TÉCNICO
LISBOA

Procedural Content Generation for Cooperative Games

Rafael Vilela Pinheiro de Passos Ramos

Thesis to obtain the Master of Science Degree in

Bologna Master Degree in Information Systems and Computer Engineering

Supervisors: Prof. Rui Felipe Fernandes Prada

Prof. Carlos António Roque Martinho

November 2015

Acknowledgments

I thank my supervisors Rui Prada and Carlos Martinho for guidance and insight provided throughout the year, when developing this work. I would also like to thank fellow student Miguel Santos for his invaluable assistance during testing even when health wasn't helping. I would like thank Sofia Oliveira for her suggestions in experimental methodology. Lastly a big thanks to my parents, my family and all my friends who put up with me year long, listening and discussing whatever subject I threw at them.

Abstract

A popular topic now more than ever, procedural content generation is an interesting concept worked on several areas, including in game-making. Previously mainly as a mean to compact a lot of data generated content has been used to generate theoretically infinite worlds. Level and content generation ease the burden on game developers and artist. As more realism and bigger diversity is being required the need for efficient and fun level generators increases. On the other hand, a cooperative option on games is becoming more and more common, with a majority of games having a cooperative mode. A noticeable amount of research has been done on both of these areas and what makes them fun, however, little work has been done on a joint effort on both these fields. This work attempts to start filling the gap, linking together concepts worked on both cooperative games and procedural level generation and what makes them interesting. We look at some of the existing research on the separate areas, seeing what seems to work and in what ways can a generator be conceptualized. We then proceed to define what needs to be done for level generation to work on a cooperative game. We then propose a solution based on our findings for a cooperative platform game, Geometry Friends. Finally we discuss how could the system be improved based on test results and where should future work be aimed at on the subject of Procedural Level Generation for Cooperative Games.

Keywords: Procedural content generation, Cooperative play, Procedural level generation, Procedural generation of cooperative challenges.

Index

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Ever since mankind started making games there have been multi-player games. Pong, one of the first videogames was a two player game, running on massive computers, controlled by custom-made controllers. The idea that two, or more people can play the same game has been around for some time. However, at first videogames had a competitive nature. Players were pitted against each other, usually in a one on one scenario. Videogames like Joust [1] came around and people could finally play a videogame cooperation to reach a common goal.

Videogames have since adopted cooperation in multiple forms, such as teams competing over victory, multi-player puzzle solving and playing against the computer. There have been cooperative storytellers and games where the race to cross the finish line meant you had to work together to overcome obstacles, only to later betray your allies.

Games today have grown in size and replay value and so these requirements have produced a demand for automatizing the process of asset creation. The Borderlands series [2], with procedural weapon generation, The Elder Scrolls: Skyrim [3] with infinitely generated quests are two modern examples on how computed generated content can assist in enriching a game. To feed the beast that is this demand some game developers have resorted to using procedural content generation. Procedural content generation has existed in several fields. In game development it has been used to produce decorative assets, such as rocks, tree, buildings, even whole cities and even characters. It's also been used to produce functional parts of the game, for instance levels, weapons, enemies or even whole worlds, which include the aforementioned Borderlands and Skyrim. There has even been procedural generation in games that have and base themselves around cooperative play and cooperative mechanics. Only in these cases the generation is unrelated to cooperative challenge.

Procedural content generation for cooperative games can display itself in many forms as there is no strict definition to what it is. In this paper we decided to focus on the generation of levels that include cooperative challenges and so we use a particular definition for the term procedural content generation for cooperative games: procedurally generating cooperative challenges. A cooperative challenge is an event or obstacle in the game that requires coordinated effort of two or more players to be overcome.

While it is true that we can generate most of a cooperative game's content, a lot of it is no different to what you would normally create for non-cooperative games. Assets like trees, rocks and other props will probably be generated the same way in a single player game. Thus these are not the focus of the work. Instead we will focus on procedural cooperative level generation, the act of procedurally generating a game's level that includes cooperative challenges.

Most research made on procedurally level generation does not take cooperation into account and as of the date of this document's writing commercial cooperative games that have procedural generation do not generate any cooperative play. In fact, games that do advertise cooperative level generation do not generate cooperative challenges and, often, cooperation and procedural generated content do not intersect. Games like Cloudberry Kingdom [4] have procedural level generation but no cooperative challenges are generated. The first player to cross the finish line ends the level and other player's actions have no influence on each player. Gems caught by all players are accounted for, which would otherwise make the game more alike to a race to see which participant finished the level first.

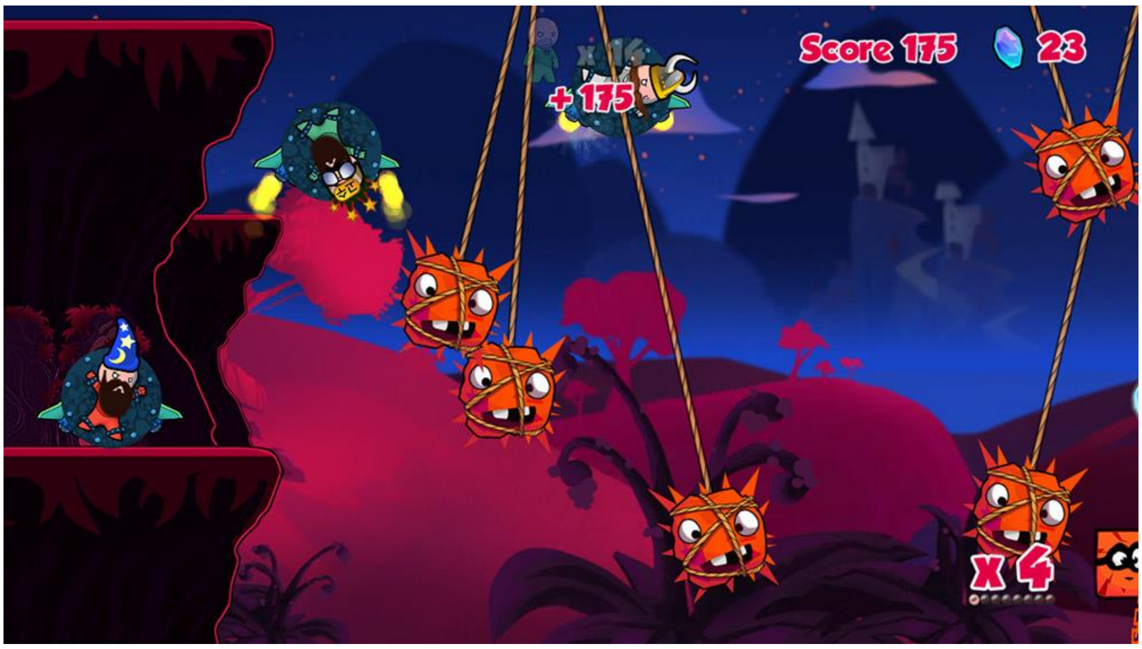


Figure 1. An example of level generation in Cloudberry Kingdom.

1.2 Problem Formulation

As we mentioned above, game development requires tools to automatize the process of developing content. Level generation generally worked on and procedural tools for level generation are now common place, even some games are based around them. We believe cooperative games would also benefit from having a tool that allowed levels to be generated automatically. Cooperative games like Portal 2 have editors to allow players to create their own levels. These levels can be marked as cooperative is the player designed the level to require cooperation. An automatic version of this editor however has yet to be created. Its existence would allow Motivated by the inexistence of a procedural level generation that takes into account cooperative play and the generalised lack of research into procedurally generating cooperative challenges, the focus of the work carried out is creating a procedural cooperative level generator.

1.3 Summary of Contributions

With this work we propose a general solution for the creation of cooperative level generators for games that does not require cooperative agents for validation, nor the need of a cooperative puzzle generator. This solution should serve as a guiding step to creating a valid level generator that includes cooperative challenges and is to be adapted to the game in questions, are levels are not all the same and much less cooperative challenge types.

We also provide an algorithm to produce cooperative levels for the game Geometry Friends, implemented as a tool, along with an editor for ease of adaptation of generated levels and information visualisation. This is accompanied an experiment in which we test the generated level's cooperative value to participants.

1.4 Outline

This document is arranged in several parts. We begin by exploring the background of procedural generation. We start by going over procedural generation in games and then proceed to go through the description and classification of some procedural generators. We then follow up with some background on cooperation in games. Afterwards we go through related work to this thesis, first exploring procedural content generation in cooperative games. We follow up with a more detailed look into recent work in procedural level generators which include using grammatical evolution to produce levels and using patterns derived from designer made levels as building blocks to generate levels. Afterwards we go into research made in cooperation in games. We explore cooperative design patterns common in cooperative games to assist game developers in designing games and cooperative performance metrics to evaluate cooperative games. This is followed up by a description of the testbed game Geometry Friends, a cooperative 2D platformer, and the tools we decided to use when developing our own procedural cooperative level generator. A description of the first approaches to making this procedural cooperative level generator follows, explaining encountered setbacks and further requests of simplification. A detailed description of the final version of the level generator follows, alongside the description of its interface and features. The next chapter is evaluation, where we explain how we designed the experiment and what results we had. Finally we have our conclusions and contributions.

1.5 Summary

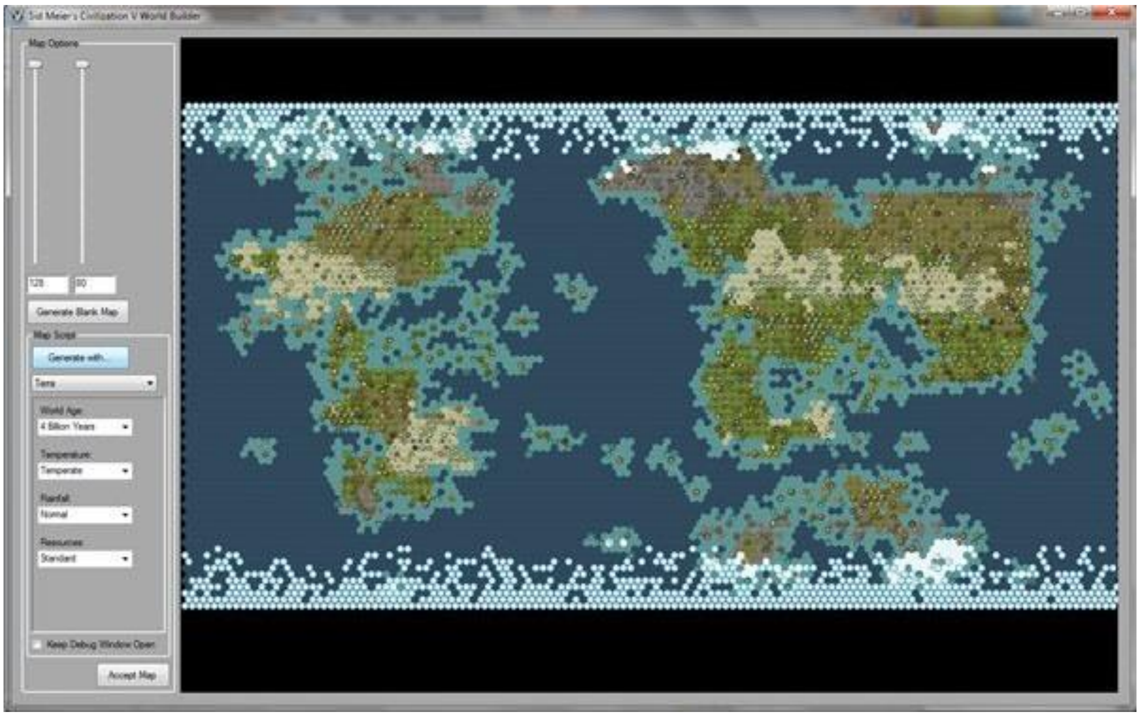
In this chapter we introduce our motivation and problem we are trying to solve. We present a quick look into what is available currently what and where procedural generation is used and what is a cooperative game. An insight into what is advertised as a procedural cooperative game in the current commercial environment is supplied which leads us to our problem formulation. Lastly we summarise expected contributions and outline the rest of this work.

2. Background

2.1 On Procedural Generation

Procedural content generation has come a long way since its humble beginnings. There used to be a time when saving disk space was a necessity we do not have today, things had to fit in a floppy disk or a memory card. A possible solution was not to have everything saved: Simply generate content on the fly from a less memory intensive algorithm. Using a pseudo-number generator and seeds developers devised a way to provide a lot of content to the end user, even if it wasn't all saved up. Early examples of this are the dungeons, monsters and items from Rogue (1983) [7] and the endless space in Elite (1984)[8].

As memory processing power and hard disk space became less of a problem another issue still remained: development cost of a game. Nowadays we also use procedural generation to speed up development. For example, the game Skyrim had its world's terrain and details on the countryside generated but villages, cities and important sites were still handcrafted [3]. This is not restricted to levels, in the same game smaller and non-story related quests would pop-up as the player travelled the land. These generated quests would relate to locations and people the player passed and gave the player the sensation that the world was alive while adding to the effect of giving the player a different experience when replaying the game. Other games, such as the indie games Minecraft[9], Dwarf Fortress[10] and Spelunky[11] use procedural generation to create an infinity of possible worlds or levels to play in, Dwarf fortress going as far as simulating weather patterns, geology, lore and history to create a believable and interesting world. As we can see, speed is not the only current use of procedural generation. It is being used as the main part of a game's selling point, adapting content to the player's playstyle as we can see with the generated quests in Skyrim. There are numerous applications to computed aided development and we believe it is safe to predict procedural generation will eventually be part of most, if not all, game development pipelines, provided complex results are required.



128
80
Terra
World Age: 4 Billion Years
Temperature: Temperate
Rainfall: Normal
Resources: Standard

Figure 2. A visualisation of the result from a world generator for Civilization V.

Different games require different things from a level generator. Minecraft and Spelunky require ever changing worlds with different resources, generated with no human interaction. Skyrim map requires realistic terrain features upon which content is to be added, and is closer to a virtual world generator. These virtual world generators are also used in more serious applications than games, such as disaster training and coastline inspection [12]. Competitive games such as Real Time Strategy or First Persons Shooters require fair maps where no side is disadvantaged [13][14]. And so, procedural generators must have restrictions based on what's needed for them. This calls for some form of classification of procedural content generators.

2.1.1 Describing a Procedural Content Generator

As noted by Shaker, N. in [15], you can define a procedural content generation technique from certain independent traits. These will be used to define what can and should be done for a cooperative version, so we will summarise the concepts here.

A procedural content generator can be online or offline. The first one runs at runtime, allowing generation of endless worlds or player adapted content. Offline procedural content generators are used outside gameplay and are used in generating complex content, a more resource and time consuming task. World Machine [16] is a good example of this, being a map generator which creates a highly detailed world which is to be exported into a more manageable height map for use in game.

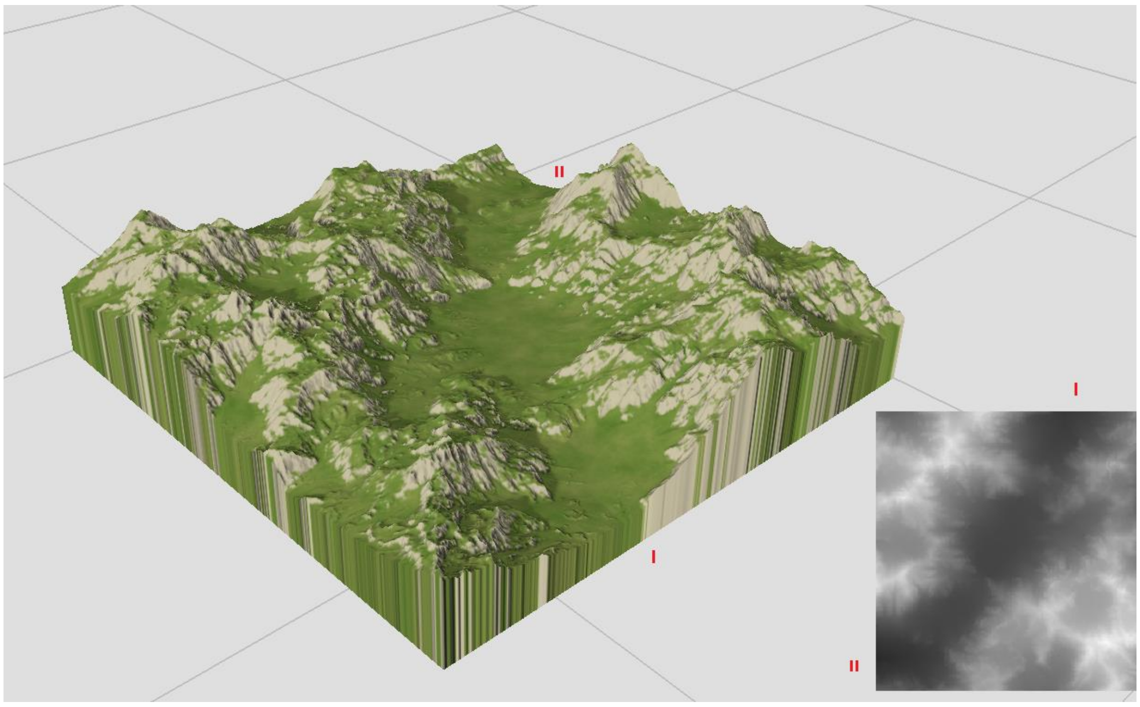


Figure 3. An image of generated terrain from height map. The height map is shown in black and white. The valley is labelled on both the terrain and the height map. This height map was procedurally generated.

