Ana Teresa d'Oliveira Campaniço

Programação Genética em Aplicações Gráficas para Jogos:

Simulação e Visualização de Plantas utilizando Flash Actionscript

Orientadores: José Benjamim R. da Fonseca

José Paulo B. de Moura Oliveira

Dissertação apresentada com vista à obtenção do grau de mestre em Informática na área, nos termos do Decreto-lei 74/2006 de 24 de Março e no Regulamento de Estudos Pós-Graduados da UTAD (Deliberação n.º 2391/2007).

UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

VILA REAL, 2008

Special thanks to,

My family,

For their never ending support.

Special Thanks

For the incentive and collaboration given by several people and the institution, to which I present the following thanks:

To the University of Trás-os-Montes e Alto Douro by the education provided while I was a student of Computer Science.

To my teacher Paulo Oliveira, supervisor of this thesis, for the source of inspiration he was while I was his pupil and for the huge support and help he gave me during the Masters.

To professor Benjamim Fonseca, co-supervisor of this thesis, for the constant help and feedback provided on my code and for helping me getting over the obstacles that kept getting in my way.

To my teachers Mark Leeney and Simon McCabe, former supervisors at Letterkenny Institute of Technology in Ireland, for their help and support in my work during my stay there.

To my family, in particular my parents, for providing me with this opportunity and being there all the way.

To my online friends, for helping me detecting bugs during the beta testing and for providing me with feedback, opinions and suggestions for my work.

Index of Contents

Index of Figures

Resumo

Tirando vantagem dos poderosos mecanismos existentes na natureza, o objectivo deste trabalho foi o de criar uma aplicação capaz de evoluir estruturas de plantas em Flash. Isto foi possível através da combinação da gramática L-System, que define a arquitectura da planta, e da Programação Genética, que evolui a planta produzida e gera uma população de filhos que diferem bastante dos pais originais em apenas algumas gerações.

O que este programa faz é a Validação da Sintaxe, a Produção e a Interpretação da planta L-System, pegando no axioma e regras de produção dadas e fazendo um constante substituição dos símbolos pelos seus respectivos sucessores durante várias iterações. De seguida a palavra é lida e cada comando interpretado para fazer o seu desenho.

Quando as diferentes plantas são atribuídas com um valor de aptidão pela sua aparência estética, as palavras que compõem a sua estrutura são enviadas para a Programação Genética a fim de servirem de indivíduos. Aí os indivíduos são seleccionados e os seus ramos aleatoriamente trocados entre pares de plantas de forma a gerar um par de plantas filho, sendo de novo enviadas para a Interpretação do L- System de forma a serem desenhadas.

Uma vez que as novas gerações de plantas são visualmente distintas das estruturas dos pais, conseguimos evoluir plantas L-Systems através da Programação Genética.

Abstract

Taking advantage of the powerful mechanisms existing in nature, the purpose of this work was to create an application capable of evolving a plant structure in Flash. It does so by combining the L-System grammar, which defines the architecture of the plant, and Genetic Programming, which will evolve the produced L-Systems and generate a population of children quite different from their original parents in just a few generations.

What this program does is the Syntax Validation, the Production and the Interpretation of the L-System plant, taking the given axiom and production rules and doing a constant replacement of the symbols with their respective successors during several iterations. Then the word is read and each command interpreted to draw the plant.

When the different plants are given a fitness value for their aesthetic appearance, the words that define their structures are sent to the Genetic Programming to serve as individuals. There the individuals are selected and their branches randomly switched between parent plants in order to create a pair of child plants, being those sent again to the L-System's Interpretation step to be drawn.

Since the new generations of plants are visually distinct from their parents' structures, we can evolve L-System plants through Genetic Programming.

1 Introduction

The natural world is a place of wonders and miracles, a palette full of ideas and hidden lessons just waiting to be discovered. Taking those natural mechanisms to simulate them in a virtual world is just another way of exploring new concepts and uncover new possibilities.

This work explores some of the systems inspired by natural systems. It takes the unlimited power Evolutionary Algorithms have to offer and attempts to evolve the plants produced by L-system grammatical structures. The global goal is to create an application which produces plants and allows the user to choose and evolve them within a Flash environment.

This chapter will explain how each of the topics was reached and the context this work is set in, along with the identified problems and proposed objectives.

1.1 Simulation and Visualization of Plants

How would you represent a plant in a computer?

It might sound like a silly thing to ask, but many people who are interested in simulating flora have come across some issues in trying to answer this question. If even a child knows how to draw a tree, then why can't the computer do the same?

Through observation of the real thing, a person builds a mental representation of what a plant should look like, taking the structure and all of its components through the same way. However the computer doesn't think, nor does it understand what a plant is without being told how it should process the information it receives.

So how can a computer represent a plant without knowing how it should look like? Someone has to tell it what the structure of the plant is. Because computers are mathematical machines, the answer has to be in a language it can understand. This brings us back to the original question: how to represent a plant. For centuries people have been trying to develop a formula that could explain how plants develop and grow (Grubert, 2001). The golden ratio was perhaps the first equation to explain how the proportion of all sections of the plant worked, but still it did not answer how the structure behaved (Olsen, 2006).

Many systems have been developed in order to solve this question, especially in recent years when computers came to aid in complex calculations (Rodkaew et al., 2004; Tan et al., 2007; Lluch et al., 2003), but no system offers a better solution so far than that developed by Aristid Lindenmayer in 1968 (Cited in Prusinkiewicz & Lindenmayer, 1990). The Lindenmayer systems, or L-systems as they are commonly known, are a formal grammar (Salomaa, 1973) that not only show how the structure of a plant is organized but allow us to see how it develops as it grows. Also, it’s a universal language, as it can be used to explain any given plant (Lindenmayer & Prusinkiewicz, 1996).

Summarily, the L-system takes in the axiom (Prusinkiewicz & Lindenmayer, 1990), a word composed by several symbols, which describe the structure of the plant, and on each iteration, or step of the growth, replaces the existing symbols by new ones, according to the production rules, rules which are used to determine how the growth will exactly happen (Lindenmayer & Prusinkiewicz, 1996).

This is basically what happens in the biological growth of the plant, the system that served as inspiration to Lindenmayer in 1968 (Cited in Prusinkiewicz & Lindenmayer, 1990).

1.1.1 Evolution of L-Systems

"Lindenmayer System is a grammar-like formalism that allows the generation of plant models. As a result of its grammatical derivation, there are strings containing the information used to draw a model of a biological organism; in the present case, a plant. Therefore, the grammar can be viewed as the genetic information of a plant. This information can be manipulated by an evolutionary algorithm, which is used to investigate the effects of applying genetic operators to evolve derived L-System plants"(Bonfim & Castro, 2005).

Despite of all expansion and refinement it suffered at the hands of other scientists (Grubert, 2001; Samuel, 2007; Chen et al., 2003; Bisoi et al., 2004; Borovikov, 1995), who wanted to improve this tool to be able to represent and simulate much more realistic trees and be able to predict their development under several circumstances, the L-system still has some limitations (Grubert, 2001; Prusinkiewicz 1986, 1993; Prusinkiewicz et al., 1990, 1994, 1997, 2000). A major one is the fact these plant structures cannot evolve (Jacobs, 1994, 1995a, 1995b, 1996; Noser et al., 2001; Ochoa, 1998).

What the L-system does is represent the genetic information of the plant and rules that determine its growth and development: in other words, the information contained within the genes (Lindenmayer & Prusinkiewicz, 1996). However the growth of an individual isn't determined just by that information, but also by outside factors and physical attributes.

A good example is a plant placed next to a window but not receiving direct sunlight. It will still grow according to the genetic information characteristic of its species, but this individual in particular will bend towards the window, in an attempt to receive as much sunlight as possible.

In order to produce plants that behave in a more realistic manner, Evolutionary Algorithms (Fogel, 1960; Rechenberg, 1973; Schwefel, 1975, *cited in* Coello, 2007), often Genetic Algorithms (Holland, 1975, *cited in* Coello, 2007), are used. By simulating the processes used by the Natural Selection to evolve populations of individuals (Darwin, 1859, *cited in* Russell & Norvig, 2003), these algorithms can optimize and solve the given problem (Golberg, 1989; Russell & Norvig, 2004). In this case, evolve a given population of plants, as seen further below.

1.2 Genetic Programming

Why Genetic Programming and not other Evolutionary Algorithms?

When talking about Evolutionary Algorithms (Fogel, 1960; Rechenberg, 1973; Schwefel, 1975, *cited in* Coello, 2007), Genetic Algorithms (Holland, 1975, *cited in* Russell & Norvig, 2004) usually come to mind. They are the most popular and the most commonly used, but that doesn't mean they are often the best approach.

The main difference between Genetic Algorithms (Holland, 1975, *cited in* Coello, 2007) and Genetic Programming (Koza, 1992) is in how the individuals within the population are represented in each. In the Genetic Algorithms they are traditionally arrays of bits of a fixed sized, while in the Genetic Programming they are small programs organized in a tree structure.

Because each node can take in operations (arithmetic, logic, etc), atomic values, or even L-system symbols, this makes the Genetic Programming more suitable to handle the evolution of L-system plants (Koza, 1992, 1993, 2007; Coello, 2007). Also, the fact tree structures aren’t limited in size like the arrays in the Genetic Algorithms, is another advantage, considering the recursive nature of the L-systems can make the axioms grow greatly (Koza, 1992, 1993, 2007).

However, it’s not mandatory to use Genetic Programming to evolve L-systems though. For example, in one application it’s used an array to store the three components that define all L-systems (Bian et al., 2004), while another simply generates the axiom before sending it to evolve (Bonfim & Castro, 2005).

The first approach is more suitable to evolve through Genetic Algorithms, because the size of the individual doesn’t vary during the evolutionary process, only the information contained within it. The second one works better with Genetic Programming because of the similarities the axiom and the tree structure share.

Both techniques are valid. Basically, it all depends on which approach the programmer thinks is best to solve the problem. Some even prefer to use systems outside the Evolutionary Algorithms or the L-systems, like Image Processing (Quang et al., 2006) or Particle Systems (Rodkaew et al., 2004) to reach the representation of the plants. Again, this is all up to the programmer.

1.3 Computer Games and Graphical Applications

So what’s the connection between generating virtual plants and computer games?

Most people outside the world of computer games usually don't realise it, but the graphical content necessary to produce them and the amount of resources it takes is a big issue within games (Lecky-Thompson, 2001; Azevedo, 2005; Leutenegger & Edgington, 2007). This is not a new situation however, but something that has always been part of them.

Basically, there are two approaches when developing graphics for games. The first one is Handmade Graphics, very unique looking images produced by artists (Sims, 1991). Their strength is on the highest level of quality, but there are some drawbacks. Besides the amount of money and time they take to produce and the amount of memory they often take, they limit the flexibility of the game itself (Azevedo, 2005).

After being created it's very hard to change the image. If it's necessary to use a different one in a later step of production, this means having to redraw it again, wasting more time and money again. The same applies to any variations in order to avoid overpopulation of duplicated items, which only consume even more memory and other valuable resources.

The second approach is Procedural Content Generation, the use of code to generate the graphics, and other elements, on the fly (Gibbs, 2004; Roden & Parberry, 2004, 2005). Compared to the handmade ones, the procedural graphics take very little space, a few kilobytes of code to the couple of megabytes for full images. Back in the beginning of computer games, when memory was a very limited resource, having algorithms to generate the different levels was an efficient way to save space (Gibbs, 2004; Roden & Parberry, 2004, 2005; Prachyabrued et al., 2007), but there are some drawbacks on this approach as well. Besides the great amount of time and effort it can take to develop the code, unpredictable and undesirable results can happen, especially when dealing with graphical content (Gibbs, 2004; Roden &

Parberry, 2004, 2005; Prachyabrued et al., 2007). A good example is particle systems. Despite the great graphical quality computers have today, it's still not possible to render fire or water out of pure code without people saying it looks fake.

A way to get around this is using hybrid versions to take a pre-fabricated work and have the code altering and generating new objects out of it. This is the approach most games take today (Prachyabrued et al., 2007; Parish & Müller, 2001; Wonka, 2006).

The connection this has with the artificial evolution of virtual plants is that L-Systems and Genetic Programming are ways of generating Procedural Content. Both of them generate the content on the moment of request, simply taking the rules that define them to produce the end result. Noise, fractals, particle systems, pseudo-random generators and many others, are all forms of Procedural Content Generation.

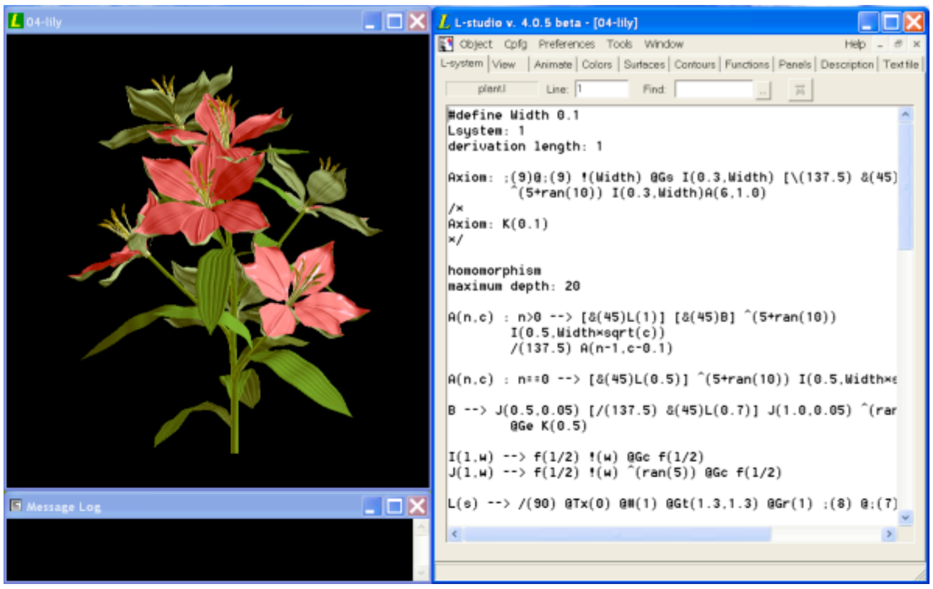
As for the connection with games, because there has been an increasingly higher demand for richer, more detailed and longer gaming experiences, the market of games has grown from a small handful of people to whole teams of programmers, artists, designers and such working together to strive in a highly competitive market (Roden & Parberry, 2005; Azevedo, 2005).

In order to aid this continuous search for diversity and originality, limited by deadlines and tight budgets, the use of externally developed resources has become more and more indispensable, both for programmers and designers, and many of these tools are based on Procedural Content to generate their elements.

1.3.1 Developed Applications

Though the concept of L-systems has been around for some time, the idea of developed applications is fairly recent. This is mostly because of the great boost computers experienced in terms of memory, graphical and processing capacities in the late years. Up until then, the machine couldn't keep up with the requirements of most heavy computations. In the case of L-systems, their recursive nature and the exponential amount of processing power require to process greater number of iterations.

Currently, there are many applications developed, too many to list. Some are simple experiments or small programs done to illustrate how the L-systems work, while others are more complex works that can realistically simulate the different components of plants and their behaviour under several circumstances. L-Studio (Prusinkiewicz et al., 2000; Prusinkiewicz & Karwowski, 2004) and SpeedTree are two good examples of professional applications. The first one is a more scientific driven program used to simulate and study plants in their different stages of development with the possibility to put it under several different kinds of environment to see how those affect their growth.


04-lily
Message Log
L-studio v. 4.0.5 beta - [04-lily]
Object Cpfg Preferences Tools Window Help
L-system View Animate Colors Surfaces Contours Functions Panels Description Text file
plant.l Line: 1 Find:

```
#define Width 0.1
Lsystem: 1
derivation length: 1

Axiom: ;(9)@;(9) !(Width) @Gs I(0.3,Width) [\(137.5) &(45)
        ^(5+ran(10)) I(0.3,Width)A(6,1.0)
/*
Axiom: K(0.1)
*/

homomorphism
maximum depth: 20

A(n,c) : n>0 --> [&(45)L(1)] [&(45)B] ^(5+ran(10))
        I(0.5,Width*sqrt(c))
        /(137.5) A(n-1,c-0.1)

A(n,c) : n==0 --> [&(45)L(0.5)] ^(5+ran(10)) I(0.5,Width*

B --> J(0.5,0.05) [/(137.5) &(45)L(0.7)] J(1.0,0.05) ^(ra
        @Ge K(0.5)

I(1,w) --> f(1/2) !(w) @Gc f(1/2)
J(1,w) --> f(1/2) !(w) ^(ran(5)) @Gc f(1/2)

L(s) --> /(90) @Tx(0) @M(1) @Gt(1.3,1.3) @Gr(1) ;(8) @;(7)
```

Fig.1 – L-Studio. Taken from http://algorithmicbotany.org/lstudio/whatis.html

SpeedTree on the other hand is a middleware specifically designed to produce realistic trees to populate games. It can procedurally create real-time, realistic 3D trees, as well as simulate wind and other effects on them.

Fig.2 – Game done with aid of SpeedTree. Taken from http://www.speedtree.com/

1.4 Flash Actionscript

Why Flash?

Like in the examples presented above, most developed applications are done in either C++ or Java. They are powerful and universal languages, but they aren't very intuitive when in comes to graphical development. For example, to create a rolling ball animation in Java, it's necessary to use several lines of scaffolding code before one reaches the drawing and animation part itself. In Flash the same doesn't happen (Crawford & Boese, 2006).

Flash is a tool meant to deal with graphical content and animation, especially when it comes to the web. Unlike many other applications, it offers a great deal of control and freedom to the user. When doing a web page for example, one doesn't have to worry about the code necessary to define the positions of the elements (Rosenzweig, 2003; Rhodes, 2007; Mook, 2003; Makar, 2003).

Another important aspect about Flash is interaction. It isn't just limited to generating movies and other one-way messages, so to speak, but it can also receive information and process it. The best known example is likely the amount of flash games found in the internet. Although most of them are simple, casual 2D games compared with their bigger cousins produced for consoles and PCs, they have a great popularity (Crawford & Boese, 2006; Rosenzweig, 2003; Rhodes, 2007; Mook, 2003; Makar, 2003).

However, all of this would be meaningless if there wasn't a programming language underneath to control it all. Actionscript is similar to Javascript in its structure, though more high-level. This language can control all the elements populating the movie clip and the movie clip itself. Also, it's possible to change their properties directly through code which is more reliable and more efficient than trusting the timeline and doing things by hand (Crawford & Boese, 2006; Rosenzweig, 2003; Rhodes, 2007; Mook, 2003; Makar, 2003).

Comparatively to other programming languages, like C++ or Java, Flash has some disadvantages (Rosenzweig, 2003). Some of its main weaknesses are:

- Being timeline based;
- Slow when processing heavy environments;
- Not really meant for 3D graphics;
- Having a limited feature set.

But in terms of strengths, besides what was already mentioned, Flash has (Rosenzweig, 2003):

- A rapid development rate;
- The ability to work with many different multimedia;
- A good delivery;
- And is easy to use and program.

These are the reasons why Flash is considered mostly as a web tool. It doesn't possess the requirements to produce major games or movies. Though with all the distribution the Internet has today, and its ability to be available immediately and free, Flash has become quite popular.

1.5 The Problem and Objectives

In recent years research has been done to expand the L-systems and to include the evolving nature of real plants by combining it with Evolutionary Algorithms, such as Genetic Programming. The aim of this work is to develop an application to explore this relationship between the two in a 2D environment.

Taking into account the topics presented previously and the fact that there was no Flash application found for the purpose, we are lead to believe it would be interesting to deepen the following question.

1.5.1 Problem

Is it possible to develop an application that simulates plants and uses genetic programming to optimize their graphical representation?

1.5.1.1 Sub-problems:

- What's the best structure to simulate 2D plants in a computer?
- How to optimize a graphical representation through an Evolutionary Algorithm?
- How to transfer that structure into the Actionscript language?
- Which are the parameters a user can manipulate to obtain the best graphical representations?

1.5.2 Objectives

By having the sub-problems defined, it's easier to settle a strategy on how to tackle the main problem, and through the sub-problems it's possible to define the goals the application must try to fulfil:

1. Identify a method used to build plants in a computer, or that can be adapted to work in a computer;
2. Determine how to use the Genetic Programming to evolve those plants;
3. Adapt the developed approach for Actionscript;
4. Allow some level of control to the user to select and adjust the plants.

1.6 Limitations

There are some things that pose an obstacle to the fulfilment of the milestones that are the objectives:

- The time available for the project;
- The knowledge of Actionscript 2.0;
- The lack of related code in Actionscript to study.

1.7 Dissertation Structure

After this introduction, Chapter 2 presents in more detail L-systems and Genetic Programming, as well as the main components on which the application is based on.

Chapter 3 explains the System Architecture and how the theories discussed in Chapter 2 where adapted to build the conceptual model.

Chapter 4 discusses the System Implementation through the different version of the program and how the conceptual model was translated into code, as well as the different improvements and changes each suffered.

Finally Chapter 5 presents the Final Conclusions and Future Improvements.

