Procedural level generation for collaborative puzzle-platform games.

Benjamin van Arkel

Bachelor thesis submitted to
Amsterdam University of Applied Sciences

Advisor:
Daniel Karavolos (Amsterdam University of Applied Sciences)

Supervisor:
Anders Bouwer (Create-IT Kenniscentrum)

August 11th 2015
Abstract

This paper documents my research project regarding procedural level generation for collaborative puzzle-platform games. The goal of this research is to find out what the requirements are to have a level generator create levels that can be played in a puzzle-platform game for two players. To support my claims have also created a proof of concept in the Unity game engine which is able to parse the levels from the level generator into an actually playable level. For this research I look at what kinds of cooperation are possible and which one is most the applicable and why. I also look at what game mechanics I want and how I implement these into the level generation process. After this, I describe the resulting proof of concept and give an explanation on how I created the parser. Finally, I deliver a conclusion based on my level of success regarding the level generator and the final version of the proof of concept.

Acknowledgements

I would like to take the time to thank the Amsterdam University of Applied Sciences for granting me the opportunity of doing my thesis and research project at the school itself. Of course my utmost gratitude to both Daniel Karavolos and Anders Bouwer of the Amsterdam University of Applied Sciences for guiding me for the duration of this project, as well as giving constructive feedback on my work. Your feedback and input were invaluable for the duration of my research and helped me time and time again during the development of both the level generator and the proof of concept.

I would also like to thank Dop Terlingen for bringing Daniel’s research to my attention. Without that, I would not have even been able to do this to begin with.

And finally a thank you to all the students that took the time to keep me company during the many hours spent in study rooms, as well as giving me advice and feedback on my research.
Glossary
The following list contains terms and definitions commonly used throughout this document.

**PCG**
Short for “Procedural Content Generation”, this term is used to describe the generation of content specifically for video games.

*(Game) design patterns*
A type of format to document recurring situations and decisions within a game[8].

**Tile**
A single tile inside of Ludoscope[16]. Tiles signify objects within the level space of Ludoscope.

**Segment**
A collection of tiles that has been “blown up” from a single tile. This happens when the level is split and enlarged after the level layout has been generated.

**Templates**
Templates are the layouts of actual gameplay areas that can be implemented into segments by Ludoscope.

**Level Space**
Term used to describe the layout of the level in Ludoscope.

**Game Space**
Term used to describe the layout of the level in Unity.
Table of contents

Abstract (Dutch) ................................................................................................................. 2
Abstract ................................................................................................................................. 2
Acknowledgements ............................................................................................................... 2
Glossary ................................................................................................................................. 3
Preface ................................................................................................................................. 5
1. Introduction ...................................................................................................................... 6
   1.1 Establishing the research ......................................................................................... 7
   1.2 Risk assessment ....................................................................................................... 8
2. Different types of multiplayer interaction ....................................................................... 9
3. Game Mechanics ............................................................................................................. 10
4. Design patterns ............................................................................................................... 13
5. Building the level generator ......................................................................................... 14
   5.1 What is Ludoscope? ............................................................................................... 14
   5.2 Defining the Ludoscope alphabet .......................................................................... 16
   5.3 The generation process ......................................................................................... 17
      5.3.1 Create Start ..................................................................................................... 17
      5.3.2 Create Path Right / Left ................................................................................. 18
      5.3.3 Split And Border ............................................................................................ 20
      5.3.4 Fill Templates ................................................................................................. 21
      5.3.5 Fill Encounters ............................................................................................... 24
      5.3.6 Fill Stationary Hazard (sHaz) ......................................................................... 25
      5.3.7 Fill Empty Templates ..................................................................................... 26
      5.3.8 Make Unity-compatible .................................................................................... 26
      5.3.9 The end result ................................................................................................. 26
6. Developing the proof of concept .................................................................................... 27
   6.1 Choosing an engine ................................................................................................. 27
   6.1 Setting up core gameplay ....................................................................................... 28
   6.2 Creating the parser ................................................................................................. 28
7. Conclusion ....................................................................................................................... 30
8. Future work ..................................................................................................................... 31
References .......................................................................................................................... 33
Appendix A, Phasing of the project ................................................................................... 34
Appendix B, Game Design Patterns .................................................................................. 35
Preface

I came across the idea of modular development when I visited a lecture by Joel Burgess explaining how they built game environments for their game “The Elder Scrolls: Skyrim” [1]. Burgess gave everyone a demonstration on how the workflow was put together and how they used it to create sets of building blocks that designers could then mix and match to create immense dungeons in very little time. Ever since seeing that demonstration, I have been fond of the idea of using tools that allow designers to mix and match their own game environment dynamically. This set me on the path on discovering the world of procedural content generation (or PCG) in videogames. According to Hendrickx et al [2] PCG is “the application of computers to generate game content, distinguish interesting instances among the ones generated, and select entertaining instances on behalf of the players”.

Needless to say my interest was piqued when a teacher told me about the research project of one of her colleagues, Daniel Karavolos, which was centered around procedural level generation. Procedural level generation entails the use of a PCG system to dynamically create a game environment. A meeting was arranged in which Daniel explained how he was working on a tool that could generate two-dimensional levels for videogames called Ludoscope[16]. Ludoscope is a node-based procedural level generator, allowing me to set up certain rules that allows the generator to generate adjust and change templates inside of a 2D tilemap. Depending on the amount of rules and templates I create, they can adjust how dynamic and varied the level might end up being. Further explanation on the workings of Ludoscope will be given in chapter five.

For this research project I will look at the possibilities of applying procedural level generation for a collaborative puzzle-platform game. A proof of concept will be developed in Unity using Ludoscope to test if certain game design patterns can be dynamically placed so that players are required to work together to complete any level that Ludoscope would generate for them. The main goal here is to create a system that can successfully generate a playable level that requires more than one player to finish. For this goal to be met, I will test various methods of level generation within Ludoscope to see if this is possible, as well as creating a parser that can translate the files from Ludoscope into a full 3D environment within Unity.
1. Introduction

Procedural level generation is not a new concept to the world of game development. The game Dwarf Fortress [3] was able to generate a limitless 2D world complete with its own history. Minecraft [4] showed the mainstream audience randomly generated 3D worlds for them to play in and creating new adventures for them every time they started a new game. Now, games such as No Man’s Sky [5] are working on procedural content generation systems that would be able to generate entire planets complete with terrain, wildlife and vegetation.

However, for all of game levels and worlds, very few actually focus on gameplay centered around collaboration. Most games with procedural content generation would add cooperation as a feature by adding a second player, but never actually design and include mechanics that would make cooperation a necessity to complete the level. A game such as Minecraft could encourage cooperation by making it easier to build structures when multiple people are working together, but it is never mandatory.

As such, this thesis will focus on my research done on the application of procedural content generation for collaborative puzzle-platformer games. The decision to focus on the puzzle-platform genre was made after observing several genres that could employ collaborative gameplay. Out of these genres, the puzzle-platform genre seemed to be the genre with the most reliable game mechanics, relatively easy to set up and offered opportunities to add new kinds of game mechanics.

The goal of this project is to create a dynamic level generation system that, with the push of a button, is able to render playable, diverse and challenging levels that employ a variety of different game mechanics and might even create entirely new situations by combining different mechanics together.
1.1 Establishing the research

This chapter contains information regarding the value of this research for the video game industry, as well as the research questions that have been established.

