FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Path-based Procedural Approach for Inferring Playable Game Levels

Miguel Geraldes Antunes Mendes

Mestrado Integrado em Engenharia Informática e Computação

Orientador: Pedro Nogueira

Junho de 2017

A Path-based Procedural Approach for Inferring Playable Game Levels

Miguel Geraldes Antunes Mendes

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Jorge Barbosa

Vogal Externo: Luís F. Teófilo

Orientador: Pedro Nogueira

---

27 de Junho de 2017

Resumo

Video-jogos de hoje, quer consideremos experiências de um jogador, ou ambientes multi-jogador com comunidades de grande dimensão, enfrentam um desafio significante no que concerne o que conseguem oferecer ao jogador. Independentemente do tempo gasto num projecto, o conteúdo que tradicionalmente está disponível para o jogador chega a um fim, quer em quantidade, onde já não restam experiências novas para o jogador, quer em experiências, onde um jogador pode continuar a jogar, com o custo de consumir o mesmo conteúdo de forma repetida, normalmente consistindo nos mesmo níveis, personagens, e mais.

A resposta moderna a este problema depende de geração procedimental. Evitando a sobre-utilização de trabalho manual, que pode ter muito custo, tanto em tempo como dinheiro, através de uma esperta utilização de aleatoriedade com inteligente combinação de sistemas e recursos inerentes ao jogo, é possível criar experiências que fornecem ao jogador conteúdo que nunca realmente se repite.

No entanto, a geração deste tipo de conteúdo é relativamente complexa e específica a cada jogo e experiência. Doravante, seria inerentemente interessante ter um método simplificado de estabelecer regras de geração procedimental para esse mesmo tipo de conteúdo, e também facilmente gerar esse conteúdo dentro de atributos e restrições bem definidos.

Logo, a solução proposta consiste no desenvolvimento desse mesmo método de geração de conteúdo. Com a ajuda dos sensores presentes num telemóvel, pretendemos capturar dados relativos a localização, ao longo de mais dados como a quantidade de luz em determinados locais, e proximidade de outros objetos ao smartphone. Uma ferramenta é desenvolvida, capaz de trabalhar com estes dados e, de acordo com parâmetros fornecidos pelo utilizador, gerar conteúdo jogável que consista sempre numa experiência nova e não-repetida.

Com estas ferramentas, um jogo de mecânicas simplificadas aplicáveis a tais estruturas de níveis é desenvolvido para que o método possa ser testado e verificado.

Finalmente, a confiança do conteúdo resultante é estudado cuidadosamente, questionando possíveis utilizadores e concluindo se estas experiências procedimentais serão de facto fáceis de criar, e disfrutáveis para o jogador.

Abstract

Video games of today, whether we consider single player experiences, or multiplayer environments with massive sprawling communities, face a significant challenge in what they can offer a player. No matter how much time a developer spends on a project, the traditional content that is available to its users eventually reaches an end, either in quantity, where there are no more experiences available to the player, or in regards to experience, where a player can continue playing, at the cost of going through repeated content, usually consisting of the same levels, characters, and more.

The modern answer to this problem relies on procedural generation. Avoiding the overuse of manual labor, which can be costly, both time-wise and money-wise, the smart utilization of randomness with the clever matching of in-game assets and systems, developers can create experiences that provide content to the player that never truly repeats itself.

However, the generation of this type of content is still relatively complex and specific to each game and experience. Therefore, it would be inherently interesting to have a simplified method of establishing content generation rules for procedural content and also to easily generate that content inside well established attributes and constraints.

Therefore, the proposed solution consists of developing the means to that very same method of content generation. With the help of sensors typically present in a mobile phone, location-related data is captured, along with other data such as the amount of light in said location, and proximity of objects to the smartphone. A tool is developed, capable of operating on this data and according to user-provided parameters, generating playable content that is always new and a fresh experience.

Along with these tools, a game of simplified mechanics that are applicable to such level structures is developed so that the method can be tested and verified.

Finally, the reliability of the resulting content is carefully studied, surveying possible users and concluding whether these procedural experiences are truly easy to create, and enjoyable for the player.

Acknowledgemnts

I would like to begin by extending my thanks to my thesis supervisor Pedro Nogueira, who through the entire development of the project made sure I was on the correct path, and always provided the correct amount of criticism, even if harshly so, when necessary. That sort of guidance helps us ensure that we reach the appropriately acceptable levels of mastery and success.

My special thanks to several teachers and professors at Faculdade de Engenharia da Universidade do Porto (FEUP), including Augusto de Sousa, who had major success in sprouting my interest in the field of computer graphics and rendering systems, as well as António Coelho and Rui Rodrigues, thanks to whom I was able to keep my interest in the field of video games not only alive, but more than active.

Furthermore, some special gratitude goes to Diogo Barroso and Ana Rodrigues, fellow pursuers of the engineering practice with whom I've shared my place of study and work countless times, and who have often provided their input in my work and objectives, and I've taken the chance to reciprocate.

Finally, I must not forget the constant support of my family, whose motivation keeps me going and working, no matter the cost or hardship.

Miguel Geraldes Antunes Mendes

Table of contents

Table of figures

Table list

Abbreviations and Symbols

API	Application Programming Interface
ECS	Entity-Component System
PCG	Procedural Content Generation
PGC	Procedurally Generated Content
RNG	Random Number Generation

Chapter 1

Introduction

Video games of today face a challenge of increasingly significant importance. Upon the creation of software that composes an experience for a player, one of the challenges that follows is typically the creation of all the content necessary for a prolonged experience. That content may come in many forms, and will usually include all the assets and resources that, together, contribute to a longer video-game and work towards better player engagement and retention. One major groups of assets that contribute to that objectives is the set of levels that developers are able to create for users to play through. Today, level creation faces new horizons, by aiming towards not a great number of different levels for the players to trudge through, but instead going for a possibly infinite amount of levels for a game, of potentially infinite variability, with the added capability of a player never being able to see the same level twice. This more recent method of level generation falls into the category of Procedurally Generated Content (PGC). This accomplishment in video-game design and creation is already available in many ways, but still eludes developers who are not able to programmatically design their procedural levels.

In this work, we look at a new way for a developer to create procedural levels, but through a path outlined by the user in the real world, instead of digital tools or algorithmically.

1.1 Context

The problem of level creation for game was solved a long time ago. Many starting developers will first attempt at hardcoding their levels into their game's source code, manually defining where game objects should be, how a certain object's collision should work, among many other features that belong to typical elements in a game level. Eventually, these users will find that for larger, more complex levels, this task becomes a very time consuming chore, where the amount of time invested is hardly worth the result, and small but necessary changes will require a long look and search in source code for the change to take place, and for verification that this change

achieves the intended results. Not to mention, depending on the used technology, this method usually implicates the re-compilation of all the code, which will also bring about large waiting times for the implementation of even the smallest changes.

Of course, with these obvious obstacles in place, useful solutions would soon come about. The most obvious solutions that followed would be level creation tools that would facilitate the placement of objects inside a game world, and definition of these objects' features. With these new level editors, changes would be realized not in source code, but instead in files that would stand outside of the code-base, only to be loaded by the software at runtime, not only facilitating slight changes and rapid verification of results, but these tools would also bring about much faster production of larger quantities of levels, on account of a facilitated creation method.

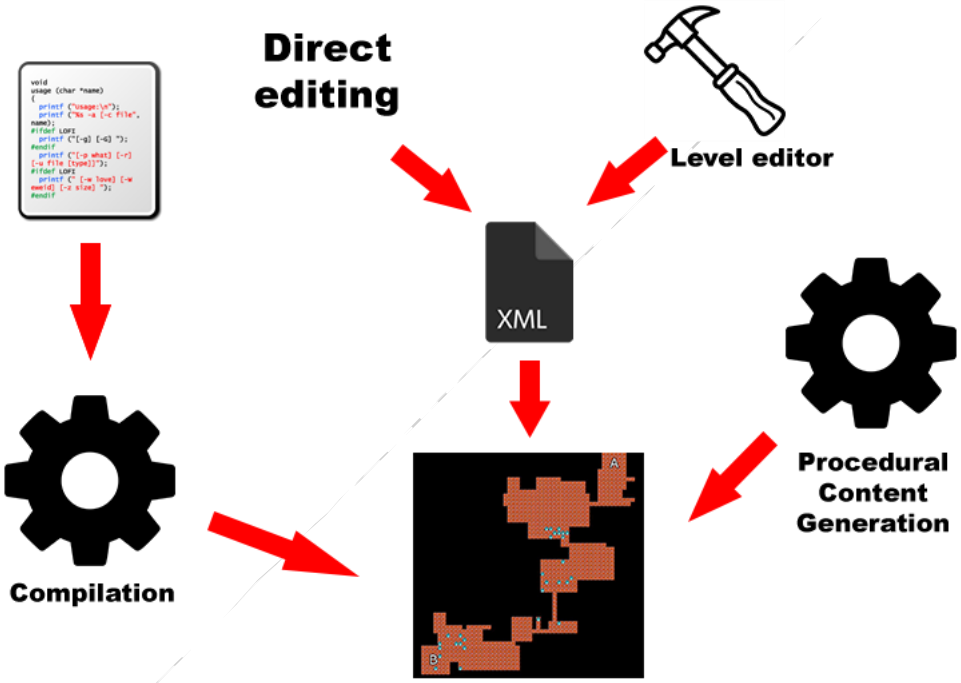

Figure 1: The three most common present methods of level creation

With an existent capability for creation of large amounts of levels, given enough time and creativity for the developer, a new objective arises, in which a creator intends on not simply providing a user with a large playable level base, but instead and infinite supply of experiences where, through Procedural Content Generation(PCG), a player is able to experience an insurmountable number of levels, and still never experience a level twice, or ever playing through a level that has been played before either by that user, or by any other.

This problem has also been solved in the past. Algorithmically, developers have been able to create levels that are always different from one another, but always playable. As should be

obvious, most time in the level creation stage of games using this method is focused on the development of the proper algorithms that create the intended types of levels. These procedurally created levels will range from simple, two-dimensional labyrinths to fully developed three-dimensional environments.

1.2 Motivation and Objectives

Taking the current scene of level creation in mind, we are left with a vacuum of tools that are usable by a developer, where parameters can be easily provided and the creation of procedural levels can be fast, while still maintaining requested features. Thus, there is an opportunity to make a method for procedural level generation where the conceptualization of the level creation procedure would be simple enough for any developer, both with a technical background, or another non-technical background where the acknowledgement of PCG algorithms and creation methods aren't a known factor.

Therefore, we see here a chance to develop a method that simplifies the procedural level creation process. Firstly, assuming a smartphone is the most accessible device to anyone at the current day and age, we focus our efforts on using that device to meet our goals. Secondly, we consider the fact that this device is capable of capturing a varied amount of several types of data, depending on the sensors that a smartphone has. Finally, assuming that a smartphone can determine its own position in 3D space, there is a possibility of a pipeline for insertion of relevant data captured by a device, a correlation of that data to a device's location, thus making the association of parameters captured in real life, including location, to the digital, three-dimensional space, possible.

Let us keep in mind that, with the possibility of the capture of the device's location in the real world, and to avoid the necessity of capturing too many points, which would, in a way, invalidate or insert undesired hardships in the method, we intend to keep the solution based in paths, by which we mean that we intend to have the input data provided by a user to be based on a path drawn by the developer, but in any exterior space.

And so, the aim currently consists of:

1) Developing a tool capable of working on smartphones, capturing any type of data necessary, from any of the device's sensors including, perhaps most importantly, position data.
2) Developing procedural generation code capable of accepting several parameters in order for the user to customize the experience however intended.
3) Applying these procedures to an end result, in this particular instance a simple video-game, capable of showing the results of the process, so that a developer can use and verify the method, and a player can experience the final result.

4) Verifying the validity of the proposed solution through user testing, while asking for relevant feedback and measuring relevant data that'll support any eventual usefulness or superiority of the new solution.

In what concerns the first objective, we intend to make use of the Android API to develop and application that will run in most Android smartphones, which will undoubtedly give us access to any Android device's sensors and the ability to output those in a useful format to use afterwards.

Regarding the second and third objectives, an already existing and premade 3D game engine will be used for the rapid prototyping and development of a game, simple enough to implement any desired features, while simultaneously allowing for the meeting of deadlines.

The final objective will be met through utilization of the final method by testers, which will eventually attempt to create levels using the developed tools. A set of instructions and questions will be provided, and from these results we will be able to assess the validity of this attempt at a new solution of procedural generation of levels through application of physical paths.

1.3 Dissertation structure

Beyond this introduction this dissertation has five more chapters. In chapter 2, existing bibliography regarding the field is revised and analyzed, and some important conclusions from what already exists and what is being worked on are withdrawn. In chapter 3, the method proposed as a new solution to the addressed problem is introduced. Chapter 4 serves to delve into the details of how the solution was developed and how it works. In chapter 5 the required testing that was performed is described and relevant results from said testing are analyzed. Finally, in chapter 6, work is concluded with an overall analysis and a final rumination on what comes next.

Chapter 2

Bibliographic revision

In this chapter current technologies are reviewed, what currently exists and what is being tried in the world of PCG. Chosen methods will be justified with the research that has already been done on the subject, moving forward, and reasons will be given to why this solution is a viable option, in the middle of what alternatives are already available.

2.1 Introduction

As one can surmise from already existing work in the field, it is clear that, in general, most efforts regarding PGC seem to be always focused towards some more specific context. There never seems to be a more generalist solution to procedural levels. Instead, most solutions tend to be aiming towards either the simulation of already existing environments, or go directly towards the specific goals of the game in which they will be applied.

Firstly, we shall take a look at the various types of algorithms that exist in the PCG scene, and how they are usually applied.

Then, we will take a look at recent work towards the application of several techniques to simulate the most common environments seen or wanted in video games.

Finally, we'll have a look at more abstract details that are usually wanted in a procedurally generated level. These details are usual quite important, being more relevant towards the playability factor of a game's level. They might not have to do directly with the way a level is created, but certainly appear as useful aids to ensure level playability, proper amount of difficulty, among other important elements that enhance player experience.

