PROCEDURAL LEVEL AND MISSION GENERATION FOR COMPUTER GAMES

DÜNYA KULAVUZ

Master's Thesis

Graduate School

Izmir University of Economics

Izmir

2020

PROCEDURAL LEVEL AND MISSION GENERATION FOR COMPUTER GAMES

DÜNYA KULAVUZ

A Thesis Submitted to
The Graduate School of Izmir University of Economics
Master Program in Computer Engineering

Izmir
2020

ABSTRACT

PROCEDURAL LEVEL AND MISSION GENERATION FOR COMPUTER GAMES

Kulavuz, Dünya

Computer Engineering Master Program

Thesis Advisor: Asst. Prof. Dr. Kaya Oğuz

August, 2020

Game levels and missions are essential mechanics that most of the modern day games implement. In most games, players are assigned various missions that must be completed to progress through the game scenario. For game designers, creating such levels is a very time consuming activity. To save time on the development, one may want to generate these mechanics procedurally.

The aim of this thesis is to investigate, if procedurally generating a game map and a fitting quest line for that map is possible. At the same time, we will explore if it is possible to reuse a game level data to generate various quest lines on demand. Also, we aim to define a quest generation method which can work on any given map, sole condition being the given map must be representable as a graph data structure.

Keywords: Mission Generation, Game Level Generation, Procedural Content Generation.

ÖZET

BİLGİSAYAR OYUNLARI İÇİN PROSEDÜREL HARİTA VE GÖREV ÜRETİMİ

Kulavuz, Dünya

Bilgisayar Mühendisliği Yüksek Lisans Programı

Tez Danışmanı: Dr. Öğr. Üyesi Kaya Oğuz

Ağustos, 2020

Oyun bölümleri ve görevleri çoğu modern oyunun içerdiği önemli mekaniklerdir. Çoğu oyunda oyunculara tamamlamaları için çeşitli görevler verilir ve böylece oyuncu oyun senaryosunda ilerler. Oyun tasarımcıları için bu tarz oyun bölümleri ve görevleri oluşturmak epey zaman alıcı bir aktivitedir. Tasarımda zaman kazanmak için bu mekaniklerin prosedürel olarak yaratılması istenebilinir.
Bu tezin amacı, prosedürel olarak bir oyun haritası ve bu haritaya uygun bir görev dizisi oluşturmanın mümkün olup olmayacağını araştırmaktır. Aynı zamanda, istediğimiz zaman aynı harita verisini tekrar kullanarak, birbirinden farklı görev dizileri oluşturmanın olabilirliğini keşfetmektir. Ayrıca, verilen herhangi bir harita üzerinde çalışabilecek bir görev yaratma yöntemi tanımlamayı amaçlamaktayız, tek koşul verilen haritanın bir grafik veri yapısı olarak temsil edilebilir olmasıdır.

Anahtar Kelimeler: Görev Yaratımı, Oyun bölümü yaratımı, Prosedür ile İçerik Oluşturma.

ACKNOWLEDGEMENT

Above all, I would really like to thank my academic advisor Asst. Prof. Dr. Kaya Oğuz for his invaluable lessons. Also I would like to express my gratitude for his everlasting support and guidance throughout my academic life and through the process of this thesis.

In this opportune moment, I would like to thank my friends; Mesut Abedinifar, Mustafa Kemal Binli, H. Saygın Portakal, Tansu Taşçıoğlu, Ayşe Ördek, Gamze N. Aspar, Beyza Kanat, Çağatay Kırmızıay, Ece Sarıyar, Sahra N. Tabakoğlu, Özge Kubilay, Altuğ Koçak, Atilla Duruel, H. Can Saçmacı and many others I did not mention, for their never-ending support, friendship.

Finally, I would also really like to express my gratitude and fondness to my family; my father Turgay Kulavuz, my mother Şenay Kulavuz and my sister Nil Kulavuz for their unbounded love, care and support, whatever the circumstances.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF LISTINGS

CHAPTER 1 : INTRODUCTION

Today video games are very popular, entertaining people regardless of their age or gender. With the help of the internet, people can play together online, even if they are hundreds of kilometers apart from each other. Today's technology also allows people to play video games on their mobile phones, which enables them to play while they commute, on their lunch breaks or ultimately, in any small time period they choose to spare.

Nowadays it's also simple to access all different kinds of video games, regardless of the genre or the platform we are on. While online stores like *Steam* (Valve, 2003), *Origin* (Electronic Arts, 2011), *Battle.Net* (Blizzard, 1996) and many more others offer lots of games with all varieties of genres, application stores like *Google Play* (Google, 2008) and *App Store* (Apple, 2008) offer dozens of mobile games available for downloading. These online stores make things a lot simpler for the game developers to deliver their work to their customers. Also game engines like *Unity Engine* (Unity Technologies, 2005), *Unreal Engine* (Epic Games, 1998) and *Godot Engine* (Juan Linietsky, Ariel Manzur, 2014), provide most of the tools required to develop a full-blown video game. This enables smaller groups or even individuals to develop video games.

No matter the choice of game engine or the genre of the game we are developing, most of all modern video games need a lot of work done in order to be called complete. People enjoy good storylines, artwork, music, and visuals, but creating those aspects can be challenging and time consuming for the developers. A big developer team can share the workload and deliver a well polished game in a relatively small period of time. However, a smaller group of developers have to face a much larger workload per capita and delivering a complete game may take a very long time. For developers, timekeeping is important and taking shortcuts on the development while preserving quality is essential. Sometimes these shortcuts can be made with the help of an artificial intelligence (AI). Employing AI for generating some of the in-game contents like; terrains, rocks, trees, particle effects and many other bits, help game designers speed up the development while designing game levels.

Missions or quests (See 2.1.2) are vital mechanics which can be found in most of the modern games. Generally quests and game maps (See 2.1.1) are designed together in a way, so that they can support each other. Level designers, design quests to tell the story of the game. These quests, take players to different areas of the game map and let them advance further through the game scenario. For instance in the game *World of Warcraft* (Blizzard, 2004), players start in a relatively friendly environment and start discovering game mechanics in this area. Also they are offered relatively easier quests overall. These

quests tell various stories and completing them yields rewards like experience and other usable items. As players complete more quests, they warm up to the quest mechanics and ultimately directed to do more quests in more challenging areas of the game map. This completing quests loop let's them; discover more of the lore and travel to various areas of the game.

However vital, level design can become a very time consuming part of the development. In our scope, we want to speed up this process with the help of an AI. To put everything into perspective, assume a small developer team is assigned to design a game, which has a large game map and they need to create lots of side quests alongside the main quest line. In another scenario, a team has to design infinitely-many quests inside a map but only have a finite time to develop it. Arguably, one of the most time consuming and maybe repetitive activity while developing the games in the mentioned scenarios are going to be, designing a game map and a quest line which fits to that map. Luckily, in these situations we can rely on Procedural Content Generation (See 2.1.3) (PCG) techniques to aid the development.

Today in the game industry, PCG techniques are already widely being used. As Hendrikx et al. (2013) mentions in their survey, PCG can be used to generate scenarios, spaces, systems and bits like fire, fog, clouds and many others in the game content scope. Popular games like, *Diablo III* (Blizzard, 2012), *The Elder Scrolls V: Skyrim* (Bethesda, 2011), *Minecraft* (Mojang, 2009) and many others employ PCG techniques to generate certain aspects of the game's contents.

This thesis aims to,

- Explore how game map and quest generation can be automated using the PCG techniques.
- Review how mission and level generation is handled in the past studies.
- Propose a series of methods which can be used to generate a game map.
- Propose another series of methods which can work on the generated game map, also with any given map with the only condition being, the map must be representable as a graph data structure (See 2.2.1).

The thesis is structured as follows; Chapter 2 is where preliminary information required for understanding some of the terms throughout the thesis is being presented. Chapter 3 presents a survey of the literature, which contains information about how the game level and quest generation problems are handled in some of the past studies conducted. Chapter 4 is going to present the methods we propose for the solution of procedural game level and mission generation problems. In Chapter 5, we analyse the

implementation and run benchmark tests of the methods we provide in Chapter 4. Finally in Chapter 6, we conclude our study and point out to different methods and subjects which can be studied on the future.

CHAPTER 2 : PRELIMINARIES

Through out the discussion, we mention several technical terms and methods. In this chapter, we elaborate these subjects in order to give more insight about the thesis. These subjects are categorized under two sections being; Computer Games and Computer Science.

2.1 *Computer Games*

In this section we provide more insight about the terms and methods which are used in computer games.

2.1.1 *Game Map*

A game map is the environment where the game scenario takes place. A game may consist of one or many maps depending on the genre of the game. For instance, in the game *Pacman* (Namco, 1980), the task of the player is to collect all the dots on the map while avoiding enemies. When this task is done, the player proceeds to the next level, thus to a new game map. This is a good example of how many maps can be used in a single game. On the other hand, in a turn based strategy game called *Age of Mythology* (Microsoft Game Studios, 2002), the players are placed in a single map and their task is to compete with each other for sources on this map and conquer each others bases. This game can be an instance for how a single map can be used in a game.

Regardless of the genre of the game or quantity of the game levels, we expect players to explore, complete quests, and perform other game related interactions while traversing this environment.

2.1.2 *Game Quest*

A game quest is a task player is given in the game environment. In most of the cases, quests are used to give player the experience of progression throughout the game scenario. Upon completion player is usually rewarded for motivation and preparation for upcoming quests.

For instance in the online role playing game *World of Warcraft* (Blizzard, 2004), the player is successively prompted quests which allows player to; explore certain areas of the map, defeat enemies and progress through the lore of the game. Completing a given quest, rewards the player with experience, gold and other wearable, consumable or usable items.

There can be all different kinds of quests, depending on the genre of the game. However sometimes we can see similar types of quests are being used for similar genres. For instance in another role playing game, *The Witcher III: Wild Hunt* (CD Projekt Red,

2016) we see the same quest characteristics with the World of Warcraft. Similarly, the player is tasked to; explore various locations on the map, eliminate enemies, gather resources, deliver items and so on. In this discussion we are going to use similar quest archetypes for our quest generation algorithms.

2.1.3 *Procedural Content Generation*

Procedural Content Generation (PCG) is a study area of artificial intelligence (AI) which governs the techniques for generating data algorithmically. Today almost all of the modern games benefit from PCG techniques at some part of their development.

As we briefly discussed earlier, Hendrikx et al. (2013) mentions in their survey that in the scope of video games, PCG can generate various in game contents like; terrains, trees, rocks, textures, particle effects like fog, clouds, fire and so much more.

For instance in the game *Minecraft* (Mojang, 2009) the whole game map (terrain, dungeons, seas, villages, mobs and so on.) is generated by procedural content generation techniques. A screenshot of the game Minecraft can be found at Figure 2.1.

Figure 2.1. Screenshot of the game Minecraft.

Source: *Minecraft* (Mojang, 2009)

As it is widely used in video games, it's also well used in other graphics related areas in the industry. For instance in the film industry, developers can generate film contents like crowds, armies, fight scenes, special effects and many more using PCG techniques.

2.1.4 *Noise Generation Algorithm*

A noise generation algorithm is an algorithm which can produce noise data in any given dimensions. The noise data is used to create natural looking patterns in the computer graphics environment.

A popular noise generation algorithm Perlin Noise (Perlin, 1985), was developed by Ken Perlin and initially used for generating textures for models in the field of computer graphics. Nowadays most of the games which employ procedural content generation relies on this algorithm.

The algorithm starts with an *n*-dimensional grid of random gradient vectors. Then we use the dot products of neighboring vectors to produce noise values between -1 and 1 throughout the grid. In Figure 2.2, a 2-dimensional grid of gradient vectors is provided for demonstration. Finally, these values are interpolated to smooth out the noise data throughout the grid.