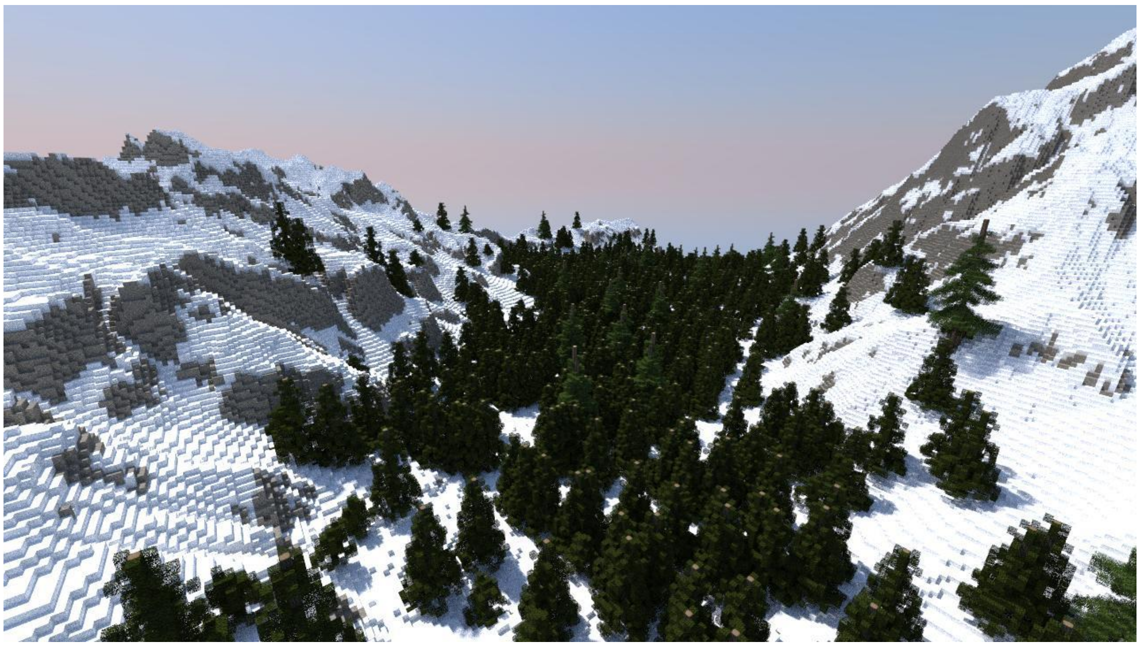


Figure 4. Previous terrain imported into Minecraft after adding some trees and biome specifications.

Procedural content generators can have necessary and optional content. They can be used to generate the level and objectives or/and they can be used to make disposable content. An example of this disposable content would be environment or bonus rooms in dungeons, through which the player is not required to pass through.

Random seeds allows a relatively unpredictable result be replicable provided the same seed is used, whereas parameter vectors provide a more controllable, objective oriented solution.

A generic procedural content generator takes no account for player's actions, contrasting an adaptive one, where content is created based on player behaviour analysis. This results in games that automatically adjust to players playstyle (as an example the game Galactic Arms Race [17]) or even "emotional intensity" (used by Michael Booth in The AI Systems of Left 4 Dead (2009) referring to the game Left4Dead [18] to describe current point in emotional flow in the game. He mentions a game should have ups and downs in activity, as to keep the player interested).

A procedural content generator can also be deterministic or stochastic. This contrasts on whether regeneration of content is possible, a stochastic generator never giving you the same exact same results, which might be just the feature one wants.

A procedural content generator can be constructive or generate-and-test. The constructive does not allow modifications when the content is generated and the process only occurs once. Generate-and-Test allows multiple generations until an acceptable solution is reached.

An algorithmic procedural content generator uses only its base parameters to generate content whereas a mixed-authorship may use human input to assist in generating the final content, such as drawing the world in tools as seen in SketchaWorld [12] and Worldmachine [16], or restricting and defining generation areas.

2.1.2 Generation Methods

A generator can be evolutionary, search-based [19] or declarative programming [20].

Evolutionary generators will create several levels and run them through some fitness function. The fittest levels will be used as parents of the next generation. Repeating this process a few times will create satisfactory results, so long as our fitness function is well defined. The issue is finding this function.

Search-based generators use brute force or heuristic guiding to find a valid solution. The validity of a solution is how it complies with a fitness function. Again, finding a proper fitness function is a challenge. Other complexities also come up such as how the algorithm should modify its current solution. The way this is made or implemented affects the overall efficiency and efficacy of the search-based algorithm.

Declarative programming generators use some sort of language or pre-fabricated code to define the level. This is then interpreted by the generator to create a valid level. This approach does not require a fitness function. It does, however, restrict the generation to what can be described and interpreted from whichever encoding the declarative approach uses. In a way, you are restricted by whatever you thought would be required when first making the encoding language.

2.1.3 A Quick Look Back

All in all what type of generation we should use is defined by what we want and need. Let us imagine a game where the player enters a level, a dungeon and gathers coins spread around rooms. Rooms will also have enemies and other threats such as traps. Each level has a boss room with a stronger opponent and a large amount of coins. There is a time constraint set on the player, at the end of which the number of coins gathered is the score. Finally a player can choose to proceed to a next level at any time.

Let us apply procedural content generation to this game. We can have levels be generated beforehand and add them to the game after a designer optimized them for maximum entertainment, but that will hurt the game's replay value. On the other hand we can generate new levels as the player enters them netting us a less refined experience but possibly always new and never-ending. This exemplifies the difference between offline and online generation. Our level generator could only generate necessary rooms or also generate optional hidden rooms, creating more variety.

The game we described is optional by nature, since you can leave a level at any time, which also prevents you being stuck in an unfavourable level. As such the level does not need to be entirely traversable, and optional non optimized content can be included although this might upset some possible players. If we decide to have an adaptive generator, our game could analyse how well the player is doing and scale up the difficulty of future levels, making the game more challenging. It could also detect an excess in difficulty, halting progress, and so lower the difficulty parameter. While on the subject of parameters, while the game we had on our minds was absolutely stochastic from the player's view, it could be deterministic from the developer point of view, parametrized values guiding generation. These parameters would likely mostly be hidden from the player but things like difficulty or level length could be initially provided.

As for generation method while it really depends on what we chose from the aforementioned characteristics we can pinpoint some strengths and weakness. Using declarative programming on an online version of the game's generator would split procedural generation into level design and level generation, the first part being alike writing a sentence in the level's grammar. An offline would strongly benefit from this, as designers could simply write the levels they wanted and the generator would take care of the rest, leaving only fine tuning to be done. An evolutionary generator will likely void the possibility of having a strictly necessary level generation approach without online infrastructure to share properly evolved results. It would also lead to the existence of bad levels, although these could eventually become rare, depending on the fitness function's quality, and possibly leading to a better player experience. For a search-based approach the development team would most likely lead to either speedy, simple and linear results or slow and complex levels. For the second we would probably want to have it running as an offline generator.

2.2 Cooperative games

Cooperative games have existed for quite some time now. There are several examples outside videogames: in sport, such as volleyball or tennis, and in board games such as Lord of the Rings [21] or Pandemic [22]. In video games examples of this mechanic are Gauntlet [23] or more recently Valve's Left4Dead [18] and Portal [24].

What is a cooperative game? It is a game where players cooperate to reach their goals, in contrast to competitive games. While both can co-exist, generally a game can be either competitive, cooperative or a mix of both. Competitive games, such as Starcraft [25] or Counter Strike [26] pitch players against each other. This last one can also be part of a mixed cooperative/competitive game since players are divided into two teams. In this type of game, players on each team cooperate with each other in order to best players on the opposing team. Another example of a mixed game would be League of Legends [27] or any game in the Battlefield series [28]. As for cooperative games, players work together to solve a puzzle or to defeat an AI opponent. Portal 2 and Left4Dead are good examples of this, much like Magicka [29] and Borderlands [2].

In pure cooperative games players face either computer controlled enemies or try to solve puzzles. To force cooperation games usually make players have the same or shared goals [44]. These mean that players have to work together in order to progress. To avoid forcing players through objectives some other strategies can be employed such as making abilities synergize with other player's classes or enforce different rules when interacting with allied players. We will further explore cooperation in a later chapter.

2.3 Summary

In this section we talked about the current state of procedural generation in games outside the academic line of work. We have also described features of procedural level generators and generation methods. We then worked out a small thought experiment in deciding what a procedural generator could look like as a solution to an example game level generation. Finally we have introduced what is cooperation in games and stressed the difference between cooperative multiplayer games and cooperative multiplayer games.

3 Related Work

3.1 Procedurally Generated Cooperative Games

After a brief search, one could see there was limited academic work on procedurally generated cooperative games, specifically none about procedurally generating a level with cooperative challenges. All that seemed to be related directly on the subject of this paper pointed to Cook et al.'s work on Angelina [30] [31] [32], an automated game designer. Angelina uses a co-evolutionary approach to generate arcade games. Cook et al.'s later work proposed ACCME, a generator of *Metroidvania* games, a subgenre of platformer games. We must remember that, while this is an interesting creation and area, it is not what we are trying to solve in procedurally generating levels cooperative games. Cook et al.'s approach intends to create whole games from either scratch or with providence of a human (or in the future, computer) designed map whereas our approach intends to solve generating levels. Furthermore, none of Angelina´s generated games are cooperative in gameplay: the use of the word here is to infer the cooperation of several, previously unattached design generation solutions, much like a pipeline. Our approach would consist of creating a level generator for an existing game and not creating the game with cooperative mechanics itself, thus Cook et al.'s work does not directly relate to our problem.

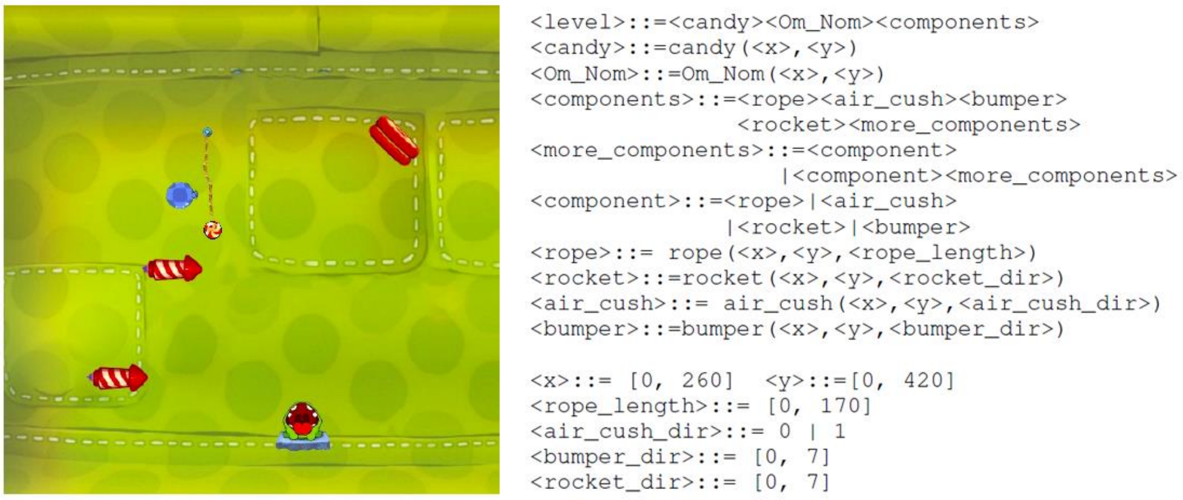
Having found that the academic sources had not explored cooperative level generation, we looked to commercial and hobbyist's attempts to solve this problem. A few forum posts and comments online showed exploration of this subject and efforts to solve it but consistently they had become irresponsive or admit the issue was too much work and dropped their prototypes. On the other hand commercial games that claim having a cooperative procedural level generation usually do not actually have a cooperative approach: level generation is purely single-player, you just have more people jumping and shooting at things at the same time, with no actual cooperative actions required. Examples of this include games like 20XX [33], Cloud Berry Kingdom [4] where procedural level generation would be the same ignoring the number of players since no character would possess cooperative abilities.

In the end we have not found directly related work, be it in the academic area, where the phrase cooperative generation is being used to describe the aggregation of distinct steps which can be fulfilled by either man or machine, or in a commercial area, where procedural level generators do not take into account cooperative actions and so levels do not have cooperation challenges, only single person challenges. Note that in a level designed for a single player cooperation can still exist and may actually be required for lower skilled players to surpass a more difficult challenge. These challenges are all currently being generated as solvable by an individual.

Facing this situation we will attempt to find a solution for this problem based on what has been worked on in the two separate areas that overlap with our subject: Procedural level generation and cooperation in games.

3.2 Procedural Level Generation

Several types of PCGs can be used; one of those is an evolutionary generator. These generators intend to reach increasingly good solutions to the problem in question. In [34] Shaker et al. explore an evolutionary approach to generating puzzle levels. They use a technique combining an evolutionary algorithm with a grammatical representation of the level which they call Grammatical Evolution (GE). Using a grammar to assemble a level is a declarative-based approach and its combination with the evolutionary algorithm makes it an evolutionary generator. In their implementation a level is a one-dimensional list of objects, which can have proprieties and record their positions in the map. As a means to test the validity of a later proposed heuristic, Shaker et al. evolve levels written in this grammar with a genotype-to-phenotype approach. The resulting phenotype is evaluated using a temporary fitness function and its fitness is returned to the evolutionary algorithm. Shaker et al. refers that the fitness parameter desired was playability and the second thing their paper was trying to solve was how to use a simulation-based fitness function that plays the level to show it can be solved. Since the level generator came first the levels in their paper are being evolved using a linear combination of several conditions as a fitness function. These conditions include four objects' positioning parameters defined by arbitrarily chosen placement rules that fit the level design, an orientation parameter, judging where rockets were aimed at and the predefined distance to exist between objects. Each condition would apply a penalty to the function and effectively pulling the level away from being desirable to the evolutionary part of the algorithm. To properly balance the fitness function values, Shaker et al. also weighted each individual parameter, choosing an arbitrary number multiplied by the number of instances each object occurs. They implemented their proposal using existing GEVA [54] [55] software, grammatical evolution for java. They used the half-and-half initialization method for 500 runs, each lasting 1000 generations with population size of 100 individuals.



```
<level>::=<candy><Om_Nom><components>
<candy>::=candy(<x>,<y>)
<Om_Nom>::=Om_Nom(<x>,<y>)
<components>::=<rope><air_cush><bumper>
                <rocket><more_components>
<more_components>::=<component>
                    |<component><more_components>
<component>::=<rope>|<air_cush>
              |<rocket>|<bumper>
<rope>::= rope(<x>,<y>,<rope_length>)
<rocket>::=rocket(<x>,<y>,<rocket_dir>)
<air_cush>::= air_cush(<x>,<y>,<air_cush_dir>)
<bumper>::=bumper(<x>,<y>,<bumper_dir>)

<x>::= [0, 260]  <y>::=[0, 420]
<rope_length>::= [0, 170]
<air_cush_dir>::= 0 | 1
<bumper_dir>::= [0, 7]
<rocket_dir>::= [0, 7]
```

Figure 5. Visual example of a level generated using GE and the grammar it generated from. This image was taken from Shaker et al.'s [34].

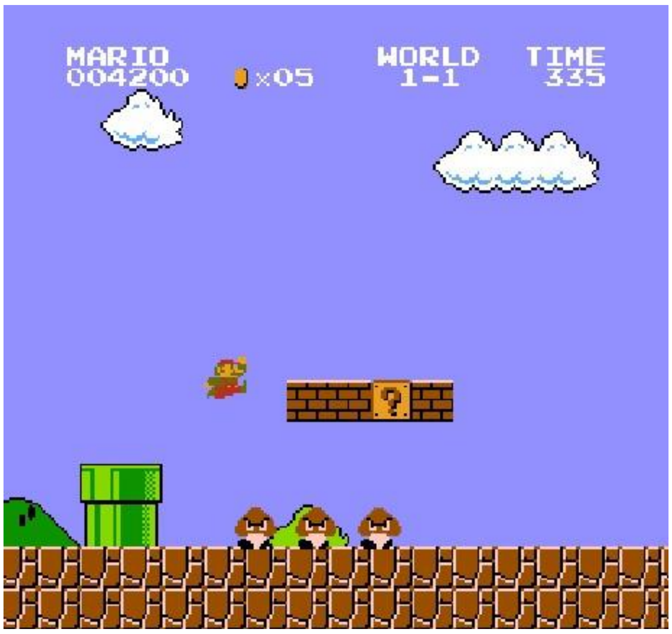


Figure 6. A group of enemies in Super Mario Bros. They can be represented in a basic pattern. The combination of the pipe jump and platform jump can be represented in a complex pattern.

Compton et al. introduced a way to segment level design in [35] into small patterns. A pattern is an agglomerate of simple game components, such as a jump challenge or a single enemy. Compton et al. divide patterns into a few categories: basic, complex, compound and composite. Basic patterns are composed of a single component, either by itself or in repetition, with no variation occurring. An example of this is the first mushroom hostile group that appears in Super Mario. Complex patterns are repetitions of the same component with tweaks and variation occurring. They exemplify this with a series of horizontal jumps increasing in width. Compound patterns alternates two basic patterns composed of different components.

Composite patterns are components placed so close to each other that require a different than normal approach. Compton et al. refer that the use of these patterns usually leads to linear experiences, and to make non-linear experiences another layer is required to deal with nonlinearity. To deal with this they propose a cell based solution with Cells and Cell structures. These are nothing more than grouping of patterns into a graph, permitting layers follow different paths.

