Using Graph-Based Analysis to Enhance Automatic Level Generation for Platform Videogames

Article · January 2013

DOI: 10.4018/ijcicg.2013010104

3 authors, including:

Fausto Mourato
Instituto Politécnico de Setúbal
12 PUBLICATIONS **18** CITATIONS
SEE PROFILE

Fernando Birra
New University of Lisbon
17 PUBLICATIONS **23** CITATIONS
SEE PROFILE

Using graph-based analysis to enhance automatic level generation for platform videogames

Fausto Mourato
Escola Superior de Tecnologia – Instituto Politécnico de Setúbal, Portugal
CITI – Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa, Portugal
fausto.mourato@estsetubal.ips.pt, p21748@campus.fct.unl.pt
Fernando Birra, Manuel Próspero dos Santos
CITI – Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa, Portugal
{fbp, ps}@fct.unl.pt

ABSTRACT
The combination of graph representations with level geometry provide additional information that can enhance automatic level generation processes. In particular, perceiving the main paths that are represented in a certain geometry allows content adaptation to apply certain game design patterns, such as the existence of path detours or the inclusion of optional content. This article explores that approach for the specific genre of platform videogames, focusing the adaptation algorithm that we have developed. Starting with a primal level structure and a corresponding graph that sketches the user path, our algorithm detects mandatory and optional path sections and adapts them in order to create more elaborate challenges to the user, forcing detours to gather specific objects or trigger certain events. In addition, we present the graph related analysis that support the referred algorithm. Our experiments showed interesting results on some popular games, where it is possible to observe the previous principles put into practise. The approach is generic and can be expanded to other videogames where similar graph structures can be used.

Keywords: platform videogames, automated design, procedural content generation, human/computer interaction.

INTRODUCTION
Procedural Content Generation (PCG) is an interesting topic in videogames research, which includes different computer science areas such as artificial intelligence, human-computer interaction, among others. It is a pertinent topic as it allows independent developers and small companies to overcome the issue of lack of resources to design game environments from scratch. The increase of mobile and casual gamming expanded even more the videogame market, which brought additional opportunities to that kind of development. Furthermore, automatic generation

processes promotes content adaptation to the user profile, allowing the creation of a level to fit certain preferences, the user skill or other features. Also, PCG brings several interesting challenges to researchers regarding the generation algorithms and all the features that have to be taken into account, such as physical validity, visual aesthetics, game flow and contexts, among many others. In this work, we focus PCG in platform videogames. In this genre, the user controls the movement of a character (or various users control their respective characters, for multiplayer games) in a scenario, performing jumps to avoid terrain gaps and traps and overcoming opponents in a simple manner, such as jumping over the enemy or shooting him with a certain weapon.

Regarding PCG, this type of videogames are interesting to study because its mechanics raises several non-trivial questions in the process of automatic level creation. At first, it is important to ensure that the gaming elements are positioned to provide a valid level where the user can fulfill a goal. In addition, the element positioning must make sense as a whole, resulting in a visually plausible scenario. Also, the level has to represent an appropriate rhythmic set of actions that the user should complete in order to keep him/her engaged. Finally, it is important to take into account that those actions represent a challenge with a certain difficulty that has to be perceived and estimated.

Concerning user interaction, this kind of games is also interesting as it is a genre that is suitable for all types of gameplay and gamers. For instance, classic titles such as *Sonic – the hedgehog* ("Sonic – the hedgehog," 1991) or *Super Mario Bros.* (Miyamoto & Tezuka, 1985), which were designed initially to be difficult and suitable especially to expert players, have recent adaptations based on cooperative play, transforming this genre in some kind of contemporary family games. In the scope of this work, the following platform videogames will be referred in order to support our tests and examples:

- The original version of the videogame *Prince of Persia* (Mechner, 1989), which has unofficial level editors and technical details available online (Calot, 2008), as well as the recently released source code for the original *Apple* version of the game (Mechner, 2012). This videogame has already been considered in research on the topic of computational complexity (Viglietta, 2012).
- *Infinite Mario Bros.*, an open-source *platformer* inspired by the classic videogame *Super Mario Bros.*, which has been used frequently in academia, namely in the development of intelligent agents to control the main character (Togelius, Karakovski, & Baumgarten, 2010; Karakovskiy, & Togelius, 2012) and automatic level generation (Shaker et al., 2010). Recently, it was adapted to use different game art based in the platform game *Super Tux* ("Supertux," 2010), assuming the new name *Infinite Tux*.
- *XRick* ("XRick", 2001), an open-source remake of the original videogame *Rick Dangerous* ("Rick Dangerous," 1989), which combines the free movement of *Infinite Mario Bros.* with closed environments similar to those in *Prince of Persia*.

- *Tux Likes You*, our adaptation of *Infinite Tux* to record gameplay data and use the generators that we present in this article, among other implementations based on some of our preceding studies.

The main motivation of this work is to provide content richness and personalization to platform levels that are created automatically in addition to simple linear gameplay. As we will see later in the next section, where related work is unveiled, there are some interesting approaches on the subject of automatic level generation for platform games. Nevertheless, the main focus of the existing techniques goes typically to the physical validity and the definition of interesting sequences of challenges and actions without any particular meaning or context. The approach described in this paper intends to be complementary as an additional step to any valid geometry generator, providing improvements in the content, such as:

- **Difficulty adjustment**, tuning the level to a certain player profile.
- **Content improvement**, through the establishment of a non-linear path that consists of identifiable tasks such as, for instance, exiting the main path to grab a certain item and getting back to use it, following a set of identified level design patterns.
- **Inclusion of optional content**, as the system detects optional areas, which can be filled with bonus content defined by the game designer.
- **Multiplayer difficulty adjustments**, to adapt levels to be used simultaneously by players of distinct skills.

Our system starts with a coarse level structure, which might be a basic geometry without additional gaming entities, and a corresponding graph based on the avatar's most relevant movements, creating an approximate representation of the players' possible paths. Furthermore, the referred graph is analysed and every vertex is contextualized regarding its role on the objective of going from the starting point to the final position. After this graph analysis, a set of possible modifications is extracted based on level design patterns used in *platformers*. Some changes are iteratively selected from the referred set in order to reach a certain final value regarding level difficulty, dimensions, game style or others.

Regarding the paper's structure, after the presentation of the related work we will start with the description of the initial requisites and premises of this project. Next, we provide details about the graph examination that we perform and its inherent potential. We proceed to the explanation of our adaptation algorithm, which is strongly based on the previous graph examination. In addition, we provide some examples of usage of our algorithm and present the results of some tests that were performed. Naturally, we will conclude with our final remarks about this approach, pointing some guidelines for further developments.

RELATED WORK

PCG has been used in videogames since its early days. The inevitable pioneer to mention in this subject is Rogue (Toy, 1980), an adventure videogame where rooms and corridors are created randomly every time a player starts a game. This approach and the main principles that were used are still applicable in some of contemporary *Role Playing Games* such as Diablo ("Diablo,"

1996). Over the time, the main contributions of PCG in popular videogames can be found in specific gaming elements such as, for instance, the automatic generation of trees ("Speedtree," 2013). The recent success of Markus Persson's *Minecraft* (Persson, 2011), an action-adventure game with procedurally created scenarios, popularized the discussions on PCG has a way to generated full gaming environments. Regarding independent game development *Spelunky* ("Spelunky," 2012) and *Cloudberry Kingdom* (Fisher, 2012) are two interesting titles to refer, in particular because they are both platform videogames with automatically generated scenario. Platform videogames, in particular in the context of automatic content generation, started to be studied in academic context by Compton and Mateas (2006) with an analysis of the main components and the definition of a conceptual model to define this type of games. Later, a more detailed analysis has been presented by Smith, Cha and Whitehead (2008). In this work, authors suggest a conceptual hierarchy to define the entities that compose a platform game level. Those principles were used in the definition of a technique to automatically generate levels based on the rhythm associated to player's input and actions (Smith, Mateas, & Whitehead, 2009), which has been applied in different systems, namely *Launchpad* (Smith, & Whitehead, 2010; Smith et al., 2011), *Rathenn* (Smith, Gan, Othenin-Girard, & Whitehead, 2011) and *Endless Web* (Smith & Othenin-Girard, 2012). The main goal behind this idea is to keep the user in a mind state that Csikszentmihalyi's referred as *Flow* (Csikszentmihalyi, 1991), representing the ideal feeling of control and immersion over a challenging task. It can be seen briefly as a state in-between boredom and frustration, meaning that the task is, respectively, too easy or too difficult. Smith and Whitehead (2010) also studied the expressivity of this approach regarding linearity and leniency, an approximation to the concept of difficulty. Finally, following the same line of work, the system *Polymorph* presented by Jennings-Teats, Smith and Wardrip-Fruin (2010) is an effort to adapt difficulty directly in the generation process as the player is moving forward in the scenario. The presented ideas provide an interesting way to generate good challenges but they tend to be mainly directed to the creation of straightforward games in open scenarios or simple casual game environments. Still, it is possible to have an initial generation step based on those principles and a further step to complement the content as the one proposed on this article.
Another important work to refer with similar features was presented by Pedersen, Togelius and Yannakakis (2010), focusing the already referred game *Infinite Mario Bros.*. Levels are generated according to certain parameters. In particular, parameterization was applied to the following aspects: existing gaps in the level, average gap size, gap distribution entropy and number of direction switches. The main study focused on the concept of difficulty and possible adjustments to fit the user skills but, again, the generation process is directed to the construction of a sequence of jumps without a particular semantic meaning, preventing this method to be directly applied to other games.
A different approach was proposed by Mawhorter and Mateas (2010). The main principle is the composition of a whole level based on small pre-authored chunks, which can be assembled together. This allows a more varied set of outputs, depending on the number, type and variety of chunks that are considered. Although the authors also focused the game *Infinite Mario Bros.*, we

