CHALMERS | GÖTEBORGS UNIVERSITET

Astrogue: A Roguelike

Using Procedural Content Generation for Levels and Plots in a Computer Game

Bachelor of Science Thesis in Computer Science

SIMON ALMGREN
LEO ANTTILA
DAVID OSKARSSON
FABIAN SÖRENSSON
SAMUEL TIENSUU

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

Astrogue: A Roguelike
Using Procedural Content Generation for Levels and Plots in a Computer Game

SIMON W. ALMGREN
LEO M. ANTTILA
C. DAVID OSKARSSON
J. FABIAN Z. SÖRENSSON
SAMUEL A. M. TIENSUU

Examiner: B. ARNE LINDE

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

The cover picture is the logo used for the game Astrogue. The logo has been designed by group member Simon Almgren for this project.

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Abstract

The aim of this thesis is to explain the process of creating a computer game in the *roguelike* genre, where *procedural content generation* is key. The case study game discussed here is called *Astrogue*, a simple tile-based 2D game with procedurally generated levels and plots. Astrogue is written in Java and uses the library *Lightweight Java Game Library* for graphics and input.

Even when creating a simple 2D game, there is a need for good code architecture. For this, we have chosen to use a design pattern called *entity-component-system*, a pattern that favours composition over inheritance. Besides smart code architecture, other interesting programming problems had to be solved. Systems for controlling game logic have been implemented, including for example an algorithm for shadowcasting and a simple enemy AI.

For both level and plot generation, different algorithms and approaches were studied and evaluated, but only some were used. Two different level generating algorithms have been implemented. One creates classical roguelike dungeons, with multiple rooms connected using Delaunay triangulation and Kruskal's algorithm. The other one creates cavelike levels using cellular automata. An algorithm for generating names from lists of old ones, using Markov chains, is part of the game. Finally, the game generates a plot by creating characters and letting them interact in an AI simulation.

Acknowledgements

We would like to thank the roguelike community for all its great games and documentation. We would also like to thank the group rooms NC7 and 3211 for (almost) always being available to book. Furthermore, we would like to thank Oleg Khaikin for proof-reading the paper. Finally, we would like to thank Ulf Assarsson for being our helpful supervisor and Johan Grönvall for being part of our project group for the first few weeks.

Wordlist

AI – Artificial Intelligence
Astrogue – The title for the game we have created
AstroPG – Short for *Astrogue Plot Generator*, the name of our plot generator
ECS – Entity-Component-System
FSM – Finite State Machine
GA – Genetic Algorithm
LWJGL – Lightweight Java Game Library
MST – Minimum Spanning Tree
PCG – Procedural Content Generation
SAPs – Short for *scenes*, *actors* and *props*
Seed – A code that allows the level and plot to be played again

Contents

Chapter 1

Introduction

This thesis is about the creation of a computer game with procedurally generated levels and plots. This introductory chapter explains the project's purpose and the background of relevant topics, such as *procedural content generation*. A problem statement is given, together with limitations to the project. We also explain method choices and a project overview. Finally, an outline for the entire paper is presented.

1.1 Purpose

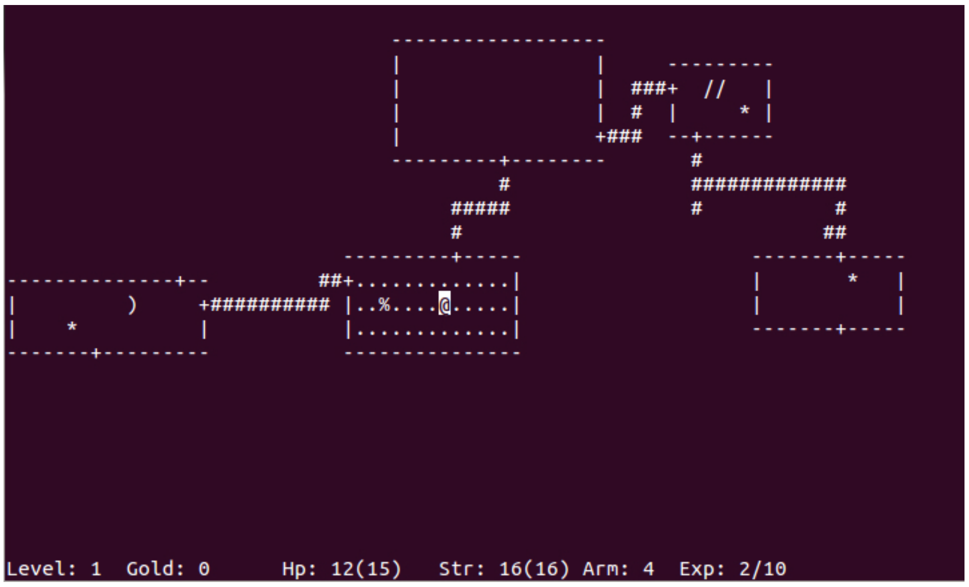
The purpose of our project has been to create a game within the *roguelike* genre (described in section 1.2.1 below), and in the process of creating it, examine and implement algorithms that could be used in our game. We have also examined different ways to implement a game engine. A large focus has been put on *procedural content generation* (PCG) algorithms, which is something that is typical for roguelikes. PCG is commonly used to create graphics in movies (Watson et al. 2008), sound effects, level design for games, and even storylines, to mention a few areas (Togelius et al. 2011). Our goal has been to use PCG to create a game that both is fun to play and creates a new experience each playthrough.

1.2 Background

This section explains the origin and definition of the roguelike genre. The section also explains procedural content generation, which is a defining element of roguelikes, and especially procedural generation of levels and plot, which are topics that have been the main focus of this project.

1.2.1 The roguelike genre

The roguelike genre started with the video game *Rogue* (Toy et al. 1980). Rogue is a roleplaying game with randomly generated dungeons. A screenshot from Rogue can be seen in Figure 1.1. Since then, there have been many games that have used elements from Rogue (hence the term roguelike). The most common elements are procedurally generated content, permanent death, and character progression. There is a more strict definition of what determines if a game is a roguelike or not called *The Berlin Interpretation*, a definition formulated by the attendees of the *International Roguelike Development Conference 2008* (*The Berlin Interpretation* 2008). The Berlin Interpretation lists properties such as turn-based game mechanics, dungeon-like levels, *hack'n'slash* gameplay and ASCII graphics. However, this definition has received criticism for being too strict and prohibitive for further development of the roguelike genre (Grey 2013). In more recent years several games have been inspired by Rogue and roguelikes, which often incorporate some of the elements from the genre. *Diablo* (Blizzard Entertainment, Inc. 1996), *Minecraft* (Mojang AB 2011) and *Spelunky* (Mossmouth 2008) are examples of newer games that use these features; however, these games are not often called roguelikes because they often fall into another category better.



```
Level: 1  Gold: 0      Hp: 12(15)    Str: 16(16) Arm: 4   Exp: 2/10
```

Figure 1.1: A screenshot from the classic game Rogue, the game that started the roguelike genre.

1.2.2 Procedural content generation

Procedural content generation (PCG) is something that is commonly used in movies and games (Valtchanov & Brown 2012), especially, in roguelike games. The term refers to content being stochastically generated instead of created by hand. In movies, PCG is often used to generate environments that look interesting without having to create every detail by hand (Watson et al. 2008), using software such as Terragen (Planetside software 2009). In games, PCG is used in various genres, most commonly for level generation, but also to generate content such as textures, characters, environmental objects, and sounds (Hendrikx et al. 2013).

Procedural content generation is very useful since it means development time can be spent on other aspects of the game rather than e.g. creating every level or texture manually, which is often very time-consuming if one wants the content to be varied (Togelius, Yannakakis, Stanley & Browne 2010). Since the content is generated instead of stored in memory, the space required for a game using PCG will be much smaller than if the content had been pre-defined.

