CMPMP2Y — Dissertation Proposal
Procedural Generation of Dungeons

Olivier Legat
Supervisor: Prof. Andy Day

Abstract

The use of procedurally content generation (PCG) is not un-common amongst games. PCG algorithms notably present the advantage of providing potentially infinite content for the players. However, implementing a good PCG algorithm that generates convincing results is not a trivial task. This proposal discusses how to undertake a project that implements a PCG algorithm that generates a 3D interact-able environment for a dungeon-like game.

Contents

1 Introduction

Level design is a crucial factor in video games that has been around for as long as games have. It consists of manually designing an environment by specifically defining every bit of detail and importing them directly into the game. Such details include, but are not limited to: Terrain shape, goals or mission objectives, location of enemies and collectible objects.

However, level designing is not the only method to create the virtual environment of a video game. Another method is to procedurally generate a virtual environment. The concept behind this method is to write an algorithm that will design a level based on randomly generated numbers. Note that the procedural method can also be referred to as level designing, but to avoid ambiguity between the two approaches they will be referred to as *level designing* and *level generation* respectively.

Level generation is not a particularly new concept. It has been around for a significant amount of time and is present in a considerate number of existing games. Some games such as Borderlands and Left 4 Dead will have pseudo-random levels, where the overall layout of the game's level is designed by hand with certain random elements to it [Togelius et al., 2011]. Other games such as Diablo will completely discard level designing and generate all the levels, enemies and props at random [Nitsche and Fitzpatrick, 2008].

Naturally, this programming-oriented method differs greatly from the traditional level design that is more directed towards an artistic path.

1.1 Purpose and motivation

The programmatic approach presents various advantages over level designing. Two major advantages are:

- **Time:** As mentioned previously, level designing is a very time-consuming process. However, computers are much faster than humans. Once a good level generator is complete it can potentially generate an infinite amount of levels in a very short time-span.
- **Amount of content:** As the direct result from the time constraint, level generation offers far more content. Video games with a fixed level design offer far less re-playability than those with procedural level generation because the game will get boring and predictable after a while. Level generation can allow a completely different feel to the game every-time it's played and can offer seemingly endless content [Compton and Mateas, 2006].

Notably, other minor advantages can emerge from this programming approach. For instance, the total memory consumption of the game when distributed could be less, because content is generated on the fly when needed and doesn't have to be packed into the distribution medium [Togelius et al., 2011].

1.2 Statement of problem

Although manually designing a level is far more flexible and easy than generating one, it often requires a lot time and skilled people. Level generation is often considered more challenging than level designing. Level generation is more than just generating terrain or buildings. A generator often needs to take other factors into account. Some examples include mission objectives, objects the players can interact with, spawning enemies, and so on. The general term for random content creation in a game can be referred to as Procedural Content Generation (PCG) [Ashlock et al., 2011a].

These factors are often very specific to the type of game. Because of this, most existing generic algorithms for level generation do not take these factors into account. Additionally, another issue is that most algorithms are based around 2 dimensions for simplicity and not many 3D algorithms have been implemented.

1.3 Aim and objectives

When reading different articles I noticed that the different methods were all focused on a specific type of environment or on a specific game. In my dissertation, I will primarily focus on generating underground dungeons but I will insure to keep the code as generic as possible so that it can be used in a number of different games. I will also be attempting to implement a 3D-based algorithm since these appear to lack much in the industry.

Overall, the summarised objectives are as follows:

- Develop an algorithm for random dungeon generation in 3-D.
- Take game features such as interact-able objects, enemies and so forth into consideration.
- Keeping the algorithm unspecific to a certain game. In other words, generalising it as much as possible.

2 Short history

Procedural content generation increases the amount of content or the value of a game. For this reason, it is not uncommon for games to include some form of PCG.

