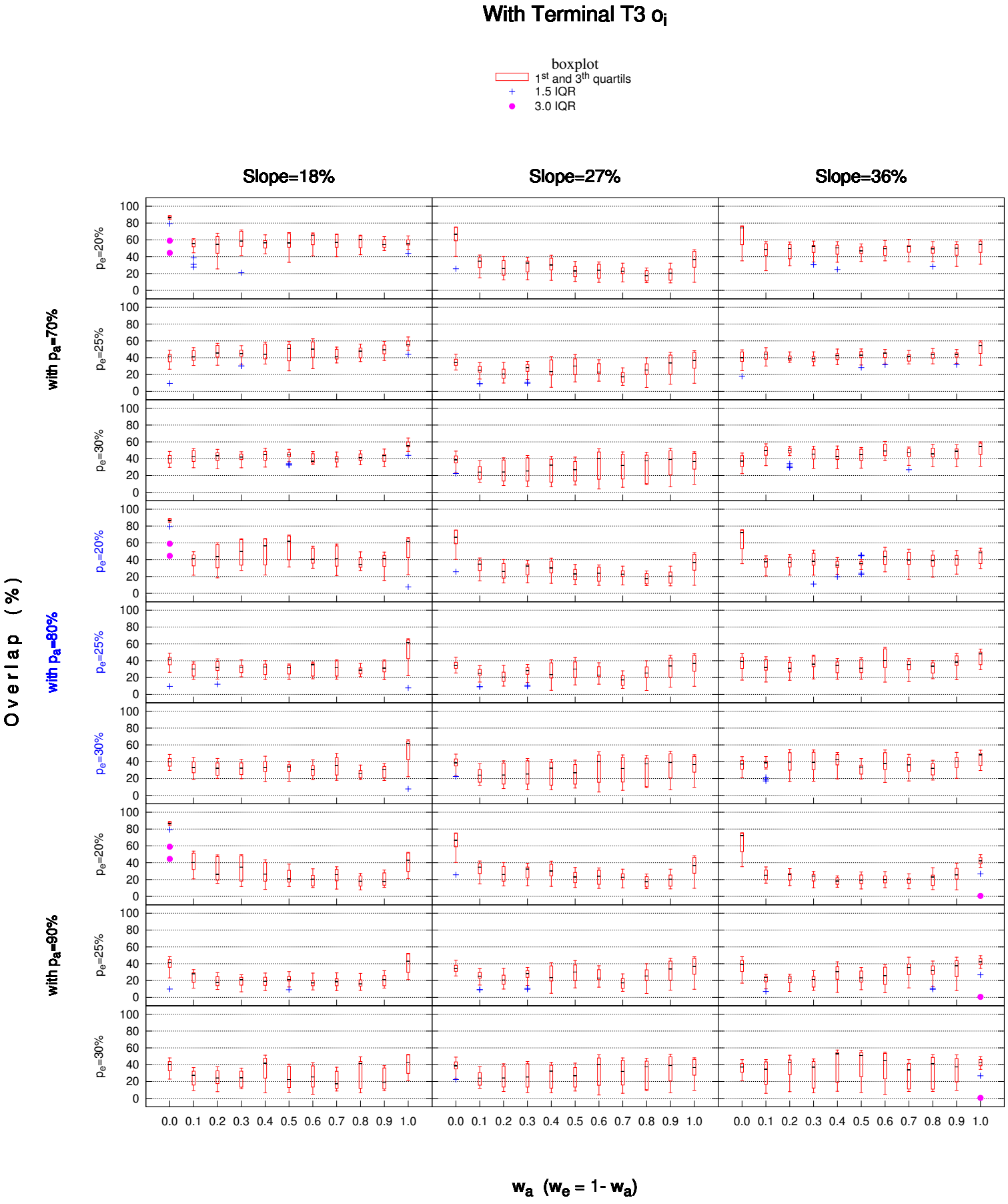

With Terminal T1
wa (we=1-wa)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Slope=18%
Slope=27%
Slope=36%
with pa=70%
with pa=80%
with pa=90%
pe=20%
pe=25%
pe=30%
Occurrences (%)
ERC
X
Y
atan
cos
exp
minus
multiply
myDivide
myLog
myNoise
myPower
mySqrt
negative
plus
sin
tan
Terminals and Functions