Right now, procedural level generation seems to be implemented mainly for single-player games with cooperation being put on as an extra feature. With this research I hope to show not only the promise of procedural level generation for collaborative games exclusively, but to give people unfamiliar with procedural level generation an idea of what might be possible.

To my knowledge, the subject of adapting PCG for level design in cooperative games has not been looked into before. Looking into the possibilities of this subject could prove to be beneficial to the rest of the video game industry, as it will allow for insight on what differences there might be between PCG for cooperative games and the way PCG is being applied to current games. This has resulted in my decision to focus this research on generating levels that focus on cooperative gameplay, requiring a minimum of two players to complete. After having made this decision, I established the following research question:

“How can I create a system that procedurally generates a puzzle-platform level that requires a minimum of two players to complete?”

To help in formulating a well-supported answer to the research question, I have established the following subquestions:

- What kinds of multiplayer interaction are possible in multiplayer platform games?
- Which mechanics can be created procedurally for more than one player?
- In what kind of ways can these mechanics be applied in the proof of concept?
- How can these mechanics be translated into design patterns?
- How can I translate these design patterns into something I can implement into the level generator?
- What are the steps that need to be made to have the level generator reliably create playable collaborative puzzle-platform levels?
1.2 Risk assessment
Before approaching this research, a summary was made to list possible dangers and risks that could be encountered for the duration of this project.

Reliance on external development tools
Ludoscope, the tool being used for generating the levels is still being developed itself. While proven stable enough to be used for this project, certain usability issues might come up during the writing of this papers. The amount of risk can be reduced by keeping in contact with the developers and Ludoscope and asking for support or fixes whenever bugs hamper the development process.

Tying together the level generation and game mechanics
It is important for the final product to be able to generate levels that can be finished and also utilise the given mechanics in a way that players are required to interact with it before they can move on. I hope to minimise this risk by carefully planning out the way layouts in Ludoscope are handled, as well as calculating the degree of freedom the players will have.

Keeping the camera focused on both players without hampering gameplay
Both players must retain the ability to look at the rest of the level in their vicinity. For example, being able to look ahead for any kind of traps or enemies. I will minimise this risk by implementing a script that dynamically follows the player, possibly zooming in and out depending on the distance between players to keep an overview of their surroundings.

Designing templates/rooms oriented around collaborative gameplay.
A great amount of attention must be paid to the design of the “templates” for the rooms so that each one connects properly to the one next to it. It is also important that the placement of the mechanics, such as enemies or level obstacles are placed in a way that would “make sense” to the player and does not make other aspects impossible. An example would be spike obstacles spawning in front of a doorway that the player would need to pass through.

In addition, it should also be noted that the rooms should be generated in a way that would encourage collaborative play, as opposed to simply spawning rooms that could be completed by one player by himself.

Encouraging collaborative play
It is important that the game mechanics are implemented in a way that encourages the players to work together. A simple way of doing this would be to create an obstacle that cannot be passed if the player is on his own. This would make it necessary for both players to work together to proceed further through the level.

Limited amount of data to prove points for research questions
Since the main research question only focuses on the plausibility of such a system being possible, it is hard to form a proper conclusion on whether or not I have found a sufficient answer. This problem will be solved by setting certain demands that the generator must adhere to.
2. Different types of multiplayer interaction

The development for types of multiplayer can be divided into separate categories. Using information from Zagal et al. [6], I can establish that there are three different kinds of interaction possible in multiplayer games. Each kind of interaction affects the way players interact with each other within the game and influences the way the game is played. I will visit each of these methods and explain which of these categories I will use for my research, below.

**Competition**

Competitive games always requires players in a team to confront players on an opposing team. While the goals of each team could vary, they must achieve their own goal while preventing the other team to achieve theirs as competitive games always have a winning team and a losing team. Example of this include games such as Counter-Strike[7] and League of Legends [8].

**Cooperation**

A game can be cooperative if the players have interests that are “neither completely opposed nor completely coincident”. Working together is not required, but players will be confronted with situations that makes this possible, usually resulting in a positive outcome for both players. However, these outcomes might not always be equally positive for all players. Zagal et al. [6] sums this possibility up perfectly by stating that “in a cooperative game, nice guys (collaborators) can finish first, as long as they make sure they are not being taken advantage of”. Games that feature cooperative gameplay include Left 4 Dead[9] and Castle Crashers [10].

**Collaboration**

The opposite of competition, collaboration demands that players work together as a team. Completing a level or game is only possible if players manage to coordinate their actions in such a way that they can traverse the obstacles that might be encountered in a level.

Collaboration differs from cooperation in the sense that collaboration states that all players must have a common goal where they all share their rewards and penalties, while in the case of cooperation, one player might benefit more from certain actions than the other player and rewards and penalties are not shared. Examples of collaborative games are Battleblock Theater [11] and Portal 2 [12].

**Choosing a category**

Out of these three categories, this research will be focused on collaboration. Creating levels for a competitive game mainly focuses on balancing the level so both teams have an equal chance of winning. In comparison, creating a cooperative or collaborative level allows for more focus on the design of mechanics oriented around playing together.

Between cooperation and collaboration, collaboration was chosen as it will force players to work together to complete the level and make it possible for them to tackle the obstacles that the generator creates. This compared to cooperation, which makes teamwork optional, allows for more focus on the design of mechanics and situations where players are required to work together.
3. Game Mechanics

Before starting to work on the level generator itself, it is important to figure out what kind of gameplay to use for the prototype. Doing this first allows me to establish a clear concept of what the final prototype should look like and also allows me to set up certain demands and requirements for the level generator itself later on.

Requirements for a puzzle-platformer

To create a puzzle-platform game, there are a few minimum requirements that must be met before it can be defined as such. A platform game, for example, requires the players to be able to jump, so they can traverse platforms. To create a puzzle-platformer, this requirement must be paired with a certain mechanic that creates interactive puzzles that the players can solve.

For basic gameplay, the players must be able to run, jump and interact with objects in the game space in different ways. Think of activating a lever or being able to pick up a player or object. The game must also include enemies or obstacles that challenge the player’s skill and offer variation in gameplay. For the implementation of puzzles, players will be required to work together to solve gate and lever puzzles which are also randomly generated by the level generator.

Jumping

Jumping allows players to travel both horizontally as well as vertically and is, as mentioned before, an important mechanic in any platform game. Both players have a set jump height and limited movement while in the air, being only able to slightly steer themselves in certain directions.

Collaborative Platforming

Players will be able to use each other as a means of reaching certain locations in the level by standing on each other’s heads. This can also be done while a player is in the air. For example, Player One jumps off of the top of Player Two’s head while being at the top of his own jump height, allowing Player One to jump almost twice his normal jump height. This mechanic requires an increase in timing and coordination between both players to work properly and serves as an encouragement for the players to work together. The mechanic itself will mostly be implemented as a puzzle element in combination with some of the other mechanics described below.

Obstacles

Designed to hinder the player’s progress, challenge the players’ skills and creating tension, the obstacle is an object placed in the level that will hurt the players if they touch it. Obstacles are an important mechanic in the game as it puts the sense of challenge in the game.