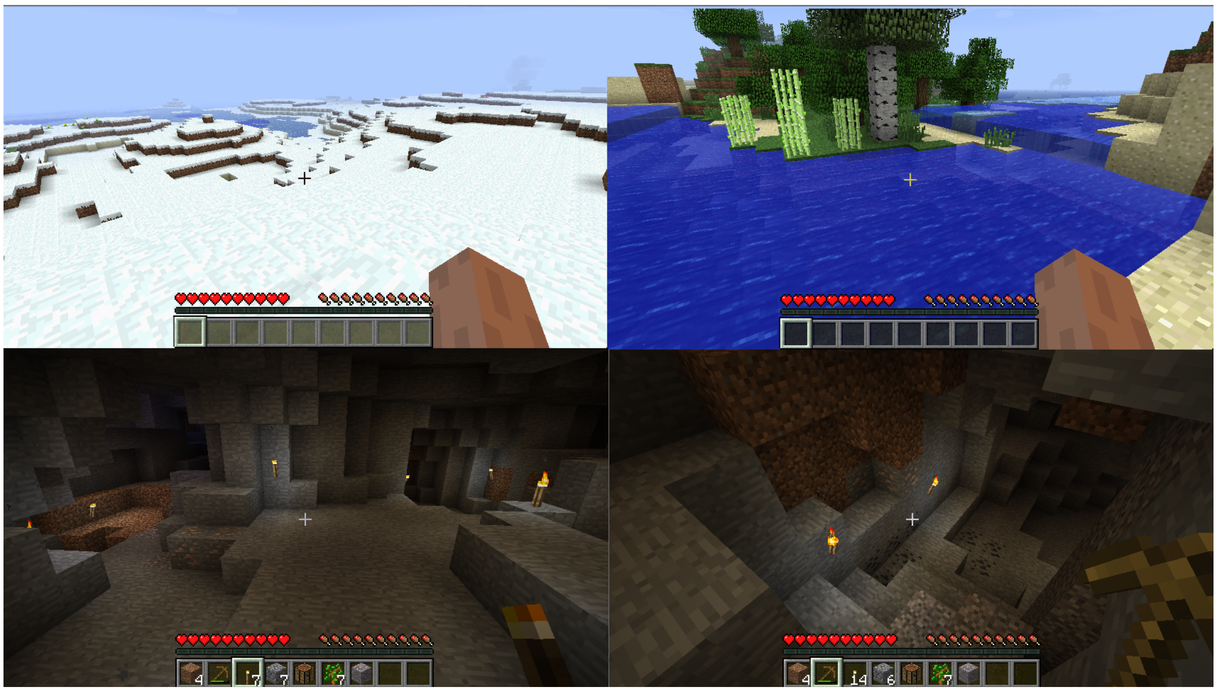


Figure 1: Procedural level design in Minecraft. Everything in the environment presented is generated at random, apart from the torches. Those were placed to illuminate the underground caves.

One notable example is Minecraft, a game originally designed by the Swedish developer Markus Persson (a.k.a "Notch") who later founded the game studio Mojang [1]. In Minecraft, the player is placed in a world composed of different types of blocks. The blocks are placed at random

[1]http://www.minecraft.net/game/credits (Date accessed: 28th Feburary)

using Perlin noise [2]. The wide range of block types allows the player to experience different biomes. The world is pseudo-infinite; meaning that as the player moves into unexplored areas the game will automatically generate more content to fill that area.

A similar game to Minecraft that also uses PCG is Terraria. Like Minecraft, it also revolves around randomly positioned blocks, with the main difference being that it's a 2D game. Terraria is a lot more focused on generating content other than terrain. In Terraria players can encounter numerous different types of enemies and can find treasure in vases and chests that are scattered across the world.

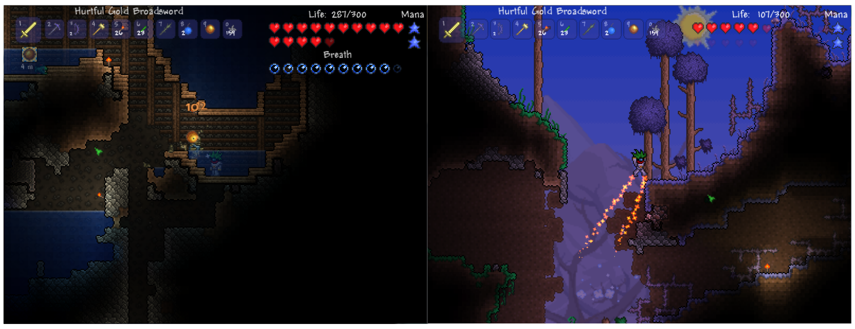


Figure 2: Terraria is a 2D game inspired by Minecraft which also utilises PCG.

