UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CIRO GONÇALVES PLÁ DA SILVA

Analysis and Development of a Game of the Roguelike Genre

Final Report presented in partial fulfillment of the requirements for the degree of Bachelor of Computer Science.

Advisor: Prof. Dr. Raul Fernando Weber

Porto Alegre
2015

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Raul Fernando Weber for his support and crucial pieces of advisement.

Also, I'm grateful for the words of encouragement by my friends, usually in the form of jokes about my seemingly endless graduation process.

I'm also thankful for my brother Michel and my sister Ana, which, despite not being physically present all the time, were sources of inspiration and examples of hard work.

Finally, and most importantly, I would like to thank my parents, Isabel and Roberto, for their unending support. Without their constant encouragement and guidance, this work wouldn't have been remotely possible.

ABSTRACT

Games are primarily a source of entertainment, but also a substrate for developing, testing and proving theories. When video games started to popularize, more ambitious projects demanded and pushed forward the development of sophisticated algorithmic techniques to handle real-time graphics, persistent large-scale virtual worlds and intelligent non-player characters.

Our goal in this work is to develop a game of the roguelike genre, and to analyze and compare the algorithmic techniques used to address the challenges of such process. More specifically, we are going to focus on two design features, the first one being the procedural generation of dungeons, and the second being the artificial intelligence for enemies.

Some of the results we have shown include, concerning generating procedural dungeons, that the techniques of basic iteration and BSP trees can be applied to any practical dungeon size, while cellular automata and maze generation through depth-first search must be optimized or restricted in some ways to increase scalability. Also, when it comes to the artificial intelligence for enemies, we concluded that both stateless and state-based techniques can be used with any practical number of concurrent actors without noticeable delay. However, we argued that state-machine actors can be designed to present more complex, flexible intelligent behavior. Regarding path-finding, we have shown that breadth-first search is satisfactory in terms of effectiveness, while in terms of efficiency it performs poorly if no substantial reductions of the search space are employed.

Keywords: Procedural Dungeon Generation. Artificial Intelligence. Game Development. Roguelike.

Análise e Desenvolvimento de um Jogo do Gênero Roguelike

RESUMO

Jogos são primeiramente uma fonte de entretenimento, mas também um substrato para o desenvolvimento, teste e prova de teorias. Quando os jogos eletrônicos começaram a se popularizar, projetos mais ambiciosos exigiram e empulsionaram o desenvolvimento de sofisticadas técnicas algorítmicas para lidar com gráficos em tempo real, mundos virtuais persistentes de larga escala e personagens não-jogadores inteligentes.

Nosso objetivo neste trabalho é desenvolver um jogo do gênero roguelike, e analisar e comparar as técnicas algorítmicas usadas para lidar com os desafios desse processo. Mais especificamente, focaremos em duas características de design, sendo a primeira a geração procedural de geração de masmorras, e a segunda a inteligência artificial de inimigos.

Alguns dos resultados que mostramos incluem, em relação à geração procedural de masmorras, que as técnicas de iteração básica e árvores de partição de espaço binário podem ser aplicadas para qualquer tamanho prático de masmorra, enquanto que autômatos celulares e a geração de labirintos utilizando busca em profundidade precisam ser otimizadas ou restringidas de alguma forma para aumentar a escalabilidade. Além disso, no que concerne a inteligência artificial de inimigos, concluímos que tanto as técnicas de atores que não guardam estados quanto as que possuem máquinas de estados podem ser utilizadas para qualquer número prático de atores concorrentes sem atraso notável. Porém, argumentamos que atores com máquinas de estados podem ser projetados para apresentar comportamento inteligente mais complexo e flexível. Em relação à busca de caminhos, mostramos que a busca por amplitude é satisfatória em termos de efetividade, porém em termos de eficiência ela tem baixo desempenho se não há reduções substanciais no espaço de buscas.

Palavras-Chave: Geração Procedural de Masmorras. Inteligência Artificial. Desenvolvimento de Jogos. Roguelike.

LIST OF FIGURES

LIST OF TABLES

LISTA DE ABBREVIATIONS AND ACRONYMS

NPC	Non-Player Character
CRPG	Computer Role-playing Video Game
DCSS	Dungeon Crawl Stone Soup
ADOM	Ancient Domains of Mystery
FOV	Field of View

CONTENTS

1 INTRODUCTION

Games have been used throughout the history of mathematics and computer science to develop, test and prove theories. Many are the emblematic examples of how mathematicians and computer scientists developed algorithms to beat humans in games, one of them being the Deep Blue computer program, which won a chess match against grandmaster Garry Kasparov (IBM, 2015). When video games started to popularize, more ambitious projects demanded and pushed forward the development of more sophisticated algorithmic techniques to handle real-time graphics, persistent large-scale game worlds and intelligent non-player characters (NPCs).

Our study revolves around this substrate, but before going more into details, it is important to characterize the Roguelike genre, as well as to acknowledge the historical evolution of it, so that we can have a clear vision on where our goals stand.

1.1 Definition of Roguelike

One simple definition of the term Roguelike is to say that it is characterized by games that were inspired directly or indirectly by the game *Rogue* (PETER, 2010). More specifically, it is a sub-genre of role-playing games (RPGs) that share certain features with the genre's archetype, such as the permanent death of the player character (or perma-death), random (or procedurally generated) dungeons and turn-based movement (ROGUEBASIN, 2013).

While that is a generally accepted loose definition (BISKUP, 2000) (LAIT, 2009) (ROGUETEMPLE, 2015), various attempts were made to define the genre more rigidly. One of the most well known is called the “Berlin Interpretation”, which was created at the Roguelike Development Conference 2008 (ROGUEBASIN, 2008). Several game features were discussed by the attendants, and were weighed as “high value” or “low value” factors. Among the high value factors, we mention:

- Random environment generation: the world is procedurally generated, and most likely the player will never see the same dungeon level twice;
- Perma-death: when the player character dies, it can no longer be played with;
- Turn-based: The game does not run in real-time, so it only changes whenever the user acts in some way;

- Grid-based: The world is represented by a uniform grid of tiles;
- Complexity: The game is complex enough so that there are several solutions to a common goal. This is achieved by having several interactions between items and monsters;
- Resource management: The player has limited resources, and must come up with strategies to manage them in the best way possible to advance in the game;
- Hack'n'Slash: Fighting a large number of monsters is an important part of the game;
- Exploration: The player has to explore new dungeon levels for every new game, and must make it so in a careful, planned way.

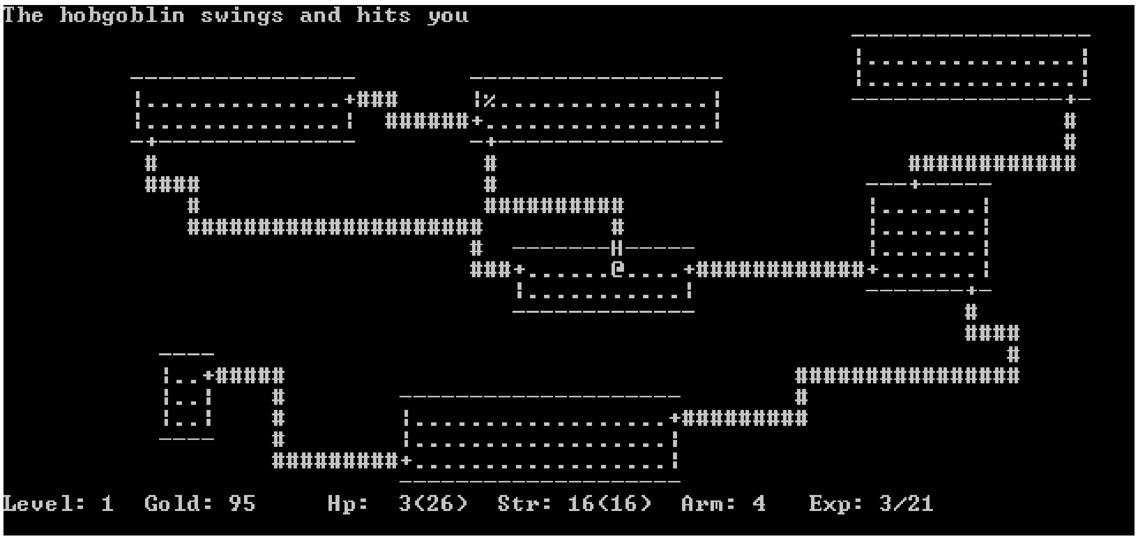
Although we can intuitively look at a game and see if it meets some or most of the above criteria, we believe it is difficult to establish a concrete definition for the genre. Ultimately, instead of saying which games *belong* to the genre, we define how "rogue-like" a game is. This means that the more features of the genre it has, the more it is considered a Roguelike. One way to better exemplify such features is to show which games have been related to the genre historically, and that is what we show below.

1.2 Historical Context

Being a sub-genre of role-playing video games, our first step towards understanding Roguelikes is to start with the beginnings of the former. Computer role-playing video games (CRPGs) are games in which the player controls one character (or a party), immersed in a well-defined world, through a computer (ROLLINGS; ADAMS, 2003). They in turn were inspired by the so-called tabletop (or "pen and paper") role-playing games. In those, players create characters and assume their roles, guided by a set of rules and the narration of a specially designated player called the game master (COVER, 2010). Originally played face-to-face, they inspired the first CRPGs to be developed for the mainframes in the mid 1970's, being *dnd* one of the first examples of games inspired by the tabletop RPGs like Gary Gygax's D&D. Other examples of games that helped define the CRPG genre include *Dungeon* and *pedit5* and later the *Ultima* series (BARTON, 2007).

From that substrate came the canon for the genre, which is Rogue. Inspired by the aforementioned dungeon crawlers, it presented a distinctive set of features that led other developers to create a number of variants for it, which made it the archetype for the "rogue-likes". In every game, the player started from scratch in an unknown dungeon, and had to fight hordes of monsters of increasing difficulty. For that, he had to pick up equipments to aid him in his quest, as well as level up his character to become more powerful. If it was killed, the only thing that was preserved was a high-score entry showing his progress up until that moment (WICHMAN, 1997). Figure 1.1 shows a typical Rogue session. The "@" represents the player, and the "H" represents an attacking hobgoblin.

Figure 1.1 – Typical Rogue session.

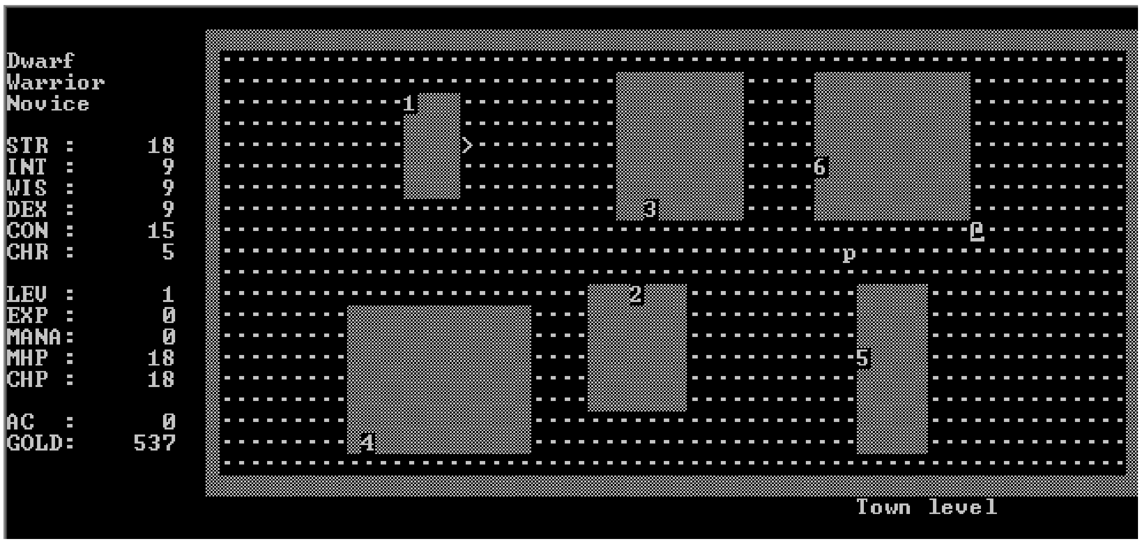


```
The hobgoblin swings and hits you
                                                            -----------------
                                                            ¦...............¦
         ----------------        ------------------         ¦...............¦
         ¦..............+###     ¦%...............¦         ---------------+-
         ¦..............¦  ######+................¦                        #
         -+--------------        -+----------------                        #
          #                       #                             ############
          ####                    #                          ---+-----
             #                    ##########                 ¦.......¦
             #####################         #                 ¦.......¦
                                 #  -------H-----            ¦.......¦
                                 ###+......@....+############+.......¦
                                    ¦...........¦            -------+-
                                    -------------                   #
                                                                    ####
           ----                                                        #
           ¦..+#####                                    ################
           ¦..¦    #        --------------------        #
           ¦..¦    #        ¦..................+#########
           ----    #        ¦..................¦
                   #########+..................¦
                            --------------------
Level: 1  Gold: 95     Hp:  3(26)  Str: 16(16)  Arm: 4   Exp: 3/21
```

Source: Screen capture of the game by the author.

Rogue was considered a very addictive game in some universities in the 1980's, especially because the combination of permanent death and randomly generated levels resulted in a high level of replayability. Two important descendants of Rogue – *Hack* and *Moria* – were an attempt of their creators to extend Rogue in some way. Hack 1.0, the first widespread version of the game featured the same number of dungeon levels as Rogue and had the same objective, though the number of monsters was doubled, as well as addition of a pet companion, persistent levels and the need for food management (BROUWER, 2003). Moria on the other hand kept Rogue's non-persistence of dungeon levels, but added an overworld, that is, a town where the player could trade and store items (GRABINER, 2015). Figure 1.2 shows a screenshot of a Moria session where the player is in the town. The presence of a town was one of the main differences from Rogue.

Figure 1.2 – Moria overworld.



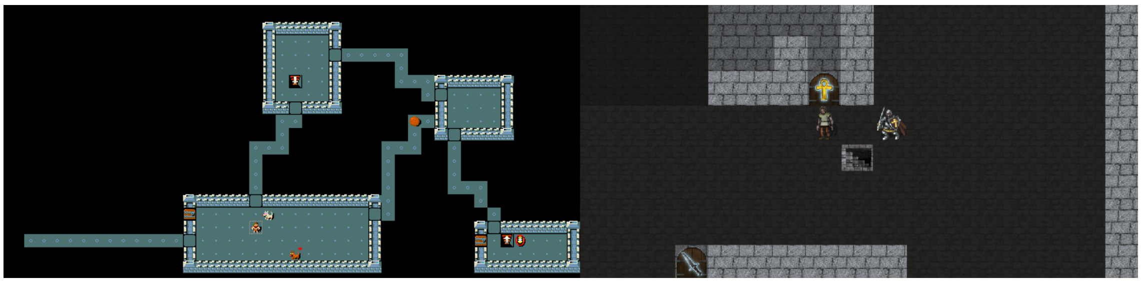
Source: Screen capture of the game by the author.

Moria and Hack in turn inspired their own, more ambitious, descendants. *Angband* (ANGBAND, 2015), the most important variant of Moria, had 100 dungeon levels, some of them with unique features called vaults, where the player would face harder enemies but would be rewarded with treasures. It also added a number of unique items, called artifacts, which would be invaluable to defeat the large number of unique monsters also added to the game.

NetHack (NETHACK, 2015), Hack's successor, unified various independent development branches of his predecessor as a means to aggregate the additions they made. The game was ported to other operating systems (as opposed to only BSD) and featured more player classes, monsters and dungeon levels, as well as more unique items and monsters.

Recently, both NetHack and Angband started to support the use of tile-based graphics, as well as the traditional ASCII style characters. Figure 1.3 shows screenshots of the tiled versions of both games (NetHack to the left, Angband to the right).

Figure 1.3 – Tiled-graphic versions of NetHack and Angband.

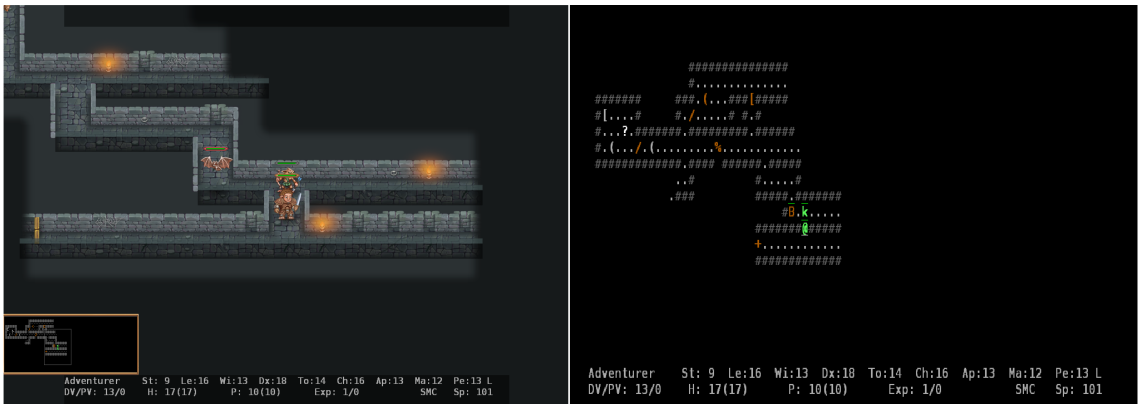


Source: Screen captures of the games by the author.

In the mid 90s, Many roguelikes were developed inspired by NetHack and Angband. Of those, we cite *Crawl*, first released in 1995 by Linley Henzell, it followed the same principle of finite resources NetHack had, which meant that the player could not stay at a same level while gaining experience. Its plot was also simple: to dive down a single dungeon and retrieve an artifact at the bottom, and subsequently rise up to the surface again. Currently, Crawl's most active variant is *Dungeon Crawl Stone Soup* (DCSS), and it is developed openly by the community (DCSS, 2015).

Ancient Domains of Mystery (ADOM), first released in 1994, on the other hand had a mix between persistent and non-persistent dungeon levels, which left some room for the so-called "grinding", that is, staying in the same place while defeating monsters and gaining experience. Also, it featured a much more detailed storyline, a number of side quests and an overworld to connect them. In 2012, ADOM underwent a crowdfunded campaign called "ADOM Resurrection". As a result, there were several additions to the game, one of which is the integration with NotEye, which made it possible to provide tile support to the game (BISKUP, 2012). Figure 1.4 shows a screenshot of ADOM both in ASCII (right) and tile-based graphics.

Figure 1.4 – ADOM in graphical tiles and ASCII.


Adventurer St: 9 Le:16 Wi:13 Dx:18 To:14 Ch:16 Ap:13 Ma:12 Pe:13 L
DV/PV: 13/0 H: 17(17) P: 10(10) Exp: 1/0 SMC Sp: 101
Adventurer St: 9 Le:16 Wi:13 Dx:18 To:14 Ch:16 Ap:13 Ma:12 Pe:13 L
DV/PV: 13/0 H: 17(17) P: 10(10) Exp: 1/0 SMC Sp: 101

Source: Screen captures of the game by the author.

We would also like to mention Tales of Middle-Earth (ToME), arguably the most successful Angband variant (DOULL, 2013). It was developed with the intent of merging several ideas from other Angband variants. Many were the additions, including an expansive overworld (comprising most of Middle-Earth), several branching quests and reworked races and classes. Also, the creators wanted to further approximate the game to J.R.R Tolkien's mythos. In 2012, the ToME 4 was released, and it was renamed to Tales of Maj'Eyal, to distance itself from Tolkien's mythology and instead creating it's own original fantasy setting.