2 Theoretical Setting

2.1 Introduction

In this chapter we will take a deeper look on how exactly L-systems and Genetic Programming work. Both of them have roughly thirty years of work added since their first appearance in the scientific community, which makes it very difficult to cover all the developed work so far.

L-systems have a huge ramification of different varieties, but basically all of them have roots on four main ones, which will be addressed in more detail. Because the problem question only deals with a 2D environment, 3D plants will not be discussed, as well as timed L-systems because the goal isn't to visualise a plant in its several stages of development.

As for the Genetic Programming, the different operators which dictate the way it works are pretty much the same in all developed applications. However, because most people are more familiar with Genetic Algorithms, comparisons are done in all the points to better illustrate how each section work.

2.2 L-Systems

L-systems are a formal grammar used for plant structure generation, developed by Aristid Lindenmayer.

Originally, it was meant to reproduce the growth of simple plant structures, but the system was so well received by the scientific community, it received contribution from botanists, mathematicians and computer programmers, expanding and refining it into a multitude of varieties (Lindenmayer & Prusinkiewicz, 1990; Grubert, 2001; Prusinkiewicz, 1997).

Because there are too many different types of L-systems, only the four main ones will be explained, as they cover the deterministic, parametric, context-free and stochastic varieties. They will be presented by order of complexity (Grubert, 2001; Fuhrer, 2005).

2.2.1 DOL-Systems

The DOL-System is the simplest of all the different types of L-Systems. It uses a tuple G = <V, w, P>, which consists of:

- **V** (the alphabet) the set of variables;
- **ω** (the axiom or word) is the equation defining the initial state of the system, formed by symbols of **V**;
- **P** is the set of production rules, or productions, defining how the variables will be replaced by the combinations of symbols. The productions consist of a predecessor and a successor.

The process starts by picking the word and replacing the symbols within it according to the rules defined in the productions. Once all the productions have been applied, the axiom will become a new word and the process can repeat itself. Each replacement step is an iteration which represents a different point in time as the system grows and develops (Grubert, 2001; Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Bonfim & Castro, 2005; Onishi et al., 2003; Prusinkiewicz, 1997).

V: a_R, a_L, b_R, b_L

w: a_R

P$_1$: $a_R \rightarrow a_L b_R$

P$_2$: $a_L \rightarrow b_L a_R$

P$_3$: $b_R \rightarrow a_R$

P$_4$: $b_L \rightarrow a_L$

a_R

$a_L b_R$

$b_L a_R a_R$

$a_L a_L b_R a_L b_R$

Fig.3 – Example of DOL system. Adaptation from Prusinkiewicz & Lindenmayer, 1990

As can be observed in the example shown in Figure 3, the initial word invokes the first production, but in the second iteration there has been the replacement of both symbols in a single step. This illustrates the parallelism feature typical in l-systems, mimicking nature in the sense of several changes happening in different places in the same time. (Grubert, 2001; Envall, 2007; Bonfim & Castro, 2005; Prusinkiewicz et al., 1988).

Unlike the Chomsky grammar, a formal language similar to the l-systems, terminals (symbols that can't be replaced) and non-terminals (symbols that can be replaced) are not used in l-systems. This is because there isn't a final goal to be reached. (Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Grubert, 2001; Bonfim & Castro, 2005). Also, because all words produced are valid models, the empty word λ can be used in a production. As long as all the symbols belong to the alphabet V, they are valid (Grubert, 2001; Bonfim & Castro, 2005; Prusinkiewicz et al., 1988).

2.2.2 Graphical Interpretation

2.2.2.1 Tree Structure

One interesting aspect of the l-system word is that it can be interpreted as a tree structure. Each symbol represents a node and by reading the word from left to right, with the left-most symbol being the root, we obtain a structure reminiscent to real plants (Prusinkiewicz & Lindenmayer, 1990, 1996; Grubert, 2001; Envall, 2007; Prusinkiewicz et al., 1988). These structures will be very much like straight lines, unless we use the branching structure so well known in plants. To achieve that we need to use two additional symbols, the square brackets to define the beginning and end of a branch $V' = V \cup \{ [,] \}$ (Grubert, 2001; Envall, 2007; Prusinkiewicz et al., 1988).

When the open bracket ([ ) occurs the values of the current node and hidden parameters are stored, as well as the position where it occurred. The word is drawn normally until another open bracket or the corresponding closing bracket ( ]) occurs. When the bracket closes the stored values for that bracket pair are then returned and replace the previous ones. The rest of the tree is drawn from that saved point and the section that was enclosed by the brackets is a sub-tree that sticks out much like a branch from the main trunk (Grubert, 2001; Envall, 2007; Fuhrer, 2005).

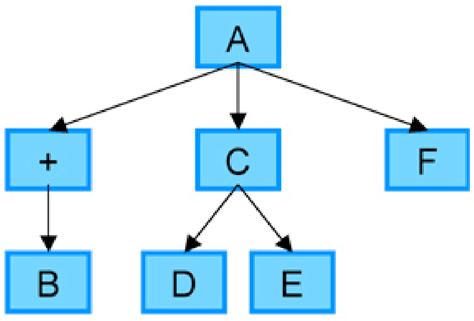


Fig.4 – Example of tree representation of the word “A [+ B] [C [D] E] F”. Adaptation from figure 1 of Grubert (2001).

2.2.2.2 Turtle Representation

Although the tree structure helps visualizing the l-system word, it doesn't give a clear idea of the appearance of the model. In order to have a full idea of their appearance the LOGO turtle graphics are used (Grubert, 2001; Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Fuhrer, 2005).

The turtle is a cursor that moves and rotates in a Cartesian coordinate system, depending on the instructions given and the defined angle. Examples of some of the commands are (Grubert, 2001; Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Envall, 2007; Fuhrer, 2005; Onishi et al., 2003):

F : Move forward a unit and draw a line from the last position to the current one;

f : Move forward a unit, without drawing a line;

+ : Rotate the cursor counter-clockwise by angle α;

- : Rotate the cursor clockwise by angle α.

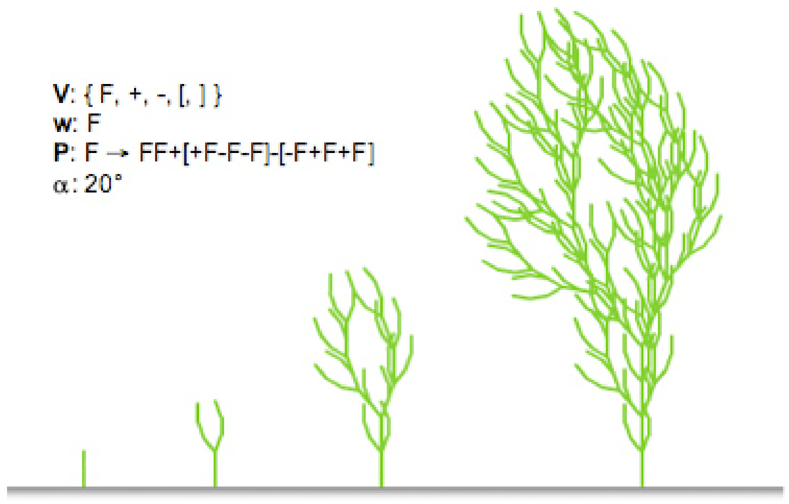


Fig.5 – Example of a turtle representation of a L-System. Adaptation from figure 2 of Grubert, (2001)

As it can been seen in the example shown in Figure 5, each iteration replaces each unit of the previous one with the whole structure previously generated, creating a model with increasing detail. This happens as each predecessor (in this case just one) is replaced by its correspondent successor (Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Envall, 2007; Chen et al., 2003; Onishi et al., 2003).

Values like angles and unit length are hidden parameters. They are set before the iterations take place and do not change as they go (Envall, 2007; Prusinkiewicz et al., 1988).

2.2.3 Parametric L-Systems

A problem with the DOL system is that when you need to use different sizes or angles, it's necessary to find a denominator common to all and combine several transformation symbols to achieve a certain rotation or translation (Grubert, 2001; Prusinkiewicz & Lindenmayer, 1990, 1996).

The Parametric L-System allows adding different parameters to different symbols. These parameters can be used to store information about the model from age and size to angle and time (Grubert, 2001; Envall, 2007; Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz & Hanan, 1990; Prusinkiewicz, 1997).

V: $\{ A \}$

w: $A (2)$

P_1: $A (a): a > 0 \rightarrow A (a - 1)$

P_a: $A (a): a <= 0 \rightarrow \lambda$

$A (2) => A (1) => A (0) => \lambda$

Fig.6 – Example of a Parametric L-System. Adaptation from Grubert (2001)

As can been seen in the example shown in Figure 6, the symbol is followed by the parameter. Like in the DOL system, every time the productions occur the value is replaced accordingly, though the conditions have to be fulfilled for the value to be altered (Prusinkiewicz & Lindenmayer, 1990, 1996; Grubert, 2001; Envall, 2007; Fuhrer, 2005; Prusinkiewicz, 1997).

In the turtle interpretation, in order to draw the model the operators used are a bit different from the ones used in DOL systems (Envall, 2007; Bonfim & Castro, 2005; Prusinkiewicz, 1997):

F(a) : Move forward a unit of length a > 0 and draw a line from the last position to the current one;

F (a) : Move forward a unit of length a > 0, without drawing a line

+(a) : Rotate the cursor by angle *a*. Counter-clockwise if *a* is positive, clockwise if negative.

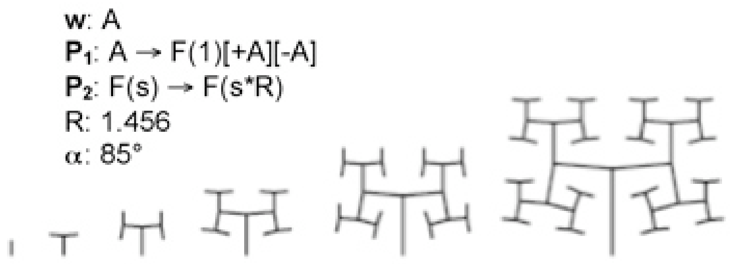


Fig.7 – Example of a Parametric L-System. Adaptation from figure 1.40 from Envall (2007)

2.2.4 Context-Sensitive L-Systems

Up until now we've only seen context free systems (OL-Systems), systems that don't take into consideration the context its predecessor is in, but any living organism suffers influence from its surrounding environment which affects its growth.

The Context-Sensitive L-System takes into consideration the different interactions between the different sections of the word by verifying what’s happening on the right and left of the said symbol (Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Envall, 2007 Bonfim & Castro, 2005; Fuhrer, 2005).

V: { A, B, C }

w: A(5) B(0) C(0)

P_1: A(n) < B > C: true → B(n)

P_a: B(n) < C: n > 2.5 → C(n – 2)

A(5) B(0) C(0) => A(5) B(5) C(0) => A(5) B(5) C(3) => …

Fig.8 – Example of a Context-Sensitive L-System. Adaptation from Prusinkiewicz & Lindenmayer (1990)

As you can see in Figure 8, the productions use the < and > brackets to check what is happening on the left and right context, respectively. It’s not mandatory to check both sides, as illustrated in the second production, but in either case the production will only be applied if the specified context occurs. (Grubert, 2001; Prusinkiewicz & Lindenmayer, 1990, 1996; Prusinkiewicz et al., 1988; Envall, 2007; Chen et al., 2003).

In the example from Figure 8 the first production checks if there’s a symbol A on the left and a symbol C on the right of the symbol B. In case that’s true then the symbol B takes the value *n* in A and replaces its own with it. The second production only checks if there’s a symbol B on the left of C, but the sucessor will only occur when the condition is true.

In the situation where both context-free and context-sensitive productions are used and both applied on the same letter, the context-sensitive has priority over the context-free production. However, if neither occurs, then the letter is replaced by itself (Prusinkiewicz & Lindenmayer, 1990, 1996; Envall, 2007).

The [and] brackets make it harder to keep track of the context, because the branches they specify don't preserve the segment neighborhood. Therefore, it's often necessary to skip the information contained within them to verify the context of the other symbols (Prusinkiewicz & Lindenmayer, 1990).

Below is an example of a word where the predecessor BC < S > G [H] M is valid to the symbol S, because it skips over the symbols [DE] to look for the left context and the I [JK] L to search for the right one.

ABC [DE] [SG [HI [JK] L] MNO]

In the turtle interpretation the + and – symbols are ignored when verifying the context in the word (Prusinkiewicz & Lindenmayer, 1990, 1996).

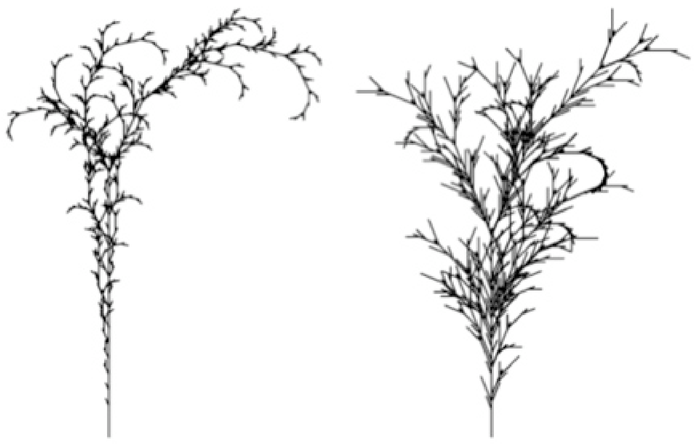


Fig.9 – Example of Context-Sensitive L-Systems. Adaptation from figure 1.31 from Prusinkiewicz & Lindenmayer (1990).

2.2.5 Stochastic L-Systems

All of the L-Systems seen so far are deterministic in nature. No matter how many times they are run, as long as the word and the productions stay unchanged, the result is always the same. In some situations this might not be a good option.

The Stochastic L-Systems introduce a variation from individual to individual, but does not alter the general information of the species. This is done by introducing a probability value that will determine which production will be used (Grubert, 2001; Prusinkiewicz & Lindenmayer, 1990, 1996; Chen et al., 2003).

V: { F, +, -, [,] }

w: F

P$_1$: F $\rightarrow$0.33   F [+F] F [-F] F

P$_2$: F $\rightarrow$0.33   F [+F] F

P$_3$: F $\rightarrow$0.34   F [-F]

Fig.10 – Example of a Stochastic L-Systems. Adaptation from figure 3 from Grubert (2001).

As can been observed in the example presented in Figure 10, the probability for each of the productions to occur is written in front of the arrow, though in some cases it's written on top, where the sum of the probabilities is 1 (Grubert, 2001; Envall, 2007). During the iterations, if more than one production occurs on the same symbol a random number is generated to select one of them, based on their assigned probabilities (Grubert, 2001; Envall, 2007).

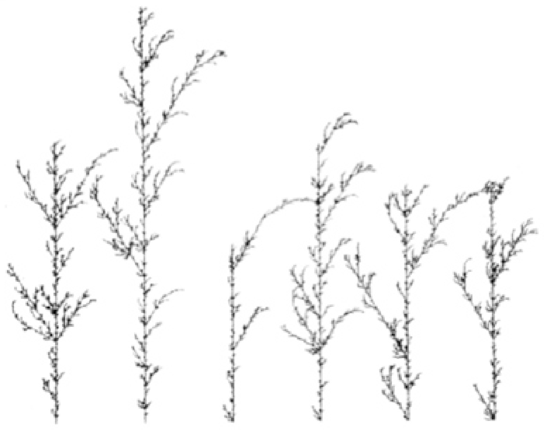


Fig.11 – Example of a Stochastic L-Systems. Adaptation from figure 1.27 from Prusinkiewicz & Lindenmayer (1990)

2.3 Genetic Programming

The theory of evolution proposed by Charles Darwin in 1882 (Cited in Russell & Norvig, 2004; Azevedo et al., 2005), in which natural selection plays a crucial role in the species evolution, is by far the best model which can explain the wide variety of life in our planet (Darwin, 1859, *cited in* Russell & Norvig, 2003). The theory has evolved, with the aid of new discoveries, such as genes, which has filled many of the gaps that Darwin struggled to explain during his time, but the theory itself is in constant evolution (Darwin, 1859, *cited in* Russell & Norvig, 2003).

Natural Selection is a process that favours certain individuals in a population in order for a species to evolve (Russell & Norvig, 2004; Azevedo et al., 2005; Vega, 2001).

Each individual carries genotype and phenotype information. Genotype is the type of genes it carries, all the information it inherited from its parents and the information its offspring will inherit from it. Phenotype is its physical attributes, like weight for example, and they determine its survival and reproduction chances (Peterson, 1997, Azevedo et al., 2005; Vega, 2001).

These traits determine the ability of an individual to survive in its environment. Those with more favourable traits are more likely to survive and pass on their heritage than those with less favourable attributes: this is the survival of the fittest principle. As new generations keep evolving and continuously adapting to their ever changing environment, there's the chance new species may emerge as they take different ecological niches (Peterson, 1997, Azevedo et al., 2005; Vega, 2001).

Evolutionary Algorithms, in which Genetic Programming is a particular branch, take advantage of this highly powerful mechanism and try to simulate it in a computer. This will be explained further below.

2.3.1 Evolutionary Approaches

The power of nature has always marvelled humanity, but only in the last few centuries has mankind truly tried to understand and harvest its potential. The creation of velcro, the invention of the airplane, all of these ideas were inspired by models that existed in nature for ages (Vega, 2001).

The same happens in Evolutionary Computation. This subfield of Computacional Intelligence takes inspiration from biological evolution and natural selection to optimize possible solutions and solve a given problem (Morais, 2003; Coello, 2007; Pappa & Freitas, 2006; Vega, 2001)

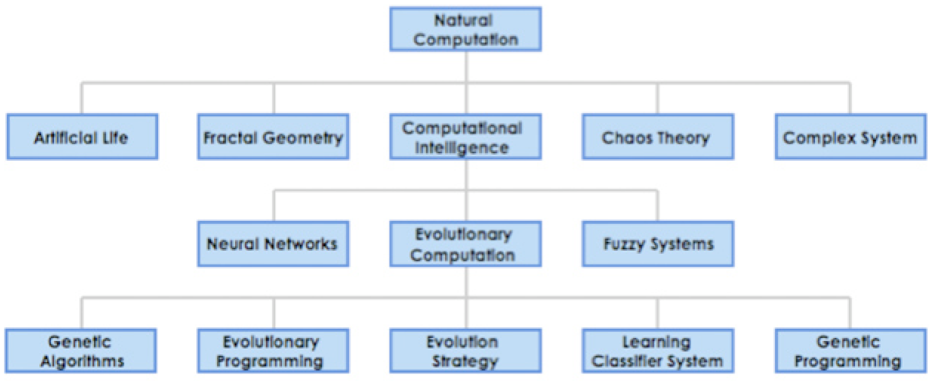


Fig.12 – Evolutionary Computation Index. Adaptation from figure 1 of Morais (2003)

As shown in Fig.12, the Evolutionary Computation branches into five different subfields: Genetic Algorithms, Evolutionary Programming, Evolution Strategy, Learning Classifier System and Genetic Programming. All of them share the same evolutionary nature but take different approaches when it comes to its implementation (Morais, 2003; Coello, 2007; Pappa & Freitas, 2006).

2.3.1.1 Genetic Algorithms

The most popular of the Evolutionary Algorithms, Genetic Algorithms, use the mechanisms of natural evolution, such as selection, crossover, mutation and fitness evaluation,and apply them to a population of individuals (Morais, 2003; Coello, 2007; Vega, 2001).

Each of the individuals (in a canonical GA) has a fixed size and is usually represented by an array of bits, in analogy to chromosomes in DNA. One of the advantages of that representation is the simplicity of the crossover operator, though it’s possible to use arrays of different sizes, but that increases the level of complexity (Morais, 2003; Coello, 2007; Vega, 2001).

```
Begin {
      t = 0;                      // Initialization of time
      initPop P(t);               // Initialization of Population
      fitnsEval P(t);             // Evaluation of the Initial Population

      While (solution not reached) {
              t ++;                          // Incrementation of time
              P’ = selctParent P(t);   // Select parents of future generation
              crossover P’(t);               // Breed parents
              mutate P’(t);                  // Add diversity to generation
              evaluate P’(t)    ;            // Evaluate fitness of generation
              P = survive P,P’(t);           // Replace population
      }
}
```

Fig.13 – Example of a Genetic Algorithm. Adaptation from figure 2 of Morais (2003).

2.3.1.2 Evolutionary Programming

Evolutionary Programming shares a lot of similarities to Genetic Algorithms, but it pays more attention to the behavioural relationship between parents and offspring, instead of mimicking the genetic operators found in nature (Morais, 2003; Coello, 2007).

The main difference in the implementation method is not using the crossover operator but relying on mutation and fitness to determine the survival. Another difference between evolutionary programming and Genetic Algorithms is the use of graphs instead of arrays in the solution representation (Morais, 2003; Coello, 2007).

```
Begin {
    t = 0;                      // Initialization of time
    initPop P(t);               // Initialization of Population
    fitnsEval P(t);             // Evaluation of the Initial Population

    While (solution not reached) {
        P' = mutate P(t);           // Add diversity to Population
        evaluate P'(t)    ;         // Evaluate fitness of
generation
        P = survive P,P'(t);        // Replace population
        t ++;                       // Incrementation of time
    }
}
```

Fig.14 – Example of an Evolutionary Programming algorithm. Adaptation from figure 3 of Morais (2003).

