Automatic level generation for platform videogames using Genetic Algorithms

Fausto Mourato
Dep. Sistemas e Informática
Escola Superior de Tecnologia
Instituto Politécnico de Setúbal
2910-761 Setúbal – Portugal
+351 265 790 000
fmourato@est.ips.pt

Manuel Próspero dos Santos
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica – Portugal
+351 212 948 536
ps@di.fct.unl.pt

Fernando Birra
CITI, Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica – Portugal
+351 212 948 536
fpb@di.fct.unl.pt

ABSTRACT
In this document we present an investigation on automatically generating levels for platform videogames. Common approaches for this problem are rhythm based, where input patterns are transformed in a valid geometry, and chunk based, where samples are humanly created and automatically assembled like a puzzle. The proposal hereby presented is to explore this challenge with the usage of Genetic Algorithms, facing it as a search problem, in order to achieve higher expressivity and less linearity than in rhythm based approach and without requiring human creation as it happens with the chunk based approach. With simple heuristics the system is able to generate playable levels in a small amount of time (one level is created in less than a minute) and with considerable diversity, as our results show.

Categories and Subject Descriptors
I.3.5 **[Computer Graphics]**: Geometric Algorithms, Languages, and Systems, I.3.6 **[Computer Graphics]**: Interaction Techniques.

General Terms
Algorithms, Design, Experimentation, Human Factors.

Keywords
Platform Videogames, Procedural Content Generation, Automated Game Design.

1. INTRODUCTION
The automatic generation of content is an area of interest for scientists from different domains in Computer Science, such as Computer Graphics, Artificial Intelligence, Human-Computer Interaction, among others. In this paper we direct our focus to the automated generation of content for platform videogames. In particular, our goal is to have a system that is able to automatically generate levels for this type of games.

Platform games, such as *Super Mario Bros*. and *Sonic – the Hedgehog*, represent one particular genre of gaming where the user controls a character and guides him through a scenario, performing jumps over gaps and confronting opponents, typically in a bi-dimensional environment. This type of games was particular popular in the 1980's. However, and maybe due to the excessive complexity of contemporary games, *platformers* are starting to appear once again in recent videogame releases, either as remakes with improved graphics or as new ideas taking advantage of contemporary technology such as the wide spread of the Internet. For instance, Nintendo released *New Super Mario Bros Wii* in 2009 and Sega released *Sonic 4* in 2010. In addition, the videogame *Little Big Planet* is a good example of a modern *platformer*.

In this article we focus the problem of automatically generating levels for this type of games, with a different approach from those that have been used up to the present time. As we will further see in Section 2 of this document, where related work is presented, some alternatives have been considered for this purpose, such as rhythm and chunk based. We present the possibility of approaching this challenge as a search problem, tackling it with the usage of Genetic Algorithms. The main contribution of this project is the approach by itself inside this context, which is new and promissory. Associated to that, we also bring the definition of heuristics to measure the quality of a level based on the geometrical content and interaction parameters. Finally, the implemented prototype provides a proof of feasibility of this approach.

Automatically generating this type of game spaces is an interesting challenge, in particular because it appears to be a simple task, although, it raises several issues and non-trivial aspects to be addressed. Though in its main principles it can be perceived as generating a generic geometry, such as what happens when a system procedurally generates a tree or a building, for instance, one has to take into account that the final geometry has to represent a challenge, associated to a certain degree of difficulty. Also, this type of geometric content is semantically sensitive, since a slight change in a small component may invalidate the whole content. For instance, a minor random change in a valid level can be enough to make it impossible to complete.

Procedural Content Generation can be used in different ways. For small companies and independent developers, this may represent a solution for the time consuming task of producing game content. More generally, the main potential is in the possibility of creating an uncountable set of levels, which solves the problem of level predictability. For instance, it becomes impossible to go online for the level solution and/or secret locations. Finally, this approach creates room for designers to conceive new mechanics that adapt gameplay to fit competitiveness in ever-changing scenarios.

Short presentation, *ACE'2011*- Lisbon, Portugal

For the purpose of this project, the main inspiration can be found in a classical platform videogame released in 1989, entitled *Prince of Persia* (screenshot provided in Figure 1). This game became a franchise with several associated releases in the last decade with contemporary graphical realism and gameplay, out of the initial platform based approach. Regarding the original game, levels typically consist of dungeons with corridors, filled with traps and gaps. Also, the character occasionally encounters and confronts enemies. In this particular game, movement challenges such as jumping through gaps and enemy confrontations are kept separate, meaning that the player will not be simultaneously moving and fighting. In Section 3 of this document we will go into the details about the generation process and we will explain the main reason of having *Prince of Persia* as the main inspiration. Nevertheless, it is important to state that the technique is not restricted to this particular game only. We believe that some of the ideas that will be presented can be generalized and used in other platform videogames.