Finally, we would like to mention one of the most remarkable examples of deeply complex roguelike experience, which is that of Dwarf Fortress. In it, you can choose to play an adventurer and explore the massive, randomly-generated world, or you can simulate the building and management of a dwarven civilization. To summarize the complexity of this game, entire worlds are randomly generated, including civilizations, wars, towns, down to single individuals. Regarding battle mechanics, it simulates individual body parts (including internal organs) and the several types of damage they can suffer from, like cut, burn, rot and freeze, among others (ADAMS, 2015).

We observe, from the context described above, that the genre's development intertwines with that of modern computers in general, starting from mainframes running simple terminal-based dungeon crawlers, passing through games still simple in graphics, though with great depth of content, and recently even fully-fledged commercial games, with sophisticated graphics and well-developed storylines.

1.3 Objectives

It is based on what was stated before, while keeping in mind the context of the area, that this work has been elaborated. Our goal is to develop the prototype of a game of the roguelike genre, and to analyze and compare the algorithmic techniques used to address the challenges of such process. Of those challenges, we are going to focus on two, the first one being procedural generation of dungeons, and the second being the artificial intelligence for the enemies.

More specifically, we will:

- Present the main challenges of developing a game of the Roguelike genre, both from a computational and a design points of view;
- For two of them, explore and compare different approaches to solve them;
- Implement a prototype of a Roguelike as a means to test and validate the approaches explored;
- Analyze the results of the implementation, from a performance point of view, as well as in terms of playability;
- Conclude with the best configuration of features, and point out the direction of possible future work in the game.

1.4 Related Work

While searching for related work, we focused in two aspects. First we searched for publications related to the Roguelike genre in general. Then, we moved on to work related to the two design features explored, that is procedural dungeon generation and monster's artificial intelligence.

1.4.1 Work Related to the Roguelike Genre

Based on our research, few works aim specifically for the genre. Most of the related work focus on “dungeon crawlers” in general, so we ought to name the few we have found.

(ALMGREN et al, 2012) is the one that most closely relates to ours. It features the development of a space-themed roguelike, and, much like our work, analyzes design features, focusing on procedural content generation. It also talks about the artificial intelligence, albeit with less emphasis that we do.

(IBÁÑEZ, 2014), similarly to the previously pointed work, deals with the development of a roguelike, and the various design choices that come with it. It also focuses on procedural generation, by generating both random dungeon maps and random items. It seemed to us that both theses focus on the game as the main end-product of their work, whereas we use it as a means to analyze, in theory and practice, the chosen design features in depth.

Another work in the genre, but from a different perspective, was found in (AMARI, 2009). In it, the authors simulate the world of a roguelike game by using artificial chemistry simulations. Then, it removes a part of the simulation to transform it into a playable game. They show that a simple roguelike can come out of a set of chemical rules embodied into an abstract chemical model.

Finally, we mention (GARDA, 2013), not as a work aimed at algorithmic techniques or actual implementation of games, but instead as a source of information for the definition of the genre itself. It also aims to characterize the term the author calls “neo-rogue”. The term would mean, in simple terms, the recent wave of games inspired by Rogue and roguelikes, which branched from the more strict design features of the genre and adopted other ones from different genres. These are also dubbed by the community “roguelike-like”, “roguelite” (this one, coming from 'rogue' and 'light', was popularized specially because of the more lenient approaches to perma-death), “roguelike-renaissance”, among others.

1.4.2 Work Related to the Design Features

Several works, including two of the ones already mentioned in the previous section, focus on procedural content generation for CRPGs, especially for dungeon level (or map) generation. We chose some of the more closely related to our chosen genre, especially in the scope of dungeon crawlers.

The first we mention is (VALTCHANOV, 2012), which explores the generation of dungeon levels by using genetic programming. The author generates random maps through mutations, crossovers and the combination of both, and then evaluates them through a fitness function, which, in the author's words, “guides the search toward finding maps that are composed of small, tightly packed clusters of rooms that are connected to efficient paths of hallway.” This work is our main source for dungeon generation using genetic algorithms, which we will talk about in chapter 3.

The next one, (DORMANS, 2012) aims at a broader spectrum of games, which is action adventure games. In his work, he makes a separation between missions and spaces: the former being a logical flow of the game and the latter serving as a medium for the events to happen. That way, by using a generative graph grammar, he creates missions in the form of graphs. He then uses that specification to generate spaces, as well as using shape grammars for generating spaces. We chose not to explore generative grammars in this work for space reasons, so his work can be considered a complementary read to ours.

Finally, in the context of procedural generation, we talk about (CRUZ, 2014). In this monograph, the author tries to evaluate the quality of procedurally-generated maps by using a number of metrics. Such metrics include the number of steps walked, items collected, money collected, among others. We studied his methodology so that we could devise our own experiments.

Concerning Artificial Intelligence, aside from the two papers we mentioned in section 1.3 that dealt with the subject, we only found more general works of Artificial Intelligence for agents. Because of that, we will mention them when exploring their theory in Chapter 5.

2 DESIGN FEATURES

When designing a game, and in our case a roguelike, there are several design features which must be addressed. For some of them, simple (or complex but fail-proof) solutions exist, while others present challenges both in terms of computational complexity as well as perceived playability. In the next section we will list some of the aspects involved in developing the game prototype. After that, we will talk more in depth about the two chosen design features for the thesis.

2.1 General Features

The following are some of the main features which must be addressed in possibly any genre of games, but with different levels of emphasis. For example, while CRPGs are heavily based on character development, platform adventures are less focused on it, but focus considerably on user interface. Thus, we will try to define them in terms of our chosen genre.

2.1.1 Game Workflow

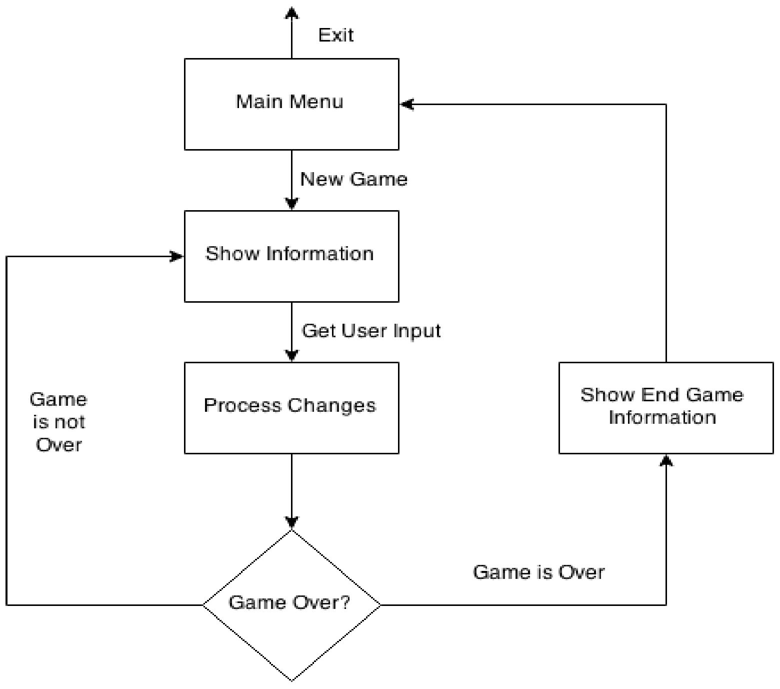
One of the first decisions a designer must make about a game is the workflow, that is, how the different game events (or screens) will be built and how they will relate to each other. For instance, a game usually starts on a main menu, and from there different choices can be made, such as playing a new game, loading a saved game, changing options and exiting the game. Usually, the game workflow is represented by a finite-state machine (FSM), where the events are states and transitions happen based on user input. Diagrams of different levels of abstraction can be made to represent the chain of events that happen in a game. Figure 2.1 shows a simple one of these.

One important aspect that must be thought about is the game pace, that is, how the game will progress. Most games fall in either the turn-based or real-time categories, although some games may present a mix of the two. In turn-based progress, the game will wait for the user-input and change the game state accordingly, while in real-time, the game will progress even when no action is performed by the player. In the case of roguelikes, the majority is turn-based, but there are a few with real-time progression, such as Angband's variant Mangband

(MANGBAND, 2015), Diablo (DIABLO, 2015), which shares most traits of a roguelike, and is thus recognized by some as one, and Pyromancer (ROGUECENTRAL, 2015).

As the complexity of the game increases, so will the game workflow become more complicated. That means careful planning of the game features, modes and interactions must be made before the game starts to be developed. The game developed for this work aimed for a simple workflow, so that the chosen features could be tested at length without unnecessary complications. This was facilitated by the fact that both features were modular, that is, their logic mostly did not depend on the rest of the game, so in theory they could be applied (with eventual necessary modifications) to any game that followed roughly the same set of rules. Figure 2.1 shows a visual representation of a simple workflow for a video game.

Figure 2.1 – A simple workflow for a video game.



Source: Created by the author.

2.1.2 Graphics

When it comes to displaying the game information to the user, different levels of graphics can be used, from pure text, to ASCII graphics to full 3D games. Roguelikes though have historically represented, and are generally recognized by their ASCII characters. The reason for that is that most of the roguelike developers choose to focus on game content rather than graphics, as developing a game with sophisticated graphics is resource demanding. Also, it is important to mention that graphics alone do not make the game user-friendly. For that, the way the user inputs commands must also be taken into account when designing the game.

The game developed for this work followed the tradition of ASCII, although we use colors to make it more aesthetically pleasing. Aside from that, different colors can be used to represent distinct creatures with the same letter. Thus, a brown “o” might represent a regular orc, while a dark blue “o” might be an orc chieftain.

2.1.3 User Interface

As stated above, well-drawn graphics do not guarantee a good player experience regarding the game interface. The game designer must also plan how the information will be displayed to the player, and also how the input devices will be applied to the game actions. For instance, most roguelikes focus the majority of commands in keyboard input, with some minor functionalities attributed to the mouse (like displaying information on mouse-hover/click). Because roguelikes tend to prioritize complexity over graphical design, the large number of commands in the games of the genre tend to make the learning of key bindings (which keys do what) an important part of becoming skilled at a particular game. Games such as ToME try to make better use of the mouse so as to provide more intuitive user input.

In our game, we kept to the traditional key bindings most classic roguelikes possess. Despite that, as our game has relatively simple interactions, the user will have no trouble memorizing them. Aside from that, we will explore limited use of the mouse, to perform simple actions such as looking at dungeon entities to find out information about them.

2.1.4 Storyline and Quests

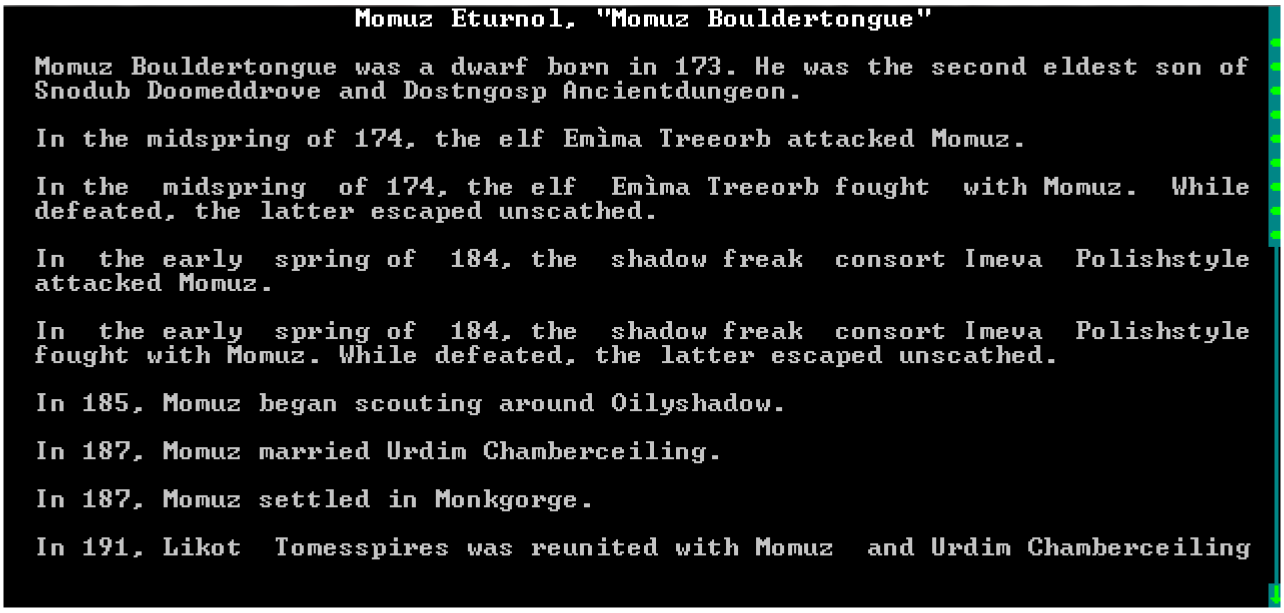
While the storyline and quests are not usually a priority for the genre (especially for the older games), in general there is a main goal for the game. In games such as Rogue, Moria and Nethack, and their descendants Angband, Crawl, ADOM and others, the goal is to go to the deepest level in the dungeon to either defeat the final boss or retrieve an important artifact. Aside from that, most of them also present side-quests. These include defeating intermediate bosses, retrieving other artifacts that facilitate or are even mandatory for the progression of the player, and some even include puzzles to the game (we cite NetHack's Sokoban puzzle and ADOM's labyrinth).

Most of those quests and storylines are devised beforehand by the game designer, although there is research on procedurally generating them (see related work). We also mention games with procedural quests or storylines, like Dwarf Fortress's world generation, which creates an entire world and simulates its history, down to the stories and relationships of individual beings (people and notable monsters). Figure 2.2 shows an example of procedurally generated story for an individual in Dwarf Fortress.

Another recent example of such system is Middle-earth: Shadow of Mordor's Nemesis System (MIDDLEEARTH, 2015), in which the players must gather information regarding orc warchiefs, by finding pieces of intelligence or by killing their subalterns, so that they can find and defeat them, so as to increase the power of their weapons. Much of this system is randomly generated, including the bosses' names, strengths and weaknesses, their hierarchy and the information that leads the player to them. The bosses also do their own quests, and may become stronger and advance in ranks on the hierarchy.

In our work, we decided to keep the game objectives simple, so as to focus on other design features. However, in Chapter 6 we will point out how our game could be expanded in this direction.

Figure 2.2 – A procedurally generated story in Dwarf Fortress.



Momuz Eturnol, "Momuz Bouldertongue"

Momuz Bouldertongue was a dwarf born in 173. He was the second eldest son of Snodub Doomeddrove and Dostngosp Ancientdungeon.

In the midspring of 174, the elf Emìma Treeorb attacked Momuz.

In the midspring of 174, the elf Emìma Treeorb fought with Momuz. While defeated, the latter escaped unscathed.

In the early spring of 184, the shadow freak consort Imeva Polishstyle attacked Momuz.

In the early spring of 184, the shadow freak consort Imeva Polishstyle fought with Momuz. While defeated, the latter escaped unscathed.

In 185, Momuz began scouting around Oilyshadow.

In 187, Momuz married Urdim Chamberceiling.

In 187, Momuz settled in Monkgorge.

In 191, Likot Tomesspires was reunited with Momuz and Urdim Chamberceiling

Source: Screen capture of the game by the author.

2.1.5 Time Progression

The time progression of a game relates to how the player perceives the changes in the game as time passes, based on how it shows the information on the screen and how user input is handled. Games are usually either turn-based or real-time, as stated before. In a turn-based game, information will be shown to the player and the game will not progress as long as the player does not enter input. In real-time, on the other hand, events will continue to happen even though the player does no action.

This means, from a design viewpoint, that turn-based game progression will imply a more sequential workflow, whereas with real-time progression, the user input will be non-blocking, thus the game design will have to be adapted, or else playability problems could arise. For instance, if there are no turns, will the monsters keep attacking the player every clock tick even if he is not active? Also, what happens if a player and a monster try to move at the same time to the same location?

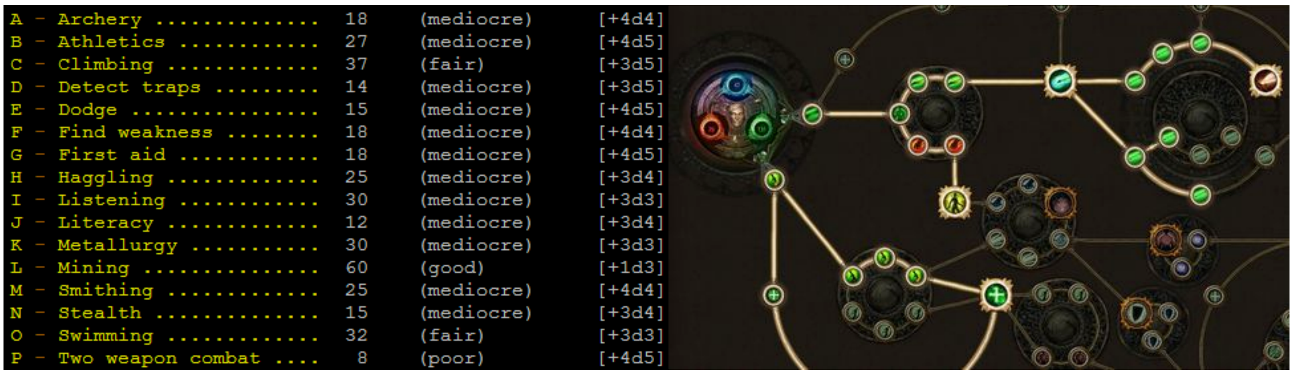
Those problems can be solved by adding a speed attribute to every individual, and making an action cost a certain amount of clock ticks. Also, there is a need to manage the order of actions based on their time duration, thus a dynamic queue is a common solution. Our game uses turn-based time progression, so that no further complications arise with relation to the artificial intelligence.

2.1.6 Character Development

Character development is one of the most (if not the most) important aspect of the CRPG genre (and its sub-genres such as roguelike), as, the name implies, the player is playing a role of a character, so it must evolve in some way. The most common way of adding character development is the experience/level model. In it, the player must kill monsters or complete tasks in order to gain experience, which in turn makes the character advance levels. When he advances a level, he becomes stronger in general, with improvements being for instance the increase of hit points (HP), attributes or abilities.

Additionally, other systems can be added to the game to make it richer in depth. One of them is the skill system, in which the character learn skills as they progress the game. Those skills will aid the player in some way, in areas such as combat, crafting items and interaction with NPCS. In roguelikes, two ways of implementing skill acquirement (and improvement) can be recognized: by spending points gained at level up and by using the skills a certain number of times. The skills themselves may be just a simple list, or they can be organized into prerequisites, which becomes the so-called “skill tree”. Figure 2.3 shows both a simple skill list from ADOM, on the left, and a complex skill tree from Path of Exile (POE, 2015), on the right.