2.3.1.3 Evolution Strategy

The Evolution Strategy works with vectors of real numbers to represent solutions. Its approach is for each parent to produce an offspring per generation by the use of mutations. It's more likely for the evolution to occur in smaller steps until the descendent shows a better performance than its parent, replacing it (Morais, 2003; Coello, 2007).

```
Begin {
    t = 0;                          // Initialization of time
    initPop P(t);                   // Initialization of Population
    fitnsEval P(t);                 // Evaluation of the Initial Population

    While (solution not reached) {
      P''(t) = selectBest (a, P(t));    // Select best parents
      P'(t) = crossover (b, P''(t));         // Breed parents
      mutate P'(t);                          // Add diversity to generation
      evaluate P'(t)    ;                    // Evaluate fitness of
generation

      if (usePlusStrategy)
              P (t+1) = P'(t) ∪ P(t);     // Child joins Population
        else
              P (t+1) = P'(t);            // Population remains same

      t ++;     // Incrementation of time
    }
}
```

Fig.15 – Example of an Evolution Strategy algorithm. Adaptation from figure 4 of Morais (2003).

2.3.1.4 Learning Classifier System

The Classifier System doesn’t use a fitness function to evaluate the individuals, but a rule based on a reinforcement learning technique (Morais, 2003; Coello, 2007). This technique works by setting the program on an environment about which it has no knowledge and through a set of actions provided, reacts and classifies the environment as it changes. The Classifier System, specifically, has a population of binary rules on which a special genetic algorithm selects the best ones (Morais, 2003; Coello, 2007).

```
// Takes decisions by If-then rules
(1) If (ship is left) then send @
(2) If (ship is right) then send %
(3) If (ship is centre) then send $
(4) If (ship is attacking) then send #
(5) If (ship is not attacking) then send
*
(6) If (* and @) then don’t set cannon
(7) If (* and %) then don’t set cannon
(8) If (* and $) then set cannon
(9) If (#) then fire cannon
```

Fig.16 – Example of Learning Classifier System rules. Adaptation from figure 5 of Morais (2003).

2.3.1.5 Genetic Programming

Genetic Programming is very similar to Genetic Algorithms, to the point that some authors consider it a subtype (Coello, 2007). The main difference is that in Genetic Programming individuals are represented as a tree instead of an array or string. Actually, they’re not a string of characters, but programs, thanks to the way of defining information (Morais, 2005; Peterson, 1997; Coello, 2007; Pappa & Freitas, 2006; Vega, 2001).

The array representation has some limitations the tree-based structure can overcome. Because of their rigid nature, they’re not suitable to represent arbitrary computational procedures or to incorporate iterations or recursion within the individual. Also, their fixed size doesn’t allow much dynamic variability being the string length given in the initial population (Koza, 1990, 1992, 2001, 2007).

```
t = 0;                          // Initialization of time
initPop P(t);                   // Initialization of Population
fitnsEval P(t);                 // Evaluation of the Initial Population
While (solution not reached) {
        t ++;                   // Incrementation of time
        if (reproduce) {
                P’ = selctProg P(t);  // Select individual programs
        }
        else {
                if (recombine) {
                        P’ = selctParent P(t);  // Select parents
                        crossover P’(t);        // Breed parents
                }
        }
        mutate P’(t);                   // Add diversity to generation
        evaluate P’(t)      ;           // Evaluate fitness of generation
        P = survive P,P’(t);            // Replace population
}
```

Fig.17 – Example of a Genetic Program. Adaptation from Algorithm 2.2 fig of Peterson, 1997.

2.3.2 Operators

According to Koza, the creator of Genetic Programming, the algorithm can be divided into two major operators and five minor ones (Koza, 1990, 1992, 2001, 2007). The two major operators are reproduction and crossover, the fundamental forces behind evolution (Peterson, 1997; Koza, 1990, 1992, 2001, 2007; Vega, 2001).

The five minor ones are mutation, permutation, editing, encapsulation and decimation. Usually, only mutation is used and sometimes editing and encapsulation as well, while permutation and decimation are quite rare (Peterson, 1997; Koza, 1990, 1992, 2001, 2007; Vega, 2001).

Koza defends that the two major operators are enough to have a working Genetic Program, but the minor ones may be used to provide extra functionality (Peterson, 1997; Koza, 1990, 1992, 2001, 2007).

2.3.2.1 Individuals and Initial Population

As stated before, individuals in Genetic Programming are programs organized in a tree structure. Their shape, size and complexity change dynamically during the evolutionary process, but because individuals are represented in a tree hierarchy, the initial population generation is more complex than in Genetic Algorithms (Peterson, 1997).

Instead of a randomly generated number, a correct tree structure has to be created. Each tree is created individually with a single function as the root and has a predefined tree depth maximum value to balance the weight of the branches (Peterson, 1997; Koza, 1990, 1992, 2001, 2007; Vega, 2001). This is important during the crossover in order to prevent unwanted individuals from being generated.

```
// F = {$f_1$, $f_2$, …, $f_n$}; set of functions
// T = {$t_1$, $t_2$, …, $t_n$}; set of terminals

Randomly select a function root $\in$ F
If (depth = $depth_{max}$ – 1) {
        Randomly select a terminal $\in$ T
}
else {
        if (root is a function) {
                for ($arg_j$ ,  j = 1,…, $\phi$(root)) {
                        $\phi$(f) = the parity of f
                        if (use grow method) {
                                Randomly select $arg_j \in F \cup T$
                        }
                        else {
                                if (use full method) {
                                        Randomly select $arg_j \in F$
                                }
                        }
                        Recursively generate subtree with $arg_j$ as root
                }
        }
        else {
                root is a terminal
        }
        This branch is complete
}
```

Fig.18 – Generating a Random Program. Adaptation from Algorithm 2.3 fig of Peterson, 1997.

Figure 18 shows the generation of an individual. Koza defines two different variations, the Grow and the Full methods. The Grow generates trees of random size and shape while the Full produces them with uniform size and shape. However, in order to ensure diversity, he uses a hybrid version of the two methods, called Ramped Half and Half method (Koza 1990, 1992, 2001, 2007; Peterson, 1997; Vega, 2001).

The maximum tree depth is ramped according to the size of the population. In a population of size N, the subpopulation will be of size N/2 (Peterson, 1997; Vega, 2001).

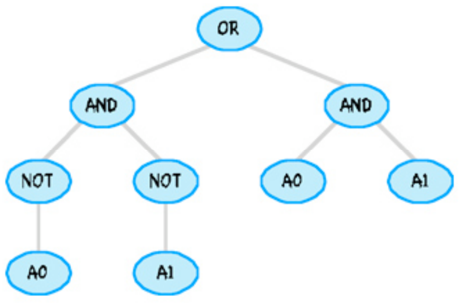


Fig.19 – Example of an Individual. Adaptation from fig.11 of [18]

Individuals may contain arithmetic operations (e.g. =, -, x, /), mathematical functions (e.g. sine, exponential, logarithms), Boolean operations (e.g. AND, OR, NOT), logical operators (if-then-else, etc), iterative operators (while-do, etc), recursive functions, among others. It all depends on the problem in cause. These will be the functions defining the body of the program (Koza, 1990, 1992, 2001, 2007; Coello, 2007; Pappa & Freitas, 2006; Vega, 2001).

The terminals can be constant atomic arguments (state variables of a system for example), constant atomic values (0, 1, etc) and even other atomic entities, like functions with no arguments (Koza, 1990, 1992, 2001, 2007; Pappa & Freitas, 2006; Vega, 2001).

When defining the set of functions and terminals for a specific problem, it's necessary to satisfy the conditions of closure and sufficiency. Closure requires that any function used can accept any value and data type returned by any other function or terminal (Peterson, 1997; Pappa & Freitas, 2006), which guarantees the validity of the program. In order to do this it might be necessary to use functions that return default values (for example, if a value is divided by zero, the function returns a default value instead of an unidentified situation).

Sufficiency requires that the given problem solution exists within the search space by using functions and terminals (Peterson, 1997; Pappa & Freitas, 2006).

2.3.2.2 Fitness

The fitness function (objective function) evaluates the aptitude of survival of the individuals in the population and it is an indispensable tool for the Genetic Programming to work. The fitness function analyses how much a potential solution can satisfy the problem (Peterson, 1997; Koza, 1990, 1992, 2001, 2007; Pappa & Freitas, 2006).

The fitness function varies according to the problem at hand, since most optimal solutions have to be calculated with different formula. But all fitness functions try to determine which individuals of the population of possible solutions is closer to the optimal result. For example, on a broom balancing problem, the smaller the time it takes to balance the broom and the longer it stays balanced, the better the fitness of that individual (Koza, 1990, 1992, 2001, 2007; Vega, 2001).

It's important that the value the fitness function returns gives enough information of how fit the individual is. For example, if it only returns either 0 or 1, the performance of the individual can't be evaluated correctly, especially in later cases where the fitness values are higher and therefore more similar (Koza, 1990, 1992, 2001, 2007).

However, if the problem requires a subjective judgement, like evaluating the aesthetic quality of an image, then it may be very difficult to define the fitness. To solve this problem an outside source, as a user, replaces the fitness function and determines which members of the population are more suitable for reproduction. In some cases the selection is done directly, while others filter part of the population beforehand (Peterson, 1997).

This interactive evolution system is best for small populations. There are many limitations the human comprehension forces upon this, such as inability to pick minor variations in the representation or not being able to deal with many individuals at the same time (Peterson, 1997). Usually the magnitude of individuals to be picked is less that 25, which reduces the diversity of the population greatly. Because of the sall population size high mutation rates are often used to compensate the lack of diversity (Peterson, 1997).

2.3.2.3 Reproduction

This genetic operator combines the information contained by parents to create an offspring. In the natural world the same happens when a chromosome pair recombines to generate a new DNA strand (Peterson, 1997; Coello, 2007).

Before the parents information is recombined, they are both selected based on the same fitness criteria. Depending on the approach to solve the problem, we can either have an asexual or sexual reproduction and reselection might or might not be allowed (Peterson 1997; Koza, 1990, 1992, 2001, 2007).

An asexual reproduction only involves one parent to generate an offspring. Several individuals of the population are selected based on their fitness (which evaluates their performance) and the probability of reproduction (which defines how many can reproduce) (Peterson, 1997).

The offspring that will form the new population are clones of their parents. As a result of that, reselection is allowed in this case (Peterson, 1997; Vega, 2001).

Sexual reproduction, also known as Crossover, recombines the genetic information of the parents to generate an offspring. The parents are selected in the same fashion as in the asexual reproduction, based on their fitness and crossover probability (Peterson 1997; Koza, 1990, 1992, 2001, 2007; Coello, 2007; Vega, 2001).

Usually in Genetic Programming the crossover is done by using the tree nature the individuals possess and numbering the tree nodes by their reading order (Coello, 2007; Vega, 2001). First, a point is chosen randomly in both trees. Second, the subtrees rooted on those points are selected and detached from the parent trees. Third and finally, the subtrees are switched between the parents, generating a pair of offspring (Coello, 2007; Peterson 1997; Koza, 1990, 1992, 2001, 2007; Vega, 2001).

Note that the resulting offspring from any crossover points are always valid programs, because their parts are taken from the parents, which were also valid expressions (Koza, 1990, 1992, 2001, 2007). The reason for this to happen is because of the closure and sufficiency requirements imposed when the population was first created.

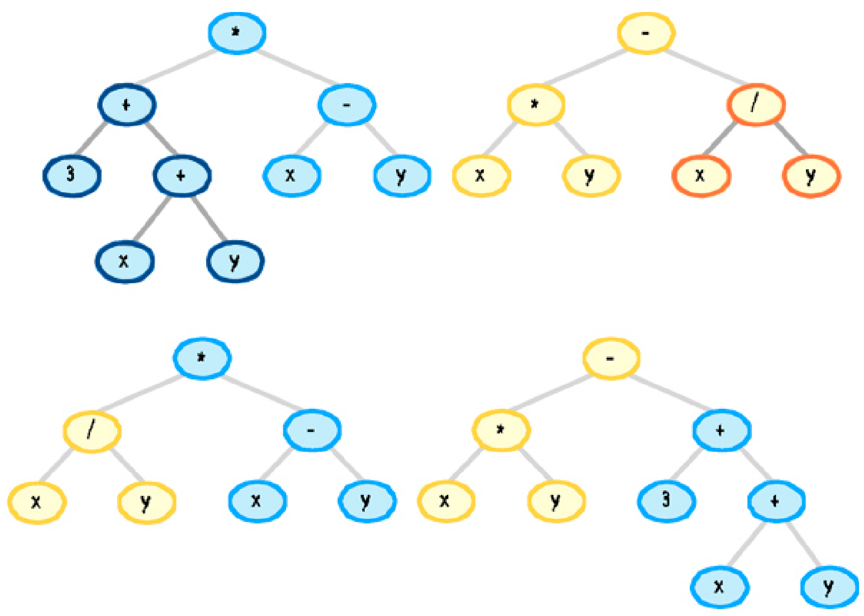


Fig.20 – Crossover example. Adaptation from fig.2.6 of Peterson (1997)

It's also possible for the root of the trees to be selected, either in one or both parents. If only one root is selected, the entire parent will be copied onto the second, while the subtree of the second parent will become the full tree of the first one (Koza, 1990, 1992, 2001, 2007). If the roots of both parents are selected as crossover points, the children will be copies of their parents, as in a reproduction and not a crossover (Koza, 1990, 1992, 2001, 2007).

In the case of a terminal and a root being selected as crossover points, this often increases the size of one of the trees dramatically (Koza, 1990, 1992, 2001, 2007; Coello, 2007). In order to prevent memory problems, Genetic Program usually imposes a limit on the maximum depth of a tree.

Compared to the Crossover operator in Genetic Algorithms, there are two main differences from the one used in Genetic Programming. First, in standard Genetic Algorithms offsprings have the same size as their parents, no matter the number of generations. In Genetic Programming there's a big chance of different sizes and shapes to be formed because of the random crossover points (Peterson, 1997; Coello, 2007).

Second, if reselection is allowed, and individual with high fitness may be chosen to act as both parents, incestuous offspring result. In Genetic Algorithms an incestuous crossover degrades the quality of the offspring to an asexual reproduction. In Genetic Programming the two offspring will likely be different, unless the same crossover points are chosen in both parents (Peterson, 1997; Koza, 1990, 1992, 2001, 2007; Coello, 2007).

2.3.2.4 Mutation

Occasionally a random change will occur during the genes recombination, resulting on a slight mutation that brings variation and new possibilities in the process of evolution.

In Genetic Programming, mutation works by randomly picking a point in the tree and replacing it with a new randomly generated subtree. A probability based on the tree's depth can prevent excessive terminal swapping or encourage bigger or smaller tress to be produced Peterson, 1997; Coello, 2007; Vega, 2001).

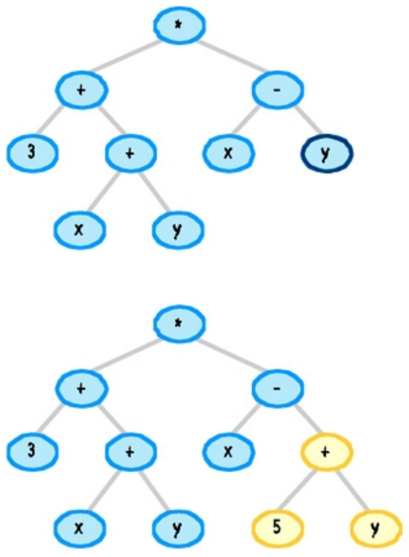


Fig.21 – Mutation example. Adaptation from fig.2.7 of Peterson (1997)

Mutation is very good to add variety and keep the population from becoming stagnated, but it might damage or render a program non functional (Peterson, 1997). It's possible to protect a certain subtree, known as a good building block, by using encapsulation. The subtree is replaced by a symbolic name that points to its real location (Coello, 2007).

Editing is another operator which helps protecting the tree. Like mutation it replaces certain subtrees with new information, but instead of using a new random subtree, it cleans up the existing one by replacing a constant valued subtree with its corresponding value. For example, (1 (+ 1)) would be replaced by the terminal 2 (Peterson, 1997). This helps avoiding waste of memory and unnecessary depth of the tree, but this parsimony can be harmful to the diversity of the population (Peterson, 1997).

2.3.2.5 Termination Criteria

The evolutionary process ends when the population reaches the solution to the problem, or gets as close to it as possible. There are many different criteria that can be used to define when the whole process terminates, including number of generations, lack of increase of fitness within the population after a certain number of generations, etc (Koza, 1990, 1992, 2001, 2007).

Once that step is reached, the best individual of the whole population is considered the optimal solution to the given problem (Koza, 1990, 1992, 2001, 2007).

There are some cases where outside factors might force the program to end before the optimal solution is found, such as: time, resources and funds.

3 System Architecture

This work intends to combine L-Systems with Genetic Programming to produce new plants which are evolved forms of their parents. The developed application takes the parents chosen by the user and draws the given L-System before receiving order to breed the plants by the user and then draw the children through the same system.

Though the graphical elements of the interface were drawn in Flash, the whole computation and plants drawing was solely produced by the code.

In this chapter the conceptual model is presented, which describes the decisions taken based on the problem. It also describes the project goals, the pseudo-code developed for each of the parts, and the system implementation, which describes how the program was implemented based on the pseudo-code as well as the adaptations that had to be made.

3.1 Conceptual Model

As the purpose of this project is to answer the problem identified in the first chapter, it’s best if we remind ourselves of the objectives we set out to fulfill:

1. Identify a method used to build plants in the computer, or that can be adapted to work in a computer;
2. Determine how to use the Genetic Programming to evolve those plants;
3. Adapt the developed approach for Actionscript;
4. Allow some level of control to the user to select and adjust the plants.

Knowing these milestones and taking into consideration the information presented in the second chapter, we can start defining the solution. This section focuses on answering the first and second points.

The Actionscript is done later in the implementation because we first need a solid idea on how the L-system has to be implemented and what will the Genetic Programming take from it in order to optimize the plants.

3.1.1 The L-System

Before we can evolve a plant, we need a plant to evolve. That's why the first point was to find a method that could represent plants and be translated into the computer. The L-system was picked from the different methods because of the reasons stated in the previous chapters.

From the different types of L-systems, the DOL-system was selected to describe the process of development of plants in Actionscript. The reason behind this method selection is because it allows an easier control of the programming stages and resulting validation.

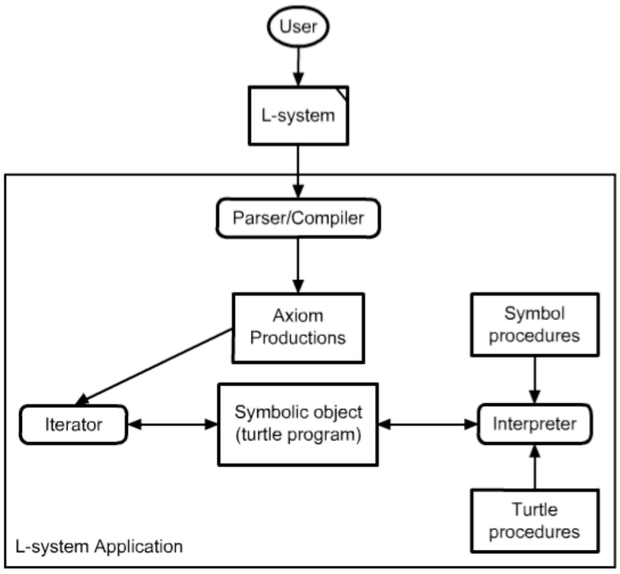


Fig.22 – L-system architecture. Adaptation from fig.1 From Noser et al., 2001

Figure 22 illustrates how the common L-system architecture works. Taking the L-system plant provided by the user, the Parser prepares it to be interpreted by the computer, resulting in a valid axiom and productions, syntax wise. With this verification done, the computer can run them through several iterations, and on each one send the resulting word to be interpreted by the Turtle Program. In this drawing stage, both the symbol alphabet of that plant and the turtle procedures are taken by the Interpreter to decipher the commands.

Taking this architecture as the model to be followed, we can divide the process to obtain an L-system into three steps:

- Syntax Validation;
- Production;
- Interpretation.

3.1.1.1 Syntax Validation

The Validation step acts like the Parser described before. It takes a given plant, defined by the alphabet, axiom, the production rules and hidden parameters, and validates them. This is done by eliminating any empty spaces or unwanted symbols so the computer doesn’t come across with any unexpected characters in the next steps.

```
V                       // Alphabet
W = "X"                 // Axiom
P = "X->F[+X][-X]FX"    // Production

While ( i  <  length of P ) {
        Read character by character
        If (character == empty space || !=  from turtle or V
symbol)
                Then remove from P
        i ++
}
```

Fig.23 – Example of Validation step

Figure 23 presents an example of a simple validation. The cycle checks the string, this case the production P, character by character and removes any symbol that doesn't belong in the alphabet of that plant or isn't a turtle command. More complete validations can include checking if all the opening brackets have their closing counterparts or see if there are rotations canceling each other.