2.2 The current state in procedural content generation

2.2.1 Categories of PGC

With the advent of procedural generation of levels for increased content for players, several algorithms, some already existing, others created purposefully for the cause, are beneficial to meet these goals. However, as already mentioned, these algorithms usually carry different aims, and as such, fall into separate categories. The algorithms used in PCG are many, but can be categorized as follows.

2.2.1.1 Map generation

These algorithms fit the purpose of creating maps, either in two dimensional spaces, or in three dimensions. Some notable examples are discussed here.

Cellular Automata, for example, work on grids of cells. Being automata, these cells will feature states, and transitions between several of these states, transitions which are subject to certain rules. Taking place in any desired dimension (from one dimension, to three dimensions, to beyond), these rules usually involve taking into account states of either neighbor cells, or the cell itself, or perhaps both, and then deciding a transition to a new state while taking the rules in mind. One very famous example application of this sort of algorithms would be Conway's Game of Life. There are many ways of initiating these grids, in order to provide a useful starting point from which the instated rules might yield interesting results. They can either be random, or some sort of noise can be applied to a base, to achieve perhaps a more natural beginning, or the initial state could already be the result of applying the rules to a set of previous states. Cellular Automata is usually applied in the creation of dungeons and caves, seen as the process usually results in fairly organic results.

There are sets of methodologies applied only to dungeon generation, even though Cellular Automata can be an option. Most approaches to the generation of dungeons takes one of two paths. Either a maze is formed, with certain paths converging into rooms that yield a more natural look to the overall structure, or several rooms are created and then linked by paths as needed, with a certain level of randomness, to always preserve originality and naturality. Two problems that usually arise with the generation of dungeons are the necessary tunneling algorithms that can rise very high in complexity in order to insure that the entire dungeon is playable from beginning to end, sometimes requiring that a whole generated dungeon be castaway to give way to one that meets this requirement. Additionally, decisions in how the geometry for the dungeon is generated might lead to problems. Many types of geometry might cause, for example, for two adjacent rooms to be connected to one another instead of discretely separated. Usually these problems will either results in readaptation of used methods or simple regeneration of dungeons that circumvent these issues.

Continuing with the relevance of procedurally generated levels in three dimensions, height maps might sometimes be used simply as input data, turning parameters displayed in a two-dimensional fashion into a three dimensional surface. Obviously not being able to create structures like caves, height maps can create structures like fields, or even help in the creation of textures, applying color values to each height value and using simple interpolation, for example.

L-systems, consisting of formal grammars that work on symbols, using them to create string with basis on certain production rules, these algorithms can work towards the procedural generation of geometric structures resembling yeast and fungi. These methods are mostly applied to generate many different types of plants and vegetation.

2.2.1.2 Sequence Generation

Certain algorithms are applied only in sequences, either of values or structures.

Markov chains, for one, take samples of linear samples, usually in very large numbers, to increase success rates. These linear sequences might be names, melodies, or even sets of colors. From these samples, tables of probability are constructed, so that after each unit of value, be it a character, or a musical note, the algorithm can take those values of probability and decide what value to generate next. As such, this algorithm is often used, for example, when we wish to create many different names or songs, which might not exist or be real, but will definitely feel as such.

Random number generation, as one might surmise consists on the generation of numbers, usually in a sequential manner. These numbers will usually be random, as the result of the generation will have an infinitesimal change of being guessed. This process can either be used in generators that simply produce numbers, or can serve in hash functions that randomly output a value based on input, but always produce the same output, given the same input. Random number generation takes place in many if not most PCG algorithms, seen as most of these use randomness to add variability and a feeling of naturality to final results.

Random number generation will many times depend on a seed. This seed usually consists of an input value, commonly a numerical value, which needs to also be random to allow for a generator to also be truly random. Usually the manipulation of this seed can result in sequences that always carry particular but constant features.

2.2.1.3 Ontogenetic

Ontogenetic algorithms will try to recreate a certain structure, environment or result, without going through the necessary steps to get there. Essentially, these algorithms try to create a believable final result off the bat, therefore not constituting a simulation. Cellular Automata, dungeon generation, L-Systems and Markov chains can fall into this category as well.

Fractals follow mathematical rules in which a main structure will repeat itself upon following the same rules that create it. The result is an infinite set of structures similar or equal to

the original, but of different scale. Not only are fractals often used in PCG due to its similarity with natural processes such as the growth of plants and the erosion of soil or socks, but fractals also allow for the mathematical definition of and infinite amount of detail, since at any scale it is able to produce new structures that complete the pattern, recursively subdividing details.

Perlin noise applies noise to a set of data, but works in layers. This noise is often randomly generated, but its behavior might change according to the applied frequency. Higher frequencies will have more points of effect but will be less pronounced, while lower frequencies will do the opposite.

A Voronoi Diagram is a method of subdividing space into Voronoi Cells. The resulting set of cells are useful, since they look like a natural formation, they can be used to represent lands, vegetation, actual cells, among many other natural and already existing bodies. Explaining how it works in two-dimensional space, it starts with a set of points, usually randomly distributed in space, and for every pixel in the space, decides to which point that pixel belongs. The point will usually be the closest one to that pixel. The resulting diagram, in this case, is two-dimensional space subdivided into several areas (cells), each corresponding to a point that was initially provided.

2.2.1.4 Teleological

As opposed to ontogenetic algorithms, teleological algorithms attempt to reproduce already existing structures and environments by simulating the very processes that would naturally lead to those final results. One might argue that this method might logically reach more believable and accurate results, but it all comes down to implementation. Logically, Cellular Automata, genetic algorithms and Markov chains would be fit examples of teleological applications.

Random scattering with the objective of procedural forest generation is another fine example of a teleological algorithm. By simulating the growth inhibition produced by trees that prevent any sprouts around it from retrieving enough nutrients, trees can be randomly placed, or used in a grid with simulated deviations, through spring mass systems and several iterations, with the result being a believable and natural-looking tree distribution.

Another teleological algorithm consists of simulating erosion in objects or surfaces by simulating the effects rain drops would have in those bodies. The rain drop algorithm accomplishes this by altering height maps, removing height by very small amounts in the location where a rain drop lands, simulating the drop travelling down slopes and surfaces, if the zone where it landed requires it to be so, and slightly increasing the height value where the rain drop stops after superficial movement, this way simulating erosion. For more solid objects, such as stones or rocks, these final height map additions can even be ignored, since we can surmise that removed mass, in a stone, would rarely return to the original body, or we can presume that the rain drop would, in any event, let go of the rock itself.

For a more complete look on Procedural Content Generation, or a more in-depth look at Ontogenetical versus Teleological algorithms, check [Pcg17].

2.2.2 Environments

In regards to procedural generation of level environments, De Carli et al [COB11] establish a very important difference between all the used methods for Procedural Content Generation (PCG). That difference stands between *assisted* methods, and *non-assisted* methods, being so that assisted methods are the ones that heavily rely on the designer or programmer's input for the production of the intended results. This required input is often based on provided equations, or picture data, drawn paths, or other significant data that fundamentally drives the final result in some important context. These methods are usually applied when the user requires some levels of authorship in the environment or level. The advantage of these methods, in this case, comes from the significant decrease in level creation time, where a designer, for instance, can finally spend more time generating the overall desired layout of a setting, instead of dwelling on environmental details which can be programmatically handled and taken care of by a procedural content generator. Conversely, a non-assisted method is very much independent from whoever uses the method, requiring no input or control, or sometimes requiring very minimal influence from the user. This typically translates into simple adjustment of values or parameters already inherently belonging to the method or algorithm. This sort of method would be more appropriate on content that has to be procedurally generated, but will largely not be interacted with by players. Additionally, this sort of work becomes even more useful when that set of non-interactable assets is in high volume.

2.2.2.1 Terrain

In regards to terrain, it's described how there is possibility for procedural generation of terrain through the single use or even combined use of noise, L-systems and fractals. Additionally, one could also tinker with height maps, in combination with erosion simulation, or perhaps use splines to control distortions of such entities like large land masses.

On assisted methods, one can use a vector-based model, with the assistance of control curves for avoiding unpredictability, using parameters within such as elevation, noise and gradient, permitting the generation of several types of landscapes, such as mountains, hills, or deserts.

One other assisted approach would consist of constraint-based generation, where the user needs to input an example of terrain data, usually regarding height patterns. By providing a sketch of the intended result, small patches are retrieved from the sample data for the analysis of its features, through Profile recognition and Polygon Breaking Algorithm. Afterwards, these features are applied to the provided sketch, resulting in a final environment, resembling the provided

sketch, but retaining the features of the initial sample data. This allows for a somewhat controlled design of a rather expansive piece of land, mountain or canyon, for example.

One more assisted method regarding terrain would be a genetic algorithm, but instead of overwhelming the user with the numerous lower level options that such an algorithm usually imposes, this method simply provides the user with several result candidates, from which the preferred ones can be selected, and detailed algorithm options may remain solely on background, only to be altered or acted upon if so desired, though rarely.

On what concerns non-assisted approaches regarding terrain, it is possible to use agents of several types to walk along an environment and construct its features accordingly. Thusly, a mountain agent would create higher height areas, with rockier textures, for example, while a beach agent would smooth the terrain along the coastline, and apply sand textures

Another non-assisted approach consists on the application of Perlin noise to a matrix structure from which the final environment would get all the required data to be formed. Relying on its neighbors, each point in the matrix would match a particular piece of the environment. This allows for the generation of terrain featuring mountains, coastlines, and beaches.

One other non-assisted method consists of the aperiodic tiling of rocks, using a corner cube generation algorithm, which ensures that the final rocks are touching each other. These aperiodic tiles are cubes filled with small rocks, generated by several points in space and associated with Voronoi cells, which are eroded to form the final geometry. Lastly volumetric models are filled with the appropriate rock tiles. This method, however, is much more useful for the generation of piles of rocks instead of actual full terrain.

2.2.2.2 Roads and Rivers

Although pure terrain can be quite realistic and interesting in its procedural nature of always creating new content, more so in environments such as deserts and the like, some additional features are able to introduce a more natural and believable look the whole scene, and this is where rivers and roads come into play. These can usually be obtained through methods such as tensor fields, interactive synthesis of urban street networks and template patterns combines with Voronoi diagrams.

An assisted technique, that gives particular attention to how a river is naturally created according to its environment and terrain, uses natural systems, which takes into account a river's footprint, shape and procedure, in order to create a river whose nature is sound, according to all the natural elements present in the relevant area. This becomes extremely more useful when a river needs to be changed, or an entire section of environment needs alteration. In the former case, a user can simply adjust its control points to ensure a new river path. In the latter case, adjusting the environment to the new desired outcome and reapplying the river generation procedure already ensures a new river, adapted to the new environment, with its alteration.

A non-assisted technique regarding roads could be based on a weighted anisotropic shortest path algorithm, through which the user simply defines the cost functions which, through parametrization, are used for the evaluation of the line integrals of such functions along a generated road in a discretized grid. Once the paths are defined, they are created and accordingly textured. Depending on the systems or roads interconnectivity, bridges and tunnels can be set up to accommodate several road interactions. This typically results in a quite natural-looking road that can be applied in a natural environment.

2.2.2.3 Cities

Contrasting the necessity to build open, natural landscapes, comes the desire to also be able to create sprawling cities. The procedural component provides further interest, seen as through it, an entirely new city can be created with each new generation. The generation of cities interweaves itself somewhat with road generation. This should be obvious, as the two are always interconnected. Since cities and their roads result from geographical and cultural reasons, it is often difficult to algorithmically guess how they ultimately look. Therefore, it is only natural that most methods for generating truly realistic city/road setups are assisted requiring input from real examples in order to match style and feel.

One assisted method for this would be in the mix of Bezier curves and actual city maps. Matching curves with existing roads, an algorithm finds important features of these curves and can develop on them, creating more content of similar, feel. With the resulting curves, everything else such as sidewalks and buildings can be modularly placed accordingly, to generate a realistic city, regarding input data.

On other assisted method for road generation in cities is like a simplified L-system, where a random travelling algorithm connects points and uses templates to define how the roads grow, being executed in three steps. From specified points, primary roads are generated. Using parametrization mapping, templates are applied to the area and secondary roads are generated. Finally a third level of roads is acted upon. This method is assisted in the sense that a user can take quite detailed control of how new roads are generated, and how densely. The whole process is quite interactive, and possible results reflect that.

We even have an assisted technique for the generation of ecosystems amid a city, through an interactive procedural system. First an urban model is generated from socio-economical and geometrical simulations. Afterwards, manageability of plants is rated throughout the city and a procedural planting algorithm decides where plants are born, after which the plants compete to determine which will remains and in which quantity. As you can tell, this brings added value to a simple city generation algorithm, by bringing the more natural aspect to it.

Additionally, there are other engines capable of generating these cities. These engines act upon provided data, usually spatial databases or statistical and geographical input data, from which they can either generate the cities, and their associated road networks, or vice-versa. Non-

surprisingly, many of these engines also use some forms of L-systems. Some of these actually act through generative grammar and shapes, though these grammars are more relevant in the following section.

2.2.3 Dungeons and Labyrinths

Keeping in mind the sort of solution that is looked for is quite based on paths, the pursuit for the generation of an open environment becomes somewhat intractable, largely due to scale. The amount of required input data of a path would simply be too large to generate and appropriate open environment. Therefore, one must focus on solutions regarding structures that are more tightly associated with those paths. In that regard, it is logical to look into dungeon generation algorithms as a correct trajectory of investigation.

Typically, dungeon generation is comprised of three elements:

1) Representational model: a simply way of representing the dungeon structure. This can either be achieved by a simple form of graph, or by programmatical data structures themselves.
2) Method for achieving the representational model.
3) Method for generating the dungeon geometry from a previously generated representational model.

Since the transition between the representation model and the actual geometry is quite trivial, most research is focused on the generation of the representation models, which are fundamentally what matters.

Current dungeon generating methods can be classified into the following sections.