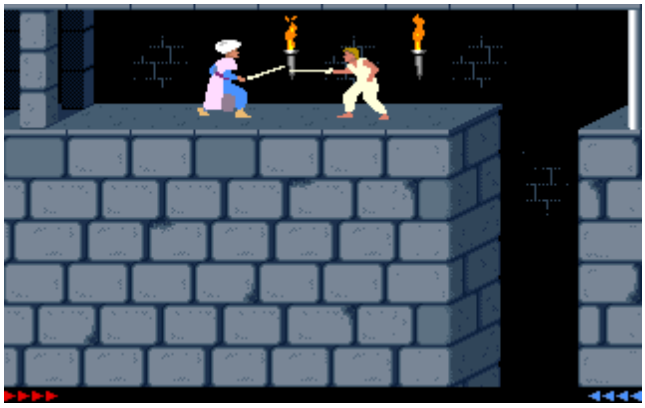

Figure 1 – Screenshot of the videogame *Prince of Persia*

Even though this work has some aspects in a preliminary stage, our results are promissory as we will see in Section 4. This approach generates several distinct valid levels. A fast generation process of not more than a few seconds provides, with almost 100% certainty, a valid level with a couple of minutes of gameplay. Rising the computation times to one or two minutes allow to perfect the generation process with particular good detail.

2. RELATED WORK

Procedurally generating game spaces is a topic that informally appeared, in a videogame entitled *Rogue*, back in 1980. The game was ASCII based and its geometry was defined recurring to simple characters. The idea behind the procedural generation was simple: rooms were generated with random positions and sizes and were interconnected by corridors also randomly created. Despite the popularity this game achieved - Version 4.2 of *BSD UNIX* included *Rogue* – the approach behind it got unnoticed. Interestingly, this topic became popular in recent times, as a way to improve "replayability" in games. Regarding independent game development, two recent titles are worth mentioning: *Minecraft* and *Spelunky*. Both have game environments that are procedurally generated. In the scientific community, this topic has also been approached as we will further see.

Generating graphical entities, such as buildings, is a problem that has been actively approached. As an example, Lipp [6] uses L-Systems as an efficient way to automatically generate buildings with the possibility of small scale control. Considering nature, the examples are numerous, and mature cases can be found back in Mandelbrot and Hudson's ideas to generate terrain procedurally, for instance, the mid-point displacement technique [7]. However, ideas such as these are recent in the generation of game spaces.

The interest behind the topic of automating the generation of levels for platform games was pioneered by Compton and Mateas [1]. They proposed some principles that could be used to interpret and describe platform levels, suitable for an automatic generation system. Concerning to movement, authors have defined a model with different possible patterns: basic, complex, compound and composite. These patterns represent the organization of a level in components, which are the platforms and other graphical entities that compose a level. They have also presented some ideas about the need of having a physics model to perceive possible trajectories to identify difficulty, but only as a theoretical need without effective concretization.

Later, Smith *et al*. [12] presented a more extensive analysis to the existing components of a platform level, with the creation of a conceptual model that defines associations and a hierarchy for the different entities. The used principles follow some of the thoughts proposed in the Game Ontology Project (GOP) by Zegal *et al*. [18], where a more generic model was proposed. The defined hierarchy by Smith *et al*. represents an interesting approach to the problem and largely covers the concepts.

The previous work led to the creation of a system that effectively generates levels for platform games [13]. This system was later named *Launchpad*. The main idea is that the generation of level segments can be based in input patterns that the player needs to match. Therefore, when the player is performing well, the sequence of actions flows naturally, like playing a melody in a piano, following the principle of Flow proposed by Csikszentmihalyi [2], which represents the ideal state of immersion and control over a certain skill based task. We can consider this type of level generation as the **rhythm based approach**.

In order to evaluate the expressive range of the previous work, the same authors proposed a method to analyse the generated content [14]. One important aspect that should be retained is that it is not only important to examine the number of different levels that are generated and the time needed for their creation, but also to extract how different and varied the results are. Authors analysed the output regarding linearity of the path and the relative difficulty measurement.

Still taking into account the concept of expressivity, the more natural way to expand it and add creativity is to include a user or a set of users tweaking the output or the process of production. This approach is referred as a **mixed-initiative**, which means that the final result is obtained after a cooperative effort of humans and machines. Smith *et al*. [15] presented a prototype system for this purpose using constraint programming.

Finally, Mawhorter and Mateas [8] presented a different approach to level generation. They introduced Occupancy Regulated Extension (ORE), an algorithm to create a game space based on the composition of pre-authored chunks. One of the main inspirations behind this work is the previously referred game *Spelunky*. This different idea can be seen as **chunk based approach**.

Our proposal is an alternative to the generation process, with a novel approach based on Genetic Algorithms, a class of Evolutionary Computation techniques that mimics real life evolution. It is based on natural selection and is commonly used in search problems with exponential growth that leads to the

impossibility of testing all potential solutions. To a more in-deep study about Genetic Algorithms we point to Holland [4] and Goldberg's [3] books. The usage of these principles in Computer Graphics and Interaction is not new, and has served various purposes. For instance, to point a few, they have been used for the reconstruction of missing parts of a real geometric object represented by volume data, as proposed by Savchenko and Schmitt [11], and to tune the parameters of an existing triangle stripping algorithm, as proposed by Lord and Brown [5]. However, Genetic Algorithms have not been used as a tool for game content generation in the way we propose, in particular a game level that presents an associate challenge rather than a simple physically valid environment.

The most similar approaches to what we propose and that have been considered to generate game content can be found compiled in Togelius *et al*. study on Search-based Procedural Content Generation [16]. This article presents a good overview about possible alternatives and important considerations in the topic that were particularly relevant in the definition of our system. In addition, the authors refer two other interesting works to be considered in the scope of this document.