This verification is more of a precaution than a real component of the L-system method. It only needs to be done once at the beginning of the program since the next steps don't generate invalid symbols or insert blank spaces between the characters, unless if badly coded.

3.1.1.2 Production

The Production step generates a word on each iteration before it's sent to be interpreted by the Turtle Program. This process is done by taking the axiom and applying the productions to replace any symbols that match their predecessors with the symbols contained in their successors, generating the word that will be the new axiom of the next iteration.

```
W = X                    // Axiom
P = X->F[+X][-X]FX       // Production
N                        // Number of Iterations

Split P into two strings where -> occurs
String1 = P's predecessor
String2 = P's successor

While ( i  <  N ) {
    W= Replace every predecessor with successor in the W
    i ++
}
```

Fig.24 – Example of Production step

As shown in the example in Figure 24, before the replacement was done, the predecessor and the successor of each production were separated and stored as separate strings. It's not mandatory to perform this separation, but instead of having a function locating the predecessor and the successor every time we need to work with the production, we only have to perform this operation once. So after we have the two strings per production, on each iteration we check the current axiom and try to find symbols matching the predecessors in it. When that occurs, we replace that match with the corresponding successor, creating a new word in result. A variation to this step is to do all iterations at once and send the final result to be drawn, instead of doing it on each iteration. To accomplish this, the drawing function is called outside of the cycle.

3.1.1.3 Interpretation

The Interpretation step is the same as the Turtle Program in the diagram. It takes the produced word, the symbols in the plant's alphabet, the turtle commands and the hidden parameters, such as branch angle and unit length, to send them to the Interpreter and draw them on screen.

```
Word            // Produced word
Alpha           // Angle
Length          // Unit Length
…
Remove empty symbols
While ( i  <  Word length ) {
    Read Word character by character
    Switch (character)
        Case F:
            Draw forward
        Case +:
            Rotate counter-clockwise by Alpha degrees
        Case -:
            Rotate clockwise by Alpha degrees
        Case [:
            Store current coordinates and angle
        Case ]:
            Restore coordinates and angle
    i ++
}
```

Fig.25 – Example of Interpretation step

As it can be seen in Figure 25, because this is a simulation of the Turtle Interpretation, it's important to remove any non-drawing symbols in the word before it's sent to be interpreted. Once that's done it reads the word character by character and compares it with the commands available, performing the corresponding task.

Note that this program only deals with 2D L-systems, because of the Flash limitations, which was mentioned in the first chapter. To simulate a 3D L-system there would one be a difference in this last step, which would include the rest of the commands on the list.

w: A

$\mathbf{P_1}$**:** $A \rightarrow B[+A]$

$\mathbf{P_2}$**:** $B \rightarrow AA$

A

B[+A]

AA[+B[+A]]

…

Fig.26 – Example of a plant that would need to be converted

In some cases, like in the one above (Figure 26), it would be necessary to convert at least one of the symbols into the drawing command F, either by using a function or doing it before this L-system was given to the program. This is because if the Interpreter was to use this word, all we would get would be a blank screen.

This is an unusual, but not impossible, occurring situation. The best way to get around it would be including a validation in the Syntax Validation step, to either warn the missing symbol or ask to replace another non turtle command one by it.

3.1.2 Genetic Programming

According to the explanation given in the second chapter, the various applications of Genetic Programming used the same operators. One of the most important topics is the decisions made on individuals and the fitness evaluation, since they will affect the whole process on how the plants result.

3.1.2.1 Initial Population

Because the goal is to optimize the existing plants in order to generate children, the decision was made to use the produced word instead of the productions as the individual, reason why Genetic Programming was chosen over Genetic Algorithms.

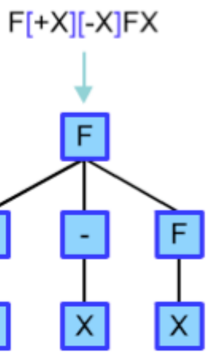


Fig.27 – Corresponding tree structure of a L-system word

As you can see from Figure 27, the L-system word easily translates into a tree representation, with each symbol being a node. This can also be translated as the individual genotype, the information contained in the gene, while the hidden parameters can be considered as the phenotypes, the physical attributes of the individual. As this type of grammar is counter-intuitive and hard to predict the outcomes of the different productions, as it can be seen in Figure 28, it was decided to use pre-developed plants for the initial population instead of randomly generated ones to ensure the validity of the individuals.

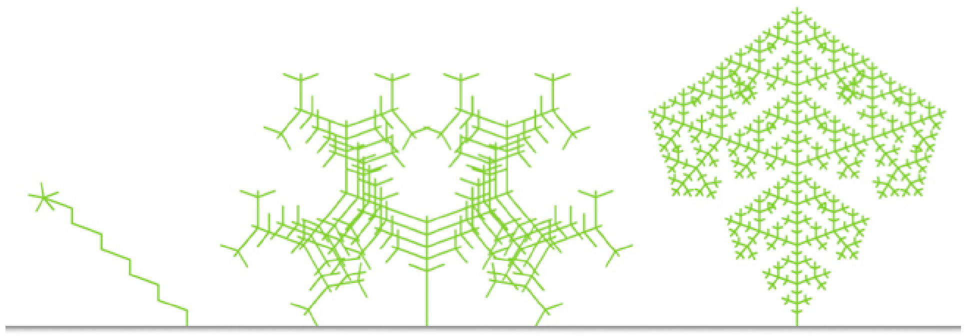


Fig.28– Examples of non-plant L-systems

3.1.2.2 Fitness Function

We're not aiming to simulate plants existing in real life, but to generate structures that look like plants. This is because this work isn't a scientific research on a certain species. We want to give room for creativity and to explore solutions that we likely wouldn't come across in real life. Because we're working with unexpected results and basing evaluation on the aesthetic appearance of the plant, the fitness will be decided by the user rather than by the computer. This evaluation process limits the amount of results we can present at a time, due to the human component.

3.1.2.3 Reproduction

Sexual reproduction, which involves two parents instead of just one, is done by randomly picking a point on each parent and exchanging the resulting subtrees between them. The reason why two parents are used instead of one, is the bigger diversity it brings to the end result. Also, the decision of having a user acting as the fitness function influenced this. If by using two parents the children are more likely to be different, then it helps having resulting offspring with bigger differences between themselves, which is important when working with a small population pool.

```
Plant1                                      // First parent word
Plant2                                      // Second parent word

Position1 = Randomly pick a node on Plant1
Position2 = Randomly pick a node on Plant2

Child1 = In Position1 of Plant1 put subtree taken from Position2 of Plant2
Child2 = In Position2 of Plant2 put subtree taken from Position1 of Plant1
```

Fig.29– Reproduction step

As you can see in Figure 29, the resulting offspring come from the exchange of subtrees from the two parents. However, because we're dealing with the L-system grammar the picking of the nodes has to be restricted.

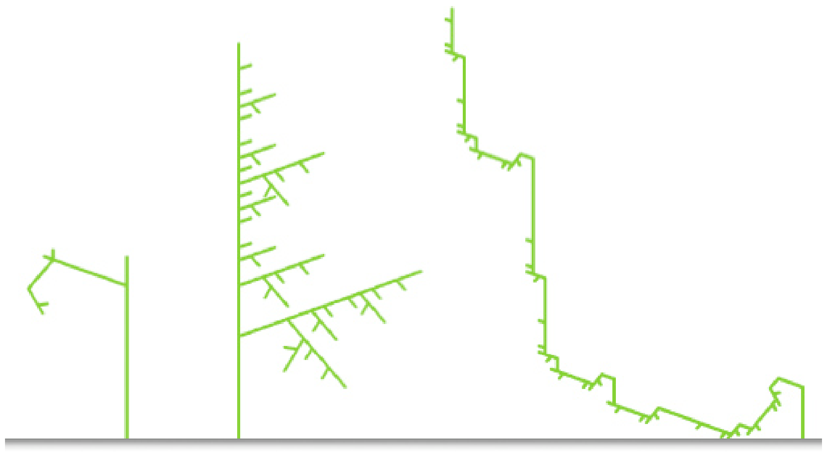


Fig.30– Examples of bad replacement of the nodes

The Figure 30 shows what happens when an extra bracket or rotation occurs. Even if none of these extreme cases happens, there's still a good chance of the resulting L-system not resulting in a plant structure.

So, the selection of nodes is restricted to the branches of each plant. Besides preventing the occurrences just shown, they are proven to be stable sections of commands since they work well on each plant. Of course, bad crossovers can still occur, but the probability is much smaller.

3.1.2.4 Mutation

This operator isn't used in this work, because it relies on randomly generated trees, the probability to generate bad lines of commands is too great, as explained in the previous points.

It would be possible to set an array of branches however, taken from produced plants, but that would involve collecting and testing their validity beforehand which was too time consuming for this project.

The inclusion of mutation isn't mandatory though, as explained in the second chapter. It would only be used if the crossover couldn't provide enough diversity in the population in order for the user to tell the difference which is not the case.

4 System Implementation

Having the concept of how the L-system and the Genetic Programming will be implemented, the next goal can be addressed: implementing these methods in Actionscript.

Although Flash is an excellent tool when it comes to interaction and graphic development, we have to keep in mind it has some limitations when it comes to code, since usually most people don't use it for its coding language. Besides, it can get very slow when processing heavy calculations such as large number of iterations involved in producing the L-system word. There are some slight differences between Actionscript and other languages like C++ or Java, which can give very confusing results.

One of them is the variable declaration. A variable once declared can be used in any part of the code, even if it's outside the function where it's declared. This means sometimes the variable will "drag" the content it has previously and not have replaced properly when used in a different section of the code, causing unexpected results like a word far bigger than it should be. To get around this it is necessary to "empty" the variable before it is used or declaring it again, which apparently had the same effect.

These type of mishaps are a fairly common occurrence, especially when one is learning a new programming language. Being more of a minor nuisances than a major hindrances, their occurrence isn't described in the explanation of the code implementation.

4.1 Versions Introduction

Instead of producing an ongoing program where each new section of the conceptual model would be implemented, it was decided to work on separate versions of the same application.

There were several reasons behind this decision, mainly:

- The constant learning of new Actionscript functions and techniques as the implementation was taking place;
- The testing of different sections of the conceptual model separately, to reduce the chance of faulty results;
- To ease the code optimization and performance of the application.

Since the Genetic Programming needs a population of plants to evolve, the 1st Version focuses on the implementation of the L-System and the Draw function.

The 2nd Version focuses on the optimization of the prior L-System code, correcting some minor mistakes and eliminating useless lines of code.

The 3rd Version focuses in the Genetic Programming code and the implementation of its operators, but only using a section of the L-System code to test the results.

The 4th Version is the first to implement both the L-System and Genetic Programming codes simultaneously, as well as providing a basic interface for the user.

The 5th Version uses a larger population of plants and allows the user to view more than a single generation, as well as attributing them with a fitness.

The 6th and final version optimizes the code used in the previous one, by removing unnecessary duplications and compressing the produced word, and improves the design of the interface.

4.2 Version 1

The first version of the project only focused on getting the L-system to work and be drawn on the screen. It didn't have any interaction and simply used a plant hardcoded into the program with all the corresponding hidden parameters.

Its purpose was to identify which parameters were exactly necessary and which changes would have to be added to the conceptual scheme in order for the program to work.

4.2.1 Syntax Validation

Because the plant was hardcoded, this step was unnecessary. We were using a plant that had already been validated by someone else, one of the many examples of successful L-system plants people use out there on the internet to illustrate how the grammar works.

No invalid symbols were used, and all blank spaces had been removed when building the solution, as it can be seen in Figure 31.

V = { X, F, +, -, [,] }	// Alphabet of the plant
W = X	// Axiom
P_1 = X->F[+X][-X]FX	// First Production
P_2 = F->FF	// Second Production
Angle = 70	// Branch Angle
Steps = 4	// Iterations
Length = 4	// Unit Length

Fig.31– L-system plant used

4.2.2 Production

Though this version was more of a test on how to get the L-system method to work, attempts were made from the beginning to make the system as universal as possible. Functions were used instead of a big block of sequence of code with many unnecessary duplications in the middle.

So taking the L-system grammar, the only element that can have multiple members is the productions. In order to ensure the L-System function can receive any type of plant, the productions were stored in an array before being sent, as seen in Figure 32.

```
// Productions

var p1 = “X->F[+X][-X]FX”

var p2 = “F->FF”

// Set productions into an array

var p = new Array (p1,p2);
```

Fig.32– Storing the productions into an array

As seen before, the first part in the Productions step is to separate the predecessor from the successor in each of the productions, but because we’re dealing with an array of productions of unknown size, we can’t define a pair of string variables for every single case.

Instead we can use two arrays, one for the predecessors and another for the successors, with the same position in each corresponding to the same production. This is easier to handle, and particularly easier to debug, rather than using one array to store both.

```
 var vect1 = new Array();         // Predecessor array

 var vect2 = new Array();         // Successor array



while ( i < this.p.length) {

        var stringVect = this.p[i].split("->");

                vect1[i] = stringVect[0];
        vect2[i] = stringVect[1];

                ++i;
}
```

Fig.33– Separating predecessor from successor

As shown in Figure 33, in this cycle that lasts while i is less than the length of the array p, the program takes each string within it and where the -> occurs, the string is split into two and stored in the array stringVect, then stores the predecessor and the successor in each respective array.

The split is a method in Actionscript that takes a string and returns an array containing two strings, separated where the delimiter string occurred. Except in the case of an empty string, it splits every single character within that string. There's also an optional parameter that limits the number of items to be put in the array, but it's unnecessary in this case.

With the predecessors and successors ready, we can then start replacing the occurrences in the word.

```
String.prototype.replace = function (from, to) {

        return this.split(from).join(to);
};
 // ---------------------------------------------------

while ( j <  steps) {

        var k = 0;

        while ( k <  vect1.length) {

                var val1 = vect1[k];
                var val2 = vect2[k];

                this.words = this.words.replace (val1, val2);

                ++k;
        }

        ++j;
}
```

Fig.34– Replacing symbols in the word

As you can see in Figure 34, on each iteration the program goes through all the different predecessors and checks for any correspondences by sending them to the function *replace* shown above. If that function finds a correspondence to the given string, it splits the word in two and then joins it back together, inserting the successor in between those strings before it's returned and replaces the old word with the new one.

The join is another Actionscript method, which takes an array and converts it into a string, inserting the given character or string in the parameter between the elements of the array, concatenating them.

4.2. 3 Interpretation

With the L-system word ready to be interpreted and knowing which symbols were used from the alphabet, all we need are the parameters necessary to draw the plant. However, this is where the problem really starts. The information that came with the axiom and the productions isn’t enough. More parameters are necessary in order to reproduce it correctly on screen.

The initial coordinates were the first ones to be added. Because we want the plant to grow upwards, like normal plants do, we picked the middle of the stage as the x coordinate and the bottom of the stage as the y coordinate.

The next ones were the properties of the line to be drawn. In Flash when drawing figures out of pure code is first necessary to declare an empty movie clip to draw them in. Only then we can change the properties of the movie clip, as seen below in Figure 35.

```
px = 300;    // X coordinate
py = 400;    //Y coordinate



this.LS = _root.createEmptyMovieClip ("drawClip", 100);

 this.LS.lineStyle (2, 0x33CC00, 100);

 this.LS.moveTo (this.px, this.py);
```

Fig.35–Changing the properties of the movie clip

The first line creates the empty movie clip as a child of the existing one (the stage), giving it a name and setting its depth on the screen. The lineStyle allows us to determine the thickness, color and opacity of the lines drawn within that movie clip, respectively. And finally the moveTo moves the current drawing condition to the given coordinates.

The last parameter that had to be added was the angle of the plant. This was detected afterwards; when drawing the plant, the line was first moving horizontally before the rotations changed its angle. To solve this problem it was necessary to rotate the plant before it was drawn, in this case 90 degrees so it would grow vertically instead.

There were two different attempts made at reading the word before sending it to be interpreted by the different list of commands. The first one was to have the plant drawn all at once, while the second shows the sequence of how the word was read and how that reflected in the appearance of the plant.

```
LSystem.prototype.render = function() {
        var i = 0;

        for (; ; ) {
                if (i >= this.words.length) {
                        return;
                }
                var charPos = this.words.charAt (i);

                this.renderInstruction (charPos);

                ++i;
        }
};
```

Fig.36–Drawing the plant all at once

As you can see in Figure 36, this function there's an infinite cycle where the word is read character by character and sent to be rendered. Actually, the cycle isn't really infinite, because once the counter exceeds the length of the word, it calls the function return and automatically jumps out of the cycle, terminating it.

Using a loop that would check the condition would probably be best. This was just done as an experiment.

```
LSystem.prototype.renderSteps = function() {

        var i = 0;
        var obj = this;


        this.LS.onEnterFrame = function() {

                var charVal = obj.words.charAt (i);

                obj.renderInstruction (charVal);

                if (i++ >= obj.words.length-1) {

                        delete this.onEnterFrame;
                }
        };
};
```

Fig.37–Drawing the plant character by character

The reason why this function was used instead of the other in this version of the program was to check how the plant was built exactly and to ensure the whole process was occurring properly.

In order to do that, we used a special function existing in Actionscript to control the movie clip. OnEnterFrame is an event handler that is invoked on the beginning of each frame before any of the other actions are preformed. So as each frame occurs, the function reads a character of the word and sends it to be rendered, but when there are no longer more words to be rendered, this onEnterFrame function is deleted.

The speed of how fast the plant is drawn varies with the amount of frames per second defined. The default in Flash is 12 fps, but it can range from 0.01 to 120.This can pose a problem however. Though it’s quite useful for debugging purposes, it takes some time to draw the plant, especially large ones, no matter the speed set. Because the final solution will deal with not one, but several plants at the same time, we can’t have the user wait for one plant to be drawn before he or she can visualize the next.

With all the parameters defined and the rendering function sending the individual characters to be interpreted, all we need to do is send the characters through the command list and implement the different actions.

Even so, before the characters can be analyzed, even before they were sent by the render function, the non-drawing symbols had to be removed from the produced word. In this case it was done right after the production of the word. The only reason this point wasn’t addressed before was because it concerns the drawing stage.

```
this.words = this.words.replace ("X", "");
```

Fig.38–Deletion of the non-drawing symbols in this plant

To do the deletion it calls the replace function and replaces all the non-drawing symbols with an empty string (not a blank space), as it can be seen in Figure 38. Considering we’re working with a hardcoded plant, we already know which are the empty symbols.

In the case of receiving an unknown plant from the user, then we would have to ask which were drawing symbols and which weren't. Otherwise, we would have to restrict the symbols that could be used.

```
LSystem.prototype.renderInstruction = function (instr) {

    if (instr === "F") {
       this.Draw (px, py, angleInit, segmentLength);
       return;
    }
    else {
       if (instr === "-") {
               this.angleInit = this.angleInit + this.angleInc;
               return;
              }
       else {
               if (instr === "+") {
                       this.angleInit = this.angleInit - this.angleInc;
                       return;
               }
               else {
                       if (instr === "[") {
                               var flag = new Cursor (this.px, this.py, this.angleInit);
                               this.stack.push (flag);
                               return;
                       }
                       else {
                               if (instr === "]") {
                                       flag = this.stack.pop();
                                       this.px = flag.px;
                                       this.py = flag.py;
                                       this.angleInit = flag.angleInit;
                                       this.LS.moveTo(this.px, this.py);
                                       return;
                               }
                       }
               }
       }
    }
    trace("unknown command: "+instr);
};
```

Fig.39–The available commands and their actions

As shown in Figure 39, the function receives the character sent by the rendering function and travels through the several if statements until either a match is found or it announces it as an unknown command. This figure only contains the commands used for this specific plant. With different grammars it is necessary to introduce more conditions.

Like in the Turtle Interpretation, F corresponds to drawing a line forward. In order to do that, it sends the current coordinates, the angle of the branches and the current angle of the plant to the draw function. This will be explained further below.

When a rotation occurs, either clockwise or counter-clockwise, the current angle of the plant is updated by adding, or subtracting, the angle of the branch to the previous angle of the plant.

As for the brackets, when the opening bracket occurs it sends the current coordinates and the angle of the plant to be copied and then pushed into an array. When the closing bracket occurs, the last values are popped from the array and replace the current coordinates and plant angle to restore them to the original position before entering that branch.

```
LSystem.prototype.Draw = function (px, py, angleInit, segmentLength) {

        this.px = this.px + this.segmentLength * Math.cos (this.angleInit);
        this.py = this.py + this.segmentLength * Math.sin (this.angleInit);

        this.LS.lineTo (this.px, this.py);
};
```

Fig.40–Drawing function

When drawing the plant we have to keep in mind that the rotations change the direction of the line and that affects the position of the final coordinates. Because we have to keep the same length to all units of the plant and Flash can't calculate directly the final coordinates just by giving the initial coordinate pair, unit length and amount of rotation, we have to calculate the projection of the segment on each of the axis in order to find the position the line has to move to (Figure 40).