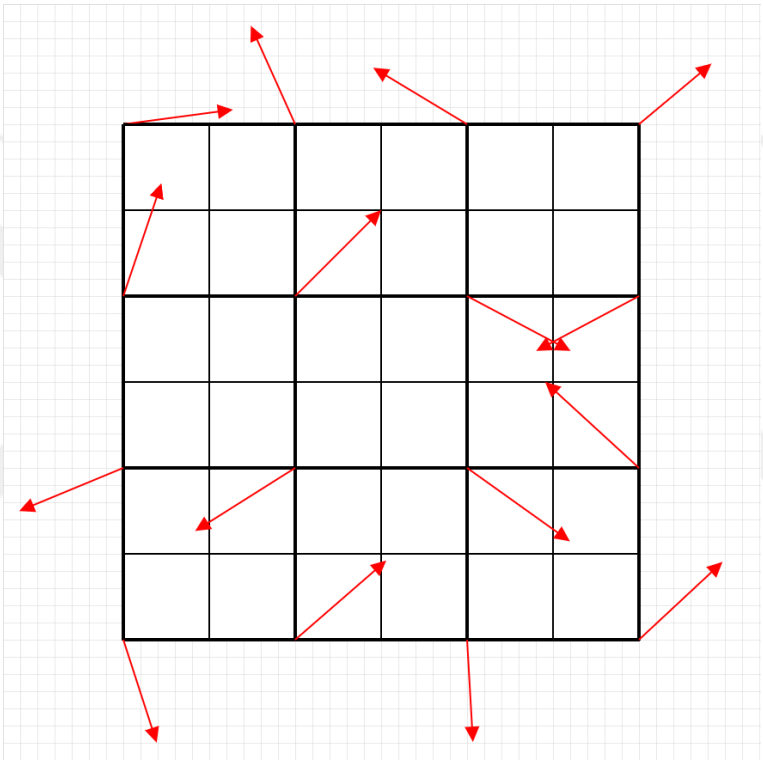


Figure 2.2. 2-D Grid of gradient vectors.

In 2001, a simplified and evolved version of this algorithm is developed and it's called Simplex Noise. This method is widely used in the industry for same purposes as Perlin Noise. In this discussion we are going to use an open source version of the Simplex Noise, which is called Open Simplex Noise.

As Gustavson (2005) demonstrates in their paper, an implementation for the 2-D Simplex Noise can be found in Algorithms 1 & 2;

Algorithm 1 Simplex Noise Algorithm (Part-1)

Noise2D(double xin, double yin)
int $grad3[][] \leftarrow$
$\{\{1,1,0\},\{-1,1,0\},\{1,-1,0\},\{-1,-1,0\},$
$\{1,0,1\},\{-1,0,1\},\{1,0,-1\},\{-1,0,-1\},$
$\{0,1,1\},\{0,-1,1\},\{0,1,-1\},\{0,-1,-1\}\};$
int $perm[512];$
for int $i \leftarrow 0; i \rightarrow 512$ **do**
    perm[i] $\leftarrow p[i\&255];$
end for
double $n_0, n_1, n_2;$
double $F2 \leftarrow 0.5 \times (sqrt(3.0) - 1.0));$
double $s \leftarrow (xin + yin) \times F2;$
int $i \leftarrow floor(xin + s);$
int $j \leftarrow floor(yin + s);$
double $G2 \leftarrow (3.0 - sqrt(3.0))/6.0;$
double $t \leftarrow (i + j) \times G2;$
double $X_0 \leftarrow i - t;$
double $Y_0 \leftarrow j - t;$
double $x_0 \leftarrow xin - X_0;$
double $y_0 \leftarrow yin - Y_0;$
int $i_1, j_1;$
if $x_0 > y_0$ **then**
    $i_1 \leftarrow 1;$
    $j_1 \leftarrow 0;$
else
    $i_1 \leftarrow 0;$
    $j_1 \leftarrow 1;$
end if
double $x_1 \leftarrow x_0 - i_1 + G2;$
double $y_1 \leftarrow y_0 - j_1 + G2;$
double $x_2 \leftarrow x_0 - 1.0 + 2.0 \times G2;$
double $y_2 \leftarrow y_0 - 1.0 + 2.0 \times G2;$

Algorithm 2 Simplex Noise Algorithm (Part-2)

int $ii \leftarrow i\&255$;
int $jj \leftarrow j\&255$;
int $gi_0 \leftarrow perm[ii + perm[jj]]\%12$;
int $gi_1 \leftarrow perm[ii + i_1 + perm[jj + j_1]]\%12$;
int $gi_2 \leftarrow perm[ii + 1 + perm[jj + 1]]\%12$;
double $t_0 \leftarrow 0.5 - x_0 \times x_0 - y_0 \times y_0$;
if $t_0 < 0$ **then**
    $n_0 \leftarrow 0.0$;
else
    $t_0 \leftarrow t_0 \times t_0$;
    $n_0 \leftarrow t_0 \times t_0 \times dot(grad3[gi_0], x_0, y_0)$;
end if
double $t_2 \leftarrow 0.5 - x_2 \times x_2 - y_2 \times y_2$;
if $t_2 < 0$ **then**
    $n_2 \leftarrow 0.0$;
else
    $t_2 \leftarrow t_2 \times t_2$;
    $n_2 \leftarrow t_2 \times t_2 \times dot(grad3[gi_2], x_2, y_2)$;
end if
return $70.0 \times (n_0 + n_1 + n_2)$;

2.1.5 *Mesh Data*

In computer graphics, mesh data structure is used to define the geometry of an object which is going to be drawn on the screen. This data is composed by vertices, edges and faces of the geometry which defines the given object. This data is then passed to the graphics engines with shading techniques applied and we see this data drawn on the screen.

2.1.6 *Mesh Normal Data*

A normal is a vector that points out perpendicular from a given object. In the computer graphics scope, graphics engines calculate this vector for each vertex of a given mesh in order to apply shading and lighting operations on it. In this discussion, we use the normal data to calculate the slopes of the faces present on the terrain mesh. The angle between the horizontal plane and the normal vector gives us the slope data.

2.1.7 *Height map*

In game development, we usually work with terrains which are non-flat. These terrains usually feature slopes, hills, holes and trenches, just like any landmass from the real world. To achieve those shapes, we start with a flat terrain which simply is a 2-D plane in a 3-D environment.

In order to transform this terrain into a 3 dimensional shape, we need to translate some of it's vertices on the Y (up-down) Axis. Since each vertices position, define the shapes of each face on this plane, this transformation will introduce height changes on the mesh. We can see the effects of these transformations in the Figure 2.3.

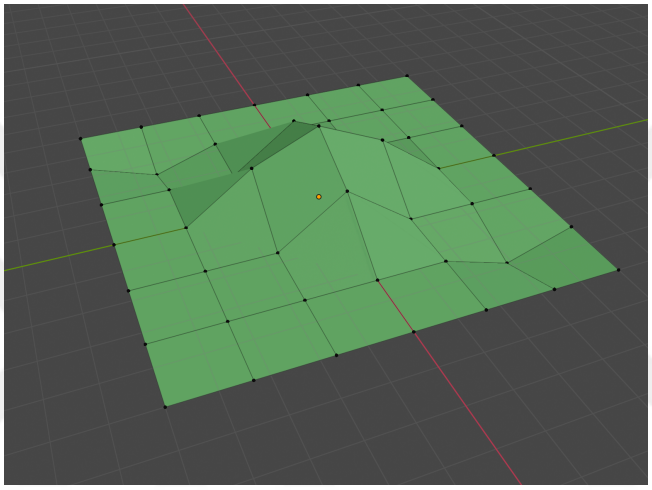
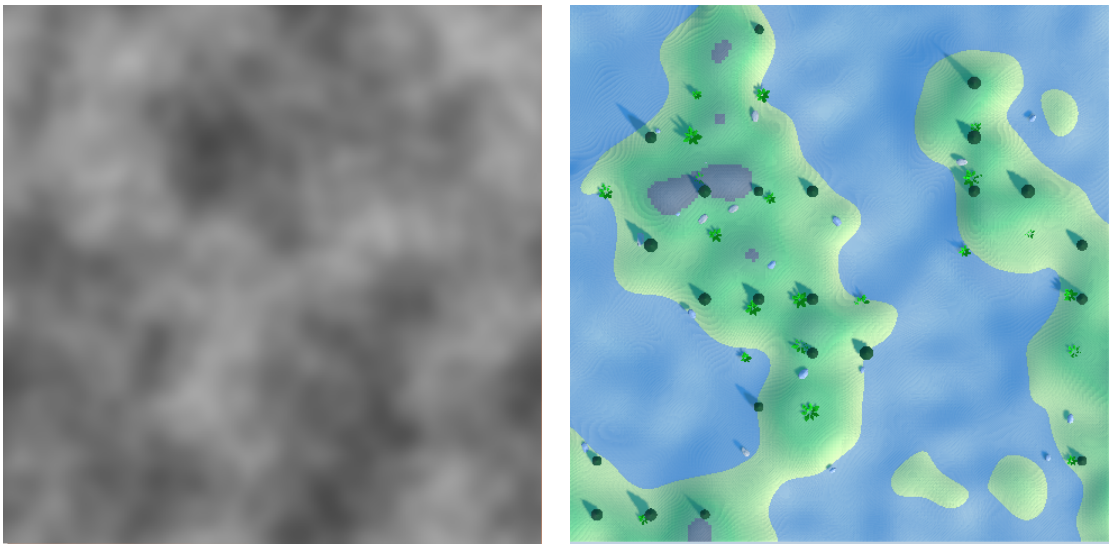


Figure 2.3. Planes vertices translated on the Y-Axis.

When we use the term height map, we actually talk about the height data of every vertex on a terrain. For convenience, this data is usually stored in a 2-D array, where each data corresponds to each vertex present on the mesh. In some cases these maps can be presented as gray-scaled images where each tone of gray represent a height value. Modern game engines feature direct import options for these images as height maps for their terrains. As an example, Figure 2.4a shows a height map generated by Open Simplex Method. This height map corresponds to the terrain which is shown in the Figure 2.4b.



(a) Heightmap generated using Simplex Noise. (b) Terrain generated by Heightmap 2.4a.

Figure 2.4. Simplex Noise Heightmap to Terrain.

Please note that we are going to cover how a height map is transformed into a terrain, and other map generation techniques in more detail at the Section 4.1.

2.2 *Computer Science*

In this section, we are going to present some technical knowledge mentioned throughout the thesis, from the field of computer science.

2.2.1 *Graph Data Structure*

In computer science, a graph is a data structure is used for storing; a list of vertices and the edges of those vertices where each edge denotes which vertex is connected to one other. A sample graph structure can be found in Figure 2.5. We can denote the graph data of this structure as following;

$$V = \{A, B, C, D, E, F\}$$
$$E = \{\{A,B\},\{A,D\},\{A,E\},\{B,C\},\{C,D\},\{C,F\},\{D,E\},\{D,F\},\{E,F\}\}$$

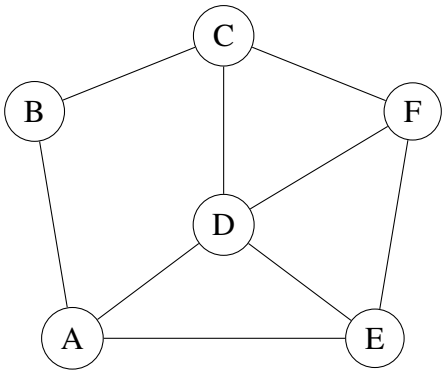


Figure 2.5. Sample Graph.

The graph in the Figure 2.5 is an undirected and unweighted graph, meaning that each edge can be used to traverse in both directions and traversing from one node to other has no cost or has a uniform cost for all edges. However in a graph, there can be rules for node traversal. If there is a cost for using an edge, we identify that graph as a weighted graph. As an example, consider the Figure 2.6 as a weighted version of Figure 2.5.

If we were to make edges of a graph one way only, in other words direct an edge to point at a certain node, then this graph is now called a directed graph. If we were to update the edges of the graph given in Figure 2.6, we would get a weighted directed graph. Consider the Figure 2.7 as an example for a weighted and directed graph.

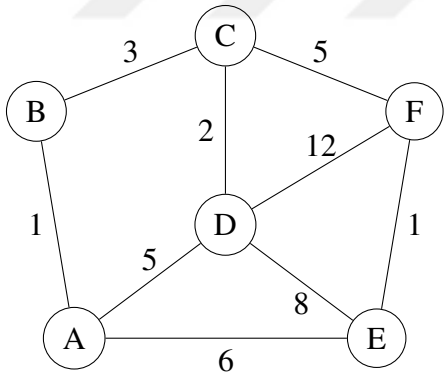


Figure 2.6. A Weighted Graph.

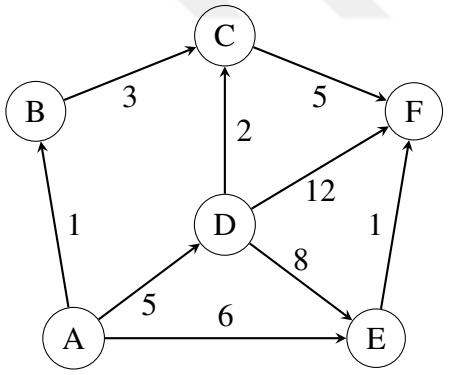


Figure 2.7. A Weighted, Directed Graph.