Estimates of the cost of content design in games have been up to 40% of a game's budget (Hendrikx et al. 2013). Currently, big titles are not always able to keep the new content creation at a rate the game communities would like. To be able to produce a lot of manually designed content at a fast rate, the game companies need a large amount of qualified personnel. Making effective use of PCG would decrease the cost without losing too much quality in the generated content.

1.2.3 Level generation

Procedurally generating new levels for each playthrough helps games to not get repetitive as fast. Avoiding repetitiveness with manually created levels would require a massive amount of pre-designed levels. When utilising procedural level generation, usually only the level generator itself has to be written, which alone can generate a large enough number of new and unique levels. Some examples of games that make use of procedural level generation are *Gran Turismo 5* (Sony Computer Entertainment Inc. 2013) with its track generator, the *Diablo* series with procedurally generated maps and dungeons (Blizzard Entertainment, Inc. 2012), and games within the roguelike genre where generally everyone uses random map generation (Togelius, Preuss & Yannakakis 2010). Level generation is important because it gives an almost endless supply of levels and therefore limitless replayability. It also reduces memory consumption and saves development time and costs (Togelius et al. 2011).

1.2.4 Plot generation

When reading a book or watching a movie, the question *"What will happen next?"* will frequently run through the consumer's mind (Roth et al. 2009). When playing a video game, the same question may often be asked, but the player may also be curious about how the game and its story reacts to the player's actions. Curiosity and suspense are driving factors in our enjoyment of older media, such as books and movies, as well as video games. The progression of the plot is central to our curiosity, while emotional attachment to characters drives suspense. Hence, the plot is an important part in the enjoyment of a video game. In a short game that is designed to be played multiple times, making the plot varied from playthrough to playthrough is of special importance to make the game interesting.

Procedural plot generation is not a typical aspect of commercial games, but rather a research topic on its own. The field mostly known for procedural plot generation is *interactive storytelling*, where plot generators create stories that adjust to the user's interactions (Crawford 2012). We believe this research should be applied more in commercial products. Just as with other types of content, producing interesting stories is expensive. Today, game companies and the like often employ more story writers and content providers than technicians and programmers (Díaz-Agudo et al. 2004). If stories instead could be generated procedurally, costs could be reduced.

1.3 Problem statement

What must be implemented to create a roguelike game with procedurally generated levels and plots? This is the general question that we try to answer with this thesis. We divide this problem into multiple subproblems, which are described in the subsections below. The results are explained in chapters 2–5 and each chapter corresponds to a subsection below.

1.3.1 Creating a game engine

When creating a game from scratch a game engine is required. We wanted to create our own and therefore one of the goals for this project has been to create our own game engine that can be used for roguelikes. We have aimed to examine how we can produce a game engine that is easy to use during the development phase, and also easily re-used and further developed in the future. It is required for the engine to be able to support different game modes and manage input.

1.3.2 Designing a game

To be able to tie together the other parts of the project, and test our game engine and content generators, a game had to be designed. When designing a game several problems will be encountered along the way. Many different systems have to be implemented to handle the game logic. What systems will be needed for a roguelike and how they should be implemented is a non-obvious problem.

A requirement for the game is that we want it to follow the constraints directed by the roguelike genre; even if these constraints are not very strict the game should follow them as closely as possible.

1.3.3 Generating playable and varied levels

One of the goals of this project has been to examine how to make use of procedural content generation (PCG) in order to create levels for a game. It is important to know what to generate, be it a full world or only levels with finite space. Should it be terrain in three-dimensonal surroundings, two-dimensional levels from a top-down perspective or from a side view? Enemies and other entities such as treasures and doors should be placed fittingly within the created levels. Above all, the levels must be playable; to be considered playable, every area needs to be reachable for the player.

The generators should be able to generate an infinite amount of levels, with a few very different characteristics. We want both levels that could be the inside of buildings and levels that are reminiscent of caves. However, generating levels within even more environments would be appealing. To successfully create numerous variants of environments, the algorithms should be general enough so that developers could adjust levels created by changing input parameters.

Another requirement we have for our level generators is that they need to accept a seed. The use of the seed acts as a key for the level. If the same seed is used on two different computers, it will generate identical levels on both machines, allowing players to either replay levels they found extra fun, or share them with their friends.

1.3.4 Generating plots

To make every playthrough feel fresh and interesting, one aim for this project has been to make a generator that procedurally generates plots. With *plot*, we in this thesis refer to the story on a higher level: what characters, places, and key items there are in the story and what the characters do to make the story advance. Since most action games usually have a simple story, where the main character carries out quests like fetching an item or defeating an enemy, the aim for the plot generator has been to create plots in a similar style.

Even simple plots should be varied and interesting, and they should fit the game. To achieve this, some requirements are put on the generated plots. They should not be too short, or they will not be interesting. They should not be too long, or they would become monotonous and would not fit the game, which is to be replayed many times and one playthrough should not be too time-consuming. They also need to be coherent, or some parts may be perceived as having a lack of meaning. To give the plots even more diversity, all characters and objects should have generated names.

1.4 Limitations

Since PCG is such a broad subject, we originally wanted to use it to generate as much as possible for the game, such as textures, sound effects, music and enemy behaviour. Due to time constraints, the focus of the PCG algorithms has instead been on level design and plot generation.

Instead of generating textures, we decided to create our own sprites for the game. These are all in simple 2D graphics. No sound support has been added, so no sound has been recorded for the game.

One early aim for the plot generation was to be able to procedurally generate the texts that convey the story. This goal was soon abandoned. Instead, the plot generator aims to create overarching plotlines and randomised names.

Artificial Intelligence (AI) is something that is required for a game with enemies to be fun. Since it is a very broad field and complex enough that a whole bachelor thesis could be written about it alone, we have decided to make the AI relatively simple to be able to focus on the PCG algorithms instead.

1.5 Project overview

In this section, we give a short overview of the project and its different parts.

Multiple components are required to create a game and to solve the mentioned problems, both in terms of code and content. The code part is especially true for a roguelike. When creating Astrogue, we spent time coming up with game ideas, designing content, and writing code. While the game design has been an important part of the project, focus has been put on programming the systems behind the game.

In the core of Astrogue is its *game engine*. The game engine is the main framework for the application. It initialises and keeps track of all other systems that make up the game, and also holds the main loop. To avoid poorly written code in the long run, we have also implemented a design pattern called entity-component-system as a central part of our code architecture.

Besides the main game engine, multiple classes have been written for tasks such as rendering graphics and controlling enemy AI. Our most interesting systems, however, are the level generators and the plot generator. Two level generators, generating different types of levels, are part of the game. Our plot generator, called *AstroPG*, is on the other hand written as a module separate from Astrogue, which can be run both as a part of the game and on its own. A diagram illustrating the project structure can be seen in Figure 1.2.

