

A Collaborative Procedural Content Generator for 2D Platformer Games

By

Bradley McFadden

COSC 4086-EL02 Project

A project submitted in partial fulfilment of
the requirements for the degree of
Bachelor of Computer Science

Bharti School of Engineering and Computer Science
Faculty of Science, Engineering and Architecture
Laurentian University
Sudbury, Ontario, Canada

© Bradley McFadden, 2022

Table of Contents

List of Figures.....	3
1. Introduction.....	4
1.1 Procedural generation for platformers in a selection of games.....	8
1.2 Procedural generation for platformers in research.....	10
1.3 Computers in the creative process.....	15
1.4 Functional and usability requirements.....	18
2. Methodology.....	22
2.1 Improving the level editor.....	22
2.2 A suitable level generator for an interactive system	24
2.3 Adding collaborative elements to level creation.....	28
2.4 Final system.....	30
3. Results.....	32
4. Conclusions & Future Work.....	39
Bibliography.....	42
Appendix A: mORE Level Generation Algorithm	45

List of Figures

Figure 1. Interface of <i>Super Mario Maker</i> 's level editor.....	5
Table 1. Taxonomy of procedural content generation.....	6
Figure 2. Possible chunk types in InfiniteTux.....	8
Figure 3. Two levels generated generated with similar parameters in InfiniteTux.....	9
Figure 4. Level generated in <i>Spelunky</i> . From [9].....	10
Figure 5. A single-cell level from <i>Hollow Knight</i>	11
Figure 6. Interface of <i>Tanagra</i> . From [27].....	13
Figure 7. Example of emergent behaviour in ORE's generated levels.....	15
Figure 8. Interface of the <i>rlbrush</i> editor for Sokoban. From [6].....	16
Figure 9. Interface of <i>Baba is Y'all</i>	18
Figure 10. Comparison between <i>Super Mario Bros.</i> and <i>InfiniteTux</i> components.....	22
Figure 11. Existing level editor for <i>InfiniteTux</i>	23
Table 2. Differences between ORE and mORE.....	25
Figure 12. mORE's default chunk library.....	27
Figure 13. Three levels generated using mORE.....	28
Figure 14. Final system with "Level Overview" panel open.....	31
Figure 15. Final system with "Chunk Library" panel open.....	31
Table 3. Descriptive statistics for each sample.....	35
Table 4. Results of significance testing for each pair of samples.....	35
Figure 16. Distribution of KL-Divergence for pairs of levels (n=10,000).....	35
Figure 17. Comparison between two levels generated by mORE (top) and by Notch (bottom).37	
Figure 18. The 40 most common 2x2 features in a Notch (left) and a mORE (right) level.....	38

1. Introduction

Among all the tasks when designing a game, level design choices are some of the most important [1, 26]. A well-designed level pushes a game's mechanics to its limits, excites the player, and makes a good game great. A great example is the original *Super Mario Bros.* game. Only the first level sees the introduction of new game mechanics, and after new enemies are slowly introduced. However, the number of units sold indicate that players found it to be a worthwhile and rewarding game in spite of the limited number of mechanics[23]. Level designers need to have knowledge across the spectrum of skills applicable in game design [1]. They need an understanding of the game's mechanics, to know how the artificial intelligence behaves, and to incorporate art assets into their levels, to match the vision of the creative lead. Additionally, level designers often have to switch between two considerations that constantly need their attention: "Is this level a good test of the skills that the player has developed at this point in the game?" and "Does this obstacle fit with the overall pacing and rhythm of the level?". Level designers frequently have to put up with resorting to a "modify and test" approach for certain game genres, as a small change can sometimes have a large impact on the playability of a level [27].

In spite of the importance of this task, the software for level designers has only improved marginally over the years. In 2D games, the software is similar to many image manipulation programs. For instance, in *Super Mario Maker's* level editor (which can be seen in Figure 1), the level designer will drag terrain around with resize and translate handles, or drag and drop entities onto the map from a pallet. A similar dilemma occurs with level editors for 3D games, such as *Trenchbroom*. In *Trenchbroom*, the level designer builds geometry as if they were creating a 3D model, then moves around entities as they please. Level editing in both cases is a direct manipulation task, but far too often level editors lack feedback that could help the designers more quickly identify and correct mistakes, to the detriment of the efficiency and pleasure associated with the level editing as a direct manipulation task [22]. It is my opinion that it would be more beneficial to the level designer if immediate feedback was given to them while working on the level before playtesting begins, so they could correct problems as they happen, which would eliminate the backtracking of the task. One example would be to inform the level designer of levels that are not playable before the designer runs playtesting. The need for better level editing tools is made apparent by other software that has lowered the barrier for entry in the

game development scene such as the wide variety of open source game engines, intelligent development environments, visual programming interfaces, and frameworks available to developers.

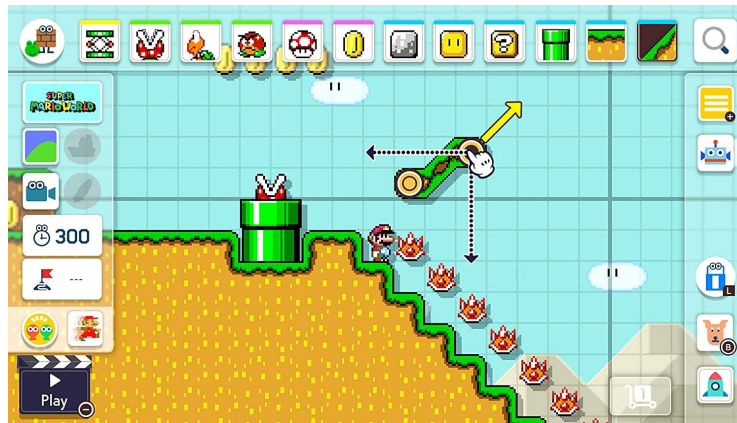


Figure 1. Interface of *Super Mario Maker*'s level editor

One approach that is often implemented for level creation is procedural content generation. Procedural content generation refers to the use of an algorithm to generate many kinds of content [25]. It has been used for terrain in 3D games like *No Man's Sky*, dungeons in games like *The Binding of Isaac*, complete worlds such as in *Endless Web* [29], and even entire games like *Yavalath* [2]. Procedural content generation is a very versatile tool. One of the motivations of this field is to create content much more quickly than a human can, and to generate many more ideas that a human game designer may not have been able to come up with by themselves. Procedural generation has been used in the past due to memory constraints - it is more space efficient to store a single integer as a seed, and to generate a world from runtime using it [24p4]. *Elite* used procedurally generated worlds for this reason. Roguelike games such as *Hades*, and *Risk of Rain 2* most commonly use procedural generation. The wide variety of encounters to explore due to the stochastic nature of procedural content generation can be very engaging, and in the case of games like *Spelunky*, it can take hundreds of games to explore the entire procedural space [11]. In commercial platforming games, it is not very common to see procedural generation used [4], in spite of the increase in the amount of content, replayability value, and development time saved by its use.

Shaker et al. describe a number of desirable properties of content generators [25p6-7]. These properties include speed, controllability, reliability, expressivity and diversity, and creativity. Many of these properties depend on the application. As an example, a level generator that creates platform games at runtime needs to be fast. It needs to be reliable so that there's a guaranteed route through a level. Additionally, it should be creative, so that the levels it generates remain interesting over time.

Procedural content generation has been previously split into a taxonomy by Togelius et al. [31], and this taxonomy is useful for describing content generators. This taxonomy is summarised in Table 1. Level editing tools are one example of offline content, and this category also includes games that ship with level editors. Adaptive content generation is present in *Left 4 Dead*. The number of difficulty of enemy encounters scale according to how well the player team is doing, so the content generator adapts to different groups of players. Content generators that use evolutionary algorithms or grammars make use of a generate and test approach, while chunk-based generation approaches are normally constructive.

Table 1. Taxonomy of procedural content generation

Property of generator	Possible values of property	
Moment of content creation	Online - Content created during playtime.	Offline - Content is either created separately, or during development.
Type of content	Necessary - The produced content is required to play the game. Includes quests, terrain, and more.	Optional - The game is playable without the generated content.
Dimension of control	Random - The generator is not controllable by a user to any extent.	Parameterized - The user may set parameters which affect the generator's output.
Applicability	Generic - The generator's output can be used in many contexts, or is applied to many different conditions.	Adaptive - The content of the generated adapts to different conditions.
Reproducibility	Stochastic - Knowing the input conditions of the generator is not sufficient for	Deterministic - Knowing the input conditions of the generator is sufficient to

Property of generator	Possible values of property	
	predicting the output.	predict its output.
Level of authorship	Mixed - The generator may require or optionally use input from a user.	Automatic - The generator is capable of producing content autonomously.
Production approach	Generate and test - The generator constructs a large number of potential solutions, and searches for an optimal selection.	Constructive - The generator sequentially constructs an acceptable solution.

Platformer games are often the subject of papers on level generation for video games. A platformer is a set of physics-based dexterity challenges that are viewed from the side [23]. The player navigates their avatar past obstacles, collects currency, defeats enemies, and proceeds to the end of the level. Platformer games often have many short levels that the player must complete before finishing the game. These levels are often time restricted, may or may not have checkpoints, and often implement a system where the player can only make a finite number of mistakes before having to restart from the beginning, as is the case for both *Super Mario Bros.* and *Crash Bandicoot*. In some game genres, it is a challenge to evaluate the difficulty of a task at a glance, but a player familiar with a platformer game can quickly gauge the challenge presented by a task, as the hazards need to be visible in order for the player to react to them adequately [30]. As an example, given a level in *Super Mario Bros.*, a player knows that falling down a pit will instantly kill them, and can reason that larger pits make for a greater challenge. Computers do not have the same level of intuition that experienced human players have about the task, and it can be difficult to model a player's behaviour to generate appropriate levels for them [23], which makes procedural content generation for platformers such an interesting undertaking. Furthermore, the rules and actions present in platformers are often simple (for instance, the primary actions in *Super Mario Bros.* are running and jumping), but creative level geometry and enemy placement can lead to a large variety of enjoyable content, despite the limited actions [5].

1.1 Procedural generation for platformers in a selection of games

There have been several interesting implementations of procedural content generation. To start, two notable openly available platformers that incorporate content generators are *Infinite Mario Bros.* and *Spelunky*.

Infinite Mario Bros. has been frequently used in the past by the research community [23]. It is a public domain clone of the original *Super Mario Bros.* game. The assets are still proprietary, so this report will use *InfiniteTux*, which is a version of *Infinite Mario Bros.* using open-source assets. Levels are generated automatically, and they slowly scale in difficulty by increasing the frequency of more difficult terrain and adding more challenging enemies. Terrain generation proceeds in several passes. First, the base of the level is generated from left to right by placing human-authored platformer patterns in a random fashion. Each chunk has a small amount of variety to it as well, but two chunks of the same category generally are very similar. Some chunks are weighted higher than others depending on the difficulty parameter.

In Figure 2, the various chunks that *InfiniteTux*'s generator uses are given. From left to right, there are straight, hill straight, jump, tubes, and cannons. Straight is the most common chunk. Jumps, tubes, and cannons are considered by the generator to be more difficult terrain, so these are added more often when it is desirable to increase the game's difficulty.

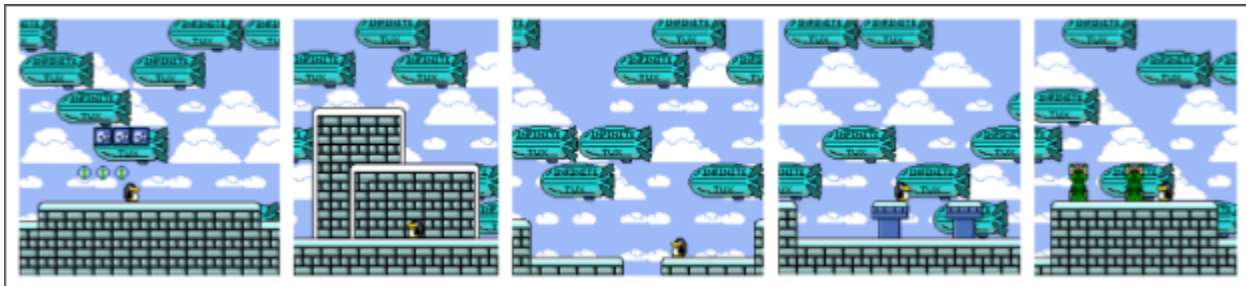


Figure 2. Possible chunk types in *InfiniteTux*

A level is constructed by connecting enough chunks to fill the desired level width. The chunks are chosen in a random fashion, with weights making some more common than others. After placing these initial chunks, a second pass over the level is performed. For every straight and hill straight chunk, a line of enemies can be added. It is also possible that coins, and a series of reward blocks are placed, as is the case in the leftmost image of Figure 1. The placed enemies depend on the difficulty of the level. More difficult enemies hurt Mario when he jumps on them,

avoid falling off ledges, or have wings, providing them a jump and an extra point of health. In Figure 3, two levels generated with the same difficulty are shown. Notice that patterns repeat frequently, like having a ceiling above several coins, the use of hills, and overall the flat appearance of the two worlds. There appears to be a low amount of variety in the procedural space of this generator. On a positive note, the generator is very reliable. Adjacent chunks are never generated more than a few blocks above or below the previous one, and ceilings cannot extend to the last tile of a chunk, so each level is guaranteed to be winnable.