Figure 2.3 – Skills from ADOM and Path of Exile.



```
A - Archery ..............  18    (mediocre)      [+4d4]
B - Athletics ............  27    (mediocre)      [+4d5]
C - Climbing .............  37    (fair)          [+3d5]
D - Detect traps .........  14    (mediocre)      [+3d5]
E - Dodge ................  15    (mediocre)      [+4d5]
F - Find weakness ........  18    (mediocre)      [+4d4]
G - First aid ............  18    (mediocre)      [+4d5]
H - Haggling .............  25    (mediocre)      [+3d4]
I - Listening ............  30    (mediocre)      [+3d3]
J - Literacy .............  12    (mediocre)      [+3d4]
K - Metallurgy ...........  30    (mediocre)      [+3d3]
L - Mining ...............  60    (good)          [+1d3]
M - Smithing .............  25    (mediocre)      [+4d4]
N - Stealth ..............  15    (mediocre)      [+3d4]
O - Swimming .............  32    (fair)          [+3d3]
P - Two weapon combat ....   8    (poor)          [+4d5]
```

Source: (left) Screen capture of ADOM by the author, (right) (IGN, 2015).

Another way of enhancing character development, which usually intertwines with skills, is that of player classes and races. Both classes and races are a way to guide the gameplay to some kind of style (like being a melee fighter or a spell caster), though races usually deal with starting characteristics and restrictions (sometimes including which classes can be played), while classes guide the way the character progresses throughout the game.

In our game, we chose to keep the player simple, without races, skills or classes. However, we added experience levels which give the player attribute improvements, as well as the possibility to collect items to aid him in the adventure.

2.2 Procedural Dungeon Generation

Procedural (or random) dungeon generation, the first of our chosen design features to focus, is part of a larger concept called Procedurally Generated Content (PCG). PCG deals with generating content in general for games in a procedural fashion. This includes dungeon levels, quests and plots, monsters, and even music and graphics, among others. We chose to concentrate on this because:

- It is a recognizable feature of roguelikes;
- A good number of techniques were designed over time for this;
- Some of those techniques may provide good results, albeit not scaling well.

Because of that, we believe testing those techniques, of varying levels of complexity, may lead us to derive useful conclusions. To start with, PCG can be categorized in a number of ways. According to (DOULL, 2008), there are seven loose types of procedural generation, namely:

- Runtime random level generation: Deals with creating dungeon levels by using randomized algorithms while the game is being played, generally when the player changes levels (in the case of roguelikes, typically when he goes down a stair).
- Design of level content: It is used when the above method does not reliably produce satisfactory results. In this case, the maps are generated randomly in a level design tool, as opposed to runtime, and then supervised for correctness by a level designer.
- Dynamic world generation: This technique uses a random seed to iteratively grow the playing field by permuting it with pseudo-random number generator techniques. It usually subdivides from a top-down perspective, thus working well with fractal generation and level of detail techniques.
- Instancing of in-game entities: Consists of randomizing the parameters of in-game entities (like monsters) so that large populations of unique entities can be created with insignificant chance of repetition.
- User mediated content: This technique employs PCG to generate a range of possibilities, which can then be chosen, and subsequently fine-tuned by the user if desired.
- Dynamic systems: Refers to the modeling of systems such as weather and crowd behavior by using PCG techniques, as a means to create (statistically) unrepeatable situations in a game, thus increasing replayability.
- Procedural puzzles and plot generation: In this category, procedural techniques are used to make the stories and challenges of a game more unpredictable. For example, randomness may be added to parts of the game's quest dependency graph, so that consulting walkthrough manuals would be less game-breaking.

Of those, we will focus on run-time random level generation, which deals with creating dungeons while the game is progressing. Also, we will experiment with instancing of in-game entities in Chapter 4. This kind of PCG is related to randomizing the parameters for the generation of content so that unique instances of content (in our case monsters) appears.

2.3 Artificial Intelligence

Artificial intelligence in games refers to handling actors not controlled by the player. When it comes to monsters, this most of the time means planning ways to attack (and defeat) the player. In most roguelikes, the classic way to achieve that is to calculate the shortest path from the monster to the player character, move in its direction, and attack if close enough. This type of behavior can be modeled by stateless actors, which would, at any given time, check for a set of conditions and act based on them, without considering any internal information they may have acquired throughout the gameplay session. Also, the problem of finding the shortest (or most interesting) path is known as path-finding.

However, if the game designer intends to deliver a more interesting experience to the player, several techniques may be employed to enrich the NPCS. We mention some of the categories of techniques which can be used, which we will analyze in Chapter 4:

- Stateless Actors: The simplest form of actors, they consist of a series of if-conditions (generally forming a tree of conditionals) which are tested every turn. Depending on the result of the tests, the actor will react accordingly.
- State-machine Actors: By adding internal states to actors, more sophisticated behavior can be modeled, since they can become less “reflexive” and more “cognitive” entities.
- Swarm Intelligence: Deals with using decentralized, collective behavior to add group intelligence to actors at low computational cost.
- Randomization of Parameters: Previously mentioned as “instancing of in-game entities”, it can be used in the context of AI, in combination with other, parameter-based techniques, to generate a large number of unique actors.
- Genetic Algorithms: They can be used to evolve the decision-making of a certain kind of monster by random mutations and crossovers.
- Emotion-based Actors: It consists of modeling human (or animal) emotions into a set of traits and adding them to actors, thus defining their “personality”, so that their personalities determine (or at least influence) their courses of action.

The above categories can be used not only by themselves, but also in combination with each other to create unique NPC behavior for a game, as well as to generate the levels of challenge, fun and replayability desired by the designer. In chapter 5 we will present the set of AI techniques implemented for our game and their practical results.

3 PROCEDURAL DUNGEON GENERATION

In this chapter, we will focus on the problem of generating dungeon levels procedurally. As stated in the previous chapter, we will specifically focus on run-time dungeon generation, that is, the techniques will be applied while the game is running. If the results take a perceptible time to calculate, a loading screen (which will serve to notify the user processing needs to be done before the game proceeds) might be needed. Otherwise, the game will simply calculate and present the new level to the player once he reaches a changing level point (like stairs or portals).

There are several techniques that can be used for this purpose, and all of them make use of randomness in some way. For that matter, a technique called random number generator (RNG) must be employed. While it is generally implicit, games actually use pseudo-random generating methods (PRNGs), which have several advantages over true random methods for this application. Two of them are:

- Non-blocking generation: While true (or natural) random number sources have a limit on the throughput (because of their limited entropy over time), PRNGs have theoretically unlimited possibilities (they must be built so as to provide enough repetition-free sequences for the specific application).
- Predictability: For developing, testing and debugging purposes, the fact that a certain seed (initial value from which the random numbers are extracted) will always result in the same sequence of numbers is desirable, because true random sources will generate unpredictable outputs, which makes analysis harder.

In the following sections, we will present several random dungeon generation techniques. They vary in computational complexity and perceived quality, the latter being a subjective evaluation (although we mentioned a related work which attempted to measure them in a more principled fashion). Because of that, in section 3.8 we will analyze and compare them in those regards.

3.1 Basic Iteration

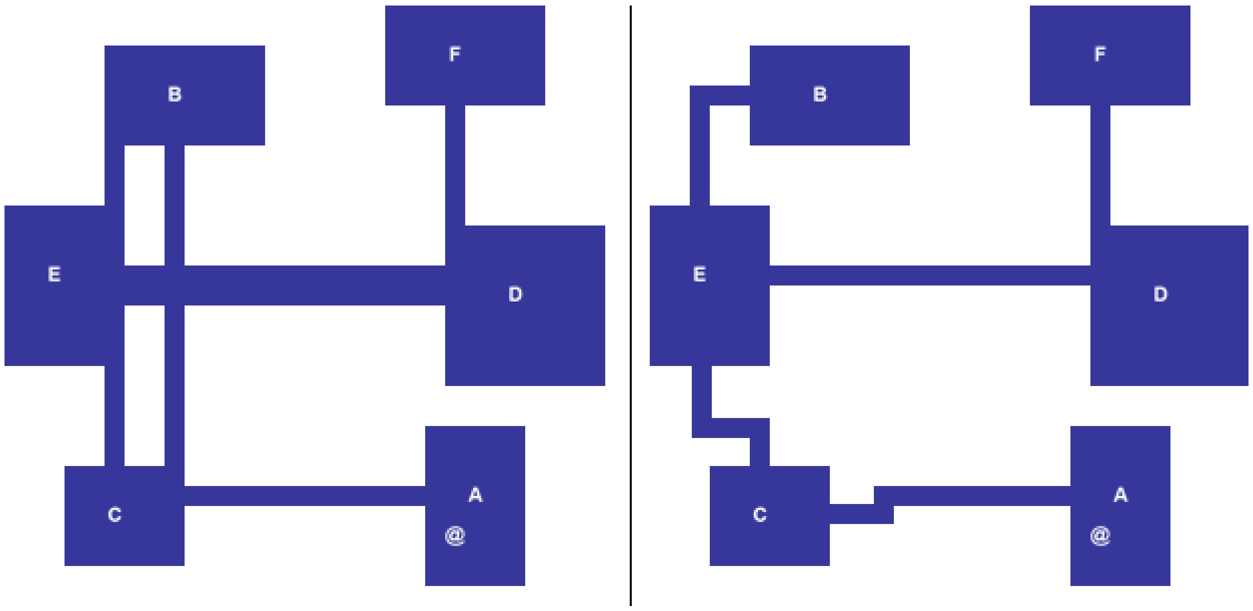
The most basic technique, present in some form in most of the classic roguelikes, consists of creating a number of randomly placed rectangles, which will represent the rooms, and try to connect them by corridors. To make sure the player can reach every room from any starting point, one must check the connectivity of the rooms. A simple way to guarantee that is to always connect a newly created room to the previous one (after the first), thus making sure every room added will keep the rooms fully reachable between each other. One characteristic of this approach is: the connection of corridors will not take into account previously created rooms, so some corridors may cross other rooms and corridors. Also, because new rooms are always connected to the last one created, the level may appear to be a single path of successive rooms. Both may not be a desired effect by the designer.

One way to avoid those problems is to model the dungeon into a graph, where vertices would represent the rooms, and the corridors would be edges. Following that, the following can be done:

- Start from any room, connect it to another room (based on some criteria, like the closest, or even a randomly chosen room) and then do a search by using algorithms such as depth-first search, breadth-first search or Dijkstra's, successively until every room is connected to the others.
- Connect every room to the others by using corridors, and then find the minimum spanning tree for this graph. After that, just prune the unwanted corridors. One interesting approach to this problem is to make the center coordinate of the rooms to be the vertices of the graph, and then apply the Delaunay Triangulation (DELAUNAY, 1934) to it. This way, the resulting edges will represent non-intersecting corridors, which may be desirable.

After we guarantee the dungeon is fully connected in some way, we can add further corridors for an added effect. For instance, a number of corridors, based on the number of rooms, may be added to increase the number of paths of the dungeon level. In addition to that, corridors that lead nowhere may be added, characterizing dead ends. Figure 3.1 shows the difference between dungeon levels generated using corridor connections to previously inserted rooms (left) and using minimum spanning trees (right).

Figure 3.1 – Room connectivity approaches in dungeon generation.



Source: Created by the author.

3.2 Unique Dungeon Features

Unique dungeon features are predefined structures, usually modular, which can be added to levels. While not necessarily random by themselves, those features are generally used in combination with other procedural generation techniques so that, in the end, their placement and varying parameters will make them a useful tool for map variability. Dungeon features can also span an entire level, although in this case the random factor of it will be generally restricted to the monsters and items inside it. Some of the most common unique dungeon features include:

- Vaults: Rooms with pre-designed structures that can contain monsters, items and traps. The general idea behind them is to offer the player a chance for high reward, at the cost of higher danger.
- Temples: Features that are geared towards the player's interaction with divinities. They contain an altar for the interaction to happen, and sometimes NPCs called priests wander inside it, with various divinity-related events.
- Shops: Shops are dungeon features that allow players to trade items. Usually, the trade is mediated by a special NPC called the shopkeeper. Items can be completely random, or the shop can have a theme, for instance a “magic shop” or an “armor shop”.

- Towns: Normally occupying an entire level, towns are generally the place for a number of shops and quest-giver NPCs. While most towns in classic roguelikes have fixed building placement, there are examples where the town itself is randomly generated.
- Fountains: Fountains are mentioned here because they greatly contribute to the randomness of a gameplay session. When drunk from, various outcomes may happen, such as simply satisfying thirst, attribute changing, mutation acquirement and others.
- Vegetation: In some roguelikes, vegetation is implemented first as a way to increase variability, but also as producing a number of interactions. Some of them include chopping trees down for wood, strategic positioning, transformation into monsters ("Ents") and agriculture. Sparse vegetation may be randomly placed in the dungeon, or more intricate arrangements may be used for added effect, such as growth based on cellular automata (as seen in ADOM's herbs).
- Waterways: As vegetation, waterways (rivers, lakes and ponds) can be used to increase level variability, as well as being strategic geographic components. They can be generated using techniques such as the Drunkard's Walk or cellular automata. The Drunkard's Walk is a form of random walk, which consists of a series of random steps in succession through a medium (in the case of waterways, randomly traversing through the dungeon rectangle). It differs from the regular walk in that its termination conditions lead to a biased ending state (VOLCHENKOV; BLANCHARD, 2011).

3.3 Mazes

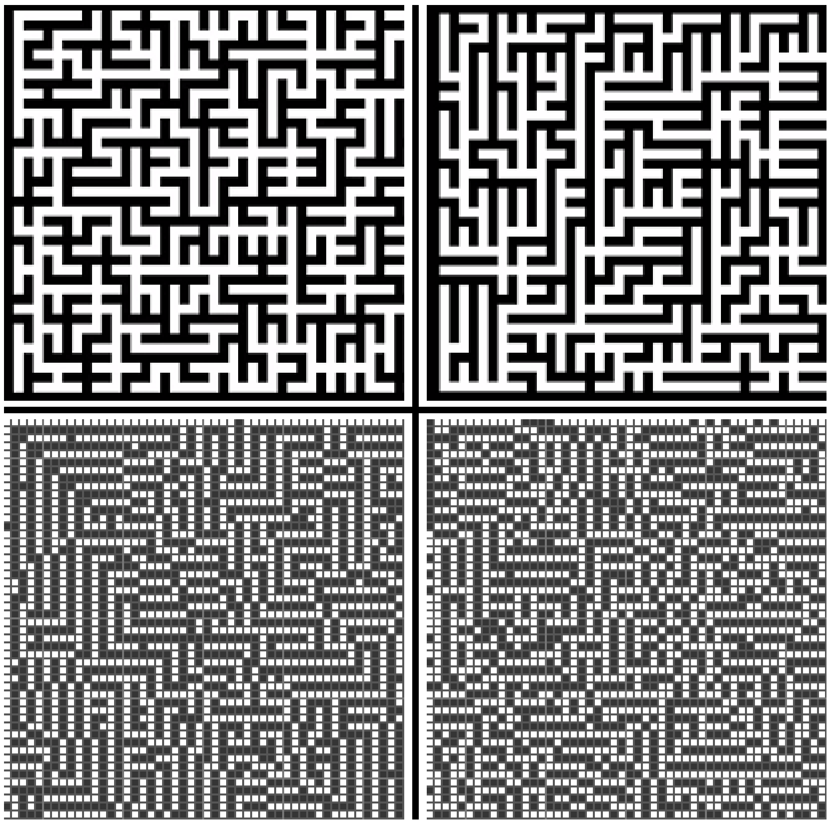
As opposed to the classic "rooms connected by corridors" approach, mazes consist mostly (if not only) of long, winding corridors, which the player must venture in order to advance in the game. Some of the approaches to generating mazes include:

- Depth-first Search: By modeling the search space to be a two-dimensional grid, with the squares as the vertices and the transition to neighbors as the edges, as well as randomizing the choice for neighbor visitation, a maze can be built. For different effects, some choices of neighbor visiting can be favored through weighted randomness. Thus, for example, if horizontal visitation is favored over vertical, the algorithm will produce more long horizontal corridors.

- Kruskal's algorithm (KRUSKAL, 1956): First, we create a list of all walls, and create a set for each grid square, containing initially only itself. After that, for each wall (randomly picked), if the squares divided by this wall belong to different sets, then remove the wall and join the original sets together. Because of Kruskal's original purpose, which is to find a minimum spanning tree on equally weighted edges, the resulting patterns will be usually easy to solve.
- Prim's algorithm (PRIM, 1957): Also being a minimum spanning tree algorithm, regular Prim will yield similar results to those of Kruskal's. However, instead of keeping a list of edges for it, we keep a list of adjacent grid squares. By doing that, and by randomly selecting adjacent square grids for visiting for cells with multiple neighbors, the algorithm will tend to branch out more in comparison to the regular approach.
- Cellular Automata: They can also be used for maze generation. Particularly, two rule sets for Conway's Game of Life (GARDNER, 1970) have been widely used for this purpose, namely Maze and Mazectric (WOJTOWICZ, 2001). The rule strings for them, which are B3/S12345 and B3/S1234, mean that a dead cell will become alive if it has three alive neighbors, and a live cell will continue to live if it has from 1 to 4 (and also 5, in the case of Mazectric) live neighbors, dying otherwise. These, based on a random starting pattern (which can be considered a seed), will result in complex maze-like structures. While the generated patterns are more complex than the previous approaches, it has some drawbacks, the most important one being not guaranteed connectivity between two points. Some kind of workaround must be used to solve that, like randomly placing the up stair and the down stair of the level, and then running a search algorithm to make sure it is connected. Another possibility is to generate a path independent of the automaton's maze, and then overwriting the resulting corridors to it, thus guaranteeing at least one path between the stairs. Another possibility would be to allow the player to dig through walls, thus not needing to worry about connectivity.

Figure 3.2 shows examples of mazes generated by the above techniques. Upper-left shows a maze generated by randomized Kruskal's, upper-right by modified Prim, lower-left by “Maze” cellular automaton rule and lower-right by “Mazectric” cellular automaton rule.

Figure 3.2 – Examples of procedurally generated mazes.



Source: (upper) (WIKIPEDIA, 2015), (lower) created by the author using the Golly simulator.

3.4 Cellular Automata

Aside from the previously mentioned application on mazes (for the rules Maze and Mazectric), cellular automata can be used to generate natural-looking, cave-like dungeon levels. To do this, first the designer must find an applicable rule set, by either choosing from previously tested ones, or by creating its own. A common rule set used in this application is known as the “4-5” rule, which states that, in terms of dungeons, a floor will “be born” if there are more than five floors around it, and it will not “die” if there are four or more floors around it. The result of this is that floors will tend to stay in organic structures, and fine-grained cells will tend to disappear after a certain number of simulation steps.

After an appropriate rule set is chosen, the designer must populate the space with floors randomly through the dungeon space. An adequate ratio of walls/floors must be found by experimenting, but we have found out that 40-50% of floors in the space yields the best results. Next, a certain number of iterations must be made in the simulation, so that the randomness is transformed into the desired cave-like appearance. Again, experimenting must be done on the number of the steps simulated for the best results, but we have found that after 3-5 steps most “artifacts” disappear from the simulation.

Finally, the last step in this process is verifying connectivity, since in this process it is not rare that isolated areas occur. One way to handle this is by using the Flood fill algorithm, which consists in, starting from a chosen point (where the stairs will be), to recursively “color” the 2D square grid until it is constrained by walls. In the end, if all the walkable spaces in the dungeon were colored, it means that it is fully connected. If there were disconnected areas, the designer must generate another level, connect the areas in some way, refrain from adding anything on unreachable areas or make wall-digging possible. Figure 3.3 shows an example of the above process. In it, “# represents walls and “.” represents walkable spaces.