3 Literature review

The literature review will examine how existing algorithms work and extract key elements from them. There are various articles that explore a numerous amount of different methods to generate levels. Here are some examples of such methods:

3.1 Genetic algorithms

Togelius et al. distinguish 2 types of content generation. The first is with the use of a *constructive algorithm* that consists of generating the content in a single run with little or no backtracking [Togelius et al., 2010b]. The other method, referred to as *generate-and-test algorithms*, carry out a number of attempts and select only the candidates that satisfy a set of tests [Togelius et al., 2010b]. Typically, such algorithms are more complicated and more time-consuming, however if handled with care they can provide results in only a fraction of a second [Togelius et al., 2010a].

Genetic algorithms are essentially generate-and-test algorithms [Togelius et al., 2010b]. A genetic algorithm can metaphorically be described as such:

> "Genetic Algorithms mimic real life evolution, in particular based on Darwins theory of Natural Selection" ... "Features change over time due to natural mutations and mutual heritance." [Mourato et al., 2011].

The idea of a genetic algorithm is to create a series of candidates, selectively pick out the best ones and then repeat this process until a satisfactory solution emerges [Mourato et al., 2011].

3.1.1 Fitness function

The key element in any genetic algorithm is the *fitness function*, which is the function that defines 'how good' a solution is. This function is used to test the validity of a candidate [Sorenson and Pasquier, 2010].

[2]http://pcg.wikidot.com/pcg-games:minecraft (Date accessed: 28th Feburary)

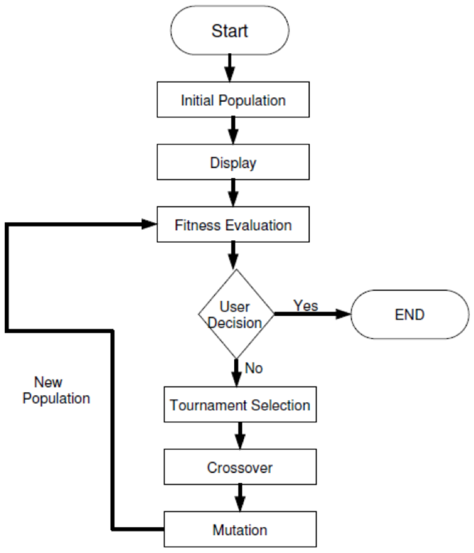


Figure 3: Flowchart representation of a genetic algorithm. Image sourced from [Walsh and Gade, 2010]

Based on the usage of this fitness function, we can obtain entirely different results. Sorenson and Pasquier demonstrated that it is possible to conceive a generic framework for PCG by considering the fitness function as the amount of fun a level can convey. Naturally, the concept of fun is a rather abstract concept from a mathematical point of view. Their fitness function is dependant on a challenge measurement function, which is specific to a certain genre of game. This abstraction on the fitness function allows the algorithm to generate content both for a 2D side-scrolling platform game (such as *Super Mario Brothers*) as well for a dungeon-like top-down view game (such as *The Legend of Zelda*) [Sorenson and Pasquier, 2010].

The fitness function can measure anything based on what is required. For instance, using a fitness function which measures fun is probably more viable for creating then environment of a video game [Sorenson and Pasquier, 2010], whereas a function which measures the aesthetic of a terrain is more suited if the algorithm should be designed to generate beautiful terrain [Walsh and Gade, 2011].

3.1.2 Genetic operators

When candidates are selected they will evolve in some way. Raffe et al. distinguish two kinds of selections: *parent selection* and *gene selection* [Raffe et al., 2011]. Distinguishing between the two is not necessary, but in doing so one gets an increased amount of control on how the algorithm behaves [Raffe et al., 2011].

Mutation This is where a single candidate evolves on its own, meaning certain genes will change into different values. The mutation can be random or selective [Mourato et al., 2011].

Cross-over This is where two candidates are mixed together. How these two candidates are mixed depends on the type of problem being solved and the desired result [Mourato et al., 2011].

3.1.3 Search-Based

As described by Togelius et al. Search-based procedural content generation (SBPCG) algorithms are generally based upon some sort of evolutionary algorithm [Togelius et al., 2010a].