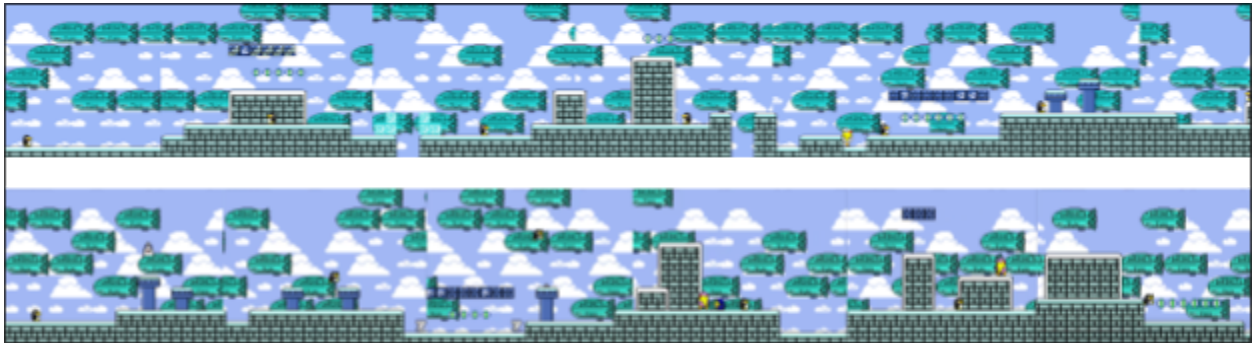


Figure 3. Two levels generated generated with similar parameters in *InfiniteTux*

Spelunky is a very different game than InfiniteTux, but the two generators share many similarities. The gameplay of Spelunky encourages exploration, and allows more ways for a player to traverse, including destroying large amounts of terrain with bombs, and using ropes to ascend high walls. As a result, it is easy to guarantee that a Spelunky level is winnable. Spelunky always starts the player at the top of the screen, and the exit is at the bottom, so rooms are chosen in such a way that there is a guaranteed path downward toward the exit. Levels in Spelunky are composed of 16 equally sized chunks composed in a 4x4 grid [25p49-51]. The chunks contain hooks that subsequent passes of the generator use to place enemies, traps, or rewards. Due to playability constraints, it is possible for a player to recall every level configuration in approximately 500 playthroughs of the game [10]. An example annotated Spelunky level is provided in Figure 4, which has each chunk numbered by its type, and a red line indicating the path to the end of the level.

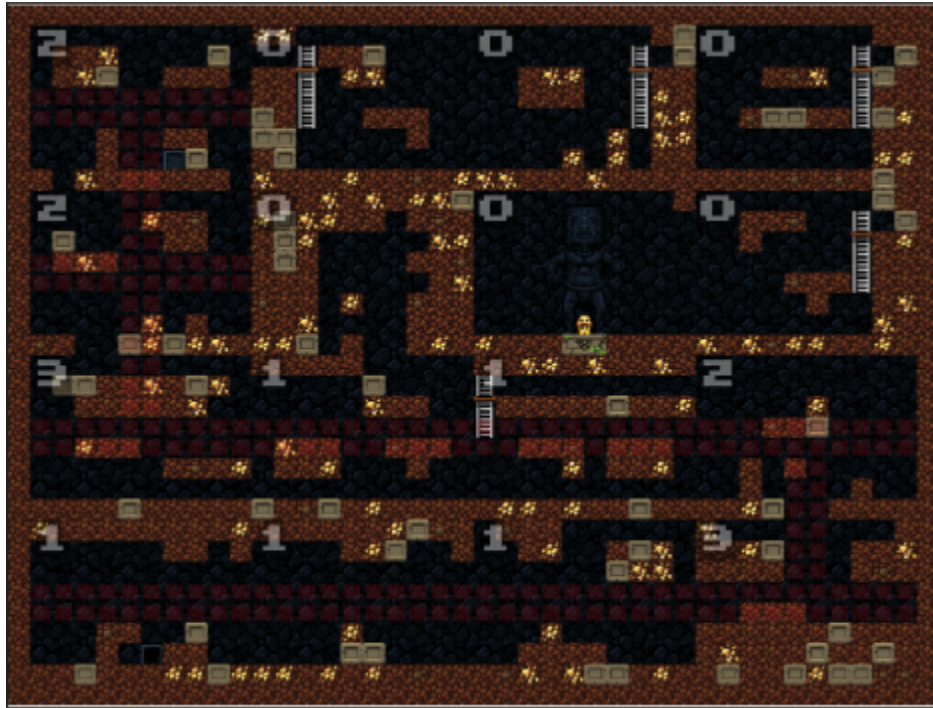


Figure 4. Level generated in *Spelunky*. From [9]

1.2 Procedural generation for platformers in research

Compton & Mateas produced one of the conceptual models of platformer games [4]. In turn, their model is inspired by a model from music theory for representing complex patterns in African and African American music. The model separates platformer levels into atomic components, such as platforms, hills, and vines. In other research, components are referred to as “beats” [5, 26]. Beats are grouped into patterns, chunks, or rhythm groups [5], which are sequences of gameplay terminated by a break in rhythm. Rhythm groups can be combined into a unit called a cell, which is one or more separate patterns placed together in a linear sequence. Cells can be connected in a variety of ways, which makes a level non-linear. The paper also defines a level design algorithm based around this model. The level design algorithm proceeds by creating a context-free grammar to represent cells, and rhythm groups. Each cell is connected with some possible pattern like a branch, or a parallel pattern. Within each cell, zero or more rhythm groups are placed, each with its own series of beats. A grammar that implements this algorithm is later created by Shaker et al. [24]. Later, Smith et al. published research that extended the rhythm group model to be more complete [26]. Their research clarifies the meaning of rhythm group to a more precise meaning. A rhythm group ends when there is a change in the

rhythm of a level, such as if the player has reached a safe zone from a previously unsafe zone. In Figure 5, a cell of *Hollow Knight* is given with its rhythm groups represented by non-overlapping green regions.

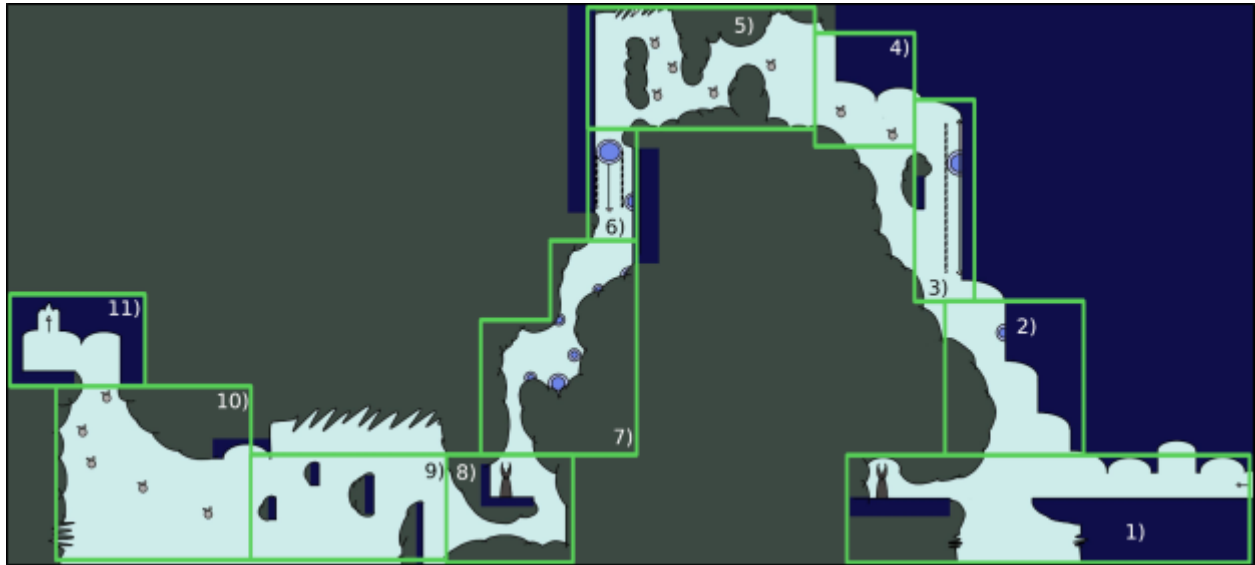


Figure 5. A single-cell level from *Hollow Knight*

The individual rhythm groups of the cell in Figure 5 are described in detail:

1. The player starts on the right hand side of the screen. The rhythm group extends out until the first statue.
2. The player must wall jump alongside the ceiling for right angles. The rhythm group ends after the saw blade, as they can stall here by nail-jumping on the saw for a break.
3. The player reaches this rhythm group after wall jumping and dashing to it. They must avoid the moving saw blade along the wall. The rhythm group ends at the peak of the wall, as they can wall jump until ready for the next part.
4. This rhythm group contains two flies that the player must bounce on to traverse it. It ends after they double jump off the second fly to reach the wall
5. This rhythm group extends from the wall in the top right, to the area above the moving saw blades. The player must use their dash and bounce off the flies, until reaching the left wall.
6. Here, the player must slip past the moving saw without hitting the static saw. Then, they can rest by jumping against the wall.

7. In this rhythm group, the player must bounce off the saw blades and slip through the narrow passage until reaching the statue.
8. This rhythm group contains the statue and the long dash wall.
9. This rhythm group contains the 4 vertical walls. The player must climb each wall and dash to the next one.
10. This rhythm group contains 5 flies that the player must bounce on, and dash between.
11. The last rhythm group is the final platform containing the cell exit.

Smith would later create a level generator based around the extended rhythm framework [28]. The *Launchpad* generator outlined in the work uses two grammars, one for generating beats, in the form of actions that the player performs. Beats have a start time and a duration. The next part of the generation phase is geometry generation that creates a level from the generated beats. Since generation by grammars is fast compared to other methods, a large number of levels are created, and two critics that evaluate the levels find an optimal level that is playable and not flat. Important conclusions from this work include the call for quantitative evaluations of analysing levels, instead of qualitative measures, as well as the use of difficulty analysis in a generator. An extension to this work is *Polymorph* [9], which is a game that uses the Launchpad generated for dynamic difficulty adjustment. A limitation of Polymorph that is cited by the authors is that difficulty is measured per beat, and does not consider how adjacent beats might interact with each other to become more challenging. As an example, jumping from a moving platform onto a platform containing a moving enemy is more complex than landing on an empty platform, or avoiding the enemy without having to dismount the platform.

Another interesting application of Smith et al. was *Tanagra*, a “mixed-initiative design tool...in which a human and computer work together to produce a level” [27]. Tanagra is pictured in Figure 6. Tanagra allows a human designer to work with a procedural generator, while validating level playability, generating alternative designs, and allowing regeneration of specific portions. To describe it in terms of the procedural generation framework [31], Tanagra is offline, constructive, and uses mixed authorship. A notable feature of Tanagra is the beat editor. Where most level editors operate on a level of placing individual tiles, Tanagra allows its users to modify its generated beats as a whole. Level generation in Tanagra proceeds by satisfying the following requirements: 1) Generate a level. 2) Respond to designer input by placing and moving existing geometry. 3) Respond to the designer modifying the beats of the level. 4) Ensure all

levels are playable. The authors call for additional extensions to *Tanagra*, including some form of automatic difficulty analysis, the ability to view level paths, and support for more types of rhythm groups.