The first work shows one possible usage for Genetic Algorithms in the context of Game Content Generation, proposed by Togelius and Schmidhuber [17]. The authors presented a system that evolves rule sets for Pacman-like games, converging to alternative game variants.

The second example, proposed by Padersen *et al*. [10] shows a level generation for platform videogames. A simple and linear game is constructed randomly based on a small set of parameters, such as the average gap size. This simple construction process was used in a system where the main goal is to predict user emotional state based on the referred parameters. That prediction is accomplished with the usage of neural networks based on the user profile.

Next, in Section 3, we will present our approach and justify our option of testing Genetic Algorithms as a possible way to generate the level, particularly in comparison with other techniques for search problems.

3. APPROACH

3.1 Main principles and motivation

As previously referred, the generation process was created focusing, in particular, the videogame *Prince of Persia*. However, we believe that, with proper changes, a similar approach can be used for a generic platform game. The most significant aspect that guided that inspiration is that this game, like many others, has areas represented in a grid. Essentially, each level is composed by cells, grouped in screens of 10 by 3 cells, as it is possible to see in the screenshot provided in Figure 2, where cells have been delimited.

This structure based on cells allows us to think about two main aspects. First, it is theoretically possible to generate all conceivable levels for this game by generating all possible combination of cells. Secondly, it is plausible to construct a system that can test a generated level regarding movement (and possibly some more aspects) and reasonably perceive its quality. Consequently, the main issue is that, in practice, it is not possible to test all conceivable levels. A simple screen where, to make it simple, cells have only three possibilities (empty, wall block and simple floor, as show in the images of Figure 3) consists of 3^{10*3} combinations and, as a matter of fact, one single screen is not much of a level. With this in mind, a stochastic solution appears to be plausible as a way to tackle the problem. In one hand, it would provide different results in different runs and, in the other hand, it provides an adequate sampling on all possible solutions without testing them all. Inside stochastic algorithms and techniques, the usage of Genetic Algorithms appeared as an interesting solution because this is a case where it is not trivial to define an operator to explore alternative solutions. There is no direct perceptible relationship among levels to be represented in a tree as it is complex to define a set of successors for a particular level. Also, the previously referred cell based representation for levels can be mapped with some ease in a structure that can be used with genetic operators, as we will see next.

Figure 2 – *Prince of Persia* – Division in Grid

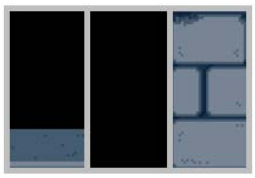

Figure 3 – Three simple blocks that allow the construction of one simple valid level

3.2 Genetic Algorithms overview

As stated before, Genetic Algorithms mimic real life evolution, in particular based on Darwin's theory of Natural Selection. In short terms, this theory states that living beings that fit best their environment are more willing to survive and reproduce. Consequently, their features are reinforced in future generations. Features change over time due to natural mutations and mutual heritance.

In a Genetic Algorithm, one represents Individuals, coded with certain data (genotype) that will manifest some effective features (phenotype), in the same way it happens in nature. To represent evolution, the system has to be able to perceive the inherent quality of each individual. Genetic Algorithms simulate the process of evolution by sorting a set of individuals (a generation) and making the most scored more whiling to continue to the next generation. For this purpose, a *Fitness Function* is defined to evaluate an individual with a certain score. In addition, after a new generation is defined, according to some probability parameters, mutations are applied and some individuals are combined among themselves.

In the next sub-sections we present a possible level representation, a corresponding fitness function and crossover and mutation operators.

3.3 Level Representation

In this system, an Individual is one possible level representation. For this, we adopted a direct genotype/phenotype mapping, which means that coded information represents features directly. Effectively, the implemented genotype represents the whole grid, cell by cell, in a bi-dimensional array. Also, the genotype has explicit representations for the starting and ending cell of the level. The main advantages of this mapping are locality, because it will be possible to perform small changes in a level, and representability, as all solutions have a matching representation. The main disadvantage that can be pointed is that this is the most expensive mapping alternative in what concerns to storage. However, considering the original game and spending one byte for each cell, even a large level is still representable with a few kilobytes, which surely does not represent an issue.

3.4 Fitness Function

To calculate the fitness value for each level, we established a set of heuristics to represent what a possible human evaluation would ponder. The main considered aspects were the following:

- **Path Structure**. The level has to represent a good and immersive path. In particular, it is important to have alternative routes to avoid excessive path linearity, which could result in single closed corridors. Also, it is important to prevent excessive path branching, resulting in a complex maze. To accomplish this, a set of possible moves is defined and access to all cells is calculated, such as moving to adjacent floor cell or jumping through a gap with no more than four cells, among others. In addition, a graph is created, thus it is possible to perceive the cost (i.e. the number of movements needed) to reach any cell from the starting position. This gives a brief perception about the level difficulty. To achieve a more detailed evaluation one needs a more complex alternative. In a previous work [9], we proposed a framework to measure difficulty based on level structure and gap features, which can be an alternative for this purpose. The main issue that this may cause is an increase of complexity for the fitness function, which will result in higher computation times to produce one valid level.
- **Individual cells analysis**. Each cell has a particular meaning and is analyzed individually. The system defines good and bad cells as they make sense or not in the level. A wall cell is always valid. A floor cell is only valid if it is part of any of the possible paths. Finally, an empty cell can be valid if it is used as path (for instance, to create a gap to jump over) or if it has aesthetic purposes. For the last, we defined that an empty cell has aesthetic purpose if it has a valid empty cell in the neighborhood. This specific aspect allows the system to construct levels with open rooms instead of only closed corridors.
- **Ending.** The placement of the level ending cell has to assure, at first, that the level is valid and, secondly, that an interesting challenge was created, consisting on an acceptable cost (i.e. a high percentage of the maximum identified cost). Starting position was not considered because it already has implications on path structure.
- **Aesthetic balance**. To keep the generation with some visual balance, the usage of each particular block should be similar, meaning that the number of used blocks of each type should be about the same. As we stated on individual cell analysis, a wall block is always valid, so this balance forces the evolution process to avoid an excessive use of this type of blocks.
- **Level usage**. The level is supposed to take good use of the provided space, by the means that the full path length should be proportional to the number of cells. This specific aspect reinforces the aesthetic balance, as it favors the definition of long paths, strengthens the definition of a good ending point and supports low branched paths.

These heuristics were applied independently from each other to extract a specific score. To keep control over the range of values, every obtained score is set between zero and one. The extracted individual scores are weighted according to a set of parameters to generate the final fitness value, also normalized to a value between zero and one.

3.5 Genetic Operators

As stated, genetic operators typically consist on mutation and crossover. This sub-section covers the basis of their implementation in our system.

3.5.1 Mutation

Mutation occurs with a certain probability and can be applied in many forms. It is important that mutations are able to make an individual diverge sufficiently to skip local maxima. In our case, we considered the smallest possible mutation as being the change of one particular cell in the grid to another value. Basically, the algorithm picks a random cell and sets it to a random value. In our tests we observed that changing only one cell represents a minor variation and does not provide enough divergence, so one mutation consists on more than one change at a time. The number of changes in each mutation can be tuned, as it is a system parameter. We also implemented two types of mutation, defined as Random and Selective Mutation. Random Mutation simple changes some of the cells in a level, as previously stated. In Selective Mutation we consider that some cells are more suitable to be changed. For instance, isolated floor cells are not aesthetic so they are more suitable to be changed. Also, cells that are not in the main path and are not accessible by any way are more likely to be mutated to a wall block. Naturally, other mutations can be considered as possible ways to improve this operator.

3.5.2 Crossover

Crossover is the operation that blends two (or more) individuals in a new one, as a mimic to reproduction in real life. This operator was implemented to cross elements in pairs. Crossing more than two elements was tested without relevant improvement on the final results. Due to the level structure, based in cells, a simple crossing mechanism can consist on constructing each new individual by taking random cells from another two. However, cells by themselves do not represent much information and should

be considered in relation to the whole level, in particular, to its neighborhood. So, we decided to take mainly into account the more relevant paths that exist in each individual to be crossed, rather than only the isolated cells. When two levels (individuals) are crossed, the main path of the first is kept intact, the main path of the second is also kept intact as long as it does not contradict the first one and, finally, other cells are chosen randomly from one or the other individual. A visual representation of the crossover mechanism is provided in Figure 4 for a small level of 2 by 2 screens. We start by presenting two different levels in the first row and their corresponding path on the second row. The third row presents the overlap of both paths. Cells that correspond to path in both levels are highlighted and, as stated, the attributed value corresponds to the first individual. In the fourth row we added the cells that have the same content in both levels to represent the granted content after the crossover operation is done. Final row presents a possible result by filling the remaining cells taking the value randomly from the first or the second individual, as previously explained. This crossover operator performed better than the simple random selection of cells previously referred, which had a very similar behavior to the mutation operator.

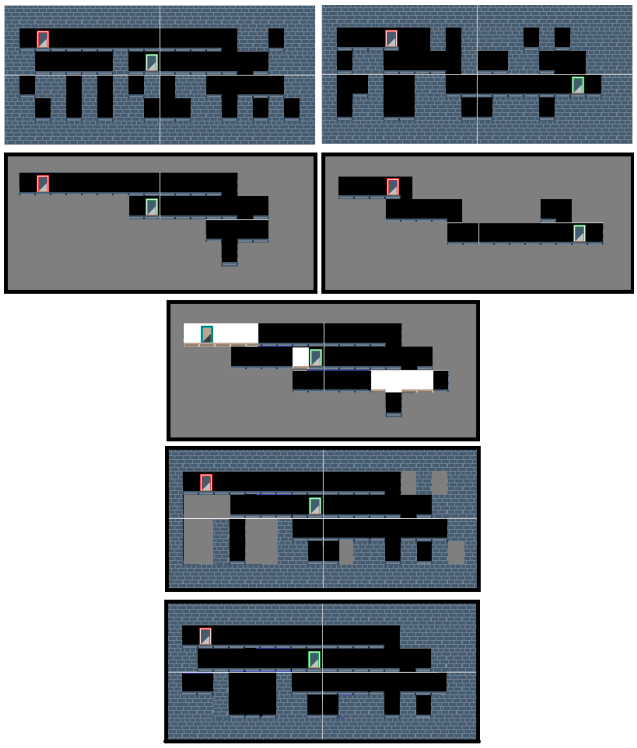

Figure 4 – Example of the crossover operation
top row – original levels
second row – corresponding path for both levels
third row – path overlap
fourth row – common cells added
last row – possible final combination

3.6 Level post-processing

We have focused the generation of valid levels by means of setting the adequate blocks in positions that, in the end, can be interpreted as the level geometry. As an example, Figure 5 presents a level geometry created in our system. However, there is more to consider in a level besides its main geometry. In the particular case of the reference game *Prince of Persia,* scenarios have visual complementary elements on the walls such as torches and windows. This provides aesthetic richness to the scenario. Also, other gaming entities should complement the scenario to make a more diverse and complete gameplay, such as enemies and traps.