Obstacles can be spawned with different kinds of behavior, being either stationary, moving or timed. Stationary obstacles do exactly as the name suggests, standing still until an unfortunate player hits it. Moving obstacles move between two set points with a set speed and the timed obstacle which turns “on and off” at set intervals. The last two obstacles are harder to overcome in comparison to the stationary obstacles, as it requires an increased amount of attention to the timing of the players.
**Enemies**
Enemies could be categorized as obstacles in the sense that they can hurt the players and create both challenge and tension. However, an enemy is also very different in the sense that it can be defeated by the players. Be it through shooting or jumping on top of it, an enemy can offer a more interesting way of challenging the players because they can interact with it. An enemy might also seek out a player when in range of one, attempting to damage them by running into them.

**Moving platforms and elevators**
Moving platforms do exactly as the name suggest, being platforms that the player can stand on that move back and forth in between two set points. These platforms will only be able to move left to right with varying speeds. Elevators function similarly as moving platforms, but move vertically between two points instead of horizontally.

Adding moving platforms and elevators creates more variety in the way the player can move through the level. These mechanics add a new sense of challenge when used as the only means to bridge a certain part of the level and it requires players to actually plan their jump as to not miss the platform. Pairing this mechanic with either obstacles or enemies can also create a bigger sense of tension as this also adds the risk of players being able to die.

**Trampolines**
The addition of trampolines creates another way of traversing the level. When a player jumps up against a trampoline, they will be bounced into the opposite direction with a set amount of force. As with the previous “moving platform” mechanic, trampolines can be combined with other mechanics to create more challenging and tense situations for the players. For example, I can add enemies in the trajectory of a trampoline, meaning that when a player bounces off of it, he will have to take in account the danger of possibly hitting one of these enemies. This will result in more tension and challenge for the player, as well as requiring increased skills in timing to avoid hitting anything dangerous.

**Gates and levers**
This mechanic will form the base of many of the “puzzle” sections within the prototype. Somewhere during the level, players will encounter a gate that halts their progress through the level. To pass this gate, players must find and interact with the proper lever somewhere nearby to open this gate. There is some variety amongst the types of gate / lever systems that might be created. Some levers might be set on a timer, meaning players must be quick to both get through the gate before it closes. Other levers might require a player to stay near the lever while the second player goes through and finds a way to get the first player through the gate as well.

While not being one of the most creative puzzle-elements known in video games, this mechanic is easy to develop and can be combined with other game mechanics such as obstacles and enemies in an attempt to keep it more interesting.
Grabable objects
Players might be able to grab certain objects in the game world. These objects can be pushed, pulled and/or thrown around by the players. These objects might serve as simple boxes that the player can use as a platform. Objects with this property can be used as puzzle elements within the level.

For example, the players spot a grabable box at the top of the ledge, but only barely lacking the jump height to reach the ledge on their own. Thus, they figure out they can reach the top by using the Collaborative Platforming mechanic discussed earlier in this chapter. Player One jumps on top of Player Two’s head and manages to reach the ledge with the box. Next, Player One can grab and push the box down the ledge to Player Two, who can then position it in a way that he can use the box as a new platform, giving him the little bit of extra height he needs to reach the ledge as well.
4. Design patterns

With the game mechanics established, I move on to see in what way these mechanics can be applied in combination with the level generator. Instead of simply generating certain rooms that contain a random mechanic, I have decided to instead take the mechanics and categorize these into so-called “game design patterns” [13]. As stated by Bernd Kreimeier, design patterns are “simply conventions for describing and documenting recurring design decisions within a given context”.

Applying this to the current level design mindset allows me to create patterns in the form of gameplay situations. I have created a template which allows me to easily set up these design patterns. This template states that each pattern must contain the following:

- An abstract name with either the label SP and/or COOP, applicable to either one or both players.
- A brief description of what the pattern is.
- The affordances that the pattern has.
- The resulting effect on the players’ experience.

An example of one of the design patterns can be found below:

**Environmental hazard (SP)**

Description:
The stationary hazard serves to slow down the *pacing* by placing an object in the level that hurts the player if touched. To pass the obstacle, a player must have the skill to combine *timing* with

Affordances:
- Size of the hazard
- Amount of hazards placed
- Type of hazard (stationary, moving or timed move)

Consequences:
The environmental hazard serves to slow down the pacing of the game by making the player pay extra attention to their movements. It also increases the challenge due to the fact that the player will be hurt if he/she fails and hits the hazard.

By implementing these design patterns, I now have a list of possible gameplay situations that players might come across. I then decide to further categorize these patterns into specific “encounters”. For example, both the “environmental hazard” and “horizontal enemy” patterns are similar to each other because both are an element of danger to the players. As such, I can now fit both of these patterns, as well as patterns that serve a similar purpose into the category of “danger encounter”.

What this does is that when creating the level generator I can point out that, instead of indicating I want certain enemies or obstacles placed somewhere randomly, I can tell the generator to instead randomly pick an encounter that forms an element of danger to the players. From here the level generator might decide to create either an environmental hazard, horizontal enemy, or whatever else kind of pattern I might include into the danger encounter category. This allows for an extra layer of generation within the generation process.

Further explanation on how these encounters are designed and implemented into Ludoscope will be given in chapter 5.4 and a full list of all the game design patterns created is documented in the appendix B at the end of this paper.
5. Building the level generator

To build a functioning level generation system, I decided to use the Ludoscope tool [14]. The tool already contains a large number of functions and a number of prototypes I was shown proved to me that it can be a reliable level generator if used correctly [15]. My supervisor for this project has also spent time using and developing Ludoscope and allows for me to more easily reach out for help when I get stuck.

5.1 What is Ludoscope?

Ludoscope is a tool developed by Joris Dormans [16] and is made to dynamically create 2D levels for video games with a system of modules, tiles and grammar. Each of the core aspects of Ludoscope will be explained in this section, as well as their importance to the generation process.

Tile definition

As Ludoscope is a 2D level generator, it uses a tilemap system to visualize the level. Any kind of block or object inside of the level is represented with a tile, a small colored square that I can customize to their desire. This allows me to create my own assortment of tiles that I feel are necessary to define their level. The higher the amount of tiles that are defined, the more the variety increases. However, this also increases the complexity of the level generation process as it might make it harder for me to have all the different tile types to mix and match with each other in the correct way.

Tiles can also be given certain values to accompany it. The purpose for defining these values is that certain tiles might require aspects set them apart from other tiles just like it. An example would be the gate and lever systems. Each gate and each lever requires a corresponding ID that matches the right ones together. These variables allow the generator and the parser to distinguish which gate matches which lever inside of the level. This makes the assigning of variables very important in the process of level generation.

Figure 2. The levelStart tile, an example of a tile within the alphabet.
Rules
Once I have defined the tiles that are to be used by the generator, they can begin defining the rules that the generator has to adhere to when generating the level. The process is as follows: I can define a situation that the generator might encounter during generation in the so called “left hand” and then defines all possible outcomes when coming across this situation in multiple “right hands”. The amount of rules you define and how you define them is crucial in the generation process. Creating rules that are too specific will result in the level feeling like a cookie-cutter level with the same patterns repeating over and over. It is important to keep in mind that in this project the generator is supposed to do most of the generating. It is possible for me to create various rules that can work in sync with each other through the use of Encounters, which will be explained in chapter five.