believe that this is a valid principle for other videogames, in particular for the generation of scenarios that represent closed environments with rooms interconnected with tunnels. These structures are likely to be obtained using merely small-sized chunks. For instance, the previously referred videogame *Spelunky* uses very similar principles and consists on that type of environments.

Another example of automatic level generation can be found in our prior work (Mourato, Santos, & Birra, 2011), using evolutionary computation, in particular genetic algorithms, to search for good solutions according to design heuristics applied to a fitness function and operators to perform mutations and crossovers. As this approach can be integrated with the adaptation algorithm that we are presenting in this document, we provide additional details later in the base level structure generation sub-section.

One exceptional case that is directed to complementary content rather than simple movement and jumps was presented by Nygren *et al.* (2011). The authors proposed a method to integrate puzzle based content that requires the player to explore the level. It involves a three step process which consists, namely, in: graph generation, graph to level structure transformation and content creation. In the first step, a graph of possible positions is randomly created using genetic algorithms. That graph is then transformed in a valid geometry that fits those movements, with a search-based process. Finally and again with a search-based process, possible modifications are identified for each level segment from which one is selected to meet certain criteria. Once again, the experiments were directed to the videogame *Infinite Mario Bros.*. In some of the results it is possible to notice the existence of multiple alternatives and paths that lead to a dead-end, thus creating an exploratory theme. In our approach, as we will further see, similar situations are identified but, in this case, the adjustment algorithm takes advantage of those features to compose additional challenges and force exploration.

SYSTEM PREMISIS

Initial conditions and content

As previously referred, our algorithm is based on graph examination. This means that, in addition to the basic level structure, it is required to have a graph representation of that structure or a method to obtain it. The main advantage of our approach is that we not only identify alternative paths and detours but also use them to place additional gaming entities to encourage exploration.

As a starting note, it is also important to clarify that we are working with directed graphs, as some transitions might be unidirectional, such as the character falling into a hole, which has no way back. To provide a clearer notation, we will avoid mixing representations. Therefore, all edges will be considered to be directed and the cases of undirected edges connecting vertices v_1 and v_2 will be represented as a directed edge from v_1 to v_2 and another directed edge from v_2 to v_1. Besides the notation issues, in some cases this distinction would be also mandatory because costs from v_1 to v_2 and from v_2 to v_1 might be different, as they may represent, for instance, a difficulty measure.

Level representation

The algorithm that we propose is intended to be generic and suitable for the majority of platform videogames, as it works with high level concepts. However, even within a specific genre, each different videogame has its own specificities and content is represented in diverse ways, albeit the similarity of concepts. This raised the need of having a representation scheme that could describe levels from different games. Otherwise, it would not be possible to conceive a generic technique. For this purpose, we have defined a level representation framework that provides a way to describe the content of a certain *platformer* and create levels within that same representation. We have assumed that levels are represented in a grid, as this is the common approach to define levels within this genre and most PCG algorithms that were presented in the related work also lay in this type of representation. Therefore, a level can be described as a two-dimensional array of cells (or tiles). Following that representation, a game has a certain set of blocks that can be used to fill the level cells, which will be referred henceforth as the level cell set. In addition, each block in the cell set defines a collection of properties that can be established in each instance of a block in a cell.

To allow a better organisation of the existing blocks, the cell set is structured as a hierarchy, meaning that a certain block can be decomposed recursively in sub blocks that are a certain particularization of their parents. For instance, different types of wall can be grouped under a general wall block. In addition, a categorisation mechanism was implemented allowing the definition of categories that represent a certain set of blocks, obtained by applying logical operators to the hierarchy, which can be useful in ambiguous situations. As an example, in *Prince of Persia* one can find floor tiles that break after being stepped by the character, meaning that they can be used as regular floors for a limited time and as empty spaces afterwards. As there is no clear classification in the main block hierarchy, this block can be represented isolated. In the categories definition, empty cells will be described as the main empty cells represented in the hierarchy joint with the loose floor blocks. A similar joint will be performed to unite those loose blocks with regular floor blocks. These two organisation mechanisms have two main goals. In one hand, they are intended to improve the usage of this framework within a manual level editor, providing a better interface to select blocks, which is naturally easier having a grouping mechanism in comparison to a plain list of blocks without any particular structure. In the other hand, this additional information about the structure of a level may serve automatic generation processes as we will further see in this document.

At this point, the intent of this representation scheme is purely structural. This means that all semantic content, besides element relative positioning, which are explicit described as the array of cells, can only be interpreted implicitly and the effective meaning of that content can only be confirmed in the corresponding game engine. This abstraction can be described recurring to the original version of the videogame *Prince of Persia* and the more recent remake *Prince of Persia Classic* (“Prince of Persia Classic,” 2007) that presents the original game rebuilt in a three-dimensional engine. These videogames have identical level structures, as the hero will travel

equal paths and face opponents and traps in the exact same positions. Regarding our representation framework, with one single description it is possible to represent levels for both games but, in the end, they are responsible for the effective meaning of the described concepts. For instance, in this case, the implemented fighting mechanism differs from one game to the other, even though it lays on the same element types and positioning. Another example regarding the abstraction of the used representation is to think of a game with altered gravity. For instance, we can think of a two-dimensional *platformer*, where the gravitational force pulls objects from right to left. There is nothing in the previous representation that would forbid this implementation but, as explained, it is the effective usage of this mechanics in the game what brings semantics to the structural information, making the cells represent tunnels, gaps or other types of features.
As graphs will be used as complementary information to the level structure, naturally the main physical rules or concepts will have to be included somehow. That is explained in the next subsection of this document.

Automatic graph extraction

We have presented the main requirements for the algorithm to work. Those are: a previous existence of a certain level geometry, which can be a sketch of the main structure without specific gaming entities; and a graph to represent the main movements within that structure. If a level is created by a designer, it would be possible to ask him/her to describe the possible main movements within that level in a graph structure. However, to use this type of approach in a system that generates levels from scratch and, therefore, without that possibility, it is important to think that manual graph creation is not an option. Consequently, it is important to have a method to create automatically those graph representations based on a level structure. A plausible solution would be the usage of physical simulations based on the game rules (or the game engine itself) to calculate all possible movements for the avatar. However, this would be a complex and time consuming task which could compromise online level generation. In addition, this type of simulation would require different implementations for every distinct game. In this subsection, we present two alternatives for the automatic graph extraction that could be used to support the adaptation algorithm without the previously referred issues, and that can be configured to work with distinct videogames.

Graph extraction based on pre-defined rules. The previous representation allows mapping the structural information of a level and, eventually, its implicit respective visual expression. However, it lacks of semantic description that could allow the inference of gameplay data. For instance, it is not possible to perform a simple validity test to check if there is a valid way from the beginning to end of the level. In a common design situation, this is not an issue, as one can assume that the designer will have knowledge and perception to create correct scenarios and that a further testing phase can also confirm that validity. Nevertheless, in an automatic level

generation scenario, the human perception is not available and levels are typically used without a testing phase.

To overcome the previous issues, the formerly presented representation framework has been expanded to allow the description of elementary avatar movements within a level, based on the cell structure. This is done by associating certain cell patterns to sub graphs that represent the avatar movement for that specific situation. In Figure 1 it is possible to observe two example rules used to define the movement for the videogame *XRick*. On the left part, it is possible to observe the rule to define the horizontal movement of the character and, on the right part, the movement among ladder cells is represented.