These entities are added in a post-processing stage, defined by a simple set of rules. For instance, in each occurrence of *n* floor cells in a row we add an enemy or any particular one-celled trap. Currently, those traps can be floor spikes and intermittent blades. As it is possible to see in Figure 6, this final step produces a good complement to the initial processing phase. This culminates in something that could be, in fact, one interesting level to play.

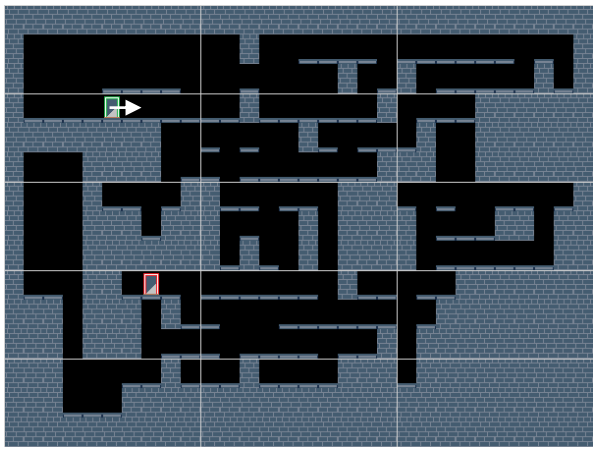

Figure 5 – Example of the generated level geometry
(Green door with arrow = Start position; Red door = Goal)

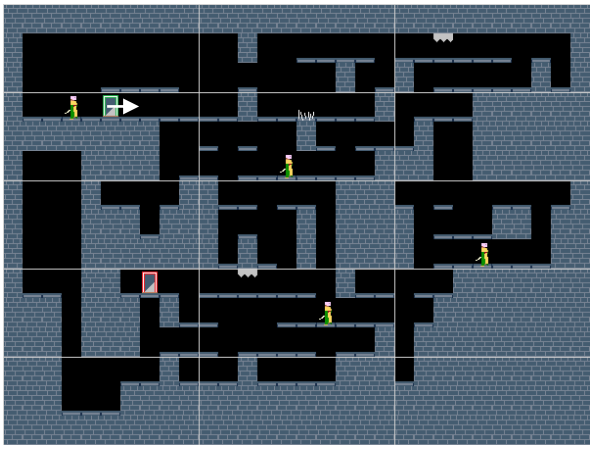

Figure 6 – Example of post processing in one level, automatically adding gaming entities

Other heuristic rules might be applicable such as adding life potions in some optional path cells, substituting empty cells with loose floor if they are adjacent to a floor cell or adding gates and a respective trigger when a path branching is identified in the graph, among others.

4. RESULTS

As previously referred, we implemented a system that does the generation process taking into account the guiding enumerated principles. To give a better notion we provide a screenshot of the prototype interface in Figure 7. In this section we will look at the results that can be achieved with our approach.

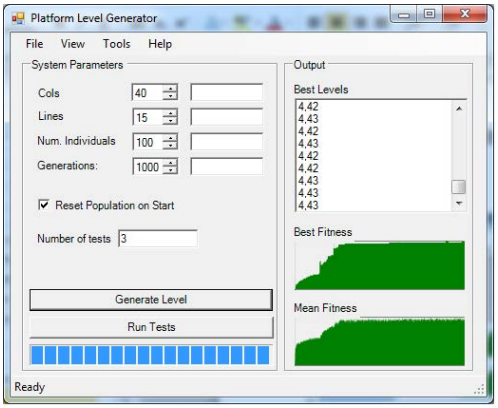

Figure 7 – Prototype screenshot

Our prototype is a program that allows the user to individually configure a set of parameters related to the implemented Genetic Algorithm, such as number of individuals, number of generations and probabilities for crossover and mutation. Our main tests consisted on the following:

- Comparisons on parameter set and theoretical output quality, based on the fitness function;
- Comparisons on generation time vs. level size;
- *Ad-hoc* observation of the results and informal perceptions.

In the first tests, we wanted to understand the differences caused by changing parameterization in relation to output quality, based on the fitness function, and the required time to achieve it. The objective was to perceive the appropriate parameters for further tests and to extract possible limitations. For this, our tests are based on changing the population dimension and the number of generations allowed in the evolution process. A grid of 4 by 5 screens was considered as it represents a reasonable sized level compared with the original reference game. Our application computed 20 evolution processes for each considered combination of population size and number of generations in an Intel Q9300 machine running at 2.5 GHz with 4GB RAM. Table 1 shows the average computational time for those runs, associated to the average achieved fitness. Corresponding standard deviations are also presented. Fitness was normalized to values between zero and one, considering the whole range of values. Theoretically, the worse possible level is scored zero and the best level has a score of one.