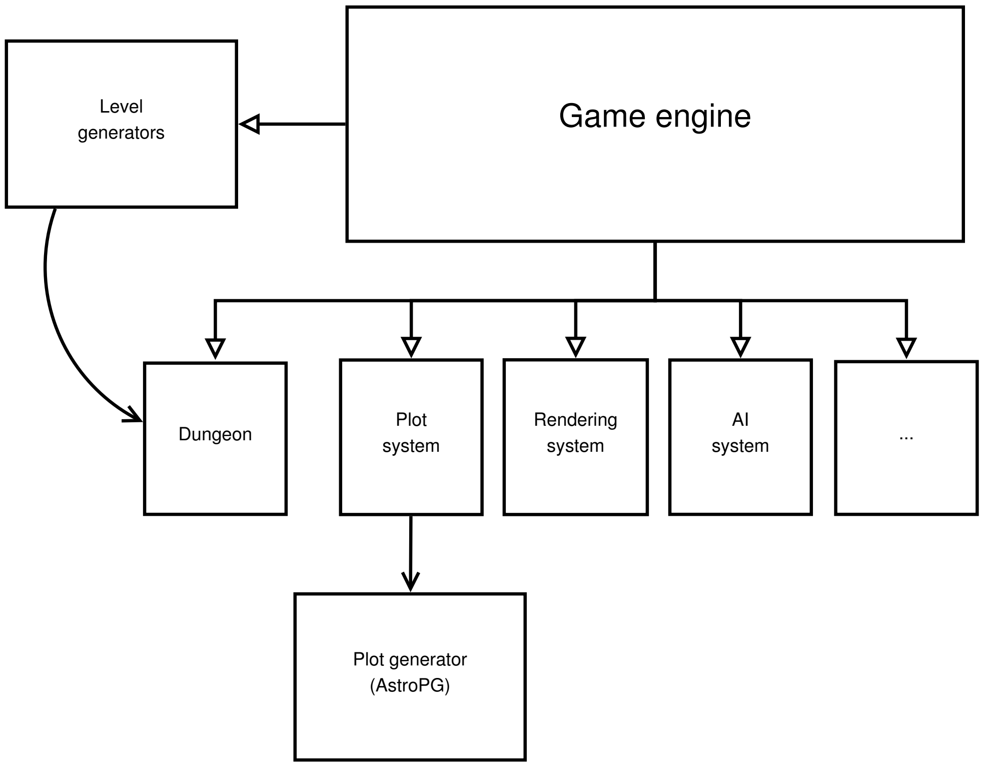


Figure 1.2: A diagram illustrating the structure of our project. At the core of Astrogue is the *game engine*, which keeps track of multiple systems for tasks such as rendering and enemy AI. The game engine also holds a plot system that communicates with our plot generator *AstroPG*, and a dungeon that is generated by one of our level generators.

1.6 Method

In this section, we describe the different methodological choices we have made, such as planning our work in a Scrum-like manner and using Java as our programming language.

1.6.1 Agile development

During the project a variation of the agile development strategy Scrum was used. As suggested by Scrum, the project was divided into several iterations of complete parts that lasted one week each. In the beginning only the game engine was built, but later on more functionality was added with each iteration. The team was split into several smaller groups each week, each group working on a different aspect of the game. Using an efficient project management methodology unquestionably decreased the time needed for the project and enabled us to implement more functionality into the game.

1.6.2 Programming language and framework

This project was programmed in Java for multiple reasons. Java has all the functionality needed and, since it uses a virtual machine, works across all different platforms. While Java may not be as efficient as for example C++ (Prechelt et al. 1999), it is good enough for a simple 2D game like Astrogue.

When developing a game, it is often convenient to use a pre-written framework and not write everything from scratch by yourself. There is a variety of frameworks for simple 2D graphics to choose from. This project needed something very lightweight, since most textures that have to be rendered are small in size. Slick2D was chosen which essentially is a wrapper for the Lightweight Java Game Library (LWJGL). This is a very easily managed and lightweight framework, which takes care of the rendering on the screen, handling of sprites and on top of that all input from the keyboard and mouse. This choice was made because Slick2D is one of the most prominent frameworks on the market and advanced enough for this project (Bevilacqua 2013).

1.6.3 Version control

When developing a larger project, it is advantageous to use a version control system to manage the code and facilitate group work. A number of version control systems exist, but for this project we chose to work with Git. Git works on all the different platforms we have used and is easy to manage. GitHub was chosen to host the code, since it allows for Git access, has a good web interface to browse the code, and makes it easy to release the code as open source.

We decided to release Astrogue as open source, since most other games in the roguelike genre are released this way. We also believe that the source code for a research project like this should be free for everyone, since it can help the understanding of the research.

1.7 Outline

This report covers the implementation of Astrogue and our content generators. Chapters 2–5 all cover previous work and our results, ending with a discussion. In chapter 2 we discuss how the engine and backend for the game have been implemented. In chapter 3 different aspects of game design are discussed along with our implementation of the game logic. Chapters 4 and 5 cover the generated content: chapter 4 covers generated levels and chapter 5 covers plot generation. Chapter 6 describes the final product and summarises the results from earlier chapters. The chapter also shortly discusses the results and what we have learned, and gives an evaluation of our method choices. Finally, chapter 7 is a conclusion which summarises the whole project.

Chapter 2

Game engine

This chapter will cover the design decisions made when designing the game engine for Astrogue. The chapter will cover commonly used code architectures for creating game engines, followed by a brief description on how games typically handle states and input. Finally, it will cover the implementation of the game engine and end with a discussion of our implementation's advantages and disadvantages.

2.1 Previous work

In this section we will go through the background for the key parts of the engine. First, we will discuss the code architecture, which is important for building a well-designed engine. The section will feature common code architectures for game design, as well as discussing their advantages and disadvantages. Secondly, we will discuss how the game can handle being in several different states. Finally, we will go through input management and how that can be handled.

2.1.1 Code architecture

One of the challenges with creating a game is setting up the code architecture, choosing which design patterns should be used and designing the hierarchy of classes and objects. Since we wanted to implement our own engine for Astrogue, we were first required to examine existing architectures. A design pattern for a smaller project is not always necessary, but for larger projects it becomes a necessity to help avoid having to refactor the code too often (Croft 2004), which is very time consuming. Design patterns should preferably be chosen before the implementation begins. By choosing a code architecture early and making sure it is easy to work with, we can help ensure an easier integration with the features of the game. This is the case for the plot and level generation that has been developed in parallel with the game engine and will be discussed in later chapters. This would allow for the game to have additional functionality added with minimal changes to the game engine (Folmer 2007). Additionally, a design pattern that allows for scalability is preferable. It allows the game to be played from a very early stage and for more functionality to be added during the development phase with minimal interference from other systems. Even with a large planning phase it can be hard to predict how the code should be structured, therefore scalability is very important.

One of the drawbacks of using the standard inheritance model that is otherwise commonly used for programming in Java is that Java does not support multiple class inheritance. Implementing a special enemy type could for instance require implementation of Enemy and Renderable, where it would receive its rendering functions from renderable and its movement and AI logic from Enemy, which would not be possible. A solution to this would be to use interfaces that would require the new enemy type to implement the methods required for Enemy and Renderable, but this could very easily result in code duplication and could require changes done to one of these methods to be changed in each implementation (Croft 2004). Another way to manage inheritance is to use base classes that create an inheritance chain as can be seen in Figure 2.1. In the figure, the class Enemy inherits from Renderable and each specific enemy type inherits from Enemy, as can be seen with the Ghost and Snake classes. This works well until we decide that the Ghost should always be invisible. This might force us to create another base class such as *Invisible Enemy* instead of enemy. This could work well for special monsters, but it could end up being hard to manage in the long run (Lord 2012).

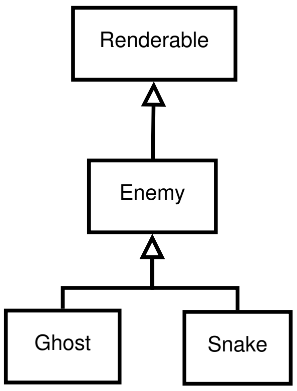


Figure 2.1: An example showing how an inheritance chain can be constructed for enemy classes.

Another design pattern is to use a component based model. In an entity-component-system model, the code is separated into three parts: entities, components, and systems (West 2007). The entities make up most elements in the game, such as the player character, the enemies, and menu buttons. They contain no data or logic on their own, they only contain a set of components which make up the entity. The components contain all the data, for instance a health component can keep track of the maximum health and current health of the player, and a position component can keep track of its X- and Y-coordinates. Like the entities, they contain no logic of their own. All the logic is handled by the systems, which have entities registered to them and operate on the registered entities each time the systems' update method is called (usually each time the game loop iterates).