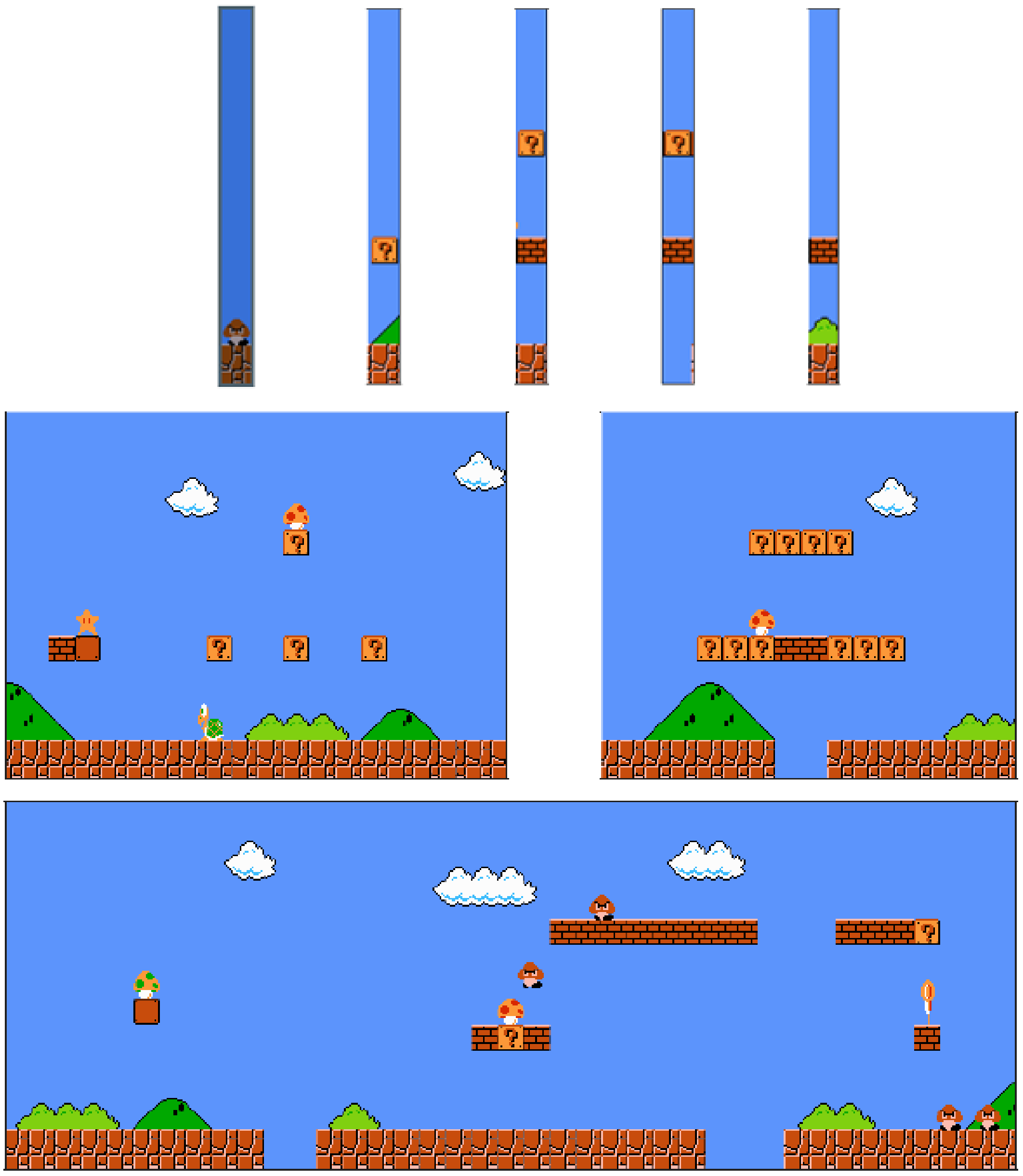


Figure 7. Micro (top), Meso (middle) and Macro-patterns (bottom) identified by Dahlskog et al. in their work.

Compton et al. only tested pattern building with a simple context-free grammar. Using the level as a start symbol, patterns and cells as non-terminal symbols, cells structures and pattern combinations as productions and components as terminal symbols they've developed a simple way of generating strings that represent a valid level. A level is composed of a cell structure which can be composed of other cell structures or actual cells. These cells will be attributed a pattern which in turn can be recursively expanded into more patterns. Finally the resulting patterns are converted into components creating a level description, which can be interpreted into an actual in-game level. To evaluate generated levels a hill-climbing algorithm is used over a difficulty calculator. Compton et al.'s difficulty calculator predicts possible paths and trajectories a player can take in a platformer, such as judging how high a player can jump. It can estimate the temporal window a player has to change direction mid-air to cope with more complex challenge designs too. This nets a smooth fitness graph which is fed to the hill-climbing algorithm. Having few local maxima the hill-climbing algorithm has a fair chance or returning an optimal solution, and even non-optimal solutions are usually not far enough from the desired mark to be noticeable by the player.

In [36] and [37] Dahlskog et al. have extended the previous work on patterns, defining meso-patterns and macro-patterns, and redefining previous patterns as micro-patterns. Meso-patterns are local combinations of micro-patterns, which consist of small challenges to the player. They notice how these usually take up a screen in Super Mario. We think that meso-patterns end up being what would be in Cells mentioned in Compton et al.'s work. Macro-patterns are combinations of several meso-patterns that form a bigger length of a level, usually with an objective. These can be used to control a steady increase in difficulty or to teach players game mechanics, where a pattern would first appear in a simpler form and then in a more complicated. Dahlskog et al. further explore this concept and analyse the game Super Mario Bros, identifying meso-patterns and macro-patterns that combine usually 3 meso-patterns. In their work, Dahlskog et al. create an automatic level analysis tool that reads any Super Mario Bros level encoded in a specific simple file and returns a list of micro-patterns together with their frequencies and the order of all meso-patterns. These last are organized into categories with distinctive names such as *Pipe-Valley*, *Three-Horde*, *Stair*, *Risk and Reward* and *3-Path* (last two pictured in figure 7). Being able to extract pattern from known functioning levels present in the original game or its procedural generated version *Infinity Mario Bros*, they use these patterns as a basis for an evolutionary algorithm. Their search-based approach uses a fitness function that rewards the presence of meso-patterns with a simple evolutionary strategy $A + B$ where $A = B = 100$ combined with the operators *single-point mutation* and *one-point crossover*. This method makes it so that in a population of 1000 individuals the lower fit half is discarded and the remaining better half is used as parents pairwise. Offspring is also subject to mutation. Unplayable content is deal with by setting their fitness values extremely low. A mutation changes a set of up to 5 slices in a random position with a new set of random 5 slices. Their fitness function measures the presence and order of meso-patterns by string searching. A value is awarded when a sub-string is detected that represents a meso-pattern taken from Super Mario Bros.

3.3 Cooperation in Games

As for cooperation in games, a good amount of research has been developed over the last decade. Researchers recognize the importance of cooperation on several situations and gaming is one of them. But how can we define and section cooperation in games? Rocha et al [38] set out to define characteristics of cooperation in games. They tell us that there has been an increase in interest on cooperative gaming in the market, highlighting the game genre MMORPGs. Other games are also mentioned such as Counter-Strike [26] and Team Fortress 2 [39], games with cooperative team based gameplay, and Lego Star Wars [40], a game which rekindles the focus of cooperative gaming. Many titles have since been launched so an update is in order. Titles like Portal [24], Left4Dead [18], PayDay [41], League of Legends [27] and Magicka [29] have been released which explore the cooperative gameplay much further than what existed back in their paper's time. These titles are completely cooperative multiplayer focused and we believe Rocha et al.'s work would benefit being revisited, not because its content has been invalidated but because the increase in titles and ideas would increase the diversity of game mechanics and designs encountered and defined.

When tackling a subject one should come up with a way to identify its important features or identifying characteristics that make it differ. Rocha et al identified several design patterns when developing cooperative games. Design patterns help us make more uniform the complex abstract concepts design patterns that exist in programming and to help virtualise a problem leading to a more generalised solution to be found. In this case it enables clumping games together design-wise. The mechanics identified include complementarity, synergy between abilities, abilities you can only use on another player, synergy between goals, shared goals and special rules for players in the same team. Rocha et al. intended these patterns for use developing cooperative games, but we will be using them to support our definition of a cooperative procedurally generated level and its evaluation. It is interesting to note that while aimed at cooperative games Rocha et al.'s work is sufficient to define competitive games or even game mechanics, for example the unit diversity in an RTS such as Starcraft [25]. Thus the typical rock-scissors-paper should be applied both to unit balance in a RTS and to player character compatibility in a game like Trine [42]. Next we will expand game mechanics and design patterns interesting to our problem.

Complementarity – A cooperative game can have characters that complement each other's actions or roles. A heavy knight should block the foes arrows while your archer makes it rain shafts on your foes. Not having all the abilities available in your character makes teamwork a requirement or at least something pleasant to have. In platformers this usually expresses itself as complementary abilities, which are needed to solve puzzles. In Geometry Friends [5], the circle can jump, something the rectangle can do. However sections exist where the circle cannot fit. Here comes in play the rectangle's ability to change its length and thickness. When generating levels we will have to take into account not only what actions a single entity can perform but also what actions can both players perform together. Only then would level generation be truly cooperative.

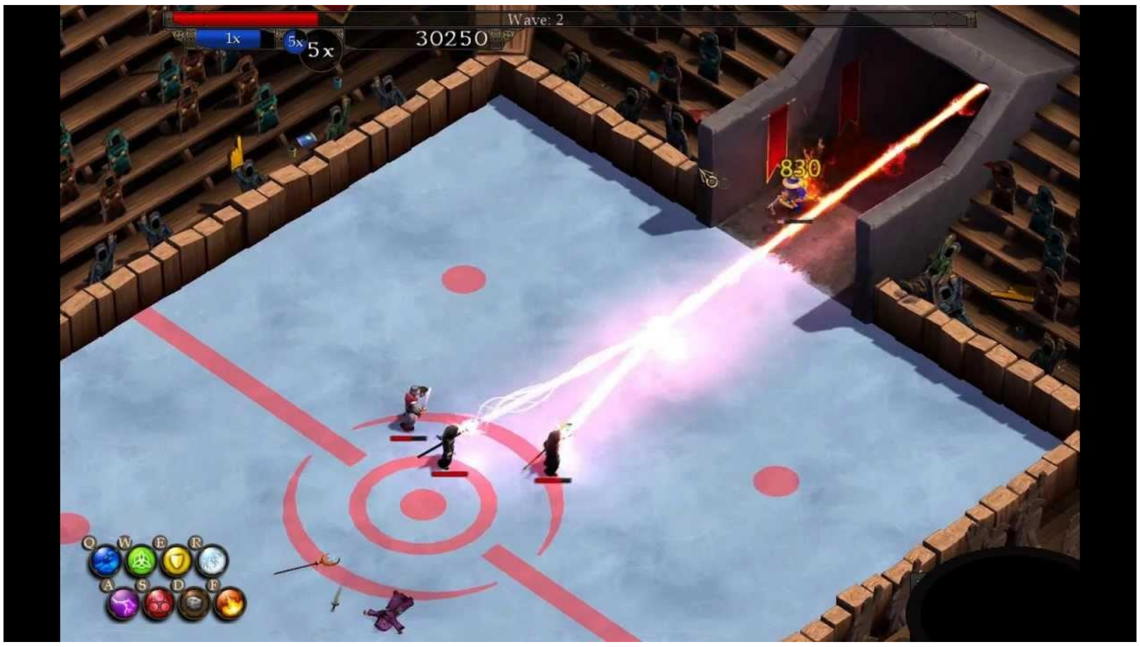


Figure 8. Two players join their beams in order to maximize damage to an entry point.

Synergy between abilities – Abilities can have different effects that augment each other. Players stick together to pack a greater punch that both alone would accomplish. In Magicka, two mages can cross their same coloured beams in order to join them into a stronger beam, which has a different direction than both separate beams. In Geometry Friends if the circle jumps as the rectangle makes itself taller, the circle will go higher than it normally would.

Abilities you can only use on another player – These are quite self-explanatory. A player's positive abilities can only be used on another player, such as Soraka's *Heal* in League of Legends, and so cooperation is required to fully utilize a character.

Shared goals – In massively multiplayer games players are forced to cooperate by the presence of common goals which require the presence of another player to complete. It forces cooperation between players even in situations where it would not be required. While this does not sound good, it is a tactic employed by several designers in an attempt to connect the users. Of course, this is not a negative thing in a non-open world game, such as a platformer. In this case players are cooperating to complete the level, such as in Portal 2.

Synergy between goals – As a way to make players cooperate through goals without actively forcing them to do the same thing one can encourage cooperation with an action both players benefit for different reasons. As an example, in a MMORPG where a craftsman would have a quest to make a bone tooth collar and a leatherworker had to make a jacket both could team up to hunt wolves, one taking the teeth, the other keeping the pelts. These are less frequent since it is harder and more time consuming to design a net of co-dependencies between goals, but makes for a better experience.

Special rules for players in the same team – Making abilities and actions affect teammates differently is a tactic enforced to give a choice to a player between cooperating and competing. For example, League of Legends' Orianna's ability Shockwave will create an area which will slow hostiles and speed up allies. The ability can be used to chase down targets or to help teammates escape. In another game, Robocraft [43] Nano-Disrupters disintegrate hostile robots but repair allied robots. Since most other weapons are more effective as combat, Nano-Disrupter wielding robots must make the choice between supporting an ally and count on him to deal the damage while keeping him alive or to head to the front yourself and risk getting focused. In bigger games, like MMOs, developers force players to join clans and faction battles in order to have access to all content such as quests that are only accessible to a same-clan group of players.

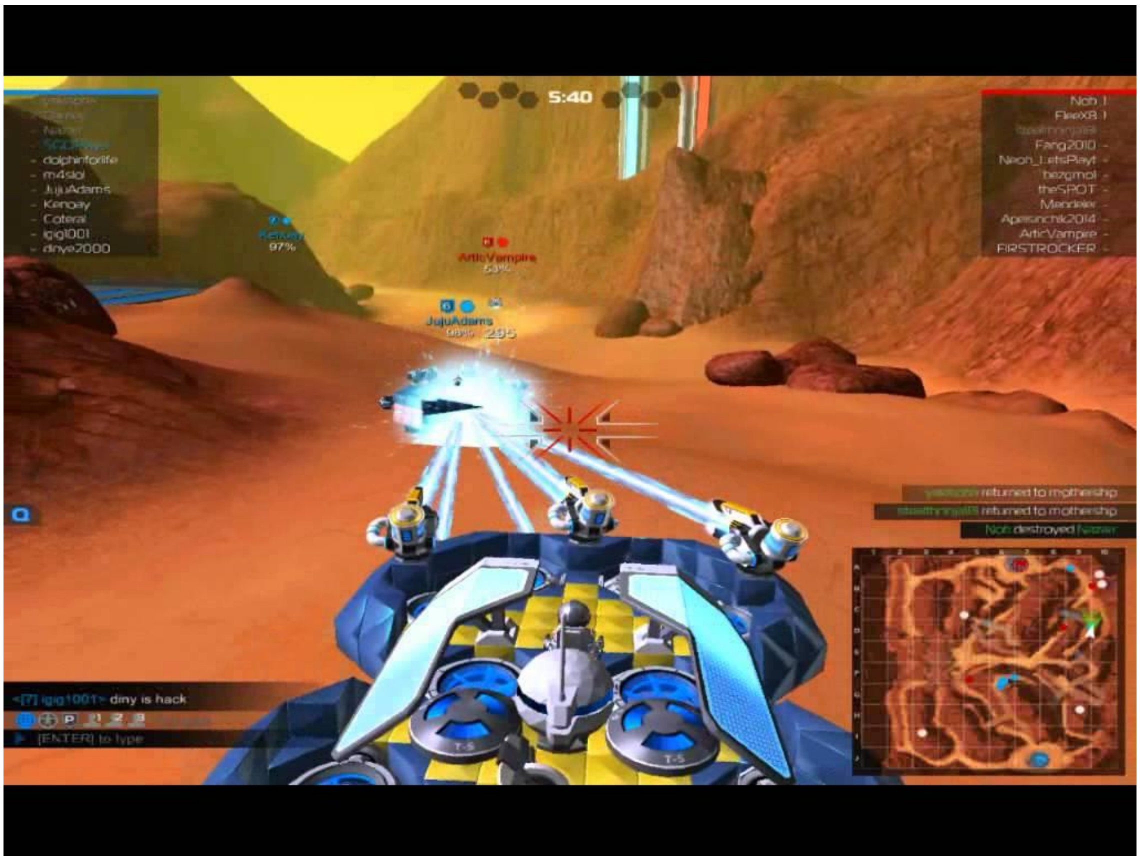


Figure 9. In Robocraft, a Nano-Disrupter wielding medic robot opting to heal a friendly robot rather than to charge the opponents.

These kinds of game mechanics are being used in a variety of games in a variety of combinations and different ways. The manner in which they are implemented usually decides if the cooperative part of the game is fun and interesting. It is also important to figure out how to make these work when you are generating the level. Cooperative game mechanics are usually implemented manually and if a procedural level generator in a cooperative game is to be successful it must produce interesting and fun results, in a cooperative sense too.

3.4 Evaluating Cooperative Games

But how do we see if our generator is creating a fun cooperative level? To arrive at an acceptable answer we have to use a method of evaluating fun in a cooperative game or at least how good it is. El-Nasr MS. et al. acknowledge this predicament and defend that designing good cooperative patterns and methods to evaluate cooperative play would be of benefit to the game industry. In [44] they extend design patterns worked by Rocha et al., previously mentioned in this document.

El-Nasr et al. explored several titles available at the time using game design theory. With peer review they've identified a several pattern, including ones previously described in this paper. The additional patterns *include Camera Setting*, *Interacting with the same object*, *Shared Puzzles*, *Shared Characters*, *Special characters targeting lone wolf*, *Vocalization* and *Limited resources*. We will follow with a quick description of each.

Camera Setting:

A visual design characteristic, El-Nasr et al. found three successful ways to implement camera settings. The screen can be split vertically or horizontally, commonly used on living room games, running on consoles, such as in Rock of Ages [45]. In a single screen, camera can focus on a single target. Finally, also in a single screen, the camera can focus all the characters, not moving unless all characters are close together. A good, more recent example of this last one is Magicka's camera.

Interacting with the same object:

Players can be provided with objects that can be manipulated by their characters' abilities. Simultaneous interaction with the object will net different results. In their paper Little Big Planet's push and grab abilities [46] is used as an example of this mechanic. Up to 4 players can band together to pull an otherwise impossible to move lever, allowing them to proceed in the level. This mechanic is maintained in later instalments of the series.

Shared Puzzles:

An emphasis on cooperative puzzles comparing to the usual *Shared Goals* pattern, Shared Puzzles englobes all cooperative design puzzles. Little Big Planet and Portal 2 are good examples of this category.

Shared Characters:

The existence of a character with special abilities that players can take over but only one can. This pattern has been applied several times over the years. An old example is the Metal Slug series, where in some levels a vehicle would show up and one player could use it, becoming stronger and having access to new abilities.