Recipes and Tilemap Operations
Besides creating the grammar, I can also create recipes. Recipes are used to store a set amount of commands within Ludoscope. Within Ludoscope itself there are several commands called “Tilemap Operations” that I can save. An example of this is the MarkPath command. Markpath allows for a simple path of a defined type of tile to be generated between two tiles. These tilemap operations can be recorded into recipes so they can be executed at a certain module.
### Modules

As mentioned before, Ludoscope utilizes modules that can carry out certain tasks in the generation process. Modules can be linked together to create a step-by-step process that allows modules to iterate onto each other. This method allows for the creation of our procedural level generator. Below in figure 1, you can see an example of one of the older versions of the generator. Each module has its own function within the generator. These modules will be explained more thoroughly later in this chapter, but this can give a general idea on how the generator might work.

![Figure 5. The level generation process visualized in the form of modules.](image)

### 5.2 Defining the Ludoscope alphabet

Within Ludoscope, the alphabet is the term used for the collection of all the tiles I have defined for Ludoscope that Ludoscope can utilize within the project. It is important to define as much of the alphabet as possible before starting to work on the generator itself, as it will help you later on when designing rules for later in the project. An example of this would be the piece of alphabet that I use for shaping the level geometry, displayed below in figure 2. These tiles are used to indicate the direction the player will take while traversing the level. Out of these tiles, tiles “1H” and “1V” are fairly straightforward, signifying a simple horizontal or vertical segment of the level respectively. The tiles that follow signify the different types of corners that could be generated. Tile “LCU” is an annotation for Left-Corner-Up, indicating that the player will approach this segment from the left, then move through the corner upwards and a corner that supports this kind of movement must be placed accordingly. Defining these types of tiles with this kind of annotation helps the developer in seeing how players will progress before changing the tiles into actual rooms suited for gameplay.

![Figure 6. Part of the alphabet used for the generating of the path of the level.](image)

The alphabet is also color-coded with various aspects of the generation process kept in mind. Tiles with a white background usually indicate “level-direction” tiles, while tiles with a red background signify an object that could be hazardous for the player.

Besides this, there are also some different shading of a color for certain tiles, these are to separate the “object” tiles from the “encounter” tiles. “Object” tiles are tiles that indicate actual objects that will be created in-game, while “encounter” tiles are tiles that are used to define a certain situation in a part of the level. How these “encounter” tiles are used will be discussed in chapter 5.3.5.

Finally, this way of managing the alphabet with color codes and logical annotation helps me in visualizing my level inside of Ludoscope and seeing what the flow of the level will be.
5.3 The generation process
In the following subchapter I will explain each of the steps that were necessary to create a level generator that reliably generated puzzle-platforms levels aimed at collaborative gameplay.

5.3.1 Create Start
The first step of the generation process is to figure out what the size of the level will be. The gameplay is focused mainly around horizontal platforming, meaning that most of the time the players would be moving in a horizontal direction but still allowing for segments where the players must climb or descend in the level. To keep an overview of what the level will look like globally, I made the decision to make the level layout six tiles wide and three tiles high.

Within this layout, I want the player to always start on the left side of the screen. To do this, I allocate all the tiles on the left most side of the level as “Start” (s) tiles. After placing these three tiles, I create a recipe that removes two off these Start tiles at random, leaving only one start tile which will then become the starting position of the level.

After this is done, a single tile adjacent to the start tile will be changed into an “End” (e) tile containing the variable firstPath=true.

Right now the End tile only signifies the first step of the path that the level will follow but later in the process the firstPath variable will serve to indicate in which direction the Start tile will need to connect to the path, further explanation for this is given in chapter 5.3.4.

Figure 7. Creating the End tile.
5.3.2 Create Path Right / Left

Now that I have a Start tile along with our End tile as a first step of the path of the level, I can now proceed with expanding this path. To expand this path, I first created the CreatePathRight module, of which the function can be deduced by the name alone. CreatePathRight repeats a set of rules that will move the End tile either up, down or to the right. These steps will be repeated until none of these rules can be executed anymore, which would mean, when it reaches one of the corner tiles in the right of the level.

This process required a lot of fine-tuning, with each step requiring the generator to remember which direction the path is supposed to go. To do this, I created a system of tiles that signify in which direction each tile is supposed to go.

**The pathing system**

The system works as follows, a tile that simply goes straight in a horizontal way is labeled as “1H” or “Straight Horizontal”. A tile that goes straight in a vertical way is labeled as “1V” or “Straight Vertical”. While this seems simple enough, the system is mostly centered around way that the corner pieces are defined. Each “Corner” tile is defined by three letters. The first letter indicating the direction from which the player arrives into the tile, with “L” for left, “D” for down, et cetera.

The next letter would be “C” to indicate the tile is marked as a corner piece. Finally, the third letter indicates the direction in which the players will exit the tile, using the same kind of indications as the first letter. For example, the corner tile “LCD” would stand for “Left-Corner-Down”, meaning that players would enter the tile from the left side, follow the corner and exit the tile again by going down. By figuring this out, I can already predict that the first letter of the next tile would be “U”, since players would enter the next tile in the path from the top of the tile.

Note that I use “D” for down and not “B” for bottom; while some might find this to sound better for orientation, it would also mean that I would have to replace “U” for up with “C” for ceiling, resulting in corner pieces being called “LCC” or “RCC”. I found that using two letters with different meanings in this system to be impractical and slightly confusing at times, making it harder for me to keep track of what direction the path was moving in.

Developing this system gave me a clear overview of how the final path would run through the level. It also allowed me to create specific templates that fit the orientation of specific corner pieces. Spending time to create the pathing system in this way showed more promise than my previous attempts. One of which consisted of a much more simple approach that only contained “Horizontal”, “Vertical” and “Corner” tiles. The idea behind this earlier system was to simply use these three types of tiles, and have them mirrored or rotated when needed to fit the path appropriately. This quickly turned out to be impractical as it resulted in rotated templates not corresponding with templates adjacent to it, a problem that was most prominent with corner tiles that were right next to each other. With one corner tile being up-side down and the other being rotated ninety degrees, players were unable to progress any further due to the fact that these two templates were not designed to fit together in such a way.
The “path=” variables

The pathing process is divided into two separate modules, “CreatePathLeft” and “CreatePathRight”. This was done so that the level would always be a minimum of seven tile, ensuring that the levels can be long enough to implement various kinds of patterns. However, even a path this short would only occur in the rare situation that CreatePathRight generates a path that is spawned along the width of the level and stops in one of the corners there, followed by CreatePathLeft not moving the End tile during the eight iterations.

While executing the CreatePathRight module, each tile that is added to the path gets the variable path=right assigned to it, indicating that the tile belongs to this module.

Besides the path= variable, a second variable called “finalPath” is always added to the latest tile that is added to the path (not the End tile itself). This tile serves purpose that is similar to that of the “firstPath” from chapter 5.3.1, indicating the orientation of the final End segment in the level so it can properly connect to the generated path.

After this module has been executed the next module, “CreatePathLeft”, is executed. CreatePathLeft, as the name implies, does the opposite of what CreatePathRight does, building a path going towards the left side of the level. The difference between the two is that CreatePathLeft has a maximum of eight iterations, meaning that it can only iterate its ruleset eight times before stopping. This means that even when there is still unused “space”, the path can come up shorter than that, causing the path to the left to be of varying lengths.