Expected trends are extracted directly, such as the increment on the fitness with the growth on population or number of generations. Naturally, increasing any of these values results on in a higher computation time. Within that aspect, the worse presented case on the table shows a computation time of less than 2 minutes for an average fitness of .93. This represents, in practice, correct and reliable levels that could be blindly delivered to the user. Typically, levels with a score over .85 have no relevant flaws or inconsistent content and present an adequate challenge. Without compromising the final results, time can be lowered to less than one minute, resulting in an average fitness of .92. Finally, considering a computational time of no more than half a minute, values near .9 are still achievable. Particular large values besides those on the table were tested as well to verify scalability. For instance, generation processes were tested with 5000 generations of 500 simultaneous individuals, resulting in average computation times of approximately 12 minutes. However, convergence was obtained in the first 2000 generations, for a fitness value of .99 so, in fact, there was no need of such computation.

Table 1 – *Average fitness and generation time for number of generations and individuals parameterization and corresponding standard deviations.*
(Time t in seconds; Fitness f in 0 to 1 range)

Generations Population	200	500	1000	2000
20 individuals	$\mu_t = 0.5$ $\sigma_t = 0.1$ $\mu_f = 0.72$ $\sigma_f = 0.04$	$\mu_t = 1.7$ $\sigma_t = 0.6$ $\mu_f = 0.77$ $\sigma_f = 0.06$	$\mu_t = 4.5$ $\sigma_t = 1.9$ $\mu_f = 0.82$ $\sigma_f = 0.05$	$\mu_t = 13$ $\sigma_t = 4.5$ $\mu_f = 0.87$ $\sigma_f = 0.04$
50 individuals	$\mu_t = 1.6$ $\sigma_t = 0.4$ $\mu_f = 0.76$ $\sigma_f = 0.03$	$\mu_t = 5.6$ $\sigma_t = 3.3$ $\mu_f = 0.85$ $\sigma_f = 0.04$	$\mu_t = 14$ $\sigma_t = 5.4$ $\mu_f = 0.86$ $\sigma_f = 0.04$	$\mu_t = 26$ $\sigma_t = 10$ $\mu_f = 0.89$ $\sigma_f = 0.03$
100 individuals	$\mu_t = 3.3$ $\sigma_t = 1$ $\mu_f = 0.81$ $\sigma_f = 0.05$	$\mu_t = 9.5$ $\sigma_t = 3.3$ $\mu_f = 0.84$ $\sigma_f = 0.06$	$\mu_t = 19$ $\sigma_t = 3.8$ $\mu_f = 0.89$ $\sigma_f = 0.06$	$\mu_t = 51$ $\sigma_t = 22.5$ $\mu_f = 0.92$ $\sigma_f = 0.04$
200 individuals	$\mu_t = 7.5$ $\sigma_t = 3$ $\mu_f = 0.83$ $\sigma_f = 0.07$	$\mu_t = 23$ $\sigma_t = 12$ $\mu_f = 0.87$ $\sigma_f = 0.05$	$\mu_t = 46$ $\sigma_t = 17$ $\mu_f = 0.92$ $\sigma_f = 0.05$	$\mu_t = 102$ $\sigma_t = 39$ $\mu_f = 0.93$ $\sigma_f = 0.06$

For our second test, we simply wanted to state the speed of the program and the consequences of generating larger spaces. For this, we ran a set of tests with fixed parameterizations (1000 generations of 50 individuals) and measured the generation time against level dimension. Again, a set of 20 tests was computed for each combination using the same computer. The average measure time of each level size is presented on Table 2.

Table 2 – *Measured processing times for a certain level size and the corresponding standard deviation.*
(Time t in seconds)

Horizontal Vertical	4 screens	5 screens	6 screens
4 screens	$\mu_t = 11$ $\sigma_t = 4.4$	$\mu_t = 13.6$ $\sigma_t = 6.1$	$\mu_t = 12.3$ $\sigma_t = 5.2$
5 screens	$\mu_t = 12.9$ $\sigma_t = 5.5$	$\mu_t = 14.6$ $\sigma_t = 7.6$	$\mu_t = 14.9$ $\sigma_t = 6.8$
6 screens	$\mu_t = 13.2$ $\sigma_t = 6.2$	$\mu_t = 16.2$ $\sigma_t = 7.4$	$\mu_t = 13.7$ $\sigma_t = 5.8$

As previously referred, the fitness function that was defined to evaluate each level takes into account the study of possible paths inside it. As path calculation may become particularly complex, the main objective of this test was to identify possible limitations and bottle necks. However, computational complexity seems to be linear with level dimension and no particular limitation in this aspect was identified. Naturally, a double sized level will not only result on twice the computational time because more generations will be needed to achieve an acceptable fitness value, as more transformations are expected to occur. Still, it is plausible to think on generating slightly larger spaces without major concerns.