2.2.3.1 Cellular automata

Generating dungeons through the use of cellular automata consists of having the representation model as a discrete grid of any finite number of dimensions. By having each cell in an initial state, and with each cell being aware of the status of neighboring cells (which neighbors a cells is aware of, depends on implementation), particular rulesets may be applied to each and every cell, which operate on its current status and the status of its neighbors. Having this done multiple times to every cell, often results in patterns. These patterns will depend on the applied ruleset.

This method poses some restrictions. For example, it is impossible to determine before-hand how many rooms the resulting dungeon will have. Even if there were rooms, it would be impossible to decide which overall structure they would take when interconnected. The separation between this kind of generation and gameplay is clear, and makes this type of dungeon somewhat unpredictable, becoming even more so unreliable.

2.2.3.2 Generative grammars

Avoiding the implicit and sometimes incontrollable randomness from cellular automata, generative grammar give a somewhat more balanced equity between levels that are always different, and a constant structure that is defined by the user.

In this sense, grammars are abstracts ways of defining this structure. Commonly defined through graphs, they can either defined and constant layout, or even a sequence of steps a player might have to go through one or several times until reaching their goal. So assuming that a certain grammar defines a structure A, any method that acts upon that grammar will ensure that A remains, but will apply the necessary randomness for the creation of interesting and new levels that take that constant structure in mind, although such happens in the dungeon geometry generation stage, considering the grammar itself to be the representation model.

The advantages of this method is clear, as the separation between levels and gameplay almost disappears, especially in case of grammars that pay particular attention to gameplay itself.

2.2.3.3 Genetic algorithms

Discarding the common details of genetic algorithms, such as how chromosomes are defined and how mutations take place, since this can be achieved by trivial translation of proper data structures into strings or binary, these algorithms provide significant progress in specific elements of level generation.

Firstly, regarding simple level generation, is possible to generate tree structure for dungeon crawler levels. Fitness functions will always depend on implementation preference, of course. But with the appropriate functions, many interesting patterns might emerge. With a fitness function preferring tightly packed rooms connected by hallways, for example, results are rather elegant, with a pleasant placement of rooms.

Secondly, genetic algorithms provide a way of mingling story with how the level is created. Through space trees, in which a node can either be a relevant plot point or a “bridge”, genetic algorithms can take action in these trees, optimizing them for the best player experience, as long as they’re given the appropriate fitness function, typically one that can evaluate common traits and patterns of good story-based level arrangement. However, one must mind performance with this method, as If no proper solution are found, entire space trees can be outright discarded.

Keep in mind that all these methods are highly parameterizable, therefore being quite flexible in their results, adaptable to several needs.

2.2.3.4 Constraint-based

Possibly the only method expressly created with the generation of 3D levels in mind, constraints act upon a topology of undirected graphs and sub-graphs in order to create interesting and relevant levels. The randomness here comes from all the possible levels one can obtain from

these graphs while still respecting these constraints, of course. Like cellular automata, however, this is a technique that creates a serious disconnect between possible generated levels and gameplay, unlike some previously mentioned levels which actually take gameplay and even story in mind upon generation.

M.E.R.C., for example, is a game which construct's its dungeon-like levels by basing itself in Hilbert curves, creating a layout from altering a subset of Hilbert curves and adding some shortcuts. Subdividing these curves into sections allows the algorithm to randomize the level generation in interesting ways. By making the subdivision into repeatable parts, a developer can create several assets that would suit those repeatable parts, since those assets will always associate well. In this way, even with an exact match of another level, the variation in different parts for each subsections will always yield a different final level. More about M.E.R.C.'s level generation can be read in Graham Davis' publications about the procedure [Dav17].

There are many approaches, related to the ones described above. However, most of them either stem from similar practices, deviate in the amount of control that is provided, or consist in hybrid mixes of two or possibly more methods of dungeon generation.

2.2.4 Abstract design preferences

There are more details concerning the generation of procedural levels that need to be taken into account for the proper creation of levels that are playable, and can generate enough interest in the player through proper variability, proper level of difficulty, and possibility for conclusion of the actual level, with the aim to eliminate any possible frustrations.

As such, several characteristic turn out to be quite useful, some even plain necessary for the proper creation of levels that may be always different, but still appropriate as a gameplay experience.

2.2.4.1 Feasibility

The first concern when creating a level procedurally should go through making sure that the level itself is feasible, which means that the level should be playable from beginning to end. In other words, it should be possible for the player to start playing the game, in that level, and reach the end of the level itself, making a win condition a possibility. In more complex games, such as platformers, both two and three dimensional, this problem is usually solved through the creation of an AI that possesses all the capabilities a real player would and, with knowledge of rules and how the game environment works, attempts to complete the level on its own. Assuming this AI is built with the ability to beat any level that is possible to beat, we can reach the conclusion that

if the AI is unable to complete the level, then it is considered invalid, and either the rules for level creation should be changed, or a new level that meets the AI skill in completing it should be created.

A rather complete look at how an AI demonstrates its ability at solving prodrually generated levels for a platforming game can be obtained at Jordan Fisher's publications on the subject [Fis12].

In games with simpler layouts, however, such as maps composed of open spaces, grids, labyrinths, and other similar structures, different algorithms of a simpler nature or pure path-finding algorithms can be applied to ensure that there is a path that goes from a start point to an end point with no impassable obstacles in between.

2.2.4.2 Player engagement / interest

A procedural level generator can be able to produce a more than sufficient amount of levels, and they may be playable from start to finish, and yet be unable to arouse the player's interest or actually engage any sort of user to any continuous interaction with the game. This often results as a consequence of bland and uninteresting levels where its structure, though varied from other iterations, carries no substance that incentivizes exploration, or different courses of action from a user.

This lack of generated interest is usually fought through more intricate structures, wherein a player suddenly needs to perform decisions either in the moment in a mental model in order to be able to beat a level. This can be done, for example, by diverging from linear layouts in a map, for instance. Should a level's representational model consist of a graph, a linear graph does not present much a problem in the mental model of who solves it. A more intricate graph that provides branching, multiple decisions and ways of solving the overall problem, suddenly might make it much more interesting not only for a user to defeat the current challenge, but perhaps even looking forward to any potential future challenges that might be presented along the way, while playing the intended game.

2.2.4.3 Level of challenge

Depending on the challenges that a player faces when attempting to complete a certain game level, the difficulty at completing said challenge might vary by a significant amount. This of course will depend on the type of challenges, how they interact with the player, how they interact with each other, how many there are in a level, among many other decisions that a creation algorithm will keep in mind while creating a level.

All these elements are usually controlled by parameters that are set before or at the time of the level creation. The hardship in controlling challenge in procedural levels through these parameters is figuring out the correct values for the intended amounts of difficulty. Usually,

playtesting will become a useful tool at this stage, since no AI would be able to correctly assess the human perception of difficulty of a level. Through extensive playtesting, it is possible to define what parameter values correspond to an absurdly easy level, and what values correspond to a very difficult, but humanly feasible level.

Thankfully, as soon as these value are determined, the correct values for any type of difficulty can commonly be found through interpolating of the two previously found values for each extreme.

2.3 Summary and conclusions

There was a determination of what is available, in what concerns procedural generation of levels. Much of procedural generation today consists of the utilization of algorithms, depending on the situation, be it the simulation of some sort of environment, structure or level type. There are many types of algorithms that suit each of these needs, and there were put into categories.

Some of these algorithms try to generate sequences, some generate maps, and some simply focus on generating natural-looking structures that resemble real-life counterparts. Several examples were given on how many different algorithms apply to structures like terrain, rivers, roads, cities and, perhaps most importantly, dungeons.

We went a step beyond and analyzed what work is usually added to procedural level creation to ensure proper quality in output, and how these quality-adding elements serve us towards a better experience for the user, balancing elements like feasibility, player interest, and level of difficulty and challenge.

All these methods will rely on input data and parameters. On one hand, parameters are very important to the control of how levels are generated, and a change in parameters can greatly influence a level in terms of structure, shape, playability, and level of enjoyment. Input data, on the other, can simultaneously be part of these parameters, but can also be the foundation through which the levels are built. A level that tries to take a particular shape might, for example, take a few important points as input data, points which will probably determine that very shape, the same way forest generating algorithms randomly generate their own input data, by providing initial points of reference from which to create a realistic distribution of trees and other types of vegetation.

However, let us consider the current procedural level generation scene. Firstly, there aren't many tools available for public use. The ones that exist for public use fit very specific purposes, and even the ones that aren't available for public use are developed particularly for the games they serve. Since most solutions are based on algorithms that are applied as necessity dictates, and such can be performed in many different ways, depending on the game and the developer, we will most likely never see a generalist tools that serves most or any purposes for wide-spread

procedural level creation, and in the near future, it will continue to fall to the developers who can implement their desired algorithms to develop these methods.

Additionally, and because of how data is usually provided to level creation methods, no current tool or method uses smartphones as the definitive source of input data for procedural generation algorithms, and no current methods includes smartphones as part of the procedural level creation pipeline. Thus a vaccum is found that may be filled with a new proposed solution.

Chapter 3

A Path-based solution to level generation

To satisfy the lacking of a procedural level generation procedure that utilizes smartphones as the primary source of input data and method of control of level parameters, thus making the smartphone part of the core process of level creation, a solution is proposed that incorporates the smartphone in early stages of the creation process, once a developer has a defined algorithm that creates the intended types of levels, given the sort of input data that a smartphone can provide.

What is trying to be renovated in the usual procedure is the method of inserting input data into the pipeline which, instead of using digitally created data, either intentionally or randomly, uses data produced likely in an outside environment, through the use of a smartphone.

This perhaps would feel counter-intuitive to any seasoned level creator who is used to tools keeping a developer in a desk, mass producing levels with most common methods, either programmatically, to create algorithms and manually define input data, or through the use of more common level creation and editing, where procedural generation is much less of a feature. However, it is foreseen that the proposed solution would not only benefit these seasoned creators, who might enjoy a few minutes or hours outside of the office, while still doing productive work, but also creators of less technical background, who would perhaps feel more able of creating their customized input data in a way that would perhaps feel much more natural and intuitive.

Keep in mind that in the context of this work, the basis of a “path” for level creation will continuously refer to a path that a user can create or walk in any ample space or environment with a certain smartphone that is intended to capture the necessary sensor data.

3.1 How it is intended to work

The proposed method for creation of input data, control of relevant parameters, and ultimately, the procedural generation of playable game levels, first makes a few notable assumptions:

1) The game in question already has an algorithm that generates levels for itself procedurally, and this algorithm is capable of taking input data and parameters into account in order for any creator that is responsible for the creation and control of said data and parameters to be able to do so and feed them into the algorithm, so eventually a result can be obtained.
2) Input data is mostly based on positions captured, somehow, so that it is possible to provide and overall shape and structure to the level that is to be created.
3) Parameters would be based on data captured by other means, in this case through the use of the remaining sensors in a smartphone, and these parameters would be able to influence several aspects of the created levels, to that the relevant abstract design preferences can be controlled and maintained, such as level of difficulty, for one.
4) As levels are supposed to be based on a path, the generation algorithm produces output which fits this description, in this case levels that would take the shape of that path.
5) Regardless of provided data, generated levels always fulfill the basic requirement of being playable from start to finish.

As such, here is the intended level creation procedure, as envisioned:

1) The user, present in an ample space, most likely outside, creates a mental model of the level that is intended for creation.
2) Carrying a smartphone in hand, through its use, the user defines a path, using its location.
3) While using the smartphone and defining the path, the user makes use of the smartphone's several sensors, as desired, to control any previously defined parameters that are relevant to level creation and control.
4) The user will then carry the created data over to the level creation tool, in this case on pc, but adaptable to any desired platform, which will create an infinite amount of levels based on the provided data.
5) If the user so wishes, the level can be immediately played.

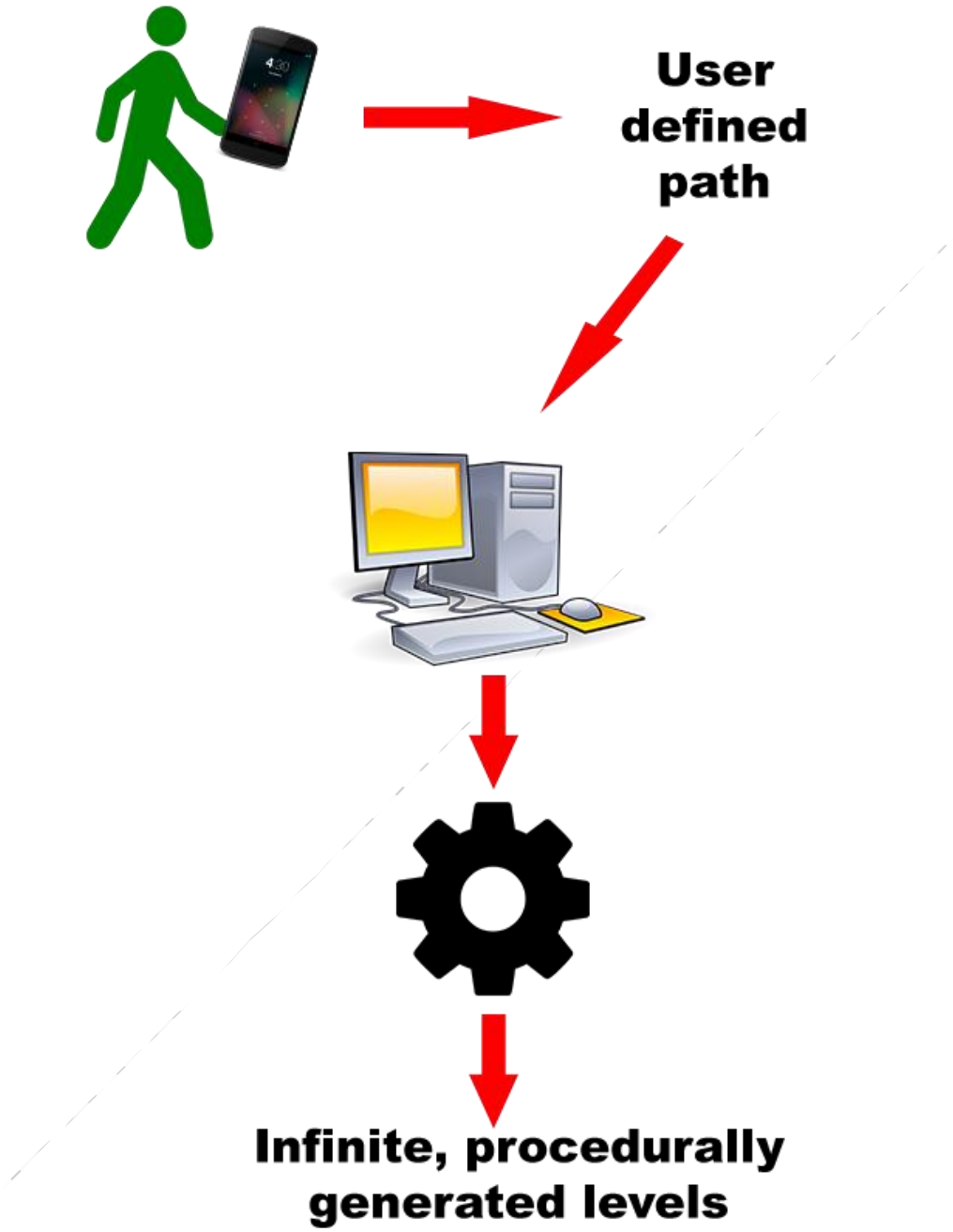

Figure 3.1: Design for the app's role in the pipeline

Now the details of how the proposed solution was developed for this work are carefully described. This will include the several pieces of software that were created specifically towards the experimentation of the proposed method. Three elements in particular were the focus towards the development of the solution. An example game was necessary for the required proof and

experimentation of level creation process, a procedural level generation algorithm was written for this game to be able to show the output and make it playable, both to us and any other potential user, and an Android application was developed, so that a smartphone could be used as the primary method of input data gathering and parameter control.