Figure 1 - Example of two rules for graph construction, consisting on a pattern and the corresponding graph entry

For a certain game, a set of pattern matching rules can be defined. The complete set of rules is applied by going through the entire level grid searching for matches. This process has showed to be particularly efficient in situations with small range of movements, such as those with *rotoscopic* animations. Figure 2 shows a sample of the first level of the game *Prince of Persia* (on the left) and the corresponding graph (on the right) that will be considered on the analysis. The mentioned graph was obtained with the referred cell based pattern matching approach using about twenty rules.

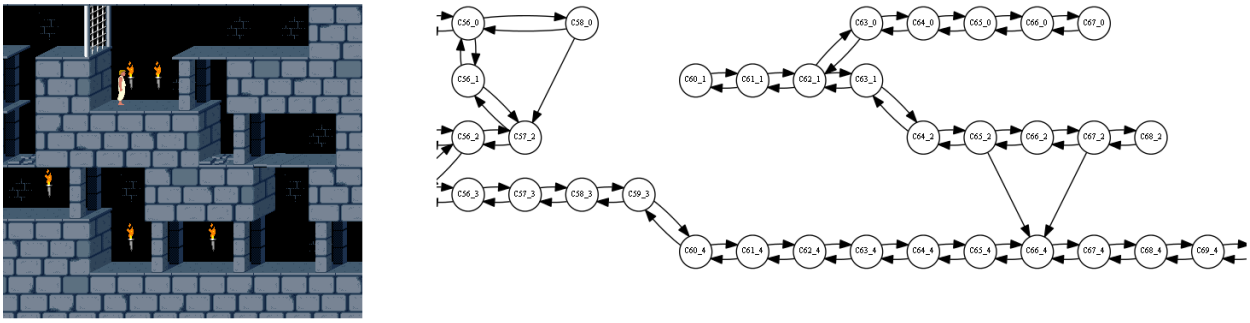

Figure 2 - A sample of the first level of the game Prince of Persia (on the left) and the corresponding extracted graph (on the right).

Graph extraction from example levels. As an alternative to the previous method for graph extraction, we have also tested the possibility of having the system automatically learning the most relevant rules to be useokd to sketch the avatar movement, based on gameplay testing sessions. This can be useful in situations where the character has a wide range of motion that is difficult to express within a reasonable set of rules. We believe that a possible scenario of usage for this alternative is the case where the game has already been created with a set of humanly designed levels and one wants to add new game content automatically to expand the current set

of levels without an additional designing effort. Currently, we have tested this alternative with our prototype *Tux Likes You* with some interesting results. The game was configured to record the avatar positions each time the user played the game. Representing the recorded positions into the map allows identifying the main areas where the player was positioned. Figure 3 shows an example level sample on the left part and, on the right, it is possible observe the most common regions for the avatar positions. Areas where the avatar rarely passes by were faded out.

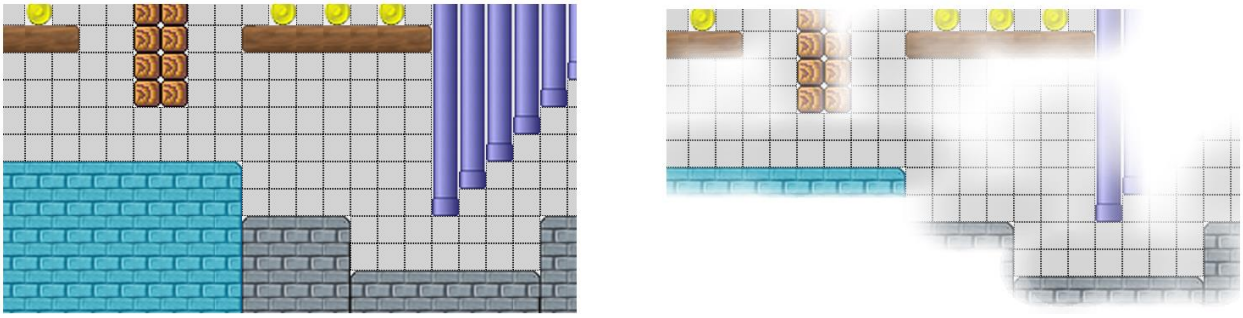

Figure 3 – Gameplay mapping of avatar positions.

In addition to the simple area analysis, we have detected which movements represented cell transitions and used that information to create level graphs. In Figure 4 we present, for the same level situation, the obtained graph for the avatar's movements.

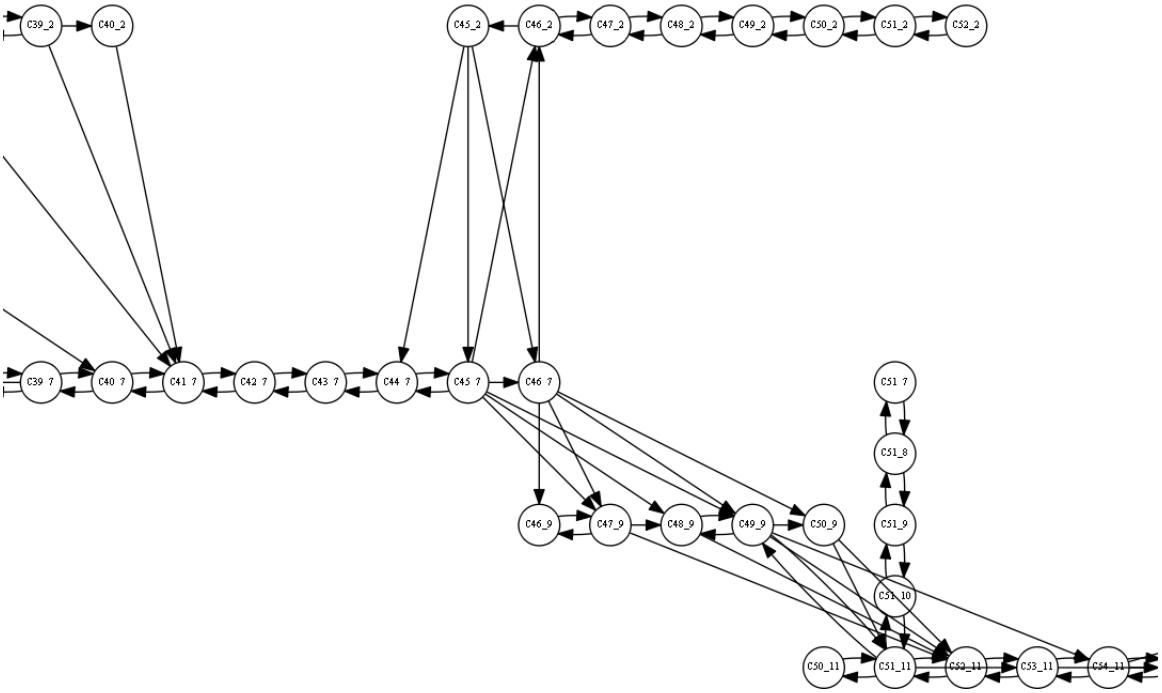

Figure 4. Obtained graph based on gameplay data

The generated graph and the original level structure allows the automatic extraction of some movement rules. This can be achieved by analysing the cell neighbourhoods in pairs of vertices that represent transitions between equally spaced cells. For instance, considering only this small level sample, our system detected that transitions between side by side cells occur frequently in

the situations presented in Figure 5. This means that those situations represent potential rules for that horizontal movement. To confirm those rules, the potential situations are searched in the whole set of levels and, if whenever those situations occur, the corresponding graph entry can be found, then the rule is confirmed. In the example case of Figure 5, all rules are confirmed except the third one, since there are several occurrences of that pattern without a corresponding graph entry.

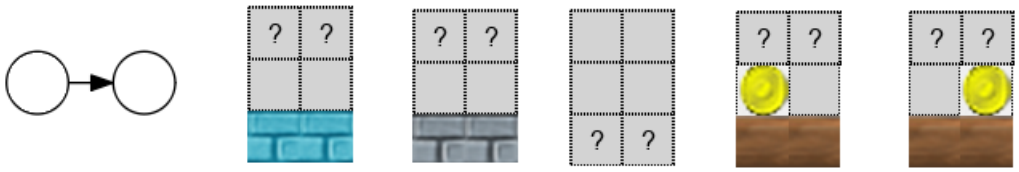

Figure 5. Example of five extracted situations that are frequent for the represented sub-graph.

In this process, we have detected two major issues that could compromise the usage of this technique. First, even though it is possible to map the majority of the level with just a small number of gameplay sessions, it is particularly hard to achieve global coverage. Small level parts tend to be unused even though they are reachable. In second place, a simple bug in a gameplay session can compromise the whole reasoning. For instance, if the avatar crosses a solid block inadvertently, that move is likely to be assumed as plausible, which may lead to inaccurate conclusions. To solve both the previous situations we transpose to this domain the concepts of a low pass filter. The erratic detection of transitions due to inadvertent moves, caused by possible game implementation bugs, can be corrected by ignoring transitions that only occurred sporadically (formally, with a percentage of existence under a certain threshold). In the same manner, the confirmation of potential rules should not search for 100% matches but a value over a certain threshold, which should be tuned according the existing data.