Component systems have become increasingly popular in game development since they allow for easier development where components and systems can be implemented when needed or integrated from side projects as they are completed (Folmer 2007). A system change generally does not have much effect on unrelated systems and can therefore be developed independently. Entities can then be added or changed by changing the components that they contain. A player entity and an enemy entity is of the same class, albeit with a different set of components. This allows for the development process to scale really well and allows for features to easily be added by changing or adding a system, instead of having to modify code in several classes. This simplicity is further expanded on by not having the components know of any other components or systems (Doherty 2003). The downside is that all systems need to be generic enough to be able to work on all entities that have the right components (Gregory 2009). Due to the growing use of component based architecture in game development and its scalability features, we decided to use the entity-component-system pattern when developing our game.

2.1.2 Managing game states

The game engine is required to support different states since the game requires several game modes. Each of these states requires different systems to run. Running the game engine as a finite state machine (FSM) is a helpful way to achieve this and at the same time reduces the complexity of the code. This prevents large sets of if-cases from being created to manage special cases that can occur. By using an FSM design from the beginning, the game only has a set number of states that it can be in, allowing for a reduced code complexity. This in turn makes the code easier to maintain and make changes to (Brownlow 2004).

2.1.3 Input management

While ECS does not touch input in its implementation, there are still several ways to manage it, all with their own advantages and drawbacks. Since ECS was used, the chosen way to handle input had to at least cooperate well with it. The simplest way would be to have each system get the input via the tools we get from the Lightweight Java Game Library (LWJGL) API (LWJGL 2009*a*) (LWJGL 2009*b*). These tools enable access to the keyboard and mouse at all times. The apparent drawback to handle input this way is the fact that filtering of the input would be required in each system. Another way is to implement the subject/observer pattern (Hannemann & Kiczales 2002). This would allow the systems that want input to subscribe and get notifications from a special input system that acts as the subject when a key is pressed or the mouse is clicked. This would still require parsing of the notifications sent, but the systems would instead receive notifications with instructions on what to do rather than the raw key presses. By using the subject/observer pattern, data polling can be minimised but requires a new system for input management that acts as the subject. Due to to the simplicity of the observer pattern, we have chosen to use this to implement the input manager.

2.2 Results

In this section, we will first cover our interpretation and implementation of an entity-component-system framework which is used to manage the assets in the game engine. Secondly, we will describe how game states are handled by the engine and how state changes are managed. These two subjects are key to understanding how the game engine works. Lastly, we will go through our implementation for input management, which describes how all the input is managed in the game which is essential for game interaction.

2.2.1 Entity-component-system

In the implemented entity-component-system architecture the engine plays a significant role. A possible way to store the entities is to register them in a global list that can be checked by each system when needed. To avoid having the systems loop through this list for each run and check if anything should be done with the entity components, the engine instead registers the entities beforehand with each relevant system based on the component key. This allows the systems to only iterate over relevant entities instead of all possible game entities. Storing references to entities in each system is more memory consuming than using a global list, but speeds up the game loop by reducing the number of entities that should be checked. Both techniques were attempted early on, but storing entities in each relevant system was preferred due to the performance gain.

Component keys are binary flags stored as bits in an integer. Each component has a unique flag that has been defined in the engine by the component author. When a new component is added to an entity the component key is changed by using the logical OR function with the already existing component key and the new component's flag. For instance in Figure 2.2 an entity's component key is shown with the component Attributes, which has the component flag of 0001_2, and a health component added to it, which has the component flag of 0010_2, the resulting component key would be 0011_2. We can see how these component flags adds new parts to the key.

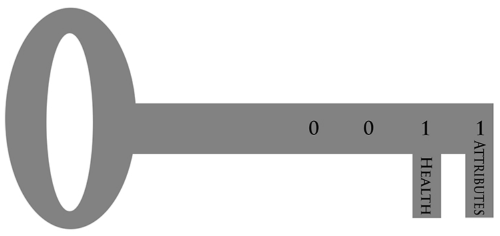


Figure 2.2: An abstraction showing how a component key is constructed for an entity with two components added to it.

To easily be able to define new component flags and minimise the risk of two component flags being the same, an integer is used to keep track of how many components has been added to the system. This integer is then used to bitshift a 1 once for each component that has been added, making sure that each component gets a unique component flag as can be seen in Code 2.1.

Code 2.1: Code example showing how an integer can be used to set the component key by bitshifting and incrementing the integer.

```
private static long componentID = 0;
public static final long CompAttribute = 1 << componentID++;
public static final long CompHealth = 1 << componentID++;
public static final long CompInput = 1 << componentID++;
```

This would give the Attributes component a flag of 0001_2, the Health component a flag of 0010_2 and the Input component would have the flag 0100_2.

The engine keeps track of these keys and they are publicly available for every other system, so that each system can use the flags to identify which components an entity contains. When a new system is defined the author of the system can make use of these component flags and define which entities that a system is interested in using. When a new entity is registered to the engine, it checks the component key and if it contains the required flags, registers it to the system. Similarly, when an entity is removed the same technique can be used to see which systems the entity has been registered to and remove it from those systems.

Each component implements an interface. The interface serves mainly to identify components as a component and only requires them to implement a clone method so each entity can be cloned. How to store and retrieve the components from an entity was something that took some time to figure out. The solution proved to be very easy: the components are stored in using a hashmap, with the component class name as a key and the component itself as the value. This allows for the components to be accessed by requesting them using the component class name and was made possible since each component implements the same component interface, allowing them to be stored in the same hashmap. Since the only aspect separating entities from each other is the components they hold, only one standard entity class has been created and each enemy, stack of gold or even the player is simply an instance of that entity class with different components added to it.

Each system implements an interface designed to make sure that each system is able to add and remove components as well as having an update method that runs during each iteration of the game loop. The entities that have been registered in the systems only contain the components, and no data of their own. Furthermore, the components only contain data which the systems make use of. Each system performs different actions with its registered entities, such as rendering the entities' sprites or moving them through the game world based on their input. The systems are required to work on all the components that have been registered to it. This requires the systems to be written in a general way that is not specific to a given entity, but must behave the same way for each entity they operate on. This has proved troublesome at times but as a result the code is easier to maintain and further develop due to the lack of special cases that can clutter up the code.

2.2.2 States

The engine is designed and implemented to use different states. It keeps track of the current state in a static enumerable variable, allowing other systems to access it. The game can have four different states: Dungeon, MainMenu, Overworld and GameOver. The Dungeon state is set when the player is in the dungeon crawl mode, MainMenu is set when the player is in the menu of the game, Overworld is set when the player is in the overworld and lastly the GameOver state is set when the player lost.

The game loop is constantly running the update method for each relevant system. Depending on the states, different systems are considered relevant. Several systems are updating regardless of the state the game is in, but the internal functions might differ based on state. Therefore, it is important that the state flag is accessible for all systems. The rendering system is one clear case where this is important. Regardless of state, the rendering system needs to render every entity that is registered within the system such as buttons, players and enemies. However, when the Overworld state is set, the overworld menu should be rendered; if the Dungeon state is set, a minimap and character status should be rendered on the side of the screen; if the MainMenu state is set neither the character information nor the side menu should be rendered.

An alternative solution that was originally used for state management was to have the engine use different update methods in the systems based on several status flags, instead of only one state flag. This was attempted early on in development and was deemed unsatisfactory in the long run due to code in each update method often being very similar to each other with only minor details differing between them. Using a single state flag produces much more manageable code. Additionally, using one update method regardless of state allows for simplicity in the engine. Therefore we decided that the best solution for the engine would be to use one state at a time. Using a single state flag allows outside systems to read the engine state and act accordingly.

Changing states

We will now briefly describe how changing the engine's game state is managed by the use of five different methods. These methods are loadDungeon, load-Overworld, loadMainMenu, gameOver and newGame. Depending on what the current state is, various tasks might be required before the new state is set; these methods handle that.