Each tile that is added in this module also gets the path=left variable assigned to it, indicating that the tile is part of the CreatePathLeft module. The reason that the path= variable is so important, is because it makes it so that during these iterations, the generator can discern which corners belonged to which module, which prevents it from accidentally creating tiles that attach onto tiles from the previous path. For this prototype, simply the path=left variable would have been enough, but adding the functionality of the path=right variable as well would allow us to potentially switch these modules around, allowing the generator to also do this when the level is generated right to left.
5.3.3 Split And Border

The “SplitAndBorder” module is fairly straightforward. It takes the current layout of the level and magnifies every single tile into a large “segment” made up of 20x20 tiles of that type. This step in the process gives the first look at what the final level will look like, although without any clear definition besides how the path runs through the level.

I chose specifically to magnify the single tiles into 20x20 versions because I found that 20x20 segments gave me sufficient space to design templates for the following step without them feeling too massive.

Should a designer want their segments to be bigger or smaller for any reason whatsoever, it is important that this is done before or during this step in the development process. This is due to the fact that during the next step, rules will be made based on the specific sizes of the segments and changing the sizes of the segments would also result in having to adjust every single rule made during this next step.

With the level being magnified to this new scale, I used a simple “Border” command to add a ring of special “Wall_Border” tiles around the level. These tiles cannot be interacted with in any way and only serve to keep the players within the boundary of the level.

Figure 10. Expanding the level with the SplitAndBorder module.
5.3.4 Fill Templates
At this point in the generation process I have a layout of the final level with indications on what kind of rooms should be generated and where these should be placed. In the “Fill Templates” module I start filling in the large placeholder segments, using rules and their corresponding templates to generate actual rooms that are fit for gameplay.

These templates will contain tiles such as walls and Stationary Hazards, as well as implementing so called “encounters”. The encounter tiles within the generator serve as a probabilistic element in the generation process, being able to create certain game design patterns in various ways. The conversion of these encounters into actual tiles is done in the “FillPatterns” modules, which are covered in chapter 5.3.5.

Below I will describe the different kinds of rules that were made for this module, what their function is and what it is required to get these rules to consistently produce level geometry in a correct and logical way.

FillStart / FillEnd
The “FillStart” rule is applied to the now 20x20 “Start” segment. For FillStart, there are three different variations of the rule, representing the possible orientations of either up, down or horizontal.

Right now the FillStartHorizontal rules is only applied while facing towards the right, as the current generator always has the Start segment on the left side of the level. However, since the horizontal room can simply be mirrored if needed, the FillStartHorizontal room can also be applied if the Start segment were on the right side of the level.

The FillStartUp and FillStartDown have however been divided into two separate rules. This is done because a room that is designed to be traversed by descending cannot always simply be mirrored to create a room fit for ascending. To create single rule where this is possible would require more time and the designing of specific segment layouts. Therefore I have decided to split the two rules, allowing for much more freedom in the different kinds of ways a segment can be created for either ascending or descending gameplay.

To ensure that orientation of the Start segment is always in the right direction, I use the firstPath variable that was assigned back in the CreateStart module. By adding this variable as a requirement within the rule, we can always be assured that the Start segment always connects to the next segment of the path, no matter how many or what kind of segments might surround it.

Creating the “FillEnd” rules is done in a similar way as the FillStart rules. This rule is applied to the 20x20 “End” tile segment and again we have three varying orientations: up, down and horizontal. This time, if the “FillEndHorizontal” rule is applied, it can be mirrored if need be as in the case of the End segment, it could face either left or right. The rules for up and down are split again for the same reasons mentioned above regarding the FillStart rule. In place of the firstPath variable used by FillStart, the FillEnd rules uses the “finalPath” variable, which will indicate what segment was the last to be generated in the path of the level. This ensures that the End segment will always be connected to the right segment of the path, regardless of how many other kinds of segments might surround it.
**FillStraight**

For each variation of the FillStraight rule, the segment is required to be a 20x20 border of StraightHorizontal (1H) or StraightVertical (1V) tiles with an extra line of tiles to be added on whichever side the segment is oriented towards. This border acts as a means to prevent two same segments that are adjacent to each other from overlapping.

In contrast to the FillStart rule, the FillStraight has four different variations of the rule for each direction being upwards, down, left or right. I cannot make “Horizontal” variation as I did before due to the fact that some of the segments within this rule employ design patterns like gates and levers, which require specific placement; the lever always being placed before the gate, for example.

To fix this issue for the FillStraightLeft and FillStraightRight rules, I created a `path` variable that can be either “left” or “right”. This variable indicates the direction in which the player will be moving through the room, allowing the generator to recognize whether a FillStraight rule should be applied either towards the left or the right.

For the FillStraightUp and FillStraightDown I initially tried a different approach. Instead of using a variable for defining in which way the player would be travelling I tried to have the generator indicate which way a room would be oriented based on the tiles that were adjacent to it. For example, I can deduce that a StraightVertical segment is facing downwards if the bottom of the segment is not connected to a “LeftCornerUp” (LCU) or “RightCornerUp” (RCU) segment.

However, this idea did not turn out the way I wanted it to due to the way the generation system works. When the generator started processing the tiles, it would already process the corner tile below it in such a way that it would negate the previously created rule of not being connected to LCU or RCU tiles due to the fact that all these tiles had already been replaced. The result would be that the StraightVertical would randomly be oriented either upwards or downwards. This lead me to applying the same approach I used for the StraightHorizontal rules, creating a new variable `climbing` that indicated whether the player would be climbing up or down through the segment.

![Figure 11. An example of FillStraightVerticalUp.](image)
Finally, we have the FillCorner rules. I have a total of eight rules for all these corners, each pertaining to its own orientation. Since I made the decision of declaring these corners in the previous module, I only need to fill out the rules with different templates that can be implemented. I only implemented the failsafe that the corner cannot be adjacent to any borderWalls.

For example, an LCU (LeftCornerUp) tile only needs to be filled with segments containing platforms that allow the player to travel upwards. The only thing to be wary of is to place the platforms in the corner in such a way that do not block the entrance or exit of whatever segment it might connect to. To do this, I make sure to not place any platforms near the edges of a segment (save the single one on the inside of the corner, as seen in figure X)
5.3.5 Fill Encounters

The previous module in the generation process creates the basic shape of my level, it indicates where
the walls and obstacles might be but also the placement of tiles I have called “Encounters”. Below I
will explain what these encounters are and what kind of role they serve within the generation
process.

Encounter types
An “Encounter” is a generic term for a single or a group of game design patterns that have similar
effects on the player experience when encountered. I decided to take this approach, as opposed to
just making a rule for each pattern separately, because it allows the generator to choose from
different kinds of game design patterns from each encounter, allowing for more possibilities on what
the final encounter might look like, but still have a similar effect on the player.

Using the design patterns which were previously established in chapter four, I looked at which
properties each pattern had and came up with the following types of encounters:

JumpEncounter
The JumpEncounter entails all design patterns that influences the way the player has to move
through the level. This could be due to a trampoline, a moving platform or an elevator to name a
few.

DangerEncounter
The DangerEncounters contain patterns that pose a threat to the player and can hurt them if not
approached correctly. DangerEncounters can come in the form of simple stationary spikes that the
players have to jump over, a breaking block that can make the player fall down when standing on it
for too long, or hazards that activate and deactivate at intervals.