This is done by applying the rules of trigonometry to calculate exactly how much each unit will measure on each axis, depending on the rotation applied to them, and added to the current coordinates to find out where the final ones will be. After that, all we need to do is call the lineTo function which draws a line from where the drawing cursor last was to the new position.

```
Cursor = function (px, py, angleInit) {

    this.px = px;
    this.py = py;
    this.angleInit = angleInit;
};
```

Fig.41–Cursor function

Data structures in Flash don't work in the same way they do in C++ or Java. While in C we would have to declare a structure outside of the program sequence and define all the parameters inside, in Flash, it is done as shown in Figure 41 and declared as a regular function.

The reason why it was used is because it makes it easier to store all the information concerning one bracket in a single position of the array instead of pushing and popping three all the time.

4.3 Version 2

The second version of the program still focused exclusively on the L-systems, this time cleaning up unnecessary code used on the first version and correcting some minor mistakes that could have some negative effects when including the Genetic Programming and producing the new plants.

4.3.1 Syntax Validation

Hardcoded plants were still used, so again, this step was seen as unnecessary since there had been some preparation prior to their inclusion in the code. The only difference this time was that two new plants were included to test the effectiveness of the code and to later be used as parents in the Genetic Programming.

```
// Shared Variables

var px = Stage.width / 2; // X coordinate

var py = Stage.height;              // Y coordinate

var plantAngle = Radians(-90);      // Plant Angle

var steps = 4;                      // Iterations

var lenght = 3;                     // Unit Length

var color = 0x33CC00;               // Color

//Plant 1

var w1 = "X";                                   // Axiom

var p1 = "X->F[+X][-X]FX";                      // First Production

var p2 = "F->FF";                               // Second Production

var productions1 = new Array(p1, p2);           // Production Array

var angle1 = Radians(70);                       // Branch Angle

//Plant 2

var w2 = "F";                                   // Axiom

var p1 = "F->FF-[-F+F+F]+[+F-F-F]";             // Production

var productions2 = new Array(p1);               // Production Array

var angle2 = Radians(20);                       // Branch Angle

//Plant 3

var w3 = "X";                                   // Axiom

var p1 = "X->F-[[X]+X]+F[+FX]-X";               // First Production

var p2 = "F->FF";                               // Second Production

var productions3 = new Array(p1, p2);           // Production Array

var angle3 = Radians(25);                       // Branch Angle
```

Fig.42–Different L-system plants used

Many of the variables which define the plants were shared, as seen in Figure 42, while the ones that determine their individual appearance were defined separately. The reason for this was to compare how each behaved when having the same number of iterations and unit length (the other shared values pay little importance in their appearance).

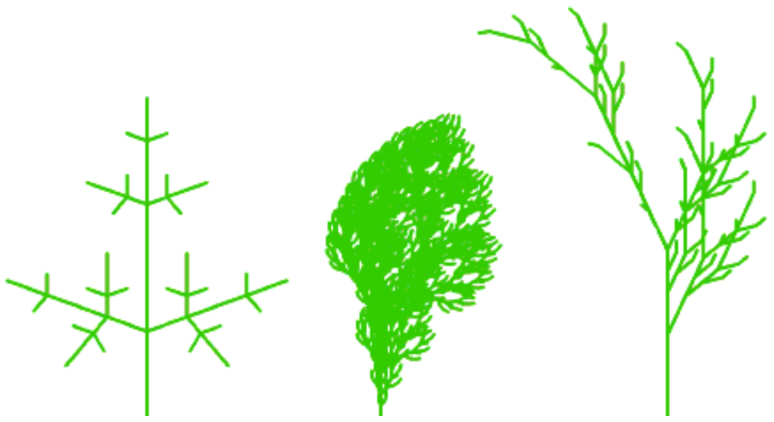


Fig.43–Resulting plants

As you can see from Figure 43, the same parameters can result in quite different plants, depending of the axiom and productions used. The first plant would need more iterations, while the second would need less. The lengths would also have to be adjusted, if we want the plants to have roughly the same size.

```
Radians = function (degrees) {

        return degrees * Math.PI / 180;
}
```

Fig.44–Radians function

There was one mistake that wasn't detected on the first version that caused some issues. When working with angles, Flash uses radians instead of degrees. The function from Figure 44 takes the angle we want to use and converts it into radians by applying the conversion formula:

$$\alpha \times (\prod \div 180°)$$

4.3.2 Production

In the Production stage there was only one minor change done. This was eliminating unnecessary code in the L-system function, which deals the separation of the predecessors from the successors and the constant replacement of the word during the several iterations (Figure 45).

```
while (i < this.productions.length) {

        var stringVect = this.productions[i].split("->");

        vect1[i] = stringVect[0];
        vect2[i] = stringVect[1];
        ++i;
}

while (j < steps) {
        i = 0;

        while (i < vect1.length) {

                this.words = this.words.replace(vect1[i], vect2[i]);
                ++i;
        }
        ++j;
}
```

Fig.45–Production step within the L-System function

4.3.3 Interpretation

The major change done in this section of the code was to replace the series of if conditions by a switch statement, which is much more functional (Figure 46). Besides being easier to add new commands if necessary, it is less prone to mistakes like the large amount of brackets for example.

```
LSystem.prototype.renderInstruction = function(instr) {


    switch (instr){
            case "F":

            this.Draw(px, py, angleInit, unitLength);
    return;
            break;

            case "-":

            this.angleInit = this.angleInit + this.angleRot;
    return;
            break;

            case "+":

            this.angleInit = this.angleInit - this.angleRot;
    return;
            break;

            case "[":

            var flag = new TurtleCursor (this.px, this.py, this.angleInit);
            this.stack.push(flag);
    return;
            break;

            case "]":

            flag = this.stack.pop();
            this.px = flag.px;
            this.py = flag.py;
            this.angleInit = flag.angleInit;
            this.LS.moveTo(this.px, this.py);
    return;
            break;

            default:

            trace("unknown command: "+instr);
            break;
    }
}
```

Fig.46–The RenderInstruction function using the switch instead of if statements

4.4 Version 3

This version is more of a separate program to prepare and test the Genetic Programming section of the solution. Using what was developed before for the L-system, it takes the produced word and extracts all the branches within it and then selects one randomly to be exchanged with another plant.

4.4.1 Individual

Because this whole version is meant to test the Genetic Programming reproduction stage, we're only working with one individual and not a whole population.

To generate the individual we took a plant used previously in the other versions and generated the word the same way it was done in the production step. The resulting word was as shown below in Figure 47.

```
var w1 = "X";                               // Axiom

var p1 = "X->F[+X][-X]FX";                  // First Production

var p2 = "F->FF";                           // Second Production

var productions1 = new Array(p1, p2); // Production Array

var steps = 4;                              // Iterations


Word: FFFFFFFF[+FFFF[+FF[+X][-X]FFX][-FF[+X][-X]FFX]FFFFFF[+X][-
X]FFX][-FFFF[+FF[+X][-X]FFX][-FF[+X][-X]FFX]FFFFFF[+X][-
X]FFX]FFFFFFFFFFFF[+FF[+X][-X]FFX][-FF[+X][-X]FFX]FFFFFF[+X][-
X]FFX
```

Fig.47–Plant and resulting word after Production step

4.4.2 Branches

Once we have the word to work with, we can start taking all the branches that occur within it and store them in an array, along with the initial and final positions where they occur, so that when the exchange takes place we know exactly which section of the plant will be replaced.

```
while (i < words.length){

        if (words.charAt (i) == "["){
                position.push (i);
        }

        if (words.charAt(i) == "]"){
                j = position.pop () + 1;

                branches.push (j);
                branches.push (i);

                while (j < i){
                        subtree = subtree + words.charAt (j);
                        j ++;
                }
                branches.push (subtree);
                subtree = "";
        }
        i++;
}
```

Fig.48–Extraction of the branches within the plant

As it can be seen in Figure 48, whenever an opening bracket occurs, the position is pushed into the position array. When the closing bracket occurs, the position of the last opening bracket is taken out. The reason the value is incremented by one is to have the opening bracket start in the next position, so it doesn't store the bracket itself and cause a missing bracket when the replacement takes place.

After storing the initial and final positions of that branch in the branch array, all the characters occurring in that interval are copied into a variable and stored in the array as well, as exemplified in Figure 49. After that, the string variable is cleared to prevent the next branch to be added to the previously copied one and cause an error.

```
Branches: 19,21,+X,23,25,-X,15,29,+FF[+X][-X]FFX,35,37,+X,39,41,-
X,31,45,-FF[+X][-X]FFX,53,55,+X,57,59,-X,9,63,+FFFF[+FF[+X][-X]FFX][-
FF[+X][-X]FFX]FFFFFF[+X][-X]FFX,75,77,+X,79,81,-X,71,85,+FF[+X][-
X]FFX,91,93,+X,95,97,-X,87,101,-FF[+X][-X]FFX,109,111,+X,113,115,-
X,65,119,-FFFF[+FF[+X][-X]FFX][-FF[+X][-X]FFX]FFFFFF[+X][-
X]FFX,137,139,+X,141,143,-X,133,147,+FF[+X][-X]FFX,153,155,+X,157,159,-
X,149,163,-FF[+X][-X]FFX,171,173,+X,175,177,-X
```

Fig.49–Collected branches and their positions

4.4.3 Random Pick

The next step is to randomly pick a branch of that plant and take out its initial and final position, as well as the symbols that occur in that interval.

Flash has two different random methods, random() and Math.random(). The first one takes a value and generates an integer number between zero and the given value minus one. However, currently it’s barely used being the second one preferred, even recommended by Macromedia Flash itself.

It generates a random number between 0 and 1, but in order to obtain an integer Math.round(), Math.ceil() or Math.floor() are used. The first rounds the number up or down to the nearest whole number. The second always rounds up, while the third rounds down. In order to get a random number between the values Min and Max, this function is used, though there are variations:

```
randomNumber = Math.floor (Math.random () * (Max – Min + 1)) + Min;
```

```
randomBranch = Math.floor(Math.random() * (branches.length / 3));

var initialPos = branches[randomBranch * 3];
var finalPos = branches[randomBranch * 3 + 1];
var branch = branches[randomBranch * 3 + 2];
```

Fig.50–Random pick of a branch

In this case we want to pick a position within the branches array that ranges from zero to the last branch introduced. Because the minimum value is zero, there’s no need to add it in the function.

As shown in Figure 50, the length of the branch is divided by 3 to ensure only the initial positions are picked, so that when we retrieve the information of the branch we get the right information. As for ignoring the + 1, arrays are always their length minus one, so if we were to use the + 1 we would be accessing a position that doesn’t exist and get an undefined as value, instead of a branch.

4.4.4 Drawing Optimization

Outside Genetic Programming, this version also implemented a drawing optimization for the L-system.

When working with complex plants or too many iterations, this starts to weigh too much on Flash and takes some time to process. A way to reduce the size of the plant is eliminating the empty symbols once the word is produced, but that's often not enough. One other thing that consumes too much memory is the in and out of functions when drawing the plant.

In order to reduce this, it was decided to separate the interpretation in two parts, the calculation of all coordinate positions the plant will take and then the drawing of those coordinates.

```
for(i = 0; i < words.length ; i++){
        var instr = words.charAt(i);
                switch (instr){
                case "F":
                        px = px + lenght * Math.cos(plantAngle);
                        py = py + lenght * Math.sin(plantAngle);
                        coords.push(px, py);
                        break;
                case "-":
                        plantAngle = plantAngle + branchAngle;
                        break;
                case "+":
                        plantAngle = plantAngle - branchAngle;
                        break;
                case "[":
                        var flag = new TurtleCursor(px, py, plantAngle);
                        stack.push(flag);
                        break;
                case "]":
                        flag = stack.pop();
                        px = flag.px;
                        py = flag.py;
                        plantAngle = flag.plantAngle;
                        coords.push(words.charAt(i));
                        coords.push(px, py);
                        break;
                default:
                        trace("unknown command: "+ words.charAt(i));
                        break;
        }
}
```

Fig.51–Calculation and storage of the coordinates

As shown in Figure 51, the calculation of the coordinates is very much like the renderInstruction used in the previous version, except that it stores the coordinates in the coords array instead of drawing them. One other difference is storing the closing bracket and the coordinates previous to the branch as well.

```
for(i = 0; i < coords.length ; i++){

        var instr = coords[i];

        switch (instr){

                case "]":

                c = coords[i+1];
                if(c != "]"){
                        i++;
                        a = coords[i];
                        b = coords[i+1];
                        LS.moveTo(a, b);
                        i++;
                }
                else{
                        i++;
                }
                break;

                default:

                a = coords[i];
                b = coords[i+1];
                LS.lineTo(a, b);

                i++;
                break;
        }
}
```

Fig.52–Drawing the coordinates

In Figure 52 the *for* loop reads the coords array one value at the time and not in jumps of two. This is because there are closing brackets in between, so the array isn’t of an even number length.

When the character isn’t a bracket, then it means it’s a set of coordinates. When this occurs it draws a line from where the cursor last was to the new position, incrementing the counter so it may correspond to the beginning of the next coordinate pair.

Yet whenever a bracket occurs, it checks if the next position of the array is a bracket as well. If not, it jumps to the next position and moves the cursor to that coordinate, incrementing the counter again so when the counter increments at the top of the loop it corresponds to the beginning of the next coordinate pair. If it's another bracket, then it moves to the next position, since it already knows what's in that position isn't a coordinate pair.

4.5 Version 4

This version finally merges the L-system with Genetic Programming and generates a pair of evolved children. Also in answer to the last objective, an interface was implemented to allow the user to pick the parents and generate the offspring once he or she has satisfied with the choice.

As we intend to exemplify the evolution of L-system plants through the application of Genetic Algorithms on the words, in this program we only simulate the reproduction of parents and the resulting offspring.

4.5.1 Individuals

For each of the parents we allow the user to pick one of the three plants we used before on the second version. As shown in Figure 53, each button contains the information of the corresponding plant and when pressed it sends it to the main program. This is achieved by taking advantage of the event handlers Flash has.

```
on(press){
        w = "X";                              // axiom

        var p1 = "X->F[+X][-X]FX";   // production 1
        var p2 = "F->FF";                  // production 2

        productions[0] = p1;
        productions[1] = p2;

        branchAngle = 70 * Math.PI / 180;   //branch angle
        plantAngle = -90 * Math.PI / 180;    // plant angle

        steps = 5;      // iterations
        lenght = 1;     // unit length

        px = 250;       //X coordinate
        py = 149;       //Y coordinate

        color = 0x33CC00;    // color

        depth = 1;      // depth of movie clip
}
```

Fig.53–Example of the code in a button

The on is a mouse event that triggers an action, in this case whenever the mouse presses the button while the pointer is still over it.

The depth was an extra parameter added to all plants. This will be explained more accurately below, but this value is what allows multiple plants to be drawn at the same time.

```
plant1_a.onRelease = function(){

        word = LSystem(w, productions, steps);
        Draw(word, branchAngle, plantAngle, steps, lenght, px, py, color, depth);

        storage[0] = branchAngle; storage[1] = plantAngle; storage[2] = lenght;
storage[3] = steps;
}
```

Fig.54–Example of an action triggered by clicking on a button

On the main code there is another event handler triggered by the release of a certain button. Each button has an instance name attached to it, so this event will only be triggered by that specific button, as exemplified in Figure 54.

When released the event will run the L-System and Draw functions, the same ones developed in the second and third version respectively. The only difference, besides reorganization and cleaning of the code, is that the Draw function takes the new depth parameter.

This parameter is to be used when creating a new empty movie clip. As long as the depth of each movie clip is different, it's possible to have multiple ones on the same stage, but if they are the same then the latest will replace the other. This is why we only have two parents and no more on stage. The first three plants have a depth of 1 while the other three have a depth of 2. Thus, whenever a button of the same color is clicked, it will replace the parent it was previously there, and the same applies to the offspring, though their values are 3 and 4, so they won't erase the parents from the screen.

The storage array saves the parents which were picked to be later sent to be processed by the Genetic Program. We can't use push and pop in this situation because the second parent has the 4th, 5th and 6th positions in the array reserved for it.

4.5.2 Breeding

Once both parents are picked, the user can breed the two plants and generate a pair of offspring. It doesn't matter if they're the same type of plant or not, as long as there are enough branches in the plant to provide a good pool, the chances are the offspring will result different from each other.

```
breed.onRelease = function(){

        if((word != "") && (word2 != "")){

                nodes = Segments(word);
                nodes2 = Segments(word2);

                randomB = RandomBranches (nodes);
                randomB2 = RandomBranches (nodes2);

                Children (word, randomB, randomB2, storage[1],
storage[0], storage[2], storage[3], 250, 394, color, 3);
                Children (word2, randomB2, randomB, storage[5],
storage[4], storage[6], storage[7], 450, 394, color, 4);
        }
}
```

Fig.55–Example of the breed button event

The if statement ensures no action is performed if the word of any of the parents wasn't generated yet, preventing an error when performing the crossover later on (Figure 55).

The Segments function is the picking and storage of all branches occurring in that plant. Once all nodes valid for exchange are collected, the array is sent to the RandomBranches function to pick one of those branches, returning an array with the initial and final positions an the corresponding subtree. Both of these functions are the same as the ones developed in the third version of the program.

The last part is sending both parents, the picked branches and the parameters to generate and draw the offspring.

```
Children = function (parentWord, random1, random2, plantAngle, branchAngle,
lenght, steps, px, py, color, depth)
{
        var child = "";
        var i;

        child = Replace(parentWord, random1, random2);

        Draw(child, branchAngle, plantAngle, steps, lenght, px, py, color, depth);
}
```

Fig.56–Children function

The function in Figure 56 starts by creating the string that will receive the word defining the child and empty one instead of an undefined variable like most ones in Flash because otherwise the first character will have an error. Afterwards, it calls the function Replace to do the switching between the two picked branches and then it sends it to be drawn like a normal L-system.

```
Replace = function(words, randomBranch1, randomBranch2)
{
        var newWord = "";
        var i;

        for(i = 0; i < words.length; i++){

                if((i < randomBranch1[0]) || (i > randomBranch1[1])){

                        newWord = newWord + words.charAt(i);
                }
                else{
                        if(i > randomBranch1[1] - 1){

                                newWord = newWord + randomBranch2[2];
                        }
                }
        }
        return newWord;
}
```

Fig.57–Replace function

Taking the parent word where the replacement will take place, the function in Figure 57 keeps copying the characters of the parent plant into the string, while outside the limit defined the edges of the branch.

While the counter is inside the interval defined by the initial and final position of that branch, the loop just keeps running until the position just before the end of it is reached, adding the string of commands of the branch from the other parent plant.

The reason why it’s done this way is to prevent constant duplication of that string in the word, creating an error, and because the branch from the other plant is likely of different length, we can’t use the counter to travel through the string like before.

4.5.3 Validation

In order to verify if the application reaches the goal of taking two L-system plants and generating two evolved and distinctively different children, we're going to illustrate the program by presenting screenshots of several different results obtained.

4.5.3.1 Parent Plants

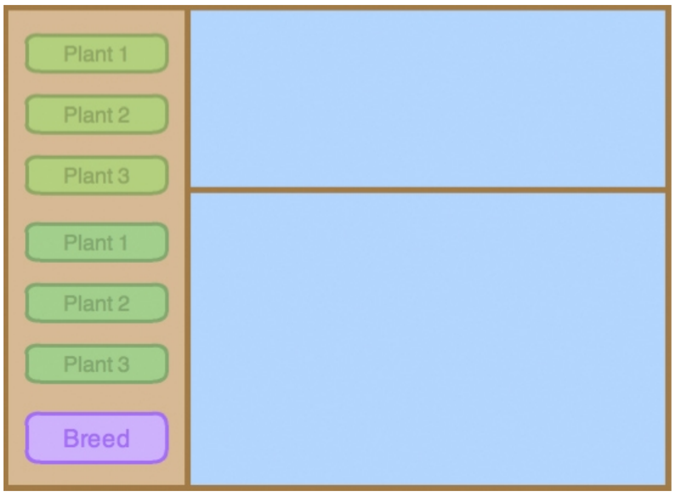


Fig.58–Program interface

As seen in Figure 58, the application has three different buttons for each parent, each containing the information for the respective plant. They will draw the parents on the top section of the window while the children will be presented below.