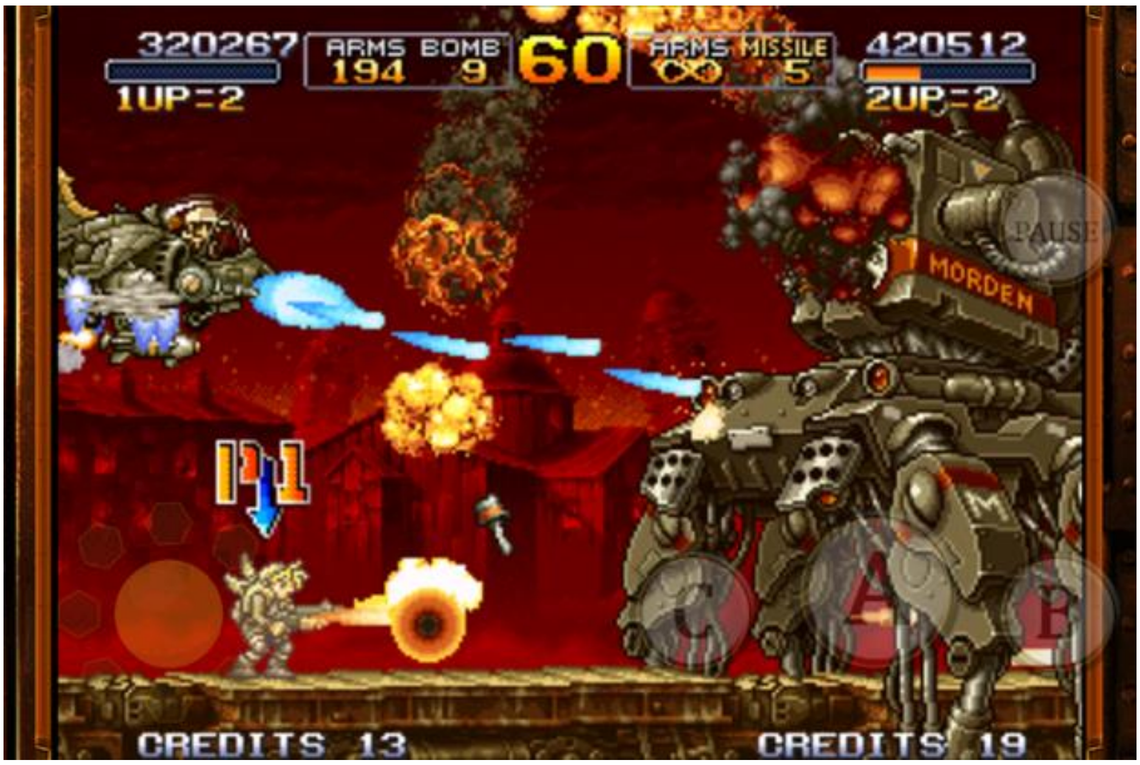


Figure 10. A shared character example in Metal Slug 2, a game for android. In this picture Player 2 is using a vehicle to target the upper part of the boss whereas Player 1 can only target its feet.

Special characters targeting lone wolf:

As way to encourage players sticking together, games can employ lone wolf targeting NPCs, such as the Smoker and Hunter in Left4Dead, and Charger and Jockey in Left4Dead 2. In these games, lone wolf targeting NPCs immobilize players and when suppressing a player only another player can stop them. These can appear in several amounts, so that even travelling as a pack of two is not safe.

Vocalization:

Some games include a vocal form of communication, with automatic pre-recorded messages. These can be used to inform other players of changes in situation and are a good reason to stay in shout's range of your teammates. FPS games have used this mechanic over the years, Counter-strike for tactical communication, Battlefield and Team Fortress series to call a medic. A noteworthy example of how vocalization (verbal communication) does not have to be in a known language is Journey [47], where players are able to shout at different intensities and use that as a form of communication.

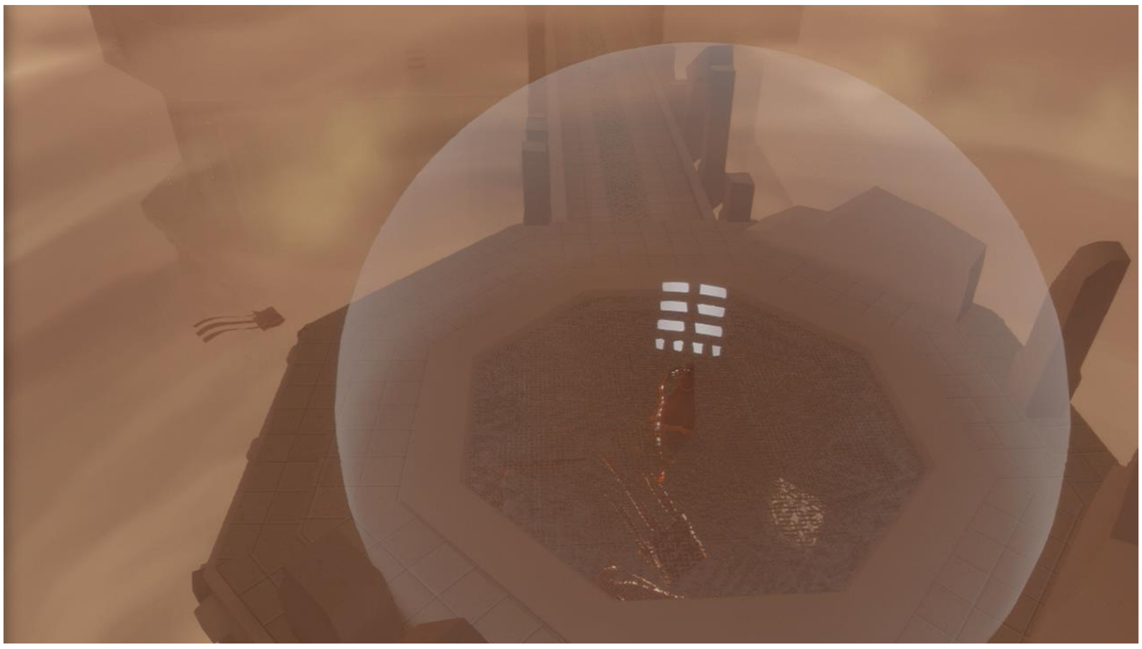


Figure 11. A player in Journey, chirping as a call for teammates.

Limited Resources:

Limiting the available resources creates a tendency to share or trade resources to research the same goal. This mechanic is present in most types of games, including the Civilization series [48], and Resident Evil series [49]. El-Nasr et al. also refer that this mechanic is a commonplace in board games.

After providing new and updated design mechanics El-Nasr et al. propose a set of Cooperative Performance Metrics (CPMs) to be of use when analysing cooperative games. These include *Laughter and Excitement together Events*, *Worked Out Strategies*, *Helping*, *Global Strategies* and *Waited for Each Other*.

These metrics are useful when trying to evaluate cooperation in a game. They allow you to see if the game is promoting cooperation and of this cooperation is positive. *Laughter and Excitement together Events* happened when all the players laughed at a situation occurring in the game. In their paper they restrict counting these events to once per cause. We believe this metric was used to gauge how comedic situations are in the game, be it from success or failure in the game. When *Worked Out Strategies* occurs, players gather to discuss solving shared challenges or divide a problem into smaller parts each player can resolve. It notes events in the game where players have to join together, or simply are worthy of cooperation. It is a mechanic that marks complexity and/or good cooperative challenges. These usually interest players who like a game for its cooperative gameplay and, as such, should be sought. However one must keep in mind that complexity is not always a good thing. *Helping* is a metric that occurs when players actively talk about controls and how one can use game mechanics, tell each other a correct way of overcoming a shared obstacle or saved and rescued a player who was failing. In their paper they note that researchers confuse this tactic with *Worked Out Strategies*. As a solution they only refer events as

Helping events when a player is helping another and not when both are helping each other. We think that this metric will show up when players' skill level is mismatched or when the game complexity is high. Players might lose interest in the game if they have to carry other players around or if they are always going back to help another player. The metric *Global Strategies* refers to events where players take up different roles to complement each other's abilities and responsibilities. We think it is a metric that should be present whenever a game tries to have complementarity. A good example is how players in *League of Legends* organize themselves into roles, such as damage dealer, support and tank, without the game actually forcing it. The last metric is *Got in each other's way*. It is defined by having a player prevented to do an action because another player has lagged behind or is trying to do another action that interferes with the other player's action. We do not think it is necessarily bad for this to happen but game sessions where this metric is high will definitely become frustrating to either player.

In their work El-Nasr et al. analyse 4 commercial games played by a group of children and record the play session. The session video is later analysed and CPMs are identified from the players' actions. Average, standard deviation and confidence intervals are taken into account, alongside with frequency and cause, netting a comparison between the 4 games, and identifying their pronounced patterns. With this they mean to identify what patterns work in games and why, presenting the results of the analysis as design lessons.

3.5 Driving Players to Cooperate

To develop cooperative games, or in our case a cooperative level generator, one should explore what drives players to cooperate. In [50] Zagal et al. explored board games and presented several lessons computer cooperative games could learn from their board game counterparts. They define a true cooperative game as a game where you do not backstab your teammates. In several games where cooperation is a part of gameplay there is a moment where you can backstab your teammates. Betrayal can be properly timed so as to create a competitive advantage in an otherwise competitive game. They mention Diplomacy [51] as an example where the key to success is establishing the right alliances and knowing when to backstab your allies. This is behaviour present in most competitive games which offer collaboration and we can give more modern examples such as the Civilization series and Diablo 2 [52], where a player could get help from other players to defeat a boss and pick up all the loot it dropped, proceeding to log out. Zagal et al. tell us that this is exactly what shouldn't happen in a collaborative (collaborative is how they refer to cooperative) game and as such one of the challenges of designing a collaborative game is dealing with the competitiveness players innately have.

Zagal et al. explore the board game Lord of The Rings [21]. In it you play as a hobbit out of a fellowship of up to 5 players that race together to reach Mordor and destroy the One Ring against the powers of corruption of the dark lord. If the hobbit currently bearing the ring is corrupted the game is lost. They draw a few lessons and pitfalls from the game:

Lesson 1: *To highlight problems of competitiveness, a collaborative game should introduce a tension between perceived individual utility and team utility*. Explaining the problem, they say that the objective lesson is to create something similar to a social dilemma, situations where individual rationality leads to collective irrationality. Allowing choices between personal gain and team gain one can highlight the problems of competitiveness.

Lesson 2: *To further highlight problems of competitiveness, individual players should be allowed to make decisions and take actions without the consent of the team*. Allowing players to be selfish and make competitive choices if that is the way they want to play. We think that allowing the player to make their own choices makes for a more interesting game, in contrast to a game where a player's actions could be overruled by the team. A player who cannot control its actions loses interest in the game, and so, even if it is a good way of making players to cooperate, forcing cooperative action through popular decision ultimately kills the game. The selfish player can still be persuaded by the rest of the team to make cooperative decisions and the communication creates a reason for a non-individualistic approach. This is essential to a good cooperation.

Lesson 3: *Players must be able to trace payoffs back to their decisions*. Good readability is a must in today's games but especially when individual versus group decisions are present a player needs to be informed of how their decision will affect the game. Positive or negative feedback from an action will reinforce the player's choice, and negative results from a selfish decision will make the player experience expectation failure where the player benefitted from his decision the most but later realizes not helping everyone actually makes his game harder.

Lesson 4: *To encourage team members to make selfless decisions, a collaborative game should bestow different abilities or responsibilities upon the players*. Assigning strengths and weaknesses to characters and making players having to choose a character that specialises in a part of the game helps players organize and decide where sacrifices should be allocated. In the board game Pandemic [22] a player with the medic character can more easily remove infection from cities. The objective of the game is to find cures, but the game can end if too many cities are infected. Since other characters have abilities that ease the researching of cures the medic can decide to remove infection from cities while allowing the researchers to research rather than heal. Likewise, if the medic player is closer to finding a cure a researcher might take up the remove infection task allowing the medic to spend his turn finishing the research even though both would be less effective at these tasks.

Pitfall 1: *To avoid the game degenerating into one player making the decisions for the team, collaborative games have to provide a sufficient rationale for collaboration*. Zabal et al. refer that it is easy to a collaborative game to degenerate into a solitaire game, which is a game that can be abstracted to one player performing all the actions. In these types of collaborative games the most skilled person would tell everyone what to do and the game would become boring. They also show us a few techniques to avoid this such as giving players different roles and abilities and to make the problem sufficiently difficult so that all the players need to cooperate to solve it. The first can also be improved upon by also adding hidden resources, forcing communication.

Pitfall 2: *For a game to be engaging, players need to care about the outcome and that outcome should have a satisfying result*. It is important that all players care about the outcome of the game. If players do not care or find the outcome unsatisfying or boring they will become uninterested in the game. We believe that for a cooperative game to succeed all players involved must feel useful throughout the game. If the specialized task a player has stops existing early in the game and the player simply follows along the team, while other players accomplish things and solve challenges that player will lose interest in playing that character again, or worse the game itself.

3.6 Summary

In this section we have explored the latest research and commercial endeavours into procedural generation for cooperative games. Having found a lack of these, we set out to see what has been done on both separate fields relating to this work's goal, procedural level generation and cooperation in games.

We have looked into how Shaker et al. have used an evolutionary algorithm to create a level, represented in a grammar, to be selected and evolved to fit chosen heuristics. We have also looked at Compton et al. and Dahlskog et al.'s work on defining patterns from designer made levels and also proceeded to generate levels, their fitness being how closely they matched patterns.

As for cooperation, we have run over Rocha et al. and El-Nasr et al.'s design patterns found in cooperative games. These are used to identify strongpoints and guide game designers into making better cooperative experiences. El-Nasr et al. also introduce Cooperative Performance Metrics for use when analysing cooperative games. Finally, we have read through Zabal et al.'s lessons and pitfall when designing a cooperative game, and what to do to avoid introducing competitiveness.

4. Testbed Game – Geometry Friends

The game we will be making a level generator for is Geometry Friends. This game was developed by Rocha et al. in their work [6]. It was designed to be a fun cooperative game, built around cooperative challenges.

As per its current host [56], Geometry Friends is a 2D platform game where two players cooperate to gather a set of diamonds in a level in the least amount of time. The characters players control are built around complementarity and each has its special abilities. The yellow circle is able to jump and roll, its movement is based on friction and jump height is constant, therefore once you are in the air you cannot change where you go. It can also change size and weight, being able to increase its size twofold. This ability can be worked with to allow for shorter jumps and rolling through gaps that would normally cause you to fall off the platform. The other character is the green rectangle. The green rectangle cannot jump nor push itself up or down but its movement does not follow the same rules as the yellow circle. It can, in most situations lob itself left or right, unless it's stuck. The ground-based green rectangle can also change its shape, thinning or widening itself into a taller or shorter form. This allows it to pass through smaller gaps and cross some ridges. This ability can also allow the green rectangle to climb stairs, provided adequate space is available.

The two players control different characters and should use the different abilities they have in order to complete levels by collecting diamonds, navigating through obstacles. These diamonds can be placed in areas where only one of the characters is able to go into and they can also be placed in areas where a character can only enter with the other character's help. As such the players should determine which collectibles require cooperative actions and which do not and divide the collection tasks presented in order to complete the level as fast as possible. Characters can be controlled using either a wiimote[53] or 4 keys on the keyboard.

The game is being further developed, new features such as different obstacles being added, and used as an AI cooperation game competition platform. The yearly competition has drawn several attempts at making cooperative AIs that can complete levels with cooperative challenges.

5. Tools

5.1 C#/XNA framework

XNA is the Microsoft's framework for games that was used in making Geometry Friends. It features a typical game cycle skeleton and supplies basic functionality classes, speeding up development of a game. The coding language it uses is C#.

5.2 Java

Oracle's Java is another coding language choice, Java applications run on a virtual machine independent of computer architecture. This makes it a great choice for cross-platform compatibility, while it does mean it won't perform as well. Since we were not working directly on the game Geometry Friends we could afford to make different choices as to whether use the same tools or integrate our tool into the game.

5.3 Choice

Since the base game functionality and controllers for sound and input wasn't a concern the choice was between choosing a coding language. C# offers object oriented capabilities and arguably a better performance than Java. On the other hand, while I had recently worked with both coding languages I had used C# with the Unity platform and would require to unlearn several habits I had developed working with the platform. With java that was not the case. Both coding languages have widespread communities and open-source libraries available. In the end we have decided to use Java for its rapid prototyping ability and cross-platform compatibility despite the game we are making levels for being made using the XNA platform. Levels would be generated and then exported into the XML format Geometry Friends uses. GEVA [55] also was made with Java so if we planned on using it Java was required.

6. Level Generator

6.1 Brainstorming

Our objective is to have a functioning level generator that generated levels with cooperative play. What we have in this document shows we have found no academic attempts to have a procedural level generator take into account cooperative play. The commercial field has also failed to provide us with a procedural cooperative level generator. As so we must look at current procedural generators and devise how can we make one include cooperative play and what challenges will we face.

The generation methods we have seen in current cooperative games in games that employ procedural level generation simply ignore cooperative mechanic. To avoid cooperative challenges, the game developers seem to have decided that a player alone can complete the level by himself. Furthermore some cases end up having the characters not interact which each other, leaving the game without cooperative play when resolving a puzzle. In the case of CloudBerry Kingdom this effectively turns the game into a race to see who fails less and who gets to the end first. Sure, one could argue that the players can still help each other by explaining how things work, and work out strategies to where an individual should go but that level of play could be present even in a single player game, where the player had a friend watching the game session. We chose to say that is not cooperative play.

Faced with this single-player generator, a procedural level generator that only takes one player into account, we've looked at ways in how to address solving cooperative challenges. A cooperative challenge is basically a puzzle that requires player A to be at said place and do said action while player B be at the same or another place doing another action. What we need to do is take into account players possible actions, alone and together. This would make it simple to adapt an already existing generation method, where you use an agent with a random set of actions, and simply generate a level through where he passes. As Compton et al. in [35] we can the same example of a physics model of the player to generate a level through players' actions, difficulty being assessed from how difficult an action is. If we could have a cooperative AI be the physics model the issue would be easy to solve. Alas, as seen by the results of the Geometry Friends AI competition, a cooperative AI that can complete a level that contains cooperative challenge is currently non-existing, so we had to use another approach.