Figure 14. An example of a possible JumpEncounter, note the DangerEncounter(dEnc)
implemented which will in turn also be processed when the module is executed.

LeverEncounter
I had the most trouble to find a way to implement the LeverEncounter. At first, I had wanted to do
this for both lever and gates so that Ludoscope could procedurally create these encounters.
However, it proved to be a lot of trouble to place gates in such a way so that it would always work
with whatever kind of segment had been generated before. Because of this I decided to have the
gates be generated in the previous FillTemplates module and to have the levers be generated in the
FillEncounters module. Generated levers can be either timed or normal. A timed lever, as the name
might suggest, has the lever set off a timer when it is activated to open a gate. When this timer runs
out, the lever deactivates again, closing the gate.
CheckpointEncounter

Finally, the CheckpointEncounter has a fifty percent chance of placing a checkpoint object tile. These checkpoint tiles will act as a location where players can respawn if they manage to reach it and activate the checkpoint. I have given this tile a fifty percent chance of being generated because I felt that having it as a certainty took away too much challenge and risk when playing the game.

The FillEncounters module

With the different types of encounters established, I can move on to implementing these into the FillEncounters module itself. I have created a set of rules that would check each encounter and fill in the respective patterns belong to that encounter type. Along with this I created a “startPattern” variable which ensures that when processing the encounters, the segment is orientated in the right direction in a similar way as done before with the path= variables.

To add an extra layer of procedural generation, it also possible for some encounters to generate smaller encounters of a different type when they are processed. For example, a DangerEncounter always places something like a spike or an enemy, but it might also place a new JumpEncounter along with it. Due to the way the rules are executed, the module can then also apply the JumpEncounter, creating something like a moving platform or a trampoline along with it.

By implementing encounters and combining them in such a way, I allow more variety for Ludoscope to create more possible gameplay situations.

5.3.6 Fill Stationary Hazard (sHaz)

The “Fill_sHaz” module places “Spike” tiles on the locations of the “Stationary Hazard” (sHaz) tiles and corrects the orientation of these tiles if needed. Right now the obstacles within the level come in the form of these Spike tiles that are placed within the segment according to whatever rule is applied. However, situations might arise in which a Spike might be facing it’s pointy end against a wall. While this does not influence the function of the Spike itself (which only utilizes a box collider for collision detection), it does seem a bit odd for the players if they see themselves get damaged by simply hitting the side of a spike.

Figure 15. Executing the fillEncounters module.
First, the module replaces each Stationary Hazard tile with a Spike tile that is either facing towards the right or upwards, depending on which side there is an “Air” tile. Next I made a rule that can flip a Spike in the opposite direction if the spike is facing something that is not an Air tile.

Correcting the rotation of the spikes is not essential within the generation process, but I have made the choice of doing this in Ludoscope instead of having to do this within the parser in Unity later on. This is because that, in my opinion, it would take more time doing this in Unity as it would require me to assign a new “orientation” variable to all the spike tiles and have the parser subsequently be able to read and parse this variable as well.

5.3.7 Fill Empty Templates
This module serves to fill any of the empty spaces that are left within the level. This is done by replacing every single undefined tile with a normal wall block. This module is mostly aesthetical as I found it gave the level a feeling of actually being solid geometry instead of a series of lines running through each other.

5.3.8 Make Unity-compatible
The “Make Unity-compatible” module is the final step in the generation process. Each level that is generated by Ludoscope can be exported into a simple text (.txt) file, which we would then import and parse through a script in Unity. However, these text files not only contain the names of each individual tile, but also the names and values of variables that are assigned to the tile. This means that right now I have a “finished” level, but due to all the assigned variables like “finalPath” and such still being around, it makes it more complicated to build a working parser that can catch and separate all these types of variables when reading the text file.

To make this process more straightforward, this module serves to remove any obsolete variables that have been assigned. Right now, this means every single variable within the level, except for the “leverID” variables, which act as the variable that ties all the gates and corresponding levers within the level.

By removing all of these obsolete variables, we are left with a simple, cleaned-up layout of the final level, which I can now easily export to Unity for parsing.

5.3.9 The end result
Finally, I have added a “ViewResult” module which lets me view the final version of the level. From this module, I save the level as an Expression(.xpr) file, which contains all the data I need to visualize it in Unity.

Figure 16. The final level, ready to be exported as a text file.
6. Developing the proof of concept

6.1 Choosing an engine

For the development of the gameplay prototype, my preference went out to whichever engine suits the needs of this project. Engines that supported rapid prototyping were also preferred since this would most likely be the method of development for the duration of this research. Usage of either C# or Javascript was also preferred as those are the programming languages in which I had the most proficiency. On top of this, I also looked for specific engines that have been used for several games already released today. The reason for this was to support the argument that the final proof of concept was made with an engine that could be considered an accepted standard by the game development industry.

Due to time constraints I have limited my search to two engines that I have had previous experience with. This search brought me to following possible game engines that could be used:

**Unity** - Open source game engine developed by Unity Technologies[17]

**Unreal 4** – Open source game engine developed by Epic Games[18]

After trying both engines and observing the possibilities of each, the choice was made to use the Unity engine. The engine was chosen due to having more prior experience with the engine, especially on the subject of developing puzzle-platformers. Besides this, Unity also boasts a large amount of tutorials for creating games, as well as a sizeable community that can offer support when a developer gets stuck on something.

Reasoning behind this was that the Unity engine is used much more frequently for puzzle-platform games compared to Unreal. Another reason is that I personally have more working experience with the Unity engine, as well as already knowing a number of possibilities on how to transition Ludoscope levels into Unity.
6.1 Setting up core gameplay
The prototype required certain functionalities before being able to begin testing the parser for Ludoscope files, all of which are stated below.

Physics Platformer Kit
The decision was made to purchase the PPK [19] as a means to reduce development time that could better be spent working on the generation system and the parser. The PPK contains most of the basics that were required for this proof of concept, such as a controllable player that could run, jump and be affected by physics. It also contained some things that showed so much potential that they were added as game mechanics within the proof of concept, such as the trampoline or elevator.

Implementation of gameplay mechanics
Now that I have started work on the actual prototype, I am able to implement the mechanics that were established in chapter three into the game and see how this works. With the help of the PPK, it was fairly easy to set up a player character that could run, jump and interact with objects. Although the PPK was designed for single-player game development, I was able to make some changes to the movement and interaction scripts that allowed for a second player object to be spawned. This second player has the exact same features as the first, but with different controls, which now allows to play the game with a second player.

Camera movement between two players
To put more emphasis the idea collaborative play, the camera was adjusted so it would stay in the middle in between the two players. This solution was chosen as opposed to split-screen as it would allow for either player to go run off on their own. The fact that both players share the same camera encourages them to stick together, because if they move too far apart, neither player will be able to properly observe their surroundings.

6.2 Creating the parser
To bridge the gap between Ludoscope and Unity, a parser is required that can translate the text files saved from Ludoscope into commands and cases that can be used by Unity. Below you will find an explanation on how this was achieved.

Parsing Ludoscope tiles
Since Ludoscope files are saved as a simple text file, getting Unity to read it was not very difficult to achieve. Cases were made for each type of tile from Ludoscope to be recognized in Unity, allowing for the engine to generate an object based on the tiletype that the parser would read. For example, when the parser parses a “borderWall” tile from the text file, it instantiates a borderWall game object within Unity.