Togelius et al. define a search-based algorithm as such:

> "Search-based procedural content generation (SBPCG) is a particular type of generate-and-test PCG, where the generated candidate content is not simply rejected or accepted by the test but graded on one or several numeric dimensions, and where a search algorithm is used to find better content based on the evaluations of previously generated content." [Togelius et al., 2010a]

Ashlock et al. presented another interesting generic method for generating maze-like structures. They used an algorithm which is independent from the representation of the maze and demonstrate the execution on three kinds of representations [Ashlock et al., 2011a] :

- **Direct:** Represent by a grid where each element is a bit that represent either a wall, by the value 1, or a walk-able area, by value 0.
- **Indirect positive:** Values are stored in an array and represent the walls of the maze.
- **Indirect negative:** Values are stored in an array and represent the empty space (rooms and corridors) of the maze.

Naturally all these representation can be expanded to include more possible values, and in doing so allow different kinds of terrain / walls across the maze [Ashlock et al., 2011b].

Ashlock et al. also clearly illustrate the importance of the fitness function. Here are some example of different results obtain by merely using a different function [Ashlock et al., 2011a]:

- **Exit Path Length Fitness:** The function causes the maze to be composed of a single long path.
- **Primitive Re-convergence Sum Fitness:** This can produces a maze with dead-ends, also referred to as cul-de-sacs.
- **Isolated Primary Re-convergence Sum Fitness:** This function produces a mazes with paths that split off and meet up again at the end.

3.2 Occupancy-regulated

The main disadvantage with purely randomised levels is that things can look a little bit unnatural or lack the sense of creativity. Human-designed levels present the advantage of being more unique and more suited to the type of game developed [Mawhorter and Mateas, 2010]. Mawhorter and Mateas have suggested an occupancy-regulated algorithm that allows random levels to be used whilst maintaining liberty and control for human level designer.

The authors summarise the algorithm as such:

> "Occupancy-regulated extension works by assembling a level using chunks from a library." [Mawhorter and Mateas, 2010].

The idea is that level designers need only to design small key sections of the level that are referred to as 'chunks'. Each chunk consists primarily of empty space along with certain aspects of the game, such as a jump, an enemy spawn point and so forth. Along with this each chunk has a set of pre-defined locations where the players can legally situate themselves. These locations are called anchors [Mawhorter and Mateas, 2010].

The algorithm then uses these anchors to blend several chunks together. It will take one anchor from each chunk and map those anchors to the same location on in the world. Once the level is complete the post-processor will fill up the empty areas with additional things such as decorations, un-passable terrain and so forth. [Mawhorter and Mateas, 2010].

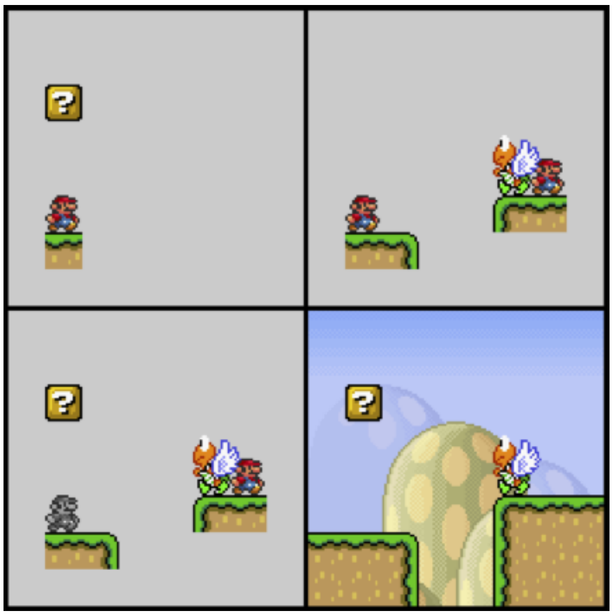


Figure 4: An illustration of Mawhorter and Mateas' algorithm. The two top panels are individual chunks. The lower left panel presents the two chunks mixed. The lower right panel is the result after post-processing. Image source: [Mawhorter and Mateas, 2010]

3.3 Rhythm-Based

In platform-genre video games such as *Super Mario* and so forth, the aspect of timing and rhythm is particularly important [Smith et al., 2011]. Such games revolve around the fact that the player must hit the right keys at the right time. One example of timing is the act of running and jumping to get across a gap in the level. If the player jumps too early, they will not reach the next platform and fall. If the player presses the jump button too late then their character would fall before starting the jump [Smith et al., 2008].