The method newGame is called whenever a new game is started, such as the first game or when a new game is started after game over. It restarts all non-critical systems so that a new game can be started, and unregisters all the previously registered observers from their subjects. The method instead registers the new instances of the systems as observers, allowing the old systems to be garbage collected. Restarting the systems is a good way to clear and reset them, while also allowing the same code to be used for the first start up.

The first game state that is set by the engine once a new round is started is Overworld. This is done by the previous method newGame calling load-Overworld. The method will then check the state and notice that it is set to MainMenu, requiring the menu system to unregister all its menu buttons from the engine. Similarly, if the current game state is set to Dungeon, that means that the dungeon system is required to unregister all its entities from the rest of the game before the overworld can be loaded, otherwise the player would see the dungeon's entities such as gold and enemies on the overworld screen, which would not be correct.

Each time a dungeon is loaded, even when it is just a change between two dungeons, the loadDungeon method in the engine is called. If the current game state is set to Overworld, the method makes sure the overworld has unregistered all its entities, which would be the stars. If the current game state was already set to Dungeon, the method will unregister the previous dungeon's entities before the new dungeon's entities are registered to the systems via the engine. There are several systems that require a reference to the new dungeon to be able to interact with it, to spawn items and move entities such as the player and enemies. loadDungeon will then update the systems' dungeon references before lastly setting the new game state to Dungeon.

The loadMainMenu and gameOver methods work similarly and will unregister entities from their systems before either the MainMenu or GameOver state is set.

2.2.3 Input using the subject/observer pattern

Observer is a design pattern in which one class acts as a *subject* that can send out notifications to other classes that subscribes to it, called *observers*. It is a useful way to manage event handling and the method we chose to use to handle input.

For handling all the input to the game, we chose to implement the subject/observer pattern. This pattern consists of an input manager that is always checking the keyboard and mouse for any key presses or mouse clicks. When a key is pressed, the input manager translates the key press to an action and notifies each system that subscribes to receive input. Then the system itself has to decide what to do with the action received. Most systems only operate on a small subset of actions and therefore disregard most actions, such as the interaction system which only cares if the player pressed the *F*-key.

2.3 Discussion

One of the goals we had with the project was to create our own game engine. Since this was something the whole group wanted to do we spent very little time researching existing game engines. It is still interesting to discuss the question of how the game would have turned out had we used a pre-existing engine. While we can not be sure of how it would have changed the results, we do not think the outcome would have been very different. An existing engine might have given us better graphics by having an advanced graphics engine already constructed and existing assets that we could have used or modified, but having an advanced graphics system was never one of our goals for the game engine. Although we do make use of Lightweight Java Game Library (LWJGL) for its input management and OpenGL bindings, we do not make use of it as an actual game engine.

While it is possible that we could have saved time by using a pre-existing engine it is also possible that the same amount of time would have been required to learn how to use the engine. The time we spent on implementing the features mentioned in chapter 3 would also still probably have been the same. Therefore, we are happy that we have produced our own engine and in the process have learned how to use the code architecture ECS and how to manage states.

Learning to use ECS was very hard in the beginning and we initially regretted our decision, but we soon felt that we had devoted enough time to it that it was too late to go back to a more familiar design pattern. After some time in the development phase we got used to managing it and are now quite happy with how it all has worked out. While it would sometimes have been easier to write code outside of the ECS pattern, we now have a very clean code base that enables us to further add new features with minimal effort.

We believe our game engine has been easy to work with and that it fulfills all our requirements, such as being able to handle input and manage different game modes. We are therefore pleased with the results.

Chapter 3

Game design

In this chapter we will discuss the design decisions made when developing the game aspect of Astrogue. First, we will discuss how we wanted to design the game and some common design options. Secondly, we will show the results of our implementation of the design choices we made. The last section will discuss the different design choices we made, and why we made them.

3.1 Background

When developing Astrogue, there were a lot of design choices to be made at first, which then later determined exactly how the game was implemented. Several questions needed to be answered before implementation of the engine could begin. From which perspective should the player observe the game world? What is the objective of the game? How should the world be represented on the screen?

Since the genre in which the game was developed was set from the start, some of the design choices were almost predetermined. The roguelike genre typically includes features such as permanent death, simple graphics, procedurally generated levels, turn based interaction and an advanced character system, according to the Berlin Interpretation (*The Berlin Interpretation* 2008). Even if the Berlin Interpretation could be too strict, it is good to have in mind when trying to please an audience that has played a lot of different roguelikes. Due to this, we have chosen to implement these features as well.

Permanent death is implemented to add a sense of danger and planning into the game. Permanent death also leads to the game having to be replayed from the beginning many times, which means all generated content has to be re-generated. Therefore, more emphasis is put on the procedural content generation. Furthermore, most roguelikes implement a turn-based system instead of a real-time system. The difference is that the player takes his or her turn and when finished, the other entities in the game world take theirs.

3.1.1 Previous work in artificial intelligence for enemies

The field of artificial intelligence (AI) is large. Within games, AI can handle almost all imaginable actions. An easy way to implement AI is by scripting: writing actions that will be executed in a specific order (Spronck et al. 2006).

While this method makes it easy to extend the script with additional actions, it also makes the enemies predictable, which can lead to players finding the game less challenging.

A more balanced method is *dynamic scripting*, which gives the enemies adaptive behaviour. This method makes the enemies able to adapt to different players' skill levels (Spronck et al. 2006).

The AI could also try to stay ahead of the player by having map awareness and an implemented probabilistic network, to try to predict the player's next step and move accordingly (Nareyek 2004).

Since the beginning of the project we wanted a basic AI. To be considered good enough, the enemies' AI should have some kind of pathfinding and basic decision making. We therefore decided to implement a pathfinding algorithm and ability to attack in a scripted way.

Pathfinding algorithms

Good pathfinding algorithms have existed for a long time. One of them is Dijkstra's algorithm which is guaranteed to find the shortest path in a graph (Dijkstra 1959). However, A* is said to be the optimal pathfinding algorithm because of the possibility to use heuristics (Dechter & Pearl 1985) (Nareyek 2004). That means, the algorithm can be told to trade speed for accuracy and vice versa. This option makes it possible for A* to behave precisely like Dijkstra's algorithm or strictly like Greedy Best-First Search, all depending on what heuristic is used. It is also possible to adjust the heuristic to be anything in between the other two mentioned algorithms. We have chosen to implement A* because this enables the possibility of changing the behaviour of the pathfinding algorithm by changing heuristic at a later stage.

3.2 Results

In this section, we first present an overview of the game. We then examine, in more detail, how its different systems are implemented and how they are related to each other.

3.2.1 Overview

When first launching the game, the player is presented a menu screen wherein they can choose to start a new game or browse a tutorial of the controls. After clicking *New game*, the player may choose a seed to use as well as pick a race and class from the three options avaliable. As soon as the choices are made, the player is presented a large space background with a number of stars. This is the overworld. By clicking on the different stars, the player may travel to them, land on them, and explore their respective dungeons. Sometimes, traveling to a planet will present the player with a popup explaining the game's plot. Once a planet has been chosen and traveled to, the dungeon crawl mode of the game may commence.

In dungeon crawl mode the world is tile-based and the actions are turn-based, meaning the player and AI take alternating turns to act. In this mode, enemies may also be attacked from a distance and potions may be used. Enemies are placed randomly in the dungeon and will try to attack the player if the player is within their line of sight, further explained in section 3.2.5. The dungeon mode is mostly about exploring the dungeon, looking for new items to equip, and monsters to fight for experience points and treasure. Equipping items will change some of the player's attributes and make the player better at something, for example gaining more health, not missing as much when attacking, or becoming better at dodging enemy bullets. The dungeons can have more than one level, which means you can find a flight of stairs and descend to another dungeon below.