We can use utility functions for manipulating and accessing data of a graph. Some of these functions can be; adding/removing vertices and edges, setting/getting a vertex value, fetching all neighbors of a vertex and so on.

A graph can be used in various applications. A popular use case of graphs are for finding shortest paths between nodes. We are going to use this data structure for this purpose in this discussion. Our map will contain several regions and we expect to generate paths between these regions, which we will later use in our quest generation algorithm.

2.2.2 *Dijkstra's Shortest Path Algorithm*

Dijkstra's Shortest Path Algorithm (Dijkstra, 1959), is used for finding shortest paths between a graph's vertices. The algorithm takes a graph data, and a starting node as parameters. Then the algorithm proceeds to check every node and update their distance values with respect to the starting node. The algorithm returns a distance array, which denotes each vertices distance value to the source node. Also another array is stored and returned for keeping track of each nodes previous node, which is required if we wish to make path reconstruction. The pseudo-code for this algorithm can be found in the Algorithm 3.

Algorithm 3 Dijkstra's Shortest Path Algorithm

```
1:  Dijkstra(Graph, source)
2:  List vertices ← Empty Vertex List;
3:  Vertex current ← Empty Vertex Pointer;
4:  distance[ ] ← Empty Array;
5:  previous[ ] ← Empty Vertex Array;
6:  for each Vertex v in Graph do
7:      distance[v] ← Infinity;
8:      previous[v] ← Null;
9:      Add v to vertices;
10: end for
11: distance[source] ← 0;
12: while vertices is not empty do
13:     current ← Item in vertices with minimum distance[current];
14:     Remove current from vertices;
15:     for each Neighbor n of current in vertices do
16:         newDistance ← distance[current] + distance from n to current;
17:         if newDistance < distance[n] then
18:             distance[n] ← newDistance;
19:             previous[n] ← current;
20:         end if
21:     end for
22: end while
23: return distance, previous;
```

The distance array can be used to check quickly if the starting node is connected with any other node. If there is no connection, the distance value for that node should be infinity. Otherwise we can use Algorithm 4 to reconstruct a path between these nodes.

Algorithm 4 Constructing a path.

```
1: ConstructPath(Vertex goal, Vertex[ ] previous)
2: path ← Empty Vertex List;
3: current ← goal;
4: if previous[current] ≠ Null OR current equals source then
5:     while current ≠ Null do
6:         Add current to list path;
7:         current ← previous[current];
8:     end while
9: end if
10: return path.Reverse();
```

CHAPTER 3 : RELATED WORK

In this chapter, we present some of the past studies which relates with our thesis objective. We accept studies to be in our scope if they are in the field of quest and map generation and if these aspects are generated harmoniously. Studies which define the structure of game missions, game spaces and game levels in general are also accepted in our scope of study materials.

In the generation of maps and missions, we have the ability to prefer which aspect to generate first. We can generate missions and the map simultaneously or generate missions then map or vice versa. In the literature, there are several cases of studies implementing these generation methods for various games.

Dormans (2010) demonstrates in their article, how mission and space data can be generated using generative grammar techniques which originates in linguistics. In their methods, a series of symbols are created in order to use in the grammar data. These symbols consist of the structures which define a game level. Then these symbols are combined with each other using various grammar rules to generate strings which makes up a game level structure. A sample set of grammar rules and an alphabet for generating a mission from the article can be found in Tables 3.1 and 3.2 respectively.

Table 3.1. A sample set of grammar rules.

Source: Dormans (2010)

Dungeon	–>	Obstacle	+	treasure				
Obstacle	–>	key	+	Obstacle	+	lock	+	Obstacle
Obstacle	–>	monster	+	Obstacle				
Obstacle	–>	room						

Table 3.2. A sample set of grammar alphabet.

Source: Dormans (2010)

bl	=	boss(level)	G	=	Gate	l	=	lock
bm	=	boss(mini)	g	=	goal	lf	=	lock(final)
C	=	Chain	H	=	Hook	lm	=	lock(multi)
CF	=	Chain(Final)	ib	=	item(bonus)	n	=	nothing
CL	=	Chain(Linear)	iq	=	item(quest)	S	=	Start
CP	=	Chain(Parallel)	k	=	key	t	=	test
e	=	entrance	kf	=	key(final)	ti	=	test(item)
F	=	Fork	km	=	key(multi piece)	ts	=	test(secret)

Dormans demonstrates, modifying generative grammars to generate graphs instead of strings is possible. These graphs can later be used to generate mission data by combining an alphabet which consists of mission structures. The same logic can be applied to generate game space as well. Dormans uses shape grammar techniques which contains an alphabet like; “door”, “corridor”, “wall” and “connection” to generate space data. Then finally, combines the mission data and shape data to generate the game space data.

In another paper, Dormans (2011) uses model transformation to formulate automated content generation in the scope of level design. His methods demonstrate creating mission data, mapping it to the game space and finally refining the whole game level (as in Figure 3.1). These methods again use graph grammar techniques as we mentioned while reviewing his earlier article. He also mentions how an alternative model transformation technique would work for a “space then mission” generation (as in Figure 3.2) technique, however article does not elaborate the subject further. In our discussion, our methods for space and mission generation, fits best to the “space then mission” label.

Figure 3.1. Transformation series for “mission then space”.

Source: Dormans (2011)

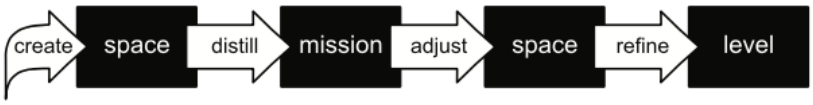


Figure 3.2. Transformation series for “space then mission”.

Source: Dormans (2011)

Dormans also argues in the same article that, reusing the same level for different quests as a base would be economic and can have game play benefits. Since each mission won’t need a brand new game space, it will save time on the level design for the developers also during game play, players need to learn fewer game levels and accommodate better in the game environment. The methods we aim to propose, supports the idea of reusing the game space for next generation missions as well.

Karavolos et al. (2015) in their article, mentions a game level generation technique using graph-based transformations, which is used in the game *Dwarf Quest* (Wild Card

Games, 2013). In this technique, again a grammar generation method is employed to generate the mission data. Then this data is represented and used as a graph data, which is then transformed into game space data. The game space data is harmonized with the mission data, then used to construct the game level. Also there is another mention of a technique called tile-based transformations, where grammar data is based on tile data instead of graph data. This technique is used in a game called TickTick++. Later on, these techniques are compared as two separate game level generation techniques, one for Dwarf Quest and the other for TickTick++. These games use both "mission then space" and "space and mission in parallel" techniques for generating game levels. Also in an other paper, Karavolos et al. (2016) proposes a search-based generation method which employs evolution techniques to alter mission graph data, which is generated by graph grammar generation techniques. In our case, even if we don't use grammar generation techniques, we transform our generated map data to a graph data to be later used in the mission generation methods.

Liapis (2017) in their paper, introduce a way to refine and evolve game levels. After generating the game level, results are maybe unsatisfactory and one might want to refine this generated game space. The methods they demonstrate in their paper can be used to refine a game space data. Also in another paper Liapis et al. (2019), describes how orchestration of content generation should be done, and how missions should cover the generated virtual space.

A popular noise generation algorithm Simplex Noise is the evolved and simplified version of the classic noise algorithm, Perlin Noise. In their paper Gustavson (2005), demonstrates how this algorithm performs, also complexity calculations for the algorithms are provided as well. As mentioned previously, we are using an open source version of this algorithm, which is called Open Simplex Noise. The algorithm is built-in and provided by *Godot Engine* (Juan Linietsky, Ariel Manzur, 2014) which we use for the implementation of our methods.

As we briefly mentioned in the Section 2.1.3, Hendrikx et al. (2013) mention how usage of PCG can be very effective in game development. They categorize the usage of PCG as follows;

- Generation of game bits like special effects such as fire, fog, clouds, explosion, land vegetation and textures which can be used in any game.
- Game spaces which can be indoors or outdoors maps, bodies of water like rivers, lakes.
- Game systems like the generation of ecosystems, cities, roads.
- Game scenarios like puzzles, quests, stories and levels overall.

- Game design which governs game rules, settings, board designs for board games.
- And other derived content like broadcasts, news, leader-boards and so on.

As we mentioned earlier, there can be similarities in the quest archetypes of some quests found in games which share the same genre. In his paper, Dickey (2007) mentions several of these quest archetypes which can be found in role playing games. He demonstrates and evaluates quest types like;

- Collection quests, where player needs to collect a variety of objects.
- Good-will quests, where a player needs to help another low level player.
- Messenger type quests where player needs to carry information from a source to a destination.
- Escorting a non-player character (NPC) quests, where player escorts an NPC while fending off enemies.
- Bounty quests, where player needs to eliminate a unit for an in game reward.

He also demonstrates the similarities of these quest types and shows how the player interacts with some of the popular quests types, again found in RP games. In our scope, we use the same quest archetypes as prototype quest types for our quest generation algorithm.

There are various mission types we can integrate games with. A popular type is the Lock-Key type of missions, where player needs to satisfy preliminary tasks to progress through to the next objective. In their paper, Ashmore and Nitsche (2007) describe how they used this mission mechanism for their quest generation algorithm, which is used in a game called Charbitat. This game takes place in a procedurally generated world, where space generation depends on the players actions. Also the game uses key and lock mechanics in their generated puzzles where, the player needs to search for the key in the game space and apply it to the lock to progress further. Similarly in our scope, we also use the key and lock mechanism in our generated quests. The quests consists of multiple objectives which take place on multiple regions. We require the player to traverse a path of these regions and complete each objective in this paths order.

The subjects we mentioned are important studies which are conducted in the field of level generation in video games. Our approach is a solution for procedural game level generation problem, which uses “space then mission” technique. Our mission generation methods can be used on maps which can be represented as a graph data. In theory, this feature will let us generate missions for indoors maps like dungeons, rooms and also outdoor maps with little to vast terrains. The methods we use feature low complexity,

which makes implementation relatively simpler for game developers. Also the quest generation method we propose can be run during game play, which gives developers flexibility in quest generation.

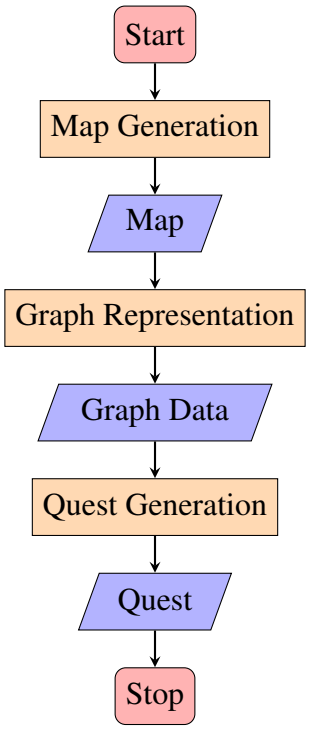
CHAPTER 4 : METHODOLOGY

Our goal on this thesis is to generate quests using the game map data. We want the player to traverse regions of the given map and do various quests while traversing them. In this chapter we are going to discuss about the methods on how we can achieve our goal.

We are going to start off by generating a map which we can experiment on. Then we will partition the map in order to create regions. We need a region structure, so we can benefit from different characteristics of various areas in the map. These characteristics will play a key role in the quest generation algorithm. Also these regions are going to be represented as a graph data so we can generate paths between them. In order to introduce characteristic abilities to the regions, we are going to generate properties for each region. These properties can be amount of light, amount of resources or enemies present in an area or many more other options depending on the game structure. We can expand these properties to fit any games genre. We are also going to discuss about how we can create quest objectives and options then we will map these to corresponding properties for each region. After all is done, we will discuss how we will generate a quest and conclude this chapter. Consider Figure 4.1 as the flow diagram of our methods.

Since we are going to use graph data to generate our quests, theoretically we can use any given map for the quest generation. The only condition being, the map data should be convertible to a graph data. Once that condition is satisfied, we can generate quest on any map by following the same procedures we discuss in this chapter.

Figure 4.1. Flow diagram of the roadmap.



4.1 *Generating a Map*

Generating a map by hand may not be necessary since most of the modern game engines provides tools for terrain generation. However if we want to generate our own map, a simple technique using a noise generation algorithm (See 2.1.4) can be as follows;

- Generating a 2-D noise data by using a favorable noise generation algorithm.
- Applying this generated data to a 2-D plane mesh (See 2.1.5) as height map. (See 2.1.7)
- Determining a water line at a height. If there is no other reason, we determine this height to be zero. Any area below this line counts as water and above counts as land.
- Coloring the terrain using the generated height map. This can be done by assigning each height range a land color. For instance a height range of $[0-m]$ can be sand color, $[m-n]$ can be grass color and $[n-t]$ can be snow and so on.
- We can also generate vegetation depending on the height of the land. If we are above the sea level, based on a random threshold we can place rocks, trees, grass and so on to fill the land.