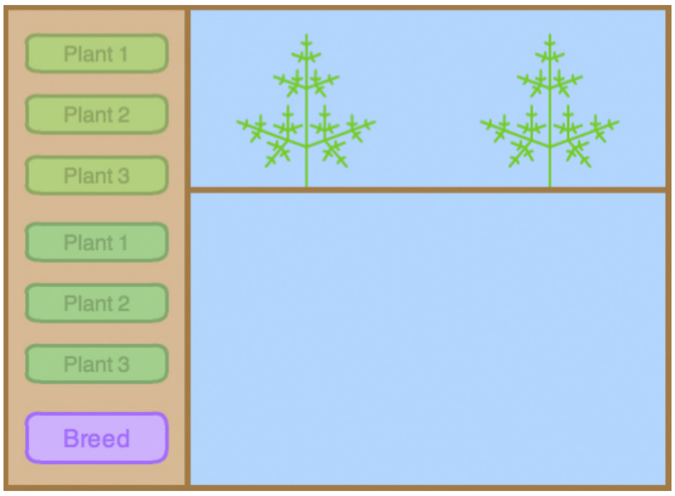


Fig.59–Two identical parents

As said mentioned before, it's possible to use the same type of plants as parents, in this case being the first one (Figure 59). The reproduction will be like an asexual one despite two parents being used, because they're both from the same species.

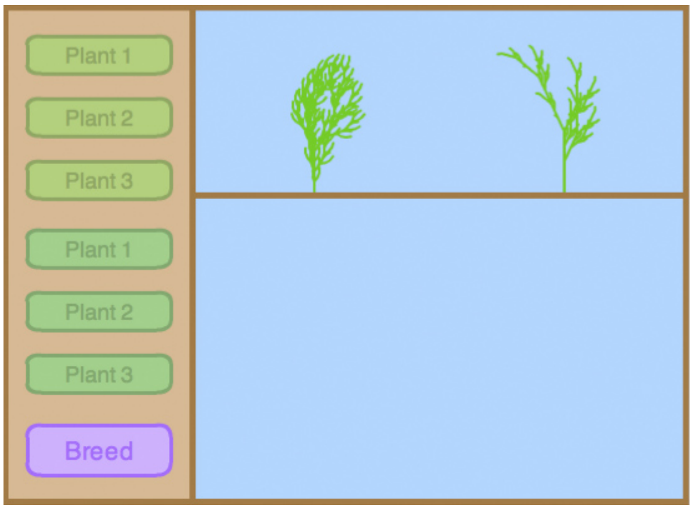


Fig.60–Different parents

In Figure 60 the first parent is the second plant, while the second is the third one. The resulting offspring will tend to look much more different from this sexual breeding than the asexual one showed before.

4.5.3.2 Asexual Reproduction

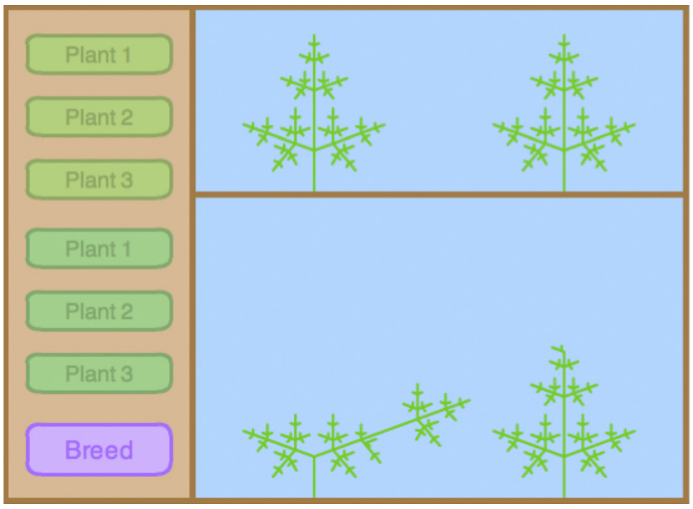


Fig.61–Offpring of Plant 1

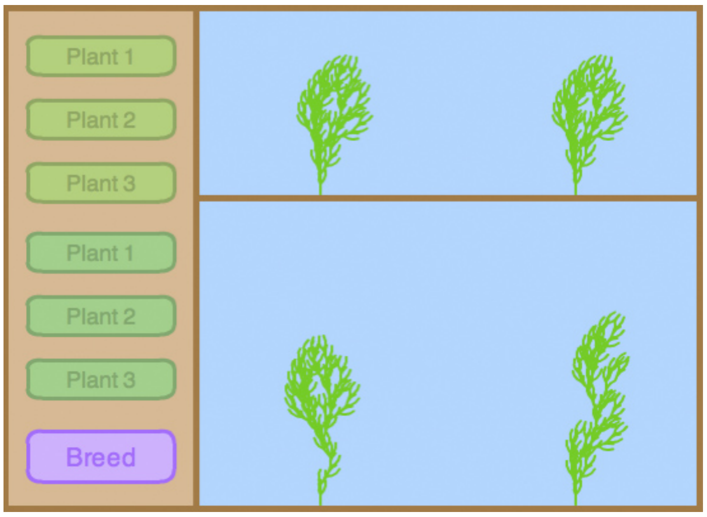


Fig.62–Offpring of Plant 2

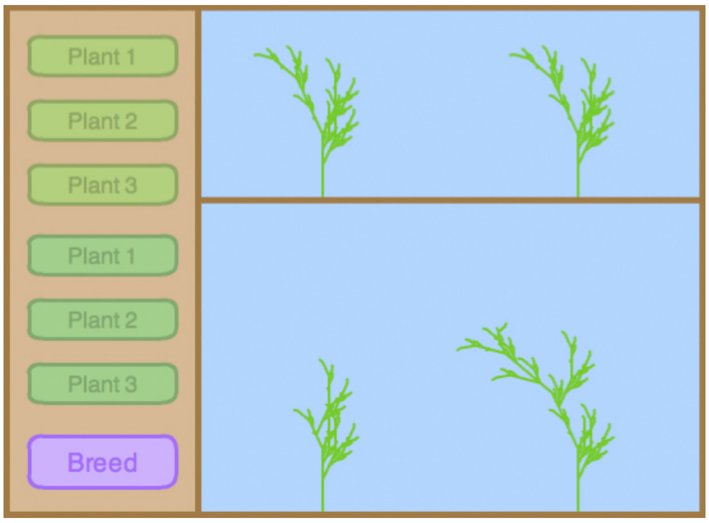


Fig.63–Offpring of Plant 3

As you can see in Figures 61, 62 and 63, the children have clear differences from their parents, even when they're both of the same type.

The amount of change they'll suffer depends of which branches were selected. Sometimes they might be minor, like a small branch turned into the opposite direction as shown in the second child of the figure 61, but generally they're quite noticeable.

4.5.3.3 Sexual Reproduction

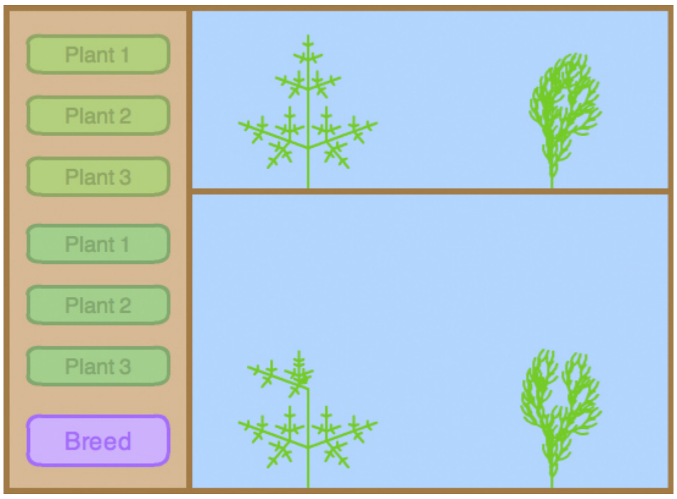


Fig.64–Offpring between Plant 1 and 2

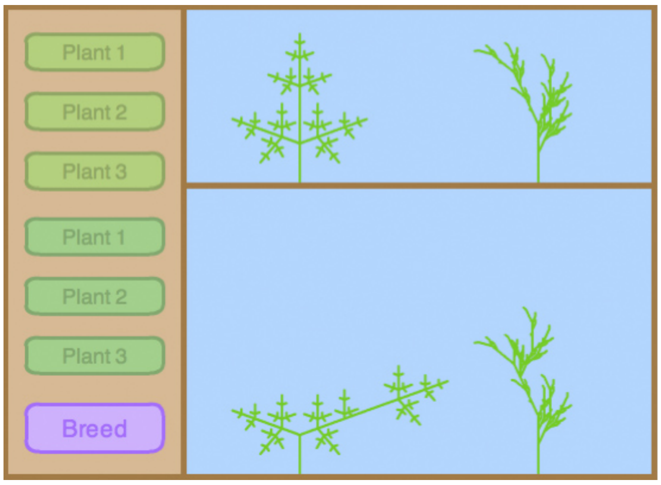


Fig.65–Offpring between Plant 1 and 3

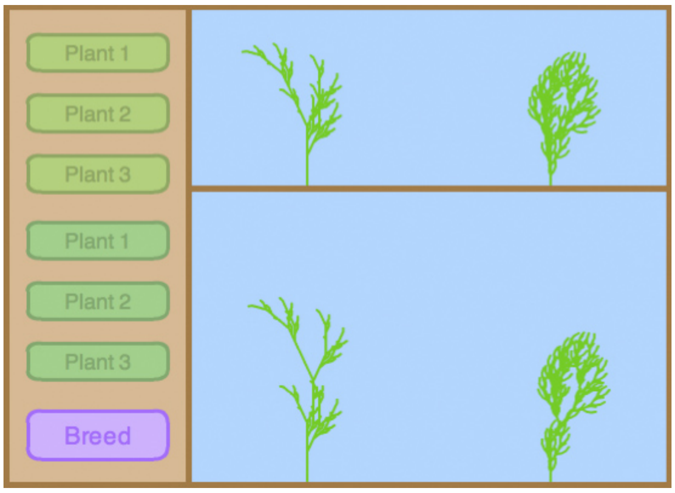


Fig.66–Offpring between Plant 3 and 2

The resultsseen in Figures 64, 65 and 66 might not look too different from the ones obtained through the asexual reproduction, but some of the results obtained by the crossing of two parents from different species wouldn't occur otherwise.

4.5.3.4 Bad Offspring

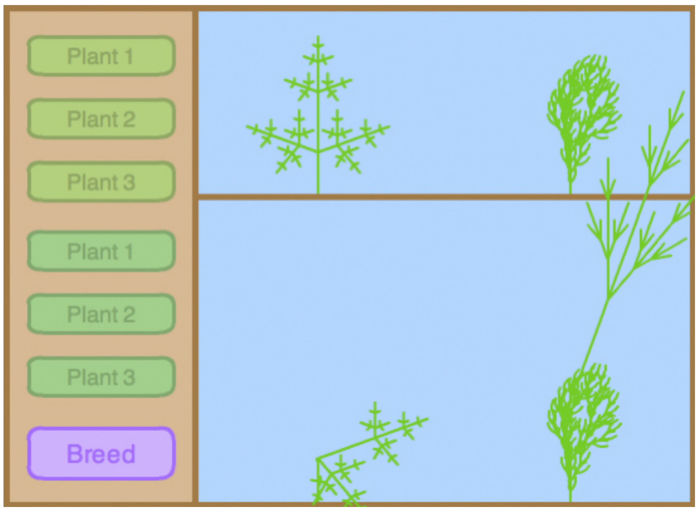


Fig.67–Bad offspring between Plant 1 and 2

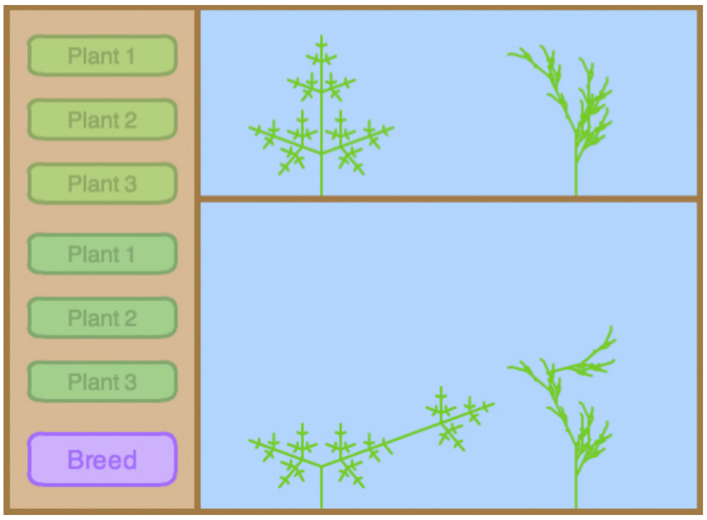


Fig.68–Bad offspring between Plant 1 and 3

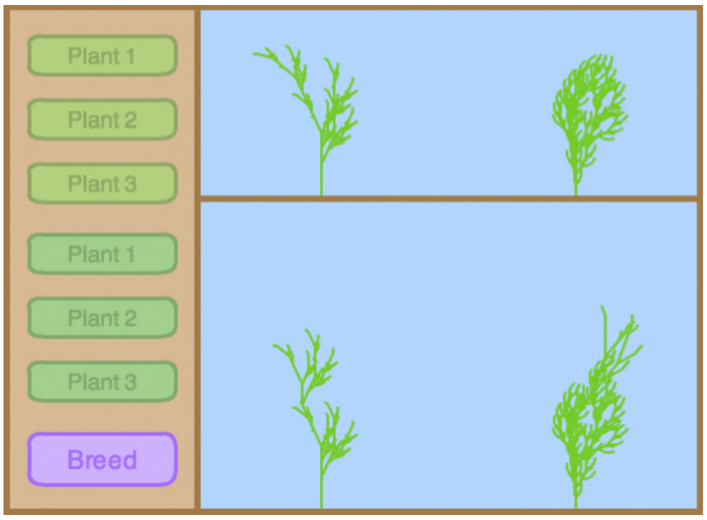


Fig.69–Bad offspring between Plant 3 and 2

Sometimes occasional bad crossings occur, as can be seen in Figures 67, 68 and 69. What can be considered as a bad child is quite relative since we're dealing with a human factor as a fitness function and therefore with a relative perception of what's right and wrong in a plant.

Generally branches that are too long, bending in odd angles or showing a completely different structure from the rest of the plant are viewed as wrong by most people.

4.6 Version 5

Having achieved the goal of using Genetic Programming to evolve L-System plants and proven the resulting offspring are quite different from the parent plants even though only a single generation was produced, this version focuses on increasing the parent population to allow more combinations to take place.

There were some changes in the interface as well. Besides providing more information on each selected plant, the user was given the opportunity to manipulate some of the parameters (branch angle and unit length).

More importantly, the user now needs to determine the fitness value of each of the selected plants in order to determine their probability to generate new offspring, instead of just matching them directly as before.

4.6.1 Individuals

Unlike the previous version, all the code was built in the stage instead of having each button sending the information of the corresponding plant back and forth. The reason for this was to prevent hunting down where the code was located, especially when the number of iterations, the size and the branch angles had to be adjusted.

Also more information was added, such as an identification number, number of the generation and fitness value. The identification and the generation numbers are to identify which of the 10 plants and generation is currently being viewed when the information is presented to the user. As for the fitness, it’s a necessary value for the Genetic Programming.

```
//Initial plant variables

var w1 = "F";                                                      //axiom

var productions1 = new Array("F->FF-[-F+F+F]+[+F-F-F]");           // productions

var steps1 = 3;                                                    // iterations

// Coords

var xx1 = Plant1._x + (Plant1._width / 2);     // X coordinate

var yy1 = Plant1._y + Plant1._height - 2;      // Y coordinate

// Plant structure

plant1 = {};

plant1.word = "";

plant1.branchAngle = 20 * Math.PI / 180;       // branch angle

plant1.plantAngle = -90 * Math.PI / 180;       // plant angle

plant1.lenght = 6;                             // unit length

plant1.color = 0x128729;                       // color

plant1.px = xx1;                               // X coordinate

plant1.py = yy1;                         // Y coordinate

plant1.fitness = 0;                            // fitness

plant1.generation = 0;                         // number of generation

plant1.number = 1;                             // number of plant (id)

plant1.depth = this.getNextHighestDepth(); //depth of movie clip
```

Fig.70–Example of the information of each Plant

As you can see in Figure 70, the initial L-System information and the X and Y coordinates were kept separate from the plant structure. This is because the L-System is only calculated for the parent plants. The offspring result only from the branch exchange between parent plants, not from constant calculation of the axiom and productions through the several steps.

In other words, the evolved plants aren’t technically L-Systems but strings of commands. This was detected in the previous version and is a trait that was kept in the later ones.

This was because the purpose of the program is to produce evolved plants, not evolved L-systems (the L-System is simply used to generate the initial string). If it were the other way around, then the productions and not the final word would be evolved.

4.6.2 Information Presentation

Initially, when the program starts the information from each plant is taken and drawn on the stage with the L-System and Draw functions developed in the previous versions.

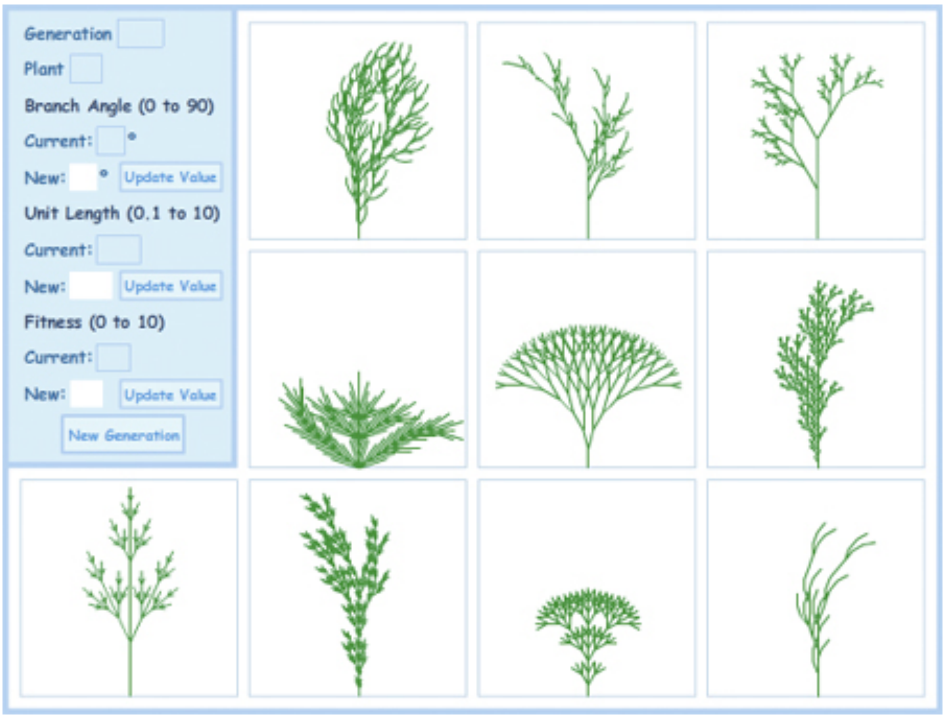


Fig.71–Program Interface

As seen in Figure 71, each of the squares where a plant is drawn is a movie clip with a mouse event attached. Each time the mouse is released while over one of these movie clips, Flash will search for the corresponding event and perform the code contained within it.

```
Plant1.onRelease = function(){
        // displays parent plant info
        text_generation.text = plant1.generation;
        text_plantNo.text = plant1.number;
        text_branchAngle.text = plant1.branchAngle * (180 / Math.PI);
        text_unitLength.text = plant1.lenght;
        text_fitness.text = plant1.fitness;
        text_word.text = plant1.word;
        // clears the values in the updating windows
        text_newBranchAngle.text = "";
        text_newUnitLength.text = "";
        text_newFitness.text = "";
}
```

Fig.72–Example of Information Presentation code

What the code in Figure 72 does is take the information from the corresponding structure and send it to be written in the left and bottom text boxes, as well as clear the information in the update boxes.

```
bt_fitnessUpdate.onRelease = function(){
        // checks if value is valid
        if(text_newFitness.text != "" and text_newFitness.text >= 0 and
text_newFitness.text <= 10){
                // updates the corresponding plant
                if(text_plantNo.text == 1){
                        plant1.fitness = text_newFitness.text;
                        text_fitness.text = plant1.fitness;
                        Draw(plant1.word, plant1.branchAngle,
plant1.plantAngle, plant1.lenght, plant1.px, plant1.py, plant1.color, plant1.depth);
                }
                (…)
        }
}
```

Fig.73–Example of Value Update code

As shown in Figure 73, the unit length, branch angle and fitness values can be updated by the user. The new values can be written in the white text boxes and updated by clicking on the corresponding button on the right, which triggers another mouse event.

Before the value is updated, the code checks if the value is within the determined limits. If it is, it takes the value written on the text box, writes over the old one in the structure and sends the plant to be drawn again, so the user can decide if he likes the changes or not.

4.6.3 Breeding

The breeding of plants takes place when the New Generation button is pressed. The process is the same as in the previous version, except there's a roulette to determine the parent pairs.