Figure 3.3 – Example of a cave-like dungeon level using cellular automata.

```
############################################################
#########......###...##........####...####.....#############
####............................##.....##............#######
###................###.....##..........................#####
###...............#####...####..........................####
###...............#####....####............#............####
####...###.........####.......##.......................#####
##########..........###........##.....................######
##########...........#..........##...................#######
#########.......................#####..............#########
######.........................#######...#......############
#####..........................############....#############
####...........................###########......######..####
###..........##.................#########................###
##.......#######........#..........######...###.........####
##......########.......##............###...######.....######
###.....#######...............#####........########..#######
###......#####...##...........######........################
###......#####..####...........#####.........###############
#######..#####..####............###...........#######....###
########..###...#####......###.................#####......##
########.......######......####.................###.......##
########.......######.......##....##..................##..##
#######.......######....##.......####................####..#
######.......#######....###......####..........###..#####..#
#####........######.....######....##..........##########...#
######........###........######...............########.....#
##########...............#######.............########.....##
#############...#######..###########...#####################
############################################################
```

Source: Created by the author.

3.5 BSP Tree

The binary space partitioning method (BSP) can also be applied to level generation. It is used to make recursive subdivisions of a given space by using hyperplanes. In the case of a two-dimensional, grid-based game, a rectangle with the dimensions of the dungeon level is recursively subdivided, horizontally or vertically, into two smaller rectangles of arbitrary size, resulting in the end in a structure called the BSP tree. Dungeons can be generated with this method by using the following procedure:

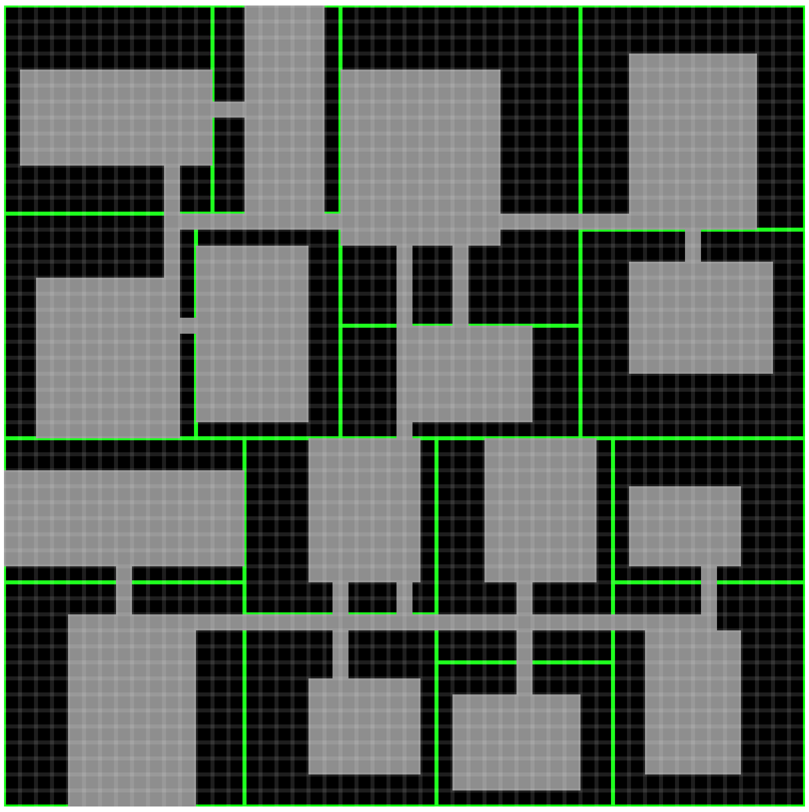
1. Start with a rectangular dungeon filled with walls.
2. Randomly choose a splitting direction, horizontal or vertical, and the corresponding coordinate for the split.
3. Split the dungeon rectangle into two sub-dungeons.
4. Repeat steps 2 and 3 for every resulting sub-dungeon a set number of times, or until no further subdivisions can be made, the criteria being a given minimal rectangle size.
5. For each sub-dungeon, create a room inside it, with size ranging from the minimum room size to the size of the sub-dungeon rectangle.
6. After all rooms are created, connect all leaves (last sub-dungeon divisions) from the tree to their sister.
7. Go up one level in the BSP Tree and connect all sub-regions to their sisters, the same way done in step 6, thus connecting a room in a sub-region to a room in their sister, or even to corridors.
8. Repeat step 7 until every level of sub-dungeon is connected to their sister.

One direct advantage of this process is that, because of the properties of a BSP Tree, the rooms will all be reachable between themselves. Figure 3.4 shows an example of this process. In it, the green lines represent the binary partitions, made in 4 iterations on a 50×50 map, with equal ratio for horizontal/vertical splits.

3.6 Genetic Algorithms

Genetic algorithms is a paradigm inspired by biological evolution in which candidates for the solution of a problem are successively mutated, crossed-over and selected through a fitness function, so that better candidates evolve, eventually possibly to the best possible solution. When applied to dungeon generation, the candidates are the dungeon levels themselves (coded in some way), the mutation and crossover operations are, respectively, random changes in a dungeon level and the combination between two different candidates.

Figure 3.4 – Example of a dungeon level generated using the BSP tree.



Source: Created by the author using the Eskerda dungeon simulator (ESKERDA, 2015).

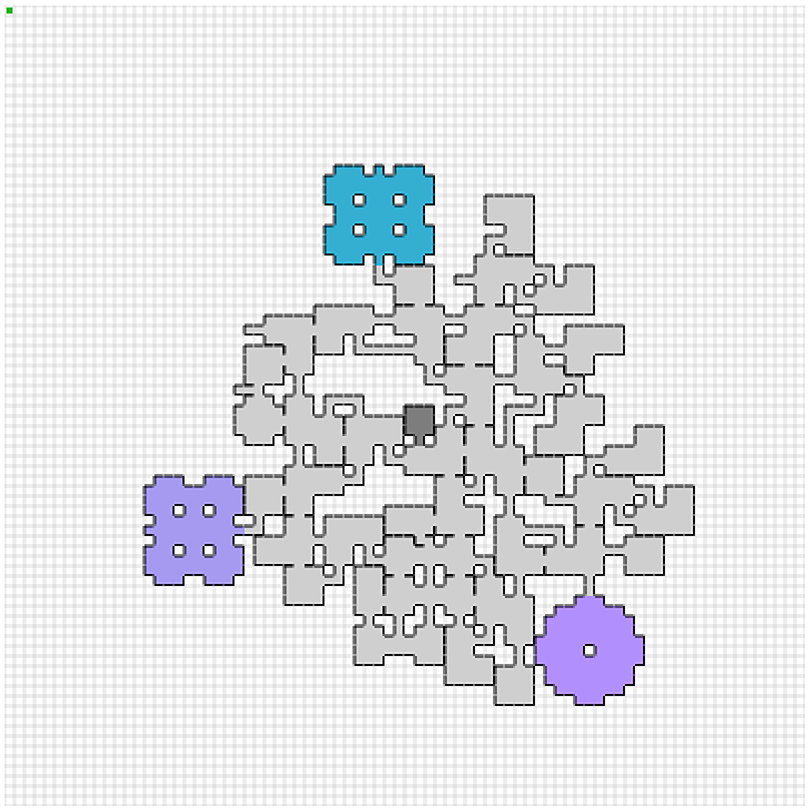
The first step is to find a way to represent candidates, commonly called chromosomes, in an appropriate format. One simple (yet impractical) alternative is to represent each cell of the dungeon level as a bit, which will be turned on if the cell is a wall, and turned off otherwise. Another way, encountered in (VALTCHANOV, 2012), is to represent the rooms in a tree structure, with the vertices representing rooms, and edges representing the corridors (or connections) to other rooms.

Then, the operations of mutation and crossover must be defined. An example of mutation operator for tree-structured chromosomes is to randomly select a number of nodes with at least one available door (leading to a corridor) and then add another number of child nodes to them. For crossover, we mention exchanging a random sub-tree between two candidates.

Finally, when it comes to selection, a fitness function must be defined. This means that a way to evaluate the quality of a given dungeon level must be embodied in a function, so that the process of selection gradually improves the candidates throughout the generations. Valtchanov's fitness model is characterized by favoring clusters of rooms with little space between them, connected by efficient hallways. It also favors maps with up to three unique feature rooms which are close to the edges of the map. The way those characteristics are analyzed is by first translating the tree structures to actual dungeon levels, and then evaluating them.

After the fitness function is defined, the candidates of a generation are randomly organized into groups called tournaments. Then, each tournament has their candidates sorted by the fitness function, and the bottom half of them is replaced by the children of the top half. This process is repeated by a set number of generations, or until a certain threshold of fitness is reached. Figure 3.5 shows an example of map generated by the above process. The colored areas indicate special rooms.

Figure 3.5 – An example of dungeon level generated by using genetic algorithms.



Source: (VALTCHANOV, 2012).

3.7 Parametrization

Parametrization is the concept of adding parameters, that is, directives on how a generator should work, and then varying their values, as a means to increase the variability of results. They can also be set by the player in a new game setup, as a way to generate the same levels, in the scope of the whole dungeon, not just a single level. Common parameters are:

- Seed: A constant (numeric value or string) used to build a level (or a dungeon) in a specific manner. It is generally passed to the pseudo-random number generator, which affects the whole dungeon (or world) creation. Multiple seeds can also be used to generate different aspects of the game.

- Motif: A parameter which represents the type of environment which will be generated. For instance, a dungeon can be set to be generated with a volcanic, aquatic or cavernous motif, among others.
- Dead ends: Corridors which lead the player nowhere can be generated at a given rate, from no dead ends to many.
- Unique features: This parameter can mean the rate in which unique features such as vaults, shops and temples appear throughout the dungeon (none to many).
- Dungeon size: The size of the dungeon can be decided by a parameter. It is important to note that if the dungeon size is bigger than the game window size, a decision must be made on what approach will be used to handle that. For instance, the game may be always centered on the player, so the game draws only the parts visible to the player's current position. Another way is to use a sliding window, in which the game slides the viewing screen a certain amount whenever the player reaches the edges of it.
- Type of corridors: Corridors can be set to form straighter or more convoluted patterns.
- Difficulty: Sets the difficulty the player will encounter throughout the game. It can affect the game in different ways, such as the monsters' spawn rate, monster strength and intelligence, the number of traps, among others.

3.8 Analysis and Comparison

In this section we will briefly analyze every technique described above, and compare them both in complexity and apparent quality of the results, starting from the standard dungeon algorithm. When creating a dungeon with classic “rooms and corridors” layout, the technique of iteratively creating a room and connecting it to the last created room with a corridor will yield results in linear time. If we call the operation of setting a tile to walkable setToWalkable, let *n* be the maximum number of rooms to be created, and for every room creation we will have to set to walkable a rectangle of cells with maximum height *h* and width *w*, we will have, in the worst case, $n \times w \times h$ operations of setToWalkable. However, since in practice *w* and *h* are bounded to a small maximum number (for example 10) due to the usual size constraint of rooms in roguelikes, we can say that the algorithm will run the so-called makeRoom n times, making it linear in the number of rooms. In fact, experiments showed that after a number of rooms have been created, most room creation attempts will fail because of lack of space, thus this technique will run fast enough even for big dungeon sizes.

Concerning unique dungeon features, most of them, like shops, temples and castles have fixed size, thus their construction can be considered constant in time complexity. However, features such as waterways or vegetation may use techniques which do not run in constant time. For techniques which use Cellular Automata, we will mention them below. For the creation of a waterway using the Drunkard Walk, which we call drunkardWalk, we define its size by the number of cells we want it to occupy. In theory, this number, which we can call waterwaySize, is bounded by the dungeon size, which is w×h. So, in the worst case, waterwaySize = w×h, and drunkardWalk(waterwaySize) $\in$ O(w×h). In practice, waterways rarely occupy the whole dungeon, instead they are a small fraction of the dungeon, like 5%. This means that it has a negligible computational cost even for large dungeons.

Mazes, on the other hand, will need slightly more processing time to be created. Using depth-first search will result in every possible walkable cell being visited exactly once, which means, if we call the number of cells (or vertices) V = w×h, and E = V×4 (because each cell will have four visitable neighbors in the case of non-diagonal maze-creation, not counting map edges), they will run in O(V) time complexity. It is important to note that, since this technique may involve deep recursion, especially at large dungeon sizes, it may cause stack overflow problems, thus it is advised to program it in an iterative way, by storing backtracking information in the maze itself.

Kruskal and Prim, being minimum spanning tree algorithms, will involve further calculations such as sorting the weights of the edges, they will run slower than the depth-first search. Kruskal was shown to run in O(E log V), while Prim runs in O(|E| + |V| log |V|) if it uses a Fibonacci heap and an adjacency list as data structures. In practical terms, any of these alternatives will run efficiently enough for any practical dungeon size.

In the case of Cellular Automata, the most intuitive way of running the simulation for a fixed-size dungeon level is, at every generation, to calculate the number of neighbors for each cell and decide if they become alive, stay alive or die. This means, for *g* generations, dungeon size w×h, and the maximum number of neighbors for a cell being nine (including the cell itself), the algorithm will perform 9×w×h×g cell updates. However, every application of cellular automata mentioned in this work uses a fixed, small number of generations, for example five. This means that, in our case, the worst case will be O(w×h). In practice, because of this fixed number of generations, we can assume that all of the applications of cellular automata in this paper will run efficiently enough for any practical map sizes.

Concerning the binary space partitioning technique described in this thesis, the number of subdivisions made in the map space will be constrained by a set number of iterations, or until it reaches a minimum rectangle size. For the version with set number of subdivisions *k*, the algorithm will have to run the makeRoom function 2^k times. Thus, it is important to keep *k* small, or else, for large maps, the amount of function calls may exceed the stack size. Fortunately, the number of iterations is generally a small number, like five.

According to (VALTCHANOV, 2012), experiments with on-demand dungeon generation by using genetic algorithms, setting the number of generations to 500 and a maximum of 100 rooms, dungeons were generated in approximately 30 seconds, using a single core of a 2.4 GHz processor, with room for optimization. This means that, if this technique were to be used in runtime dungeon generation, some kind of measure should be employed so that this processing time would not affect gameplay. Solutions include adding a loading screen after the user changes a level and generating the next level while the user is exploring the current level.

Finally, as parametrization is simply a way to increase variability through configuration, and not a technique to generate maps by itself, the running time in this case depends on the techniques employed by a given configuration.

Next, we will compare the above regarding overall quality. While this is a highly subjective measure, we can get some guidelines by talking about desirable characteristics they can have. Some of them include:

- Scalability: This relates to how it scales on larger dungeon sizes. It is tightly related to computational complexity.
- Connectivity: Whether it is fully connected by default, or further methods must be employed to guarantee it.
- Combinability: Ease of combination with other techniques or unique design features.
- Reliability: Relates to how much the technique is expected to give useful results in a practical time.
- Variability: How different each random generation of this technique will be from the others.
- Uniqueness: This means whether the technique creates results that are not made by others.

Based on the above characteristics, and keeping in mind the subjectivity of quality, we chose to compare the techniques visually by using a table. In table 3.1, we list the techniques and cross them with the desired characteristics. When no comparison is applicable, “--” is used.

Table 3.1 – Procedural dungeon generation techniques vs. desired characteristics.

Technique	**Scalability**	**Connectivity**	**Combinability**	**Reliability**	**Variability**	**Uniqueness**
Standard	Scalable	Fully connected	Yes	Reliable	Medium	Not unique
Unique Features	Scalable	--	Yes	Reliable	--	Unique
Mazes	Scalable[1]	Fully connected	No	Reliable	Low	Not Unique
Cellular Automata	Scalable[2]	Verification needed	No	Non-reliable	High	Unique
BSP Tree	Non-scalable	Fully connected	Yes	Reliable	Medium	Not unique
Genetic Algorithms	Non-scalable	Fully connected	Yes	Non-reliable	Medium	Not Unique
Parametrization	Scalable	--	--	Reliable	High	Unique

Source: Created by the author.

1 After implementing the technique, we discovered that it can only be considered scalable if it is implemented in an iterative way.

2 After implementing the technique, we discovered that the number of generations must be small, or an efficient evaluation technique must be applied for it to be scalable.

4 ARTIFICIAL INTELLIGENCE

In this chapter, we will describe, analyze and compare a number of artificial intelligence techniques which can be applied to agents in games. More specifically, we will express them in terms of their application in roguelikes.

One distinction which comes up when defining artificial intelligence for games is that of “pseudo-intelligence”. This means that, even though the actor works as intended, which is to provide challenge and fun for the player's gameplay, the actual programming may sometimes be considered “mindless”. For instance, agents following a set of if-then rules may not be considered rational, whereas an agent which possesses complex tactical planning, learning capabilities and emotions that regulate its interactions with the environment, may be called truly “artificially intelligent”. Regardless, in our work we will acknowledge those differences, but ultimately ignore them. The reason for that is because in the area we are exploring, which is games, what really matters is the resulting gameplay effect (and also performance), not philosophical considerations.

The following sections will contain a number of techniques for actors' artificial intelligence, fine-tuned to roguelike games. After that, in section 4.8 we will analyze them in terms of computational complexity, qualities and drawbacks, so that we can increase our understanding of what techniques to use in a practical implementation, by themselves or in combination.

4.1 Path-finding

To start with, we discuss a technique which, despite being essentially a shortest path graph problem, is a cornerstone technique for most games that involve actors traversing over a terrain. It consists of finding the best path an actor can traverse to reach a certain goal. However, the definition of “best” path depends on the application, because the game designer has to set the parameters for evaluating path quality. For instance, the most basic (and most commonly used) parameter is distance, which means, in the case of roguelikes, the number of squares in the grid the actor must walk in order to reach its goal. Other parameters include exposure to enemies, smoothness of path and avoidance of undesired areas.

Next, we mention some examples of techniques for path finding used in roguelikes. These are:

- Euclidean distance: In this technique, the actor will calculate the straight line between it and its target, and move in that direction. The euclidean distance can be calculated this way:
  - Calculate the vertical and horizontal distances, *dx* and *dy*, between the actor and its target;
  - Calculate the euclidean distance between them, by using $D=\sqrt{(dx)^2+(dy)^2}$.
  - Normalize it to length one, and separate it into two coordinates, by using $dx=round(dx/distance)$ and $dy=round(dy/distance)$.
  - Move the actor by offsetting its current position by *dx* and *dy*.