With Terminal T2
wa (we=1-wa)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Slope=18%
Slope=27%
Slope=36%
with pa=70%
with pa=80%
with pa=90%
pe=20%
pe=25%
pe=30%
Occurrences (%)
ERC
X
Y
atan
cos
exp
minus
multiply
myDivide
myLog
myNoise
myPower
mySqrt
negative
plus
sin
tan
Terminals and Functions


With Terminal T3
wa (we=1-wa)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Slope=18%
Slope=27%
Slope=36%
with pa=70%
with pa=80%
with pa=90%
pe=20%
pe=25%
pe=30%
Occurrences (%)
ERC
X
Y
atan
cos
exp
minus
multiply
myDivide
myLog
myNoise
myPower
mySqrt
negative
plus
sin
tan
Terminals and Functions


With Terminal T1 o_i
boxplot
1st and 3th quartils
1.5 IQR
3.0 IQR
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
Overlap (%)
w_a (w_e = 1- w_a)


With Terminal T2 o_i
boxplot
1st and 3th quartils
1.5 IQR
3.0 IQR
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
Overlap (%)
w_a (w_e = 1- w_a)


With Terminal T3 o_i
boxplot
1st and 3th quartils
1.5 IQR
3.0 IQR
Slope=18%
Slope=27%
Slope=36%
with p_a=70%
with p_a=80%
with p_a=90%
p_e=20%
p_e=25%
p_e=30%
Overlap (%)
0 20 40 60 80 100
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
w_a (w_e = 1- w_a)

Appendix C

List of Publications

List of publications achieved during the research period.

Peer-reviewed journals:

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta, Automatic Evolution of Programs for Procedural Generation of Terrains for Video Games, in *Soft Computing Journal (2011 impact factor 1.880)*, 22 pages, 2012, doi:10.1007/s00500-012-0863-z, (Frade et al, 2012b)

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta, Breeding Terrains with Genetic Terrain Programming - The Evolution of Terrain Generators, in *International Journal of Computer Games Technology*, vol. 2009, Article ID 125714, 13 pages, doi:10.1155/2009/125714 (Frade et al, 2009b)

Peer-reviewed conferences:

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta, Aesthetic Terrain Programs Database for Creativity Assessment, in *IEEE Conference on Computational Intelligence and Games*, 5 pages, 2012 (in press) (Frade et al, 2012a)

- Nelson Rodrigues, Miguel Frade, and Francisco Fernandez de Vega, Development of Chapas an Open Source Video Game with Genetic Terrain Programming, in *VII Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB)*, 8 pages, Valencia, Spain, Set. 2010 (Rodrigues et al, 2010)

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta, Evolution of Artificial Terrains for Video Games Based on Obstacles Edge Length, in *IEEE Congress on Evolutionary Computation 2010*, pages 1-8, IEEE, Jul. 2010 (Frade et al, 2010b)

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta, Evolution of Artificial Terrains for Video Games Based on Accessibility, in Chio CD, et al (eds) *Applications of Evolutionary Computing*, pages 90-99, Springer, Lecture Notes in Computer Science, vol 6024, Apr. 2010 (Frade et al, 2010a)

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta, Adding Zoom Feature to Terrain Programmes, in *VI Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB)*, pages 293-300, Málaga, Spain Feb. 2009 (Frade et al, 2009a)

- Miguel Frade, Francisco 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. (Frade et al, 2008a)

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta. GenTP – Uma Ferramenta Interactiva para a Geração Artificial de Terrenos. In *3rd Iberian Conference in Systems and Information Technologies (CISTI 2008)*, pages 1-12, Ourense, Spain, 2008 (Frade et al, 2008b)

- Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta. Modelling video games' landscapes by means of genetic terrain programming - a new approach for improving users' experience. In M. Giacobini, et al

(eds) *EvoWorkshops 2008*, volume 4974, pages 485-490, Napoli, Italy, 2008. Springer, Lecture Notes in Computer Science, vol 4974 (Frade et al, 2008c)

Exhibitions:

- Miguel Frade, Francisco Fernández de Vega, Carlos Cotta, Genetic Terrain Programming, exhibition in *UMA - Universidade Mostra Arte 2009*, Universidade Positivo of Curitiba, Brazil, 6/Oct - 4/Nov 2009