Placement of objects in the game space
Bringing the parsed tiles into the game space was done by creating a 2D array that read out the rows and columns from the Ludoscope files and translated this into coordinates within the game space. In combination with the case system from before, the parser is able to spawn a specific tiletype on set coordinates according to the text file from Ludoscope. These tiles are spawned inside of Unity with the help of so called “prefabs” which are game objects with set components and values that can be instantiated into the game as many times as needed. This resulted in a hundred percent accurate 3D representation of the Ludoscope level in Unity.
Important to note is that I found the collision detection in Unity to be less than ideal for simply instantiating each game object with its own collider as this resulted in the player constantly performing collision checks when moving which resulted in player movement being seen as “jittery” and allowing the players to climb up alongside walls by tapping the spacebar.

Since this problem was centered around the placement of the walls. This problem was solved by having the parser check from top to bottom how many walls were found in a column and then instantiating one wall with a single collider as a result, as opposed to several smaller pieces with their own colliders.

**Linking object ID’s from Ludoscope in Unity**

Besides simply importing and instantiating certain tiletypes, some tiles also required certain values to be taken over by Unity. An example of this is the gate / lever mechanic that is implemented, which requires a gate to be matched to a lever through the use of a unique integer. These values must be carried through the parser and interpreted correctly by Unity.

Each lever and gate that is instantiated contains a unique ID that ties them together. The parser interprets this and matches the corresponding gates and levers so the right ones can interact with each other. This allows for the levers and gates to retain their appropriate connections from Ludoscope.
7. Conclusion

By the end of this project, three possible forms of multiplayer interaction had been defined and the decision was made to settle on a collaborative approach to gameplay, forcing two players to work together to complete a level. Next, a set of game mechanics was designed that could be implemented into the proof of concept. Game design patterns were also created, describing certain scenarios the players would encounter and the consequences of these scenarios. These patterns were then implemented into the generator through the use of “encounters”, allowing the generator to assign certain areas of the level space to be used for implementation of these design patterns inside the level generator.

I have managed to create a functional level generator within Ludoscope which randomly generates paths, patterns and encounters to determine the layout of a level. Employing the use of game design patterns, I am able to create “encounter” zones wherein the generator can implement design patterns in varying ways. Coupling this with the randomization of the segments, it allows for the generation of levels where two identical levels are nearly impossible to create.

Through Unity it was then possible to parse the finished Ludoscope level into a game scene and correctly render a game space complete with objects and mechanics in which two players are able to interact with and finish the level.

Based on the results and conclusion given in the previous chapter, I can determine that I have succeeded in achieving the results that I wanted when I started this project. A functional level generator was created that can generate puzzle-platform levels, which can then be imported into Unity and parsed into a fully functioning game space, all in the span of a few minutes. This results in proof that this method of generation can be used to reduce development time spent on creating levels by hand.

In the end, results have shown a lot of the benefits of procedural level generation. When able to set up a system that employs the required mechanics and properties for your level, if done correctly, makes it easier to create levels suited for gameplay as opposed to having to make these levels completely by hand.

An important note is that the efficiency of such a system is also closely related to the quality of the generation system. Things like an organized alphabet and a clearly defined rule set determine the amount of work the generator can do by itself. However, if done correctly, this method of level generation allows for the creation of many diverse levels in a short time, while retaining a sense of progress and flow through the use of the design patterns. Besides this, the mixed-initiative approach also allows for a certain degree of freedom for designers to apply their own changes by hand if certain aspects of the level are not deemed satisfactory.
8. Future work

Although this project showed positive results, there is also enough room for improvement. Stated below are some suggestions on what could further be added to this research in the future.

Measuring the flow

Currently, the proof of concept executes the fundamental functionalities that are required to generate a level for collaborative gameplay. However, there is not really a sense of flow within the level. Flow is measured by values of pacing and challenge of a level, none of which is being done in the current version of the level generator. The generator picks one template out of a set number of designs and goes on to implement more patterns within, but there are no further rules or requirements for the generator on when to implement certain templates or patterns.

Therefore it seems like an interesting idea to look into the possibility of keeping track of the level’s flow or, for ease of measuring, the sense of difficulty. This would result in an increase of data and feedback that designers could obtain from using the generator. It would allow designers to observe at what parts of the level players should be feeling more relaxed or more tense and adjust these values to what they wish them to be.

To make this possible, I would suggest looking into the possible application of something of a heatmap, which tracks the challenge value of certain rooms or segments of a level. Based on this, the generator could detect these values and decide on how much the challenge level would be increased or decreased in the next room.

Possibility of application in different genres of videogames

While the current research was focused on puzzle-adventure genre, there are many more genres within the world of videogames. As such, there are a lot of questions about whether or not this method of level design is applicable towards other genres, like the MOBA or FPS genre.

For example, procedural placement of different types of enemies within an FPS could change the way players would play the game. Instead of busting into a room expecting four armed enemies, they could encounter a room filled with eight of them, two being tougher than a normal enemy. It could change the way players decide to play the game, if the enemies are randomized, players could instead shift their focus on the environment and where they can best place themselves to face whatever encounter the game could throw their way.

Or imagine a world in which you play a MOBA that generates a new level for each match you play. Where there could be two lanes, or three, with various points of interest scattered around the map both strategically and fairly for each of the teams.

These are only some examples of ways that this method of level generation might be used in different areas of game design, but it holds a lot of promise and might be worth taking a look at.
Using the current method of generation for full 3D worlds

Even though Ludoscope is a tool that generates level within a 2D space, the resulting levels can still be implemented for 3D environments[15]. While the current version of the prototype shows promise in generating levels for 2D gameplay, it would be interesting to see how this method of level generation could be applied to games that employ a full 3D environment. This would allow for the generator to become even more dynamic in size, scale and also opening up the possibility of applying this method of generation to more modern video games.

A possible way of developing this would be to create a new template designer for Ludoscope, featuring a 3D viewport as the one seen in modelling programs such as 3ds Max. Editing a template would be more similar to 3ds Max as well, clicking and dropping “tile types” on a 3x3 grid, drag-selection and easier ways of placing multiple blocks in certain spaces. The rest of the module system in Ludoscope does not have to be altered, simply the way the templates and levels are shown within Ludoscope.
References


Appendix A, Phasing of the project

To better organize and keep track of progress, the project has been divided up into the following phases:

Literature studies and establishing the research question
The first part of this paper will contain information regarding the establishment of a research question, as well as the sub questions used to provide a logical conclusion. During this phase I will also search for relevant papers and other literature regarding the subject of PCG and multiplayer interaction. When finished, this part forms the base of the rest of this research.

Development of the generating system
The first phase will see the creation of the procedural level generator in Ludoscope. The main idea behind this is to come up with a system that can reliably create playable and challenging levels for the players. For this be accomplished, different types of tiles, gameplay patterns and rules were designed that the generator can make use of for the generation of said levels.

This would result in a basic workflow being established on how to set up a random level generator of this kind within Ludoscope. Certain design decisions and the overall process of coming up with this system is described in detail, explaining why certain rules are necessary for the generator function as it does. Ideally, the final version of the generator would only require the user to press “generate” and export the level’s text file to the Unity engine.