A level generator called Launchpad has been designed based on this key concept of rhythm [Smith et al., 2011]. The generator creates sections that are either 'jump' or 'walk'. The frequency of these jumps is defined by the rhythm of the algorithm. Additionally the length and height of the terrain will vary at random [Smith et al., 2011].

N. Shaker et al. then conceived an optimisation to such an algorithm using Artificial Intelligence (AI) [Shaker et al., 2010]. The algorithm consists of first generating a level at random. Afterwards, an AI player plays through the level and that player's actions are recorded. Based on those actions the algorithm will then optimise the level to make it more enjoyable.

Needless to say, such algorithms are directed specifically to plat-forming games and are not directly applicable to other genres. Although these algorithms are aimed at 2D games a 3D implementation could still be possible but would involve a large increase of constraints.

3.4 Procedural Modelling

Procedural modelling is special kind of PCG in which the algorithm generates a 3D mesh. Algorithms that we have explored until now focused on designing the structure of a maze, or the layout of platforms. However, none of these were particularly focused on the modelling aspect of an environment.

In an article published in 2003 the authors present an interesting way of procedurally modelling skyscrapers in real-time [Greuter et al., 2003]. The method consists of building the floor-plan by piling primitive shapes on one another and obtaining the union of that region [Greuter et al., 2003]. The border of this floor-plan is then extruded to form a 3D object and this process is repeated for the succeeding stories [Greuter et al., 2003]. This method is a simple, elegant and cost-efficient way of procedurally modelling a random room. A method such as this one could be used to generate the rooms of the dungeon in the dissertation.

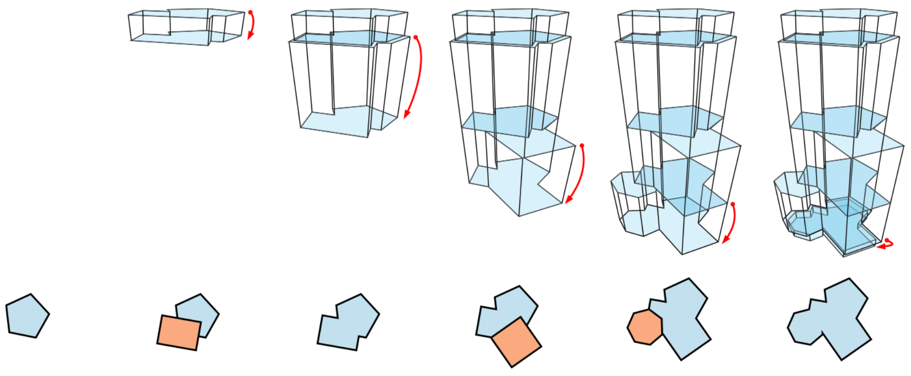


Figure 5: The various step of the floor-plan generation. Image source: [Greuter et al., 2003]

Another interesting topic is how to interconnect structures together through bridges, staircases and so forth. One simple approach to link two structures together is by simply interconnecting them linearly using something such as a box for instance [Krecklau and Kobbelt, 2011]. However this may not always look convincing since some and is not sufficient to generate interconnections with curves and deformed pieces [Krecklau and Kobbelt, 2011].

Various methods exist to generate non-linear interconnections. One example is to use a rigid chain, a structure which is best described as a:

> "sequence of revolute (rotational) and prismatic (translational) joints." [Krecklau and Kobbelt, 2011]

The angles and positions of these joints and be calculated through the use of Inverse Kinematics (IK) [Krecklau and Kobbelt, 2011]. However when solving an IK problem a solution may not always exist, such as if the target point is out of arm's reach [Tolani et al., 2000]. A good IK solving algorithm must not only be able to find a solution if one exists, but it should also detect and report if a result is unreliable or no solution could be found [Tolani et al., 2000].

4 Evaluation

Many interesting techniques have been discussed in the literature review on how to procedurally generate content for a video game. Sadly, not all of these techniques can be applied in the dissertation together. For instance, the rhythm-based approach is too aimed towards platform games. The notion of rhythm in different game genres, such as shooting games, is perhaps less important. Therefore the rhythm-based technique cannot be used to conceive generic algorithm in the scope of this dissertation.