3.2 The design

The way the new method for procedural level generation was designed is separated in two stages, input data and parameter creation in one stage, and the actual generation of how many levels the user wishes in a second stage, utilizing the data that was created in the first stage.

In order to apply the solution, a game is needed in which a path-based level can be created and played, to verify the method. Since these paths can take any shape in the three dimensional space, it is then chosen to create a game where levels can take some sort of shape in three dimensions.

A design that seems appropriate involves levels that are composed of several segments or corridors that can go in any direction. This would allow us to take any path-shaped data and apply that so those segments could take that general shape.

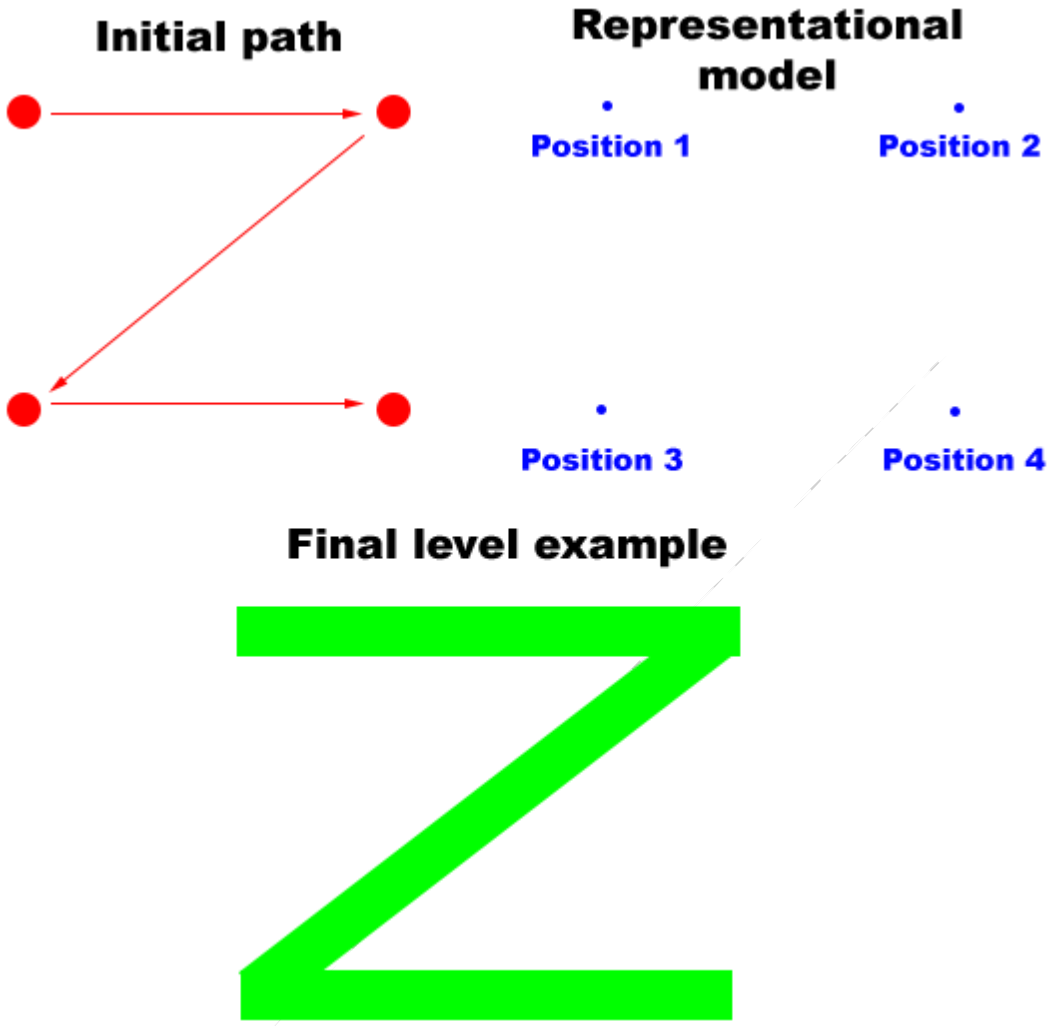

Figure 4.1: Translation from real world path data to a representational model, to a final playable level shape

Thusly, a system can be determined for the levels that follows the necessary features of dungeon generation. The proper representational model can immediately be thought of, as a set of segments or corridors that go in any direction in 3D space can be achieved by a list of positions in three-dimensional space, to determine the points in which each corridor ends and another starts. Such a representation model can be achieved through the capture of positional data using the smartphone and converting that data into useful programmatical data structures in the intended game application. Finally, with the representational model defined and solidified, all is left is to generate the geometry that will make the actual level. Simplicity in implementation is desired, so these corridors can simply be rows of blocks that extend in the required direction.

3.3 An example game

As previously defined, in order to accomplish a level that is based in a path, there is a drive to make levels composed of segments that can go in any direction in 3D space, and to this goal segments are made to be composed of rows of blocks that extend in the intended direction.

The reason the use of rows of blocks is chosen instead of continuous corridors is to make movement in the game discrete, instead of continuous. This simplifies not only the game but several verifications that would be required to ensure that levels meet the desired requirements of feasibility, for example. These checks lose complexity as they no longer have to run against continuous space, but can rely in much simpler procedures that can simply operate on grids to check these requirements and enact any changes if necessary. Lost complexity ensures that the game can be realized and developed quickly, and that more effort can be spent in the development of the remaining elements of the entire proposed solution.

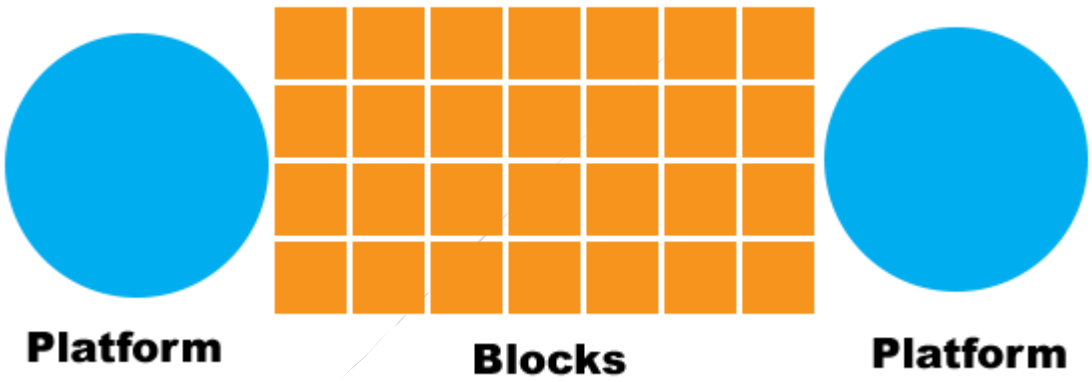

Figure 4.2: Two dimensional representation of a segment

In between each segment, a platform is placed, not only for clear definition between segments, even if they are collinear, but also serves as space of transition for the player, as some adaptation will be required between different inclinations. However, most importantly, these platforms will correspond to the three dimensional position values that define and compose our representational model. Therefore, to these platforms not only can position be assigned, but so can the various parameters that will control several features of the levels that will be created.

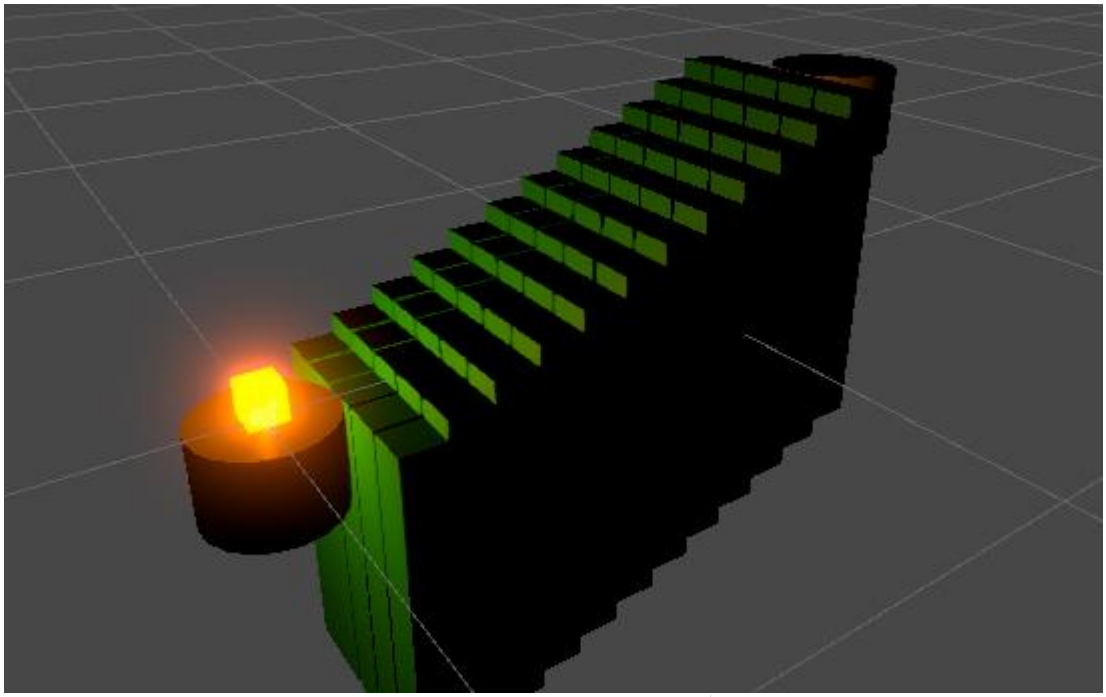

Figure 4.3: Example of segment with inclination

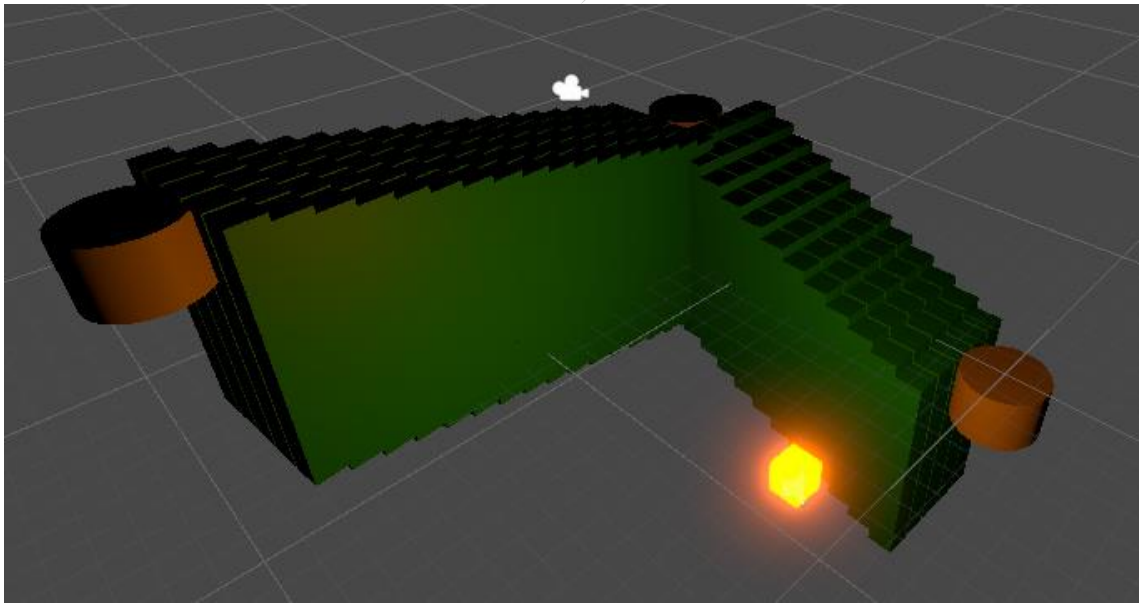

Figure 4.4: Example of multiple interconnected segments

Gameplay is made particularly simple here as well. Taking advantage of the grid-like nature of the segments in the level, the player controls a cube-shaped character that moves discreetly between each grid cell, on top of each block. Moving along the several roads, the player is able

to progress along the several segments that compose the level, and achieve a win-condition by reaching the final platform.