The above technique can generate a terrain which relatively looks like a landscape without too much effort. In this thesis we are going to use this technique to generate our maps. Since our objective is to be able to generate quests in any given map, our map generation algorithm is designed in a way to generate maps based on a seed value. This seed value is passed to the noise generation algorithm which uses this value as a base to generate noise data. In other words, different seed values introduce different maps. Figure 4.2 shows the implementation of this algorithms results;

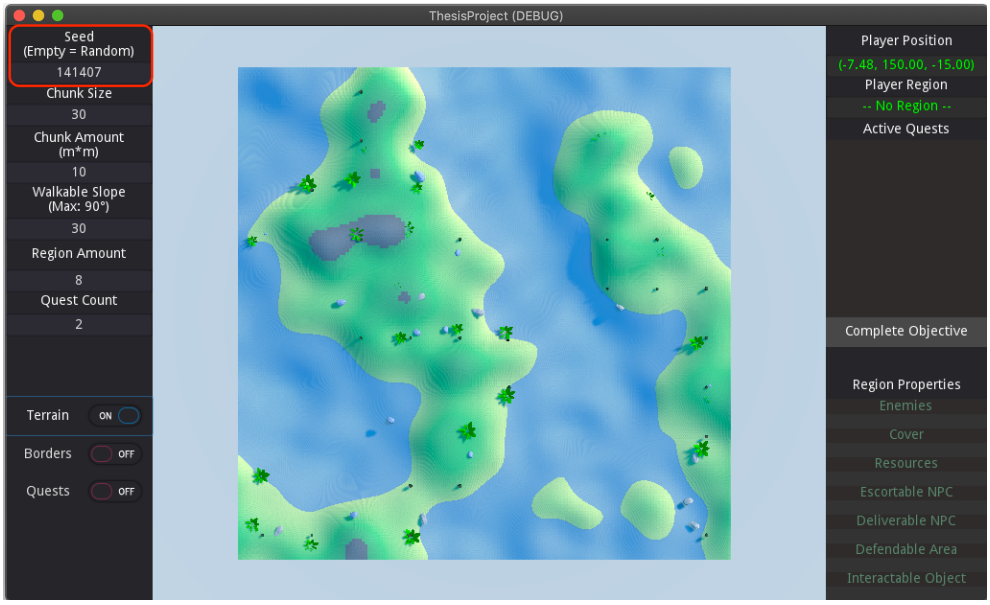


Figure 4.2. Map generation seed: 141407

Also in the Figure 4.2, we can see each height with different colors on the terrain. In this demonstration, we chose water level to be 0 and anything below 0 on the Y-Axis counts as water level. As this height increases, our terrain colors change with this height value.

4.2 *Partitioning the Map*

Either we create our own map or a game engine provides us one, we need to be able to present the map as a graph. As we mentioned earlier, this step is very important because we are going to generate paths between the regions found on the given map. First step for region creation is the partitioning of the map. In order to partition, we need to divide the map into smaller chunks. We can simply achieve this by dividing map width and map height to a favorable amount.

It would be logical to determine this amount to be a scalable value so keeping the maps dimensions but changing the chunk size and chunk amount is possible. Let's assume we determined to divide the map to $width \times height$ values of the map. No matter the dimensions, we are always going to end up with the same sized unit squares with an area of 1 unit square. For instance, assume our map width and height are both 100 units with an area size of 10,000 unit squares. If we were to divide this map to $width \times height$, we would end up with 10,000 squares, each having an area of 1 unit squares.

If we want to increase or decrease the chunk amount and size, we should introduce another variable to scale our division up or down. Assume we want to keep our map at 10,000 unit squares but we need a finer grained chunk grid. We can simply change our division formula to $width \times height \times scale$, where scale determines the size and amount

of the chunks. For instance, a scale value of 5 now yields 50,000 squares each with having an area of 0.20 unit squares. Likewise for a coarser grained grid we can update our formula to $width \times height \times 1/scale$. Now a scale value of 5 yields, 2,000 squares with each having an area of 5 unit squares.

We may determine this scale value based on our games needs. If we wanted to fit more regions inside a map without changing the dimensions, we need to divide by a larger number, thus a larger scale value. Please note that dividing to a too large number will introduce a very large chunk population. Unless we want to work with a very big population of regions, we should avoid doing so since storing and computing such population will be very resource heavy. After determining the scale, this chunk grid now forms our map. Now we can use some of these chunks to form our regions.

Notice that, we will not be able to do quests in some of these chunks. For example, some parts of the map in Figure 4.2 are below water level. If our game does not feature quests in water terrain, we may want to exclude those chunks from being selected while forming a region. This is an optional step but may become necessary if we want to discard unworkable terrain from our region selection. We can also discard steep and mountainous areas from our workable terrain set as well. We can do this by determining a maximum walkable slope for our player, then we can calculate the average slope of each chunk by looking up mesh normals for that chunk. If the calculated slope is larger than the maximum walkable slope, we can discard that area from region selection as well. We can simply identify usable chunks as 1 and unusable ones as 0. Then we can form a 2-D matrix of these 1's and 0's, which leaves us with a grid of the map. Applying our partitioning method on the map in Figure 4.2 yields a result as shown in the Table 4.1.

Table 4.1. Partitioned map as grid.

0	0	0	1	0	0	0	0	1	1
0	0	1	1	1	0	0	0	0	1
1	1	1	1	1	1	0	0	0	1
1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	0
0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0	0
0	0	1	1	1	0	0	0	1	0
0	0	0	0	1	1	1	1	1	1

This data represents the usable terrain as 1 and unusable terrain as 0. But if we want

to use the same grid for regions as well, we might want to change these 0's and 1's to some other integer value since regions will be identified with numbers starting from 0 as well. We can simply change these values to -2 and -1 for 0 and 1 respectively. Once we update the grid, we get the data in the Table 4.2;

Table 4.2. Partitioned map grid updated.

-2	-2	-2	-1	-2	-2	-2	-2	-1	-1
-2	-2	-1	-1	-1	-2	-2	-2	-2	-1
-1	-1	-1	-1	-1	-1	-2	-2	-2	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-2	-2	-2	-1	-1	-1	-1	-1	-1	-2
-2	-2	-2	-2	-2	-1	-1	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-1	-1	-1	-1	-2	-2	-2	-2	-2
-2	-2	-1	-1	-1	-2	-2	-2	-1	-2
-2	-2	-2	-2	-1	-1	-1	-1	-1	-1

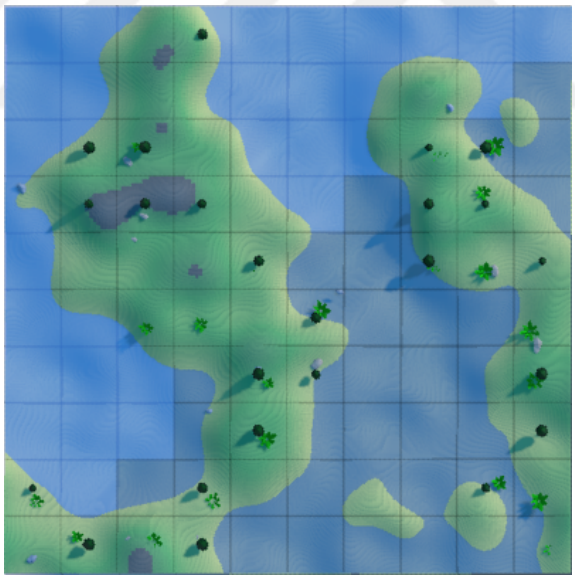


Figure 4.3. Game map divided into smaller chunks.

In Figure 4.3 the partitioned version of our game map can be found. After separating usable terrain from unusable, now we can work with usable chunks to create our regions.

4.3 *Creating Regions*

Creation of regions can be done with a various number of methods, but a simple technique can be;

- Creating one cell for each region on a random usable cell at start.
- On each iteration, each region adds usable neighboring cells to their territories.
- Repeat until a there are no more growth in the territories of the regions.

Table 4.3 shows the map from Table 4.1, altered with randomly selected 5 usable cells as regions starting points. After selecting these points, we can now spread our regions onto neighboring usable cells. Spreading is done by checking if neighboring cell is usable and not taken by any other region. If this criteria is satisfied, we can convert the neighboring cell into a region cell. We do this each iteration for every region present on the map.

Table 4.3. Region Creation at Iteration 0.

-2	-2	-2	-1	-2	-2	-2	-2	-1	-1
-2	-2	-1	-1	4	-2	-2	-2	-2	-1
-1	-1	-1	-1	-1	-1	-2	-2	-2	-1
-1	-1	-1	-1	-1	-1	-1	-1	3	-1
-2	-2	-2	-1	-1	-1	-1	0	-1	-2
-2	-2	-2	-2	-2	-1	-1	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-1	-1	-1	-1	-2	-2	-2	-2	-2
-2	-2	-1	-1	-1	-2	-2	-2	-1	-2
-2	-2	-2	-2	2	-1	-1	-1	1	-1

Table 4.4 shows the resulting data of our spread algorithms first iteration. The stopping condition for this algorithm can be any number of iterations reached, however it would be logical to stop when there are no more record of region growth.

Table 4.4. Region Creation at Iteration 1.

-2	-2	-2	-1	-2	-2	-2	-2	-1	-1
-2	-2	-1	-1	4	-2	-2	-2	-2	-1
-1	-1	-1	-1	4	-1	-2	-2	-2	-1
-1	-1	-1	-1	-1	-1	-1	0	3	3
-2	-2	-2	-1	-1	-1	-1	0	-1	-2
-2	-2	-2	-2	-2	-1	-1	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	-1	-1	-1	-1	-2	-2	-2	-2	-2
-2	-2	-1	-1	2	-2	-2	-2	1	-2
-2	-2	-2	-2	2	-1	-1	-1	1	-1

On this instance, the spread algorithm concluded on Iteration 12. Yielding data can be observed in the Table 4.5. We can also see the results on the game map as well in the Figure 4.4.

Table 4.5. Region Creation concluded at Iteration 12.

-2	-2	-2	4	-2	-2	-2		3	3
-2	-2	4	4	4	-2	-2	-2	-2	3
4	4	4	4	4	0	-2	-2	-2	3
4	4	4	0	0	0	0	0	3	3
-2	-2	-2	0	0	0	0	0	3	-2
-2	-2	-2	-2	-2	0	0	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	-2	-2	-2
-2	2	2	2	2	-2	-2	-2	-2	-2
-2	-2	2	2	2	-2	-2	-2	1	-2
-2	-2	-2	-2	2	1	1	1	1	1

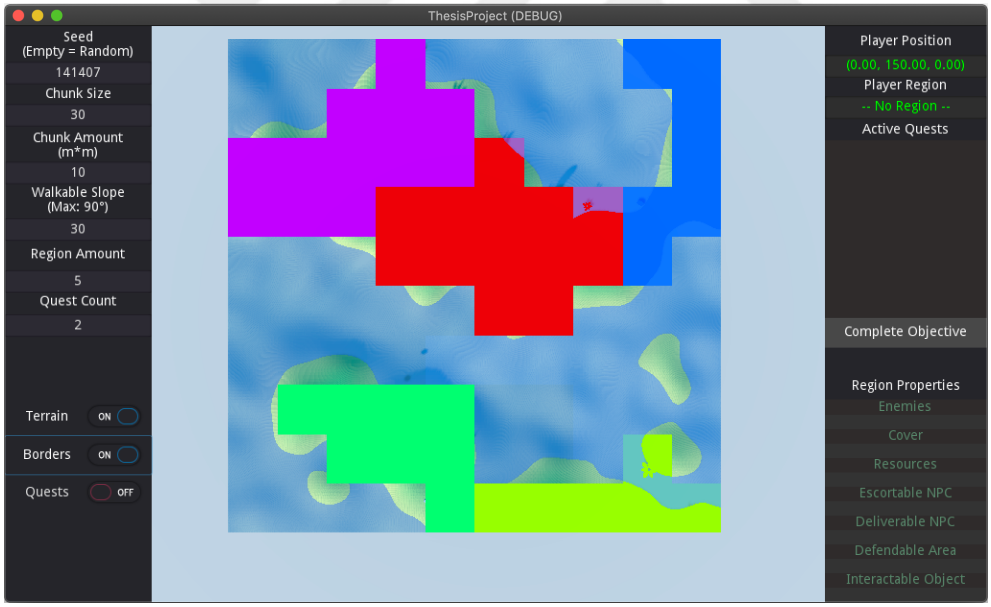


Figure 4.4. Region Generation Halted.