Integration in Unity
After finishing the level generation system, development is started on a parser which utilizes the Unity engine. This parser translates the text output of the Ludoscope files into commands that the engine can read and carry out. Using a 2D array, the parser then places all tiles from the Ludoscope file accordingly into a 3D environment. This applies to anything generated in Ludoscope, ranging from simple walls that define level geometry, to the placement of players, obstacles and enemies.

Besides placement of the geometry and game objects, the parser must also allocate certain values critical to the gameplay. An example would be the randomly generated lock and key mechanism; at a certain point in a level, players are confronted with a door that requires a key to unlock. However, each key in the level is unique, requiring both keys and locks to have a similar ID assigned to it, otherwise any key found in the level would unlock any lock found by the players. This too is handled by the parser, which receives values from Ludoscope that assign ID’s to things such as the key / lock systems.

Furthermore, a certain amount of game mechanics and functionality is added to the game. Using the Physics Platformer Kit helps in saving time on development which can instead be used to focus on the research aspect of this project. Only slight tweaks are made to the mechanics of the basic PPK. Changes are made to incorporate two players and a camera script is made to track both players on screen at the same time.

When finished, the Unity game will be able to parse a set of text files from Ludoscope into playable and functional levels, allowing players to go straight to playing the game. This should also prove the possibility of applying Ludoscope to one of the more well-known videogame engines within the industry.
Appendix B, Game Design Patterns

This document contains a list of all the game design patterns implemented in the prototype, as including related terminology that describes the impact of said patterns on gameplay.

The format of describing these game design patterns is based on the works of both Hullett & Whitehead [1] as well as Reuter et al [2]. This format describes a gameplay situation, the affordances that said situation might have, and finally the consequences that the pattern will have on general gameplay.

1. Terminology

Below is a list of terms used to describe certain values in gameplay. These values may increase or decrease based on the pattern that is applied.

**Pacing**
According to Davies [1], pacing raises or lowers the tension, tempo or challenge of the level for the player, this is usually summed up as the “flow”

**Challenge**
Determines the level of difficulty the player might have. According to Malone[2], challenge is defined by the fact that it always has an uncertain outcome, stating that there is never a certainty that the player will either win or fail. Attention must be paid to not make the goal easy to attain, as it will become boring. Nor can it be made too hard or unclear, or it will become frustrating for the player. Challenge can be influenced by creating different levels of difficulty for certain segments of a level.

**Timing**
Describes the amount of attention the player must pay to the timing of using certain mechanics. Amount of timing required will increase depending on the amount of coordination that is needed for a player to successfully go through a pattern.

**Coordination**
The term used to present a measureable value of the amount of coordination between two players who work alongside each other. An increase in coordination would mean players need to pay more attention to the actions of their teammate as well as requiring an increase in communication between them.

**Example**: timed gate/lever system would require an increase in coordination to complete, as one player would need to be positioned at the lever and the other at the gate, as well as timing the lever with going through the gate.

Closely related to timing but differs through the fact that coordination is focused on collaborative aspects of the game, as opposed to timing, which can also apply to a single player.

**Competence**
The sense of reward a player has after completing a certain action or interacting with a certain mechanic. Competence is used to reward players for completing certain actions or interacting with certain patterns.

An example would be passing an enemy encounter by killing the enemy, which in turn awards you with an increase to your score.
2. Design patterns

**Grounded enemy encounter** (SP)
Can be destroyed by jumping on top of the enemy, can be both a single-player as a multi-player encounter.

**Affordances:**
- Amount of enemies placed
- Health amount of enemy
- Damage done by enemy

**Consequences:**
Increases the challenge by adding an obstacle that can either be passed or destroyed in exchange for an increase in score.

**Environmental hazard** (SP)
**Description:**
The stationary hazard serves to slow down the pacing by placing an object in the level that hurts the player if touched. To pass the obstacle, a player must have the skill to combine timing with

**Affordances:**
- Size of the hazard
- Amount of hazards placed
- Type of hazard (stationary, moving or timed move)

**Consequences:**
The environmental hazard serves to slow down the pacing of the game by making the player pay extra attention to their movements. It also increases the challenge due to the fact that the player will be hurt if he/she fails and hits the hazard.

**Environmental hazard (moving)** (SP)
**Description:**
A variation on the environmental hazard that moves horizontally along the level.

**Affordances:**
- Movement speed of hazard
- Amount of hazards placed

**Consequences:**
Same as the normal environmental hazard, but results in a further increase of challenge and timing due to the fact that the player must now also be careful with the movement of the hazard.
Environmental hazard (timed) (SP)
Description:
A variation on the environmental hazard that is only lethal on certain timed intervals
Affordances:
- Amount of time for intervals of hazard
- Amount of hazards placed
Consequences:
Same as the normal environmental hazard, but results in a further increase of challenge and timing due to the fact that the player must time their moving with the interval wherein the hazard is not lethal.

Elevator sequence (SP)
Description:
A moving elevator that the player can grab onto.
Affordances:
- Speed of the elevator
Consequences:
Serves to slow down the pacing of the game as the player is only required to hold onto the elevator while it goes up / down and let go at the appropriate time.

Trampoline sequence (SP)
Description:
An object that can launch the player in the opposite direction when hit.
Affordances:
- Placement of trampoline
Consequences:
Slight increase in challenge as the player must try and steer themselves in the right direction after being launched.

The upsy-daisy (COOP)
Description:
An obstacle that requires two players to time their jumps together to get onto a platform. After this, the player can push an object down to the other player so they can both get up and proceed further through the level.
Affordances:
- Obstacles near the platform that needs to be reached
- Distance of platform above the ground.
Consequences:
Causes a sharp increase in challenge and coordination for the players. Both players must time their jumps together so that one of the players can use the other player's head to get high enough to reach the platform.
**Lever / gate (COOP)**
Description:
A segment that requires one player to operate the lever so the other player can go through the gate.

Affordances:
- Placement of lever
- Placement of gate

Consequences:
Possible decrease in pacing while players are attempting to find and reach the lever.

**Timed lever / gate (SP + COOP)**
Description:
A lever that, when pulled, opens the related gate and starts a timer. When the timer runs out, the lever is reset and the gate closes.

Affordances:
- Amount of time before lever resets
- Amount of obstacles between the lever and the gate

Consequences:
An increase in challenge and also pacing because of the fact that the player must now hurry to get through the gate before the timer runs out.

**Key / lock (SP+COOP)**
Description:
An object that requires a key to go through.

Affordances:
- Placement of lock
- Placement of key

Consequences:
A decrease in pacing and a possible increase of challenge, dependant on how many obstacles are placed between the player and the key object.

**Checkpoint (SP)**
Description:
Reaching and touching the checkpoint allows for the players to have a place to respawn in case one of the players dies somewhere along the level.

Affordances:
- Amount of checkpoints
- Difficulty of reaching checkpoint

Consequences:
The tension is lowered as the player will have just reached a “safe zone” ensuring that they will not have to re-do a large part of the level.
Breaking blocks (SP)
Description:
Blocks in the level that will disappear after having been touched. Respawns after a set amount of seconds.
Affordances:
- Amount of blocks placed
- Delay of blocks before breaking
Consequences:
Causes an increase in pacing and challenge as the player must continually move forward to avoid falling down after the blocks have disappeared.

3. Sources
Games cited
Battleblock Theater, The Behemoth.
Super Meat Boy, Edmund McMillen.

Literature
Design pattern terms
Referencing Hullet, Whitehead 2010, P3.