4.1 Focus

Although the one of the aims of my dissertation is to develop a generic algorithm it will focus specifically on *dungeon-like* environments. The reason for this specialisation is that algorithms which target a specific type a problem often provide more satisfying result than those which generalise over different types of problems.

A genetic algorithm approach can allow a descent amount of generalisation through the use of an abstract fitness function. Therefore, a generic algorithm would be an interesting approach because it would allow users to be even more specific as to how the dungeon should look like. Rhythm-based PCG won't be directly supported since those algorithms are mainly aimed at 2D plat-forming games. However, with the use of an abstract fitness function it could potentially be possible for the user to specify how to generate a rhythm-based environment. Unfortunately genetic algorithms are notoriously slow [Togelius et al., 2010a], thus generating content in real-time isn't a key objective.

Another key focus point of this dissertation is to generate a 3D dungeon. Naturally, the algorithm will have to consider some sort of procedural modelling in order to pass vertices to graphics device. This can be done using some techniques discussed in the literature review. Rendering the environment efficiently is also a crucial factor. Basic concepts such as view-frustum culling, vertex and index buffer objects should be used. The algorithm should avoid generating high-poly meshes to save render-time and memory. Additionally the environment should be appropriately textured and the texturing process should remain generic.

The generator will be based around the concept that a dungeon consists of *rooms* and *corridors*, somewhat like an indirect negative representation of a maze-like level [Ashlock et al., 2011b]. Rooms can be constructed using Greuter et al.'s method of building a floor-plan [Greuter et al., 2003] or using an occupancy-regulated algorithm [Mawhorter and Mateas, 2010]. These two classes can be sub-classed to specify different types of rooms or corridors. Krecklau et al. have presented an interesting way to interconnect structures using inverse kinematics, this could be used the generate the corridor that connect rooms together [Krecklau and Kobbelt, 2011]. Additional content, such as doors, traps and so forth will be generated in post-processing because they are specific to a certain game.

4.2 Work-plan

The project will be developed using agile programming methods. This means that the aim is to get minimal working software done as soon as possible. Features will then be added or enhanced based upon existing code. These are the steps that will be taken in the development process:

1. **Initialisation:** This consists of getting a very primitive environment generated at random from scratch. If a generic algorithm is going to be used then some data must already exist in order to evolve it. The initialisation algorithm will simply place random rooms at random positions in a 3D space and randomly link them together. The precise randomisation methods will remain abstract in order to keep the initialisation generic.
2. **Evolving:** Once the initial dungeon is complete it will be time to enhance its features based on a fitness function. Again, the evolution process will remain as abstract as necessary to keep the whole project generic.
3. **Post-processing:** This phase consists of adding anything extra thing which cannot be generalised. This phase is the least important and will mainly be used for the purpose of demonstration.

4.3 Risk analysis

- **Hard-drive failure:** The impact of a hard-drive failure or lose is very severe and often un-resolvable if a back-up doesn't exist. For this reason, I will keep various back-up copies of my work. I will frequently copy content from my laptop's hard disk onto my external back-up drive. Additionally, I will store all work on a Sub-version (SVN) server to back-up my work.
- **Software developement delay:** Development delay is very common. In most cases when it occurs, some software features need to be cut down. To avoid having this done I

will always make sure to leave empty days in my schedule to allow myself to catch up and remain on time. Naturally, if this doesn't work then some features will inevitably have to be removed.