3.2.2 Menu system

The menu system in the game is visible when the game is started and when the player dies. As can be seen in Figure 3.1, the menu is very simplistic and straightforward. It makes heavy use of the entity-component-system pattern where everything visible is an entity. Each button has Sprite and Position components, allowing it to be placed on the screen. To make each button clickable, an instance of Java's Rectangle (Oracle 2014) has been created with the same position, height and width as the graphical button. The menu system is registered as an observer to the input manager that notifies it when a mouse event occurs. The coordinates of the mouse click will then be compared with the rectangle's coordinates using the *contains* method in Java's Rectangle class. If the rectangle contains the mouse position, the button has been pressed.

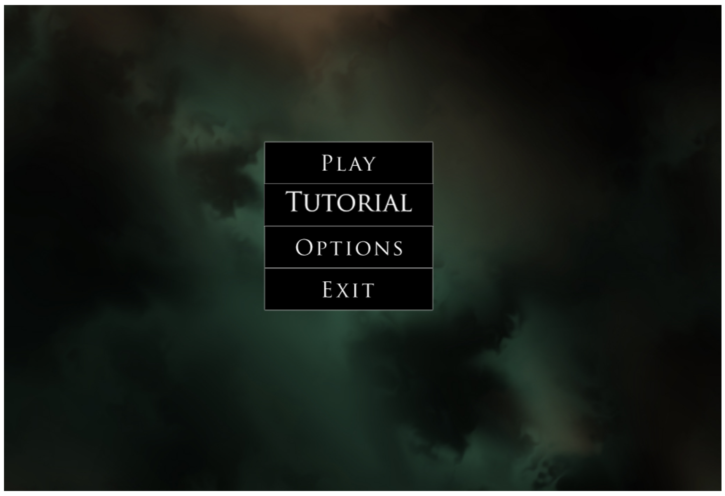


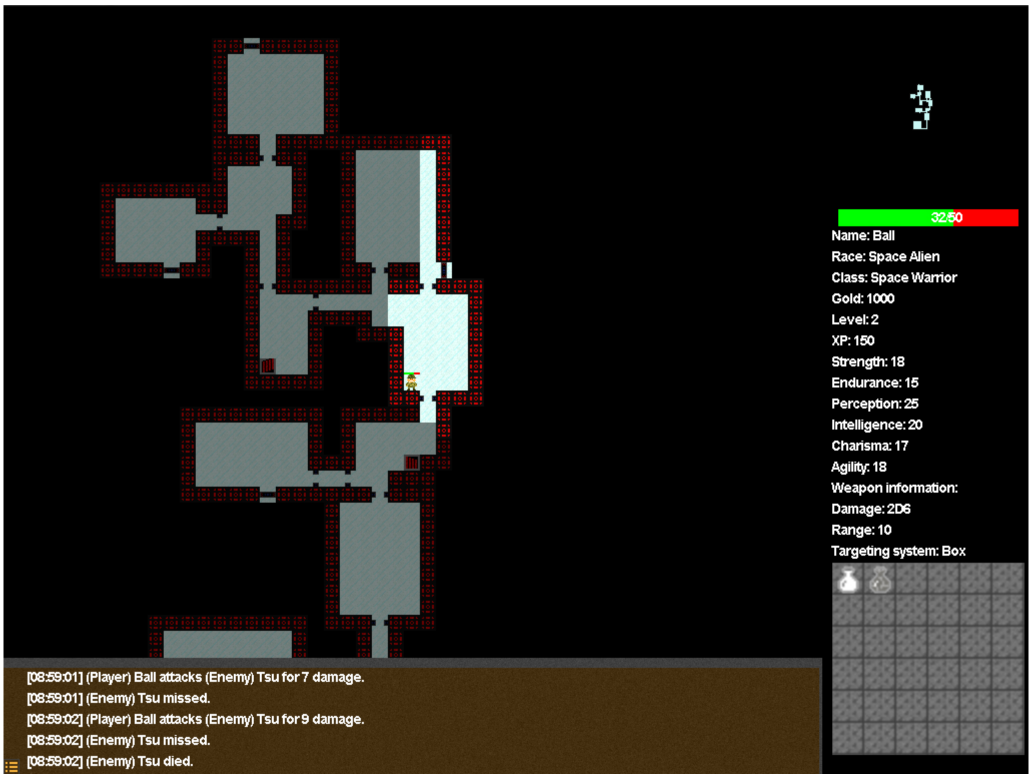
Figure 3.1: The main menu shown when starting the game.

To allow different menus to be displayed, the menu system is built like a state machine where only the rectangles that have visible buttons in that state will be checked. Every button displayed is registered to an array, which is used during state changes. When a state change occurs, every button in the array is unregistered from the engine so that they are no longer rendered nor displayed. Afterwards, every relevant button for the new state is entered into the array for future state changes. The same array is used to unregister all entities from the engine when the game starts and the menu system is no longer used.

3.2.3 Dungeon crawl

The dungeon crawl mode is the more classic roguelike mode, where the player can perform actions such as walk around, explore, attack enemies and find items.

A screenshot of the world in the dungeon crawl mode can be see in Figure 3.2 and is represented with a grid of square tiles that can be walls or floors. The tiles are then stored in a two-dimensional array and each enemy, item or the player is placed on one of those tiles. The tiles are outside of the entity-component-system pattern and keep track of what items have been placed on them, which sprites they are using, if they block line of sight and if they are walkable or not. If the tile is not walkable, the tile is treated as a wall. The tiles themselves do not keep track of their own positions. Their positions are determined by their locations in the dungeons tile array.



```
32/50
Name: Ball
Race: Space Alien
Class: Space Warrior
Gold: 1000
Level: 2
XP: 150
Strength: 18
Endurance: 15
Perception: 25
Intelligence: 20
Charisma: 17
Agility: 18
Weapon information:
Damage: 2D6
Range: 10
Targeting system: Box
```

```
[08:59:01] (Player) Ball attacks (Enemy) Tsu for 7 damage.
[08:59:01] (Enemy) Tsu missed.
[08:59:02] (Player) Ball attacks (Enemy) Tsu for 9 damage.
[08:59:02] (Enemy) Tsu missed.
[08:59:02] (Enemy) Tsu died.
```

Figure 3.2: A screenshot from the game showing the game interface and part of a dungeon.

When adding an entity, its coordinates are used to determine where in the dungeon's tile array the entity should be placed. Storing the entities this way allows for the dungeon to be able to register and unregister itself and all its entities from the engine when the player switches between dungeons or game modes. When a dungeon is loaded it registers all its entities to the engine, which subsequently registers them to the relevant systems. Similarly, when the player leaves a dungeon, the same list of entities is used to unregister all the entities from the engine. This enables the next dungeon or game mode to be loaded without having entities from the previous dungeon interfering.

Moving in a dungeon

A special movement system has been devised. It will run and update for each game loop when the game state is set to Dungeon. Every entity having an Input and Position component will be registered to this system and for each iteration of the game loop, the system will loop through all its registered entities and check if any of them has input that will cause the entity to move.

A player can move horizontally, vertically and diagonally with directions named after the cardinal points. These movement directions are represented by enumerables in the input system and are sent to the movement system when the buttons for movement are pressed. Since turning requires no additional tile to be available it will always be possible.

The movement system always has the current dungeon registered to it and the Dungeon class has a method to check if a specific tile can be walked upon or not. The tile can be considered untraversable if it has its walkable flag set to false (for example: a wall) or if one or more entities that contain the *BlocksWalking* component have been registered to it (for example: a door or an enemy). Since an entity always moves an entire tile and because of how tiles are stored collision control is easy. The neighbouring tile can quickly be looked up and examined for its walkable status. To determine which tile should be examined for its status, the moving entity's X and Y coordinates can be looked up from its position component. Based on which direction it is attempting to move, the tile can be looked up using Table 3.1.