If we fall into Pitfall 1 of the aforementioned Zagal et al.'s lessons we get a game that can be played by a single person controlling all the characters. This isn't good in game development but we believe it might actually be beneficial to think this way when making a level generator. If one can abstract the combined players' actions into a physics model that accounts both players abilities we could use that to generate our levels. Using the Geometry Friends example, imagine there was a platform at a height only the cooperative action of the circle jumping from over the rectangle would reach. This could be described as a jump action by the circle, on a non-existing platform and would mean to reach said platform the rectangle has to be in X position. These actions could be abstracted by the combined players of wherever circle and rectangle can reach circle's jump can be higher. For our physics model the platform would be reachable as long as the players could reach the required places. Sectioning the level into these reachable areas we could just say that when an area is reachable by the rectangle and the circle areas that require a circle and rectangle combined jump would be reachable and we could use this to make fully traversable levels. We shall call this approach the combined-player approach.

There is also the need for good level design for each level. We mentioned we can make a fully traversable level by linking areas. These areas would be linked to other areas that could be reachable by either of the players or both combined. The cooperation puzzles could occur inside these areas, or we could look at them like little black boxes where some characters can come in and others can come out. Using patterns, as stated in [37] we can build levels like assembling building blocks. If we label each block with a character label it is possible for us make building blocks with player presence requirements. A block that allowed both players could link to any block. A block that only allowed one of the players could not link to blocks that allowed the other players and it couldn't link to blocks that required cooperative action. A connected grid of blocks (2D or 3D) that followed the previous rule would be entirely traversable.

Now while the blocks architecture allows us to have long spanning levels, with multiple dimensions if do required, for a level to be fun its challenges must be more localized. As a solution each block is have a pattern. If the level is a macro pattern, or a series of them, each block can be viewed as a meso-pattern, composed of micro-patterns. Knowing what players can enter which block allows us to restrict what meso-patterns can be chosen for each block. These patterns would be acquired by analysing designer made levels, and would exist as the fitness function for our generator. The designer made levels would include cooperative challenges and those would be passed on into the level. The type of generation is another problem. Evolutionary generation, the method favoured by the referenced researchers, does provide us with interesting results, pattern hybrids and a lot of new content but also creates impossible blocks. Blocks that cannot be traversed and blocks that cannot be used for a level. Given enough time it is probable that a solution that players can properly traverse shows up but a lot of levels or blocks generated, along varying levels of fitness will have impossible challenges. This is a problem shared with lot of generator types that would only be easily soluble by guaranteeing that a level is traversable. The usual method of solving this, making an agent to verify that the level is traversable, is not yet available so this is something to keep in mind. However, we believe this would be a good and interesting way to create a procedural cooperative level generator.

To summarize, our general solution to add procedural level generation to a cooperative game would consist of a few steps. First off a representation of the level is required. This needs to be readable by the evaluation part of the algorithm and should contain all information required to define a level. Once a representation has been settled on, patterns should be identified on existing or new designer-made example levels. These patterns are to be defined in a way the evaluation part of the algorithm can compare the generated level representation with them, so similarities can be identified. Then, the generating part of the algorithm is to be created. This should be adapted to the game itself but in a general sense, it should create valid levels that can be analysed by the evaluation part and compared to patterns. Upon comparison we can use an evolutionary approach to select the best (closest matching to cooperative patterns and/or other metrics such as presence of cooperative challenges or bias to certain players) levels and create new generations, refining the selected ones. When available, a validation method should be used, such as a pair of cooperative agents or metrics.

We then set forth to apply this general concept to Geometry Friends.

6.2 First Approach

Applying what we said above to Geometry Friends means cutting a lot of things. Firstly, the game's level is restricted to a single screen, no more, no less. This restricted the number of blocks the level would be able to support. Secondly, the number of already available meso-patterns for analysis weren't numerous. The game simply did not have that many diverse levels. After a meeting I was advised to simplify what was to be done, as we had a limited amount of time and by past experience working with Geometry Friends, projects usually weren't as simple as they seemed and ended up being so complicated that only small steps were successful. Cooperation seemed to be a tricky thing to emulate on a computer. Taking this advice into account we've come up with a simpler plan.

The first generator we developed had a system of nodes and links. A node represented the top of a platform, containing height and width of said area that was traversable. The first node was the base of the level marking the floor. Every node would be initially linked to every other node twice, with an input and an output. The algorithm would run through every link and assign a value representing whether the circle, the rectangle, or only the circle with rectangle's assist could go from the node on the output of a node, to the input of the other node. What the circle or rectangle could reach was decided by checking whether other nodes were inside an umbrella shaped area above the output node. This umbrella shape was altered by how high the player could jump and how fast the player could be moving at the edge of the platform. Other node's landing areas would be used to make "shadows" by tracing the direction the closest tip of the original node and tips of the platform of the other node. After removing the shadowed part, if the landing area (top part) of a node is inside the remaining "Umbrella" the link is flagged to allow the character. This Umbrella could have 3 different heights from the platform: rectangle height, circle height and cooperative height. The last one would add the max height the rectangle could have to the jumping height. In figure 11 we can see an example of this umbrella shape for circle in platform A. The lighter, green area represents the umbrella shape, the darker, brown area represents the shadows that are to be cut out. In this case the circle can reach node B and the floor node from A but not node C.

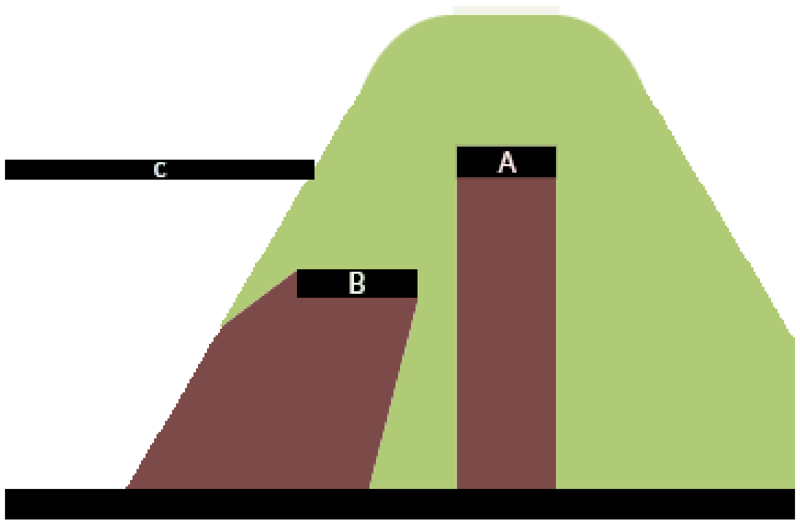


Figure 12.Nodes A and B casting shadows on Node A's "Umbrella" for the circle.

After all the links had been populated, empty ones would be deleted so as not to slow down parts of the algorithm to follow. The algorithm would then chose a node that had a cooperative link leading to it. If such a node could not be found then the algorithm would pick a random link from the remaining. The selected node would become the goal node. The origin of the link would be considered the start node. The algorithm would then search for nodes that could reach the start node, for both the rectangle and the circle individually. These nodes would be added, together with the start node to a list of possible start points for each of the players. One of would be chosen randomly for the circle and another for the rectangle. This ensured both the rectangle and circle could reach the platform where the cooperative action was needed and made it so that there was always at least one cooperative challenge per level, unless the level had no cooperative challenges based on jumping from node to node. The algorithm would then search for existing links, starting at the start node where the cooperative challenge was. At this point only links that allow at least one type of player to reach the input node remained, so we were basically seeing what was reachable from the start node. All nodes reachable would be added to a list of optional diamond nodes. The obligatory diamond nodes would be the goal node, and on later versions any node that was reachable by cooperative action. The algorithm would then choose a random number of optional diamond nodes to place a diamond marker in, also placing diamond markers in the obligatory nodes. Finally the existing data would be converted into a level format, with written with simple grammatical rules that would be used by an evolutionary part of this algorithm. This data was also able to be converted into a XML level, readable by the game. Conversion turned all nodes into black obstacles and randomly placed a diamond inside the bounds of the umbrella of a player that could reach the marked node.

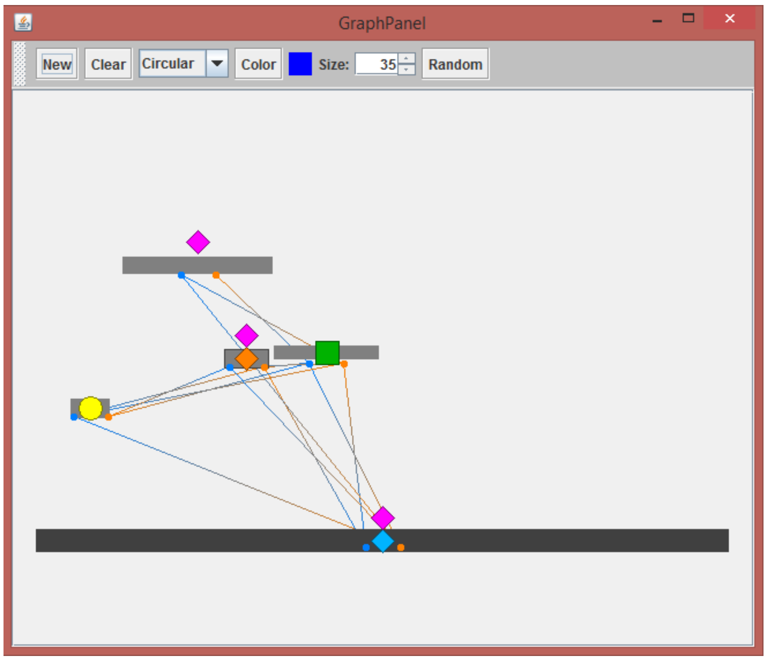
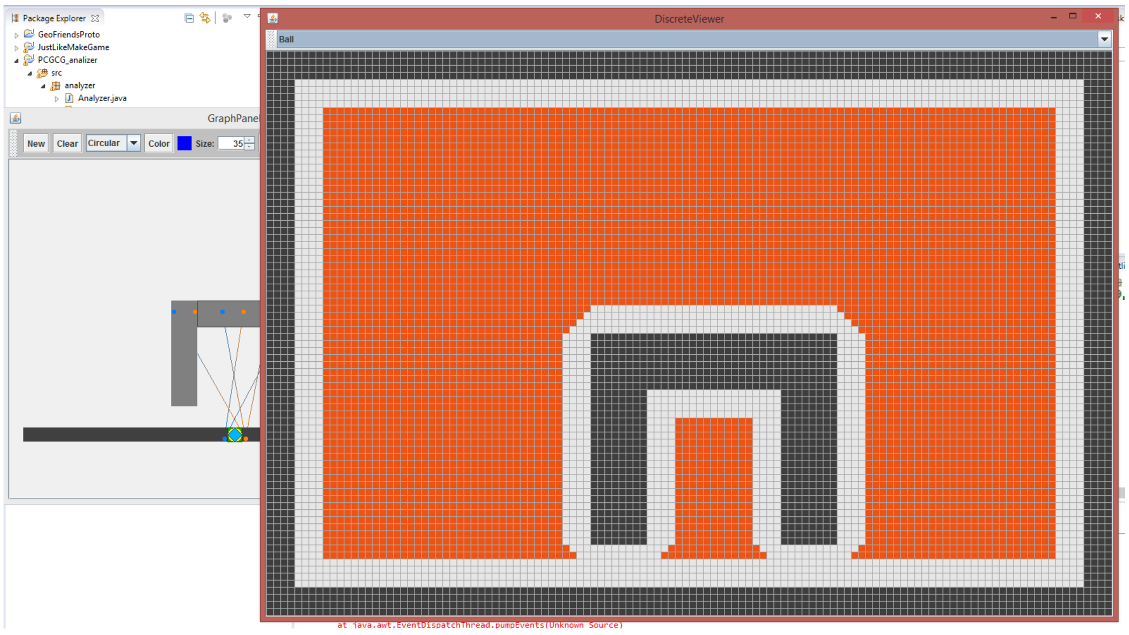


Figure 13. A visualisation of an example output from the first approach's algorithm, before conversion.

This version of the algorithm produced vertically interesting levels, when generating its own platforms. It could both read platforms from an existing file to use as nodes or it could generate its own nodes in a simple fashion. This node generator was capable of placing nodes to employ cooperative play and then extend those nodes to improve the circle's aerial gameplay. The levels were interesting for the circle player, but the rectangle player generally wasn't so fortunate. Sometimes the rectangle player would start in position for the circle player to do all the work. The algorithm also had problems with restricted areas, and separating the field of play into two. There wasn't a clear solution to how we could solve these issues and make the generated levels more interesting for the rectangle, so we added a second, discrete step.

6.3 Discrete Approach

To address the issues of the previous algorithm such as identifying unreachable areas and dealing with identifying areas where only the rectangle could go and assign those as areas of interest we came up with a different solution following some suggestions. This new algorithm divided the available space into cells. A cell represents an arbitrary area we use to analyse the level. Obstacles and the level's default edges would be read and where obstacles were present cells would be marked as occupied. Afterwards, the algorithm ran through every cell marking the distance to the closest occupied cell. This was done by checking every neighbour cells for an occupied cell, and if not found simply take the lowest value found. This meant the distance we are using is a Manhattan distance [58]. This information would be used in another step, again running over every cell, where the distances to occupied cell would then be used to verify whether a player could fit in a cell. For the circle the cell was simply required to be above a certain value away from occupied cells but the process for the rectangle was more complex. The cell would be marked as fit for the rectangle using the same restriction as the circle's however in this case, the closest occupied cell was farther than a lower value that corresponded to the minimum width of the rectangle, cells, vertically or horizontally would be searched to check to see if the rectangle, in a stretched state, would fit through the gap, vertically or horizontally.



```
at java.awt.EventDispatchThread.pumpEvents(Unknown Source)
```

Figure 14. A visualisation of cells where circle fit in orange.

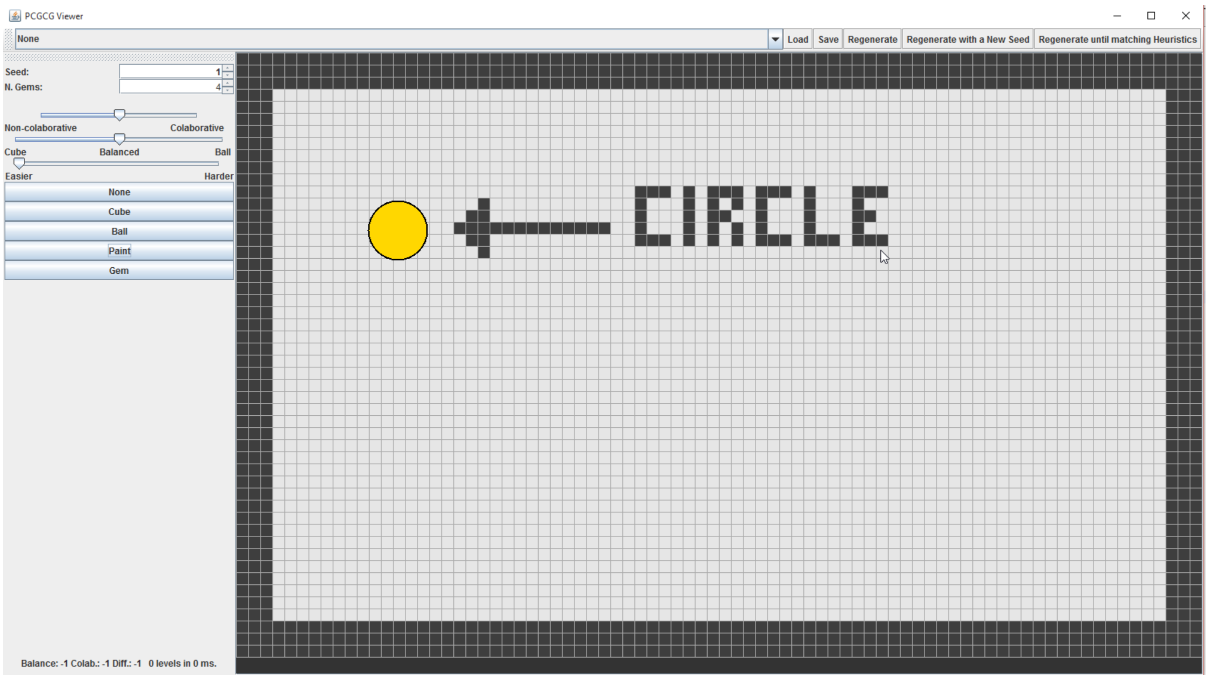
This added information would be then used in conjunction with the previous algorithm. After all the links were populated and the starting positions for the players chosen the algorithm would run the described discrete approach and have a cell map to work with. Separate areas of circle or rectangle fitness would be identified through a two-pass connected component labelling algorithm [59]. At this point the algorithm would go back to what was previously described, using this new cell map to assist deciding what nodes were valid candidates to be marked for diamonds and later when placing diamonds would be used to further reduce the size of the umbrella used when choosing a diamond's location for the node. After this it would scan for areas where the circle couldn't fit and belonged to the rectangle's starting connected component area or label and randomly place a diamond in the location of one of those cells.

This increased the diversity of combination of platforms (or a level's base) the algorithm could handle and also increasing level diversity generated. It also solved the problem of having both players separate, something the previous algorithm couldn't handle properly. However many problems still loomed. Placement was far from perfect, diamonds sometimes generated in impossible places and sometimes even outside the bounds of the map. Also progress was slow to make in platform generation, making the actual base levels. To make interesting levels we needed to either find out how to procedurally generate interesting cooperative challenges, or cooperative puzzles. This was not a trivial task. The general solution proposed to this problem would be replacing machine learning of what a good cooperative level is with an evolutionary algorithm that attempted to match designer made patterns. This proved a too big problem to tackle in the available time and the limited diversity Geometry Friends offers would make this effort have a diminished return. The algorithm producing impossible to finish levels was something that we also deemed unacceptable. So, upon suggestion to yet again simplify it down, we decided to focus some parts and drop other. The focus was to be on diamond placement from an already existing level base. Platforms would be already present and player starting positions would be already defined. So the remaining issue was knowing where to place diamonds to create cooperative challenges and promote cooperative actions while keeping the game interesting to both players.