  While this technique is simple to implement and negligible in computational cost, the actor may move unnaturally and may be stuck in corners, so the designer must find ways to handle these issues.
- Breadth-first Search: In this technique, the actor will search the space for its target by iteratively increasing its search range. It will start by looking at all adjacent cells (that is, of distance 1). If it finds the target, it means the target is right next to him and he can act directly. If not, it will increase the range by 1, thus covering all squares of distance 1 and 2. It is important to note that the process does not start all over when the range is increased, but instead just continues from the previously visited nodes. When the target is found, the algorithm backtracks the tiles that led to the target and builds a path. This process is guaranteed to end and to find the target, if such a path exists.
- Dijkstra's Algorithm: A technique for finding the shortest path between two nodes in a graph, Dijkstra's can be applied to roguelikes by modeling the dungeon map as a graph, where the squares in the grid are vertices and the edges are represented by which neighbors a certain square can access. After that, the algorithm can be applied directly as intended, by visiting every other nodes, starting from the actor's node, and successively calculating the shortest distances to them until the target node is visited, which will then mean a shortest distance to it was found.
- A* algorithm: While the shortest path is guaranteed to be found by Dijkstra's algorithm (if there is any), depending on the size of the dungeon and the number of simultaneous agents, the cost for its worst case scenario, which is to visit every node

in the graph, can be prohibitive. A* algorithm is a generalization of Dijkstra's, where the function cost is, instead of the real distance cost, an approximation of it by the use of heuristics. What this accomplishes is, at the cost of possibly losing the guarantee to always find the best possible path, the number of searched nodes can be greatly reduced when adequate heuristics are chosen. According to (STERREN, 2008), "*The heuristic function typically provides an estimate of the remaining costs to the destination, such as the vector length divided by the maximum speed.*"

Finally, it is important to mention that several optimizations can be done to path-finding. Those can be either aesthetic or performance optimizations. In the case of A*, as an example of aesthetic optimization we cite the procedure of straightening paths in (RABIN, 2008). In it, bearing in mind that there may be several shortest paths to a target, the idea is to slightly penalize the heuristic value of non-straight paths, for instance by -0.0000001, so that in the end the straightest path will be marginally shorter than the alternatives, thus being chosen. However, in this process the game designer must be aware of the performance tradeoff, because this procedure will make the pathfinder consider many more permutations to find the straightest one.

One way described by Rabin to resolve this issue, which is also an example of performance optimization, is to utilize *hierarchical pathing*. This consists of breaking down the terrain into zones (in the case of roguelikes, a room would be a perfect characterization of a zone), so that an actor that wants to reach a target in another zone will compute only the path to the next adjacent zone in the path to the target instead of the whole path. Consequently, the permutations of paths will drastically reduce, and the extra steps for path straightening can be considered trivial in processing cost.

4.2 Stateless Actors

Stateless actors are the most simple form of artificial intelligence for monsters. They consist of a set of cause-effect rules which an actor will check at every game cycle and act based on them. This means that they do not possess internal memory whatsoever. For instance, a simple monster could follow this procedure:

1. If I can reach the player, then attack him;
2. Else if I can't reach the player, but can see him, then move towards him;

3. Else, stand still.

By repeating this procedure every turn, the monsters will blindly move towards, and attack the player. While this works, further checks and reactions can be added so that more complex behavior can appear. For example, the actor might check if its health is too low and run away from the player, or it can use ranged weapons while in range, and only use close combat attacks when cornered, among others.

4.3 State-machine Actors

State-machine actors, on the other hand, have an internal memory, that is, pieces of knowledge, innate or acquired through experience, that will help them act more flexibly. When it comes to innate knowledge, it can be a piece of information the actor has intrinsically, such as the number of current and total hit points it has. Knowledge gained through experience, on the other hand, is gained through the observation of events that happen during the actor's life. For instance, a monster might evaluate the strength of the player by watching it battle other monsters, and run it he is too strong, or charge immediately if he is too week. Another example would be a monster getting angry if he watches the player kill another monster of the same species, and change from a state of caution to anger.

The kind of state we exemplified as the monster being angry or cautious is called by (DILLINGER, 2010a) “tactical state”. This is a form of internal memory that indicates, as the name implies, in which tactical situation the monster is at a given moment. For example, a monster could be ignoring the player, and based on some event in the game, like being attacked by the player, or getting cornered, it could become hostile.

Table 4.1 shows an example of a state-based monster, with its possible states and transitions. In it, the “State” column denotes the name of the state, “Observation” means the observation level, “loud-noises” being only hearing noises loud enough to wake it up, “regular” being observing anything casually visible, and “extended” meaning observing everything attentively. The “Missing Treasure”, “Hungry” and “None” columns are the transition rules. Thus, for example, if an actor currently in the state SLEEPING hears loud noises, it transitions to state WAKING. In addition to that, for every state, there is a corresponding behavior pattern, implicit in the names of the states for brevity.

Table 4.1 – Example of a state-based monster with tactical states and transitions.

State	Observation	Loud-noises	Missing Treasure	Hungry	None
SLEEPING	loud-noises	WAKING		HUNGRY	
WAKING	regular		ENRAGED	HUNGRY	CURIOUS
ENRAGED	regular				WAKING
HUNGRY	regular		ENRAGED		
CURIOUS	extended		ENRAGED	HUNGRY	

Source: Adapted from (DILLINGER, 2010a).

4.4 Swarm Intelligence

Swarm intelligence is a technique of stateless intelligence in which a number of agents coordinate their movements so that collective intelligence emerges from irrational behavior. One example of swarm intelligence technique was inspired by the way migratory birds fly in formation by local coordination, which is called flocking. In (RAYTHEON, 2008), the author talks about four rules, labeled *steering behaviors*, which can be used to govern groups of autonomous agents so that they present realistic patterns of conduct. These are:

- Separation: An individual should steer to avoid collision with flockmates.
- Alignment: Which means steering toward the average flock direction.
- Cohesion: Steering to move toward the average position of the flockmates.
- Avoidance: A behavior in which an individual will steer to avoid running into local obstacles or enemies.

The idea, then, is to calculate a velocity vector for the flocking agent at each turn, guided by the above rules. This means, in terms of a turn-based roguelike, that we must calculate the direction in which the agent will move, so that those rules are satisfied the best way possible. Also, it is important to decide on conflict resolution, when those rules favor antagonizing movements. A simple way to do that is to set priorities on the rules, like prioritizing separation, so that the agent will try its best not to collide with the flockmates, or crowd an area too much.

The designer can also define a number of constraints which can be applied to how the agents can move and react. These change the way the overall flocking behavior will happen. For instance, what Raytheon considers possibly the most influential constraint is the perception range of each agent, that is, how far it can look to check for flockmates, obstacles and enemies. Another constraint is regarding speed, which, in the case of roguelikes, would be the time it takes to move one tile or execute an action.

Although these four rules were defined in the context of flocking, the general idea can be applied to other patterns. For instance, the rules could be defined in a way such that the flockmates would always try to approach a single target all at the same time, which would characterize ambushes instead of squad formation. There are also several other techniques for swarm intelligence, such as the bees algorithm and ant colonies, but as we couldn't find applications for them on roguelikes (or dungeon crawlers in general), we chose not to explore them in this work.

4.5 Genetic Algorithms

Applying genetic algorithms to actors, the chromosomes would represent a set of built-in knowledge and behaviors, which would be progressively modified by genetic operators as a means to be selected through a fitness function. In this case, the fitness function would be defined so as to select the best features for an actor, such as smart decision-making and natural behavior. As usual with genetic algorithms, the designer has to model and fine-tune the technique specifically to the application. In the case of roguelikes, (DILLINGER, 2010b) talks about the setup of several parameters, including:

- Population size: Total number of individuals in the simulation. Large population sizes will make the process take too long for convergence, while small ones may lead to fast convergence, but unimpressive results.
- Mutation rate: This represents both the frequency in which mutations occur to individuals, and the amount of change each individual mutation will do. Dillinger argues that it has to be considered in combination with the types of mutation operators and the fitness function. Functions that generate fitness landscapes with broad curves and a few local optima should have a high mutation rate, with each mutation changing only small portions of the candidates, whereas for complex landscapes, it is better to have low mutation rates, since the changes made will necessarily be large.
- Gene map: The gene map, also called chromosome, usually a vector or sequence of values, will represent the characteristics of the candidate, and the way this is modeled is important. One useful property is that of locality, which means that features that tend to influence each other should be close together on the vector. This way, if the combination method is set to the common cross-over operator, which consists of transferring a contiguous number of values of both parents to the new candidate, those

closely related features will be transported together to it. One way to define the gene map for stateless agents is, because they are a set of nested if-conditions, to map the logical tree structure into the vector, with each value representing an if condition. This way, the intermediate nodes in the tree will represent the if-conditions, and the leaves will be the actions. This is advantageous to the combination operator, as explained below.

- Method of combination: With the tree structure, you will get isomorphic structures between different candidates, thus the same gene will always mean the same thing for any individual in the population. Also, the combination operator can be defined as sub-tree swapping, which means you remove a branch of one candidate, and replace it with another branch from a different candidate, thus preserving the complex hierarchical relations on the swapped branches. This means good combinations of genes will be selected together throughout the simulation, and that way convergence to the best possible set of genes will be faster.
- Fitness function: Usually the most important, yet the hardest parameter to define, it is no exception in roguelikes. Dillinger states that fitness functions are particularly hard to define for RPG games, because most enemies have very short lives (the time it takes for the player see them and reach them). One possibility is to define the criterion as the average time a certain candidate lives, but that would only means the monsters would become good at running away, which is not necessarily the best option for every type of monster. Another option is the amount of damage inflicted to the player, but most monsters in roguelikes don't damage the player at all, and the most challenging ones inflict hardship to the player by other creative means. Finally, he states that a good technique for measuring fitness is by evaluating, over a monster's lifetime, how much it cost to the player, for instance in the form of damage, spell charges and equipment damaged or destroyed.
- Elitism: Expresses how much a more fit individual will be preferred for gene propagation over the other less fit ones. It is important to find a middle ground between high elitism, which would potentially discard good genetic material, and pure random choice, which would give no advantage to more fit individuals whatsoever. The procedure preferred by him consists of picking three candidates randomly, ordering them according to their fitness value, then assigning chances to breed and to

be replaced depending on their rank within the three. Table 4.2 shows examples of percentages for this form of elitism.

Table 4.2 – Degrees of elitism for three randomly chosen candidates.

Candidate	**“Normal” elitism**		**“Small” elitism**		**“Slight” elitism**	
	Breed	**Replace**	**Breed**	**Replace**	**Breed**	**Replace**
Best	50%	20%	40%	25%	35%	32%
Medium	30%	30%	35%	35%	34%	33%
Worst	20%	50%	25%	40%	31%	35%

Source: Adapted from (DILLINGER, 2010b).

4.6 Emotion-based Actors

By employing psychology models of emotions to actors, they can behave in a more natural, unexpected way. For instance, a monster would be more prone to run away from an encounter if its courage parameter was lower than the average. Such models of emotion are taken from research in areas such as psychology and cognitive science.

One extensively used model of emotion is that of OCC (ORTONY, 1988). In it, emotions are broken down into three categories, taking into account their timescale:

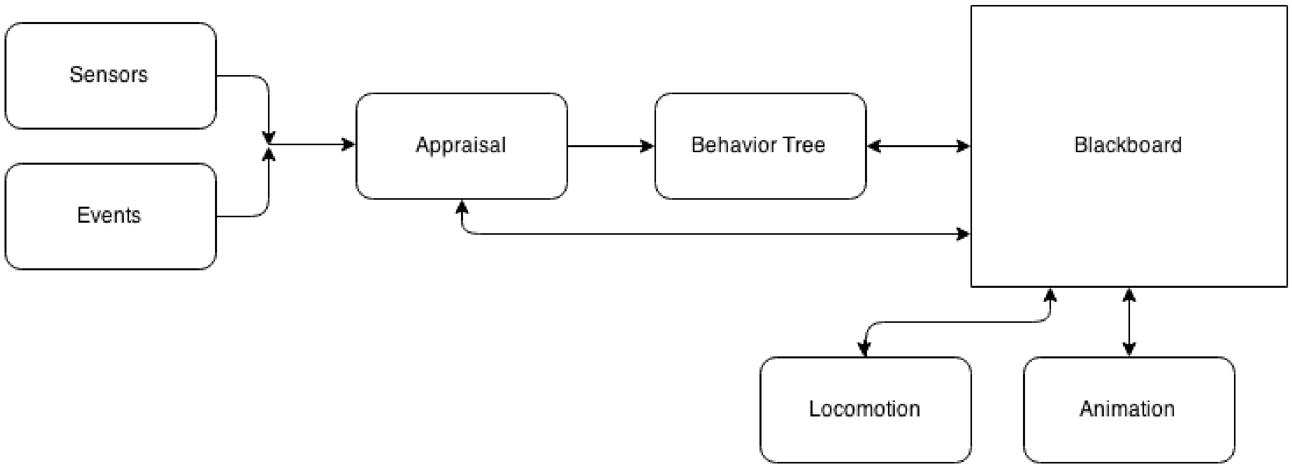
- Emotion: This category is represented by reactions to events, objects and agents expressed on a small timescale, such as minutes and even seconds. An example on roguelikes would be a monster watching a player attack another monster of the same category and disliking it.
- Mood: Emotional situation over a timescale ranging from a few days to a few months. A way to represent this in a simplified manner was proposed by (EGGES, 2004), in which the emotional state of an actor varies in the range -1 to 1, floating-point. This way, negative values would represent “bad mood”, which would affect the way the actor perceives events accordingly.
- Personality: Personality is a set of traits that guide the actor's actions throughout its existence, rarely (if ever) changing. The OCEAN model, presented in (MCRAE, 1996), separates the traits as openness, conscientiousness, agreeability, extroversion and neuroticism. Each of these traits are represented by a floating-point number ranging from 0 to 1. Thus, for instance, an actor with neuroticism set to 1 would be very prone to attack a possible foe at first sight, whereas if set to 0, it would always wait for provocation before attacking.

After an emotion model is chosen, it must be embodied in an architecture so that it can have practical effects on the actors' behavior. In (CARLISLE, 2011), the author utilizes an emotional framework consisting of a blackboard which will be inspected by a simple behavior tree. A blackboard is a common knowledge base, which is iteratively updated by a number of specialist knowledge bases, starting from the problem specification and trying to achieve a solution. In this case, the actor's behavior tree, which is in practice a state machine that guides the actor's conduct, will serve as the specialist.

After that, an appraisal technique must be defined. It serves the purpose of creating new goals and evaluating objects and events. Also, a way to map sensory input into changes in the agent's emotional framework must be defined, which Carlisle calls arousal.

An example of both processes in working in conjunction would be the behavior tree executing a sequence of nodes, result in the query for the availability of food in the vicinity. The appraisal/arousal class, then, has to determine the agent's emotional reaction to each sensed object. Thus, from a list of possible sensed objects, it will evaluate and sort them, based on previous experience, current mood and personality. This will influence the actor to try and attain the best object in the sense of nutrition it has evaluated. Figure 4.1 shows an example on how a typical emotional update loop could be represented.

Figure 4.1 – A diagram representing an update loop for an emotion-based agent.



Source: Adapted from (CARLISLE, 2011).

4.7 Instancing of In-game Entities

Instancing of in-game entities is a technique of procedural content generation (PCG) which can be applied to monsters' artificial intelligence. In it, the parameters of the creation of an actor would be randomized so as to generate a population of unique individuals, with statistically insignificant chance of repetition. For example, if the parameters of emotion-based actors were randomized, actors with different “personalities” would appear. Examples of parameters that could be randomized while instancing include:

- Attributes: Strength, dexterity, intelligence and others can be varied to create unique enemies. While these are not considered artificial intelligence techniques by themselves, the monster can be programmed to act accordingly to its strengths and weaknesses. Thus, for instance, a weak monster could favor ranged combat.
- Personality: As stated before, emotion-based techniques can be randomized so that a population of monsters present greater variability. In this case, the more slow-changing, or even fixed aspects of emotions should be varied, which means personality. Personality traits such as extroversion, neuroticism and agreeability could be randomized in certain ways so that actors with unpredictable emotional behavior can appear to the player.

It is important to note that, by blindly randomizing the parameters of actors, unsuitable results may appear. For example, a monster which was randomized to have extremely low hit points, high ranged combat capabilities, yet had the emotional propensity to blindly charge at the player on first sight, would be an impractical one. Thus, it is important that the designer guides the process in some way, like adding constraints and cross-exclusions so that undesired combinations can't be created.

4.8 Analysis and Comparison

In this section, we will briefly analyze and compare the artificial intelligence techniques described above, when applied to roguelikes. To start with, we will consider the path-finding problem, which is used, with varying levels of sophistication, in every actor in CRPGs.

Path-finding is the problem of finding the shortest way to a certain target. This can be as simple as blindly walking toward the target in a straight line, as in the euclidean distance procedure, or it can potentially involve going through every square in the dungeon at least once. In the case of euclidean distance, the procedure is constant in time complexity. In the case of Dijkstra's shortest path algorithm, it has been shown that its complexity in worst-case scenario, using Fibonacci heaps, is $O(|E| + |V| \log |V|)$, where V is the number of square cells in the dungeon level, and E is the number of sum of the possibilities of movement for every cell (edges).

For the A* algorithm, the time complexity depends on the heuristic used. If there is a single goal, the search space is infinite, and the error of the heuristic function will not grow faster than the logarithm of a hypothetical perfect heuristic, then its time complexity will be polynomial in the branching factor. In the case of usual square-grid roguelikes, the branching factor is never higher than eight, since there are at most 8 neighboring squares to visit. Also, in roguelikes the search space is finite (limited to the size of the dungeon), and there is a single target in the search, which is the player or a target square to move by the monster. Finally, we mention that the search space can be greatly simplified by the use of the hierarchical path-finding, as it will be reduced from a graph of square cells to a much smaller graph of interconnected rooms.

Next, we will talk about the complexity of stateless actors. Given their simple reflexive nature, their computational cost is trivial. They consist of a loop of checks, with each check being a constant-time if-condition. Thus, this technique runs in constant time complexity. However, if any of the checks include an operation that is more complex in time, then the whole procedure has the time complexity of that operation. For instance, if the monster has the possibility to attack all other actors, including monsters (he may be enraged), and he has to evaluate the best creature to attack, then he will have to check a list of all visible actors for such an evaluation.

Thus, in the worst case scenario, which is the monster being able to see all other actors in the dungeon level (for instance in a big room fully illuminated), the procedure would be linear in the number of monsters in the dungeon. This leads us to conclude the time complexity for stateless actors can only be properly defined after designing the specific procedure, and not in a general basis. In practical terms, as stated before, we can consider their processing time to be negligible.

Similar to stateless actors, the loop itself of state-machine actors is trivial. The monster will check for its current state, input and internal knowledge, and will trigger a certain Artificial Intelligence behavior. Also, if the inputs trigger a state transition, it will simply change the state variable, which is also trivial. Thus, the time complexity for this kind of actor is dependent on the kind of artificial intelligence techniques employed for each of its tactical states.

Swarm intelligence is a form of collective behavior which supposes mindless actors which act coordinately, which means it is also a form of stateless intelligence. This means the processing cost for this kind of technique is dependent on the specific behavior which is going to be emulated. For instance, in the case of flocking, the method each actor uses is to check the position of each other visible flockmate and calculate a motion vector which maximizes the steering behaviors the best way possible. In the worst case scenario, the agent will have no visibility constraints and will be, at a given moment, seeing every other flockmate at the same time. This means that the time complexity will be linear in terms of the size of the flock. In practice, most practical uses for flocking will add visibility constraints to the actors, thus reducing the number of flockmates to be checked to a fraction of the flock size.

Unlike the previous techniques, genetic algorithms applied to roguelike actors take too much time for run-time scenarios. This means it is considered a pre-processing technique, in which programmed knowledge or past gameplay experience is used to define the fitness function to be used by the selection process. According to (DILLINGER, 2010b), all the parameters of the process must be fine-tuned so that convergence to desirable candidates happen in a feasible time. For instance, population size must not be too large, or each simulation step will take unacceptable amounts of time. Also, the operators must be tested so as to find good combinations, or else the process may get stuck in local optima, or even not converge at all. Thus, the game designer must decide when this process will take place. Alternatives include once every certain number of games, whenever the player starts to learn the monster patterns and becomes too efficient at defeating them and whenever a player character dies.