As a little addition, seen as moving on top of segment blocks would impede more vertical gameplay in segments that acquire a more elevated inclination, a second mode of play was introduced, out of necessity and convenience, so that the player, in segments where inclination goes beyond 45º relative to the floor, the player floats through the air towards the next platform, instead of jumping from block to block. The movement is continuous, and the player cannot control, only being able to move side to side or up and down, still in a discrete fashion, instead of continuous.

For the application of parameters in the level creation process, the game also introduces enemies along the several segments. Enemies will behave the same way as the player, in what concerns movement, except that they will only move every second or so, in random directions, provided the selected direction isn't invalid, or doesn't leave the current segment. Furthermore, as an additional obstacle, several places where blocks would normally be can instead have no block at all. It is important to note that either touching an enemy or moving towards a grid position where there is no block results in the player being thrown back towards the very beginning of the entire level.

And thus, the game design and implementation is kept relatively simple, so that there can still be focus on the other core elements of the new proposed method for level creation.

3.4 Path-stage level data creation

The initial stage in creating the procedural levels passes through the definition of the initial input data and parameters to control the future output of the actual generation stage. To this end, an Android application was developed that would help any user in doing so, by defining a physical path in any appropriate space.

At the time of development, the application targets a bare minimum of any Android device that runs Android 5.0 or above. As such, and as the first step in the application, a feature in the Android API called the "SensorManager" is used to gather a list of all available sensors in the used device. This will typically yield a proximity sensor, light sensor, accelerometer, and others. The available sensors actually depend on the device and on how modern it is, but more complete devices can yield such sensors as gyroscopes.

One more capability is necessary in this list. There is still a need for a method for defining the device's position in three dimensional space. Initially, it was thought that the accelerometer would be enough to determine the device's movements, but research and actual testing on the

developed application proved that this method would be too imprecise, and captured points in space would be very far from ever resembling any path defined with the device. As such, since most smartphones possess GPS capabilities, GPS was added as an option as the final "sensor" to be used, which the user can select, in order to also capture the device's GPS location at will. Although GPS can also have its own level of imprecision, it is precise enough that a few restrictions ensure a worthwhile process of positional data collection.

1) A minimum of 5 meters of distance between each registered point ensure a definite distinction between each point
2) 5 seconds of wait before registering a point in space ensures the GPS can pinpoint the device's location as precisely as possible

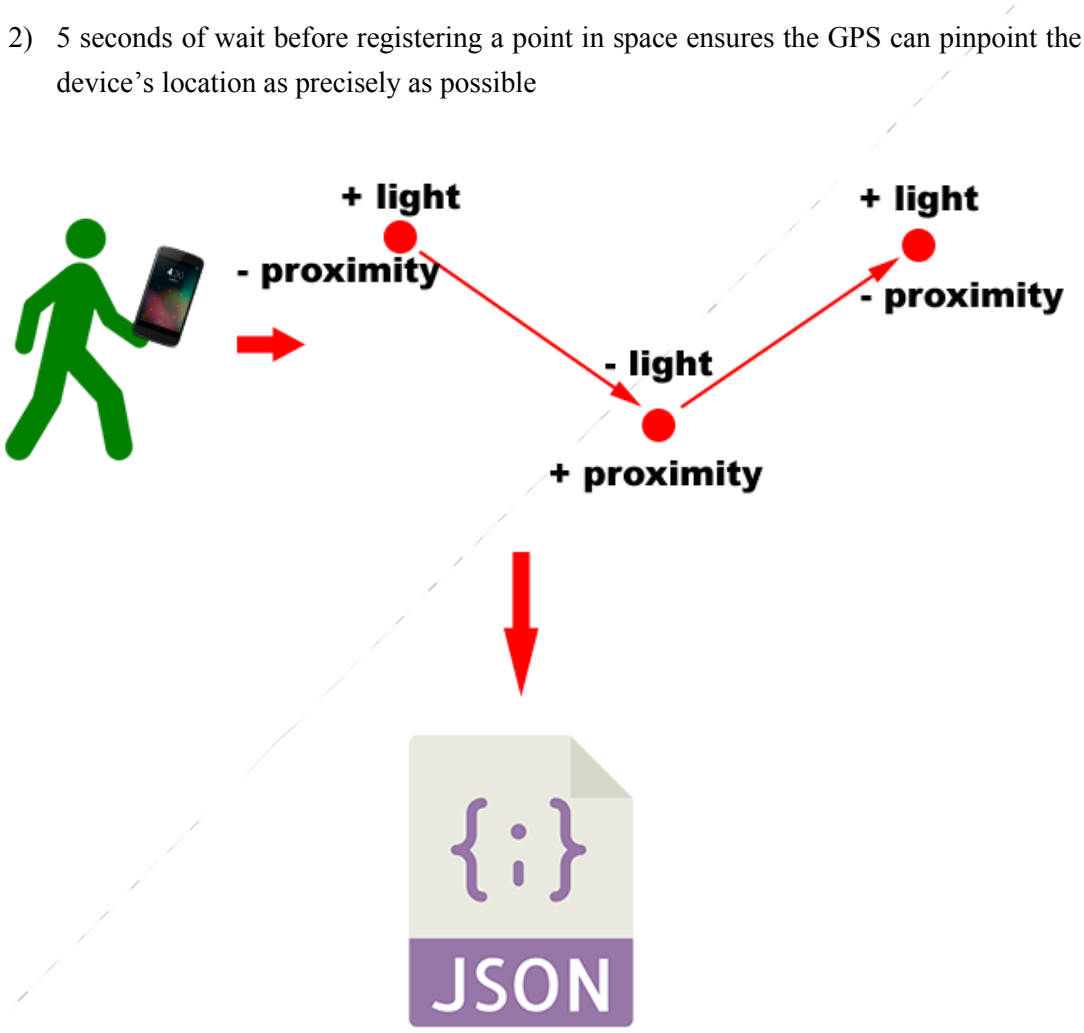

Figure 4.5: App process from capture of data to final JSON output

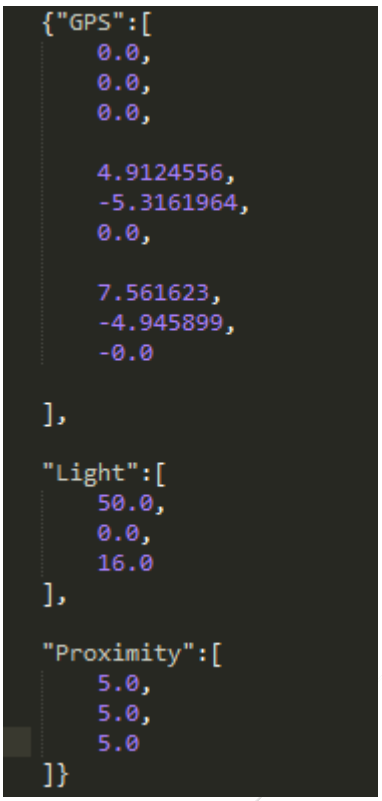

Figure 4.6: Example JSON data file created by the Android application

Finally, after selecting the intended sensors to use, the user is presented with a simple interface with only two buttons. A first button that, upon pressing, registers the user's location and any value associated to all selected sensors in the previous screen, while also adding a point to a list to denote how many points have been captured so far. The second button finishes the entire process, while outputting both values of GPS coordinates and sensor values for each one of those coordinates into a file in JSON format, ready to be used as a source of input data and control parameters.

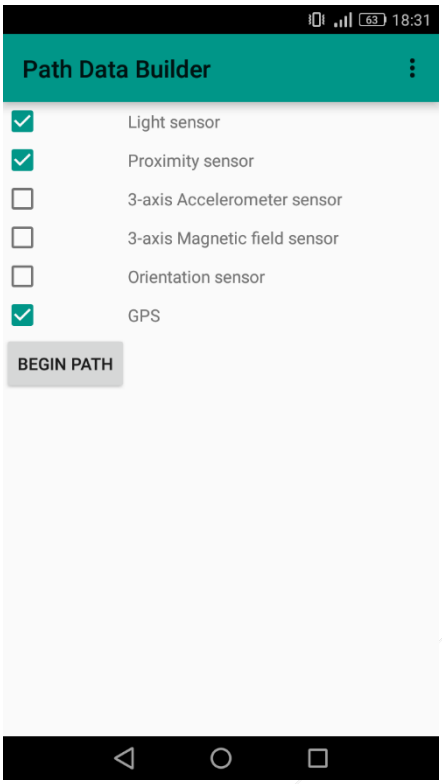

Figure 4.7: First screen in Android application, where user can select sensors to use

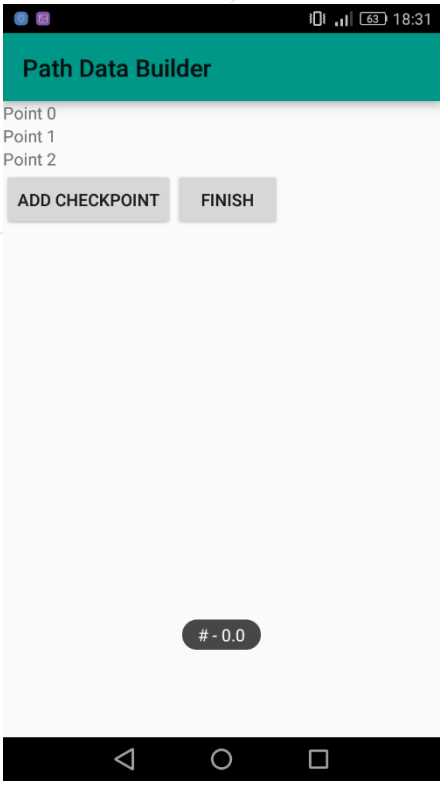

Figure 4.8: Second screen in Android application, where user can add points to a path or finish the process

If for example, a user intended to make a “V”-shaped path, one would simply have to capture the necessary points that compose a “V”. That would usually just require 3 points, but a user a can use as many points as wanted, provided that the shape is large enough. Plus, if the user wanted the minimum possible value in the proximity sensor in the down most tip of the shape, all that would be required would be for the user to cover the sensor, wherever it might be in the device, while pressing the button that captures the current point.

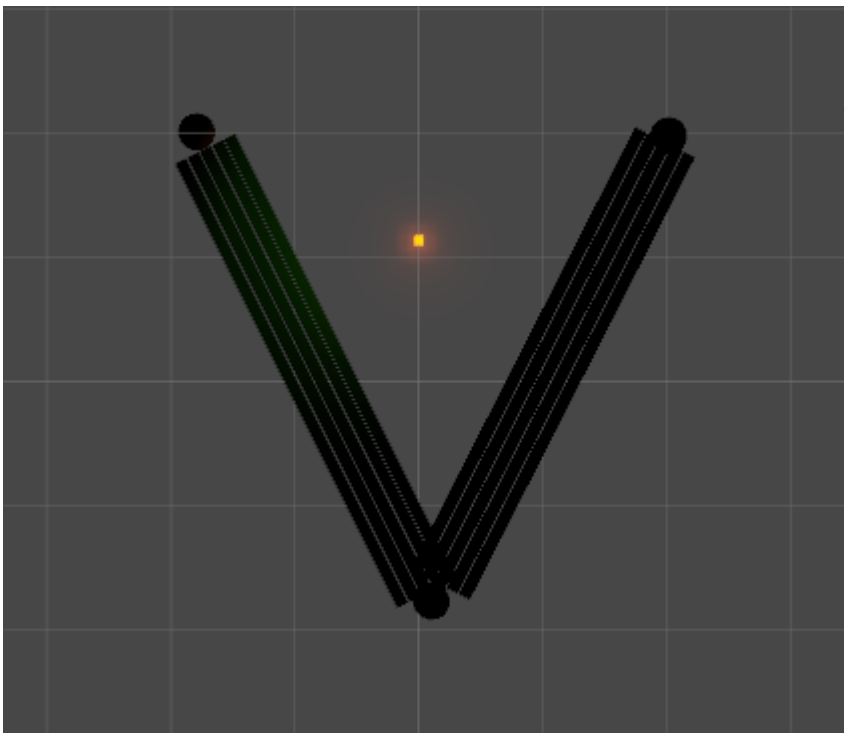

Figure 4.9: Example output of level design upon providing input data for a “V” shape

3.5 Level generation

Both the implementation of the game and level generation were realized through a premade game engine called Unity. Unity is a game engine that works both in 2D and 3D domains. As is the nature of a premade game engine, Unity would suit our needs in rapid prototyping and fast implementation of a game as simple as the one that was envisioned for testing of the path-based method. Additionally, at the time of writing, Unity is a very commonly used engine in the game development scene, and therefore it seems useful to develop these tools in such a widely used framework.

Unity manages its game objects through a very common pattern in game development. Its Entity-Component System (ECS) provides an appropriate method for managing any elements and game objects that is needed to add or control in our game and associated levels. There is still some

discussion between developers about whether Unity actually possesses an ECS, on account of most game objects (entities) depending only on either large components that possess a lot of functionality by themselves, or on a large quantity of small components, which will usually not functions by themselves. With no actual system to manage these entities and applying all the necessary logic to any entities with the right components, the "System" portion of Unity's implementation comes into question, although this is a matter that doesn't belong in the scope of this work.

In what concerns procedural level generation for the game implemented here, one must consider that there already is a solid system of corridors/segments that composes the path which will define the level structure. Taking positional input data, one can place all necessary intermediate platforms in space and generate the segments between each one. As a side note, while it is maintained that the user should keep a minimum of 5 meters of distance between each point, that distance is too small for the game's segments, when it comes to gameplay. Therefore, at the time of input data collection, on part of the algorithm, those positional values are multiplied by a factor of 4, to get sufficient distance between every point. Since each positional value has an associated value for each selected sensor, each platform gets not only a position in 3D space, but a parameter value for each used sensor, not counting GPS. For the implemented game the following association of sensors and parameters was made:

- Enemies – Controlled by light. On Android, light values can go anywhere between 0 and 10000. Therefore, it was made so 5000 created the maximum amount of enemies, which already causes a 5% chance of an enemy appearing in any given block. Sufficiently low values can be captured either in the dark or in the shade. Sufficiently high values can be caught with sunlight or with the help of a flashlight, should the application be used at night.
- Obstacles – The lack of walking blocks in segments is caused by the proximity sensor. All a user has to do for a higher chance of obstacles near the captured point is cover the sensor with any object. This causes a value of 0 in this sensor while no obstacle at all would give it a value of 5.