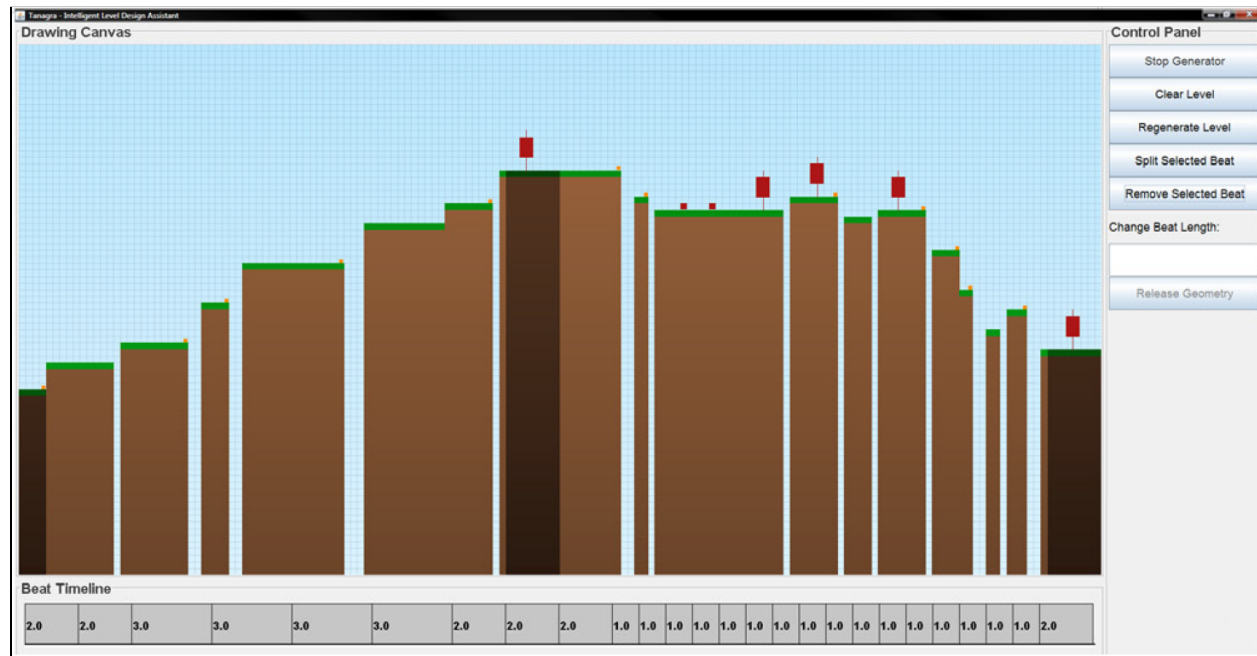


Figure 6. Interface of *Tanagra*. From [27]

In 2010, the Mario AI Championship was held [23]. The competition had several tracks, including participants competing to create agents that could play procedurally generated levels well, and participants competing to create level generators that created the most enjoyable experience, as determined by pairwise comparisons using 15 judges. Six teams participated in the competition, each contributed to a report to document their methodology.

The winner was Ben Weber, who created the “Probabilistic Multipass Generator”. His approach involved five passes from left to right over a level, which created the ground, hills, pipes, enemies, blocks, and then coins. Each pass, some event is chosen from a uniform distribution. A constraint exists that limits events so that the level remains playable.

Players were asked to play levels before testing the generators, and gameplay metrics were available to competitors to incorporate into their algorithms. Shimizu and Hashiyama used this player data to estimate a player’s skill level, and to describe how likely they were to break blocks, collect coins and kill enemies. Sections of human-authored content were then strung

together according to the player's playstyle. Shimizu and Hashiyama cite that the benefits of this approach were a better correspondence to a player's playstyle and skill, and no complex fitness function. The authors state that the primary weakness is that the expressivity of the levels is limited by the chunks available.

The next approach was by Sorenson and Pasquier, and is detailed in [30]. Their method uses an evolutionary algorithm to generate rhythm groups, which uses a fitness function that estimates the difficulty of a rhythm group according to the margin of error associated with jumps. At the same time, the fitness function is biased to select levels where the challenge alternates between successive rhythm groups. They assessed their model on how successful it was at generating levels with a similar difficulty curve to the first four levels of Super Mario Bros, under the assumption that these first four levels are examples of well-designed Mario levels. Their generation took several hours, using an initial population of 200 individual levels evolving for 10,000 generations. For any interactive software, this amount of delay is unacceptable, so the approach is more suited to use in an offline generator.

Another interesting level generation approach was Peter Mawhorter's entry, which is expanded upon in [16]. The authors describe their approach as occupancy-regulated extension (ORE). They use a library of chunks, which are annotated with the frequency in which they'll be placed, and have additional metadata for the generator in the form of "anchors". Anchors represent possible places that the player might occupy in a chunk, and the generator tries to preserve these places. Three main steps occur in the generation process. Context selection chooses an existing anchor point in the level to expand starting with the initial two anchors that mark the beginning and end of a level. Next, chunk selection occurs, which determines which chunk from the library should be placed into the context. The last step is to integrate the chunk into the existing level. Post processing steps are also applied to generated levels to ensure that constraints are respected, such as power-ups not being too close together. The ORE generator occasionally produces some emergent or clever level design. In Figure 7, a strange powerup block was placed that only allows small Mario to reach it, or as normal Mario with a crouch-jump. The authors state that the strength of their algorithm is game independence, since the generator only cares about anchor points, and the post-processing controls game-dependent constraints. They also state that tagging the chunk library with keywords that provide more information about each chunk in the library may allow difficulty to be incorporated into

generation. The authors mention that ORE could easily be adapted into a collaborative or mixed-initiative system, as the generation could be interrupted as long as anchor points are preserved. Some of the weaknesses cited include the large chunk library needed for variety in the level; the authors used 42 chunks, which were all human-authored. Additionally, there is no strict guarantee that the generated levels are playable.



Figure 7. Example of emergent behaviour in ORE's generated levels

1.3 Computers in the creative process

Computers have supported people in the creative process for some time. Software that helps users in design tasks is commonly referred to as computer-aided design software, or CAD. The motivating ideas behind CAD are to provide users with a way of visualising their designs on the computer, and to have the computer aid in applying constraints to the work [12]. Lawson & Loke argue that computers could do more to facilitate their user's creativity in the design process, and provide three ways in which CAD systems could contribute: 'man/machine roles in creative thought', 'parallel lines of thought', and 'mind sharing'.

In order to play a greater role in creative thought for a task, CAD systems can provide the users with suggestions or new ideas that could enhance the quality of the design. In the context

of a level editor, the level editor could provide the user with suggestions for enemy placement, or recommend that a powerup be placed. An example of such a system is *rlbrush* [6], which is a level editor for *Sokoban*, a tile-based game where the goal is to push boxes into holes. Its interface provides the user with several suggestions that can optionally be implemented, but not with any rationale about why its suggestions were given. The system is shown in Figure 8, and several different suggestions are visible above the working canvas.

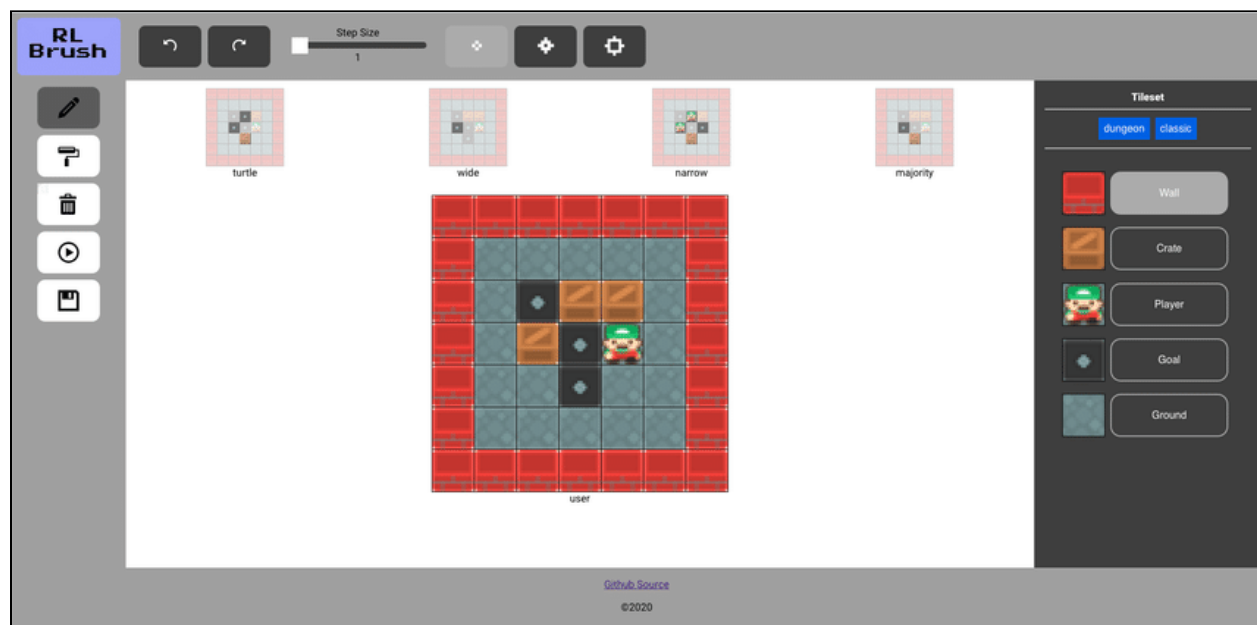


Figure 8. Interface of the *rlbrush* editor for *Sokoban*. From [6]

Parallel lines of thought refers to a system playing more than one role in the design process at a time. Several roles are outlined, including critic, collaborator, and informer. Existing level editors have had the software operate as a collaborator before [7, 13, 27]. Of these, *Sentient Sketchbook* operates as both a critic and collaborator. It enforces playability and informs the user of the fairness of the current map, while also generating many alternatives as iterations on the user’s design. The idea of CAD software playing different roles was also discussed by Lubart [14]. Lubart outlines four different roles: nanny, where the system guides the user, colleague, where the system works with the user, pen-pal, where the system helps the user to communicate their ideas to others, and coach, where the computer acts as an expert system that helps to teach the user.

The last point is mind sharing, which refers to the notion that two minds working together can be more creative than a single mind. This relates to CAD systems that learn from their mistakes and successes, and learn the habits of their user to create better suggestions. The *Morai Maker* system described in [7] used three different types of agents that aimed to learn their user's designs, and use this information to automatically add to the user's levels. According to the study, the participants saw that the system could bring value to their levels, and it would take the form of a collaborator, nanny, or manager. The collaborator and nanny roles were similar to what had been described in Lubart's work. The manager role arose from participants feeling that the system was giving them instructions, or evaluating their performance.

When the user and system co-operate on a task, the system can be described as a "mixed-initiative system", as both the system and user work together towards a goal [25p191-200]. Some systems placed more emphasis on the human role, and some more emphasis on the computer role. A procedural content generator for levels that accepts some parameters at the start of the process technically has the user take some initiative, but a majority of the work is done by the content generation algorithm. Similarly, in the field of computer-aided design, tools like *Blender* allow a user to rapidly create 3D models, and the computer can automate some work, but the effort is mostly on the user's end. Mixed initiative systems can also be characterised by how they handle three types of initiative: task initiative, speaker initiative, and outcome initiative. Speaker initiative refers to the mechanism for determining when the user takes a turn on the task, versus when the system takes its turn. The speaker initiative of the ORE level generator [16] would be a form of turn taking. The level generator does its work, then the user decides whether or not to regenerate any sections. Outcome initiative refers to the process of deciding how the work of the problem solution should be divided.

Baba is Y'all is an example of a mixed-initiative system where the user and system interact in a more equal manner [3]. An image of the interface is provided in Figure 9. The system is able to make suggestions based on the history of all levels that have been created by every user. It provides suggestions for rules to implement into the game that haven't been thoroughly explored. It is also capable of helping players find levels with similar mechanics to struggling players. *Baba Is Y'all* demonstrates speaker initiative by allowing the user to work on a level, and providing suggestions. These suggestions happen asynchronously while the user edits, so the user would have to end their turn for the system to incorporate its suggestions. In

regards to the outcome initiative of *Baba is Y'all*, the user decides whether or not to incorporate the suggestions of the system, so the outcome initiative is managed entirely by the user.