Base level structure generation

As formerly referred, the technique hereby presented intends to perfect a previously created level. We believe that this kind of adaptation technique is particularly useful in a fully automated generation system. In one hand, this allows reducing the generation complexity in a first step, as the whole generation process is divided in multiple stages and several constraints can be ignored in primal stages. In the other hand, this type of adaptations is a way to provide content reutilization as one single valid geometry that is produced can be adapted multiple times with different results. For instance, in the videogame *Prince of Persia*, with the same basic level geometry, it is possible to define different levels just by placing gates and triggers to open them in different positions.

Next, we will present two alternatives that we tested for the initial basic level creation and explore how the usage of graphs can also be used to improve them.

The first alternative is more suitable to levels consisting o closed spaces, by which we will direct our examples to situations in the game Prince of Persia. For situations with more open scenarios

our second solution is more adequate, and we will use examples mainly in our implementation of *Tux Likes You*.

Genetic Algorithm Level Generator. For our initial tests, we have generated basic level structures using a genetic algorithm implementation from some of our previous experiments, as referred in the related work. Basic level structures can be created applying heuristic design patterns in the individual evaluations. In our case, we implemented our algorithm to evolve levels guided by the following heuristics:

- Paths should represent some alternative routes but avoid excessive branching that would result in complex mazes;
- A level should have a solution using the majority of the created geometry;
- Cells should be logically correct when interpreted within their neighborhood;
- Available blocks should be used with similar proportions;
- Level space should be used evenly.

In particular, the implementation of the first two heuristics were adapted from the initial work to use the graph structures that we have previously presented. That allowed the direct usage of the explained movement rules, making the principle more generic. Since this approach has showed to be more efficient in closed environments, it has been used in our test for the videogame *Prince of Persia* and *XRick.*

Simple Overlapping. For the creation of basic open scenarios, such as those in *Infinite Mario Bros.* or *Infinite Tux*, we implemented a simple composition algorithm to build levels based on a small set of examples. This algorithm is suitable for the generation of levels where practically all gameplay consists on going in the same direction, which is the case of the aforementioned games. The creation process consists on generating content from left to right. Starting in the beginning (left part) of a random level, the algorithm copies sequentially its columns until it reaches a column that occurs in one or more levels from the example set. In these cases, which we refer as transition points, the copying process may shift to one of the matches at the corresponding column (a shifting probability is established). The process continues until the level achieves a desired length.

In our tests, we detected one main issue in the usage of this approach directly that, however, can be fixed recurring to our movement rules that were previously presented in this document. This problematic case is the creation of impossible game situations due to incompatible transitions. For instance, a common and natural transition point occurs in gaps, as they consist of multiple columns composed only by empty cells. This match is a sort of wild card situation (every gap matches another gap) that breaks the main principles of the overlapping approach. Without any additional conditions, the presented algorithm may shift multiple times from level to level in gap parts generating an enormous gap that the character cannot jump across. These situations could be solved by preventing level shifting in gaps. However, this would not be an elegant solution as it consists on a particularization in something that was meant to be generic. Additionally, this *ad-*

hoc solution would not prevent the occurrence of similar situations in *implicit gaps*, meaning situations where the avatar has obligatorily to jump between two separate platforms, such as the situation represented in the left part of Figure 6 (the two higher platforms represent an implicit gap).

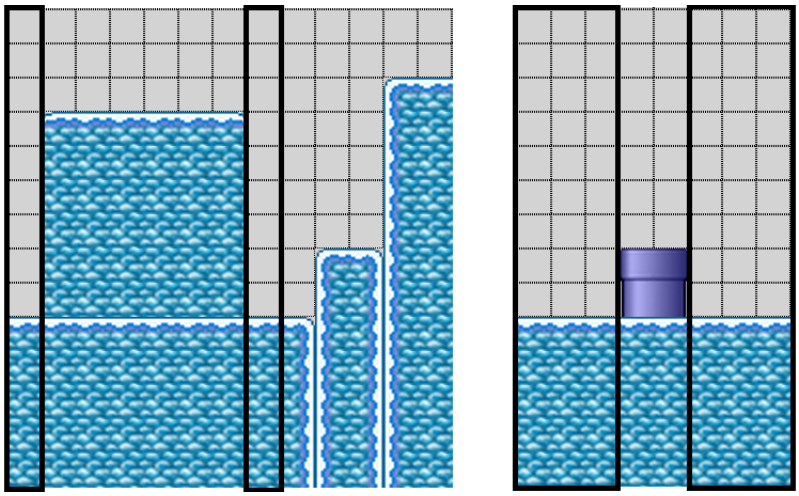

Figure 6. Example of two arbitrary level parts that allow shifting. Equal columns are marked with rectangles.

Considering that same figure, which presents two arbitrary level situations, it is possible to verify that columns 1 and 8 in segment A matches the first and the last three columns in segment B. Considering possible shifts from A at column 8 to B at column 1 and B at column 8 to A at column 8, the result would be the impossible situation presented in Figure 7.

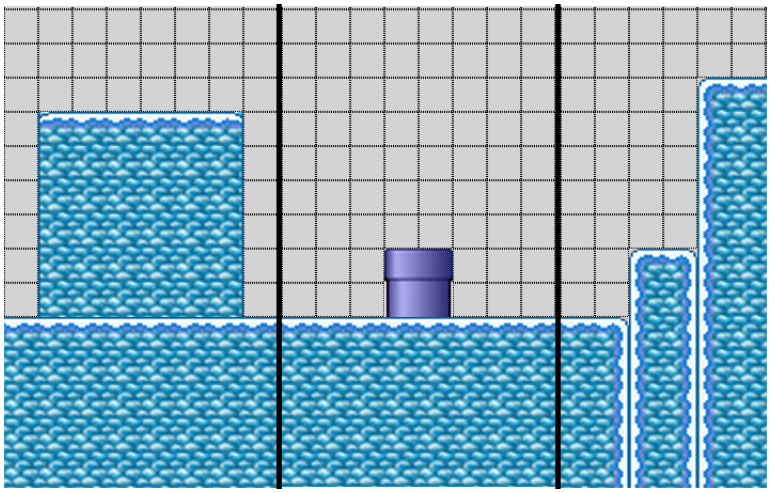

Figure 7. Example of a wrongly generated level. The transitions are marked with the strong vertical lines.

The usage of graph information allows us to overcome this issue. Every time that a new section is generated, the movement rules should be applied to obtain the new level graph. As the left part of the level is still the same, there is only the need of applying these rules to the newly generated part and an additional buffer of previous content to map the transitions between the new segment and the remaining of the level. As long as new segments allow reaching the right part of the

level, the algorithm can continue. Otherwise, the last created segment must be ignored and the process should rewind to the last shifting process.
In fact, with this final adaptation, this technique uses similar principles to those in the chunk based approach previously presented, but working with particular large chunks and using an automated mechanism to obtain them and to tag relevant positions.

GRAPH ANALYSIS

So far, we have presented our approaches for generating basic level geometries and how to automatically associate graph representations to those levels. We will now cover the main graph processing stage that will allow the usage of our algorithm.

Initial graph processing

In order to reduce the computational effort of the algorithm, the first step is a graph compression that results on a reduced version with a smaller amount of vertices and edges. This compression involves removing vertices that are not significant to path computation processes and that can be seen as obvious intermediate steps in major transitions. In particular, the two following rules are applied:

- If a certain vertex v has exactly one incoming edge e_0 (from v_0) and one outgoing edge e_1 (to v_1), this means that, regarding path calculations, v is just a transitional step from v_0 to v_1. In this case, vertex v and edges e_0 and e_I are removed from the graph structure. A new edge is created directly from v_0 to v_1 with a cost corresponding to the sum of the previous costs defined for e_0 and e_I.
- If a certain vertex v has exactly two outgoing edges e_{O1} and e_{O2} (to v_1 and v_2, respectively) and two incoming edges e_{I1} and e_{I2} from the same vertices, this means, in a similar way to the previous rule, that the vertex v is an intermediate step in the connection between v_1 and v_2 in both directions. Again, this vertex is removed and the previous edges are also replaced by one edge from v_1 to v_2 with a cost value summing the costs associated with e_{I1} and e_{O1} and another edge from v_2 to v_1 with a cost that sums those from e_{I2} and e_{O2}.