6.4 The Final Generator

The previous program had several inconsistencies and organization problems that surfaced due to the fact the program was a patchwork of features added together as issues appeared. While working on it we remained knowledgeable of what existed and where was what, but that would become a major problem if we stopped working on it for a while, and that isn't something we wanted to have. So we decided to scrap what we had and start over fresh.

Firstly we developed a quick interface for the program to replace Frankenstein-like visualisation method we previously had. This interface and the initial work on the program resulted on a simple editor, that would load a level into a cell grid and mark them occupied where appropriate. It featured some basic editing tools, such as a cell paint tool that allowed the cell's occupied state be toggled and tools to change the players' starting positions and to place gems. It also had a save feature that translated the information displayed into a XML format level file, to be used in Geometry Friends. This level would have information where the players started, gems position and platform position and size. Platforms were derived from the cell grid using a simple system we developed. The two-connected-component labelling algorithm would be used to assign grouped occupied cells to Blobs. These blobs could be of various shapes and therefore wouldn't translate directly into a rectangular platform. In order to have rectangular blobs, for each blob we would select the top leftmost cell and proceed to find the longest uninterrupted sequence of occupied cells belonging to the blob to the selected cell's right, in the same line. Then the algorithm would check the next line for the same amount of occupied cells belonging to the blob starting from the cell bellow the selected cell. As long as complete lines were found under the previous line the algorithm would keep going until it reached the end of the blob, vertically. This would give us the biggest rectangular shape with the longest X size we could find, starting from the selected cell. We decided not to look for the biggest in both axis as it would be an unneeded waste of processing time. Cells in this biggest rectangular shape would be assigned a new blob and removed from the previous. This process would be repeated until the initial blob was empty, ensuring it had completely been converted into rectangular shapes. After this process the remaining blobs were used to define platform position and size. We have also given this basic editor some visualisation options for cell information, such as blob information and other features which we will later talk off, and heuristic values display and sliders. The heuristic information (which we will later discuss) would be saved in an accompanying .cmeta file whenever a level was saved, so as not to bloat the game files with unneeded information.



Seed: 1
N. Gems: 4

Balance: -1 Colab.: -1 Diff.: -1 0 levels in 0 ms.

Figure 15. A screenshot of the final program's interface, displaying the use of the paint tool to paint cells into letters.

The actual generator was developed in two steps, with the addition of heuristic guidance being part of step two. We will however focus on the final version of this generator instead describing it in two parts. This generator featured a cell grid but, unlike the previous cell implementation which was to be used as an auxiliary method of analysing a base level, this one was the main event. The algorithm can be separated into a few parts. A cell evaluation phase, three reachability phases, a gem generation phase and finally a heuristic evaluation phase. We will go into further detail for each of these phases.

Let's explain what we start from. In the beginning we have a grid of cells, sized 16x16 each, that cover the space the entire interface of Geometry Friends occupies when playing the game. The reason for this is that when the game read a level file its coordinate system starts at the top left corner of the interface instead of, for example, starting at the available level space, not obstructed by the interface. Since the interface in Geometry Friends also acts as the level's border it is solid and must be taken into account. Therefore the first act the generator does is set the cells at the edges of the cell grid as occupied. This 3 cell border is often ignored on the cycles pertaining reachability and fitting tests we will be seeing further ahead. However it is present exactly because of the first phase of the algorithm, cell evaluation.

The cell evaluation phase is quite simple, and one could say the legacy of the previous algorithm's cell map. In this phase we simply run through each cell, not counting the 3 cell border, and when the cell isn't occupied we check whether the circle fits and the rectangle fits. These checks are made in the same way the checks were made in the previous algorithm but with further refinement. For the rectangle we simply search the eight neighbourhood (known as Moore's neighbourhood, r=1 [60]) for occupied cells. If none is found the cell is marked as fitsRectangle. Notice that while the rectangle can fit through a 3x3 cell gap it is actually larger than this and so whether the cell is reachable or not is another matter. Due to the circle's size and fixed space the check for the circle's fitting uses the 24 neighbourhood (or Moore's neighbourhood, r=2 [60]). This means the cell will only be marked as fitsCircle when no cell adjacent to the cells adjacent is occupied. At the end of this phase our cell grid can show us where the circle and rectangle can fit.

The next phase is rectangle reachability. This phase's objective is to verify what cells are reachable by the rectangle player. We start by analysing the cell in the initial position of the rectangle, given a direction flag of 0. This flag indicates down, but can also have negative and positive values indicating left or right direction.

When analysing a cell we first verify if the cell has been marked as reachesRectangle. If it does, it has already been visited by the algorithm and, if the direction of the analysis is left, we check if the cell has the flag traversedRectangleLeft. If it does not we flag it, otherwise we simply stop the analysis for this cell. The same is done to the direction right, using the flag traversedRectangleRight. This ensures we do not end up in infinite loops of a rectangle falling from a platform and climbing it again.

Following that we verify if the cell fits the rectangle. This is done by checking the fitsRectangle flag. If this is true we then proceed to determine the next cells to analyse after marking the cell as reachesRectangle, the flag for rectangle reachability.

If the rectangle fits the cell under the current cell it is falling. Therefore we add the cell under it to the list together with one, or none, of the cells diagonally under the current cell. Which cell, left or right, is decided by the direction variable.

If the rectangle wasn't falling, we check whether the left cell has the fitsRectangle flag. If it does we analyse the cell to the left, with a direction of left (-1), and the cell above it, with a direction of down (0). If the left cell does not have the fitsRectangle flag we check the 3 cells above that cell using the left direction (-1). This allows us to take into the account the rectangle's ability to overtake small obstacles. It began as just a single cell above, but players in our first tests players proved the rectangle could overtake taller obstacles, and even climb decently sized ladders using momentum.

This concludes the cell analysis cycle, for the rectangle. At the end of this phase we have a reachability map for the rectangle. This shows us what places the rectangle can reach.

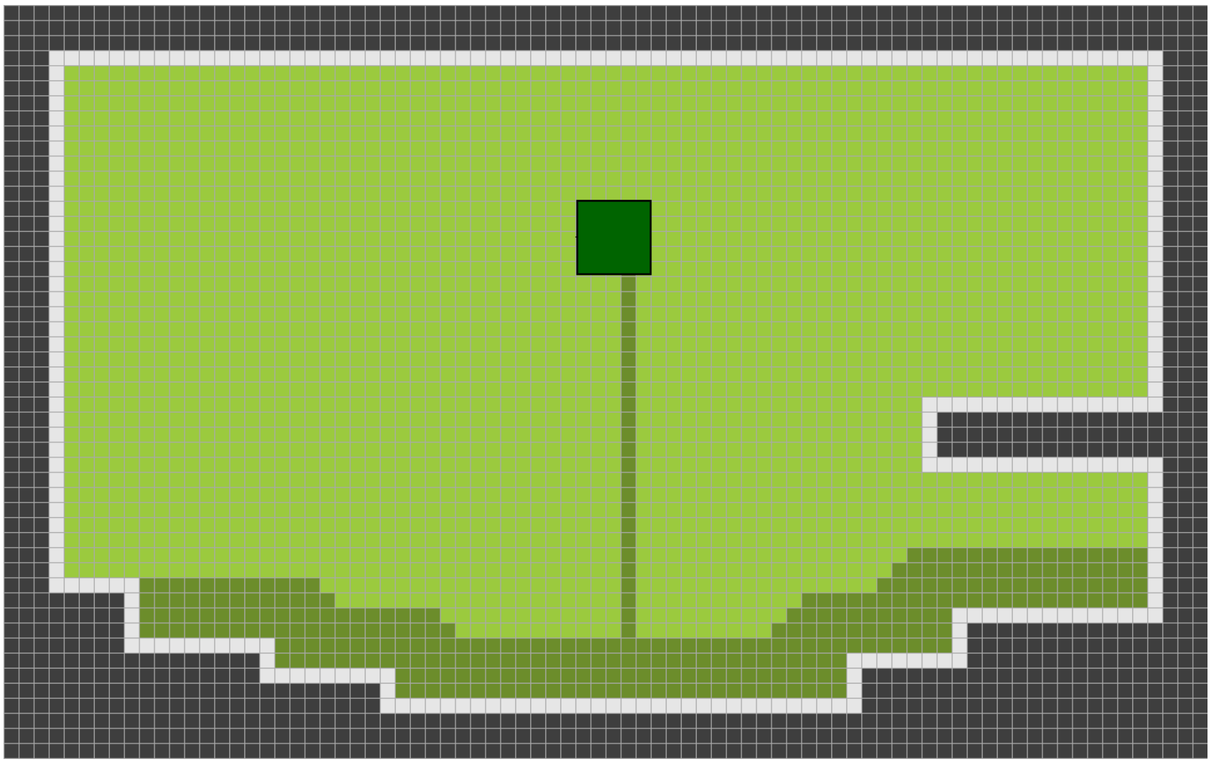


Figure 16. Visualisation of cells with flags fitsRectangle (Lighter Green) and reachesRectangle (Darker Green).

Next is the circle's reachability. For that we initialise a list of cells to analyse with the circle's starting position and proceed to analyse each cell present in the list until it is empty. The list is analysed in a FIFO (First In, First Out [61]) order. The cell analysis itself is similar to the rectangle's but had the added factor of when the circle is on the ground it can jump. The first step is to verify whether the circle fits the current cell. This is again done by checking the cell's fitsCircle flag. If it fits, the cell is marked with the reachesCircle flag. Then we look at the next cells to analyse.

If the cell below the current cell fits a circle (has the fitsCircle flag) it means the circle is mid-air. We will start by addressing when it isn't. When the cell below the current cell is occupied we set the current cell's jumpStrength to 24. This represents the number of cells the circle is able to jump. This number was an estimate that proved to be accurate through playtesting. Then if their jumpStrength is lower than 24 we add the cells left and right to the current cell to the cells to analyse list. A cell with 24 jumpStrength is certain to have been analysed and proved to be a jumping base and further analysis will not change that, so we do not need to go through it again. Then we set any cell that is directly above, the current cell (top left, top and top right cells) that has a jumpStrength value of lower than 23 to 23 and proceed to add those cells to the list of cells to analyse.

In case the circle is mid-air we first verify if the cell is mid-jump of free falling. To do that we verify if the cell has a jumpStrength of over 0 and under 24. If it does, we add any cell above the current cell that have a jumpStrength lower than the current cell's jumpStrength – 1, setting their jumpStrength to the current cell's jumpStrength -1. This means that any cell jumpStrength will steadily decrease to 0 as the analysed cell is further away from the jump point. It also means that a cell will always have its highest possible jumpStrength value. For free-falling cells we simply add to the cells to analyse list the cells below the current cell (bottom left, bottom and bottom right). This provides us of a suitable approximation of the circle's jumping arc that has proven to be quite accurate during playtesting.

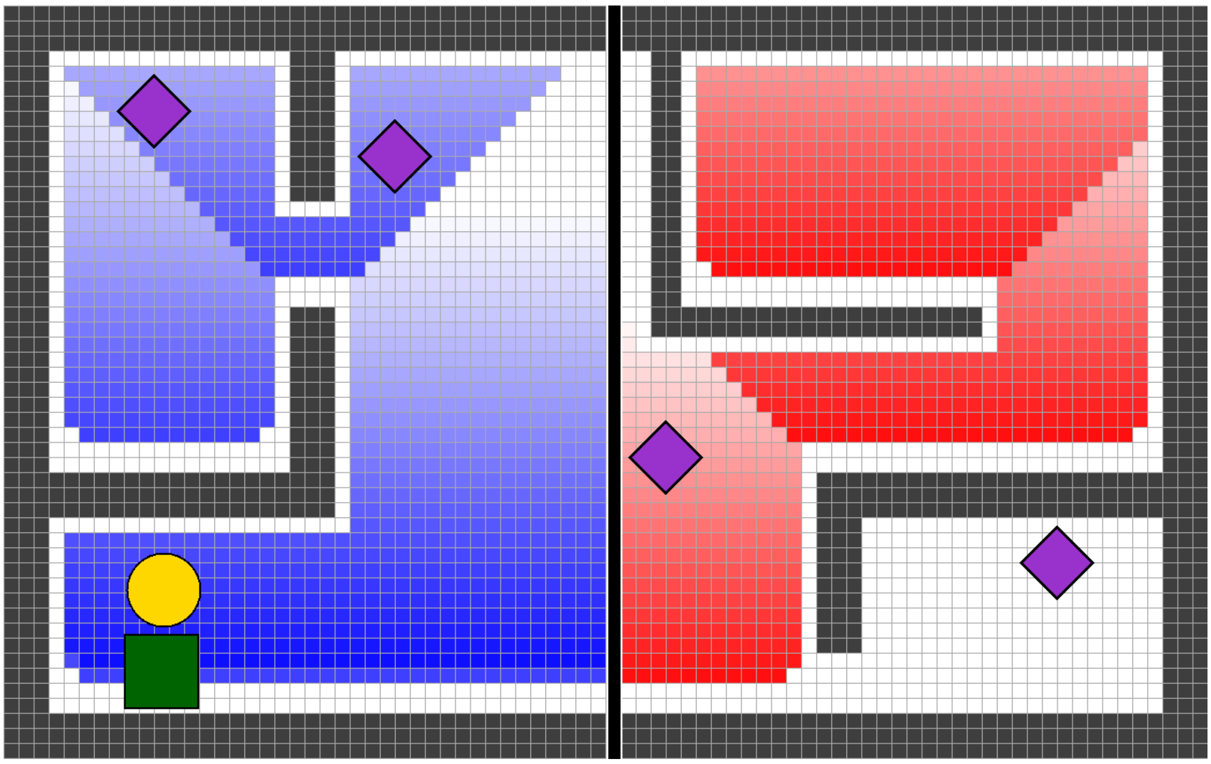


Figure 17. Side by side comparison of the jumpStrength (red) and coopJumpStrength (blue) values. Notice the lower value (less intense blue) near the bottom platform indicating a non-cooperative jump.

At the end of this phase we have flags that allow us to know what cells the circle can reach. We use this information together with the rectangle's reachability flags to have the next set of flag, cooperative reachability. This represents places the circle can reach with the help of the rectangle. The algorithm is pretty much a repetition of the previous circle reachability algorithm with a simple difference. Whenever there's a cell that allows for jumps if the cell also has the reachesRectangle flag the jumpStrength is to be set to 30 instead of 24. While not an accurate representation of the possibilities (the chosen approach simply adds height to the cooperative jump but fails to account for any extra lateral distance the rectangle could offer), the difficulty of actually doing a max speed assisted jump means most players would not be able to complete levels that took it into account. Through playtesting the choice seemed wise.

With this phase done we had access to reachability maps for the rectangle, the circle, and circle assisted by the rectangle. We then step to the final generation phase, gem generation.

For gem generation we want to have zones of interest defined. These zones of interest we will be using are what we call exclusivity zones. We will have cooperation exclusive zones, circle exclusive zones and rectangle exclusive zones. They each cell has a flag to mark it as belonging to one of these zones. A cell can be in a cooperative exclusive zone only if the cell is strictly reachable by cooperative actions. Likewise, a cell will be in a rectangle or circle exclusive zone if only the rectangle or circle, respectively, can reach the cell. Therefore we start by running over every cell and adding them to the proper list: coopExclusiveCells for a cell in a cooperative exclusive zone, rectangleExclusiveCells for cells in a rectangle exclusive zone and circleExclusiveCells for cells in a circle exclusive zone. Diamonds placed in cooperative exclusive zone's cells are what will make the level have cooperative challenges. Diamonds placed in rectangle or circle exclusive zones will not add a cooperative challenge, but when the amount of diamonds is balanced between players they add to the level, making players strategize on who gets which and divide tasks. As the algorithm might receive a base level where there are no cooperative exclusive zones, the algorithm must still be able to produce interesting levels.

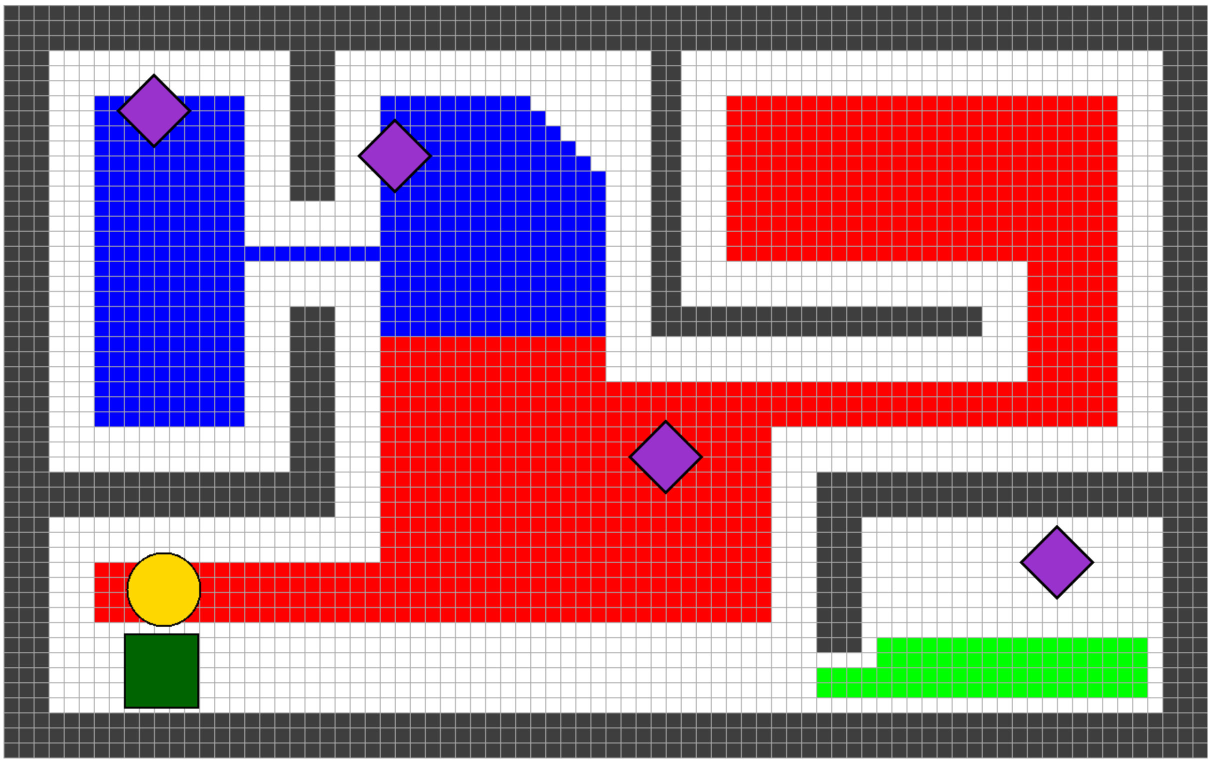


Figure 18. Visualisation of the exclusivity zones coopExclusiveCells (Blue), circleExclusiveCells (red) and rectangleExclusiveCells (green).