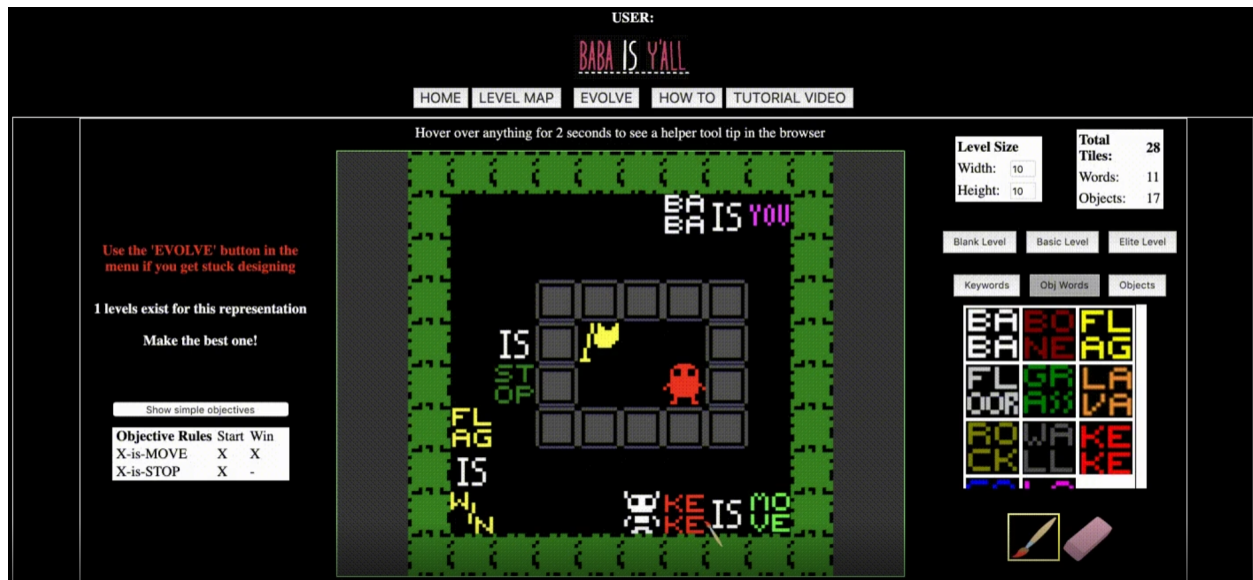


Figure 9. Interface of *Baba is Y'all*

1.4 Functional and usability requirements

For any interactive object or piece of software, it is important to consider the effect that the interaction has on the user, and to design for a positive experience. A well-designed interactive system is easy to understand, provides helpful feedback to its users about the task, maps its controls in a sensible way, and clearly signifies how its controls can be used to create the desired effect [19]. Level editors as software fall under the umbrella of interactive software, and as such are sensitive to these requirements.

A system's usefulness can be divided into two areas, the utility it provides with its functions, and its usability. Usability generally refers to the system's front end, while utility refers to its back end. Many systems have utility, but less have usability. Nielsen first proposed that a system's usability can be broken down into five areas which include learnability, memorability, efficiency, satisfaction, and how successful the system is at dealing with errors [17p22]. Learnability refers to the challenge associated with the first time a user uses a system. Memorability refers to how easy it is to remember how to use a system. An efficient system allows the user to perform their work quickly, and is careful not to waste the user's time. A

satisfying system feels pleasant to use, perhaps by having a pleasant visual design, or by the user feeling that the system has helped them with their task. A system that effectively manages its errors, provides helpful error messages when appropriate, and tries to limit the amount of errors that can be made by the user.

A system's usability can be evaluated by how well it matches the ten usability heuristics proposed by Nielsen in [18]. These heuristics attempt to measure the usability principles outlined above, and include:

- Visibility of system status: The system should keep the user informed about its state, and provide feedback within a reasonable amount of time.
- Match between system and real world: The system should use concepts and terminology that the user would know. Information should appear in a natural and logical manner.
- User control and freedom: Users often make mistakes, so the system should always allow the user to cancel actions, undo or redo, and be able to navigate backward.
- Consistency and standards: The system should use standard language for common actions, and the same words should be used for the same actions.
- Error prevention: A usable system constrains the number of errors a user can make.
- Use of recognition rather than recall: The system should make relevant information visible to reduce the user's memory load. Where possible, instructions should be available.
- Flexibility and efficiency: Shortcuts should be provided for advanced users to save time.
- Aesthetic and minimalist design: The system should only include features that add to the usability. Information that is not needed for the task should not be shown, as it can distract from relevant information.
- Error recognition and recovery: Error messages should be clear and precise, and suggest a way to recover or fix the error.
- Help and documentation: Extra help should be easy to access, clear, and helpful.

These usability heuristics provide a set of best practices to follow when designing a user interface. Usability is important in designing a level editor, because one of the main goals is to aid designers in the process of level design. A system with a lot of utility that results in desirable outcomes may be discarded by users if they have a poor experience interacting with it [19p10].

The method of interaction differs among software. In most level editors, designers interact with a representation of the level, and iterate on their design by adding elements one at a time. Often, these elements are accessed from a palette, as in popular level editors like *Halo Reach*'s forge mode, or Super Mario Maker. The designer often sees these elements as they appear in game, and can manipulate their position and properties through the interface by using various tools. This type of interaction is called direct manipulation. Direct manipulation was first described by Schneiderman in . Schneiderman observed office workers using single line editors versus using display editors, where many lines could be seen at once, and feedback on changes was immediate. Direct manipulation tasks require several properties to be useful. A continuous representation of the object needs to be available. Actions should be performed via button presses or mouse movement instead of by text commands. Operations should be rapid, incremental, and reversible. The combination of these properties leads to a situation where several desirable outcomes can occur: experts can carry out a wide range of tasks rapidly, users feel in control over the system, users can immediately see if their actions are helping them reach their goals, and in the case that they aren't they can undo their actions and try another approach. Additionally, users feel comfortable experimenting with the system because the cost of making a mistake is very low. Most level editors meet the criteria for qualifying as a direct manipulation task, but it can be hard to convey information about the experience that the level creates for the user unless the designer plays it, so the cost of making a mistake is higher in terms of time lost, which may lead to less experimentation on the part of the designer due to time constraints.

In a chapter about designing tools for game design, Rouse states,

“In order to create superior content, the design team will need to be equipped with well-designed, robust game creation tools. Therefore, one can conclude that designing a good game is about designing good game creation tools.” [21p392]

In the rest of the chapter, Rouse outlines a number of requirements for any level-editing tool. The first requirement is similar to Nielsen's usability principle of visibility. The designer should be able to see the world being created while in the process of creating it. Level editors should also provide the designer with extra information about the level, not normally visible in game. The level editor in *Halo: Reach* allows the designer to see spawning information that they have placed. The Tanagra content generator provides information about the pacing of the level to the user in the form of a beat timeline, which visually represents the rhythm of the level [28]. A

more dense beat timeline indicates a faster-paced level, since each beat is a player action like waiting or jumping. Another helpful addition to level editors is the ability to immediately playtest the in-progress level. It is well-known that testing is a critical part of any software development, and this is particularly true for games, as gameplay is usually considered the most important feature of a game. A final requirement is that a level editor should allow the designer full control over all gameplay critical sections of a level. Without full control over gameplay-critical sections, a level designer may become frustrated by seemingly arbitrary limitations that get in the way of their work.

The goal of this project is to investigate how to design a system that can assist level designers in creating better levels. As the general task of “creating better levels in video games” is fairly broad, the scope of this project will be limited to the platformer game genre. A significant amount of work of other researchers has been spent on analysing and developing level generators for platformers, and given the 12-week period available to work on this, this genre makes the most sense to develop for. First, procedural generation methods will be looked at in-depth. Then the use of computers in the creative process will be covered. It is also important to cover the requirements of interactive systems in general, and then narrow in on these requirements with respect to level editors. From these requirements, and the research in the previous sections, a system will be proposed and implemented. The system will be a level editor for *InfiniteTux* that aids the level designer in the design process, with the goal of using procedural content generation for this purpose. This game was chosen to be used for several reasons. As previously stated, *InfiniteTux* shares its mechanics with *Super Mario Bros.*, which has seen extensive previous attention in the level generation field, largely due to the series’ popularity and events such as The Mario AI Championship. Another reason to choose to improve *InfiniteTux* is that the code is openly available under an iteration of the GPL licence. *InfiniteTux* is implemented in Java, which is a language that is familiar to many programmers. Lastly, another consideration that made *InfiniteTux* an attractive project to extend was that it features an unfinished level editor, and an interface for generating levels for the game. Specific functional requirements will be outlined later. After the implementation of this system, it will be evaluated by how well it meets these functional requirements, as well as the amount of variation present between its generated levels.

2. Methodology

2.1 Improving the level editor

To start, it seems reasonable to provide an overview of the unfinished level editor for InfiniteTux, as well as some common elements from the game. It is a fair assumption that most people that have played video games are familiar with common enemies in Super Mario Bros., but InfiniteTux uses open-source assets, so a table of reference has been provided in Figure 10 that maps common components from Super Mario Bros. to their representations in InfiniteTux.

























Component Name	SMB Sprite	InfiniteTux Sprite	Component Name	SMB Sprite	InfiniteTux Sprite
green koopa			shell		
red koopa			fire flower		
goomba			mushroom		
spiky			small Mario		
pirahna plant			large Mario		
bullet bill			fire Mario		
bullet blaster					

Figure 10. Comparison between *Super Mario Bros.* and *InfiniteTux* components

The existing interface for the editor is shown in Figure 11. In order to use this level editor, the user works directly on the blue canvas to place tiles one at a time. Tiles are selected from the “Tile Picker” panel on the bottom row. The currently selected tile is shown with a white border in the “Tile Picker” component. The canvas is similar to a paint application, as the user can draw strokes of tiles to speed up the process. Beside the “Tile Picker” is a list of checkboxes that indicate the behaviour of the selected tile. These checkboxes do not currently work, but would show the user which tiles contain a powerup, have collision, or are breakable by the player, all of which is critical information for the level designer. The last part of the level editor is the top row, which shows the level name, cursor position in tile coordinates, and has controls for file management.

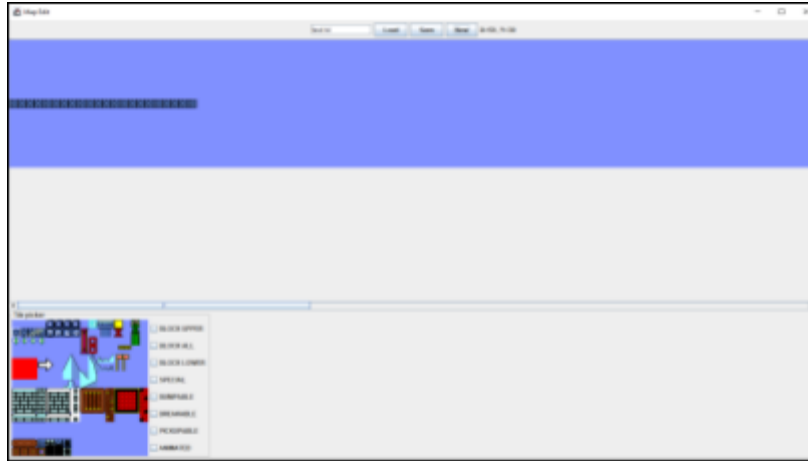


Figure 11. Existing level editor for *InfiniteTux*

Before starting the implementation of the level editor, it is reasonable to create a list of desired features based on the usability principles and level editor guidelines that have been previously mentioned. For each of Rouse’s guidelines about level editors, the functionality that the level editor should support in order to meet the guidelines will be listed.

Guideline 1: The designer should be able to see the world being created while in the process of creating it.

This means that the designer of the level should be able to see the relevant components of the level during its creation. Currently, the tiles of the level are displayed to the user, and the display updates each time the level changes. However, the representations of enemies are missing from this display.

Guideline 2: Level editors should also provide the designer with extra information about the level, not normally visible in game.

The current level editor lacks this functionality. If working, the “Tile Picker” would show the behaviours of the current tile. However, the behaviours of placed tiles are not shown to the designer at all. Additionally, other important characteristics of the level are not shown, including where Mario initially spawns, as well as the location of the level exit.

Guideline 3: Level editors should provide the ability to immediately playtest the in-progress level.