Note that region creation is limited by the available cells on the map. Meaning, if we tried to create more regions than available cells, we would only get a region number which is equal to the number of available cells on the map. On such scenario, regions growth would not be possible since initially every available cell is already taken by a region.

4.4 *Creating a Region Graph*

After we partitioned the map and created regions, now we need to hold the record of which region is connected to which, since we need to create traversable paths for our quest generation algorithm. In order to track the connectedness of maps regions, we can easily create a graph. It would be logical to represent each region as a vertex and each side connecting two regions as an edge.

To add vertex data to the graph, we give each region a unique identifier number. Via using this identifier we add each region to the graph as a vertex only once. For example, for the region data in Table 4.5 our vertices will be;

$$V = \{0,1,2,3,4\}$$

To add edge data to the graph, we can use the method given in Algorithm 5.

Algorithm 5 Algorithm for storing edge data.

```
List edges ← Empty Edge List;
Grid ← All of the cells in the map.
for each Cell C in Grid do
    if C is a region cell then
        for each Neighbor N of C do
            if N is a region cell with different ID than C then
                if Edge(C,N) OR Edge(N,C) ∉ edges then
                    Add Edge (C,N) to the list edges
                end if
            end if
        end for
    end if
end for
```

Above algorithms ensure all regions are stored in the graph as vertices, and their edge data is stored in the *edges* list. Once concluded, this algorithm yields the following graph data for our map in Table 4.5;

Vertex Count: 5

Edge Count: 3

$$V = \{0,1,2,3,4\}$$

$$E = \{\{4,0\},\{0,3\},\{2,1\}\}$$

4.5 *Quest Objectives and Options*

There are various genres of games and each game can have their own style of quest objectives which is completely different from other games. However in many games there are fundamental quest objectives, similar to games which share the same genre. As we discussed in the Chapter 3, Dickey (2007) mentions in their article about some of the popular quest archetypes of Role Playing Games. These archetypes can be commonly found in many RP games. Some of them are as follows;

- Gather a number of items.
- Eliminate a number of creatures.
- Deliver item(s) to a Non-Player-Character (NPC).
- Escort an NPC.
- Explore an area.
- Use a quest item in a specific place.

These types of quests are the ones we are going to generate for our map. We are going to create a pool of these quest objectives and determine which of these objectives are best suitable for the regions that we are going to traverse. The suitability of an objective for a region is going to be determined by checking the regions properties, which we are going to discuss further in the Sections 4.6 and 4.7.

In RP games, fundamental quests are repeatedly offered to players throughout the scenario. However game designers change some of the aspects of these quests to make completing quests less repetitive. Sometimes they introduce an optional addition to the quest. When completed, this option may grant additional rewards to the play. In our case for instance, a quest objectives of *Deliver Item(s) to an NPC* can be transformed into, *Deliver Item to an NPC without being seen by the enemy* or *Gather a number of items without getting damaged*. These optional additions change the mechanic of the quests and make completing quests more interesting. However optional, we can also create a pool of these quest options and make additions to the quest objectives, again with respect to the availability of the map.

For demonstration purposes, let us create a list of quest objectives and options as follows;

Objectives;

- None
- Eliminate
- Deliver
- Escort
- Gather
- Defend Area
- Interact
- Complete

Options;

- None
- Stealth (without being seen)
- Complete

Notice that we have introduced *None* and *Complete* types for both objectives and options. These types will help identify which state of the quest we are currently on. An option state of *None* means there are no optional additions to the objective. An objective state of *None* however, means there is no objective, therefore no quest for that area. An option type of *Complete* means the optional addition of the objective is now complete, if player finishes the quest at this state, we can reward them for completing that option. Likewise an objective type of *Complete* means, the quest is now complete.

4.6 *Region Properties*

In most cases, quests need an interactable object which is placed on the map. This may be a resource to be collected, an NPC to be interacted with, or an enemy to be dealt with. For instance, if our quest pool features a quest where we gather a number of items, those items should be available on the map. Another instance can be an explore the area type of quest, where our map needs to feature a trigger component checking if the player entered the exploration area.

Before we can generate a quest objective, we need to know if that objective is supported in that region. For instance, we don't want to send our player to "Collect 10 Flowers", if there are no flowers to collect in that area. So this way, we can define "Flowers" as a property for that region. Also any property may vary in size, meaning there can be none, a little or plenty available in the region. We need to specify the amount of these properties for each region as well.

For demonstration, let us create a region property list which corresponds to objectives

and options we created in Section 4.5. For the *Eliminate* objective we need enemies present on the map. Likewise, for *Deliver* objective, a deliverable NPC, for *Escort*, an escortable NPC, for *Gather* objective, resources on the map, for *Defend Area*, a defendable area and enemies to defend from, and finally for *Interact*, we need an interactable object present on the map. Also for *Stealth* option, we need enemies and cover properties available on the map. So to summarize, the regional properties are going to be as follows;

Region Properties;

- Enemies : *Range[0-n]*
- Cover : *Range[0-n]*
- Resources : *Range[0-n]*
- Escortable NPC : *Range[true-false]*
- Deliverable NPC : *Range[true-false]*
- Defendable Area : *Range[true-false]*
- Interactable Object : *Range[true-false]*

Notice some of the properties given above has a value range of *0 to n*, while some of them are Boolean's, meaning can be only *true* or *false*. This is convenient since enemies and resources may come in different quantities, while an escort-able NPC and an interactable object may exist is the area or not.

4.7 *Quest Objective and Option Selection*

Each quest objective and option may have one or more regional attributes corresponding to them. We have discussed how these attributes can be determined in the Section 4.6. A region can support none or many quest types at once. Our selection method will pool up the available objectives and options before any selection happens. If there are items present in the pool, we are going to make a random selection from the pool and assign that objective and option to the region. If the pool is empty, it means there are no available quest objectives for our region. Meaning we are going to skip assigning a quest to the area, which is always allowed.

Let's assume we have the following information;

Objectives

- Gather X and Y.
- Eliminate Z.
- Explore area W.

Options

- Stealth.
- Without getting damaged.

Region Properties

- X : *Range[0-n]*
- Y : *Range[0-n]*
- Z : *Range[0-n]*
- W : *Range[true-false]*
- Light : *Range[0-n]*
- Cover : *Range[0-n]*

Notice that for each objective and option we have a matching property. The method for selecting an objective and option is case dependent. Meaning that each case of quest objectives and options should be mapped to the regional properties by hand. For demonstration, using the data above, we can map quest objectives and options to the properties with the logic provided at Algorithms 6 and 7.

Algorithm 6 Objective selection.

```
1: ObjectiveSelection(Properties properties, QuestObjective objectives)
2: List availableObjectives ← Empty Quest Objective List;
3: if properties.X.Amount ≥ objectives.X.Threshold then
4:     if properties.Y.Amount ≥ objectives.Y.Threshold then
5:         Add "Gather X and Y" to availableObjectives.
6:     end if
7: end if
8: if properties.Z.Amount ≥ objectives.Z.Threshold then
9:     Add "Eliminate Z" to availableObjectives.
10: end if
11: if properties.W.Amount ≥ objectives.W.Threshold then
12:     Add "Explore area W" to availableObjectives.
13: end if
14: selectedObjective ← Empty QuestObjective Pointer;
15: if availableObjectives is not empty then
16:     selectedObjective ← Random element from List availableObjectives;
17: end if
18: return selectedObjective;
```

Algorithm 7 Option selection.

```
1: OptionSelection(Properties properties, QuestOption options)
2: List availableOptions ← Empty Quest Option List;
3: if properties.Light.Amount ≤ options.Light.Threshold then
4:     if properties.Cover.Amount ≥ options.Cover.Threshold then
5:         Add "Stealth" to availableObjectives.
6:     end if
7: end if
8: if properties.Z.Amount ≥ options.Z.Threshold then
9:     Add "Without getting damaged" to availableObjectives.
10: end if
11: availableOptions ← Empty QuestOption Pointer;
12: if availableOptions is not empty then
13:     availableOptions ← Random element from List availableOptions;
14: end if
15: return availableOptions;
```

In the Algorithm 6, we started by determining a precondition for a possible *Gather X and Y* objective selection. With a simple logic, we check if the properties of the region we are in contains sufficient amount of *X* and *Y* objects. If the answer is true, we add this objective of *Gather X and Y* to the pool of available objectives. Similar to this logic, we further demonstrated how other objectives can have preconditions for selection as well. After pooling all objectives, if there are any, we pick one of the possible candidates from our pool and assign it to be one of our objectives for our quest. The same logic applies to the quest options as well which we demonstrate in Algorithm 7.

Notice the above algorithm do not always return a quest for the given properties, also when it does, it does not always provide a quest option. This is intentional since not every region can support a quest and not every quest will have optional additions. Also notice that selection is made completely random from the available pools. This is because a non-deterministic selection will yield a greater variety in objectives and options throughout the map.

4.8 *Path Generation*

Before we move on further to quest generation, we need to discuss how we should generate paths for our quests. In the Section 4.4, we discussed how to generate a graph from our maps regions. Now we will use Dijkstra's Shortest Path Algorithm for creating paths between nodes of our graph. We have covered briefly how the algorithm works in

the Section 2.2.2. We have also provided the algorithms pseudo-code at Algorithm 3.

Using the algorithm, we can quickly check if the source and the destination nodes are connected by looking up the *distance* array returned by the algorithm. If the distance value of the node we are trying to reach is infinity, then there is no apparent path to that node from our source node. However if two nodes are connected, we can generate a path between them by back-tracking from the goal node using the algorithm provided at Algorithm 4. For demonstration purposes, assume we have the grid map in Table 4.6.

Table 4.6. Sample map for path generation.

-2	-1	-1	-2	-2	-2	-2	-2	-2	-2
-2	-2	-1	-2	-2	-2	-2	-2	-2	-2
-2	-2	-2	-2	-2	-2	-2	3	3	3
-1	-2	-2	-2	-2	-2	3	3	3	3
-1	-2	-2	-2	-2	3	3	3	3	3
-1	-2	-2	-2	-2	3	3	3	-2	-2
-2	-2	-2	-2	0	-2	3	3	-2	-2
-2	-2	-2	0	0	1	1	-2	-2	-2
-2	-2	4	4	2	1	1	-2	-2	-2
-2	-2	4	2	2	2	-2	-2	-2	-2

Generating a graph out of this grid yields the graph data as follows;

$$V = \{0,1,2,3,4\}$$
$$E = \{\{3,1\},\{0,4\},\{0,1\},\{0,2\},\{4,2\},\{1,2\}\}$$

For a test run, let us use the Dijkstra's Shortest Path algorithm to generate a path from node 4 to node 3. Initially we have an empty list of vertices, a distance array for denoting each vertices distance to source and finally, a vertex array for storing previous vertex data for each vertex. The algorithm's output is provided at Listing 4.1.

Algorithm concludes when there are no more vertices in the vertices list. As we can see at Listing 4.1, distance for the node we are trying to reach is not Infinity, meaning there is a path between the source and destination nodes. We can use the Algorithm 4 to construct a path between these nodes. The construction methods output is provided at Listing 4.2.

Listing 4.1: Dijkstra's Algorithm Output

```
Initially;
current = Null
distance[] = {Infinity,Infinity,Infinity,Infinity,0}
prev[] =  {Null,Null,Null,Null,Null}
vertices = {0,1,2,3,4}
-----------
Iteration 0;
current = 4
distance[] = {1,Infinity,1,Infinity,0}
prev[] =  {4,Null,4,Null,Null}
vertices = {0,1,2,3}
-----------
Iteration 1;
current = 2
distance[] = {1,2,1,Infinity,0}
prev[] =  {4,2,4,Null,Null}
vertices = {0,1,3}
-----------
Iteration 2;
current = 0
distance[] = {1,2,1,Infinity,0}
prev[] =  {4,2,4,Null,Null}
vertices = {1,3}
-----------
Iteration 3;
current = 1
distance[] = {1,2,1,3,0}
prev[] =  {4,2,4,1,Null}
vertices = {3}
-----------
Iteration 4;
current = 3
distance[] = {1,2,1,3,0}
prev[] =  {4,2,4,1,Null}
vertices = {}
```

Listing 4.2: Path Reconstruction Output

```
Initially;
path = {}
current = 3
-----------
Iteration 0;
path = {3}
current = 1
-----------
Iteration 1;
path = {3,1}
current = 2
-----------
Iteration 2;
path = {3,1,2}
current = 4
-----------
Iteration 3;
path = {3,1,2,4}
current = Null
```

Notice the path returned in Listing 4.2 is in reverse order. A flip operation on the list yields us the correct path data;

$$path = \{4,2,1,3\}$$

Note that if the region graph we are dealing is a very large one, and its taking a lot of time to compute a path between nodes, one may use the A* Search Algorithm (Hart et al. (1968)) instead of Dijkstra's Shortest Path Algorithm.