When it comes to the presented framework for emotion-based actors, we must break it down into its components, so that we can analyze them separately:

- Behavior Tree: Acting as the knowledge specialist in the blackboard system, it consists of organizing the execution of plans into a tree of tasks. This means that the processing time depends on the complexity of the tasks themselves. For instance, the act of shooting an arrow at a chosen target consists of a series of sub-tasks, namely: check if the quiver has an arrow, pull an arrow from the quiver, aim at the target and shoot. All of those tasks are constant in time, thus the complete process is also constant. However, there are cases where non-constant time tasks are necessary such as deciding where to run to, as a means to escape from a strong enemy, which would take the actor path-finding calculations.
- Appraisal: Acting as the primary input processor in the system, it maps the input into changes in the blackboard. It may add or remove goals, and also affect the emotional

values that are associated with entities stored in the blackboard. An example of task for the appraisal module is to return a sorted list of entities queried by the behavior tree according to their emotional value, so that the best one can be attained. Such process is linear in the number of objects within the line of sight of the actor. Another task is to simply change the emotional value of a certain entity in the blackboard, which would be constant in time.

- Blackboard: The blackboard system, being a dynamic repository of knowledge, goals and partial solutions to goals, has no laborious processing, as it simply a place where the specialists (appraisal, locomotion, behavior tree, etc), look for knowledge and act upon.

Thus, it can be said that the complexity of this framework is dependent on the application and consequently the complexity of its specialist modules. In the case of the emotional framework applied to roguelike actors, as the variables involved in the inputs, locomotion and even the behavior tree are generally simple, it seems that such an approach would scale well for all practical sizes of dungeon and number of agents.

Finally, concerning instancing of in-game entities, as it merely randomizes the parameters for the spawning of actors, which would have been spawned anyway, it can be considered negligible in time overhead.

Following, we will compare the techniques in terms of their overall quality so that we can have a better understand of their strengths and weaknesses. This way, it is easier to choose which ones to use, by themselves or in combination, while designing a game. Some of the desirable characteristics for these techniques are:

- Scalability: Whether or not the technique will run at acceptable times for a large number of actors.
- Combinability: Relates to how well a technique can be combined with the other ones.
- Reliability: In the sense of artificial intelligence for actors, this relates to the amount of trust a given technique will produce behaviors that will work in interesting ways, as opposed to, for instance, getting the actor stuck or doing remarkably senseless actions.
- Uniqueness: This means whether a technique will generate behavior that only it can.

Next, we will compare the techniques according to these characteristics. This can help understand what techniques are better for what kinds of applications. Contrary to chapter 3, we chose to make the comparisons of the techniques in the form of prose, because some of them are dependent on the “building blocks” that are used for their design, thus a table would not be instructive enough, and would have several exceptions.

Concerning scalability, which is closely related with computational complexity, we have concluded that the stateless techniques have no inherent complex computation, and are scalable for any practical number of concurrent actors for roguelikes, provided their constituent modules are also tractable. This is also the came for the state-based actors, as the addition of states does not add significant overhead. Concerning emotion-based actors, as long as the tasks of their behavior tree are not overly complex and their appraisal module does not have to process too many complex interactions with the environment, which is generally the case for roguelikes, they are scalable too. Genetic Algorithms, on the other hand, must not have a very large population size, and their operators must be fine-tuned so as to achieve convergence fast, otherwise it would be impractical. Regarding path-finding, if euclidean distance or A* with hierarchical pathing is used, large dungeon sizes, with large concurrent actors may be used without noticeable delay. Finally, instancing of in-game entities does not incur extra processing, thus it works for any number of actors.

Concerning combinability, Path-finding is used by every other actor intelligence technique, so it is fully combinable. Swarm intelligence, on the other hand, does not combine with the other techniques, as it has its own rules of behavior. The emotional framework is a form of state-based machine intelligence, so it does not work with stateless actors. Also, once the genetic algorithm procedure chooses the best candidate, its characteristics will be defined by its genes, so it does not combine with other techniques. Finally, instancing of in-game entities will work for every technique that has randomizable parameters.

In regards to reliability, the techniques of path-finding, stateless and state-based actors will always work as expected. However, the other techniques may present unexpected results. In the case of emotion-based actors, some combinations of personality traits may lead to behavior detrimental to the actors, such as suicidal or excessively indecisive behavior. As for genetic algorithms, as there is much fine-tuning to be made, the process is prone to error, and convergence to erratic-behaving actors can happen. The same goes, to a lesser extent, to swarm intelligence techniques, because the steering behaviors and constraints must also be fine-tuned, so erratic behavior may happen without thorough testing. Finally, unrestricted parameter randomization may lead to impractical combinations of parameters, so the designer must define constraints for the procedure.

Lastly, when it comes to uniqueness, we note that more unique, interesting behavior happens when more sophisticated techniques are used. Thus, for instance, while stateless actors are simple in nature, by adding a large number of conditions to them the designer can expect predictable, but interesting results. However, unique behavior comes more naturally to techniques with different proposals, such as swarm techniques and the addition of emotions to characters. We also note that, while genetic algorithms can provide interesting results, they can be programmed into both state-based and stateless actors. Ultimately, we believe the best form of adding uniqueness to the behavior of actors is to add randomization of parameters while instancing.

5 IMPLEMENTATION

In this chapter, we will detail the development process of the roguelike game for this thesis. In the next section, we will describe and analyze the general implementation of the game, from a practical point of view. Then, in the following sections, we will talk about the development of some of the techniques explored in chapters 4 and 5. More specifically, we will detail the implementation choices we made for those features after having analyzed and described them from a theoretical point of view. Finally, in the last section we will present the results of the benchmarking process made for this work, which comprises the implemented techniques for dungeon generation and artificial intelligence. Also, based on these results we will compare these techniques regarding performance.

5.1 Experimental Environment

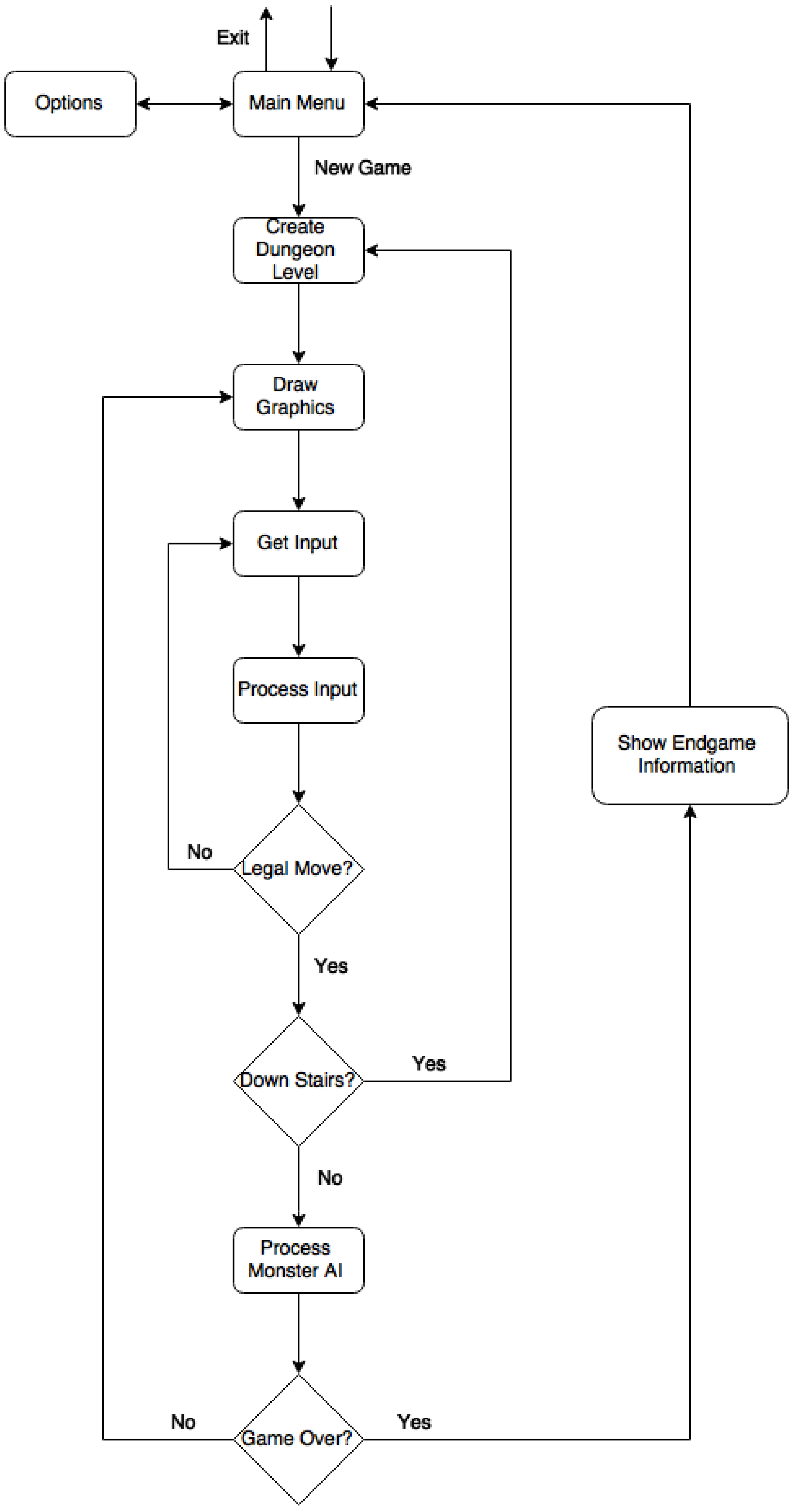
The programming language used for the development of the prototype was Python, compiled under version 2.7.9. A library called libtcod (ROGUECENTRAL, 2015), aimed at aiding the development of roguelikes, was used. It provided certain features that were not the focus of our development, such as graphical output, so that we could focus on the general implementation and chosen design features. Also, the experimentation was made on a PC using Intel® Core^{TM} i3-4150 CPU @ 3.50 Ghz, with 8GB of DDR3 RAM. The operating system running was Microsoft Windows 7 Home Premium.

5.2 General Implementation

In this section, we will detail the choices we made for the general implementation of the prototype. It comprises features that were not central to the analysis and development of the chosen design features, but nonetheless had to be carefully thought of, so that they would facilitate the embodiment of dungeon level generation and artificial intelligence techniques.

As mentioned in chapter 2, the game designer must decide how to build the game processes and the transitions between them triggered by the player's interaction with the game. Such a process is called the game workflow. Figure 5.1 shows a high-level abstraction of the workflow for the game designed for this thesis.

Figure 5.1 – Workflow for the game prototype developed for the thesis.



Source: Created by the author.

When it comes to time progression, we used a simple turn-based approach: For every turn, first process the player input (moving, attacking enemies, going down stairs, etc.), then process the turns of every enemy (moving, attacking, standing still). This equates to a simple yet effective order where the player is always the first to move, which we found to be a common choice for classic roguelikes. The drawback of this is that a speed system (where faster actors act more) is not straightforward to add to the game. A simple way we found to add speed to the player is, depending on how fast he is, he may have the chance to take a double turn, according to a dice roll. However, this was not added to the final version of the prototype.

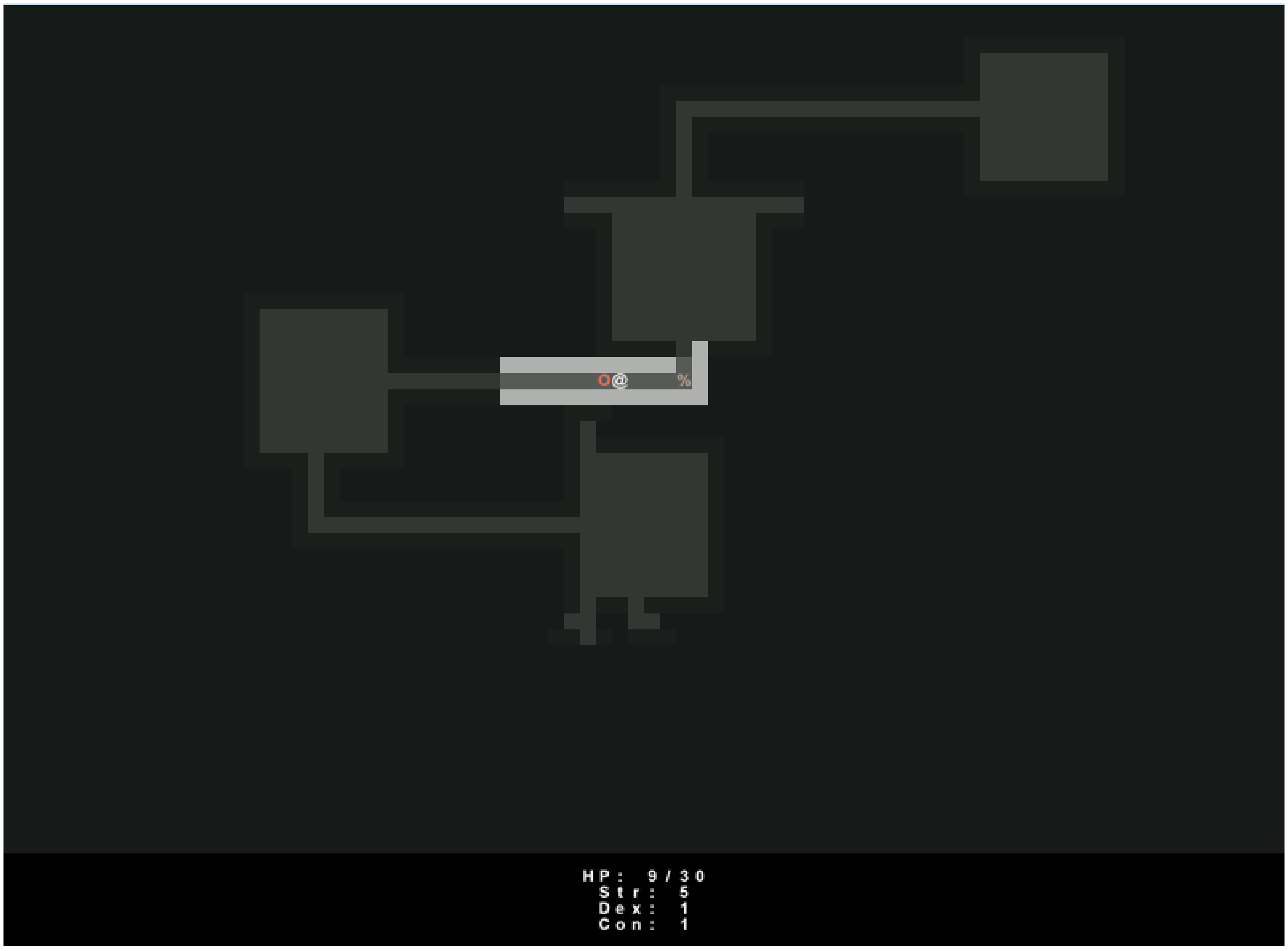
Regarding graphics, the output method used in the prototype can be called "colored ASCII", as the graphics are restricted to RGB-colored characters and also the background color of every grid cell in the viewing window. The exception to that is the main menu background, which features a picture in the png format. Figure 5.2 shows a typical session of the game. In it, the player has already explored some of the dungeon rooms, killed an enemy (whose body is represented by '%') and is now facing an Orc (represented by 'o'). We note that the player's hp is low (9 of 30), so it might be a good idea at this point to run or drink a potion.

For the user interface, we followed the traditional roguelike system, which relies heavily on keyboard interaction. In it, a set of keys are mapped to their respective effects in the game, like: Directional keys for moving the character and attacking enemies, 'i' for showing the inventory, '>' for going downstairs, ',' for picking up items and 'Esc' for going back to the main menu. We also added minimal mouse support, restricted to 'mouse-look', which means pointing at a certain entity and getting its information.

Concerning storylines and quests, our prototype featured a single, simple objective: To go down dungeon levels, defeating enemies, collecting items and getting stronger, so that in the last level the player will have to face and defeat the final boss monster. In case the player defeats the final boss, the game is considered won and ends.

The above-mentioned 'getting stronger' relates directly to what is called character development. In our prototype, whenever the player character defeated a monster, it received experience points. When he reached a certain threshold, he advanced an experience level, in which case he could improve one of his attributes, namely strength, dexterity and constitution. Each of these provided him with advantages while fighting monsters, like dealing more damage when attacking, blocking more damage when defending and having more hit points.

Figure 5.2 – A typical session of the game developed for the thesis.



```
HP:  9/30
 Str:  5
 Dex:  1
 Con:  1
```

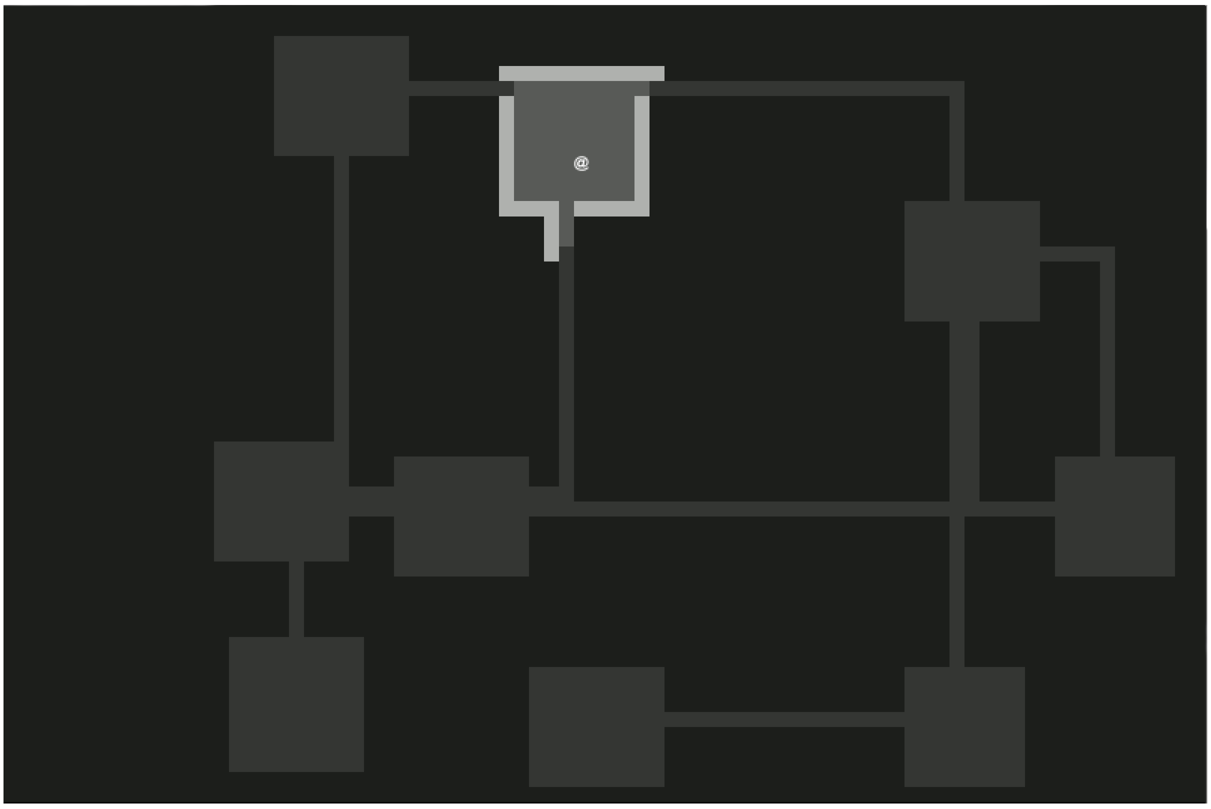
Source: Created by the author.