The current level editor lacks any in-place level editing capabilities. This could be added by connecting the produced level to the main game. To automate the playtesting process to some degree, an artificial agent could run through the level at each stage where a change happens to check it for playability.

Guideline 4: A level editor should allow the designer full control over all gameplay critical sections of a level.

The current level editor does not allow the designer to place enemies, change Mario's start position, easily see the behaviours of placed tiles, or even adjust the length of levels.

By comparing the current level editor to Rouse's guidelines, several additions have been identified: 1) Play testing should be added. 2) The designer should be able to view tile behaviours. 3) The designer should be able to move the level exit. 4) The designer should be able to place enemies in the level. 5) The resize operation should be supported, so the user can rapidly adjust the length of the level. Additionally, a few further features should be added to improve the direct manipulation task when using the level editor: 6) Undo and redo for most actions. Adding these components to the system will help improve the overall experience of creating levels.

2.2 A suitable level generator for an interactive system

Several requirements need to be met when building a collaborative level editor. These requirements include performance, creating levels that are sufficiently different from one another, and some features that allow for mixed-authorship of levels. Many of the previously discussed level generators are not suitable for an interactive environment, due to the level generation taking a large amount of time. Most levels that would be generated by evolutionary algorithms fall into this category. Since the set of all possible levels is so large, finding a satisfactory level through evolving levels takes too long for an interactive system. One of the goals of this project is to provide a means for generating ideas and constructing levels more quickly than by hand. Subsequently, a requirement of the level generator is that it should produce more interesting ideas than pasting together linear segments in such a way that InfiniteTux's current level generator (the Notch generator) does [25]. To better facilitate mixed-authorship level editing, the level generator should also be able to re-generate areas that the user has selected, so that the user and system can work together toward generating the final level.

The Tanagra level generator creates a rhythm for the level, and the user can modify either the rhythm or the level geometry to adjust the level to their liking [27]. Additionally, level segments can be re-generated if the user commands the system. One drawback to the generation approach taken by Tanagra is that it often results in linear levels where a speedrun playstyle is preferred. There is little to no emphasis on exploration. However, Tanagra is able to generate levels extremely quickly, due to its use of reactive grammars and working memory.

Alternatively, occupancy regulated extension (ORE) is able to produce levels that end up being less linear than most generators (Mawhorter & Mateas 2010, Shaker et al. 2011), and therefore can reward the player for exploration. However, this feature is dependent on a set of chunks referred to as the “chunk library”, which has to be developed before ORE can produce any meaningful output. Another drawback is that ORE generation takes longer than Tanagra. However, ORE does still allow for a collaborator to work on the level concurrently.

Given these considerations, ORE seems like it would be a better fit than Tanagra for the level generation algorithm for this system. The main advantage of ORE is that it can overlap partial level segments, and supports re-generation. Additionally, a feature that seemed appealing for a mixed-initiative system was to allow the users of the system to modify the chunk library that ORE considers. By adding or removing chunks to the library, the user can gradually tune the created levels to their needs.

In the system, a modified version of ORE was used (mORE). The reason for this is that the original ORE generator’s source is not readily available, as there was some ambiguity in the published algorithm, and the chunk library used in the paper was also not made publicly available. However, the approach is still largely based on the ideas of ORE, and pre-authored chunks are combined in an overlapping manner by anchor points. A few key differences are worth mentioning, and are detailed in Table 2. For reference, the mORE algorithm is described as pseudocode in Appendix A.

Table 2. Differences between ORE and mORE

	ORE	mORE
Chunk representation	Text file; with its own encoding, allowing some extra information like whether a solid tile is part of the ground or a platform.	Text file using the same encoding as InfiniteTux, with the exception of anchors and preserved spaces.

Clipping	Parts of the test chunk that are out of bounds of the level are discarded.	If any part of the test chunk is out of the level, the whole chunk is discarded.
Number of passes	One pass, but extended artificially by the generated adding additional anchor points when it runs out.	Two passes, with two separate sets of chunks working with the same anchors.
Chunk tags	Uses tags for frequency that a particular chunk should be selected.	Uses tags to separate chunk library into first and last sets.
Chunk library	Uses two libraries, one of 22 chunks and one of 20 chunks.	Uses one library of chunks that can be expanded by the user.
Post-processing	Uses post-processing to decorate the level, to fill in areas beneath the ground tiles. It is also used to evenly distribute power-ups across the level.	No post-processing is used.
Running time	15 seconds for a complete level of 256x15 tiles.	5 seconds for a complete level of 256x15 tiles.

The chunk library that mORE uses is important, because any content it creates is an interconnected whole of the chunks in the library. For this project, the chunk library is not important as it would be if mORE was not configurable by the user. If the user notices that some elements that they want in their levels are missing, they can always augment the chunk library to achieve the desired effect. For example, an element not present in mORE's default chunk library is the piranha plant. The user could easily draw a piranha plant in the level canvas and add it to the library, where it would immediately become available for use in the level generator. The default chunk library is shown in Figure 12. Additional information that the game and the level generator use is included in the rendering. For example, in every tile the anchor symbol is included, and these points are the anchor points that mORE uses to connect level segments together. The spaces marked by the word "keep" are "preserve points". These are points that are to be kept clear of overlapping geometry. Without preserve points, ceilings could be generated that were lower than three tiles high, so that only small Mario can pass, which presents a playability issue, as players that collected a mushroom power-up may not be able to finish the

level. Tiles with a solid red border on each edge are completely solid, and Mario cannot pass them on any side. Tiles with a solid border on top are impassable from the top. This particular chunk library is able to produce levels with branching paths by using chunks with three or more anchor points, which create a branch path. For example, the first chunk on the third row of Figure 12 contains two points connecting to the ground, and one point on top of a hill. One path can extend from the ground, and the other from the hill. Most chunks in the library with a single anchor point destroy the ability of the algorithm to expand the level, as they take an anchor point without providing any additional points to expand off of. These chunks are generally marked as “place last” so that a complete level structure is generated before they are placed. Chunks with two anchors often result in a linear section of play.

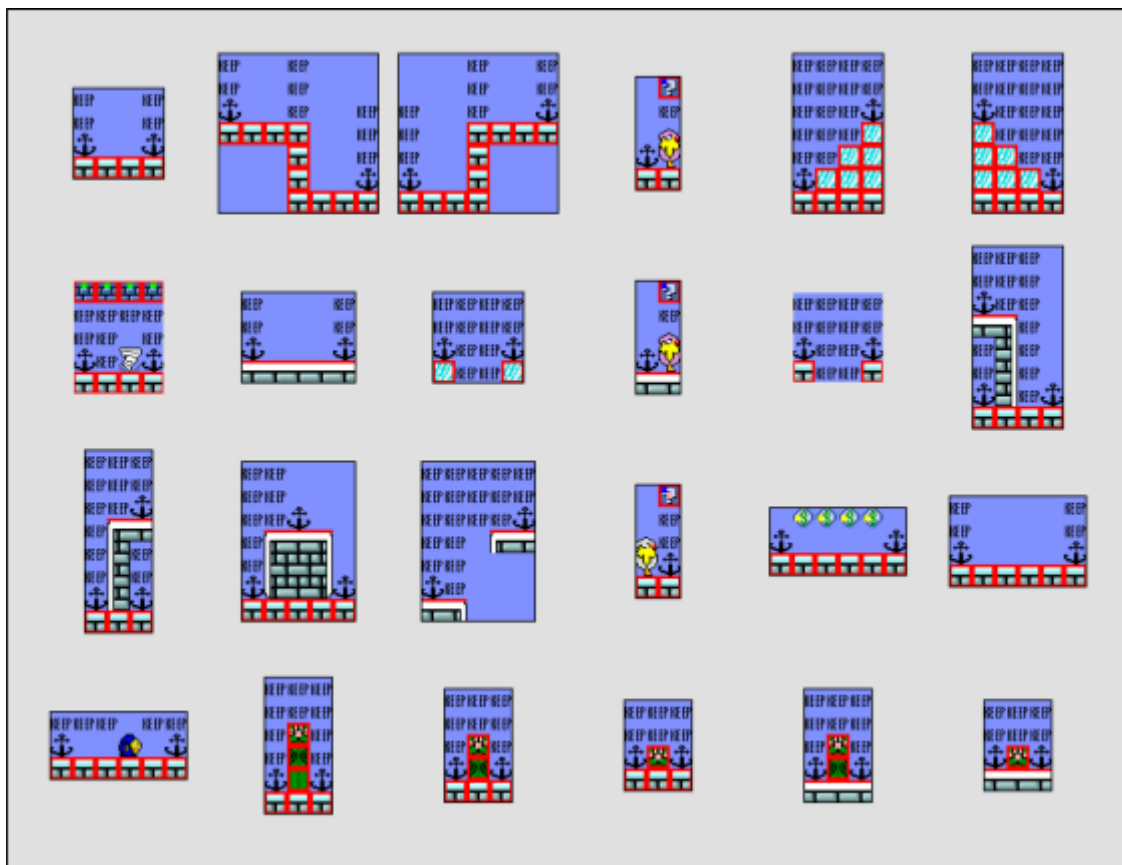


Figure 12. mORE's default chunk library

Some sample levels that mORE has generated are shown in Figure 13. A visible issue with the generator is that it produces levels that look incomplete, where hills are drawn, but only the hilltops, and the lowermost ground has an unsightly gap between it and the bottom of the

screen. However, these omissions are purely cosmetic and do not affect the gameplay of the level. The default generator of InfiniteTux, the Notch generator, addresses this issue with a function that identifies these gaps as they are built. If mORE were to resolve the issue, it would require a post-processing step to identify and fix the problem areas. In each of the sampled levels, sections of non-linear play occur in a few spots, so the player can make a choice about which path they want to take to reach the end. This effect is not nearly as common in the Notch generator, except for the occasional hill that the player can leap on for some reward.

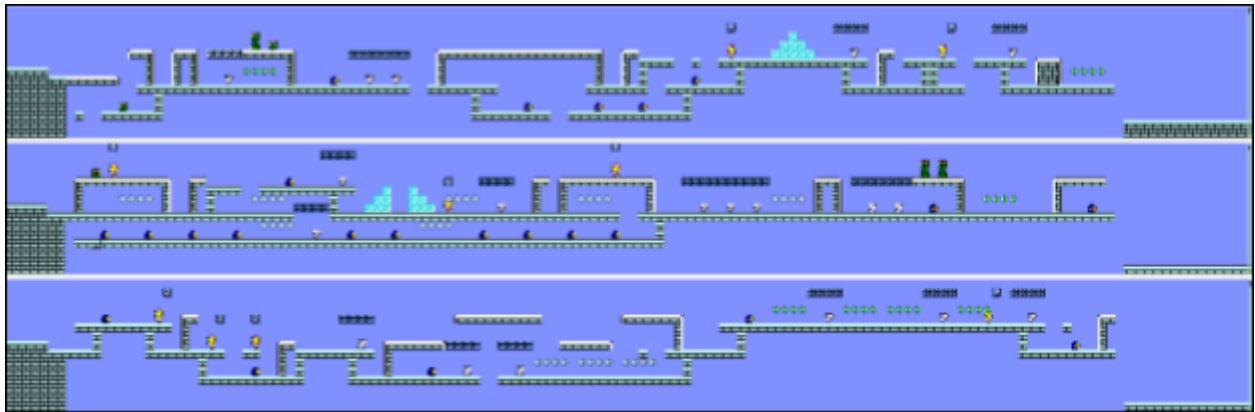


Figure 13. Three levels generated using mORE

2.3 Adding collaborative elements to level creation

So far, a level generation algorithm and general improvements to the existing level editor interface have been covered. However, the system as currently specified sits in the family of low-initiative systems. The system does not provide much in terms of feedback on the user's design, or notify the user of constraints that are important in generating 2D-levels, such as whether or not the user can actually complete the level.

To address the first issue, a section of the interface was created that shows the user information about both level pacing and a player's solution to a level. In many games, a higher enemy density indicates a higher level difficulty, as enemies present a dynamic challenge that the player must actively avoid. At the same time, the presence of power-ups will reduce the difficulty of a level. Power-ups and rewards are also often used as incentives for player exploration. Often, Mario levels are too large to fit on a single screen, and it may be difficult for the designer to quickly get an overview of the distribution of enemies and rewards. To address this issue, a time series is shown to the user that can provide a quick overview of the density of

enemies and rewards as spread across a level. At each tile, the number of enemies or rewards is recorded, and this is shown on the time series. This provides the designer with a way to see how spread out the rewards and enemies on a stage are. Using this information, the designer can recognize issues with level pacing and course-correct the level.