Table 3.1: This table shows how the position is updated when moving in a certain direciton.

Direction	X coordinate	Y coordinate
North	x	y+1
West	x-1	y
East	x+1	y
South	x	y-1
North West	x-1	y+1
North East	x+1	y+1
South West	x-1	y-1
South East	x+1	y-1

If the entity can move to the tile and has turns remaining, it is first removed from the tile it is registered to, then has its position component updated and is then registered to the new tile. If the entity contains a Direction component, it is updated with the new direction. Lastly, its TurnsRemaining component, explained in *Turn management* below, is decremented by one, since moving costs one turn.

Dungeon interaction

Using the same technique for tile lookups, an interaction system has been designed to interact with entities in the dungeon. There are two ways to interact with entities in the game. In the case of gold that has been placed on the map, the player needs only to move to the tile and it will automatically be picked up. The other way to interact is that if the player presses the interaction button, which is F on the keyboard, the interaction system will be triggered to try and interact with the environment. It will examine what is located on the tile that the player stands on and see if it can interact with that, such as a stair to another dungeon or to the overworld. If the interaction system cannot interact with anything on the same tile, it will attempt to interact with whatever is infront of the player, such as doors that can be opened or closed.

Turn management

A system was created to manage turns. Both the enemies and the player require a component that keeps track of how many turns they have. The number of turns an entity has can be a positive integer, a negative integer, or zero. In the normal case, the number of turns will only be one or zero, but it can also be higher or lower than that. A higher value can be achieved by receiving beneficial bonus turns from a potion in the game. Similarly a negative value can occur when using potions that act as paralysing poisons, making the player or enemy wait several turns before they can move again.

The turn system is created in such a way that the player always has the first turn before the enemies have theirs. Every entity that contains a TurnsRemaining component will be registered to the system, which in reality means every enemy as well as the player. During each game loop, the turn system will go through every entity and check if any entity still has turns to spend. When all entities have equal to or less than zero turns left, that round is done. Every entity will then have its number of turns left incremented by one. Incrementing by one instead of directly setting the value to one means that negative effects can last several turns.

Shadowing the dungeon

When observing the game world, we want the player's vision to be limited to the sight that the player character in the game has. Calculating *field of view* can be done in several different ways, with different results depending on how the matter is handled. We have chosen to implement an algorithm called the *shadowcasting algorithm*, which involves letting lightrays spread from the player character and then shadowing all tiles that the lightrays do not touch. This way, obstacles cast shadows behind them where the player cannot see, hence the name shadowcasting.

When implementing the shadowcasting algorithm, we first needed to identify that the screen could be divided into eight octants. If we could solve the problem for one of these octants, we could apply the same technique to the other seven but with slightly different parameters. As seen in Figure 3.3, solving it for the filled octant can be applied to the other seven octants.

For the algorithm, a map of which tiles are opaque or not is needed, as well as the position of the player. The algorithm starts by defining vectors as the diagonal and horizontal sides of the filled triangle in Figure 3.3. By recursively checking the tiles between the vectors, and lowering the top vector if a blocking tile is encountered, the octant is lit in a realistic way. Furthermore, by defining the vectors differently we can apply this same technique to the other seven octants, lighting the entire area around the player.

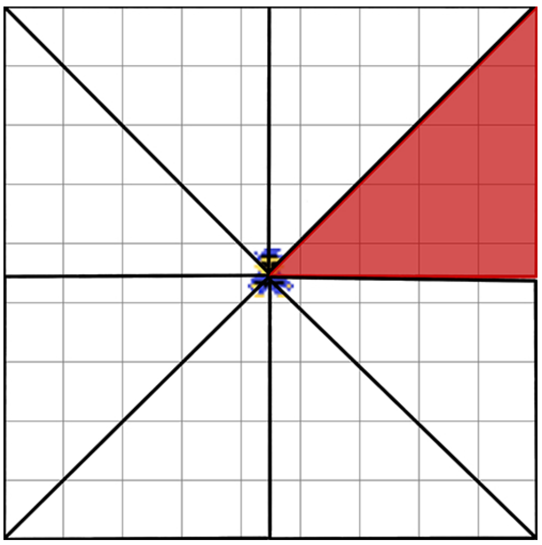


Figure 3.3: A grid with the player standing on the middle tile. The technique for solving the shadowcasting for the filled octant can be applied to any of the other seven octants.

Combat mechanics

It was decided that the combat mechanics should not be very complex, as the focus of the project is the procedural content generation. An attack is calculated by rolling three simulated 10-sided dice, adding a modifier based on the attacker's offense attribute and then comparing the result to the target's defense attribute. Subsequently, if the attack roll was higher than the target's defense, damage is calculated depending on the weapon used. Instead of using multiple dice, one could just randomise a number with all values having the same probability. The reason multiple dice were chosen was because it results in a normal distribution, making average attacks more probable.

3.2.4 Overworld

The overworld, as we have chosen to call it, consists of a space background filled with stars. Each one of these stars is an entity belonging to the entity-component-system framework. Each star has a sprite, a position and a dungeon associated with it. The overworld adds another layer to the game, bringing more depth to the gameplay.

Early on, the overworld was envisioned to be a simple dungeon, but since we wanted additional menus and the stars to not be placed in a simple grid, it was warranted to implement the overworld as a separate system.

When a new star is created, no dungeon is generated at first, only a seed. This allows for the creation of large star systems with practically no loading times. Creating a large amount of stars takes practically no time at all, but generating a dungeon can take a couple of seconds depending on the complexity of the level. Therefore, each star holds a seed that will be used by the level generator to create a dungeon when the player first enters a level. When the dungeon is generated, a reference to it is saved with the star so that if the player was to leave the dungeon and then revisit it, there would be no need to re-generate the level. This also allows for an easy way to save the state of the entities in a dungeon, meaning the player cannot force re-generation of levels with easy access to gold.

In the overworld, the main parts of the generated plot are played out. When the player travels between stars, text popups will tell the player what happens, e.g. if the main character meets someone, if the main character receives a gift or if the main character enters an important star system. Some parts of the plot, however, are played out in the dungeons; if the player has to defeat a boss or find some important item, they have to land on a planet and explore the dungeons. This is all controlled by an implemented plot system, further explained in chapter 5.

3.2.5 Enemies with artificial intelligence

The enemies in Astrogue are very similar to the player as they have the same attributes and vision, something that was easily implemented in our entity-component-system framework. Enemies will constantly check if they can see the player, using the same method as our shadowcasting algorithm. If the player can be seen, the enemy AI will calculate the shortest path to the player using the pathfinding algorithm A*, and move towards the player.

When the enemy gets within attack range of the player, it will attack. If the enemy loses sight of the player, it no longer pursues the player and instead moves arbitrarily. Our A* algorithm is based on Patels' implementation (Patel 2011) and uses the heuristic of behaving like Dijsktra's algorithm, in a world of square tiles, that allows diagonal movement. There is no mutual decision making among the enemies, they all act on their own.

3.2.6 Items and inventory

The player will encounter certain entities that can be picked up and then used or equipped. These are entities handled by the inventory system and can be divided into two subgroups: equipable and usable. The usable items consist of potions and quest items, which sometimes can have a use-effect tied to them. The potions are randomly tied to a colour and the player needs to test out the different potions to see which ones to drink and which ones to throw at enemies. This element ramps up the difficulty, but adds more thought to play. This is typical for roguelikes and adds a trial and error aspect to the game. The potions will be reshuffled and tied to new colours after the game is over and because of permanent death this will happen many times.

The other group of items encountered when playing the game are the equipable items. These consist of weapons, helmets, armor, gloves, boots, amulets, and rings, all of which increase some attribute or attributes for the player, except for the weapons which also determine the player's method and distance of attack.