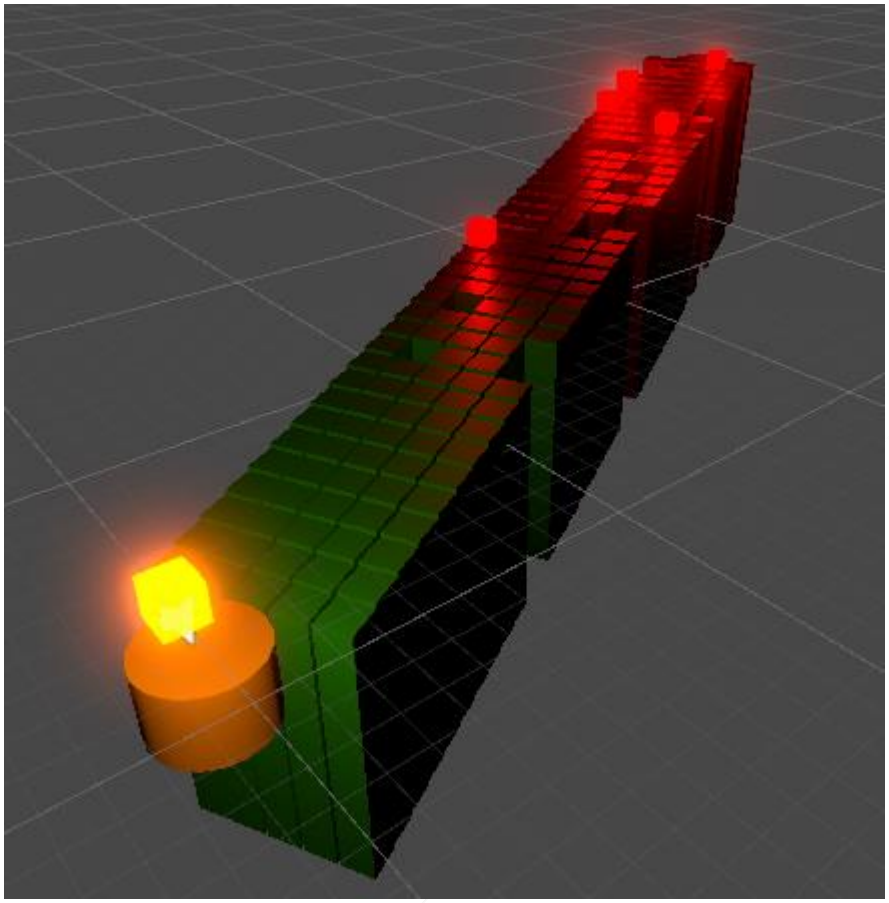

Figure 4.10: Example of single level segment, with increasing enemies and obstacles near the end

One important note here is how the values work quite differently. While high values in light yield large amounts of enemies, high values in proximity yield low amounts of obstacles.

Firstly, this comes from the way sensors work in Android. As the intended use of the application is to provide light where enemies are wanted, and to provide an obstacle to the proximity sensor where it would be desirable to have more obstacles in-game, the values turn out as such.

Secondly, this comes useful to us in testing how different methods of creating and providing data compare to the solution that is proposed. This will be further explained in a later chapter.

Succinctly, the provided parameters for the appearance of enemies and obstacles fall somewhat of the category of height map generation for several points in space but, instead of height, probabilities are being managed and controlled. Furthermore, the positions of the platforms follow the positions captured by the device in the first stage, with an additional round of Random Number Generation to ensure that any level created from the same set of input data and parameters, while retaining the same shape, is never truly the same. In this particular case a

value between -5 and 5 is added to any x, y and z value to every platform position in the final generated level.

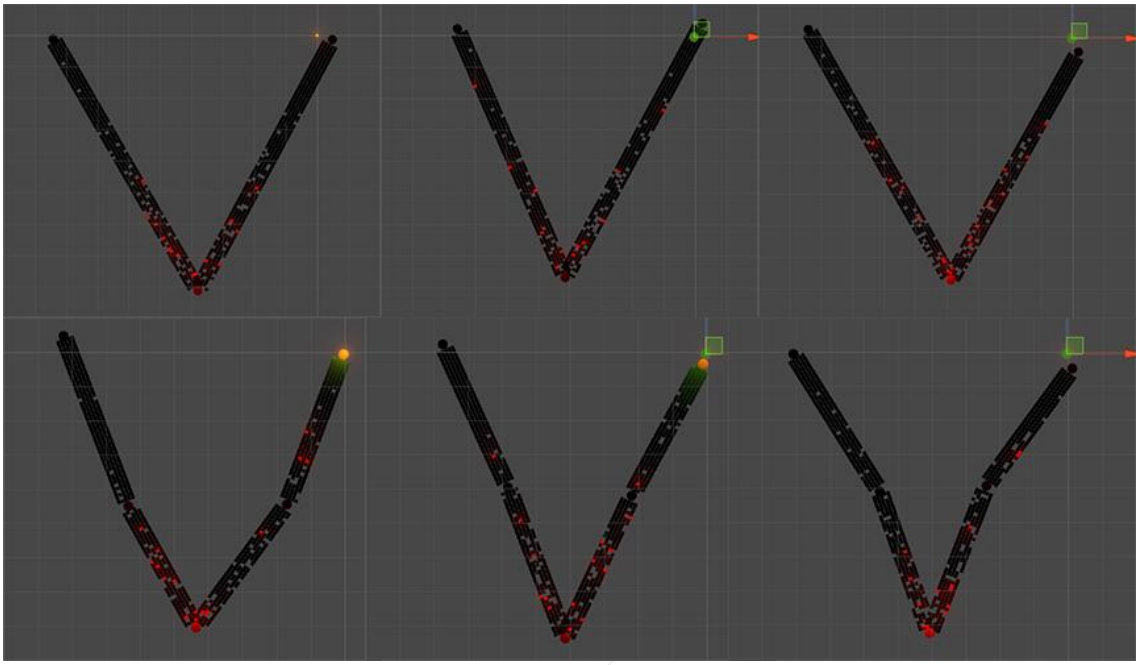

Figure 4.11: Example of level generation algorithm output with input data from a “V”-shaped path, first using 3 points (top row) and then 5 points (bottom row). Many enemies and obstacles in the bottommost tip of the path.

Chapter 4

Solution viability

After development of the new, proposed solution, the results need to be validated. To confirm whether the new procedure is on par and meets possible goals, such as decreased development time, facilitated development, and other desirable qualities, one turns to user testing, where participants are asked to go through the developed method, while important metrics are measured and user feedback is requested, so that it can be determined, through a proper evaluation, whether a path-based solution is indeed feasible and useful or not.

4.1 Test objectives

To evaluate the new proposed, path-based method, it is necessary, through user testing, to obtain data that allows us to determine, with objectivity, whether the solution is superior, and if so, in what ways. With this in mind, from the data we intend to obtain from testing, we wish to measure whether this method outperforms other already existing methods in the following categories:

- Creation time: The time that it takes to create a new level

  This can be measured simply by timing participants while they perform their tasks.

- Correctness: Whether a level was created according to what was planned or wanted

  This can consist of evaluating whether participants were able to create a level according to particular specifications or not. If not, perhaps it might be useful to know how or why they failed.

- Enjoyment: How much a creator enjoyed the process of level creation

This can come from user feedback in two possible ways. Participants can either be asked about their own perception of enjoyment, or one can go a step further and ask for their perception of how much time passed while creating a level, and compare that to what amount of time actually passed. What's more, one can then even compare the user's perception of enjoyment with the ratio of perception of passage of time and actual passage of time.

And with all of this in mind, we can carry a solid framework of testing to participants who partake in the new method for path-based procedural level creation.

4.2 Test description

In order to measure and evaluate the desired metrics and potential qualities of our solution, participants in the test are put through a specific set of tasks that allow us to judge those qualities as accurately as possible.

To do that, a set of 13 participants was taken through a set of level creation tasks, utilizing three different methods. Participants varied significantly is technological background, age and gender.

The tasks the participants were asked to accomplish were as follows:

- Procedurally create a level with a linear shape, wherein both enemies should appear with higher probability in the end section of the level. Conversely, this probability should be lower or null near the beginning of the created level.
- Procedurally create a level with the shape of an "S", with a higher chance of enemy instances in the middle section of the level, and thus lower chances near the end and beginning. Opposed to enemies, obstacles should be more common both at the beginning and end sections of the level, and as sparse as possible near the middle.
- Procedurally create a level with the shape of a spiral. The level should contain more enemies towards the end of the level and more obstacles towards the beginning. As should be obvious, there should be little to no obstacles at the end of the level, and the same should happen to enemies at the beginning.

The three methods the participants were asked to employ while creating these three levels were used one at a time, to create all three levels, one after another. Only after creating all three levels would a participant move on to the next method. Participants would did not feel capable enough to go through a particular method could always skip it and start creating levels using the next. The methods were as follows:

- Manually editing values directly into a JSON file, which would be used as source of input data and parameters for level creation.

  This would mimic one of the earliest and still somewhat commonly used method of level creation and editing.

  The data was to be formatted as follows:

```
{“GPS”: [
      X coordinates, point 1,
      Y coordinates, point 1,
      Z coordinates, point 1,

      X coordinates, point 2,
      Y coordinates, point 2,
      Z coordinates, point 2,

      (…)
      ],
“Light”: [
      Point 1, light value,
      Point 2, light value,
      (…)
      ],
“Proximity”: [
      Point 1, proximity value,
      Point 2, proximity value,
      (…)
      ]}
```

  “GPS”, as should be presumable, refers to the points that compose the level. Each point will correspond to a platform, and the generator will link these platforms in order, through grids of blocks for the player to traverse.

  “Light” values, ranging from 0 to 5000, correspond to the probability of appearance of enemies. Each light value will be correspondingly be assigned to each platform.

  “Proximity” values will range from 0 to 5 and will control the probability of obstacles in the form of “holes”, inversely. That is, 0 will maximize the probability of holes appearing, while 5 will minimize it. Values are assigned to platforms the same way as light values.

- Manually placing points in three-dimensional space through the assistance of some sort of tool, while assigning parameter values to each of those points

  This would go in hand with the most common method of level creation today, using a tool with an appropriate environment to manually structure a level, but at a higher level than pure file editing.

- Using the proposed path-based method, first by going to an open, exterior space, using the application and device sensors to design a shape and path, and defining necessary parameters along the way, then feeding the resulting output file to the level creation process, getting any final level possible from procedural generation.

  This serves as the ultimate step of comparison to determine whether it superior in any metric or user perception.

During the execution of the creation of the several levels through these multiple methods, participants are already evaluated in some regards.

Firstly, they are times on how long they take to create each type of level, in each of the methods.

Secondly, they are evaluated for their correctness in designing the requested level. Correctness would be judged based on the shape of the level created by the participants and probabilities assigned to enemy and obstacle appearance. Should they resemble the asked values or should the intentions to perform correct values be clear, respective value assigning would be deemed correct. It should be noted that, for the second level type, users would be considered to be correct in designing the level shape even if the "S" they performed were to be mirrored. In this category participants can only commit a maximum of three mistakes. One if they mis-design the level's shape, one if they assign wrong enemy appearance probabilities, and one more if they fail to assign correct obstacle appearance probabilities.

At the end of the experiment, users would be asked for their feedback, and some questions are to be noted.

Users are immediately asked for their own perception of how much time they spent, on average, creating a level for each method.

Participants are also asked, from 1 to 5, how much they enjoyed the method in question. This question is related to the previous one, since one can take this answer, and relate it to a comparison of the actual measure of time and the participant's perception of passed time during level creation.

Additionally, users are asked again, from 1 to 5, how difficult they thought the method was, for every method.

At the end of the questioning, for mere confirmation of other important characteristics present in the resulting levels, such as feasibility, procedural nature, and respect for provided input data and parameters, at least in the eyes of the user, participants were asked, from 1 to 5, whether levels were clearly based on the data they provided, whether their created levels were playable, and whether they were able to generate additional, always different levels from the same set of data that they created.

As a final confirmation, they were asked to select their favorite method of level creation.

A final “Observations” text field was added for users to provide any further feedback they thought was necessary or significant, to perhaps suggest improvements, or report or any problems or any issues that were encountered. As a free response opportunity for the participants, this field serves as an appropriate chance of input for unexpected data points of interest.

4.3 Results

After testing, let us analyze the results of the several participants. Raw data can be checked in Appendix A.

As for results, following the guidelines of important metrics that are to be assessed, here is what one can conclude from the gathered data.