An example of the compression mechanism is provided in Figure 8, which contains a compressed version of the original graph represented previously in Figure 2.

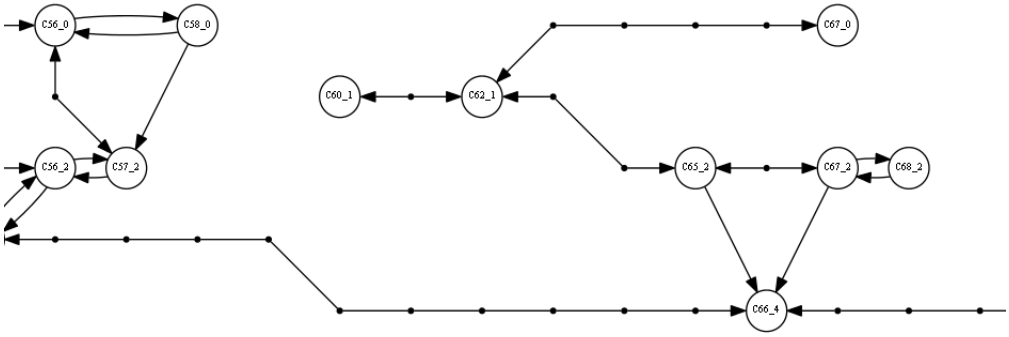

Figure 8. Example of a compressed graph. The compressed nodes are represented with dots.

Some restriction might apply in vertex removal. For instance, vertices corresponding to the start and end position should not be eliminated, as they have an active role on the level. This was implemented with the definition of a set of irremovable vertices.
In addition, some particular compression schemes might be deliberated depending on the game that is being considered. For instance, in the videogame *Prince of Persia*, it is common to have the situation represented on the left part of Figure 9, which will be represented by the corresponding graph that can be seen on the right part of that same figure. The triangular shape represented on the graph image by the three interconnected nodes could be merged into a single node without compromising path calculations. As another example, with a simple rule set, hill platforms in the game *Infinite Tux* will tend to create excessive vertical movement alternatives and prevent vertex compression, such as the case presented in Figure 10. Vertices corresponding to the middle cells of the hill platform could be removed without compromising path analysis.

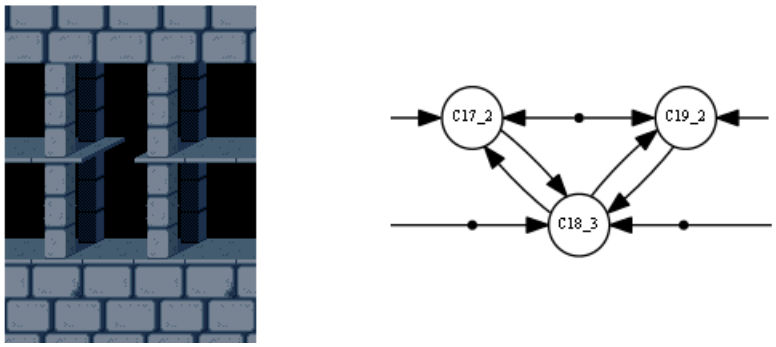

Figure 9. An example of a potential additional graph compression in the game Prince of Persia

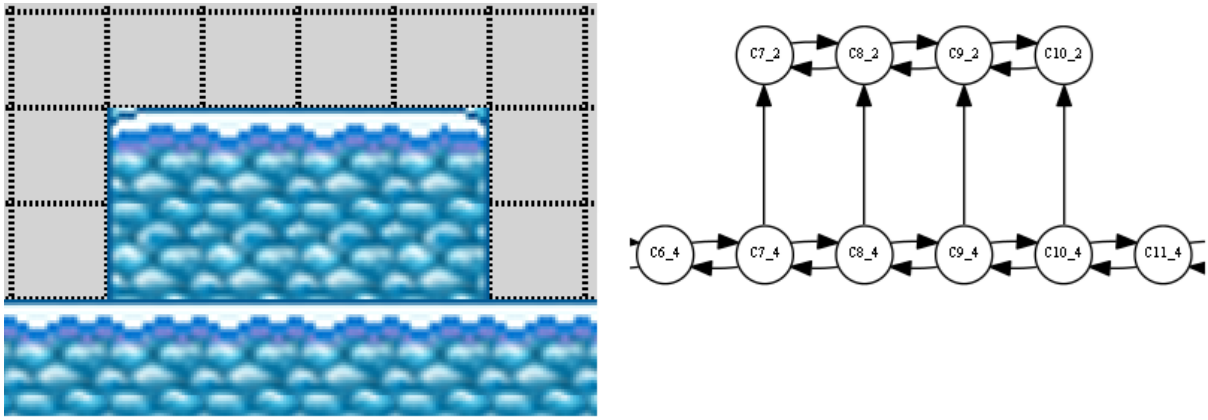

Figure 10. An example of a potential graph compression case in the videogame Infinite Tux.

Path extraction and vertex classification

Having a compressed graph representing the level, the next step consists on categorizing the meaning of each vertex considering all routes between the start and the end of the level. The

algorithm searches every possible path from the level entry to the level exit, using a breadth first search and ignoring all alternatives that visit the same vertex multiple times.
Following this step, every vertex is labelled with one of the following values:

- **Mandatory**, if the vertex exists on all calculated paths;
- **Optional**, if the vertex is only on some of the calculated paths;
- **Dead-end**, if the vertex is not part of any possible path and has only one outgoing edge to a certain vertex and one incoming edge from that same vertex, meaning that if the character reaches that position he/she has obligatorily to go back;
- **Unreachable**, if the position that corresponds to the vertex cannot be reached by the game character;
- **Vain**, if there is no way back from that vertex to the main path;
- **Path to Dead-end**, to all remaining vertices that, by exclusion, have the only purpose of providing passage from mandatory or optional vertices to a dead-end.

In addition, a tree is created representing path segments and alternatives for each segment, recursively. Leaf nodes represent trivial graph transitions. In Figure 11 we present an example level, the corresponding generated graph and the tree that was created representing the possible path alternatives within that level.

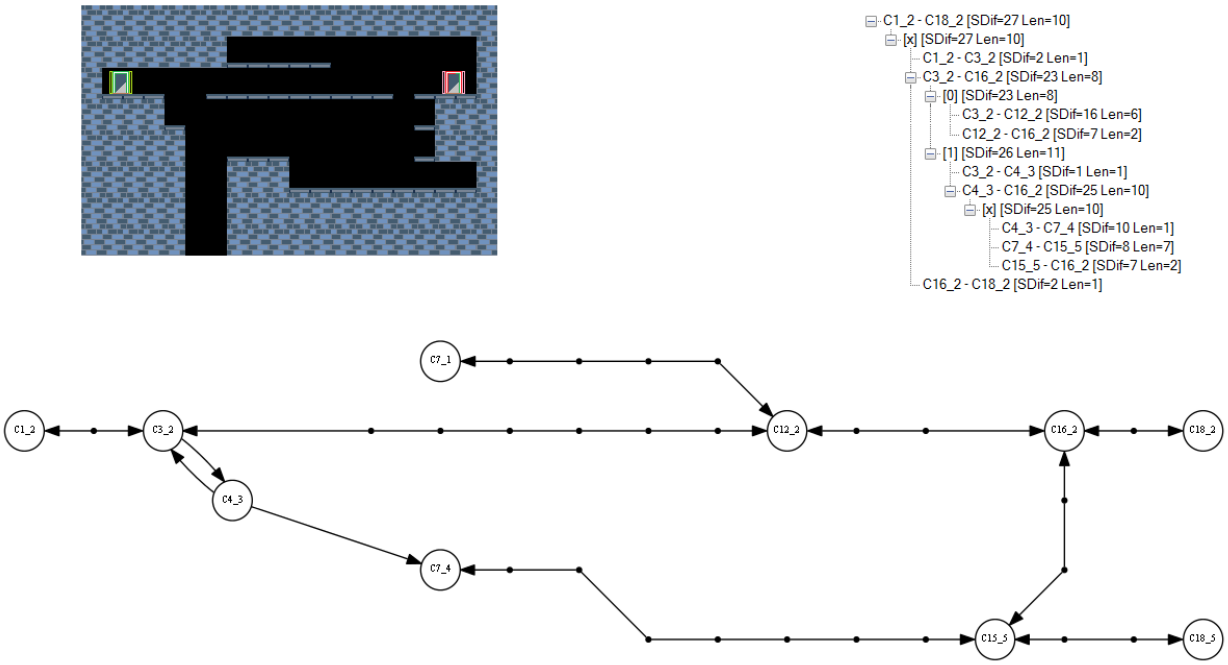

Figure 11. Example of a level, its correspondent graph and the respective segment decomposition represented as a tree.