5.3 Procedural Dungeon Generation

The first technique for dungeon generation we experimented with was the basic iterative approach, which consists of creating rectangles of variable sizes in random positions over the dungeon level, and then connecting them with corridors. We call this the iterative approach because of the way the corridors are connected: For every new room, we connect its center to the center of the previously created room. This way, connectivity is guaranteed. The parameters for this process are the maximum number of rooms (which may not be achieved depending on the size of the dungeon) and the minimum and maximum size of the rooms. Figure 5.3 shows a dungeon level using the basic iterative approach. The '@' represents the player, and the squares are brighter when inside the player's field of view (FOV).

The next technique we programmed was cellular automata, in which the life of 'organisms' is simulated on a two-dimensional, grid-based rectangle, based on a set of rules for reproduction. When applied to roguelike, we started by setting the whole dungeon level as

Figure 5.3 – Example of a dungeon level generated using the basic iteration technique.

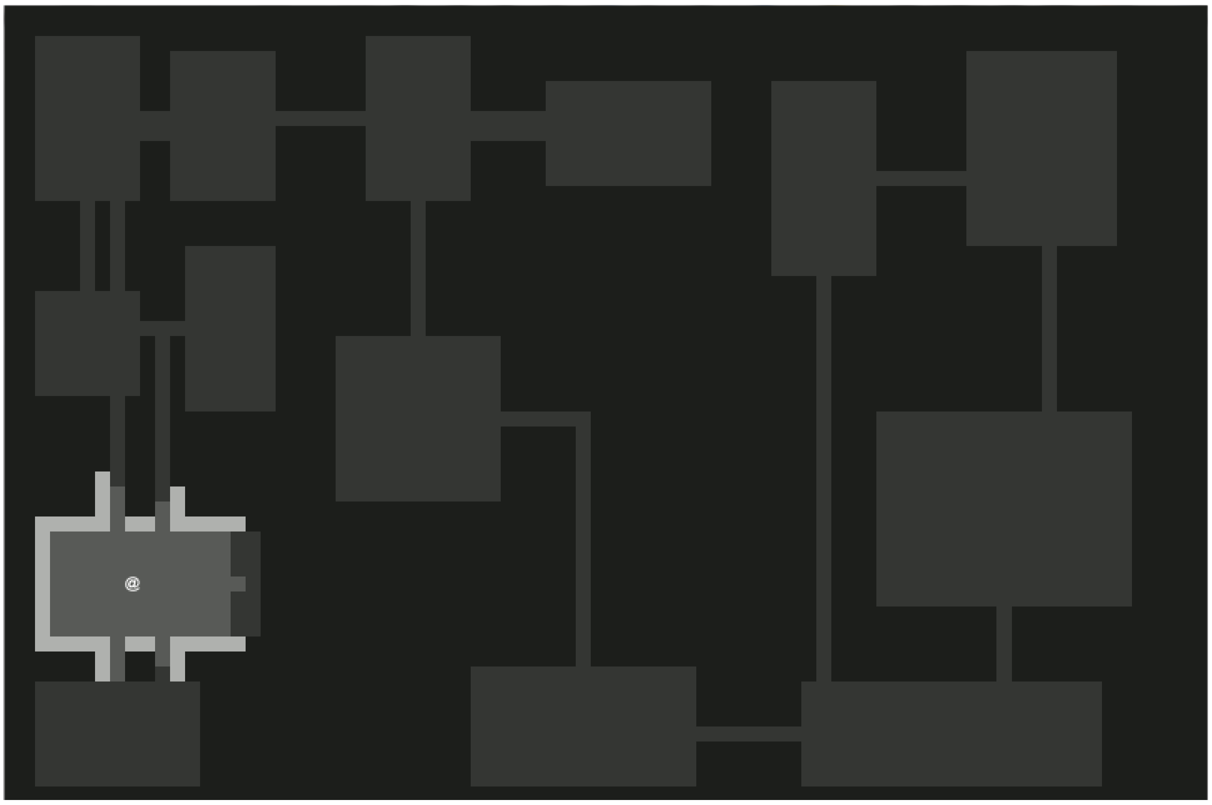


Source: Created by the author.

walls, and defined the 'organisms' as walkable tiles (floors). Then, we added randomly placed floors throughout the dungeon level, at a ratio of 50% of the wall tiles. After that we ran the simulation by following the 4-5 rule, which states that cells with less than 4 alive neighbors die of starvation, between 4 and 5 stay the way they are, and with more than 5 they become alive (which in our case means becoming a floor). The parameters for this process are number of generations and starting wall/floor ratio. Figure 5.5 shows a dungeon created using the cellular automata technique with the 4-5 rule.

The last technique we experimented with was the generation of mazes. For that, we implemented a randomized depth-first search. Starting from a dungeon level filled with walls, the algorithm advanced recursively in one of the four horizontal and vertical directions (which means not diagonal pathways), always making two moves at a time and checked if the direction to where it was going to advance was not already a floor. Because of that, a tree of corridors was generated, with its root at the starting location. Thus, no check for connectivity was necessary. We have also experimented with biases for certain directions, which means the algorithms generate longer corridors in that direction, but we did not add that to the final version of the prototype. This means there were no parameters for this technique. Figure 5.6 shows an example of a maze generated using this technique.

Figure 5.4 – Example of a dungeon level generated using the BSP Tree technique.



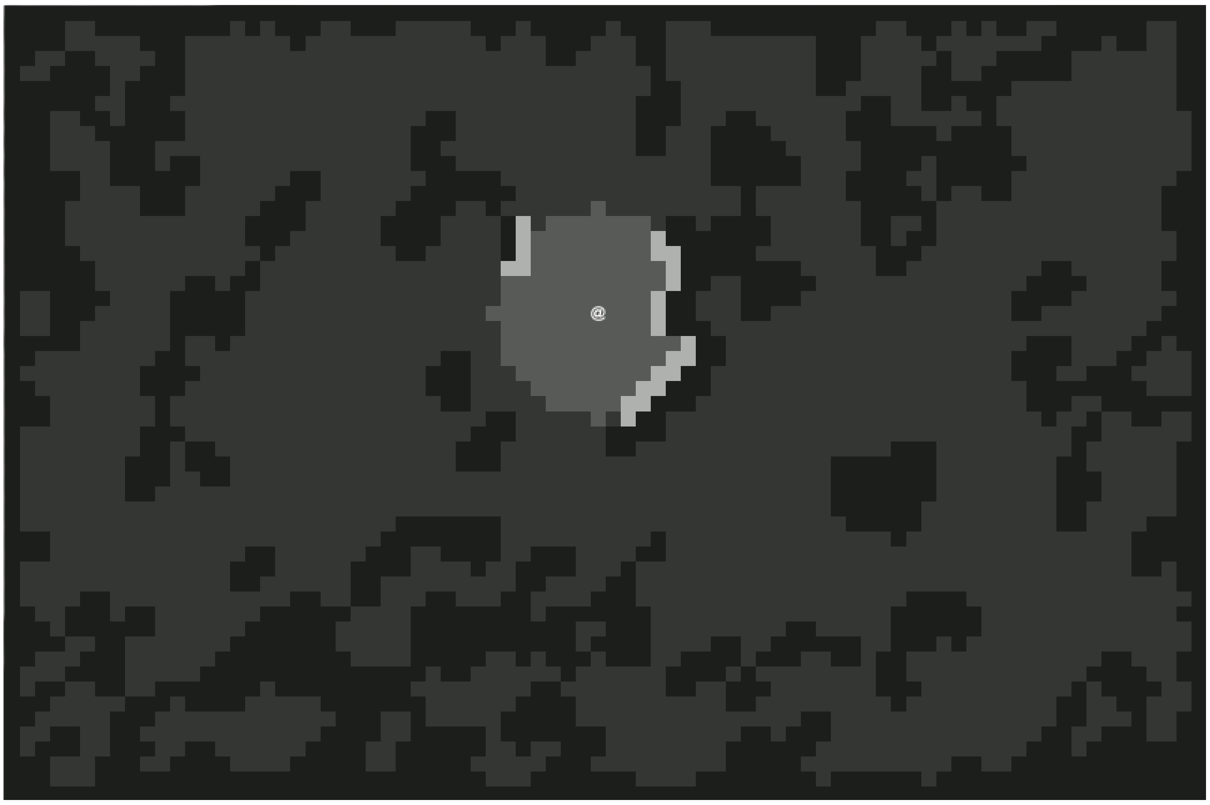
Source: Created by the author.

5.4 Artificial Intelligence

The first step when implementing artificial intelligence in a roguelike is to decide how the actors will move towards their goals (generally the player). This is called the problem of path-finding. In our prototype we experimented with two different approaches for it:

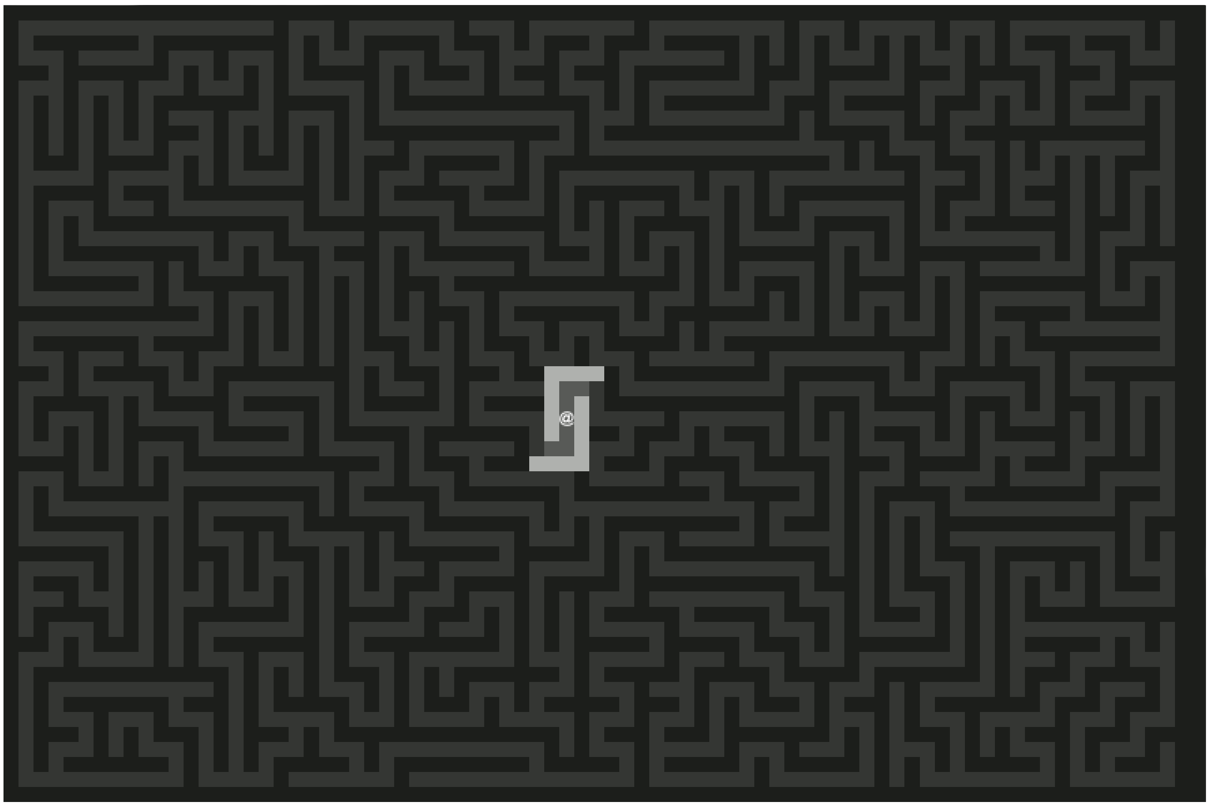
- Euclidean Chase: In this, the monster that entered the player's FOV calculated the shortest distance vector by using the euclidean distance method. After that, he would move to the next square in the player's direction. While this process is negligible in time cost, it has a drawback: if the square is already occupied (by a wall or another monster), the monster will stand still. Thus, we had to check if the square was occupied, and if so, move to an adjacent square in that general direction;
- Breadth-first search: The process of finding the target by breadth-first search is infallible, albeit costly. The search recursively looked at all the squares of a given distance before moving to distance + 1. Because of that, even a small number of concurrent actors can lead to a noticeable delay. Thus, some kind of countermeasures had to be implemented, such as calculating the path only for monsters in the player's FOV and considering only squares inside the player's FOV as eligible for search.

Figure 5.5 – Example of a dungeon level generated using the cellular automata technique.



Source: Created by the author.

Figure 5.6 – Example of a maze dungeon level generated using the depth-first search technique.



Source: Created by the author.

In regards to the actual behavior of the enemies, we started with stateless actors. That means the actor has no intrinsic information nor states in which he can base his behavior. On one side, this means actors of this kind will be less flexible, and harder to adapt to different contingencies in the gameplay. On the other, they are easy to implement and generally negligible in time cost. Our version consisted of a simple “look and chase” actor. In other words, once the monster sees the player, it will blindly charge towards him.

State-machine actors, on the other hand, possess intrinsic information, which can be innate or learned by experience. In our implementation, the information was embodied in the form of behavioral states, which would have transitions among them by certain kinds of input. Table 5.1 shows the state-based actor implemented for the final version of the prototype. The 'roaming_ai' means the monster will move randomly until the player appears, 'chasing_ai' is going towards and attacking the player, 'escaping_ai' is trying to move away from the player, 'sleeping_ai' is doing nothing until it wakes up and 'cornered_ai' is attacking the player because there aren't adjacent squares that aren't next to the player (no flight squares). Where there is a number after the state name, it represents the chance of the transition after a dice roll.

Table 5.1 – State-machine actor implemented for the game prototype of the thesis.

State	AI	LOW_HP	IN_FOV	OUT_OF_FOV	CORNERED
ROAMING	roaming_ai		CHASING		
CHASING	chasing_ai	ESCAPING		ROAMING:0.8, SLEEPING: 0.2	
ESCAPING	escaping_ai			ROAMING	CORNERED
SLEEPING	sleeping_ai	ESCAPING	ROAMING:0.2		
CORNERED	cornered_ai			ROAMING	

Source: Created by the author.

Finally, concerning instancing of in-game entities, the enemies we added to a level had their attributes randomized on an interval, based on which their experience points were calculated. In addition to that, they could be either stateless or state-based, which influences gain of experience points after defeating as well. Lastly, they would be selected from a list of 'races' taken from classic roguelikes, which came with specific names and symbols, as well as affecting the range of attributes to be randomized. Thus, for instance, a goblin would be generally weak and fast, whereas an orc would be strong and slow.

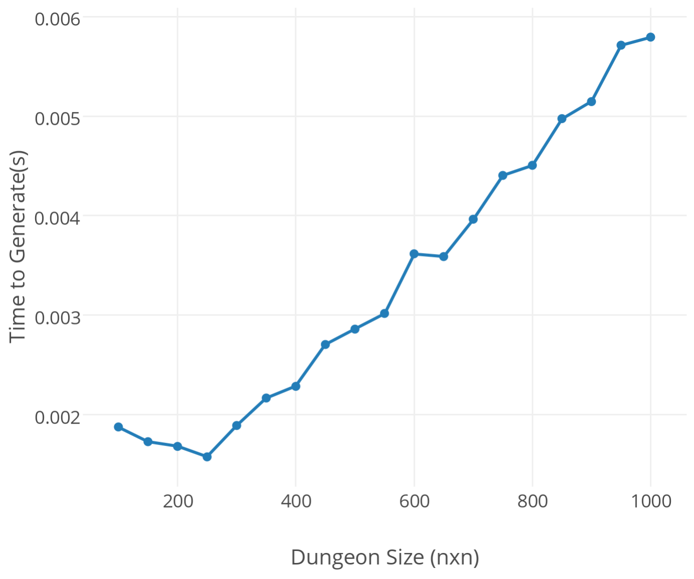
5.5 Benchmarking Results

When building the benchmarking structures for the implemented techniques, we had to define metrics and parameters. They are:

- For dungeon generation, the interval of dungeon sizes to be tested, as well as the increment between each iteration. We tried to find the best configuration possible so as to evidence the time growth of the techniques, and at the same time avoiding sizes too large for practical purposes. Because of that, we defined the dungeon sizes to range from 100×100 to 1000×1000, with increments of 50 on both dimensions. We also had to experiment with the number of times to repeat each dungeon size, so as to minimize eventual fluctuations generated by overloads in the operating system. We found out that averaging 20 times per dungeon size gave satisfactory results without taking impractical processing times.
- For the artificial intelligence techniques, the interval of number of concurrent actors to be tested was set to be 10-200, with increments of 10 actors per iteration. Just as in dungeon generation, we defined 20 repetitions for each number of actors.
- Finally, for each technique, we had to define their specific parameters, like minimum and maximum room size. We will describe these while talking about their respective techniques.

The first technique we tested was the basic iterative dungeon generator. We defined the maximum number of rooms to 1000. The reason for that is to keep it a fixed value throughout the benchmarks, and also so that the number of rooms would accompany the dungeon size, as this approach will only create extra rooms if there is space left for them. In addition to that, we set the room size range to 9-10 for both dimensions. Figure 5.7 shows a graph of the performance analysis made for the basic iterative approach. We can see from it that the interval from 100 to 250 can be considered constant in time (we believe slight downward tendency is due to fluctuation), which can be explained by the fact that for that range of dungeon size the algorithm finishes below (or right at) the accuracy time of the time-measuring function we used in python, which is time.clock(). After that, we see a linear increase in the growth function, which corroborates our expectations of this process being linear in relation to the number of rooms actually added to the dungeon, which size was increased in a linear way for the experiment. We can thus infer that this technique will work even for much larger sizes of dungeons.

Figure 5.7 – Dungeon size vs. time to generate using the basic iterative approach.

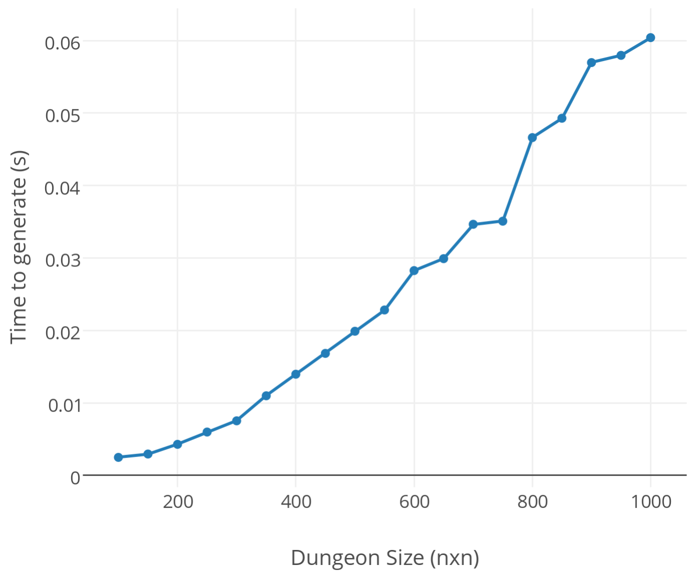


Source: Created by the author.