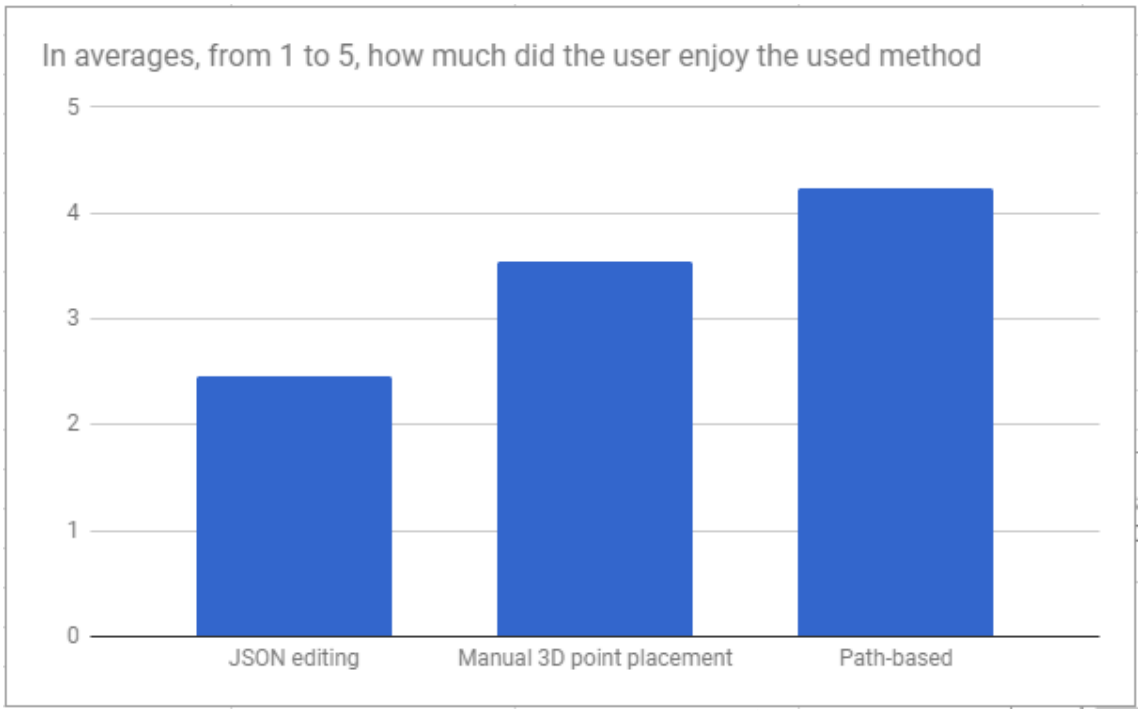

Figure 5.1: How much users rate their enjoyment of the several used methods

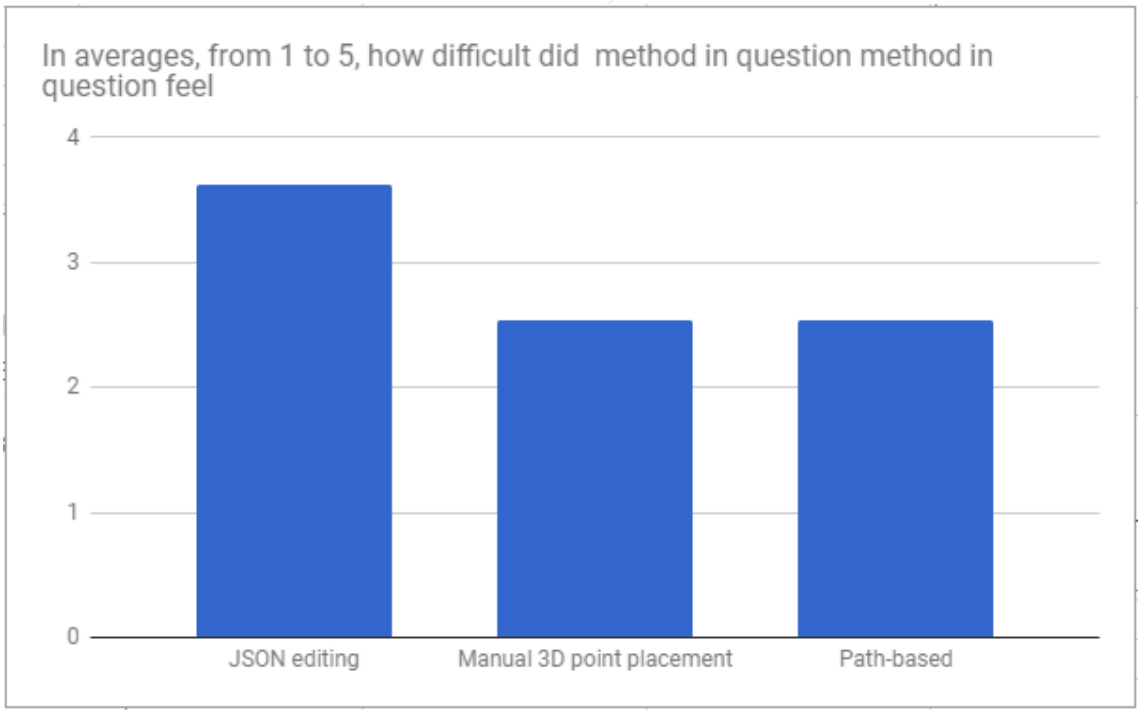

Figure 5.2: How much users rate difficulty in using each method

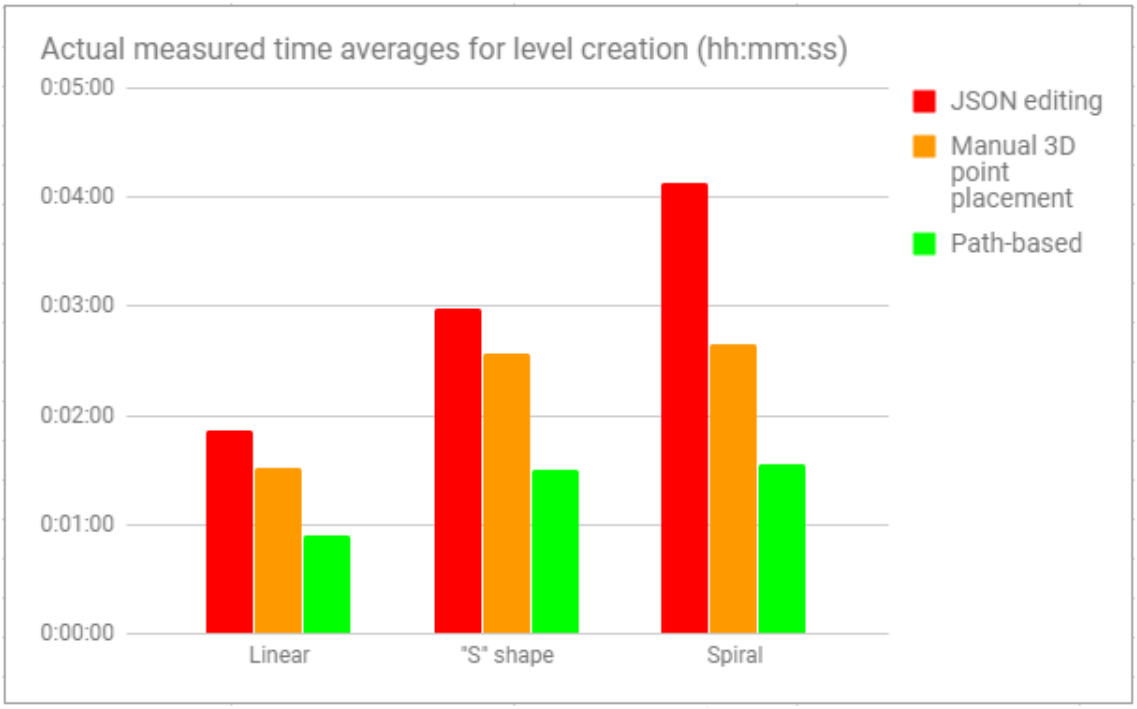

Figure 5.3: Measured times of level creation for every creation method

Figure 5.4: Total number of mistakes carried out by participants in each method

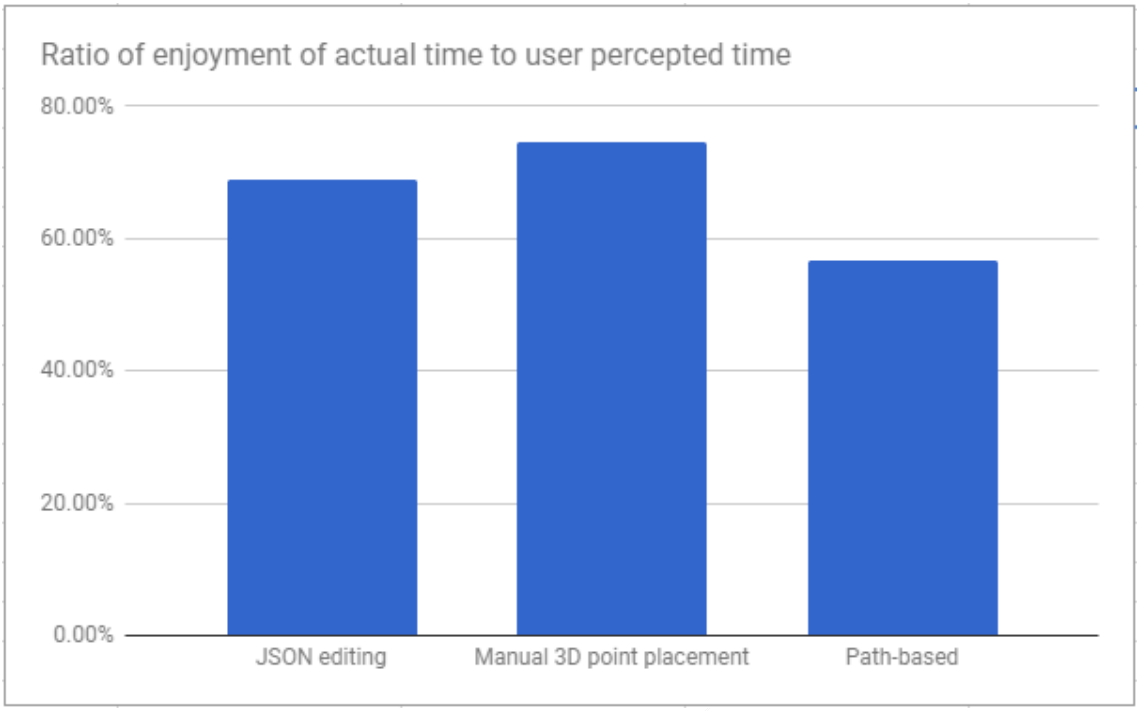

Figure 5.5: Ratio of enjoyment of actual measure time to user percepted time

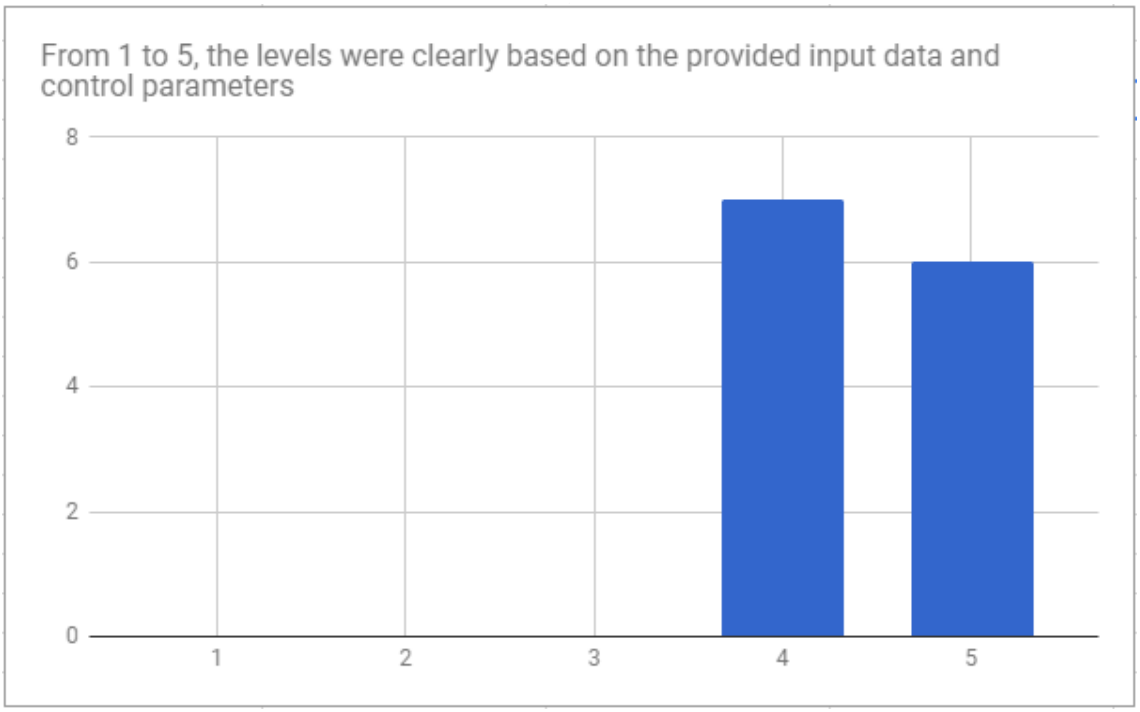

Figure 5.6: User perception of whether generated levels followed provided input data and parameters

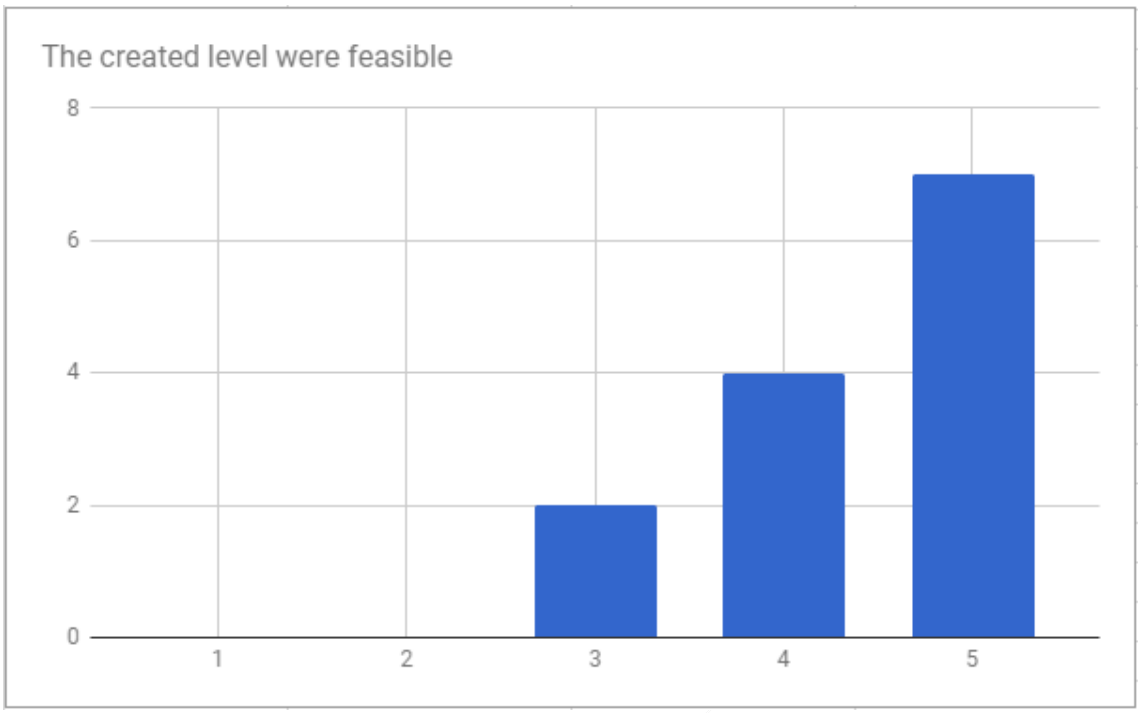

Figure 5.7: User perception of whether the generated levels were feasible

Figure 5.8: User perception on whether the created levels carried procedural characteristics

All additional feedback notes from participants are best addressed further, in a section that probably mentions possible future work and improvement on the current design.