ITERATIVE CONTENT ADAPTATION ALGORITHM

The level completion algorithm adjusts the level based on estimated difficulty and similar related aspects. For the definition of the algorithm itself, the exact concept of difficulty may remain undetailed and be implemented as the game designer considers best. The key idea that has to be present is that difficulty arises from base geometry such as jumps over gaps, from gaming entities, such as enemies and traps, and from path structure, which forces the user to accomplish additional jumps and to encounter additional gaming entities. By design, the algorithm will have little control over jumps, as adding or removing gaps would require graph recalculations. Hence, it will have a higher focus on the control of additional gaming entities and the definition of a composed challenge.

Difficulty has been characterized and estimated in distinct ways. It can be seen as a success probability (Mourato, & Santos, 2010) or a sum of coefficients (Smith et al., 2011), among other alternatives.

For the purpose of this algorithm, the requisites are the following:

- A higher difficulty value means that the level is more difficult than another with a lower value.
- Every level segment can be analysed individually to produce a difficulty value.
- A succession of analysed sections with certain difficulty values produces a final difficulty value for that succession.

In the tests that we will present, we have considered a sum of coefficients approach, where every graph transition contain a certain value that can be estimated by the game designer or mapped to users' performance after some gaming sessions. Moreover, when the path section presents two or more alternatives, the difficulty value that is considered is the lowest, as we assume that the user would pick the easiest possible solution.

The algorithm is able to apply the following changes to the base level:

- **Change the difficulty of a segment**, which is done by adding or removing an enemy or a trap or making a gap larger or smaller. Again, this is achieved with a pattern matching rule set. The graph is kept the same but a parallel data structure stores the entities that have been added to each section and the changes that have been performed.
- **Detour creation**, which consists on identifying a path to a dead-end from the main path, using the previously referred vertex classification, followed by adding a certain item in the path to the dead-end and making it required on the main path.
- **Cooperative two player game adjustment**, consisting on identifying two parallel alternatives for one particular section and using the previous two principles to adjust each alternative individually for each player. In addition, the game should contain a method to prevent both players to follow the easiest alternative.
- **Bonus entity addition**, which consists on adding collectibles or minor power-ups on certain dead-ends, creating secondary goals for the player.

The previous changes occur based on probabilistic coefficients that are established by the system after analysing the following values:

- Total desired difficulty value (whole level);
- Mean desired difficulty value (per segment);
- Player state estimator for game state on each graph vertex, detecting periods of possible boredom, flow or frustration.

The algorithm works iteratively in multiple adjustment passages, with the level difficulty being analysed for each passage. The process stops when it reaches the desired value or a limit on the number of iterations. At the end of each step, the previous features are analysed and the probabilistic coefficients are defined for each path section. For instance, if the level is too short then the detour creation probability in each section is increased.

RESULTS

Examples

The next example shows how a simple level can be tweaked by the system. We will consider the game *Prince of Persia* in particular because of the possibility of adding gates and step switches to open those gates, which is an interesting implementation of the referred detour creation concept. We have manually created the level structure represented in the background of Figure 12, which is just a simple draft. The level does not contain enemies, gates, switches or any traps. The starting point is the door on the left side and the level ends when the character reaches the door on the right. The system extracted and compressed the level graph presented as an overlay on that same image. Nodes have been named according to their coordinates on the grid, also marked in the image for convenience.

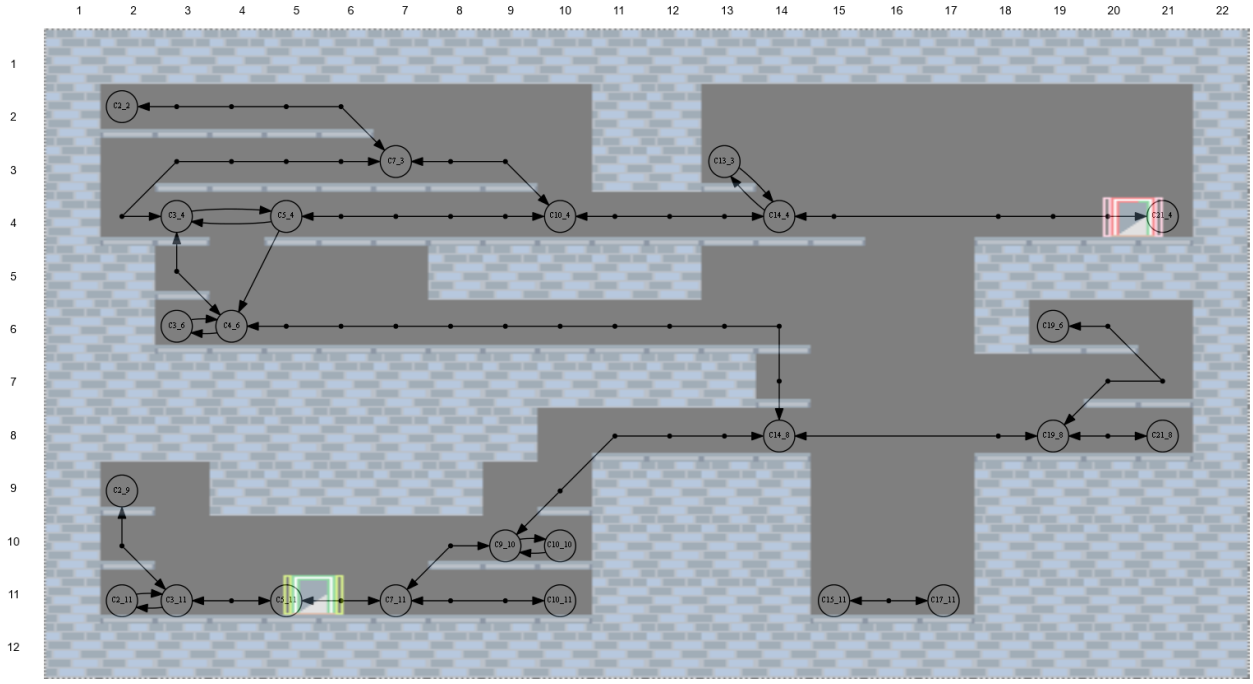

Figure 12. Graph generated from the example level

The system classified the vertices as follows:

- **Mandatory**: C3_4, C4_6, C5_11, C7_11, C9_10, C10_4, C14_4, C14_8, C21_4
- **Optional**: C5_4, C7_3
- **Dead-end**: C2_2, C2_9, C2_11, C3_6, C10_10, C10_11, C13_3, C19_6, C21_8
- **Unreachable/ Vain**: C15_11, C17_11
- **Path to Dead-end**: C3_11, C19_8

In addition, the tree presented in Figure 13 was calculated.

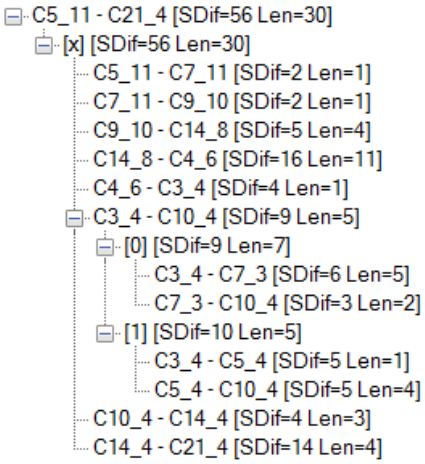

Figure 13. Calculated tree with all possible paths from the beginning to the end of the level. SDif represents the estimated difficulty and Len refers to the segment length in cells.