The second issue, playability, is much harder to account for. Since most platformer games use continuous physics, it is difficult to construct a level in a random way while ensuring playability [27]. Tanagra is able to solve this problem by using linear chunks and ensuring that adjacent chunks in a level “line up”, or the chunks start and end tiles match the y-coordinate of neighbouring chunks. However, when considering the entire space of possible Mario levels, levels must exist where neighbouring chunks do not line up. As such, this seems to be an arbitrary restriction to make sure that generated levels are always playable, but it comes at the cost of reducing the space of generated levels. Sorenson and Pasquier’s generator [30], or generators like *FUN Pledge* [20] enforce level playability by using a ballistics model for jumps. However, since the level generation algorithm being implemented is based on ORE, none of these approaches are feasible.

In order to ensure level playability, a level playthrough is simulated with an A-star agent. The particular agent that was implemented was chosen since it won the MarioAI competition, performs well, and uses a very similar level representation as InfiniteTux, so it could be integrated very easily. The agents of the MarioAI competition perform very quickly, and are guaranteed to select an action within a frame. Additionally, the agents run at uncapped framerate, and on a background thread from the level editor user interface. By taking advantage of these features, the agent was incorporated directly into the level editing process. Each time the user makes a change to the level, the agent is called to simulate playing the level. This performs fairly well, and in most cases returns results in a few seconds on a computer with a 6-core, 3.6 GHz processor. The results of the agent’s test are returned to the user in two different ways. First, if the agent was not able to clear the level, this may imply that the level is not possible. The x-coordinate of the furthest point the agent was able to reach is displayed to the user as an error message. Changes that result in the level being successfully cleared also clear the message. Additionally, a trace of the agent playing the level is created on a time series. At each frame while playing the level, the agent’s inputs are recorded. This data is shown as a series of labelled block functions for each action. From this data, the user can roughly estimate the pacing of the

level as time progresses. Sections on the graph where the only actions are left and speed probably indicate that these sections are easy. On the other hand, constant breaks in the rhythm, times where the agent had to let go of speed, or move backward, indicate that the level is more complex, or that elements in the level are causing slowdown to occur, but the system's plot doesn't offer any information on the source of the effect.

One feature of the level editor to make note of is the undo operation, which also applies to the actions of the system. If the designer does not approve of system changes, they can undo the operation quickly, and try to generate it again.

A collaborative feature of the level editor that has yet to be mentioned is the user's direct access to the underlying chunk library that mORE uses in its level generation. The user is able to directly create a new chunk for the library by drawing it on the level canvas and clicking a button in the interface. The chunk immediately appears in a separate panel, with the other chunks that the system uses. The users can modify the library by deleting or tagging chunks. A tag is a rule that the level generator uses during level generation. The only tag currently available is "place last" which can be used to force the tagged chunk to only be considered for placement after the first set of chunks is placed.

2.4 Final system

The final interface for the system is presented in Figure 14 and Figure 15. A typical workflow would be to first have the system generate a level. After generating the level, the designer consults any error messages and revises the level until it is playable. Optionally, the designer may consult the component distribution or the agent's most recent level trace to determine whether or not the level is appropriately paced. The designer may also edit the library the generator is using during their session, and regenerate parts of the level as they go. Additionally, the designer may playtest the level at any point to come to their own conclusions about the suitability of the level.

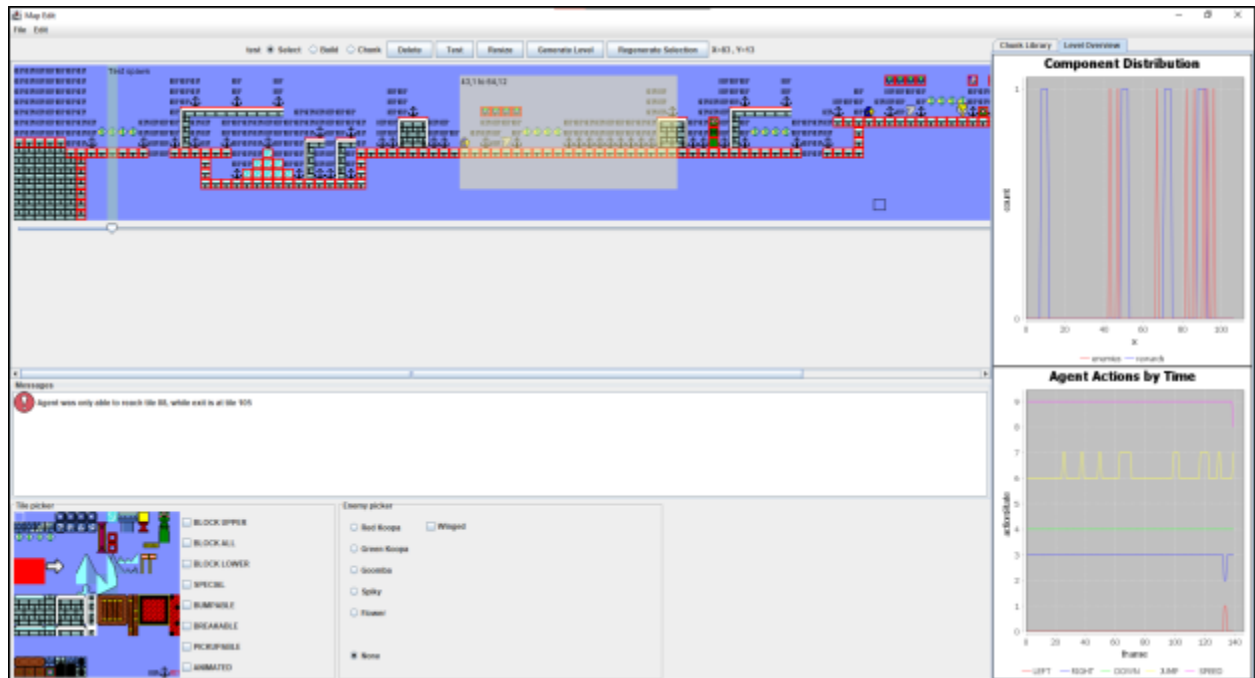


Figure 14. Final system with “Level Overview” panel open

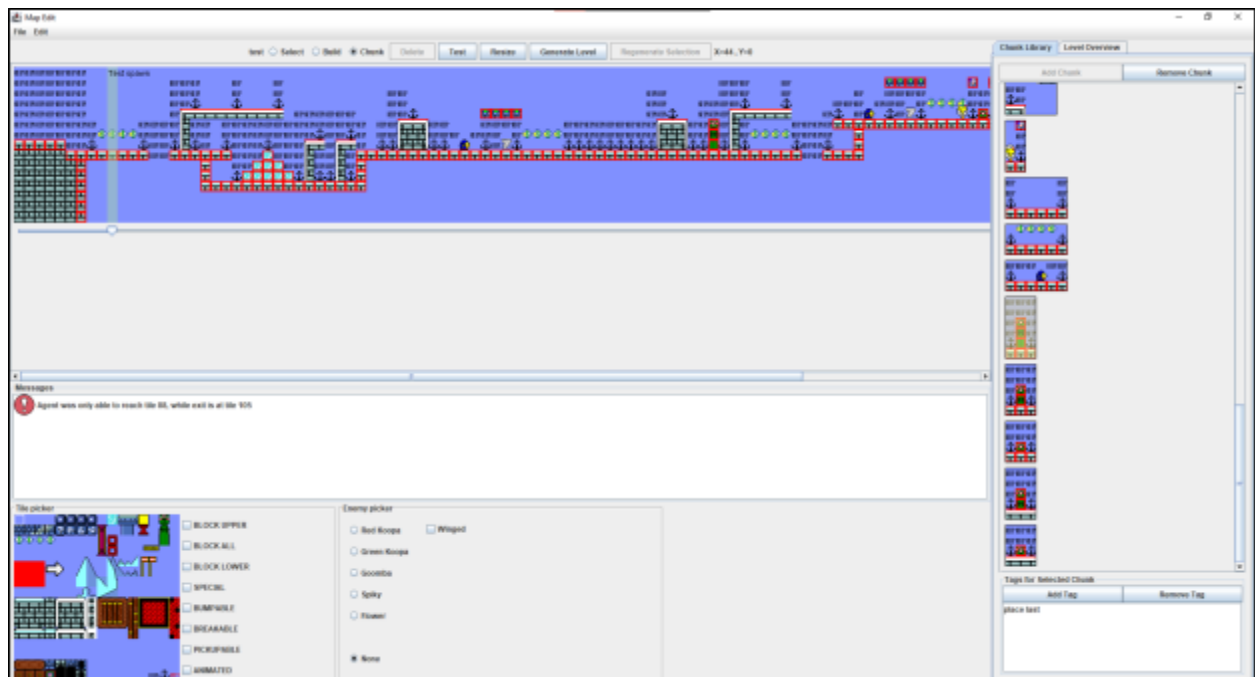


Figure 15. Final system with “Chunk Library” panel open

The successes and failures of the system as it relates to the goals defined at the start of the Methodology section will be discussed in the Results section of the report.

3. Results

For the project, three main goals were established. The usability of the level editor was to be improved to make the system easier to work with. The level generator was to be improved to better suit an interactive system, while being configurable and providing a greater amount of variety in the generated levels. Collaborative elements to level creation were to be added, so that the designer would have support when creating levels.

The usability of the system was greatly improved compared to its previous iteration. Some usability principles in particular that received a large amount of attention were efficiency, and feedback. The efficiency of the system was improved by adding bulk operations that are common to many other direct manipulation interfaces. A bulk select was added, which supports the “re-generate section”, “delete”, and “add chunk” operations. Through the use of the “Chunk Library”, the user can create patterns of level segments that can be pasted into the level via direct manipulation. This speeds up the process of level creation. The ability to generate ideas with the level generator also accelerates the level generation process, and a user that knows the inner workings of the algorithm can quickly generate a level matching their needs. Additionally, accelerators for common actions like “save”, “save as”, “open”, and “undo” were added. Feedback was provided for the actual design task itself. Whenever a change is made to a level, the user can immediately see whether or not the level is playable, and from this the user can course correct a non-playable level faster. Elements to the interface such as the “Level Overview” panel provide the user with additional information that can also be used to adjust the level, but these additions are more passive than the playability message, as they have to be interpreted by the user. These additions allow both novice and advanced users to save time, which can instead be spent on level design. However, some improvements could still be made to the usability of the system. A few usability issues that are still present in the system are a lack of visibility of system status. For example, generating a level takes a few seconds on a fast machine. This process is done on the UI thread, so the application is locked while it occurs. Instead, a loading icon, or a message should be provided to the user. The application should certainly not appear to stall. As it currently exists, the process of configuring the level generator to any degree of success is very dependent on how well the user understands how the algorithm works. No help or documentation is available for this purpose, and it certainly isn’t clear what a chunk is to a user trying to use the software for the first time.

Requirements were met in terms of the functionality that a level editor should provide as defined by Rouse. As the user edits the level, a complete representation of how the final level looks is provided. The editor does provide the user with additional information about the level, which includes playability information, information about the distribution of enemies and components over the level, information about the behaviour of each tile. The third guideline, the ability to immediately test the level, is present in the editor from a single button press. The user is also able to move the starting position if they desire, and stop testing at any point, to make it more efficient to test a small level section. The last requirement for the level editor is that it should provide the user the power to change any gameplay critical part of the level. The level editor allows this, because the user can resize the level, move the level exit, place any tile they want, and is able to place any enemies they want. Some missing option that could be added would be to allow the user to change Mario's starting state.

The level generator that was chosen is configurable. Through the use of the canvas and system dialogs, the user is able to add content that the generator can use to create levels. The speed of the level is also reasonable. On both a laptop with a low-end processor and a system with 16gb of RAM and a 12-core processor at 3.6GHz, a 256x15 level is generated in under 10 seconds, which is very reasonable for a level generator, as a level could be generated in an interactive game with this approach. To contrast, ORE generates its levels in 15 seconds with a chunk library of size 40.