4.9 *Quest Generation*

We have covered most of the auxiliary methods above. Now we can proceed to discuss how we can actually generate a quest. A quest needs to hold the information of;

- A path data which consists of regions to be traversed.
- Quest objectives for each region on the path.
- Quest options for each region on the path.
- An identifier value for identifying and comparing quests.

Basically, we need to start by generating a path of regions which are going to be traversed for that quest. Then we pass each regions property data along to generate quest objectives and options for the region. We already covered how the generation of those aspects can be done, so we can combine all of those techniques to generate a quest with the algorithm provided at Algorithm 8.

Algorithm 8 Quest Generation.

```
1: GenerateQuest(Region[ ] path, int identifier)
2: Properties properties ← Empty Properties pointer;
3: QuestObjective[ ] objectives ← Objectives array;
4: QuestOption[ ] options ← Options array;
5: for int i ← 0; i → path.Length() do
6:     properties ← path[i].Properties;
7:     objectives[i] ← GenerateObjective(properties);
8:     options[i] ← GenerateOption(properties);
9: end for
10: return new Quest(path,objectives,options,identifier);
```

Notice that if we were to run this algorithm for several times, we would eventually get the same region for quest generation more then once. There is no problem here, however we may want to avoid generating same quests for the same region over and over again. For instance, assume we assigned "Gather 10 Resources" quest to a region. Our player carried out the task and after a while came back to the same region for another quest. Chances are there can be another quest just like the one he/she finished. To avoid this situation, we can use 2 copies of region properties for each region, namely P_0 and P_1. Then we can use P_0 to update properties, right after an objective assignment, which can deplete the property and prevent another objective to be assigned in the same region. For instance, assume an objective of "Gather Resources" is assigned to the region. Right after this assignment we can deplete the Resources property of P_0. Our quest generation algorithm will check P_0 and upon realizing there aren't enough Resources on this region, it will not assign another "Gather Resource" objective to this area. Notice that P_1 remains untouched during the quest generation phase. The reason is, we want to use P_1 to display the properties during the game play time. Meaning that P_1 will only be updated once the objective is actually completed by the player. If we do not wish to prevent quest repetition on same regions, we can just use Algorithm 8 without change.

Let us observe the results of the implementation in the Figure 4.5. We can see that our quest generation algorithm generated a path from region 0 to 4 and based on the properties of the regions, assigned quest objectives and options. We can see that our quest features following objectives and options;

1. *Deliver* objective with a *Stealth* option at region 0.
2. *Interact* objective at region 2.
3. *Defend Area* objective at region 4.

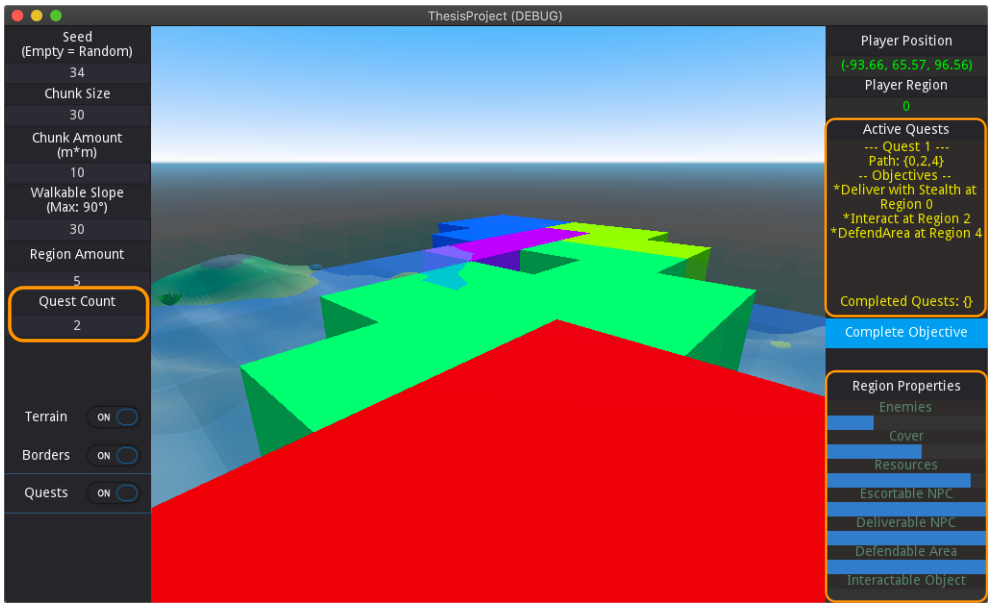


Figure 4.5. A generated quest.

Completing an objective updates the quests objective status and the properties for that region. Notice that in the Figure 4.6, we have completed the *Deliver* objective and now the state for that objective is updated to *Complete*. Also notice that *Deliverable NPC* property of that region is now depleted. As we mentioned earlier, to prevent quest repetition on same region we update this property in P_0 after quest assignment, while P_1 stays unchanged for displaying the current status of region properties during play time.

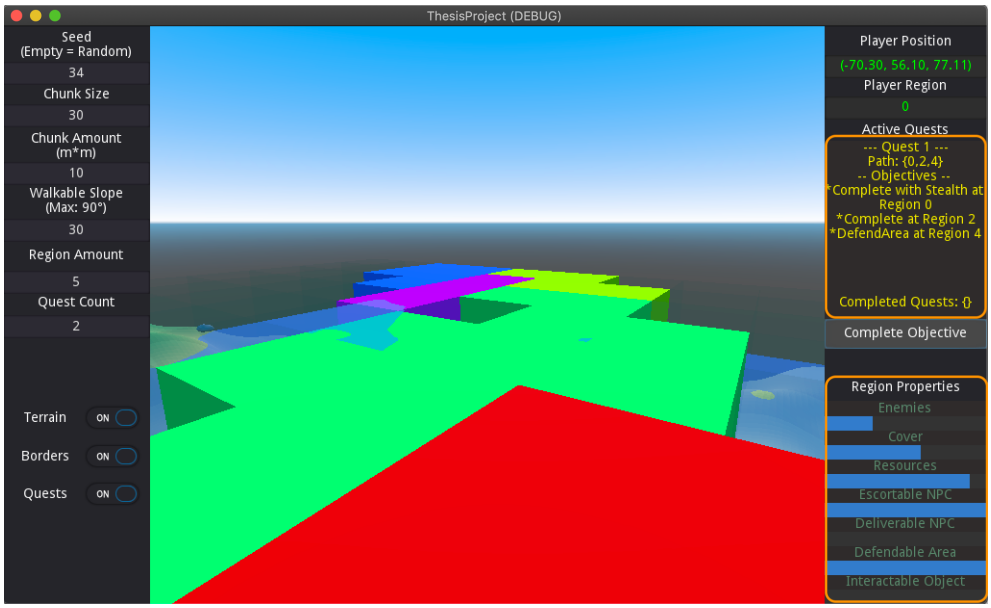


Figure 4.6. Completing objectives on quest.

After completing all objectives of a given quest, we consider that quest to be completed, we can move on to another quest. In the Figure 4.7 we can see that as player

we are tasked another quest which consists of another path with different regions and objectives.

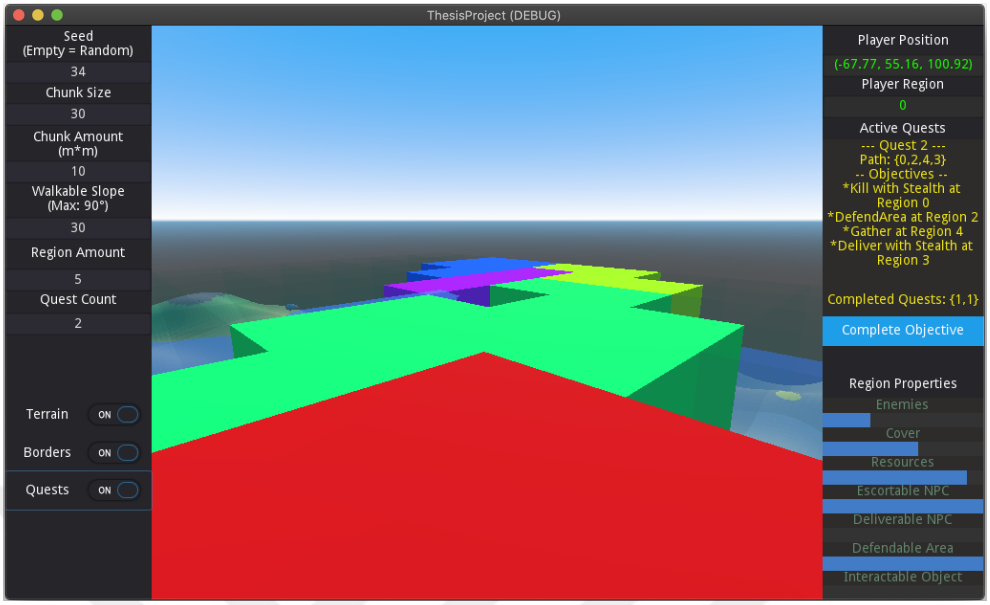


Figure 4.7. Moving to another quest.

4.10 *Complexity Analysis*

The map generation technique we mentioned in Section 4.1, is based on the generation of a 2 dimensional grid. Later on, we apply noise data to this grid which is produced by Open Simplex Noise algorithm. As Gustavson (2005) suggests in their paper, this algorithm performs in $\mathcal{O}(n)$ for given n dimensions. Therefore a map with $n \times n$ grid, should have a time and space complexity of $\mathcal{O}(n^2)$.

The algorithm for creating regions we discussed in Section 4.3, starts by randomly selecting m regions on available cells from a 2 dimensional grid. This selection can be made in $\mathcal{O}(n^2)$, since in the worst case, all of the grid can be made up of available cells, which is $n \times n$. After this step, we check every regions neighboring cells and annex the cells if they are available. For this to happen, we check the grid for available cells for m regions, meaning $m \times n \times n$ checks are made. Notice that in a scenario, if we tried to create $m = n \times n$ regions, we would have to select all available cells in the selection phase. Since every cell is taken by a region, regions can not grow over time because there are no cells available to grow into. So the growth algorithm would stop on the first iteration for that scenario. Therefore, the worst case for this algorithm is with $m \times n \times n$ checks made. Since m is significantly lesser than n, ultimately algorithm performs in $\mathcal{O}(n^2)$.

The algorithm for graphing the regions we previously mentioned in Section 4.4, performs in $\mathcal{O}(n^2)$. We check each cell available in the grid and their neighbors. This makes $m \times n \times n$ checks in total, where m is the neighbor size of each cell (i.e. north, east,

south and west neighbors of a cell) and n denotes cell size respectively.

The quest generation techniques we mentioned in this chapter, relies on the size of the region graph and how Dijkstra's Shortest Path algorithm performs in the given graph. Dijkstra's algorithm performs in $\mathcal{O}(n^2)$ where n denotes the number of nodes present in the graph. The algorithm for quest objective and option selection however performs in $\mathcal{O}(n)$, since we need to process selections for n regions present in the path of a quest. As we increase the region size, we should see a growth in time complexity of $\mathcal{O}(n^2)$, since Dijkstra's algorithm is the base speed limit for our quest generation algorithm. However, if we were to keep a fixed region size and increase quest count over iterations, we would see a linear growth in time complexity. The reason is, the calculations required for generating a path is already done for that region data. Only operation required for generating a path for given two nodes is reconstructing a path, which performs in $\mathcal{O}(n)$. Therefore, quest generation on a fixed region size should yield a time complexity of $\mathcal{O}(n)$.

We have covered our proposed methods and their implementations in this chapter. The implementation of the methods have delivered the behaviour we expected from our goal in this discussion. We can now move on further to discuss about the experimental results in the Chapter 5.

CHAPTER 5 : EXPERIMENTAL RESULTS

For the implementation of the methods we mentioned in Chapter 4 we used *Godot Engine* (Juan Linietsky, Ariel Manzur, 2014), which is an open-source game engine. Our scripting language of choice was C#. In this chapter we are going to run various tests to analyse how the methods we mentioned in Chapter 4 performs when implemented.