Regarding binary space partitioning, we defined the number of iterations to 4, which seemed to generate an ideal number and size of partitions after some testing. The room size ranged from 9-10 as in the basic iteration technique. Finally, we defined both the horizontal and the vertical ratios to be 1.0, which seemed to give satisfactory visual results after some testing. Figure 5.8 shows the performance analysis for the BSP Tree technique. Compared to the basic approach, we can see that the BSP technique is one order of magnitude more time consuming. This is due to the exponential nature of the technique, which is limited because of the small number of iterations we set.

Concerning cellular automata, we defined the chance of a square being a floor to be equal of it being a wall, which means on average half of the dungeon will start off as 'alive'. Also, we set the number of generations to 20, which may seem excessive, but after experimentation we have found that extra generations make the resulting cave-like structures more organic, and tends to remove more artifacts. Figure 5.9 shows a graph of the performance analysis for the cellular automata technique. We observe that the resulting growth function tends to a parabola (which would have been even more evident if larger sizes

Figure 5.8 – Dungeon size vs. time to generate using the BSP Tree technique.



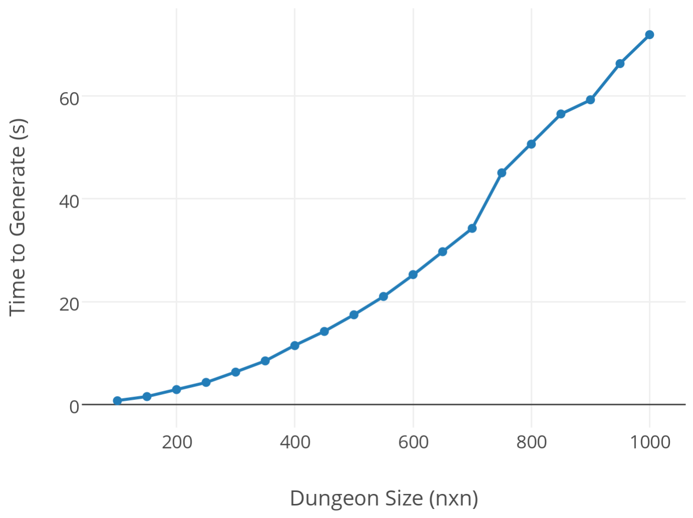
Source: Created by the author.

were tested, but we decided to test all techniques with the same size range, and a few experiments with larger sizes proved to be prohibitive in time cost). We also note that the algorithm is several orders of magnitude slower than the previous techniques. Both corroborate the fact that this technique grows linear in the total size of the dungeon, and thus quadratic in *n*, which is the side of the quadrilateral dungeons benchmarked. More specifically, it checks the neighbors of every square (including itself) on each iteration, which in total is $9n^2$ checks for every generation. Suggested optimizations to this process include: keeping track of unchanged cells, because if a cell and its neighbors haven't changed in the last step, they are guaranteed to not change in the current step; and also to use as storage one array and three line buffers, in which one buffer would be used to calculate the new states for a line and the other buffer would calculate for the next line, successively until the generation is simulated.

The last technique we tested for dungeons was the generation of mazes using depth-first search. Unfortunately, because we implemented it using a recursive procedure, none of the dungeon sizes used for the above-mentioned techniques could be applied to this technique, as the maximum function call stack size in python was exceeded. We have found out after experimentation that the largest dungeon size this approach could handle safely was 80×80.

After that, we benchmarked three artificial intelligence techniques for path-finding: euclidean chase, wort-case breadth-first search and breadth-first search with optimizations. An important parameter to be defined is the dungeon size for the tests, because it is a determi-

Figure 5.9 – Dungeon size vs. time to generate using cellular automata.



Source: Created by the author.

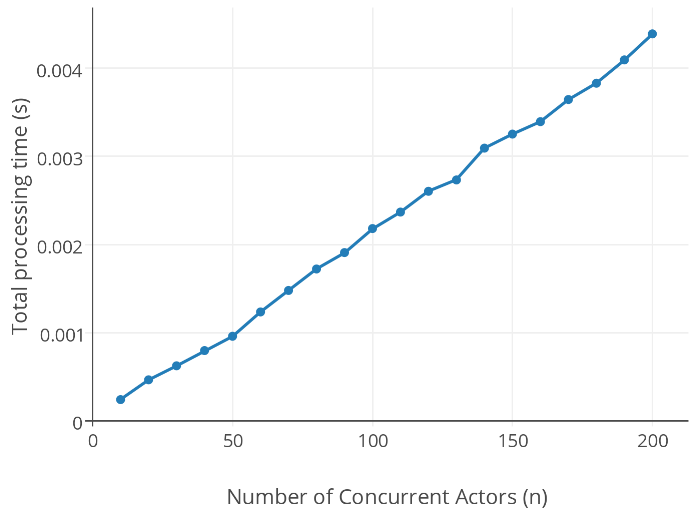
ning factor for search-based path-finders. We define the dungeon to be 200×200 forall tests, because this way there will be enough space for all quantities of concurrent actors. Also, it is important to note the reason why we choose to test artificial intelligence in terms of the number of concurrent actors, rather than other kinds of metrics: While it doesn't go far towards the understanding of the individual growth function of the techniques as would, for instance, varying the dungeon size and testing for a single actor, it helps us understand the practical limitations of such techniques in terms of concurrency, which is what we were aiming for in this case.

For the euclidean chase, we set the player's FOV to be unlimited, which meant all the monsters would see him at any time. Other than that, no other specific parameters needed to be set. Figure 5.10 shows the performance analysis for the euclidean chase technique. We can see from the graph that the total cost of computing all concurrent actors follows a linear growth function. That is what we expected, since each actor does calculates in constant time, thus the sum of their calculations is linear on the number of actors.

Finally, we talk about path-finding with breadth-first search. We analyze two versions of this technique:

- Worst-case scenario, where all monsters can see the player (unlimited FOV). Also, no optimization measures were added to this test, so that we can compare these results with the optimized technique.

Figure 5.10 – Number of concurrent A.I actors vs. total processing time in the euclidean chase.

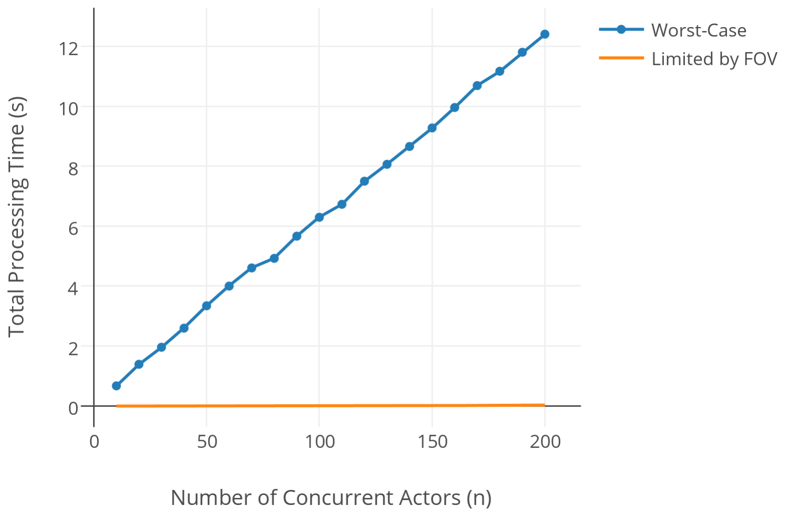


Source: Created by the author.

- Optimized by calculating paths only for monsters that are in the player's FOV. For this experiment, we set the FOV range to 9, which was also implemented in the final version of the prototype.

Figure 5.11 shows the performance analysis for both breadth-first search scenarios. We can see that, by simply limiting the path-finding search to monsters inside the player's FOV, we drastically reduce the amount of calculations needed. It is important to note that the seemingly constant optimized version appears so because of the small FOV range we set, which equates to only a few monsters searching concurrently. In fact, the worst-case scenario is basically the same technique, except that, with unlimited FOV range, and in a room big enough to comport all monsters, everyone of them will perform the search.

Figure 5.11 – Number of concurrent A.I actors vs. total processing time breadh-first search path-finding.



Source: Created by the author.

6 CONCLUDING REMARKS AND FUTURE WORK

In this work, we have gone through the process of designing and developing a game of the roguelike genre. We focused on two main design features, namely procedural dungeon generation and artificial intelligence for the enemies. First, we described several techniques for both features when applied to the genre. Then, we implemented some of these techniques and experimented with them regarding performance and overall quality, so that we could derive practical conclusions on their viability for the game prototype developed for this thesis.

Regarding procedural dungeon generation, we concluded that the techniques of basic iteration and BSP tree can be used to generate dungeons of any practical size due to their low processing cost. Concerning cellular automata, we found that for it to work on larger dungeon sizes in practical time, we must either reduce the number of iterations to a small number (below 5) or implement some of the optimization techniques suggested for the simulation process. In addition to that, we have observed that the use of depth-first search for maze generation using recursion is costly in space complexity, and may exceed the stack limit for function calls for larger dungeon sizes. Despite that, it seemed to generate dungeons in acceptable times for small to medium dungeon sizes.

With respect to artificial intelligence for enemies, we have discovered that the technique of path-finding using euclidean distance can be used in any number of concurrent actors. However, due to its disregard for obstacles, it is necessary to add countermeasures so that the enemies don't get stuck in corners or behind other enemies. Also for path-finding, we have shown that using breadth-first search without severely restricting the search space is impractical. However, after simply restricting the number of path-calculating actors to the ones inside the player's field of view, and setting the FOV range to a reasonably small number (like 9), the process becomes practically negligible in processing cost. Concerning the actual behavior of actors, we have argued that both stateless and state-machine techniques can be used without any relevant time overhead. However, we concluded that the use of state-machine actors provides richer gameplay experience and behavioral flexibility than that of stateless ones.

Regarding practical knowledge we gained in the development process, we can mention that:

- Designing a game requires a large amount of dedication and carefully thought design process, lest eventual necessary modifications become impossible to implement because of a too much coupled game engine;

- We understand the historical preference roguelike developers had for in-depth content over aesthetics, because simple graphical improvements take considerable time to implement;
- Learning a new topic using a new programming language has advantages, like porting only abstract knowledge in the form of algorithmic techniques, rather than language-specific structures. However it also has drawbacks, mainly the slower development process, due to also having to learn the specifics of the new language.

Additionally, we would like to point out some directions in which this work, and also the game which was prototyped, could be further explored:

- Implementing faster simulation techniques for the cellular automata;
- Implementing an iterative depth-first search for the maze generation technique;
- Exploring the rest of the techniques that were explained only theoretically;

Finally, we believe we have accomplished the end-goal of the work, which is to bring-closer the generally informal, ad hoc process that is game development, with algorithmic techniques that were extensively studied and formalized in the scientific environment. We hope that this work contributes towards raising awareness and interest to the fascinating game genre that is roguelike.

REFERENCES

ADAMS, T. **Dwarf Fortress**. Available from: http://www.bay12games.com/dwarves/features.html Accessed in June 2015.

ALMGREN, S. et al. **Astrogue: A Roguelike**: Using Procedural Content Generation for Levels and Plots in a Computer Game. 2014. Tese (Bacharelado em Ciência da Computação) – Bsc. Thesis, Chalmers University of Technology, University of Gothenburg, Göteborg, Sweden.

AMARI, N. Simulation Minus One Makes a Game. **Applications of Evolutionary Computing Lecture Notes in Computer Science**, Tübingen, Germany, v. 5484, p. 273-282, April 2009.

ANGBAND. **About Angband**. Available from: http://rephial.org/develop Accessed in June 2015.

BARTON, M. **The History of Computer Role-Playing Games Part 1: The Early Years**. Available from: http://www.gamasutra.com/view/feature/3623/the_history_of_computer_.php Accessed in June 2015.

BISKUP, T. **Ancient Domains of Mystery**. Available from: http://www.adom.de/adom/roguelike.php3 Accessed in June 2015.

BISKUP, T. **Resurrect ADOM Development.** Available from: https://www.indiegogo.com/projects/resurrect-adom-development Accessed in June 2015.

BROUWER, A. **Hack**. Available from: http://homepages.cwi.nl/~aeb/games/hack/hack.html Accessed in June 2015.

CARLISLE, P. A Framework for Emotional Digital Actors. In: DELOURA, M. (Ed.). **Game Programming Gems 8**. Boston, MA, USA: Course Technology, 2011. p. 312-322.

COVER, J. G. **The Creation of Narrative in Tabletop Role-Playing Games**. Jefferson, NC: McFarland & Company, 2010.

CRUZ, G. J. **Estimativa da qualidade de mapas procedurais para jogos do gênero roguelike**. 2014. 87f. Dissertação (Bacharelado em Engenharia de Software) – Faculdade UnB Gama, Universidade de Brasilia, Brasilia, June 2013.

DCSS. **Dungeon Crawl Stone Soup**. Available from: http://crawl.develz.org/wordpress/ Accessed in June 2015.

DELAUNAY, B. Sur la sphère vide. A la mémoire de Georges Voronoï. **Bulletin de l'Académie des Sciences de l'URSS**, Classe des sciences mathématiques et natureles, N. 6, p. 793-800, 1934.

DIABLO. **Diablo**. Available from: http://us.blizzard.com/ Accessed in June 2015.

DILLINGER, R. **Roguelike Intelligence – Intrinsic Information and State Machine AIs**. Available from: http://www.roguebasin.com/index.php?title=Roguelike_Intelligence_-_Intrinsic_Information_and_State_Machine_AIs Accessed in june 2015.

DILLINGER, R. **Roguelike Intelligence – Genetic Algorithms and Evolving State Machine AIs**. Available from: http://www.roguebasin.com/index.php?title=Roguelike_Intelligence_-_Genetic_Algorithms_and_Evolving_State_Machine_AIs Accessed in june 2015.

DORMANS, J. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. **Proceedings of the 2010 Workshop on Procedural Content Generation in Games**, n. 1, p.1:1 -1:8, 2010.

DOULL, A. **Winner of ASCII Dreams Roguelike of the Year 2012: T.o.M.E. 4**. Available from: http://roguelikedeveloper.blogspot.com/2013/01/winner-of-ascii-dreams-roguelike-of.html Accessed in June 2015.

DOULL, A. **The Death of the Level Designer: Procedural Content Generation in Games**. Available from: http://roguelikedeveloper.blogspot.com.br/2008/01/death-of-level-designer-procedural.html Accessed in June 2015.

EGGES, A.; KSHIRSAGAR, S.; MAGNENATTHALMANN, N. Generic personality and emotion simulation for conversational agents. **Computer Animation and Virtual Worlds**, v.15, n. 1, p. 1-13, march 2004.

ESKERDA. **Dungeon Generation Using BSP Trees**. Available from: http://eskerda.com/bsp-dungeon-generation/ Accessed in June 2015.

GARDA, M. Neo-rogue and the essence of roguelikeness. **Homo Ludens**, v. 1, n. 5, p. 59-72, Poland, 2013.

GARDNER, M. Mathematical Games – The fantastic combinations of John Conway's new solitaire game “life”. **Scientific American 223**. p. 120-123, Oct. 1970.

GRABINER, D. **Moria**. Avaliable from: http://www-math.bgsu.edu/~grabine/moria.html Accessed in June 2015.

IBÁÑEZ, R. **Creación de videojuego Rogue-Like**. 2014. Tese (Bacharelado em Engenharia Informática) – Bsc. Thesis, Universidad de Alicante, Spain.

IBM. **Deep Blue**. Available from: http://www.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/ Accessed in July 2015.

IGN. **Path of Exile: Passive Skills**. Available from: http://www.ign.com/wikis/path-of-exile/Passive_Skills Accessed in June 2015.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. **Proceedings of the American Mathematical Society 7**, p. 48-50, 1956.

LAIT, J. **About POWDER**. Available from: http://www.zincland.com/powder/index.php?pagename=about Accessed in June 2015.

MANGBAND. **MAngband**. Available from: http://mangband.org/Main/WhatIsMAngband Accessed in June 2015.

MCRAE, R.; COSTA, P. T. Jr. **Toward a New Generation of Personality Theories**: Theoretical Contexts for the Five-Factor Model. New York, USA: Guilford Press, 1996.

MIDDLEEARTH. **Middle-earth: Shadow of Mordor**. Available from: https://www.shadowofmordor.com/ Accessed in June 2015.

NETHACK. **NetHack**. Available from: http://www.nethack.org/ Accessed in June 2015.

ORTONY, A..; CLORE, G. L.; COLLINS, A. **The cognitive structure of emotions**. Cambridge, England: Cambridge University Press, 1988.

POE. **Path of Exile**. Available from: https://www.pathofexile.com/ Accessed in June 2015.

PETER. **Manapool Guide to Roguelikes**. Available from: http://www.manapool.co.uk/mana-pool-guide-to-roguelikes/ Accessed in June 2015.

PRIM, R. C. Shortest connection networks and some generalizations. **Bell System Technical Journal 36**, n. 6, p. 1389-1401, 1957.

RABIN, S. A* Aesthetic Optimizations. In: DELOURA, M. (Ed.). **Best of Game Programming Gems**. Boston, MA, USA: Course Technology, 2008. p. 247-254.

RAYTHEON, S. W. Flocking: A Simple Technique for Simulating Group Behavior. In: DELOURA, M. (Ed.). **Best of Game Programming Gems**. Boston, MA, USA: Course Technology, 2008. p. 297-310.

ROGUEBASIN. **Berlin Interpretation**. Available from: http://www.roguebasin.com/index.php?title=Berlin_Interpretation Accessed in June 2015.

ROGUEBASIN. **What a Roguelike is**. Available from: http://www.roguebasin.com/index.php?title=What_a_roguelike_is Accessed in June 2015.

ROGUECENTRAL. **Pyromancer**. Available from: http://roguecentral.org/doryen/games/pyromancer/ Accessed in June 2015.

ROGUECENTRAL. **libtcod**. Available from: http://roguecentral.org/doryen/libtcod/ Accessed in June 2015.

ROGUETEMPLE. **What is a Roguelike**. Available from: http://www.roguetemple.com/roguelike-definition/ Accessed in June 2015.

ROLLINGS, A.; ADAMS, E. **Andrew Rollings and Ernest Adams on Game Design**. Indianapolis: New Riders Publishing. 2003.

STERREN, W. van der. Tactical Path-Finding with A*. In: DELOURA, M. (Ed.). **Best of Game Programming Gems**. Boston, MA, USA: Course Technology, 2008. p. 271-283.

VALTCHANOV, V., Brown, J. A. Evolving Dungeon Crawler levels with relative placement. **Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering**, p. 27-35, New York, USA, 2012.

VOLCHENKOV, D.; BLANCHARD, P. Fair and biased random walks on undirected graphs and related entropies. In: DEHMER, M.; EMMERT-STREIB, F. MEHLER, A. (Eds.). **Towards an Information Theory of Complex Networks**. Boston, Basel: Birkhäuser, 2011. p. 347-364.

WICHMANN, G. R. **A Brief History of "Rogue"**. Available from: http://www.wichman.org/roguehistory.html Accessed in June 2015.

WIKIPEDIA. **Maze Generation Algorithm**. Available from: https://en.wikipedia.org/wiki/Maze_generation_algorithm Accessed in June 2015.

WOJTOWICZ, M. **Cellular Automata rules lexicon**. Available from: http://www.mirekw.com/ca/rullex_life.html#Maze Accessed in June 2015.