The system also calculated routes to the existing dead ends. In this case, the system identified the possible routes:

- C4_6 → C3_6
- C5_11 → C3_11 → C2_9
- C5_11 → C3_11 → C2_11
- C7_3 → C2_2
- C7_11 → C10_11
- C9_10 → C10_10
- C14_4 → C13_3
- C14_8 → C19_8 → C21_8
- C14_8 → C19_8 → C19_6

Considering our approach for estimating difficulty, a value of 56 was obtained and solution length was computed as 30 cell movements among segments. As a test, we defined the desired difficulty value of 150 with a 10% error margin, so the algorithm was expected to increase the existing difficulty. In the same manner, we defined the desired length to be 40 movements. One example run presented the result (printed to console) showed on Figure 14. The changes were then applied to the level, generating the content showed in Figure 15.

```
Add spikes @ C7_11 (Difficulty + 10)
Create Bonus Detour. Add potion @ C2_9 ()
Add spikes @ C9_10 (Difficulty + 10)
Create Detour. Add button @ C10_11 and gate @ C8_10 (Length + 3)
Add Enemy @ C12_8 (Difficulty + 15)
Create Detour. Add button @ C10_10 and gate @ C11_8 (Length + 1)
Add enemy @ C7_6 (Difficulty + 15)
Create Detour. Add button @ C19_6 and gate @ C12_6 (Length + 5, Difficulty + 20)
Add enemy @ C13_4 (Difficulty + 15)
Current difficulty: 121.
Current length: 39
Create Detour. Add button @ C2_2 and gate @ C11_4 (Length + 5)
Add spikes @ C19_4 (Difficulty + 10)
Add chopper @ C11_6 (Difficulty + 20)
Current difficulty: 151.
Current length: 44
```

Figure 14. Example of a set of computer modifications

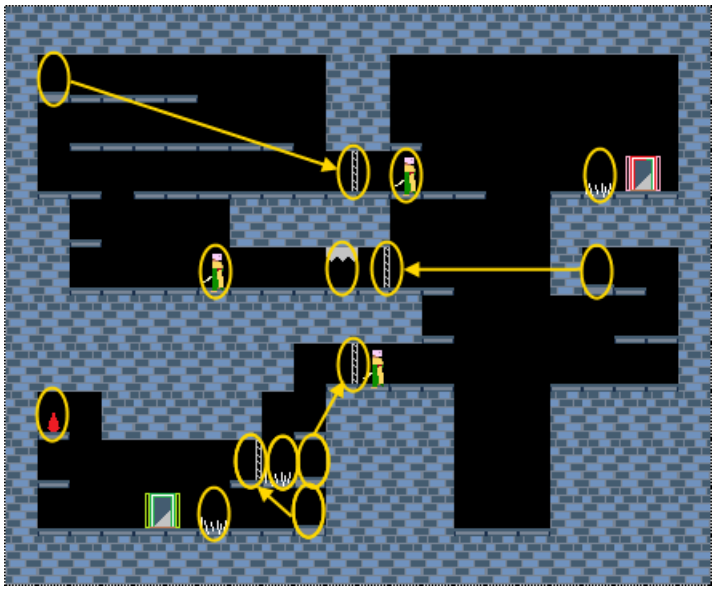

Figure 15. Example of a tuned level for the game Prince of Persia (changes are marked with ellipses).

As previously stated, we have been testing this same approach with our prototype *Tux Likes You* and *XRick*. In the first, the detour principle is unlikely to be applied as the game does not have triggering events or object gathering. However, the primal tests related to difficulty and bonus

content presented good results and the algorithm was able to adapt level segments. Coins are willing to appear as bonus on dead-ends and the number of enemies vary to match desired difficulty. Players have been asked to compete in speed runs (reach the end in the least amount of time) and coin gathering mode (reach the end as fast as possible gathering all coins in the level). On Figure 16 we show a sample of a randomly created level that has been perfected using our algorithm, matching a difficulty value defined by the user. As this game has a less restrictive set of movements, the entity placement is easier and less constrained. However, gap adjustments are hard to express because platforms can have different sizes and configurations.

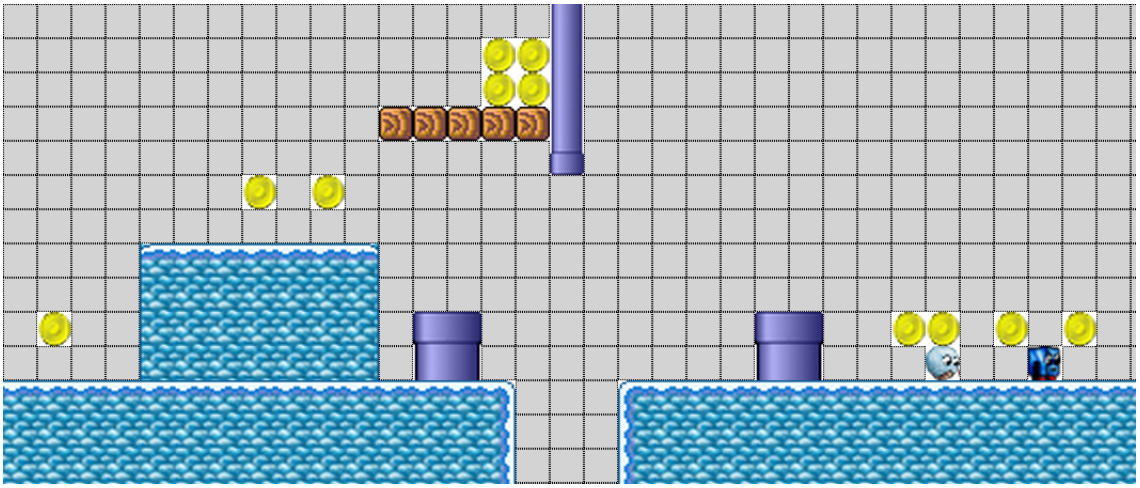

Figure 16. Example of a tuned level for our prototype Tux Likes You. Enemies and coins were added automatically by our algorithm.

Regarding the game *XRick*, primal experiments have also been done with similar results. This is a game with strong emphasis on triggers, where the detour principle is applied. In Figure 17 it is possible to observe a level that was created with that in mind. The main structure was manually created with two obvious dead-ends accessible with the ladders. The system automatically added the two guards on the bottom, the bonus sphinx on the left dead-end and a trigger on the right dead-end, marked with a stick, which removes the spikes on the bottom allowing the character to go through in the path from the left entry to the right exit.

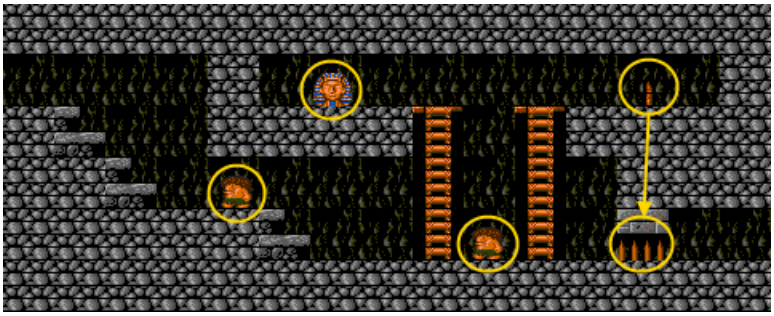

Figure 17. Example of a tuned level for the game XRick (changes are marked with ellipses).

User Tests

The previous examples show that the adaptation algorithm and the approach in general allow generating levels as desired, namely having a tweaking process that is typical in a human design process. Even though those results appear to be valid from an empirical perspective, we implemented an experience to test those same results in practice. For that purpose, a group of users was asked to play random levels in our prototype *Tux Likes You* any time they want, knowing that twenty levels were available, some created by a human designer and some created by a computer algorithm. To promote competition among users, making the experience more genuine and interesting, a table with the highest scores was available, containing the fastest time for each level as well as the highest coin gatherings. After playing a level, users were asked to rate it in a broad concept of quality and to speculate if it was designed by a human or a computer. Both these question were represented in a five point Likert scale. To avoid thoughtless answers, the response was optional. In the end of the experiment, the game was played more than 500 times and we obtained about 300 effective answers, which means approximately 15 answers regarding each level. Table 1 presents the overall results, where the quality and the perception about the generation process are represented as the mean and the standard deviation of the available responses.

One first aspect that is easily noticeable is the high standard deviation in the question regarding the generation process. For our particular case, this high value is positive, as it denotes that the users often mistook computer generated level with humanly designed and the opposite. The obtained responses are mainly guesses rather than concrete affirmations. That can be also stated observing the mean values. There is no noticeable trend and, besides the generic uncertainty that lead mean values to be around 3, stronger opinion answers are confused. For instance, human designed levels 2 and 7 tended to be wrongly perceived as generated by a computer and algorithmically generated levels 4 and 5 were incorrectly perceived in general as being humanly created. Regarding the level quality, users typically considered levels to be good (mean = 4). There is no noticeable difference in human or computer generated levels (overall means of 4.04 against 3.95). However, it is still interesting that the two highest rated levels were generated with our algorithm (levels 2 and 3).