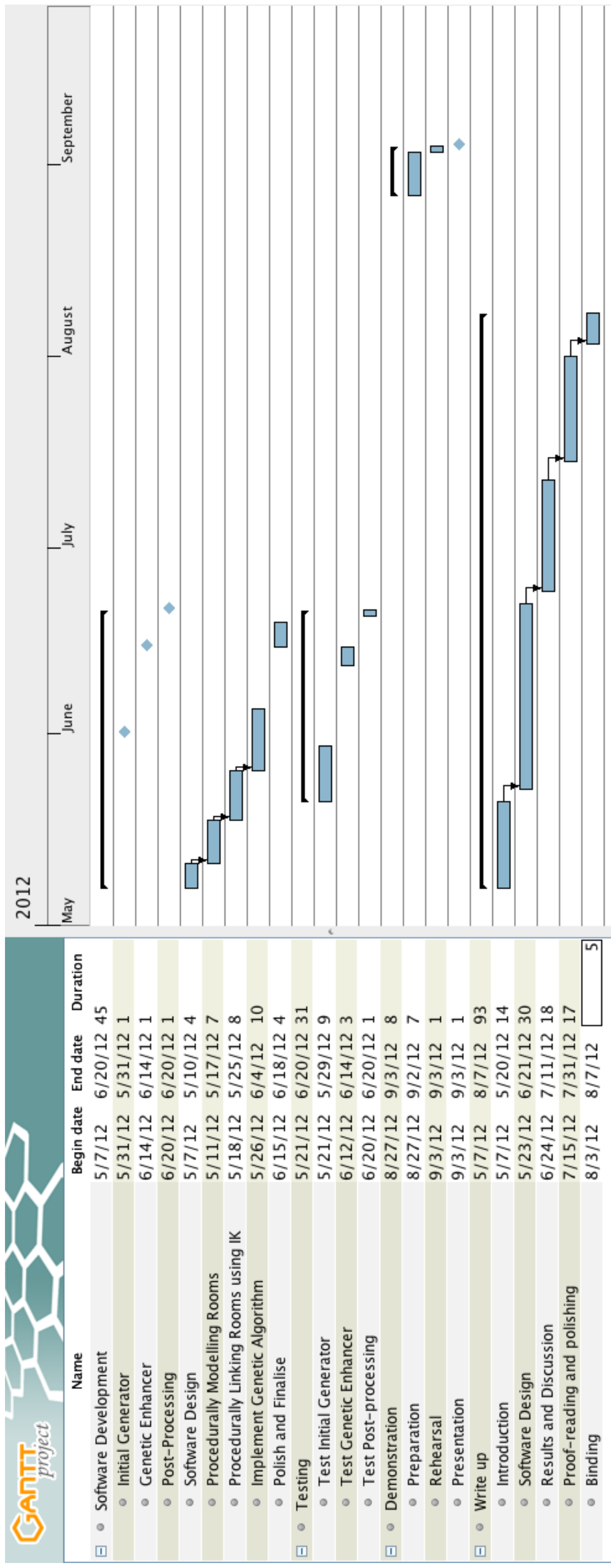
- **Overly complex task:** It is possible that as time proceeds I will realise that the problem is more complicated than I anticipated and won't be complete on time. This will generally occur towards the end of the project. Since the development process will be done using an iterative approach, working software will be available early on. If I cannot expand the software to reach it's initial requirements it will likely need to be simplified, by adding constraints or making assumptions, and the issues would be discussed to explain why the problem had to be narrowed down. However, if this problem is encountered early on then an alternative method should quickly be revised.

- **Minor Illness:** Small illnesses, such as a cold, will only last a short period and will not have a major impact. Leaving some empty days in the schedule will generally compensate for this.

5 Conclusion

Many PCG techniques have been discussed and genetic algorithms seem to be very popular amongst them. Sadly, they do not seem do be frequently used in real-time applications due to their inefficiency. We have observed that complicated PCG algorithms can be divided into set of simpler steps. The various techniques used will be used to implement a hybrid algorithm. Random rooms is will be procedurally modelled then inter-connected using IK [Krecklau and Kobbelt, 2011]. The overall dungeon will then be enhanced by a genetic algorithm.

A Appendix



Name	Begin date	End date	Duration
Software Development	5/7/12	6/20/12	45
Initial Generator	5/31/12	5/31/12	1
Genetic Enhancer	6/14/12	6/14/12	1
Post-Processing	6/20/12	6/20/12	1
Software Design	5/7/12	5/10/12	4
Procedurally Modelling Rooms	5/11/12	5/17/12	7
Procedurally Linking Rooms using IK	5/18/12	5/25/12	8
Implement Genetic Algorithm	5/26/12	6/4/12	10
Polish and Finalise	6/15/12	6/18/12	4
Testing	5/21/12	6/20/12	31
Test Initial Generator	5/21/12	5/29/12	9
Test Genetic Enhancer	6/12/12	6/14/12	3
Test Post-processing	6/20/12	6/20/12	1
Demonstration	8/27/12	9/3/12	8
Preparation	8/27/12	9/2/12	7
Rehearsal	9/3/12	9/3/12	1
Presentation	9/3/12	9/3/12	1
Write up	5/7/12	8/7/12	93
Introduction	5/7/12	5/20/12	14
Software Design	5/23/12	6/21/12	30
Results and Discussion	6/24/12	7/11/12	18
Proof-reading and polishing	7/15/12	7/31/12	17
Binding	8/3/12	8/7/12	5

Figure 6: Work-plan for the dissertation.