At this point we should mention heuristics. These are what guides algorithms into their solutions and its parameters determine their effectiveness. Ours is no exception. We have chosen 2 heuristics to guide our algorithm to interesting levels: collaboration and balance. Collaboration measures how strictly cooperative is a level. A level with the highest collaboration will only have diamonds placed in the cooperative exclusive zone. A level with the lowest collaboration will have no diamonds placed in the cooperative exclusive zone. Therefore levels with no cooperative exclusive zone will always have the lowest collaboration value. Next is balance. Balance measures how well the level's diamonds are distributed among players. Levels with a neutral balance are considered balanced and players will be required to collect the same number of diamonds. Levels with a bias will be off balance and a player will have to collect more diamonds than the other. These heuristics would be coupled with a difficulty heuristic, however we have failed to define difficulty in a measurable way that could be integrated in the system we devised. After numerous versions we decided to drop it and focus on the remaining two heuristics. Desired values for these heuristics can be set in the program, prior to level generation. For regular level generation they will only be used when decided where to place diamonds.

For the next step we first use a seed, set prior to generation, to initialize a pseudo-random number generator. Random numbers from this pseudo-random number generator will be used to decide where diamonds will be placed. This allows results to be repeatable. We use the values set prior to the generation referring to collaboration and balance. They are transformed so as to be described as -1 to 1 floats. Afterwards, for each diamond to generate, we take two random numbers between -1 and 1, coop and bias.

We first use the transformed collaboration value and compare it to the random coop value. If coop is lower than the transformed collaboration value we will generate a diamond in the cooperative exclusive zone: We randomly pick a cell from the cooperativeExclusiveCells list. If there are no cells remaining we set a success flag to false, so the diamond can be generated again. The diamond's position is the centre of the cell.

If coop is greater or equal than the transformed collaboration value we will generate either a diamond in either the rectangle exclusive zone or the circle exclusive zone. The zone where it generates is determined by comparing bias with the transformed balance value. If bias is greater than the transformed balance value we generate a diamond in a cell randomly picked from the rectangleExclusiveCells list. Again, if the list is empty we set the success flag to false, so the diamond can be regenerated. In this case the diamonds position is the centre of a random cell up to 3 cells above the selected, if no cells are occupied up to it. When bias is lower or equal to the transformed balance value, we generate a diamond in a randomly picked cell from the circleExclusiveCells list. Yet again, we set the success flag to false if the list is empty, so that we can try to generate the diamond again. The diamond takes the position of the centre of the selected cell.

This process continues until there are no diamonds left to generate or a timeout value is reached. The number of diamonds to generate is set prior to the generation start. At this point the level is completely generated and ready to be saved into an xml file and played in Geometry Friends. But our algorithm does one last thing. We evaluate the level. The way we do this for balance and collaboration is quite simple. The more than a handful ways we devised for difficulty not so much and they are the main reason we ditched difficulty, for now.

To assert the heuristics we ran every diamond's cell and checked active flags. For each diamond found in a cell with the cooperativeExclusive flag we added 1 to collaboration. When it wasn't in the cooperative exclusive zone we added -1 to collaboration and checked for the rectangleExclusive flag. If found we would add 1 to balance, adding -1 to balance otherwise. Notice that cooperativeExclusive cells have no effect on the balance. This is because 0 means the level is neutrally balanced. The numbers obtained are divided by the number of diamonds generated to net us percentages and multiplied by 10 to be displayed in the interface. This concludes standard generation.

6.5 With Heuristics

Summarising the state of the algorithm from the chapter above, the algorithm first searches for areas of interest, the exclusivity zones, then uses heuristics to figure out in which zones to place diamonds in. The levels is then evaluated and we get a new heuristic value, corresponding to the actual values on the level. We also made a system to better utilise these heuristics. It was made with an eventual shift to an evolutionary method, but for this work it remained less complicated. We use the button "Regenerate until matching heuristics" to have the program generate levels until it finds a solution that matches the heuristics initially provided, with a margin. The system is as follows. We begin by saving the initial heuristic values so we can later use them to compare with generated level heuristic values. Then we start generating levels, until one that matches the original heuristics, within a margin, is created or we reach a timeout value of 10000 generations. Each generation is done in around 1ms in our machine possessing an Intel i7-4760K 4.00GHz processor and 8 GB RAM running x64 Windows 10 and oracle's JDK 8u60. This would mean the timeout value would be reached in between 8000 ms and 13000 ms, unless other heavy processing tasks were running, something I avoided. The margin was defined as

$$margin = 0.1 + 0.1 * max\left(0, \frac{\text{timeout}}{1000} - 5\right)$$

Since timeout is an integer, only after 5000 iterations the margin would be increased. After that it would increase by 10% every 1000 iterations. At 10000 iterations it would reach over 50% making the margin too big for us to say the result matched the heuristics. So it generates one last time, using the original parameters and warns the user of a timeout. The timeout value was usually only reached by setting impossible parameters for a level base. If the level does not have cooperative exclusive zones a value of collaboration higher than -10 (the minimum on the interface) will force the generator to reach at least the first margin relaxation. This margin seems lax, however we were surprised to notice that for the generator to create a level that was close to what we had in mind when setting the heuristics the margin could often be as far as 30% without problems. To finalise, every iteration a new level would be generated and compared with the original heuristic values plus the margin. If a matching level was found the cycle would stop and the level would be displayed, along with its heuristic values and the number of iterations it took to make it, along with how long the process took. This brute force generation method was to later be replaced by another, smarter form of generation, such as an evolutionary generation, where after a few levels were generated, further iterations would mutate the starting levels until they closed on the heuristic values. But this remained as a shell in which we would later add features.

7. Evaluation

This work's purpose is to understand how a procedural cooperative level generator can be made. This is a procedural level generator that generates levels with cooperative challenges. Since we decided to use Geometry Friends as our platform, levels will be generated with the developed algorithm and arranged into a level series to be played by people. Using the same input we also have a human designer make a set of cooperative levels, to be used as a baseline for cooperation. We will also have a non-cooperative generator be fed the same input, to create another set of levels that have no cooperation. We intend to use these 3 series to determine whether people agree that levels generated by the algorithm are cooperative. As such we have two hypothesis relating to our cooperative algorithm, H1 and H2:

H1: Cooperative levels generated by the algorithm required coordinated effort to be successfully completed.

H2: Levels generated by the algorithm are indistinguishable from levels created by a person.

7.1 The Pre-tests

Having developed a functioning procedural level generator we set out to test it. We began by generating a series of 10 levels using the "Regenerate until matching Heuristics" option in our generator. The heuristics' values and gem number were set to acceptable numbers to the level's base layout, but were set to produce a neutrally balanced level with high (but not highest, in most cases) values of collaboration. The level's base layout allowed for cooperative challenges in all but one cases. The level non-cooperative layout served a comparison point, to give players a contrasting level. The other levels featured a rising difficulty curve, with more simple levels at the start of the series. The first few levels served to explore the controls of individual characters and cooperative actions. Later levels included more complicated challenges and areas where a mistake could mean a restart was required. The levels' base layouts were designed by us. These base levels were used not only in making the cooperative series (the series of levels generated by our procedural cooperative level generator) but also in making two other series, the non-cooperative series and designer series.

The non-cooperative series used a procedural level generator that did not take into account cooperation, but produced valid levels for Geometry Friends. I should note that results this generator produced were similar to results produced by our own procedural level generator if we set the collaboration heuristic to -10, and therefore forbidding diamonds to be placed in cooperative exclusive zones. The designer series was made in the editor by us and featured cooperative levels.

We planned to have players play each series, labelled A, B and C for cooperative, non-cooperative and designer series, respectively, and after each series fill out a questionnaire with questions pertaining how cooperative they found levels and whether they thought the levels were made by an algorithm. We had questions related to the whole series and questions relating to a specific level. The level specific questions were accompanied by an image of the starting level conditions. These questions were to be answered as a 5 point-Likert scale. The questionnaires also had a few questions on the player's age, gender and gaming habits. As for gaming habits we inquired how much time a participant dedicated to videogames weekly and how frequently does the participant play platform games. During each game session we would also record CPM events that occurred while each level was played. The CPMs we looked for were *Laughter and excitement together*, *Worked out strategies*, *Helping each other*, *Global Strategies and Got in each other's way*.

We first ran a quick pre-test with 2 participants and found that the 30 levels were too many, and players got bored in the second series, which affected the results. We also found that the latest version of Geometry Friends introduced some bugs and changes that made a few levels impossible to complete. In view of these results we cut the size of each series to only 6 levels, excluding some of the more complicated ones and altered some of the remaining level bases so as to avoid running into situations where the bugs occurred and hindered the players.

We then proceeded to do a pre-test with the corrected series and questionnaires. This time we had 10 users ages 21 to 53 with highly varying gaming habit backgrounds. 3 of the participants were female and 7 were male. The 10 users were arranged into 5 pairs. Each pair played the 3 series, in the order C, B, A, filling a questionnaire after each series, relating to it. While each pair played a level we counted CPM events that occurred playing that level. Every participant filled all questionnaire's answers and played every level. However some game bugs were still present and required level restart at times. These more often than not caused Laughter and excitement together events rather than frustration. The bugs included the rectangle sometimes teleporting into nearby positions, which caused the rectangle to be inside platforms or outside the field of play. The circle could also, in certain situations, jump again mid-air when it hit a platform's left wall. This was not intended, and facilitated some levels. The circle would also sometimes end up inside the rectangle, unable to be released. Whenever a bug occurred that led to the rectangle to be stuck or the circle or rectangle to reach zones they weren't supposed the level was restarted. Gameplay was recorded into a single 3 hour video which was left untreated.

Analysing these results seemed to prove our hypothesis, however it was pointed out that the experiment had some faults. We were advised to not have participants play every series as it would have an effect on results. We were also advised that results might be more favourable because the base levels and designer series were made by us. Since we know what the algorithm does and what it looks for we could have, even if unconsciously, made levels that favoured the algorithm. The same could be said about the collection of CPM events. We are definitely biased. So we decided to not use these results and make another experiment and series of tests with new participants.

7.2 The Experiment

For this new experiment we decided to have participants only play a single series. They would then fill a questionnaire relating to that series. As such we decided a higher number of levels could be afforded. So this time we decided to use 9 levels. We asked an independent researcher that had had contact with

Geometry Friends, and therefore knew the game mechanics, to design us levels. We requested that researcher provided us with levels with cooperative challenges and only a single one without, where the players were separated. Every other aspect, difficulty and complexity were left to his decision. The independent researcher provided us with 6 levels. So we had the desired 9 levels we decided to use a level from the recommended series distributed with older Geometry Friends version we had available and two from the latest cooperative AI challenge. The levels the researcher had provided were all complex and some tricky to solve in a timely fashion so the 3 levels we added were simple and serve as easier entry for participants to learn game mechanics. This formed the new C series, designer made set of levels. A and B series were then created using our cooperative generator and the non-cooperative generator respectively. We again used collaboration and balance values that fit the levels provided, where only the level where players were separated has no cooperative challenges.

The questionnaires were also changed. We decided to use a 6-point Likert scale rather than the previous 5 point one. With a pair scale we could avoid the "just pick the middle answer when bored" behaviour we witnessed in the first pre-test when the experiment went for too long, with the added fact that an indifferent answer has no value with our hypothesis. We also decided to have level related questions for more levels (for the 6 levels the independent researcher provided) and dropped most of the series related questions. Lastly we also started recording *Waited for each other* events that we missed in the pre-test.

In this experiment participants were asked to play through a single series of 9 levels and then fill out a questionnaire. As participants played the level we had an independent researcher mark the occurrence of the CPM events. This researcher had been told how to identify the different event. He also recorded completion time for each level. Completion time mattered because when participants took longer to finish a level they had more opportunity to have more CPM events.

7.3 The Results

7.3.1 Participants

Our sample consisted of 30 participants, the majority of which belonged to the male gender (N=27) and the average age is approximately 22 (M=22.4 and SD=7.050). This experiment was conducted at IST – Taguspark's campus – so all participants were related to this facilities (teachers and students).

Overall, the majority of participants reported playing less than an hour per week (8 participants). The next biggest group was participants who reported playing 3 to 7 hours weekly (7 participants) followed by 1 to 3 weekly hours (6 participants). The remaining participants reported the largest amount of hours per week reported spending more than 3 hours daily on gaming (5 participants) and lastly the less common group was participants who reported 1 to 3 hours daily (4 participants).

“Quanto tempo costuma dedicar a videojogos por semana?”

	Frequency	Percent
<1 hour	8	26.7
1 to 3 hours	6	20.0
3 to 7 hours	7	23.3
1 to 3 hours daily	4	13.3
>3 hours daily	5	16.7
Total	30	100.0

Table 1. How much time do you usually spent on videogames per week?

The majority for participants, concretely 56.7%, reported to sometimes play platform games. Participants who said they often play platform games represent 10% of the sample and 33.3% reported not playing platform games at all.

“Costuma jogar jogos de plataformas?”

	Frequency	Percent
Yes, frequently	3	10.0
Yes, rarely	17	56.7
No	10	33.3
Total	30	100.0

Table 2. Do you usually play platform games?

7.3.2 Statistical Hypothesis Testing

In order to test our first hypothesis, we verified A series’ mean report for the affirmation “Cooperation was required to complete the level.” for levels 2 to 7. Levels 2 (M = 5.4, SD = 0.699), 4 (M = 5.5, SD = 0.707) and 5 (M = 4.2, SD = 1.687) reported that cooperation was indeed required for completion. Level 3’s results were the lowest of the group (M = 1.7, SD = 1.567). This was expected as the level featured both players separate by a wall, with no possible interaction. The level 6 (M = 3.9, SD = 2.132) had particularity that is mistakes were not made the players could individually collect all the diamonds. However if a mistake happened, cooperation was required to get back in track. Lastly, level 7 (M = 3.1, SD = 2.025) did not require cooperation per se, but considering some player styles cooperation would be preferred for completion.

"Foi necessária cooperação para completar o nível"

A Series	N	Mean	Std. Deviation
Level 2	10	5.40	.699
Level 3	10	1.70	1.567
Level 4	10	5.50	.707
Level 5	10	4.20	1.687
Level 6	10	3.90	2.132
Level 7	10	3.10	2.025

Table 3. Reported results from the 6 point Likert scale from the affirmation "Cooperation was required to complete the level" for levels 2 to 7 from questionnaires for the A series.

The aforementioned tendency to use cooperation when possible is more notable in the B series which levels were made by the non-cooperative generator. This series can be argued to be easier, there were no diamonds in hard to reach places that only with cooperation could be collected. That didn't mean however that players did not cooperate. Levels were completed in far shorter time, being the only series where players often solved a level in under than 10 seconds and at worst players took 300 seconds to complete a level whereas in the other two series players took around 600 seconds at times, for the A series and 800 seconds for the C series. These higher finish times were recorded in level 2 and 5 for both series. This is mainly because players still chose to cooperate, given the choice. We observed participants outright choosing to cooperate to collect gems or make jumps even in cases where cooperation was not required, and at times prejudicial. Because of this players reported requiring cooperation in a more uniform way in the non-cooperative series, with reported values for the non-cooperative levels on other series, 3 (M = 3.0, SD = 2.582), 6 (M = 4.9, SD = 1.663), and 7 (M = 4.3, SD = 2.163), being higher. Level 3, where players are separated, remains the only level where participants on average didn't feel cooperation was required. While these results show us that players felt that cooperation was required in all level series, with less accentuated differences between levels in the non-cooperative levels, observation of gameplay seemed to suggest a lot less cooperation was actually done in the B series. Results from CPM events recorded (which we will explore further below) seem to agree with these last observations.

"Foi necessária cooperação para completar o nível"

B Series	N	Mean	Std. Deviation
Level 2	10	5.50	.850
Level 3	10	3.00	2.582
Level 4	10	5.40	1.265
Level 5	10	5.20	1.619
Level 6	10	4.90	1.663
Level 7	10	4.30	2.163

Table 4. Reported results from the 6 point Likert scale from the affirmation "Cooperation was required to complete the level" for levels 2 to 7 from questionnaires for the B series.

"Foi necessária cooperação para completar o nível"

C Series	N	Mean	Std. Deviation
Level 2	10	5.80	.632
Level 3	10	1.00	.000
Level 4	10	5.40	.699
Level 5	10	4.60	1.075
Level 6	10	3.40	1.897
Level 7	10	2.70	1.494

Table 5. Reported results from the 6 point Likert scale from the affirmation "Cooperation was required to complete the level" for levels 2 to 7 from questionnaires for the C series.