Question / Level	The level was humanly of procedurally created?		Rate the level overall quality.	
	μ	σ	μ	σ
Humanly created 1	2.89	1.66	4.33	0.82
Humanly created 2	4.00	1.41	4.33	0.94
Humanly created 3	3.00	1.41	4.13	0.78
Humanly created 4	2.00	1.22	3.50	0.87
Humanly created 5	3.14	1.36	4.14	0.99
Humanly created 6	3.33	1.70	3.83	0.90
Humanly created 7	2.00	0.93	3.86	0.64
Humanly created 8	2.25	1.48	4.13	0.78

Humanly created 9	3.29	1.58	4.14	0.83
Humanly created 10	3.09	1.08	4.00	0.85
Procedurally generated 1	2.57	1.29	3.71	0.70
Procedurally generated 2	4.00	1.00	4.75	0.43
Procedurally generated 3	2.80	1.60	4.40	0.80
Procedurally generated 4	2.33	1.37	3.50	1.26
Procedurally generated 5	2.00	1.55	4.20	0.75
Procedurally generated 6	3.11	1.37	4.11	0.74
Procedurally generated 7	3.00	2.00	3.50	0.50
Procedurally generated 8	3.55	1.62	3.73	0.96
Procedurally generated 9	3.80	1.25	4.30	0.78
Procedurally generated 10	3.29	1.98	3.29	1.28

Table 1. Overall user responses about level quality and generation perception for each of the tested levels

CONCLUSIONS AND FUTURE WORK

We have presented a technique that, considering a roughly defined platform level and a corresponding graph, finalizes the level adding optional content and adjusting difficulty. As this technique lays strongly in the analysis of the graph that sketches the avatar movement, we detailed our efforts in graph analysis and some of the interesting calculations that can be performed.

Filling a level structure with additional content and adapting paths to force some particular actions produces improvements in content richness to the topic of level generation. This approach brings context to the actions that must be performed in order to capture the user interest and to create less linear gameplay as seen in more contemporary games. Naturally, this type of adjustments are mostly suitable for games that have somehow in its mechanics the principle of gathering certain objects or triggering some events to unlock passages. For this reason we directed several examples to the game *Prince of Persia*, where we could see how our algorithm included new content. However, we have also applied successfully the technique in *Infinite Tux* (and *Infinite Mario Bros.*) to add enemies and coins.

Also, the proposed multiplayer adjustment algorithm allows cooperative play in a shared environment independently of the player skills, promoting gameplay as an interpersonal experience. The considered games and most of classic platform games are only single player, by which this particular concept is still theoretical. Some experiments were done assuming possible versions of *Prince of Persia* and *Infinite Mario Bros.* supporting two players and the output is coherent with those premises. One important aspect to consider in further developments is to apply these concepts effectively in a multiplayer *platformer* and perform tests with users to confirm the initial perceptions.

Regarding the generation prior steps we have presented two alternatives. The first consisted in the adaptation of our previous approach based on evolutionary computing. For this

particular case, the usage of a common representation scheme was particularly useful to make it more generic. Our second alternative shows our primal outcomes in mixing a chunk based approach with graph representations. We are currently gathering more user data in gaming sessions and collecting level samples in order to expand this principle to a more complete automatic chunk extraction.

Our simple user experiment shows that the algorithm can generate levels that can be played as those that have been humanly designed, without any particular differences. Users could not exactly tell the difference among them and there was no particular preference for one specific type. During the sessions, we informally asked some of the users about the reasons behind their answers. A common reasoning to state that a level is designed by a human is the detection a certain elegant meanness. For instance, users stated that only a human could think of hiding coins in certain difficult positions out of the main path, which is something that, in fact, occurs in some of the automatically generated levels of our test set.

As the approach that we have presented is based in a generic level structure and a graph representing the main movements or, more generally, certain transitions, we believe that it is interesting to expand the concepts out of the *platformer* genre. In particular, we have envisioned the usage of the previous ideas in three-dimensional platform videogames to expand the geometry generation and in Role Playing Games (RPG) to define the path structure that lead to quests in a story.

In some of our recent work we have been analysing with more detail the level design patterns that can be found generically in platform videogames, besides those that have been used with the adaptation algorithm, namely the detour creation principle. In particular, we are focusing more contemporary titles to perceive the everlasting concepts. It is intended for further developments to translate those concepts into graph situations in order to improve the adaptation algorithm to reproduce them.

ACKNOWLEDGEMENTS

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.

REFERENCES

Calot, E. (2008). Prince of Persia, specification of file formats. Princed development team.

Compton, K., & Mateas, M. (2006). Procedural Level Design for Platform Games. Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE) (pp. 109–111).

Csikszentmihalyi, M. (1991). Flow: The Psychology of Optimal Experience. (H. Collins, Ed.) (Vol. 54, p. 303). Harper & Row. doi:10.1145/1077246.1077253

Diablo. (1996). Blizzard Entertainment.

Fisher, J. (2012). Cloudberry Kingdom. Pwnee Studios.

Jennings-Teats, M., Smith, G., & Wardrip-Fruin, N. (2010). Polymorph: A model for dynamic level generation. Proceedings of the 2010 Workshop on Procedural Content Generation in Games.

Karakovskiy, S., & Togelius, J. (2012). The Mario AI Benchmark and Competitions. IEEE Transactions on Computational Intelligence and AI in Games, 4(1), 55–67. doi:10.1109/TCIAIG.2012.2188528

Mawhorter, P., & Mateas, M. (2010). Procedural level generation using occupancy-regulated extension. 2010 IEEE Conference on Computational Intelligence and Games, 351–358. doi:10.1109/ITW.2010.5593333

Mechner, J. (1989). Prince of Persia. Brøderbund.

Mechner, J. (2012). Prince of Persia Source Code. Jordan Mechner's Blog. http://jordanmechner.com/blog/2012/04/source/

Miyamoto, S., & Tezuka, T. (1985). Super Mario Bros. Nintendo.

Mourato, F., & Próspero dos Santos, M. (2010). Measuring difficulty in platform videogames. 4ª Conferência Nacional Interacção humano-computador.

Mourato, F., Próspero dos Santos, M., & Birra, F. (2011). Automatic level generation for platform videogames using genetic algorithms. Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology (ACE 2011) (pp. 8:1–8:8). New York, NY, USA: ACM. doi:10.1145/2071423.2071433

Nygren, N., Denzinger, J., Stephenson, B., & Aycock, J. (2011) User-preference-based automated level generation for platform games. IEEE Symposium on Computational Intelligence and Games.

Pedersen, C., Togelius, J., & Yannakakis, G. (2009). Modeling Player Experience in Super Mario Bros. 5th International Conference on Computer Intelligence and Games (CIG '09). IEEE Press.

Persson, M. (2011). Minecraft. Mojang.

Prince of Persia Classic. (2007). Ubisoft.

Rick Dangerous. (1989). Core Design.

Shaker, N., Togelius, J., Yannakakis, G. N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., et al. (2011). The 2010 Mario AI Championship: Level Generation Track. IEEE Transactions on Computational Intelligence and AI in Games, 3(4), 332–347. doi:10.1109/TCIAIG.2011.2166267

Smith, G., Cha, M., & Whitehead, J. (2008). A framework for analysis of 2D platformer levels. Proceedings of the 2008 ACM SIGGRAPH symposium on Video games - Sandbox '08 (p. 75). New York, New York, USA: ACM Press. doi:10.1145/1401843.1401858

Smith, G., Gan, E., Othenin-Girard, A., & Whitehead, J. (2011). PCG-based game design: enabling new play experiences through procedural content generation. Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, 5–8.

Smith, G., & Othenin-Girard, A. (2012). PCG-based game design: creating Endless Web. Foundations of Digital Games 2012 (FDG ’12).

Smith, G., Treanor, M., Whitehead, J., & Mateas, M. (2009). Rhythm-based level generation for 2D platformers. Proceedings of the 4th International Conference on Foundations of Digital Games - FDG ’09, 175. doi:10.1145/1536513.1536548

Smith, G., & Whitehead, J. (2010). Analyzing the expressive range of a level generator. Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames ’10, 1–7. doi:10.1145/1814256.1814260

Smith, G., & Whitehead, J. (2011). Launchpad: A Rhythm-Based Level Generator for 2-D Platformers. Int’l Conference on the Foundations of Digital Games (FDG 2009), 3(1), 1–16.

Smith, G., Whitehead, J., Mateas, M., Treanor, M., March, J., & Cha, M. (2011). Launchpad: A Rhythm-Based Level Generator for 2-D Platformers. IEEE Transactions on Computational Intelligence and AI in Games, 3(1), 1–16.

Sonic - The hedgehog. (1991). Sega.

Speedtree. (2013). Interactive Data Visualization, Inc.

Spelunky. (2012). Microsoft Studios.

Supertux. (2010). http://supertux.lethargik.org/index.html

Togelius, J., Karakovskiy, S., & Baumgarten, R. (2010). The 2009 Mario AI Competition. IEEE Congress on Evolutionary Computation, 1–8. doi:10.1109/CEC.2010.5586133

Toy, M., Wichman, G., Arnold, K., Lane, J. (1980). Rogue.

Viglietta, G. (2012). Gaming is a hard job, but someone has to do it. 6th International Conference on Fun with Algorithms.

XRick. (2001). http://www.bigorno.net/xrick/