To evaluate the results of the level generator in terms of producing sufficiently varied levels, a comparison against the default InfiniteTux generator, the Notch generator, was made. Previously, in the project proposal, the expressivity of the level generator was to be evaluated by its expressive range, as described by the leniency and linearity characteristics, similar to Smith's evaluation of Tanagra in [27]. Linearity measures how well a straight line fits to the level geometry, in a similar way to how a straight line is fit to a set of observations in statistics. Levels with linearity close to 1 are highly linear, while levels close to 0 are highly non-linear. On the other hand, leniency measures the amount of risk in a level, and assigns scores to a level by how likely the player is to lose based on the presence of obstacles. Being able to measure a level's leniency depends on the ability to find these gaps and dynamic obstacles in a level. In Tanagra's case, it is straightforward to do so because Tanagra understands the context that it places these obstacles in, but mORE is not as "smart", and cannot determine where gaps or certain other

obstacles exist without post-processing. Hence, the expressive range of mORE was not measured, and another metric was chosen.

Lucas and Volz created an approach for analysing the amount of variability between two levels [15], coined Kullback-Leibler Divergence (KL Divergence). KL Divergence works by counting all occurrences of patterns of some size in a level by convolving over the level. This count is then used to create a probability distribution. Two levels are the same if the KL Divergence of their probability distributions is equal to 0. Otherwise, the more positive the KL Divergence is, the more varied the levels are.

For each level generator, 10,000 pairs of levels were created. For each pair, the KL Divergence was calculated. This gives a probability distribution of the KL Divergence for the Notch and mORE generator. This process was repeated for 2x2, 4x4, and 6x6 areas. Descriptive statistics for each distribution at each feature size are provided in Table 3. To show that mORE levels are more varied than Notch generator levels, a one-tailed t-test was performed for each 2x2, 4x4, and 6x6 area. p should be less than 0.05 for the results to be significant, and to be able to reject the null hypothesis that μ_1 , the sample mean of mORE's level variability, is less than or equal to μ_2 , the sample mean of Notch's level variability. The results of these three tests are shown in Table 4, and graphically summarised in Figure 16. For each feature size, the KL Divergence distributions are compared between the Notch and mORE generators. For features 2x2 and 6x6 in size, p is less than 0.05, so the mORE generator does produce more varied levels for this feature size. For feature size 4x4, the p -value is not less than 0.05, so the null hypothesis cannot be rejected. In terms of level variability, the mORE generator does produce more varied levels with the chunk library described in Figure 12 than the Notch generator. For 4x4, it could not be shown that the mORE generator produces more varied levels at this feature size. For 6x6 features, the p -value is very small, so the distribution of 6x6 features differs significantly between the two generators. Either way, these results do show that in some cases, the mORE generator produces more varied results than the Notch generator, so in this regard it is successful. With a different default chunk library, it seems plausible that mORE could produce significantly more varied results for 4x4 features. Another statistic worth mentioning is that the distributions for mORE are more spread out than every distribution for the Notch generator. The maximum values are significantly higher than Notch's generator, and the minimum values are slightly

lower. This means that in some cases, mORE is able to produce a pair of levels that are substantially different from one another.

Table 3. Descriptive statistics for each sample

Generator and Feature Size	Min	1st Quartile	Median	Mean	3rd Quartile	Max	Sample standard deviation
Notch 2x2	1.779	7.220	8.982	9.431	11.138	39.885	3.213
mORE 2x2	-2.502	4.830	7.276	9.768	10.641	344.329	14.680
Notch 4x4	9.778	12.507	13.253	13.394	14.111	23.739	1.286
mORE 4x4	6.431	10.211	11.419	12.282	12.972	163.866	6.162
Notch 6x6	11.990	13.950	14.450	14.540	15.030	19.850	0.871
mORE 6x6	10.210	14.080	15.030	15.510	16.190	117.89	4.119

Table 4. Results of significance testing for each pair of samples

Distribution 1	Distribution 2	T-value of single tail upper t-test	Degrees of freedom	p-value
mORE 2x2	Notch 2x2	2.2438	10955	0.01243
mORE 4x4	Notch 4x4	-17.678	10868	1
mORE 6x6	Notch 6x6	23.038	10891	2.20E-16

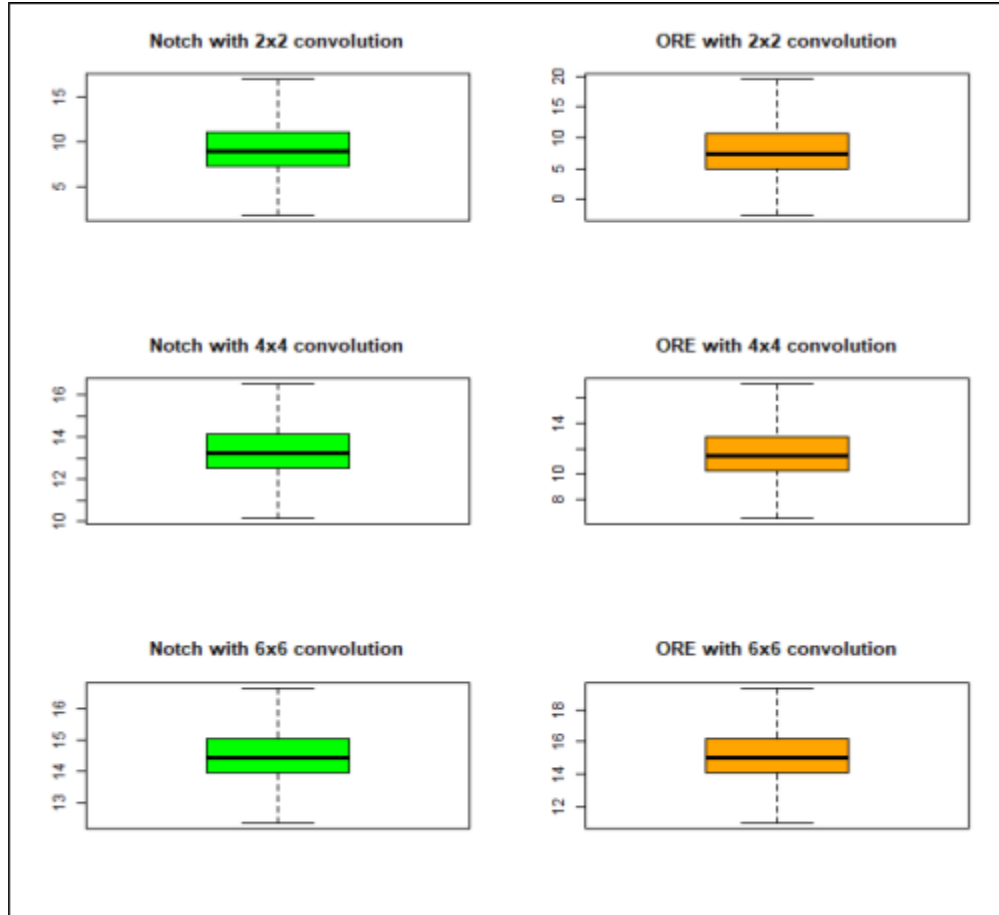


Figure 16. Distribution of KL-Divergence for pairs of levels (n=10,000)

When one looks at the previous statistics of the variability of the two generators, a question may arise: Why does the mORE generator produce more varied levels for certain features sizes when compared to the Notch generator? The mORE generator is likely able to produce these more varied levels due to two key differences between the generation approaches:

- 1) the Notch generator does not overlap its level segments, instead it attaches them by their ends,
- 2) the mORE generator is able to work with a larger library of content than the Notch generator, so it can explore more of the space of possible *InfiniteTux* levels.

To illustrate point 1), a direct comparison between two levels generated by the Notch and mORE generators is provided in Figure 17. The top image shows a level generated by mORE, annotated with the unique chunks from the library specified in Figure 12, while the bottom shows a level generated by Notch generator, with its chunks that were previously described in Figure 2. Adjacent chunks in the first image are denoted regions of different colour, while a region that combines several chunks has several different border colours. Regions with

alternating red and yellow borders are fairly linear, but regions including magenta and green borders indicate three or four different chunks in the library overlapping. There is an I-shaped region in the first image that is the result of an up-stair and a down stair overlapping with a ground region. Regions such as these produce strange features in the larger 6x6 features that KL Divergence looks at, and they are not guaranteed to appear. On the other hand, Notch levels are made of the same five or six chunks with a degree of randomness to each chunk that varies smaller features within, such as hill height, hill start, and hill end positions. Enemy and coin placement on top of these features creates some variation, but still only two variants for these features are possible, a hill with or without enemies. This may explain why there is more variation in larger-scale features in mORE levels than Notch levels.

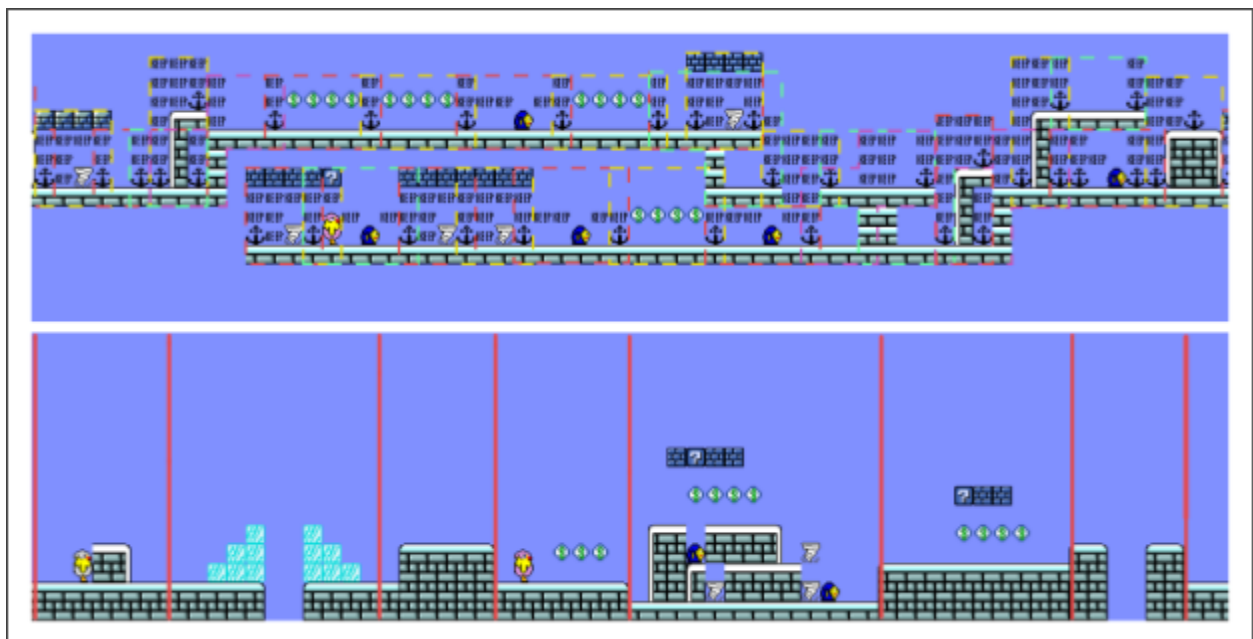


Figure 17. Comparison between two levels generated by mORE (top) and by Notch (bottom)

To begin to compare the 2x2 features produced by each generator, it would be helpful to have a visual aid. One is provided in Figure 18. The most common features in the Notch level on the left are air, plain ground, hill ground, and decorative tiles. The most common features in the mORE level on the right are air, and then various types of solid tiles. No decoration or post-processing is done in mORE, so rounded corners are only present in the chunk library, and aren't added in, meaning that this feature appears less frequently. Several other features do not appear in mORE levels due to the lack of post-processing. This has an effect on the variability of

the KL Divergence metric. For each 2x2 feature in two compared levels, KL-divergence weighs the difference between two levels higher if one level has the feature and one does not than if both levels have the feature, but it occurs more frequently in one compared to the other [15]. This effect, combined with the combining of chunks in mORE levels to create unique overlapping regions, is most likely responsible for much of the measured difference between the variation of the generated samples from both generators. It is also reasonable to expect the variation of the mORE generator to grow if more chunks were added to the chunk library it uses.