For CPMs events we focused on the results of *Worked out strategies* events for levels 2 to 7 in each series. This event was selected, as along with *Global* Strategies, is the event that only occurs with cooperative play. *Global Strategies* was not used as the game Geometry Friends forces it to exist a single event per level, as players always used the same complementary characters. Let's start with results for the A series. As before, Levels 2 (M = 2.0, SD = 1.764), 4 (M = 2.8, SD = 1.814), 5 (M = 2.8, SD = 1.814) are levels that require cooperative action and reported a higher amount of events. Level 3 (M = 1.0, SD = 0.667) reported the lowest amount of events. Level 6 (M = 1.6, SD = 0.516) and 7 (M = 1.2, SD = 0.789) also presented lower amounts of events.

Worked out strategies - A series

	N	Minimum	Maximum	Mean	Std. Deviation
Level 2	10	0	5	2.00	1.764
Level 3	10	0	2	1.00	.667
Level 4	10	1	6	2.80	1.814
Level 5	10	1	6	2.80	1.814
Level 6	10	1	2	1.60	.516
Level 7	10	0	2	1.20	.789

Table 6. Results from CPM event recording during play sessions for levels 2 to 7 of the A series.

The B series' result is highlighted by the maximum number of CPM events recorded in each level, which compared to the other two series' results are generally low, with the exception of level 6 and 7. We believe the exception is due to player's being more used to the non-cooperative levels in the B series, whereas in the other two series, these levels would pose less of a challenge compared to the previous ones. The positioning of the diamonds in the B series might also have influenced this, as it made the expected path for the circle to be less obvious. Overall these results show us that the B series had less cooperative events occur, even if participants felt that the levels were cooperative.

Worked out strategies - B series

	N	Minimum	Maximum	Mean	Std. Deviation
Level 2	10	1	3	2.00	.943
Level 3	10	0	2	.60	.843
Level 4	10	1	2	1.40	.516
Level 5	10	1	3	2.00	.667
Level 6	10	1	3	1.80	.789
Level 7	10	1	3	2.00	.667

Table 7. Results from CPM event recording during play sessions for levels 2 to 7 of the B series.

The C series has the highest mean values among all series recorded. It also has the highest minimum values for recorded CPM events. We believe this is due to the fact the series was design, with intent, by a person and as such the challenges were more refined and required greater interaction.

Worked out strategies - C series

	N	Minimum	Maximum	Mean	Std. Deviation
Level 2	10	2	6	3.80	1.549
Level 3	10	2	3	2.20	.422
Level 4	10	2	3	2.20	.422
Level 5	10	2	6	3.80	1.549
Level 6	10	1	3	2.00	.667
Level 7	10	0	2	1.60	.843

Table 8. Results from CPM event recording during play sessions for levels 2 to 7 of the C series.

In order to test H2, we decided to use an ANOVA to compare all 3 series to understand if people could distinguish between algorithm developed levels and levels designed by a person. Results suggest that participants weren't able to distinguish apart from each other. Our perspective is, as the levels' bases were exactly the same participants mainly looked to the platform placement and overlooked overlapping diamonds and the unorganized location of diamonds (compared to when levels were designed by the external researcher). The null hypothesis being rejected supports there not being a meaningful difference between both distributions however this does not support any qualitative analysis. In any case, these results confirm hypothesis 2. (See table 9).

Ideally, the levels generated by the non-cooperative generator wouldn't be identified as designed by a person, as the generator is quite simple and random in nature, for example overlapping diamonds in numerous occasions. This would contrast to levels designed by a person which would be identified as such, if no effort to disguise this was made. As such it was desirable that levels generated by the cooperative generator would be identified as being designed by a person, with results close to the ones from the levels designed by a person and different from the results of the non-cooperative level generator.

ANOVA - H2

		Sum of Squares	df	Mean Square	F	Sig.
Level 2 was designed by a person.	Between Groups	1.067	2	.533	.224	.801
	Within Groups	64.400	27	2.385		
	Total	65.467	29			
Level 3 was designed by a person.	Between Groups	3.800	2	1.900	.809	.456
	Within Groups	63.400	27	2.348		
	Total	67.200	29			
Level 4 was designed by a person.	Between Groups	1.067	2	.533	.193	.826
	Within Groups	74.800	27	2.770		
	Total	75.867	29			
Level 5 was designed by a person.	Between Groups	4.067	2	2.033	.714	.499
	Within Groups	76.900	27	2.848		
	Total	80.967	29			
Level 6 was designed by a person.	Between Groups	4.200	2	2.100	.708	.502
	Within Groups	80.100	27	2.967		
	Total	84.300	29			
Level 7 was designed by a person.	Between Groups	4.067	2	2.033	.640	.535
	Within Groups	85.800	27	3.178		
	Total	89.867	29			

Table 9. Reported results from the 6 point Likert scale's results from the affirmation "The level was designed by a person." for levels 2 to 7 from questionnaires.

8. Conclusions

In this work we developed a procedural cooperative level generator. Based on a general solution we implemented a level generator for the cooperative game Geometry Friends. This level generator works by identifying areas of interest (in this case diamond placement exclusivity zones) and then distributing diamonds following heuristic values. These heuristic values controlled level balance to each player and the proportion of cooperative challenges to non-cooperative challenges. We also developed a non-cooperative level generator for our experiment.

Our experiment consisted of having participants play a series of levels and then filling out a questionnaire intended on evaluating levels as cooperative. During playtesting CPM events were to be recorded, as to give us concrete data on whether cooperation was actually happening. 3 series of levels existed and players only get to play one, not knowing which. The A series was composed of levels generated by our procedural cooperative generator. The B series used the non-cooperative generator to produce its' levels. The C series was designed by a person. After results were collected we analysed them to test our hypothesis H1 and H2.

For our hypothesis 1 we wanted to see if players felt cooperation was required to complete levels generated by the algorithm. As seen in table 3 for the A series, players felt cooperation was required to complete levels 2, 4 and 5 all of which contained cooperative challenges, in the form of diamonds placed where the circle could only reach when assisted by the rectangle. The presence of Worked out strategies events in the A series (table 6) confirms that participants had to work together in order to overcome obstacles. However, participants also found levels in the non-cooperative B series to require cooperation in order to be completed (table 4). We can see in table 7 that the amount of cooperative events was far inferior to the other two series (table 6 and 8). This tells us players did not have to cooperate as often. The levels in the B series did not require cooperation to be completed however participants chose to cooperate in order to complete the level, as we did not restrict players to however they decided to play the game. When players were separate (level 3, all series) participants felt cooperation was not required. Otherwise, more often than not players felt the levels were cooperative. This allows us to conclude that, given a non-competitive game, where players have the same goal, when possible players will cooperate if cooperation is not restricted. The fact that the game requires two players and they have a common objective seems to be enough to cause them to cooperate.

As we can see in table 9 results the responses to "the level was designed by a person" affirmation in each level in the questionnaire confirm our hypothesis H2 that the players are not able to distinguish levels generated by the algorithm from levels generated by a human. We think this is due to the fact to the level's base similarities. Levels had the same platform layout but different diamond positioning. This result tells us that diamond positioning alone do not cause people to identify games as made by a person, in Geometry Friends.

8.1 Contributions

We proposed a general solution to procedurally generating levels with cooperative challenges. Patterns with cooperative challenges are to be analysed from pre-existing designer made levels. A level generator is to create the first generation of levels to be evolved into levels that closer match the cooperative patterns. Levels closer matching to patterns have higher fitness. Fitness can also be influenced by other metrics, such as balance and proportion of cooperative challenges to non-cooperative challenges.

We created a procedural level generator that generates levels with cooperative challenges for the game Geometry Friends. The algorithm analyses a level's base composed of the level's platforms and player position. An editor is provided to ease visualisation and edition of this level's base. From the exclusivity zones found, the algorithm then proceeds to place diamonds in the level to promote cooperative play and ensure the presence of cooperative challenges, when possible. We prepared a heuristic system to further improve results by the current generator and to later be adapted into a better, evolutionary method.

8.2 Limitations and Opportunities

The testbed game used, Geometry Friends has an important limiting factor to our idealised pattern approach. Since the levels are screen sized, we cannot have macro patterns in this game. Furthermore, actual level space is vertically limited to less than two circle jumps. If we had used a cooperative game that allowed bigger levels we could have explored pattern identification and use in evolving building blocks for the level. Our first approach featured nodes as a means of abstracting a zone into a single block, however the size restriction of the level meant there was no actual space to do anything other than place a single platform in the area the node was assigned. And so we decided, in this work to move away from implement it in Geometry Friends.

8.3 Extendibility

As described two chapters ago, the algorithm would benefit from being expanded to have an evolutionary part. The current algorithm generates a level and verifies whether it matches the desired heuristic values. If it doesn't it is discarded and a new one is generated, with no regards to previous generations. The margin system implemented allows for an increasingly diversity of values to be accepted, so long as they are close enough and a mechanism of varying the amount of diamonds to be generated is in place, which facilitates finding a solution however we believe this is not the best solution. What was intended was for the algorithm to generate a population of levels, and evolving the levels that closer matched the heuristics with few generations. We believe this would greatly improve results but not as much as the next suggestion.

The above suggestion was planned and should be easy to implement. As a means to further improve the generator we believe two things would have great impact. First would be base level generation. The current generator takes in an already made level. It is currently a mixed-authorship algorithm, with a human designing the base level. Used as such it can definitely produce good results, however taking the step to turn it into an algorithmic generation that used human help would allow a more massed use of the algorithm, such as automatic level generation in runtime. Using Geometry Friends as an example, for this we would need a generator that could create cooperative challenges such as jumps and gaps that the rectangle had to fill for the circle to pass over. We developed a simple one in the first generator, but it only generated platforms and so would generally have at most 2 cooperative challenges, and they would always be a jump. This would quickly become uninteresting for the rectangle. More diverse levels would be required.

The second is the automatic generation of cooperative puzzles. What sets cooperative games like portal is the puzzles present in the level. While our theorized solution would be able to match already made and analysed cooperative puzzle by identifying them as patterns and work towards them and it would be able to create hybrids of existing patterns it cannot create new patterns by itself. As such an algorithm that can create cooperative puzzles will certainly produce more interesting results, in the long run. Generated cooperative puzzles could then be used as a base to produce more interesting cooperative levels. This seems to be, however, a much more difficult task than what we have done here.

8.4 Summary

In this thesis we talk about procedural generation and cooperative games. We attempt to join the two fields together by making a procedural level generator that creates cooperative levels. We propose a general level generator and create a procedural level generator for the game Geometry Friends that included cooperative challenges. This had its setbacks and while we have not been able to fully recreate the proposed general generator our generator for Geometry Friends was used in experiments which results prove we can have a procedural level generator produce cooperative levels, which is what we set out to do. In this last chapter we discuss these results, list our contributions and exposed some limitations and extendibility of this work.

9. References

1. *Joust*, William Electronics, Inc. (1982) [Arcade game]
2. *Borderlands*, Gearbox Software, 2K Games (2009) [Videogame]
3. *The Elder Scrolls: Skyrim*, Bethesda Game Studios, Bethesda Softworks (2011) [Videogame]
4. *Cloudberry Kingdom*, Pwnee Studios (2013) [Videogame]
5. *Geometry Friends*, GAIPS INESC-ID (2009) [Videogame]
6. Rocha, J., (2009) *Geometry Friends - A cooperative game using the Wii Remote*, Master's Degree, Universidade Técnica de Lisboa, Instituto Superior Técnico.
7. *Rogue*, Toy, M., Wichman, G., Arnold, K., Lane, J. (1980) [Videogame]
8. *Elite*, Acornsoft, Firebird, Imagineer (1984) [Videogame]
9. *Minecraft*, Mojang (2011) [Videogame]
10. *Dwarf Fortress*, Bay 12 Games (2006) [Videogame]
11. *Spelunky*, Yu, D., Hull, A., Blitworks, Mossmouth (2013) [Videogame]
12. Smelik, R.M., Tutenel, T., de Kraker, K.J., Bidarra, R., *A declarative approach to procedural modeling of virtual worlds* in *Computers & Graphics*, Vol. 35 (2011), pp. 352-363
13. Liapis, A., Yannakakis, G.N., Togelius, J., Constrained Novelity Search: *A Study on Game Content Generation* (2014)
14. Preuss, M., Liapis, A., Togelius, J., *Searching for Good and Diverse Game Levels* (2014)
15. Shaker, N., (2012) *Towards Player-Driven Procedural Content Generation*, Ph.D, IT University of Copenhagen
16. *World Machine 2*, Schmitt, S. (2008) [Computer Software]
17. *Galactic Arms Race*, Evolutionary Games (2010) [Videogame]
18. *Left 4 Dead*, Turtle Rock Studios, Valve Corporation (2008) [Videogame]
19. Togelius, J., Yannanakis, G.N., Stanley, K.O., Browne, C., *Search-based Procedural Content Generation: A Taxonomy and Survey* in *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 3, Issue 3(2011), pp. 172-186
20. Smith, A.M., Mateas, M., *Answer Set Programming for Procedural Content Generation: A Design Space Approach* in *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 3, Issue 3(2011), pp. 187-200
21. *Lord Of The Rings*, Knizia, R., Kosmos, Fantasy Flight Games (2000) [Board game]
22. *Pandemic*, Leacock, M., Z-Man Games (2008) [Board game]
23. *Gauntlet*, Atari Games (1985) [Arcade game]
24. *Portal 2*, ValveCorporation (2011) [Videogame]
25. *Starcraft*, Blizzard Entertainment (1998) [Videogame]
26. *Counter Strike*, Valve Corporation (1999) [Videogame]
27. *League of Legends*, Riot Games (2009) [Videogame]
28. *Battlefield 1942*, Digital Illusion CE, Electronic Arts (2002) [Videogame]
29. *Magicka*, Arrowhead Game Studios, Paradox Interactive (2011) [Videogame]

30. Cook, M., *ANGELINA*, Available at (last accessed 19/12/2014).
31. Cook, M., Colton, S., Gow, J., *Initial results from co-operative co-evolution for automated platformer design* in *Applications of Evolutionary Computation Lecture Notes in Computer Science,* Vol. 7248 (2012), pp. 194-203
32. Cook, M., Colton, S., *ANGELINA-Coevolution in Automated Game Design* (2012) in *Proceedings of the Third International Conference in Computational Creativity 2012* (2012), pp. 228, Dublin, Ireland,
33. *20XX*, Batterystaple Games (early access released 2014) [unreleased Videogame]
34. Shaker, N., Sarhan, M. H., Al Naameh, O., Shaker, N., Togelius, J., (2013) *Automatic Generation and Analysis of Physics-Based Puzzle Games* in (2013) 2013 IEEE Conference on Computational Intelligence in Games (CIG) pp. 1-8, Niagara Falls, ON, IEEE
35. Compton, K., Mateas, M., (2006) *Procedural Level Design for Platform Games* in (2006) *Proceedings of the 2nd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, Marina del Rey, California, AAAI (Association for the Advancement of Artificial Inteligence)
36. Dahlskog, S., Togelius, J., *Procedural Content Generation Using Patterns as Objectives* (2014) in *Applications of Evolutionary Computation Lecture Notes in Computer Science* (2014), pp. 325-336
37. Dahlskog, S., Togelius, J., *A multi-level level generator* (2014) in *IEEE Conference on Computational Intelligence and Games 2014* (2014) Dortmund, Germany, IEEE
38. Rocha, J.B., Mascarenhas, S., Prada, R., *Game mechanics for cooperative games* (2008) in *Zon Digital Games 2008* (2008) pp. 72-80, Porto, Portugal, Centro de Estudos de Comunicação e Sociedade, Universidade do Minho
39. *Team Fortress 2*, Valve Corporation (2007) [Videogame]
40. *Lego Star Wars: The Video Game*, Traveller's Tales, Eidos Interactive (2005) [Videogame]
41. *PayDay: the Heist*, Overkill Software, Sony Online Entertainment (2011) [Videogame]
42. *Trine*, Frozenbyte, SouthPeak interactive (2009)
43. *Robocraft*, FreeJam (early access released 2014) [Unreleased Videogame]
44. El Nasr, M.S., Aghabeigi, B., Milam, D., Erfani, M., Lameman, B., Maygoli, H., Mah, S., *Understanding and evaluating cooperative games* (2010) in *CHI '10 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), pp. 253-262, New York, NY, USA, ACM (Association for Computing Machinery)
45. *Rock of Ages*, ACE team, Atlus (2011) [Videogame]
46. *Little Big Planet*, Media Molecule, Sony Computer Entertainment (2008) [Videogame]
47. *Journey*, Thatgamecompany, Sony Computer Entertainment (2012) [Videogame]
48. *Sid Meier's Civilization V*, Firaxis Games, 2K Games (2010)
49. *Resident Evil 5*, Capcom (2009) [Videogame]
50. Zagal, J., Rick, J., Hsi, I., *Collaborative games: Lessons learned from board games* (2006) in *Simulation and Gaming - Symposium: Video games: Issues in research and learning, part 2*, Vol. 37 (2006) Issue 1, pp. 26-40, Thousand Oaks, CA, USA, Sage Publications, Inc.

51. *Diplomacy*, Calhamer, A.B. (1959) [Boardgame]
52. *Diablo II*, Blizzard North, Blizzard Entertainment (2000) [Videogame]
53. https://en.wikipedia.org/wiki/Wii_Remote (accessed 8-10-2015)
54. O'Neill, M., Hemberg, E., Gilligan, C., Bartley, E., McDermott, J., Brabazon, A., *Geva: grammatical evolution in java*, Vol. 3 (2008) Issue 2, pp. 17–22, ACM SIGEVOlution.
55. http://ncra.ucd.ie/Site/GEVA.html (accessed 7-1-2015)
56. http://gaips.inesc-id.pt/geometryfriends/?page_id=6 (last accessed 7-10-2015)
57. Gosling, J., Joy, B, Steele, G., Bracha, G., Buckley, A. (2014). *The Java® Language Specification*
58. http://xlinux.nist.gov/dads//HTML/manhattanDistance.html (accessed 8-10-2015)
59. Shapiro, L., and Stockman, G. (2002). *Computer Vision* Prentice Hall. pp. 69–73.
60. Visual demonstration available at http://mathworld.wolfram.com/MooreNeighborhood.html (accessed 9-10-2015)
61. Christensson, P. "FILO Definition." Tech Terms. (August 7, 2014). Accessed Oct 9, 2015. http://techterms.com/definition/filo.