4.4 Analysis

We can start by verifying how much users enjoyed each method, and through Figure 5.1, we can reach the conclusion that the path-based proposal is a clear winner in enjoyment. However, we must still take in consideration, through Figure 5.2, that it is still perceptually considered just as hard to handle as manual point placement in 3D space, though they are, by themselves, easier than pure JSON editing to reach the intended results.

From Figure 5.3, which might the most important of the set, concludes how, on average, level creation time goes significantly down as user transition from methods we initially deemed as harder, more complex, or more time-consuming, to methods that make the task increasingly easier, until they ultimately arrive at our path-based method, where level creation times reach a definite minimum. This initial analysis already validates this method's superiority when it comes to level creation time, where we can surmise that development time on a procedurally generated game would only benefit from such a system as this one in place.

From Figure 5.4 we can also conclude that users of the several methods share a slight but evident tendency to lower the amount of mistakes made when attempting to design a specific level. This can be explained by the more natural mnemonics and simpler mental models a user has to consider when creating the level itself. Leaving the design to a physical path outside and parameter control to more dynamic activities purported by cooperating between the user and a device's sensors provides a much larger sense of ease when attempting to combine several factors that will ultimately result in the intended final result.

In Figure 5.5, we can analyze the ratio of how the users faired in creating their levels, in regards to time, to their perceptual sense of how much time they took, on average, to construct those same levels. By relating this to their perception of enjoyment, we could perhaps obtain some sort of relation that would confirm their actual level of enjoyment. However, it is seen that values of perceptual enjoyment and the previously mentioned ratios do not coincide. This can have several reasons. Least likely would be that the participants are misjudging which method they prefer the most. The most likely scenario, however, is that the users differ very much one from another when it comes to time perception. Thus wild values assigned by themselves to what time they spent performing their several activities might not be at all reliable.

Finally, we can consider the feedback that supports the developed game and algorithm's correctly attributed abstract design preferences for procedural level generation. In this case, from figures 5.6, 5.7 and 5.8, we can conclude that the created levels meet the requirement of feasibility, meet the requirement of procedural generation, seen as users tend to agree that level

generated by feeding their input into the generation algorithm yields constantly different results, and yet they also seem to agree that created levels still respect every set of input data and control parameter that they provide.

Chapter 5

Conclusions and future work

To conclude, the necessary tools for any creator to procedurally create levels were developed, with a basis of a path as input data, and the proper use of smartphone sensors as a control of level generation parameters, given that a creator already has a playable game and a working procedural generation algorithm, capable to creating a wide variety of levels, but that can still accept data and parameters that allow for a significant control of how the final results of the generated process end up looking or functioning.

> smartphone application that would empower creators to realize the first step of the imagined pipeline was developed, being able to trace a path with sensor controlled parameters for level generation.

For assistance a game was developed with simple mechanics and design that would allow for testing of results. Coupled with that game, a procedural generation algorithm was written, capable of accepting parameters generated by the application, and use those as input for control on how a final level would look like and function according to enemy and obstacle placement.

5.1 What was accomplished

By providing users with a pipeline for a more natural and intuitive way of providing input data and control parameters, through testing, some important features of the new proposed solution were confirmed.

It boosts development time, turning the entire level creation process much shorter, seen as when users make use of our method, they reduce creation time by a significant factor. This comes to some importance to parties that are interested in more efficient development processes, as shorter development times are always useful and desirable for business sustainment.

Regarding this same concern, efficient level creation when it comes to correctness is also improved. Through this method, participants were least likely to suffer from mistakes, or to

produce incorrect output from the provided specifications. As should be obvious, this goes according to general present desires from any game development business or party that wishes to augment chances of sustainability through higher level of efficiency.

Lastly, it was confirmed that this method is at least as easy as some more modern methods of level creation, although the reason participants did not consider this method easier than all other might be related to several wanted improvements and certain issues discussed further ahead.

Ultimately, one can definitely conclude that the newly developed method is objectively superior in many regards and should definitely be considered in a procedural generation scenario that has emphasis on path necessity.

5.2 Future work

Considering all the effort spent for the sake of this work, there is still room for development.

Initially, after development and analysis of the path-based solution, a natural way forward is seen in using the smartphone not only for path definition and parameter control, but perhaps go a step beyond and use it to capture the features of not a few points belonging to that path, but instead the features of an entire area. Some concerns are immediately obvious. One would eventually have to face the problem of location precision. For the current work GPS was used, which in itself already carries some significant imprecision, and the use of this technology in an entire area would require for that area to be of significant size. Additionally, the necessary act of repeated data collection in an entire area, disregarding precision limitations, would possibly create diminishing enjoyment from the user, thus a decreasing level of engagement with the method.

Some participants were quick to notice how less technical people would fare quite better in creating levels through the utilization of the proposed path-based solution. This goes on par with what was initially suggested as a possible advantage of the method, and we see it replicated on those who actually put it to test. One might even perhaps get this conclusion from the few participants who were unable to create the requested levels in methods like JSON editing, and thought were completely capable when using the new method in a much easier fashion.

One lack in the method, as noticed both by users and author, consists of no direct feedback to the user when using the app, and this problem hits in two directions. On one hand, there is currently no way of ascertaining the current value of any sensor at the time of capturing a point. A bigger level of transparency of this element in the app would definitely facilitate the control of parameters for the generation of the intended levels. On the other hand, the user is not shown the device's current location, so the method is currently dependent on the participant's mental model of the designed path. However, to facilitate the process and make that mental model easier or plain unnecessary, we could display the registered locations on a map. This would additionally let the user fight or compensate for any possible imprecision in the GPS.

References

[Amb11] Mike Ambinder. Biofeedback in Gameplay: How Valve Measures Physiology to Enhance Gaming Experience, March 2011

[COB11] Daniel M. De Carli, Marcos Cordeiro d Ornellas and Fernando Bevilacqua. A survey of procedural content generation techniques suitable to game development, November 2011

[Dav17] Graham Davis. Procedural Level Generation in Unity for MERC, available at http://www.gamasutra.com/blogs/GrahamDavis/20170130/290326/Procedural_Level_Generation_in_Unity_for_MERC_part_1_of_2.php, January 2017

[FIM16] Jose M. Font, Roberto Izquierdo, Daniel Manrique and Julian Togelius. Constrained Level Generation through Grammar-Based Evolutionary Algorithms, 2016

[Fis12] Jordan Fisher. How to Make Insane, Procedural Platformer Levels, available at http://www.gamasutra.com/view/feature/170049/how_to_make_insane_procedural_.php, May 2012

[Gal11] Alex Galuzin. How to Plan Level Designs and Game Environments, available at http://worldofleveldesign.com/categories/level_design_tutorials/how-to-plan-level-designs-game-environments-workflow.php, December 2011

[HMV12] Mark Hendrikx, Sebastiaan Meijer, Joeri van der Velden and Alexandru Iosup. Procedural Content Generation for Games: A Survey, January 2012

[LLB09] Roland van der Linden, Ricardo Lopes and Rafael Bidarra. Procedural generation of Dungeons, 2009

[LLB13] Robert van der Linden, Ricardo Lopes and Rafael Bidarra. Designing Procedurally Generated Levels, 2013

[Pcg17] Procedural Content Generation Wiki, available at http://pcg.wikidot.com/category-pcg-algorithms, last accessed in 27 of June, 2017

[SYT10] Noor Shaker, Georgios Yannakakis and Julian Togelius. Towards Automatic Personalized Content Generation for Platform Games, 2010

[TBR15] Mircea Traichioiu, Sander Bakkes and Diederik Roijers. Grammar-based Procedural Content Generation from Designer-provided Difficulty Curves, June 2015

[TSB09] Tim Tutenel, Ruben M. Smelik, Rafael Bidarra and Klaas Jan de Kraker. Using Semantics to Improve the Design of Game Worlds, 2009

[Wes08] Mick West. Random Scattering: Creating Realistic Landscapes, available at http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php, August 2008

Appendix A

7.1 Raw experimental data

Measure time for level creation (direct JSON editing)(hh:mm:ss)		
Linear level	**“S” shape level**	**Spiral level**
00:03:08	0:04:53	0:06:00
0:01:52	0:05:21	0:08:35
0:01:00	0:03:21	0:04:06
0:01:24	0:01:17	0:03:15
DNFC	DNFC	DNFC
DNFC	DNFC	DNFC
DNFC	DNFC	DNFC
0:01:48	0:02:34	0:03:21
0:02:25	0:01:17	0:02:25
0:01:27	0:01:58	0:02:34
0:01:50	0:01:49	0:02:35
0:00:56	0:04:21	0:04:18
0:02:51	DNFC	DNFC

DNFC = Did not feel capable

Table 1: Raw time measurement data for creation of levels through direct editing of a JSON file

Measure time for level creation (manual 3D point placement)(hh:mm:ss)		
Linear level	**“S” shape level**	**Spiral level**
0:04:18	0:04:01	0:05:15
0:01:35	0:02:54	0:08:12
0:01:06	0:01:58	0:02:15
0:00:41	0:00:59	0:02:30
0:01:15	0:01:24	0:01:43
0:01:07	0:03:41	0:02:02
0:01:13	0:03:04	0:01:50
0:01:04	0:01:32	0:01:28
0:01:32	0:01:20	0:01:41
0:01:14	0:01:46	0:01:26
0:00:34	0:01:14	0:01:35

0:00:45	0:03:34	0:02:26
0:03:19	0:05:55	0:02:01

Table 2: Raw time measurement data for creation of levels through manual 3D point placement using an assistance tool

Measure time for level creation (manual 3D point placement)(hh:mm:ss)		
Linear level	**“S” shape level**	**Spiral level**
0:01:30	0:02:18	0:02:33
0:00:42	0:01:10	0:03:17
0:00:45	0:01:20	0:01:19
0:01:16	0:01:29	0:01:43
0:00:27	0:01:00	0:01:08
0:00:48	0:01:28	0:00:59
0:00:45	0:01:21	0:00:53
0:00:33	0:01:43	0:01:22
0:01:04	0:01:03	0:01:32
0:01:07	0:01:12	0:01:40
0:00:42	0:00:49	0:00:55
0:00:52	0:02:27	0:01:45
0:01:14	0:02:09	0:01:03

Table 3: Raw time measurement data for creation of levels through the path-based solution

Mistakes performed during activity (direct JSON editing)		
Linear level	**“S” shape level**	**Spiral level**
	S	S
		P
P		S
	P	P
P	P	
	s	

S = Mistake in designing the level’s shape

L = Mistake in correctly assigning enemy probabilities

P = Mistake in correctly assigning obstacle probabilities

Table 4: Mistakes performed by participants upon directly editing a JSON file

Mistakes performed during activity (manual 3D point placement)		
Linear level	**"S" shape level**	**Spiral level**
SP	LP	P
		P
P	P	
		P

S = Mistake in designing the level's shape

L = Mistake in correctly assigning enemy probabilities

P = Mistake in correctly assigning obstacle probabilities

Table 5: Mistakes performed by participants upon manually placing 3D points with the assistance of a tool

Mistakes performed during activity (path-based method)		
Linear level	**"S" shape level**	**Spiral level**
P		S
	P	
	L	L
PL		

S = Mistake in designing the level's shape

L = Mistake in correctly assigning enemy probabilities

P = Mistake in correctly assigning obstacle probabilities

Table 6: Mistakes performed by participants upon attempting to create level using a path-based solution

7.2 Feedback and user perception data

User percepted time (mins)	Percepted enjoyment (1-5)	Percepted difficulty (1-5)
15	2	4
5	4	2
4	3	3
2	2	4
DNFC	DNFC	5
DNFC	DNFC	5
DNFC	DNFC	5
3	2	2
2	3	2
4	3	3
15	4	5
5	2	4
DNFC	DNFC	5

DNFC = Did not feel capable

Table 7: User perception of time, enjoyment and difficulty, regarding direct editing of JSON files

User percepted time (mins)	Percepted enjoyment (1-5)	Percepted difficulty (1-5)
15	3	4
5	5	1
3	3	2
1	4	3
20	4	2
4	1	4
1	4	2

2	4	2
2	1	4
2	4	2
5	4	2
3	5	2
NA	4	3

NA = No answer

Table 8: User perception of time, enjoyment and difficulty, regarding manual placement of 3D points in space with the assistance of a tool

User percepted time (mins)	**Percepted enjoyment (1-5)**	**Percepted difficulty (1-5)**
5	5	1
2	5	1
3	5	1
1	4	3
15	4	2
2	4	1
0.5	5	2
5	5	4
2	4	5
5	3	3
10	4	3
5	3	5
15	4	2

Table 9: User perception of time, enjoyment and difficulty, regarding the use of the path-based solution