References

Ashlock, D., Lee, C., and McGuinness, C. (2011a). Search-based procedural generation of maze-like levels. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):260–273.

Ashlock, D., Lee, C., and McGuinness, C. (2011b). Simultaneous dual level creation for games. *IEEE Comp. Int. Mag.*, 6(2):26–37.

Compton, K. and Mateas, M. (2006). Procedural level design for platform games. In Laird, J. E. and Schaeffer, J., editors, *AIIDE*, pages 109–111. The AAAI Press.

Greuter, S., Parker, J., Stewart, N., and Leach, G. (2003). Real-time procedural generation of 'pseudo infinite' cities. In Adcock, M., Gwilt, I., and Lee, Y. T., editors, *GRAPHITE*, pages 87–94. ACM.

Krecklau, L. and Kobbelt, L. (2011). Procedural modeling of interconnected structures. *Comput. Graph. Forum*, 30(2):335–344.

Mawhorter, P. A. and Mateas, M. (2010). Procedural level generation using occupancy-regulated extension. In Yannakakis and Togelius [2010], pages 351–358.

Mourato, F., dos Santos, M. P., and Birra, F. P. (2011). Automatic level generation for platform videogames using genetic algorithms. In Romão, T., Correia, N., Inami, M., Kato, H., Prada, R., Terada, T., Dias, A. E., and Chambel, T., editors, *Advances in Computer Entertainment Technology*, page 8. ACM.

Nitsche, M. and Fitzpatrick, R. (2008). Designing procedural game spaces: A case study.

Raffe, W. L., Zambetta, F., and Li, X. (2011). Evolving patch-based terrains for use in video games. In Krasnogor, N. and Lanzi, P. L., editors, *GECCO*, pages 363–370. ACM.

Shaker, N., Yannakakis, G. N., and Togelius, J. (2010). Towards automatic personalized content generation for platform games. In Youngblood, G. M. and Bulitko, V., editors, *AIIDE*. The AAAI Press.

Smith, G., Cha, M., and Whitehead, J. (2008). A framework for analysis of 2d platformer levels. In *Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, Sandbox '08, pages 75–80, New York, NY, USA. ACM.

Smith, G., Whitehead, J., Mateas, M., Treanor, M., March, J., and Cha, M. (2011). Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(1):1–16.

Sorenson, N. and Pasquier, P. (2010). Towards a generic framework for automated video game level creation. In Chio, C. D., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A., Goh, C. K., Guervós, J. J. M., Neri, F., Preuss, M., Togelius, J., and Yannakakis, G. N., editors, *EvoApplications (1)*, volume 6024 of *Lecture Notes in Computer Science*, pages 131–140. Springer.

Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., and Yannakakis, G. N. (2010a). Multiobjective exploration of the starcraft map space. In Yannakakis and Togelius [2010], pages 265–272.

Togelius, J., Yannakakis, G., Stanley, K., and Browne, C. (2010b). Search-based procedural content generation. In Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekrt, A., Esparcia-Alcazar, A., Goh, C.-K., Merelo, J., Neri, F., Preu, M., Togelius, J., and Yannakakis, G., editors, *Applications of Evolutionary Computation*, volume 6024 of *Lecture Notes in Computer Science*, pages 141–150. Springer Berlin / Heidelberg.

Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):172–186.

Tolani, D., Goswami, A., and Badler, N. I. (2000). Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62(5):353 – 388.

Walsh, P. and Gade, P. (2010). Terrain generation using an interactive genetic algorithm. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE.

Walsh, P. and Gade, P. (2011). The use of an aesthetic measure for the evolution of fractal landscapes. In *IEEE Congress on Evolutionary Computation*, pages 1613–1619.

Yannakakis, G. N. and Togelius, J., editors (2010). *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, Copenhagen, Denmark, 18-21 August, 2010*. IEEE.