Our final observations are clearly more subjective but are still important and allow perceiving some important characteristics, benefits and issues. In Figure 8 we present a set of examples obtained from our prototype without any particular parameterization (size was chosen to best fit a column in the article and generation time is less than a minute). Basically, we ran the system four consecutive times and those were the obtained levels without any particular selection, post-processing or tune. Relevant empiric common sense insights can be stated. A first impression allows perceiving that outputs are valid game spaces in their basic structure, which is the main goal of all the work. Several other outputs were generated and, in the end, we only came with a few sporadic cases of unrealistic levels, even though they were all possible to complete. Also, it is particularly interesting to perceive diversity in the provided examples. The first level presents a branched path with a maze of tunnels. This opposes especially the fourth example, where the path is nearly direct. Movement trend is also different in each case. Last level focuses mostly on running but, in the third case, there is a strong emphasis on climbing after an initial long run. Open halls are also created to serve different visual purposes. In the second level, the major hall on the left represents a possible big dramatic fall for the avatar. In the third and fourth examples the halls represent high ceiled zones. Finally, even the global structure is varied. For instance, in the second case, practically the last column of screens could be discarded, as the action takes place on the rest of the game space. In the last two cases, the top row screens are the ones that could be possibly discarded without any particular impact on gameplay or level appearance.

All these aspects allow us to perceive that this method represents an interesting way to provide different, varied and playable platform levels.

5. CONCLUSIONS AND FUTURE WORK

In this document we presented our study on the automatic generation of game spaces for platform videogames with the usage of Genetic Algorithms. Our main objective of proving this as a potential alternative for the generation of game spaces for platform levels was successful. Our results are levels that are valid and that could be exported to an engine and played. The implicit rules forced by the calculations in the fitness function makes the process to converge, at least, to a physically valid level in a matter of seconds. Considering a processing time of not more than a couple of minutes the outputs are not only valid levels but have also a balanced structure representing a good challenge.

Comparing to the existing techniques for similar purposes, the presented approach brings advantages concerning level variety. Existing alternatives, presented in Section 2, focuses side scrolling action, typically from left to right. In our levels, the solution is not straight and sometimes not even unique, which allows usage in other variants of platform gaming. Still, simple side scrolling action levels can be achieved with proper parameterization. In addition, the graph structure allows level complementation. For instance, optional path zones may have bonus entities that the character may gather.

In order to make more effective tests to the produced levels and extract several user related aspects, a simple game prototype is planned for further developments. This will allow retrieving users' opinions as well as gameplay metrics that may tweak the generation process as a feedback system.

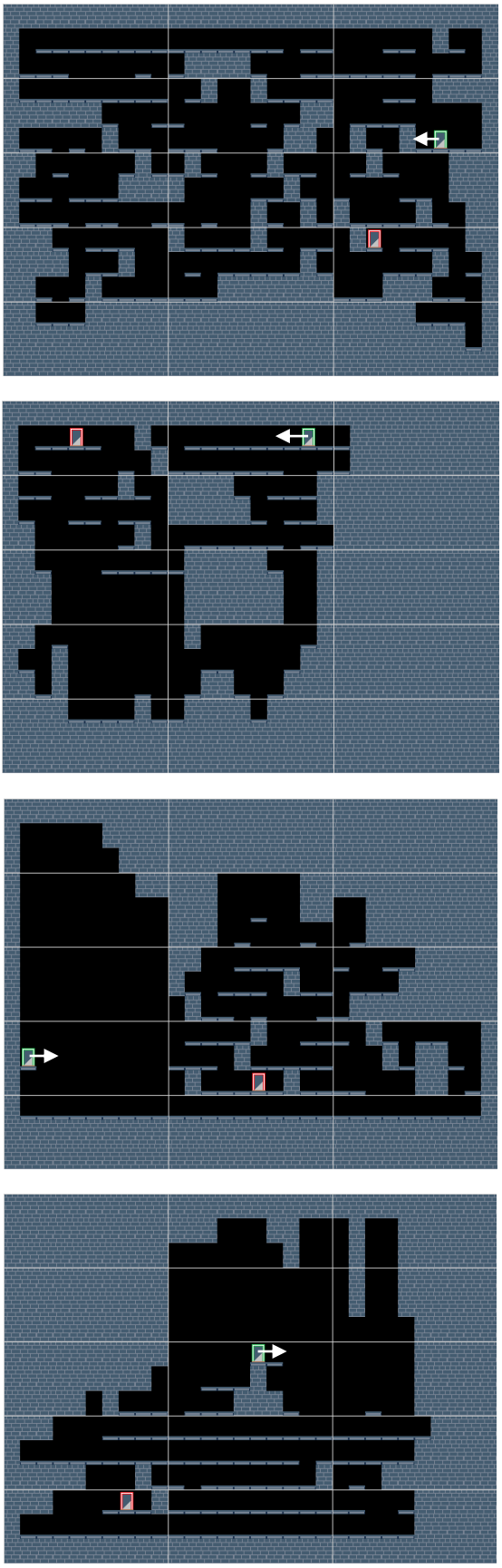

Figure 8 – Example of generated levels

We verified fast convergence to valid solutions with no particular flaws that would render a level unsuitable to be blindly provided to a user. Still, this fast convergence represents the algorithm going in the direction of a local maximum. For the main purposes of the created system, which is the fast generation (similar to videogame loading time) of a possible level to be immediately played, it does not represent a problem. The system focuses in one direction and finds a good solution based on that.