Figure 18. The 40 most common 2x2 features in a Notch (left) and a mORE (right) level

The final goal of the system was to provide a collaborative system for creating levels. Of the described roles that a mixed-initiative system can take according to [12], the system most closely fits the collaborator and informer roles. The system collaborates with the user by generating levels on command, or regenerating level sections as directed. By making use of the chunk library that the user directs it to, it is able to accept input from the user. It informs the user of errors regarding playability, component distribution, and the agent's last playthrough, but it takes no action to correct these problems itself. As a collaborative system, it would be more helpful to give the user more actionable feedback about their design than just showing graphs that can be interpreted. However, this would likely require some kind of background processing similar to how the A* agent runs on a background thread to determine playability. A more helpful suggestion to the user would consider the chunk library perhaps, and make recommendations to fill in what's missing. Perhaps the specific fixes could be provided to the user instead of the graphs, such as "there is a steep difficulty spike due to too many enemies in the area between 10 and 30", or "there is a large cluster of power-ups in the area 10 to 20,

perhaps they should be spread out”. This type of feedback could help to provide better information to the user, and would lighten the load on them by removing their need to interpret the graphs currently displayed in the “Level Overview”.

4. Conclusions & Future Work

During this project, procedural generation was discussed in depth. Several approaches for generating levels for 2D platformer games were examined. Additionally, the role of computers in the creative process was discussed. Functional and usability requirements were briefly researched to improve the utility of the designed application. From this initial research, a system was implemented that would allow a level designer to work collaboratively to create interesting InfiniteTux levels.

The goals of this project were to create a level editor that helps a game designer create varied levels for a game quickly, by working collaboratively. In these regards the system was successful. The system allows the game designer to quickly iterate through ideas by using a procedural generation approach that was shown to have increased variation over the default generation algorithm in two of three cases. The system used a rather small chunk library, and has the potential to provide the user with far greater varied ideas with a more fleshed-out chunk library. The level editor provides the user with the ability to participate in the level generation process with a fair amount of initiative being shared between the level designer and the system. The user is able to modify the most basic units that the generator works with - chunks, and is also free to modify on a per-tile basis themselves. If the user is dissatisfied with the output, they can immediately re-generate a level or a subsection of a level, to view more alternatives. However, a fair amount of power is placed in the hands of the designer. The designer must take the initiative to start changes on a level, to tell the system when it should take its turn to participate in the task, and when actions should be undone. It was shown that the implemented procedural generation approach was able to produce more varied output for 2x2 and 6x6 feature sizes than the default level generator for InfiniteTux.

Most of the focus of the project was spent on level generation, when much more can go into facilitating the designer’s creativity than having varied levels to be inspired by, or influence over the generation. One of the original visions of this project was to make editing levels as easy as writing code in a fully-featured integrated development environment (IDE). Many IDEs

provide the programmer with several different suggestions at a time. Warnings are provided when the programmer may cause a runtime error due to using an outdated function, or does an unsafe operation. Suggestions are provided to nudge the user toward better style, or to use newer features that may prove more advantageous than older features. Currently, the improved editor for InfiniteTux just provides warnings about potential playability issues to the user. One could imagine that in the future, suggestions are given about repositioning level elements to create a better rhythm, or using hills instead of solid tiles to give the user multiple paths.

This work has contributed to the field of procedural generation, and of mixed-initiative systems by incorporating a procedural generation system into a level editor to improve the level editing task by reducing creative and manual effort required by the user. The implications of this contribution suggest that a similar approach could be applied to other creative tasks such as drawing, the generation of music, or perhaps generating models of terrain. A fair amount of effort has been devoted to the specific field of platformer games, in order to gain an understanding of their structure and existing methods for procedurally creating content for these games. If a collaborative system were to be developed for any of the listed domains, the developers should also research existing generation methods in the domain, in order to find an approach that would have similar strengths to that of the mORE generator used for the improved InfiniteTux generator. A few notable strengths of the generator are its ability to overlap content, to be able to work with a user by allowing its content library to be modified, and by taking turns with the user to generate and re-generate sections of a level.

Another implication of this project is that it would prove advantageous to be able to adapt an existing content generator, instead of writing an implementation of a closed-source system. If this were done, much more time could be spent on the portion of the system that collaborates with the user. This may allow more focus to be given to the collaboration between the user and the system, so that there could be a better dialogue between them. As it stands, the current system is rather analytical, where instead it could provide interpretation along with its analysis. However, the generator part of the system would need to have an understanding of the context of the content that it produces, which may hinder its ability to quickly produce results and generate ideas for the user. Perhaps the generator could be separated into two parts, where a content analyzer determines improvements that could be made after the generator does its work. Either way, this content analyzer component would be interesting to expand on in further research.

To expand on the system's ability to provide suggestions to the user, it would be interesting if the user could first design several levels that the system analyses, and when asked for suggestions, the system could use a metric like KL Divergence to generate levels until some threshold of acceptable difference from the user's levels is reached. In other words, the system could analyse the designer's behaviour, and attempt to produce levels or suggestions for areas that are intentionally very different to what the designer would produce themselves. This feature could serve to produce ideas for the designer that they may not be able to think of themselves.

A problem often encountered during the selection of the generator and during considerations of what extra information to provide the user with, is that it seems to be a very difficult task to define what makes a platformer level fun. A single level generation approach is not dominant over others, as indicated by the wide variety in approaches that was detailed in the introduction. More research could be done on both platformer games and level generation approaches to determine both what is fun in a platformer game, and how these findings can be used in a level generator to create engaging levels.

Future work could extend further than using user-defined chunks to generate levels. Perhaps, the program could generate these chunks, or the user could act as a supervisor for a system that learns to create interesting levels. The system is highly context dependent. For each game that it's used with, a new chunk library would need to be created by a designer familiar with the game. However, the part of the algorithm that places level geometry could require changes if some tiles or level components in another game are not 1x1 in size, as they are in InfiniteTux. Regardless, it would be interesting to see this approach to level editing made more general so it could be applied to a variety of games, and not just limited to 2D tile-based platformers.

Bibliography

- [1] Blezinski, C. (2000). The Art and Science of Level Design. Available at: <https://www.gamedevs.org/uploads/the-art-science-of-level-design.doc> [Accessed January 22, 2022]
- [2] Browne, C. (2011). Evolutionary Game Design.
- [3] Charity, M., Khalifa, A., & Togelius, J. (2020). Baba is y'all: Collaborative mixed-initiative level design. *2020 IEEE Conference on Games (CoG)*, 542–549.
- [4] Compton, K., & Mateas, M. (2006). *Procedural Level Design for Platform Games*. 109–111.
- [5] Dahlskog, S., & Togelius, J. (2012). Patterns and procedural content generation: Revisiting Mario in world 1 level 1. *Proceedings of the First Workshop on Design Patterns in Games*, 1–8. <https://doi.org/10.1145/2427116.2427117>
- [6] Delarosa, O., Dong, H., Ruan, M., Khalifa, A., & Togelius, J. (2021). Mixed-initiative level design with rl brush. *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, 412–426.
- [7] Guzdial, M., Liao, N., Chen, J., Chen, S.-Y., Shah, S., Shah, V., Reno, J., Smith, G., & Riedl, M. O. (2019). Friend, collaborator, student, manager: How design of an ai-driven game level editor affects creators. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 1–13.
- [8] Iyer, V., Bilmes, J., Wright, M., & Wessel, D. (1997). A novel representation for rhythmic structure. In *Proceedings of the 23rd International Computer Music Conference* (pp. 97-100).
- [9] Jennings-Teats, M., Smith, G., & Wardrip-Fruin, N. (2010). Polymorph: Dynamic difficulty adjustment through level generation. <https://doi.org/10.1145/1814256.1814267>
- [10] Kazemi, D. (2009). *Spelunky Generator Lessons*. <https://tinysubversions.com/spelunkyGen/>
- [11] Kazemi, D. (2009, September 29). *Spelunky's Procedural Space*. <http://tinysubversions.com/2009/09/spelunkys-procedural-space/>
- [12] Lawson, B., & Loke, S. M. (1997). Computers, words and pictures. *Design Studies*, 18(2), 171–183. [https://doi.org/10.1016/S0142-694X\(97\)85459-2](https://doi.org/10.1016/S0142-694X(97)85459-2)
- [13] Liapis, A., Yannakakis, G. N., & Togelius, J. (2013). Sentient sketchbook: computer-assisted game level authoring.

- [14] Lubart, T. (2005). How can computers be partners in the creative process: Classification and commentary on the Special Issue. *International Journal of Human-Computer Studies*, 63(4), 365–369. <https://doi.org/10.1016/j.ijhcs.2005.04.002>
- [15] Lucas, S. M., & Volz, V. (2019). Tile Pattern KL-Divergence for Analysing and Evolving Game Levels. *Proceedings of the Genetic and Evolutionary Computation Conference*, 170–178. <https://doi.org/10.1145/3321707.3321781>
- [16] Mawhorter, P., & Mateas, M. (2010). Procedural level generation using occupancy-regulated extension. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 351–358. <https://doi.org/10.1109/ITW.2010.5593333>
- [17] Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann Publishers Inc.
- [18] Nielsen, J. (2005). *Ten usability heuristics*. <https://www.nngroup.com/articles/ten-usability-heuristics/>
- [19] Norman, D. (2013). *The design of everyday things: Revised and expanded edition*. Basic books.
- [20] Ripamonti, L. A., Mannalà, M., Gadia, D., & Maggiorini, D. (2016). Procedural content generation for platformers: Designing and testing FUN PLEdGE. *Multimedia Tools and Applications*, 76(4), 5001–5050. <https://doi.org/10.1007/s11042-016-3636-3>
- [21] Rouse III, R. (2004). *Game design: Theory and practice*. Jones & Bartlett Publishers.
- [22] Schneiderman, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(08), 57–69.
- [23] Shaker, N., Togelius, J., Yannakakis, G. N., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., Smith, G., & Baumgarten, R. (2011). The 2010 Mario AI Championship: Level Generation Track. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(4), 332–347. <https://doi.org/10.1109/TCIAIG.2011.2166267>
- [24] Shaker, N., Nicolau, M., Yannakakis, G. N., Togelius, J., & O'Neill, M. (2012, September). Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 304–311). IEEE.
- [25] Shaker, N., Togelius, J., & Nelson, M. J. (2016). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer. ISBN 978-3-319-42714-0.
- [26] 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*, 75–80. <https://doi.org/10.1145/1401843.1401858>

- [27] Smith, G., Whitehead, J., & Mateas, M. (2010). Tanagra: A mixed-initiative level design tool. *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, 209–216. <https://doi.org/10.1145/1822348.1822376>
- [28] Smith, G., Whitehead, E., 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*. <https://doi.org/10.1109/TCIAIG.2010.2095855>
- [29] Smith, G., Othenin-Girard, A., Whitehead, J., & Wardrip-Fruin, N. (2013, November). Endless Web. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.
- [30] Sorenson, N., & Pasquier, P. (2010). *The Evolution of Fun: Automatic Level Design Through Challenge Modeling*.
- [31] Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3), 172-186.

Appendix A: mORE Level Generation Algorithm

```
// createLevel by expanding anchor points within level with compatible
// chunks from the first set chunk library.
// Halts when no more chunks can be placed at any anchor point, then repeats
// for the second set of chunks.
// Note that a chunk is a small level fragment.
//
PROCEDURE createLevel(chunkLibrary: contains firstSet, secondSet) {
  create a starting platform
  create an initial anchor point

  anchorPoints <- {startingAnchor} // set of all anchors
  terminalAnchors <- {} // set of anchors that cannot be expanded
  FOR EACH (currentChunkSet IN {chunkLibrary.firstSet,
    chunkLibrary.secondSet}) {

    // loop until all anchor points are terminal, cannot be expanded on
    WHILE (terminalAnchors.size() < anchorPoints.size()){

      anchorToExpand <- anchorPoints.selectRandom()
      // get chunks that can be placed at the selected anchor without
      // overwriting existing level geometry or spaces marked 'keep'
      compatibleChunks <- filter(chunkLibrary, anchorToExpand)

      IF (compatibleChunks.size() = 0) {
        terminalAnchors.add(anchorToExpand)
      } ELSE {
        selectedChunk <- compatibleChunks.selectRandom()
        integratedChunk <- integrateChunk(
          selectedChunk, anchorToExpand)

        FOR EACH (anchorPoint IN integratedChunk.anchorPoints) {
          anchorPoints.add(anchorPoint)
        }
      }
    }

    anchorPoints <- anchorPoints.union(terminalAnchors)
    anchorPoints <- anchorPoints.subset(0, anchorPoints.size()/10)
    terminalAnchors <- {}
  }

  create an ending platform
} END createLevel
```