The Roulette Wheel Selection is an operator often used in Genetic Algorithms for the parent selection. Taking the total fitness and the fitness of each individual, it assigns a probability of selection to each one, according to the Probability Formula:

$$P = \frac{\text{Favorable cases}}{\text{Possible cases}}$$

The Roulette Wheel Selection can also be compared to a pie chart, where each slice corresponds to the fitness of a certain individual. The larger the probability that individual has, the bigger the chance of being selected.

```
Roulette = function(){
        var totalFitness;
        var randomNumber;
        var fitnessArray = new Array();
        var parentPairs = new Array();
        var i;
        var j;
        // filling the fitness array
        for(i = 1; i <= 10; i++){
                if(i == 1){
                        for(j = 0; j < plant1.fitness; j++){
                                fitnessArray.push(i);
                        }
                }
                (…)
        }
        // getting the sum of all fitness values
        totalFitness = fitnessArray.length;
        // picks a random number within the total fitness and stores the picked parent in the array
        for(i = 0; i < 10; i++){
                randomNumber = Math.floor (Math.random () * (totalFitness + 1));
                parentPairs.push(fitnessArray[randomNumber]);
        }
        return parentPairs;
}
```

Fig.74–Example of the Roulette Wheel Selection code

Because the fitness values are given by the user and there’s no constant total fitness, an array was used to store the identity number of the plant the same amount of times of its fitness, as seen in Figure 74.

After going through all the plants, a random number is generated between 0 and the total fitness and the identity number of the corresponding plant is taken out and stored in an array to return all the picked pairs.

Having the pairs determined, the rest follows the same steps as in the previous version, taking each pair of plants corresponding to the identification number in the array, generating the offspring and drawing the results on the stage.

4.7 Version 6

Although the previous version was successful in randomly pairing parent plants according to the probability determined by their fitness and generating plants through several generations, the code itself was inefficient because of unnecessary duplication.

Besides the code cleaning and the fixing of some minor bugs, the compression of the L-System word was implemented. This was done to reduce the amount of drawing commands, and consequently, the amount of time spent on that task.

The interface was rearranged as well, in order to make it more intuitive for the user. The information displayed is the same, but the update buttons were replaced by horizontal sliders, and instead of 10 small plants, thumbnails and a full view were used to make it easier to tell which plant is selected and how the changes look.

4.7.1 Individuals

Taking advantage of a piece of code commonly found in Flash games to generate multiple enemies and bullets, instead of having 10 different structures declared separately, empty movie clips were declared in a for loop and the parameters defined there.

```
// creates movie clips where plants will be drawn and inserts the general information
for(i = 0; i < 10; i++){
        var newName = "Plant" + (i + 1);
        _root.createEmptyMovieClip(newName,_root.getNextHighestDepth());
        // movie clip position and size
        _root[newName]._x = 250 * i;
        _root[newName]._y = 0;
        _root[newName].width = 240;
        _root[newName].height = 400;
        // X and Y coord of plant
        _root[newName].px = _root[newName]._x + (_root[newName].width / 2);
        _root[newName].py = _root[newName]._y + (_root[newName].height / 2);
        _root[newName].word = "";                               // word
        _root[newName].color = 0x128729;                        // color
        _root[newName].plantAngle = -90 * Math.PI / 180;        // plant angle
        _root[newName].fitness = 0;                             // fitness
        _root[newName].generation = 0;                          // generation number
        _root[newName].number = i + 1;                          // plant ID number
        _root[newName].depth = _root[newName].getDepth();// movie clip depth
}
//Initial plant variables
Plant1.w = "F";                                                 //axiom
Plant1.productions = new Array("F->FF-[-F+F+F]+[+F-F-F]");      // productions
Plant1.steps = 3;                                               // iterations
Plant1.branchAngle = 20 * Math.PI / 180;                        // branch angle
Plant1.lenght = 6;                                              // unit length
```

Fig.75–Example of the generation of the Plants

In Flash it's possible to refer to a symbol either directly by its instance name or with _root[…] and a string containing the name. The previous version used the first approach, but that has the disadvantage of duplicating code when using more than one movie clip performing the same tasks. Besides reducing the code, the second approach can be applied to a virtually infinite amount of symbols (Figure 75).

4.7.2 Word Compression

One of the things noticed in the previous version was a large quantity of commands repeated in the row when displayed on the interface. As mentioned before, the Draw function reads and performs the commands one by one. This means time, memory and processing power is being wasted in a task that could be optimized.

Two approaches were developed to solve this situation. The first one was to store the word in an array with the first position reserved for the amount of times it repeated itself before a different command took place and the second for the command. This solution would also allow expanding the program and developing Parametric L-Systems along with the DOL ones.

The second solution was to keep using a string to store the word, but reserving the first two characters for numerical values (could be more, but anything above 100 involves plants far too big for the memory to handle) and the third one for the command.

4.7.2.1 Array Solution

Because it was being tested the possibility to expand the program to draw Parametric L-Systems as well, the array solution was implemented on the axiom and productions.

The problem however was that the split and join methods used in the previous versions don’t work on arrays, and Flash doesn’t provide similar methods for them. Instead of being able to replace a certain symbol in all the places it occurs in the string, it was necessary to read the word symbol by symbol and search for the corresponding successor in the successor array for each of them.

This involved using several loops within loops and the result was so heavy on the processor that Flash asked to terminate the program before any of the plants could be drawn.

4.7.2.2 String Solution

Although this solution could be used to implement the Parametric L-Systems as well, it has the disadvantage of having a restricted number of characters for the numerical values. This is especially true when they don't use whole numbers.

The compression was tested both during and after the production of the L-System word. The second one proved more efficient because in order to apply the production rules it was necessary to decompress the word again before they could be applied.

```
w = FF              // axiom
p = F->FF[+F]       // production


        Original                Compressed

0:      FF                      02F

1:      FF[+F] FF[+F]           02 FF[+F]

        Compressed

        02F[01+01F]02F[01+01F]
```

Fig.76–Comparison between both Compressions

As it can be seen in Figure 76, if the word isn't decompressed before applying the productions the result is an invalid plant, since the interpreter is expecting for the first two characters to be numerical values and the third the command.

The only exception to this rule is the brackets. When compressed as well, the Draw function wasn't able to restore the position and angle when it reaches a closing bracket. In order for the plants to be drawn correctly, the function checks for an occurrence of a bracket before converting the first 2 characters into a numerical value.

```
Compression = function(words){
        var temp = "";      // temporary word storage
        var i = 0;
        var j = 0;
        var n = 1;
        // reads the word and condenses it
        for(i = 1; i <= words.length; i++){
                // checks the number of occurrences of the same symbol
                if(words.charAt(j) == words.charAt(i)){
                         n++;
                         if(words.charAt(i - 1) == "[" || words.charAt(i - 1) == "]"){
                                 temp += words.charAt(i - 1);
                         }
                }
                else{
                        // doesn't condense the brackets
                        if(words.charAt(i - 1) != "[" && words.charAt(i - 1) != "]"){
                                if(n < 10){
                                        temp +=  words.charAt(i - 1) + String(0) + String(n);
                                }
                                else{
                                        temp += words.charAt(i - 1) + String(n) ;
                                }
                        }
                        else{
                                temp += words.charAt(i - 1);
                        }
                        j = i;      // places the flag in the position of the new symbol
                        n = 1;      // resets the counter
                }
        }
        return temp;
}
```

Fig.77–Example of the Compression function

What the Compression function in Figure 77 does is take the given word and read it character by character, comparing the symbols and incrementing a counter until a different one occurs. When that happens it stores the counter and the command before setting the counter back to zero and updating the position flag used in the symbol comparison.

When a bracket occurs, however, it adds it to the temporary string before updating the counter and the position flag.

4.7.3 Color and Thickness

There was an attempt to make the plants look more realistic by giving the branches different colors and thickness depending of their level.

```
ColorSetting = function(words){
        var colors = [];   var count = 0;   var max = 0;   var i;
        for(i = 0; i < words.length ; i++){
                if(words.charAt(i) == "["){
                        if(count == max){
                                max++;
                        }
                                count++;
                }
                else{
                        if(words.charAt(i) == "]"){
                                count --;
                        }}}
        for (i = 0; i <= max; ++i){
                colors[i] = Math.floor(255 * (1 - i / (max + 3))) << 8;
        }
        return colors;
}
```

Fig.78–Example of the Color Setting function

As shown in Figure 78, the levels are determined by the largest amount of sub-branches that plant possesses. Thus, the function travels through the word and counts how many opening brackets it can find before a closing one occurs. If that number equals to the maximum number of brackets found, then it's incremented.

After determining the number of levels, it determines the hexadecimal value of the green spectrum and stores it in an array.

In the Draw function, there's a variable called levels, which contains the size of the color array. That value is decremented or incremented every time an opening or closing bracket occurs. That variable is then applied on the lineStyle method which determines the line's color and thickness before it's drawn.

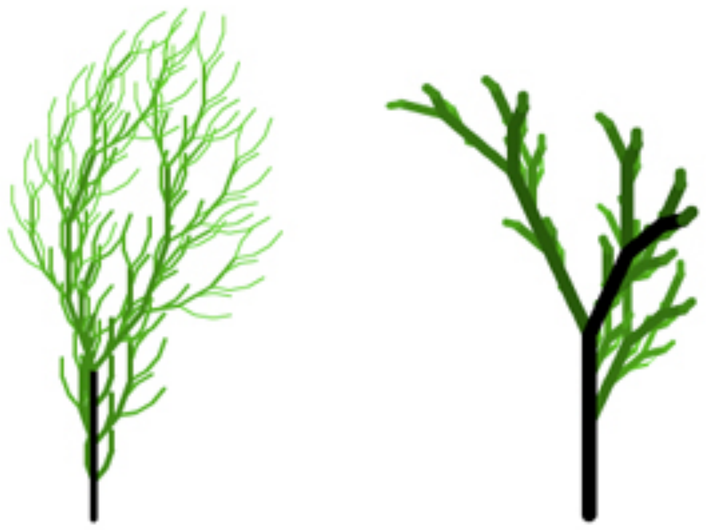
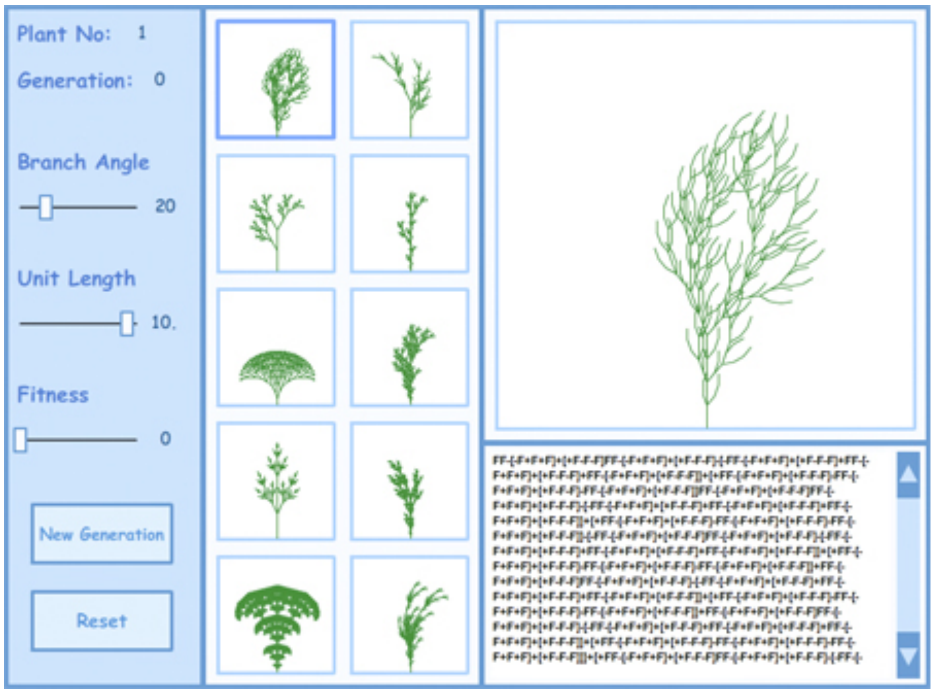


Fig.79–Example of Color and Thickness Levels in Two Different Plants

As you can see in figure 79, the result looks significantly different from plant to plant. It's not that the second plant is badly defined or an error occurred with the colors and thickness, but the brackets are defined in a different sequence than from the first plant. This only proves the L-Systems are counter-intuitive and hard to predict the result when reading the raw word.

4.7.4 Information Presentation

When the program starts, the L-System information of each plant is taken and the words are produced. Before the compression and drawing can take place, the words are clean from unnecessary code, such as canceling rotations or empty brackets.



```
FF-[-F+F+F]+[+F-F-F]FF-[-F+F+F]+[+F-F-F]-[-FF-[-F+F+F]+[+F-F-F]+FF-[-
F+F+F]+[+F-F-F]+FF-[-F+F+F]+[+F-F-F]]+[+FF-[-F+F+F]+[+F-F-F]-FF-[-
F+F+F]+[+F-F-F]-FF-[-F+F+F]+[+F-F-F]]FF-[-F+F+F]+[+F-F-F]FF-[-
F+F+F]+[+F-F-F]-[-FF-[-F+F+F]+[+F-F-F]+FF-[-F+F+F]+[+F-F-F]+FF-[-
F+F+F]+[+F-F-F]]+[+FF-[-F+F+F]+[+F-F-F]-FF-[-F+F+F]+[+F-F-F]-FF-[-
F+F+F]+[+F-F-F]]-[-FF-[-F+F+F]+[+F-F-F]FF-[-F+F+F]+[+F-F-F]-[-FF-[-
F+F+F]+[+F-F-F]+FF-[-F+F+F]+[+F-F-F]+FF-[-F+F+F]+[+F-F-F]]+[+FF-[-
F+F+F]+[+F-F-F]-FF-[-F+F+F]+[+F-F-F]-FF-[-F+F+F]+[+F-F-F]]+FF-[-
F+F+F]+[+F-F-F]FF-[-F+F+F]+[+F-F-F]-[-FF-[-F+F+F]+[+F-F-F]+FF-[-
F+F+F]+[+F-F-F]+FF-[-F+F+F]+[+F-F-F]]+[+FF-[-F+F+F]+[+F-F-F]-FF-[-
F+F+F]+[+F-F-F]-FF-[-F+F+F]+[+F-F-F]]+FF-[-F+F+F]+[+F-F-F]FF-[-
F+F+F]+[+F-F-F]-[-FF-[-F+F+F]+[+F-F-F]+FF-[-F+F+F]+[+F-F-F]+FF-[-
F+F+F]+[+F-F-F]]+[+FF-[-F+F+F]+[+F-F-F]-FF-[-F+F+F]+[+F-F-F]-FF-[-
F+F+F]+[+F-F-F]]]+[+FF-[-F+F+F]+[+F-F-F]FF-[-F+F+F]+[+F-F-F]-[-FF-[-
```

Fig.80–Program Interface

As seen in the Figure 80, the plants in the thumbnails are resized versions of their original size. If the coordinates were calculated directly for that size then there would be a larger margin of error when the values are rounded to fit in the pixels. Also, it would mean calling the Draw function a second time when the plant is presented in full size. Instead, a scaling property the movie clips possess is used, taking advantage of Flash's vectorial nature.

Like in the previous version, each time one of the plants is clicked, the information is presented on the left box. The main difference however was the use of horizontal sliders to facilitate the picking of the values and immediately present the changes on the plant, instead of having to click on a button to apply them.

```
fitness_button.onMouseMove = function() {

            if(fitness_button.flag){

                        var fitness_val = 0;

                        var fitness_plantNo = 0;

                        // calculates the position of the button in relation to the
beginning of the bar

                        var fitness_xx = fitness_button._x - fitness_button.x1;

                        // calculates the corresponding value

                        fitness_val = Math.floor((fitness_xx * (fitness_bar.max - 1)) /
fitness_button.distance) + 1;

                        // writes the value

                        text_fitness.text = fitness_val;

                        // updates the plant value

                        fitness_plantNo = "Plant" + text_plantNo.text;

                        _root[fitness_plantNo].fitness = fitness_val;

            }

}
```

Fig.81–Example of the Fitness onMouseMove event

When the buttons on the sliders are clicked, this triggers an event. While pressed, it allows the button to be dragged within the limits of the slider bar, but it also turns a flag to true. When released the drag method stops and the flag returns to false (Figure 81).

The onMouseMove event, unlike the other two, is called every time the mouse moves and not just when it moves the corresponding button. To keep it from performing code unnecessarily the flag was used. What the rest of the code does is to calculate the value corresponding to the coordinates the button is currently at, then writing it on the text box and storing on the corresponding variable before sending it to be drawn. The fitness is the only one that doesn’t redraw the plant because its value doesn’t change its physical appearance.

The New Generation button uses pretty much the same code as the previous version. The only adaptation was using for loops to go through the different plants, thanks to the use of movie clips instead of structures, which reduced the code duplication by 10. As for the word compression, it wasn’t necessary to make any adaptations for that because the brackets don’t have a numerical value attached to them.

4.7.5 Validation

It had been proven the Genetic Programming is capable of taking two different plants and combine them to generate different offspring, but that alone isn’t enough to prove the plants are actually evolving.

According to the theory of Evolution, the fittest individuals pass on the traits that aid their survival to their offspring through heritage, while harmful or less fit traits become less likely to occur. If there isn’t a constant change of their environment that forces them to adapt in order to survive, then the evolution tends to converge and stagnate after a certain amount of time.

So, in order to validate the evolutionary process of this program, a series of tests were made in order to prove the occurrence of convergence.

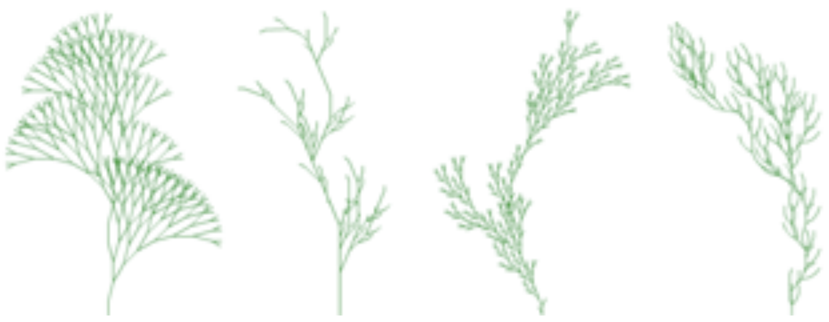


Fig.82–Example of some evolved plants

4.7.5.1 Evaluation System

We have the fitness value to determine which plants are apt to pass on their traits to their children, but how is that value determined in the first place? Ideally, we would have an evaluation system that would be able to classify the plants all in the same manner and help determine the precise value of the fitness.

There are two ways to analyze the plant, Quantitative and Qualitative evaluation. The first focus in obtaining objective values from attributes can be measured. The second uses a subjective evaluation and interpreters according to certain criteria.

As discussed before, the evaluation of the plants can't be determined by numerical values alone because it depends on an aesthetical evaluation. This means the type of evaluation to be used is a Qualitative one.

4.7.5.1.1 Qualitative Evaluation

The criteria decided were the Verticality, Excessive Angles, Odd Branches and Density. More criteria could have been used, but they wouldn't be mutually exclusive, which would result in a duplication of traits and ambiguity in the determination of the values.

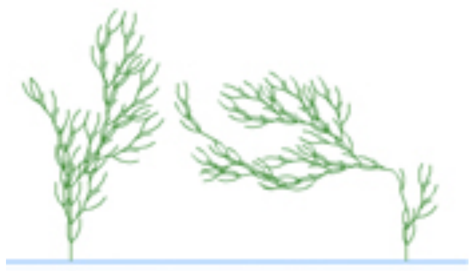


Fig.83–Example of a Good and Bad Verticality criteria

As illustrated in Figure 83, the Verticality criteria checks if the plant is in a good upright position, or it's pending too much to one of the sides, which would make it look unrealistic because of gravity.

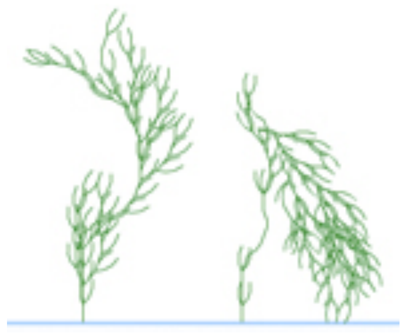


Fig.84–Example of a Good and Bad Excesive Angles criteria

As illustrated in Figure 84, the Excessive Angles criteria checks if the plant doesn’t have any “broken branches”, which result from too many rotations in the same direction.

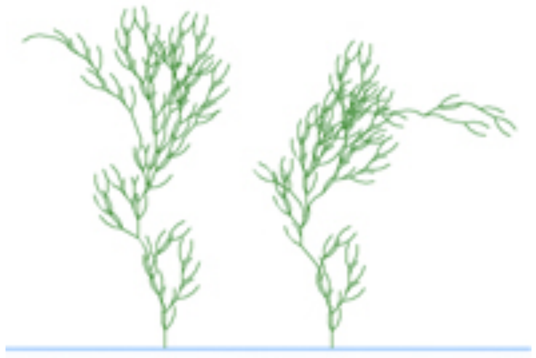


Fig.85–Example of a Good and Bad Odd Branches criteria

As illustrated in Figure 85, the Odd Branches criteria checks if the branches seem to belong to the same species of the plant. This occurs more often when selecting plants from different species, though occasionally, within the same species, a branch will jut out and give an unnatural feeling.