Some of the presented tests focused performance, namely to extract generation time under certain conditions. Even though it is possible to achieve interesting results in short times, the obtained levels with more computation time showed better features in individual details. One possible aspect to consider in the future to improve the obtained generation times is the usage of parallel computing, which is not new on Genetic Algorithms. In fact, effective implementations already exist, such as the *Parallel Genetic Algorithm Library* (http://sourceforge.net/projects/pgal/) available at the *Source Forge* community.

Likewise, one aspect to consider in the future, even though it does not provide direct improvement on the results, is to expand the set of objects in the system. We plan to add other gaming entities available in *Prince of Persia*, such as lose floors, opening and closing gates, among others. However, this is envisioned to be a post-processing step, such as the presented case of the already considered additional gaming entities. Controlling this process may also allow adjusting the level difficulty within a certain range, by adding more or less entities. Naturally, it is also important to verify that the added entities do not invalidate the level. For this, simple concepts may be applied. For instance, any empty cell where the user is supposed to fall may be transformed in a loose floor. As another example, when the path splits in two, this post-processing step may create a closed gate in one branch and a switch to open it in the other one.

In the same manner, more aesthetic items can be considered, such as torches, windows, hanging rugs, among others. To give practical use to the presented work, and since the system is able to generate levels that could be played, it is planned to set a way for those levels to be effectively played. One possible way is to export the outputs to the original *Prince of Persia* format. This alternative is possible and there are even communities where the original game is customized with user levels, sprites, etc. For instance, the *Princed Project* community provides several tools for the process (http://www.princed.org/).

Finally, it is important to refer that improvements can always be achieved in the future by doing optimization in the Genetic Algorithm itself, with additional parameter tuning and by adapting employed evolution techniques, namely the fitness function and mutation and crossover operators. We also intend to tackle these aspects in the near future gathering additional experts knowledge.

6. ACKNOWLEDGEMENT

This work was partially funded by Instituto Politécnico de Setúbal under FCT/MCTES grant SFRH/PROTEC/67497/2010 and CITI under FCT/MCTES grant PEst-OE/EEI/UI0527/2011.

7. REFERENCES

[1] Compton, K., Mateas, M. 2006. Procedural level design for platform games, In *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment International Conference* (AIIDE).

[2] Csikszentmihaly, M. 1991. *Flow: The Psychology of Optimal Experience*. Harper Collins, NY.

[3] Goldberg, D. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley.

[4] Holland, J. 1975. *Adaptation in Natural and Artificial Systems*, Ann Arbor, University of Michigan Press.

[5] Lord, K., Brown, R. 2005. Using genetic algorithms to optimise triangle strips. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (GRAPHITE '05). ACM, New York, NY, USA, 169-176.

[6] Lipp, M., Wonka, P.,Wimmer, M. 2008. Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph*. 27, 3, Article 102 (August 2008).

[7] Mandelbrot B., Hudson R. 1982. *The Fractal Geometry of Nature*. W. H. Freeman and Company, New York.

[8] Mawhorter, P., Mateas, M. 2010. Procedural Level Generation Using Occupancy-Regulated Extension. *CIG-2010 - IEEE Conference on Computational Intelligence and Games*.

[9] Mourato, F., Próspero dos Santos, M. 2010. Measuring Difficulty in Platform Games. Interacção 2010 – *4ª Conferência Nacional em Interacção Humano-Computador*.

[10] Pedersen, C., Togelius, J., Yannakakis, G. 2009. Modeling player experience in super mario bros. In *Proceedings of the 5th international conference on Computational Intelligence and Games* (CIG'09). IEEE Press, Piscataway, NJ, USA, 132-139.

[11] Savchenko, V., Schmitt, L. 2001. Reconstructing occlusal surfaces of teeth using a genetic algorithm with simulated annealing type selection. In *Proceedings of the 6th ACM symposium on solid modeling and applications* (SMA '01), David C. Anderson (Ed.). ACM, NY, USA, 39-46.

[12] Smith, G., Cha, M., Whitehead, J. 2008. A Framework for Analysis of 2D Platformer Levels, In *Proceedings of the 2008 ACM SIGGRAPH symposium on video games*, pp. 75-80.

[13] Smith, G., Mateas, M., Whitehead, J.,Treanor, M. 2009. Rhythm-based level generation for 2D platformers, In *Proceedings of the 4th International Conference on Foundations of Digital Game*.

[14] Smith, G., Whitehead, J. 2010. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the Workshop on PCG in Games*, Monterey, CA, June 18, 2010.

[15] Smith, G., Whitehead, J., Mateas, M. 2010. Tanagra: A Mixed-Initiative Level Design Tool. In *Proceedings of the 2010 International Conference on the Foundations of Digital Games* (FDG 2010), Monterey, CA, June 19-21.

[16] Togelius, J., Yannakakis, G., Stanley, K., Browne, C. 2010. Search-based procedural content generation. In *Proceedings of the European Conference on Applications of EC (EvoApplications)*, volume 6024. Springer LNCS.

[17] Togelius, J., Schmidhuber. J. 2008. An experiment in automatic game design. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*.

[18] Zagal, J., Mateas, M., Fernandez-Vara, C., Hochhalter, B. Lichti, N. 2005. Towards an Ontological Language for Game Analysis, In *Proceedings of the Digital Interactive Games Research Association Conference* (DiGRA 2005), Vancouver B.C., June, 2005