Unfortunately, we did not have access to the applications of the past studies which we can use to compare our results with. However, we may test our algorithms by running them with changing certain parameters. For the tests we mention in this chapter, we used the following operating system and hardware;

- OS : Mac OS Catalina Version 10.15.5
- Processor: 2,8 GHz Quad-Core Intel Core i7
- Memory: 16 GB 2133 MHz LPDDR3
- Graphics: Radeon Pro 555 2 GB - Intel HD Graphics 630 1536 MB

5.1 *Benchmarks*

5.1.1 *Sample Output*

Let us observe the sample output of our implementation by generating the same map we used in our demonstrations in the Figure 4.2, with the following parameters.

- Map Seed : 141407
- Chunk Size : 30
- Chunk Amount : $10 * 10$
- Area : 3,000 unit squares.
- Max. Walkable Slope : 30
- Region Amount : 5
- Quest Count : 2

The output we get from the resulting map is given in the Listing 5.1. The resulting data contains results of the methods we discussed in the Chapter 4. As we can see the data contains the region data, graph data and generated quests data.

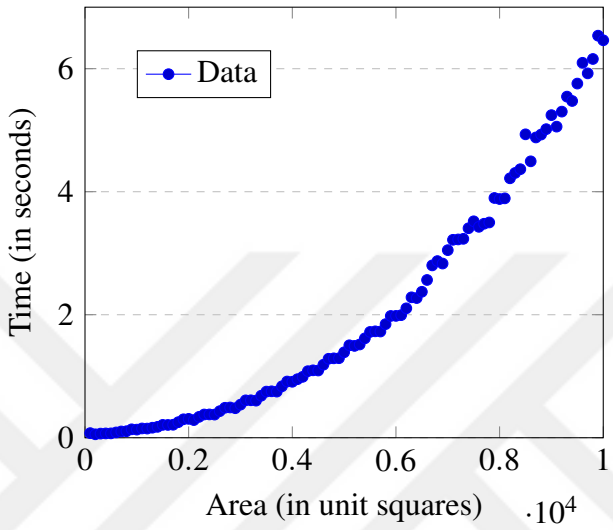
Listing 5.1: Map generation output

```
-- Generating Map --
Seed: 141407
--Region Map--
[-2 ][-2 ][-2 ][ 1 ][-2 ][-2 ][-2 ][-2 ][ 2 ][ 2 ]
[-2 ][-2 ][ 1 ][ 1 ][ 1 ][-2 ][-2 ][-2 ][-2 ][ 2 ]
[ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][ 1 ][-2 ][-2 ][-2 ][ 2 ]
[-2 ][-2 ][ 1 ][ 0 ][ 0 ][ 1 ][ 0 ][-2 ][ 2 ][ 2 ]
[-2 ][-2 ][-2 ][ 0 ][ 0 ][ 0 ][ 0 ][ 0 ][ 2 ][-2 ]
[-2 ][-2 ][-2 ][-2 ][-2 ][ 0 ][ 0 ][-2 ][-2 ][-2 ]
[-2 ][-2 ][-2 ][-2 ][-2 ][-2 ][-2 ][-2 ][-2 ][-2 ]
[-2 ][ 4 ][ 4 ][ 4 ][-2 ][-2 ][-2 ][-2 ][-2 ][-2 ]
[-2 ][-2 ][ 4 ][ 4 ][ 4 ][-2 ][-2 ][-2 ][ 3 ][-2 ]
[-2 ][-2 ][-2 ][-2 ][ 4 ][ 4 ][ 4 ][ 4 ][ 3 ][ 3 ]
--Region Graph--
Vertex Count: 5
Edge Count:3
V = {0, 1, 2, 3, 4 }
E = {1-0, 0-2, 4-3 }
-- Completed Generating Map --  Elapsed Time: 0.584 seconds.
--- Quest 1 ---
Path: {2,0,1}
-- Objectives --
*Escort with Stealth at Region 2
*DefendArea with Stealth at Region 0
*Interact at Region 1
--------------Elapsed Time: 0.004 seconds.
--- Quest 2 ---
Path: {1,0,2}
-- Objectives --
*DefendArea at Region 1
*Escort with Stealth at Region 0
*Kill with Stealth at Region 2
--------------Elapsed Time: 0.002 seconds.
```

5.1.2 *Map Generation*

We have tested our map generation algorithm by increasing the map size gradually and keeping elapsed time for each generation. The resulting plot data can be found in Figure 5.1. In this benchmark, quest generation is not added to the equation. Also region size is fixed at 5 and does not change throughout the generations.

Figure 5.1. Map Generation: Map size over time

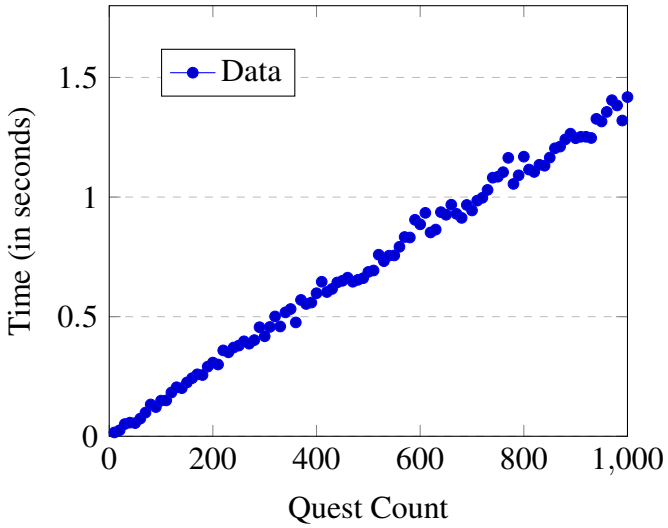


As we can observe from the Figure 5.1, the time complexity of our map generation algorithm grows exponentially as the size of the map we generate gets incremented.

5.1.3 *Quest Generation*

On our next benchmark, we test the quest generation algorithm by gradually increasing quest count each generation while keeping time elapsed. In this benchmark region size is fixed at 100 and does not change throughout the generations. The plot data for the benchmark is given in the Figure 5.2

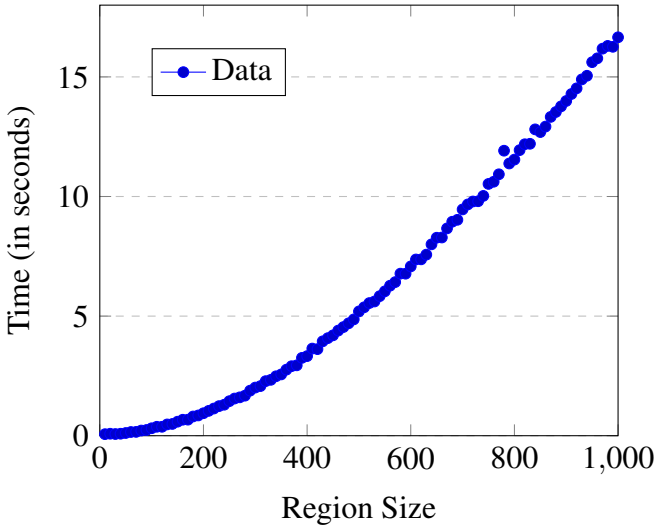
Figure 5.2. Quest Generation: Quest count over time



As we can deduce from the Figure 5.2, there is a linear growth in time complexity as the quest count increases with each generation. We expected to see such plot since, our quest generation algorithm does not feature overheads as the quest count improves without altering region size.

For another benchmark, we can run a test for increasing the regions size and generating a fixed size of quests for each generation. For this test we determined a fixed quest size of 100, which does not change through iterations. The plot data is provided at Figure 5.3.

Figure 5.3. Quest Generation: Region size over time



We can observe the plot data provided in Figure 5.3, suggesting that as region size increases, our quest generation algorithms time complexity exponentially grows.

For a final benchmark, we can increment all of the variables; map size (area in unit

squares), region size and quest count at each iteration, while keeping time elapsed and static memory use for each generation. Data for this benchmark is provided in the Table . D.1 in the Appendices section of the thesis.

We have presented some of the data from our benchmark tests which we used to test our algorithms. These data can be used to approximate, how our algorithm performs when we change some of the parameters of our methods for mission and level generation. We may now proceed to conclude our studies in the next chapter.

CHAPTER 6 : CONCLUSION

In the previous chapters, we have discussed how procedural map and quest generation is handled in the past studies. We have demonstrated our methods which can procedurally generate a map and we further discussed how we can generate quests on this generated game level. We have implemented the methods we discussed and run several tests to see how they perform in general. We have seen that, our algorithms can provide a solution for the procedural mission and space generation problem, which is the aim of this thesis.

One advantage of using our methods can be that, it would be relatively easy to integrate to any game. As we have stated earlier, our quest generation methods use graph data. This enables us to generate quests for any given map that can be represented as a graph. Another advantage can be, quest pools and regional properties can be expanded to fit any genre, meaning any game featuring quests can benefit from the methods we demonstrated. Also the methods we have covered overall do not feature high complexity. Understanding and implementing these methods can be relatively easy, beneficial for small groups or individual developers. Finally, our quest generation method can work during the game play. Developers can choose to generate quests anytime they want, which is a great flexibility.

One limitation for our methods can be, as we pointed out earlier, if the given map thus the graph is too large, Dijkstra's Shortest Path algorithm may take a very long time to generate paths between nodes. In this case, one may want to switch this algorithm with A* Search Algorithm, which is a Best First Search algorithm, utilizing a heuristic function that can generate paths faster than Dijkstra's Shortest Path algorithm. Also another limitation can be, quest pool and regional properties that we use to generate quests needs to be written by hand. If a quest depends on one or many regional properties, one needs to specify those rules and feed it to the algorithm. Unfortunately there is no method to automate that task.

In the future, one may want to further deepen the research in this area. It would be beneficial to make effort to study a version of procedural quest generation where each quest generated can tell some part of story of the game. Since almost every game has a story to tell, generated quests featuring part-storylines would make developers jobs a lot easier.

REFERENCES

Apple (2008), *App Store*, [Online] iOS, USA: Apple Inc.

Ashmore, C. and Nitsche, M. (2007), *The quest in a generated world*. [Online] Available at: https://www.researchgate.net/publication/228771322_The_quest_in_a_generated_world (Accessed: 04 September 2020).

Bethesda (2011), *The Elder Scrolls V : Skyrim*, [Online] PC, USA: Bethesda Softworks Inc.

Blizzard (1996), *Battle.Net*, [Online] PC, USA: Blizzard Entertainment Inc.

Blizzard (2004), *World of Warcraft*, [Online] PC, USA: Blizzard Entertainment Inc.

Blizzard (2012), *Diablo III*, [Online] PC, USA: Blizzard Entertainment Inc.

CD Projekt Red (2016), *The Witcher 3 : Wild Hunt*, [Online] PC, Poland: CD Projekt Red Inc.

Dickey, M. D. (2007), *Game design and learning: a conjectural analysis of how massively multiple online role-playing games (MMORPGs) foster intrinsic motivation*, Educational Technology Research and Development **55**(3), pp. 253–273. [Online] Available at: https://doi.org/10.1007/s11423-006-9004-7 (Accessed: 04 September 2020).

Dijkstra, E. W. (1959), *A note on two problems in connexion with graphs*, Numerische mathematik **1**(1), pp. 269–271.

Dormans, J. (2010), *Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games*, *in* 'Proceedings of the 2010 Workshop on Procedural Content Generation in Games', PCGames '10, ACM, New York, NY, USA, pp. 1:1–1:8. [Online] Available at: http://doi.acm.org/10.1145/1814256.1814257 (Accessed: 04 September 2020).

Dormans, J. (2011), *Level Design As Model Transformation: A Strategy for Automated Content Generation*, *in* 'Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games', PCGames '11, ACM, New York, NY, USA, pp. 2:1–2:8. [Online] Available at: http://doi.acm.org/10.1145/2000919.2000921 (Accessed: 04 September 2020).

Electronic Arts (2011), *Origin*, [Online] PC, USA: Electronic Arts Inc.

Epic Games (1998), *Unreal Engine*, [Online] PC, USA: Epic Games Inc.

Google (2008), *Google Play*, [Online] Android, USA: Google Inc.

Gustavson, S. (2005), *Simplex noise demystified*. [Online] Available at: https://www.researchgate.net/publication/216813608_Simplex_noise_demystified (Accessed: 04 September 2020).