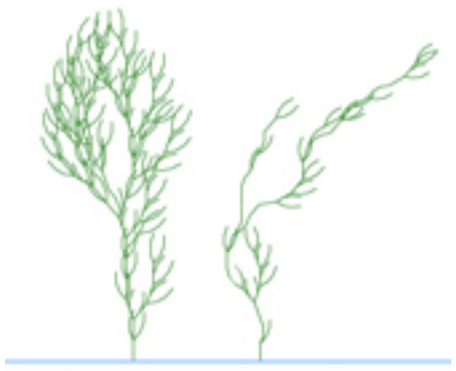


Fig.86–Example of a Good and Bad Density criteria

As illustrated in Figure 86, the Density criteria checks if the plant doesn't thin out and loses its shape as a result from bad branch replacements.

4.7.5.1.2 Evaluation Problems

Qualitative evaluation is never a transparent process because it depends of the user's personal opinion and psychological background, but in this particular situation it is even less.

The evaluation of the plants is done by taking in the different criteria and verifying how well each is fulfilled. The trouble is, because the parent pairing and the branch picking are random, the appearance of the plants varies greatly and is rarely the same. This means that, with exception for the initial population, the evaluation of the plants can only be done once.

Taking into account that in order to prove this Qualitative Evaluation method is reliable, it's necessary to evaluate the plants by an N number of individuals, it becomes impossible to obtain 100% of reliability on the data produced.

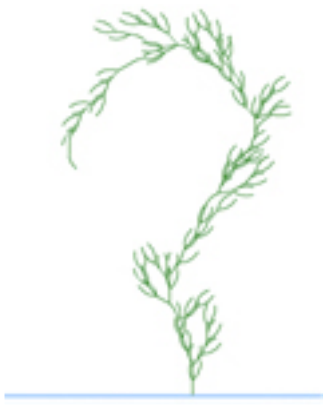


Fig.87–Example of a possible exception

Another problem is that not all plants that don't match the criteria can be considered as bad plants, as seen in Figure 87. Depending on the personal interpretation done by the user, this plant could be considered plausible or not under the right circumstances (for example, a plant blocked partially by an object would bend itself in order to reach the sunlight).

As for the criteria themselves, it’s not possible to define an interval to determine from which point a plant is acceptable and from which is not. The reason why is for the same reason each needed a different number of iterations and unit length when they were first determined. Each plant is different, so the same values can’t be applied to the others, and the same is true for the offspring.

4.7.5.2 Convergence

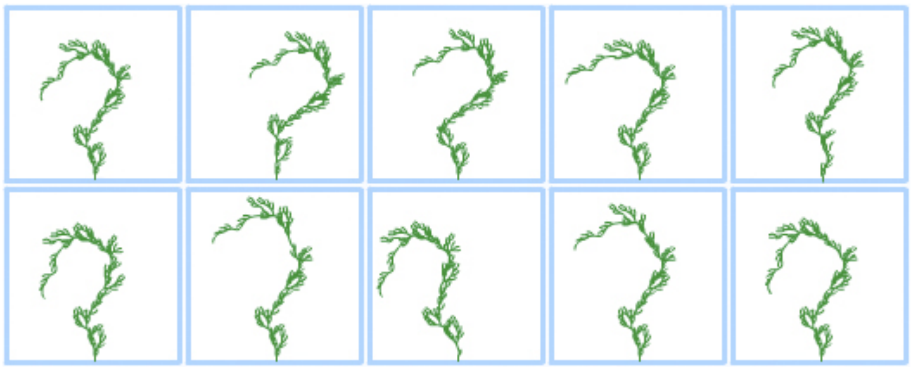


Fig.88–Example of convergence

As more generations take place, the more alike the population starts to become. This is because the words that form them start to become more and more alike. However, there is always some variation, as you can see above in Figure 88.

The amount of generations necessary to reach a minimal state of convergence depends mostly on the number of parents plants picked. The larger the amount of parents picked, the more variation between the words exists. This allows different types of combinations to occur, especially when the plants are from different species, but it takes longer to reach a convergence than by picking a single plant to create the offspring of the next population.

Chance also plays a part in the amount of generation necessary for the plants to converge. Because the appearance of the offspring results from random exchange of branches, there’s no way to predict the outcome. Sometimes the offspring match the criteria; sometimes the branch exchange results in bad children, and sometimes the whole population turns out as invalid plants which forces the user to start from the beginning.

So the amount of generation it takes to reach a convergence depends on the population produced and the choices the user makes. It’s not possible to determine an exact number, but from the results observed, usually, it starts occurring between 5 to 10 generations, varying with the species in question.

5 Conclusions

As presented in Chapter 1, the problem we set out to solve was:

> ***Is it possible to develop an application that simulates plants and uses genetic programming to optimize their graphical representation?***

Being the subsequent sub-problems:

- What's the best structure to simulate 2D plants in a computer?
- How to optimize a graphical representation through an Evolutionary Algorithm?
- How to transfer that structure into the Actionscript language?
- Which are the parameters a user can manipulate to obtain the best graphical representations?

Knowing these goals, in this chapter we will discuss the final conclusions drawn from the learnt procedures, the obstacles found and the achieved results.

We will also discuss the future improvements and applications that could be produced from this work.

5.1 Achieved Objectives

Taking into consideration the objectives we've set out to fulfill and the results obtained in the final program, we can say all goals were achieved:

1. To Identify a method to build plants in a computer, or that can be adapted to work in a computer;
2. Determine how to use the Genetic Programming to evolve those plants;
3. Adapt the developed approach for Actionscript;
4. Allow some level of control to the user to select and adjust the plants.

The results presented show that the L-system method was clearly identified and adapted to work in Flash. Comparing the plants generated in this application to the ones done with normal LOGO turtle graphics, they're exactly the same.

The Genetic Programming implementation deployed to evolve the parent plants proved to be successful, as the resulting children are clearly different from their parents. As for the adaptation of these methods to work in Actionscript and allowing the user to adjust the plants, these goals were also reached.

The user can pick which plants to breed, determine their fitness and adjust their size and branch angle. Also, it's possible for the user to evolve and adapt his or her plants during several generations until he reaches what he or she considers the optimal solution.

5.2 The Application

Since the purpose of this project was to prove we could take an L-system and evolve it through the use of Genetic Programming, we can say the application was successful.

Taking a deeper look into the obtained results, although the offspring go through several generations, they keep a certain resemblance with the original parents. This is because only the change of branches occurs in the Genetic Programming, not altering the information within them.

The recursive nature of the L-systems proved to be frustrating when dealing with memory issues. When drawing more than eight iterations the program takes a long time to process and draw the word, sometimes even stops working. This situation isn't something that can't be fixed however, since it's a necessary evil when working with L-systems, but the problem doesn't occur just in Flash. Other applications developed with different languages suffer from this as well.

One of things that was only realized during the implementation of the application was the difference between the initial population and the ones generated afterwards. The initial plants are generated by the L-system, using the axioms, productions and number of iterations to produce the final word, but the offspring result from the exchange of branches between the parent plants. In other words, the children are no longer L-systems. This means that if we wanted the offspring to have more iterations, to add more complexity to their appearance, it wouldn't be possible. This drawback was detected while developing the 4th version.

Depending what the next objective would be, this would either have a major or minor impact on the program. If the goal was to generate a plant but still expect to be able to control the optimal solution like a normal L-system, then the whole Genetic Programming approach would have to be reworked in order to evolve the axiom and the productions instead of the final word.

As for Actionscript as a programming language, the conclusion drawn from the ongoing learning process and techniques found is that this language has a great potential, especially when it comes to developing graphical and interactive applications. The only drawback was its lack of processing power compared to other languages, which was visible when calculating several iterations in a L-System and drawing multiple, detailed plants at once.

5.3 Future Improvements

One of the ideas that came to mind, but there wasn't enough time to develop, was to create a plant editor where the user could generate and personalize his or her own plant. This could be done by either allowing him or her to edit the axiom, productions and number of iterations, or by implementing the Stochastic L-system.

In the beginning it would be necessary to limit the symbols that could be used and the amount of productions the plant could have. Also, validations would be in order to prevent errors like unopened or unclosed brackets. As for the second, there would be a list of productions that could be picked, depending of the axiom selected (having productions with the "X" predecessor when the axiom is "F" would just waste time and memory, because they would never occur). Introducing productions could also be accepted.

As for the plant personalization, leaves could be added and the color of the plant changed. The leaves would be movie clips placed in the library that would be duplicated and placed at the end of each branch, aligning them with the rotation calculated by the interpreter. Changing the color would be done by using sliders and converting the RGB values into hexadecimal, so they could be applied to the Line method.

Another future application could be a program capable of generating plants without the aid of the user. The fitness value would be calculated by the computer, using some of the criteria identified and quantifying it into numerical values. This would allow working with a much larger population and at a much faster pace, though it's uncertain if the results would be pleasant to the eye.

Another idea is having the application eliminating some of the children that don’t qualify to be part of the population. Things like branches growing outside the boundaries of the window or, especially, below the horizon line are situations that aren’t dependent on an aesthetic evaluation and therefore can be processed by the computer.

6 References

[1] Azevedo, E, 2005, *Desenvolvimento de Jogos 3D e Aplicações em Realidade Virtual*, Campus (eds.), Rio Janeiro.

[2] Bian, R, Hanan, J, and Chiba, N 2004, 'Statistical data directed evolution of L-system models for botanical trees: Calibration of an L-System model'. *FSPM*, viewed, 15, Oct. 2007, <http://amap.cirad.fr/workshop/FSPM04/proceedings/4thFSPM04_S5Bian.pdf>.

[3] Bisoi, AK, Mishra, SN and Mishra, J 2004, 'Growing a class of fractals based on combination of classical fractals and recursive mathematical series in L-systems', *MG&V* vol. 13, 3, p. 275-288.

[4] Bonfim, D & Castro, L 2005, 'Projeto Evolutivo de Sistemas Lindermayer: Uma Abordagem Baseada em Programação Genética', *XXV Congresso da Sociedade Brasileira de Computação*, UNISINOS – São Leopoldo/RS (eds.), Rio Janeiro.

[5] Borovikov, IA 1995, 'L-systems with inheritance: an object-oriented extension of L-systems'. *SIGPLAN,* Vol. 30, 5, p.43-60, viewed 7 Oct. *2007*, <http://doi.acm.org/10.1145/201937.201944>.

[6] Chen, YP and Colomb, RM 2003, 'Database technologies for L-system simulations in virtual plant applications on bioinformatics'. *Knowl. Inf. Syst.* 5, 3, p.288-314., viewed, 15 Spt. 2007, <http://dx.doi.org/10.1007/s10115-002-0087-0>.

[7] Coello, CA 2007, 'Evolutionary Algorithms: Basic Concepts and Application in Biometrics, Image Pattern Recognition: Synthesis and Analysis in Biometrics'*, World Scientific,* p.289-320, Singapore.

[8] Crawford, S and Boese, E 2006, 'ActionScript: a gentle introduction to programming', *Journal Comput*. Small Coll. 21, 3, p.156-168.

[9] Fogel, DB 2000, *Evolutionary computation*, 2nd edition. IEEE Press (eds.), Piscataway, NY.

[10] Fuhrer, M 2005, 'Hairs, Textures, and Shades: Improving the realism of plant Models Generated with L-Systems', Master Thesis. Department Computer Science, Calgary.

[11] Grubert, M (2001), '*Simulating Plant Growth*, L-arbor', viewed, 12, Oct. 2007, <www.acm.org/crossroads/crew/marco_grubert.html>.

[12] Jacob, C 1994, 'Genetic L-System Programming', PPSN III - Parallel Problem Solving from Nature, *International Conference on Evolutionary Computation*, Lecture Notes in Computer Science 866, p.334-343, Springer-Verlag, Berlin.

[13] Jacob, C 1995, 'Genetic L-System Programming: Breeding and Evolving Artificial Flowers with Mathematica', IMS´95, Proc. *First International Mathematica Symposium, Southampton, Great Britain*, Computational Mechanics Publications, p. 215-222, Southampton, UK.

[14] Jacob, C 1995, 'Modeling Growth with L-Systems & Mathematica', in*: Mathematica in Education and Research*, Volume 4, No. 3, pp. 12-19, TELOS-Springer.

[15] Jacob, C 1996, 'Evolving Evolution Programs: Genetic Programming and L-Systems', Genetic Programming 1996: *First Ann. Conf. MIT Press,* Cambridge.

[16] Karwowski, R & Prusinkiewicz, P 2004, 'The L-system-based plant-modeling environment L-studio 4.0, *In FSPM04.* wiewed 30 September 2007, <http://amap.cirad.fr/workshop/FSPM04/proceedings/4thFSPM04_S5Allen.pdf>.

[17] Koza, JR 1990, *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems*, Technical report STAN-CS90 -1314. Stanford, CA: Stanford University.

[18] Koza, JR 1992, 'Genetic Programming', *On the programming of computers by means of natural selection*. Cambridge MA: The MIT Press. Cambridge, Massachusetts.

[19] Koza, JR 1994, *Genetic programming II: automatic discovery of reusable programs*, MIT Press, Cambridge, MA.

[20] Koza, JR 2001, 'Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems', *Proceedings of the 6th European Conference on Advances in Artificial Life,* p.659-668.

[21] Koza, JR 2007, 'Introduction to genetic programming', In *Proceedings of the GECCO Conference Companion on Genetic and Evolutionary Computation* (London, United Kingdom, July 07 - 11, 2007). GECCO '07. ACM, New York, NY, p.3323-3365, viewed 12 Oct. 2007 <http://doi.acm.org/10.1145/1274000.1274116>.

[22] Lecky-Thompson, G 2001, *Infinite Game Universe: Mathematical Techniques*. Pub., Charles River Media (eds.), Boston, Massachusetts.

[23] Leutenegger, S and Edgington, J 2007, 'A games first approach to teaching introductory programming', In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education* (Covington, Kentucky, USA, March 07 - 11, 2007). SIGCSE '07. ACM, p.115-118. New York, NY, viewed 22 Oct. 2007, < http://doi.acm.org/10.1145/1227310.1227352>.

[24] Lindenmayer, A & Prusinkiewicz, P 1990, '*The algorithmic beauty of plants*'. Chapter 1: Graphical modeling using L-systems. Springer-Verlag (eds).

[25] Lluch, J, Camahort, E, and Vivó, R 2003, 'Procedural multiresolution for plant and tree rendering'. In *Proceedings of the 2nd international Conference on Computer Graphics, Virtual Reality, Visualisation and interaction in Africa* (Cape Town, South Africa, February 03 - 05, 2003). AFRIGRAPH '03. ACM, New York, NY, p.31-38, viewed 25 Oct. 2007 <http://doi.acm.org/10.1145/602330.602336>

[26] Quan, L, Tan, P, Zeng, G, Yuan, L, Wang, J and Kang, SB 2006. Image-based plant modeling. In *ACM SIGGRAPH 2006 Papers* (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06, p.599-604. ACM, New York, NY, viewed 25 Oct. 2007, <http://doi.acm.org/10.1145/1179352.1141929>.

[27] Mook, C. 2003, *ActionScript for Flash MX; The Definitive Guide*, San Francisco, CA: O'Reilly & Associates. Ford H Odling-Smee A, New Handmade Graphics, Beyond Digital Design (eds).

[28] Morais, SF (2003), '*Computação Evolutiva e Lógica Fuzzy*', Master Thesis. Universidade Federal do Rio Grande do Sul – ufrgs. Brasil.

[29] Noser, H, Stucki, P, Walser H, 2001, ‘Integration of Optimization by Genetic Algorithms into an L-System-Based Animation System’, *Proceedings Computer Animation 2001* (November 7-8), pp. 106-112, Seoul, Korea.

[30] Ochoa, G 1998, ‘On Genetic Algorithms and Lindenmayer Systems’, *Lecture Notes in Computer Science 1498*, p.335--344.

[31] Olsen, P 2006, *The Golden Section: Nature’s Greatest Secret*, Wooden Books Ltd. (eds.), Glastonbury, Somerset.

[32] Onishi, K, Hasuike, S, Kitamura, Y and Kishino, F 2003, ‘Interactive modeling of trees by using growth simulation’, In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (Osaka, Japan, October 01 - 03, 2003). VRST '03. ACM, New York, NY, p.66-72, viewed 7 Oct. 2007, <http://doi.acm.org/10.1145/1008653.1008667>

[33] Pappa G & Freitas A 2006, ‘Towards a Genetic Programming Algorithm for Automatically Evolving Rule Induction Algorithms’, *Lecture Notes in Computer Science* Publisher Springer Berlin (eds.), p. 341-352, Heidelberg.

[34] Parish, YI and Müller, P 2001, ‘Procedural modeling of cities’, In *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '01. ACM, New York, NY, p.301-308, viewed 12 Oct. 2007, <http://doi.acm.org/10.1145/383259.383292>.

[35] Peterson PR, 1997, ‘*A Genetic Engineering Approach to Texture Synthesis*’, Simon Fraser University School of Computing Science Theses, *viewed 12 Oct. 2007* <http://fas.sfu.ca/pub/cs/theses/2005/>

[36] Prusinkiewicz P, 1986, ‘Graphical applications of L-systems’, *Proceedings of Graphics Interface '86 / Vision Interface '86*, p. 247-253.

[37] Prusinkiewicz, P, Lindenmayer, A and Hanan, J 1988, ‘Development models of herbaceous plants for computer imagery purposes’ In *Proceedings of the 15th Annual Conference on Computer Graphics and interactive Techniques* R. J. Beach, Ed. SIGGRAPH '88, p.141-150ACM, New York, NY, viewed 7 Oct. 2007, <http://doi.acm.org/10.1145/54852.378503>.

[38] Prusinkiewicz, P & Hanan J 1990, ‘Visualization of botanical structures and processes using parametric L-systems’. In D. Thalmann (eds.), *Scientific Visualization and Graphics Simulation*, p.183–201, Chichester, J. Wiley Sons.

[39] Prusinkiewicz P 1993, ‘Modelling and visualization of biological structures’, In *Proceedings of Graphics Interface ’93*, p. 128–137.

[40] Prusinkiewicz, P, Hammel, MS and Mjolsness, E 1993, ‘Animation of plant development’, In *Proceedings of the 20th Annual Conference on Computer* [] *Graphics and interactive Techniques* SIGGRAPH '93, p.351-360. ACM, New York, NY, viewed 13 Oct. 2007, <http://doi.acm.org/10.1145/166117.166161>.

[41] Prusinkiewicz P, Hammel M., Hanan J and Mech, R 1996, ‘L-systems: from theory to visual models’, In *proceedings of the* 2nd *CSIRO symposium on computational challenges in life science*.

[42] Prusinkiewicz P, 1997, Modelling of Spatial Structure and Development of Plants: Review.

[43] Prusinkiewicz, P, Hanan, J & Mech, R 2000, ‘An L-system-based Plant Modeling Language’, *Lecture Notes in Computer Science 1779*, p.395–410. Springer- Verlag, Berlin.

[44] Roden, T, and Parberry, I 2004, ‘From Artistry to Automation: A Structured Methodology for Procedural Content Creation. *In Proceedings of the 3rd*

International Conference on Entertainment Computing (Eindhoven, The Netherlands, September 1-3), p.151-156.

[45] Roden, T, and Parberry, I 2005, *Procedural Level Generation, Game Programming* Gems 5, Charles River Media (eds.).

[46] Rodkaew Y, Chongstitvatana P, Siripant S, Lursinsap, C 2004, 'Modeling plant leaves in marble-patterned colours with particle transportation system', *In FSPM04*, viewed 13 Oct. 2007,http://amap.cirad.fr/workshop/FSPM04/proceedings/4thFSPM04_S7Rodkaew.pdf

[47] Russel S, Norvig P, 2003, *Inteligência artificial*. 2ª Edição, Campus (eds.), Rio Janeiro, Brasil.

[48] Salomaa, A, 1987, Formal languages, Academic Press Professional, Inc., San Diego, CA,

[49] *Samuel, E 2007, 'M-Systems, A Conformal Approach to Plant Morphogenesis'. Uppsala* Universitet. <http://www.math.uu.se/studie/grundutb/exjobb/exarbeten.php>

[50] Sims, K. 1991, 'Artificial Evolution for Computer Graphics', In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, p.319--328.

[51] Tan, P, Zeng, G, Wang, J, Kang, SB, and Quan, L 2007, 'Image-based tree modeling', In *ACM SIGGRAPH 2007 Papers* (San Diego, California, August 05 - 09, 2007). SIGGRAPH '07. ACM, New York, NY, p.87, viewed 3 Oct. 2007, http://doi.acm.org/10.1145/1275808.1276486.

[52] Vega, FF 2001,'Parallel and distributed Genetic Programming Models with applications to Logic Sintesis on FPGAs, PhD Thesis. Computer Science Department, Universidad Extremadura. Espanha.

[53] Wonka, P 2006, 'Procedural modeling of architecture'. In *ACM SIGGRAPH 2006 Courses* (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06. p.17-83, ACM, New York, NY, viewed 11 Oct. 2007 <http://doi.acm.org/10.1145/1185657.1185713>.