Hart, P., Nilsson, N. and Raphael, B. (1968), *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*, IEEE Transactions on Systems Science and Cybernetics **4**(2), pp. 100–107. [Online] Available at: https://doi.org/10.1109/tssc.1968.300136 (Accessed: 04 September 2020).

Hendrikx, M., Meijer, S., Van Der Velden, J. and Iosup, A. (2013), *Procedural Content Generation for Games: A Survey*, ACM Trans. Multimedia Comput. Commun. Appl. **9**(1). [Online] Available at: https://doi.org/10.1145/2422956.2422957 (Accessed: 04 September 2020).

Juan Linietsky, Ariel Manzur (2014), *Godot Engine*, [Online] PC, Latin America.

Karavolos, D., Bouwer, A. and Bidarra, R. (2015), *Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation*, *in* 'FDG'. [Online] Available at: http://www.fdg2015.org/papers/fdg2015_paper_25.pdf (Accessed: 04 September 2020).

Karavolos, D., Liapis, A. and Yannakakis, G. N. (2016), *Evolving missions for Dwarf quest dungeons*, *in* '2016 IEEE Conference on Computational Intelligence and Games (CIG)', pp. 1–2. [Online] Available at: https://doi.org/10.1109/CIG.2016.7860391 (Accessed: 04 September 2020).

Liapis, A. (2017), *Multi-segment Evolution of Dungeon Game Levels*, *in* 'Proceedings of the Genetic and Evolutionary Computation Conference', GECCO '17, ACM, New York, NY, USA, pp. 203–210. [Online] Available at: http://doi.acm.org/10.1145/3071178.3071180 (Accessed: 04 September 2020).

Liapis, A., Yannakakis, G. N., Nelson, M. J., Preuss, M. and Bidarra, R. (2019), *Orchestrating Game Generation*, IEEE Transactions on Games **11**(1), pp. 48–68. [Online] Available at: https://doi.org/10.1109/TG.2018.2870876 (Accessed: 04 September 2020).

Microsoft Game Studios (2002), *Age of Mythology*, [CD-ROM] PC. USA: Microsoft Inc.

Mojang (2009), *Minecraft*, [Online] PC. Sweden: Mojang Studios.

Namco (1980), *Pacman*, Pacman Machines, Japan: Namco Ltd.

Perlin, K. (1985), *An Image Synthesizer*, SIGGRAPH Comput. Graph. **19**(3), p. 287–296. [Online] Available at: https://doi.org/10.1145/325165.325247 (Accessed: 04 September 2020).

Unity Technologies (2005), *Unity Engine*, [Online] PC, USA: Unity Technologies Inc.

Valve (2003), *Steam*, [Online] PC. USA: Valve Inc.

Wild Card Games (2013), *Dwarf Quest*, [Online] PC, Netherlands: Wild Card Games Inc.

APPENDICES

Appendix A-Map Generation: Map size over time data

The data we tested our map generation algorithm by increasing the map size gradually and keeping elapsed time for each generation is presented at Table A.1.

Table A.1. Map size over time data

Sample No.	Map Size in Area	Elapsed Time
1	100	0.075
2	200	0.056
3	300	0.067
4	400	0.07
5	500	0.071
6	600	0.086
7	700	0.102
8	800	0.106
9	900	0.137
10	1000	0.132
11	1100	0.151
12	1200	0.149
13	1300	0.163
14	1400	0.177
15	1500	0.209
16	1600	0.208
17	1700	0.211
18	1800	0.253
19	1900	0.3
20	2000	0.307
21	2100	0.287
22	2200	0.339
23	2300	0.378
24	2400	0.379
25	2500	0.376

Table A.1 continued from previous page

Sample No.	Map Size in Area	Elapsed Time
26	2600	0.432
27	2700	0.487
28	2800	0.492
29	2900	0.48
30	3000	0.538
31	3100	0.608
32	3200	0.609
33	3300	0.603
34	3400	0.683
35	3500	0.75
36	3600	0.755
37	3700	0.75
38	3800	0.836
39	3900	0.914
40	4000	0.91
41	4100	0.952
42	4200	0.99
43	4300	1.082
44	4400	1.096
45	4500	1.093
46	4600	1.187
47	4700	1.284
48	4800	1.291
49	4900	1.293
50	5000	1.389

Table A.1 continued from previous page

Sample No.	Map Size in Area	Elapsed Time
51	5100	1.503
52	5200	1.493
53	5300	1.516
54	5400	1.616
55	5500	1.72
56	5600	1.73
57	5700	1.727
58	5800	1.847
59	5900	1.98
60	6000	1.981
61	6100	1.993
62	6200	2.104
63	6300	2.283
64	6400	2.271
65	6500	2.373
66	6600	2.565
67	6700	2.802
68	6800	2.872
69	6900	2.832
70	7000	3.05
71	7100	3.218
72	7200	3.224
73	7300	3.234
74	7400	3.406
75	7500	3.519

Table A.1 continued from previous page

Sample No.	Map Size in Area	Elapsed Time
76	7600	3.429
77	7700	3.479
78	7800	3.5
79	7900	3.898
80	8000	3.881
81	8100	3.892
82	8200	4.218
83	8300	4.305
84	8400	4.366
85	8500	4.933
86	8600	4.494
87	8700	4.882
88	8800	4.929
89	8900	5.017
90	9000	5.245
91	9100	5.058
92	9200	5.302
93	9300	5.546
94	9400	5.475
95	9500	5.758
96	9600	6.096
97	9700	5.924
98	9800	6.158
99	9900	6.537
100	10000	6.462

APPENDICES

Appendix A-Quest Generation: Quest count over time data

The data we used to test the quest generation algorithm by gradually increasing quest count each generation with fixed region size while keeping time elapsed is provided at Table B.1.

Table B.1. Quest count over time data

Sample No.	Quest Count	Time
1	**10**	**0.016**
2	20	0.025
3	30	0.051
4	40	0.057
5	50	0.055
6	60	0.074
7	70	0.099
8	80	0.133
9	90	0.122
10	100	0.149
11	110	0.15
12	120	0.183
13	130	0.205
14	140	0.201
15	150	0.225
16	160	0.243
17	170	0.259
18	180	0.256
19	190	0.291
20	200	0.308
21	210	0.3
22	220	0.359
23	230	0.351
24	240	0.371
25	250	0.379

Table B.1 continued from previous page

Sample No.	Quest Count	Time
26	260	0.397
27	270	0.387
28	280	0.402
29	290	0.456
30	300	0.418
31	310	0.457
32	320	0.501
33	330	0.459
34	340	0.518
35	350	0.532
36	360	0.476
37	370	0.57
38	380	0.553
39	390	0.559
40	400	0.598
41	410	0.646
42	420	0.603
43	430	0.616
44	440	0.643
45	450	0.65
46	460	0.663
47	470	0.646
48	480	0.654
49	490	0.661
50	500	0.687

Table B.1 continued from previous page

Sample No.	Quest Count	Time
51	510	0.693
52	520	0.759
53	530	0.732
54	540	0.755
55	550	0.756
56	560	0.792
57	570	0.833
58	580	0.831
59	590	0.905
60	600	0.886
61	610	0.934
62	620	0.852
63	630	0.864
64	640	0.937
65	650	0.926
66	660	0.968
67	670	0.93
68	680	0.913
69	690	0.967
70	700	0.944
71	710	0.985
72	720	0.997
73	730	1.03
74	740	1.081
75	750	1.085

Table B.1 continued from previous page

Sample No.	Quest Count	Time
76	760	1.104
77	770	1.164
78	780	1.055
79	790	1.091
80	800	1.169
81	810	1.115
82	820	1.105
83	830	1.135
84	840	1.131
85	850	1.165
86	860	1.204
87	870	1.211
88	880	1.241
89	890	1.265
90	900	1.246
91	910	1.252
92	920	1.252
93	930	1.247
94	940	1.327
95	950	1.316
96	960	1.356
97	970	1.405
98	980	1.383
99	990	1.32
100	1000	1.418

APPENDICES

Appendix A-Quest Generation: Region size over time data

The data we used to test for increasing the regions size and generating a fixed size of quests for each generation can be found in Table C.1.

Table C.1. Region size over time data

Sample No.	Region Size	Time
1	10	0.065
2	20	0.083
3	30	0.07
4	40	0.083
5	50	0.113
6	60	0.155
7	70	0.16
8	80	0.216
9	90	0.239
10	100	0.312
11	110	0.374
12	120	0.373
13	130	0.472
14	140	0.492
15	150	0.586
16	160	0.668
17	170	0.671
18	180	0.804
19	190	0.847
20	200	0.94
21	210	1.038
22	220	1.136
23	230	1.237
24	240	1.295
25	250	1.444

Table C.1 continued from previous page

Sample No.	Quest Count	Time
26	260	1.554
27	270	1.604
28	280	1.673
29	290	1.884
30	300	2.013
31	310	2.075
32	320	2.281
33	330	2.336
34	340	2.485
35	350	2.562
36	360	2.761
37	370	2.906
38	380	2.942
39	390	3.254
40	400	3.328
41	410	3.643
42	420	3.619
43	430	3.945
44	440	4.079
45	450	4.196
46	460	4.395
47	470	4.542
48	480	4.699
49	490	4.862
50	500	5.196

Table C.1 continued from previous page

Sample No.	Quest Count	Time
51	510	5.368
52	520	5.545
53	530	5.611
54	540	5.834
55	550	6.036
56	560	6.271
57	570	6.422
58	580	6.775
59	590	6.773
60	600	7.078
61	610	7.368
62	620	7.373
63	630	7.57
64	640	8
65	650	8.277
66	660	8.286
67	670	8.666
68	680	8.945
69	690	9.026
70	700	9.46
71	710	9.668
72	720	9.792
73	730	9.802
74	740	10.025
75	750	10.526

Table C.1 continued from previous page

Sample No.	Quest Count	Time
76	760	10.617
77	770	10.929
78	780	11.915
79	790	11.38
80	800	11.543
81	810	11.938
82	820	12.185
83	830	12.204
84	840	12.807
85	850	12.694
86	860	12.923
87	870	13.328
88	880	13.533
89	890	13.768
90	900	13.993
91	910	14.285
92	920	14.522
93	930	14.895
94	940	15.048
95	950	15.616
96	960	15.777
97	970	16.18
98	980	16.296
99	990	16.258
100	1000	16.656

APPENDICES

Appendix A-Map and Quest Generation: Map, Region, Quest size each generation data

The data we used to test our algorithm by increasing map, region, quest sizes each generation and their memory usage and elapsed time data can be found in Table D.1.

Table D.1. Map an quest generation overall generation data.

Sample No.	**Area**(in $unit^2$)	**RegionSize**	**QuestCount**	**Memory Use (MB's)**	**Time**
1	800	11	11	56.67369	0.364
2	1200	12	12	80.83285	0.367
3	1600	13	13	104.7043	0.363
4	2000	14	14	128.2798	0.365
5	2400	15	15	157.0803	0.44
6	2800	16	16	193.8265	0.494
7	3200	17	17	230.6252	0.508
8	3600	18	18	267.1478	0.508
9	4000	19	19	311.7321	0.588
10	4400	20	20	368.6057	0.728
11	4800	21	21	425.6895	0.674
12	5200	22	22	482.137	0.681
13	5600	23	23	550.662	0.796
14	6000	24	24	629.9821	0.92
15	6400	25	25	707.91	0.916
16	6800	26	26	785.4722	0.938
17	7200	27	27	883.6686	1.083
18	7600	28	28	1000.473	1.24
19	8000	29	29	1116.783	1.234
20	8400	30	30	1233.23	1.272

Table D.1 continued from previous page

Sample No.	**Area**(in $unit^2$)	**RegionSize**	**QuestCount**	**Memory Use (MB's)**	**Time**
21	8800	31	31	1363.108	1.507
22	9200	32	32	1506.793	1.602
23	9600	33	33	1650.464	1.598
24	10000	34	34	1794.388	1.599
25	10400	35	35	1973.491	1.817
26	10800	36	36	2181.823	2.044
27	11200	37	37	2390.478	2.035
28	11600	38	38	2599.102	2.039
29	12000	39	39	2827.632	2.271
30	12400	40	40	3072.375	2.537
31	12800	41	41	3318.713	2.539
32	13200	42	42	3563.673	2.543
33	13600	43	43	3828.924	2.829
34	14000	44	44	4127.971	3.147
35	14400	45	45	4426.854	3.154
36	14800	46	46	4725.58	3.169
37	15200	47	47	5072.804	3.421
38	15600	48	48	5443.165	3.73
39	16000	49	49	5814.575	3.794