3.3 Discussion

The menu system is quite easy to use and add new buttons to, but the fact that it requires both a rectangle and a visible entity for each button is not ideal. The system would be a lot easier to use if it only required one of those. It would be possible to add a rectangle component to each entity, but that would require constant fetching and comparing with each active button, so we felt this way was easier in the long run. It also created an easy standard to use throughout the game where other clickable events needed to occur.

The dungeon structure was one of the first things that were implemented in the game once the engine had its basic functionality working. The basics of the dungeon structure and the tiles proved to be very well planned, which is evident since very few changes on those classes have been required since their initial creation.

The tile-based world and how every entity registers to the tile it stands upon made both collision checks and movement very easy to implement. Similarly, interactions could simply check the current tile and the tile in front of the player. Combat was easily implemented thanks to cheap collision controls and we had no need to utilise more advanced tools, as our system could simply draw a line between the attacker and the target and use those coordinates to look up which tiles were affected.

When implementing shadows in the dungeon we wanted it to feel as real as possible and spread in a natural way. Therefore we have chosen to use the shadowcasting method and in a way let lightrays fly in every direction from the player and light up every tile that they touch the center of. Implementing a simpler version of this would have taken significantly less time but it was perceived to have a great impact on the overall impression and therefore the more advanced method was used instead.

An alternative way could have been to let the dungeon completely re-generate each time, but we did not want that to happen since the treasure would also be re-generated. Another alternative would be to save the dungeon to disk once generated, and reload it when required, but since no other states or entities are saved to the disk we felt that it would not be worth it to implement that just for this purpose. The last idea was that instead of saving the dungeons, a list of the entities and their states could be saved, since the dungeons take up the majority of the space in the memory (due to the fact that a 50 x 50 dungeon would take up 2500 tiles). This idea was conceived late in the development phase and there has not been enough time to test this idea out, since it would require changes in the dungeon, the overworld and the level generators.

It was decided from the start that the main focus should not be to implement a very advanced enemy AI. Therefore we implemented the bare minimum and we can honestly say that we feel satisfied with that decision.

The implementation of items and inventory was pretty straightforward due to the fact that there are not many ways they can be implemented. A little more depth could be added to the items by creating them using procedural content generation. This would have been an interesting addition to the game, but would have taken significantly more time to implement. Furthermore, we chose to incorporate equipable items in addition to the usable ones for more depth and complexity to the game.

Chapter 4

Level generation

In this chapter, we will mention the most common algorithms that are currently used to generate maps and levels. We will then proceed with describing the two algorithms that we have implemented. Finally, we discuss these two algorithms and other options.

4.1 Previous work within level generation

Using procedural content generation (PCG) to generate levels is desirable for many reasons. One reason why, is because levels often take up a lot of memory, and by using PCG methods, the memory usage can be heavily reduced (Togelius, Yannakakis, Stanley & Browne 2010). Another reason why level generation is important is that it reduces development time and costs. This has led to procedural level generation becoming popular among game creators.

There are many ways to categorise PCG when creating maps and levels. One way is to divide the level generation by when it is used: during run-time or during the development phase. Either the level generator is part of the game, running on-the-fly, or it is used by the level designers as a tool during the design stage (Doull 2008) (Togelius, Yannakakis, Stanley & Browne 2010).

One can also divide the generation of levels into *generation of the space*, the geometrical layout of the level, and *generation of the mission*, the series of tasks the player needs to complete to get to the end of the level (Dormans 2010). Both of these are formidable challenges by themselves, but making these work in combination with each other is important when trying to generate levels similar to levels designed by human designers.

There are many different genres of games, with each one requiring different kinds of levels. This creates a demand for different kinds of level generators, where each generator creates levels after distinct requirements and criteria. For example, a level generator that generates levels for a platformer (Compton & Mateas 2006), like Spelunky (Mossmouth 2008), would not be able to generate levels for a racing game (Cardamone et al. 2011), such as Gran Turismo (Sony Computer Entertainment Inc. 2013), and vice versa.

Three-dimensional terrain can be generated by height maps, a data structure describing the height as a function of the horizontal (X,Y) position. Fractals are commonly used to generate these height maps (Miller 1986) (Musgrave et al.

1989) (Olsen 2004). Another way to generate these height maps is by using Perlin noise (Hnaidi et al. 2010). A disadvantage of height maps is that a single height map cannot be used to represent caves or overhangs, since each position only have one height value. Having more than one height map escalates the complexity. Caves are often generated with another algorithm, like in Minecraft (Mojang AB 2011).

Genetic algorithms (GA) can be used to improve the maps created. GAs are evolutionary algorithms that mimic natural selection (Mitchell 1998). They are applied in many fields such as economics, social systems, and optimisation. In level generation, GAs use a fitness function to evaluate how well levels fit to some purpose. Defining this fitness function is the main problem of using a GA, since defining what gives value to a level is not obvious. In practice, a first set of levels is generated and the GA evaluates each one. Then a second set of maps is generated, based on levels from the first set that best met the given criteria according to the fitness function. This way the population of levels only keep favoured mutations, and as the number of generations goes on, the population keeps evolving to better fit the purpose. While there are implementations of GA alone, such as the canonical implementation (Whitley 1994), it is also possible to mix the genetic method with level generation algorithms, e.g. *cellular automata* (Mitchell et al. 1996). The possibility of using GA together with other algorithms is a big advantage. However, since there are a lot more levels generated, in different steps, the time it takes to get the output level is significantly longer.

One problem with level generation is generating playable levels. Evaluating the playablity of a certain level can be difficult. Research have been done in this field (Liapis et al. 2013); however, we chose not to focus on this problem. Not only should the level be playable, it is often desirable for the level to be fun to play too, which is another problem (Compton & Mateas 2006).

We have focused on algorithms for generating indoor areas. One of the algorithms we have chosen is an implementation that is a combination of multiple techniques rather than one named algorithm. This method is derived from *TinyKeep*'s (Phigames 2013) method of dungeon generation, which uses *Delaunay triangulation* (Lee & Schachter 1980) to connect rooms. The reason we chose this method was because the output met our requirements and matched how we envisioned our levels. Another algorithm we have chosen is cellular automata.

4.1.1 Cellular automata

The concept of cellular automata was discovered by John von Neumann and Stanislaw Ulam as a possible approximative model of biological systems (Neumann & Burks 1966). A *cellular automaton* is a grid of cells where each cell has a finite amount of states. Each cell has a neighbourhood which is a set of cells that are related to the first cell. Upon these cells acts a rule which generates new states for individual cells based on the previous generation of cells. One famous cellular automaton is Conway's Game of Life (Conway 1970). A characteristic of the cellular automaton evolution is that it is irreversible. This leads to the algorithm showing several self-organisational properties given a random initial configuration (Wolfram 1983).

Given that cellular automatons have these self-organisational properties, cellular automata offers a solution that is both efficient and reliable for PCG. Other

PCG methods such as using fractals, while efficient, have a limitation in that their only parameter is a random generator seed. This leads to huge differences in output with only a small difference in the seeds. The advantage of cellular automata is that they are more controllable since they are more parameterisable, meaning that they can have more parameters defining the outcome (Johnson et al. 2010).

4.2 Results

To fulfill our requirements, we figured that multiple level generators had to be created to give us a wide enough range of variation in the levels. For example, a building will have entirely different layouts and requirements than a cave. Thus, we created two generators: one creates levels in man-made areas and one creates more natural-looking environments. Both of our generators always generate maps that are guaranteed to be playable. Since they are based on seeds, it is possible to generate the exact same map again using the same seed. Additionally, the nature of the map can be slightly changed when giving the algorithms different parameters.

4.2.1 Generating levels with connected rooms

In this section, we describe our first level generator, which generates levels that give the feeling of being inside a building, as it produces a map of connected rectangles. The steps are explained as they appear in the procedure, starting with the creation of rooms. Then we explain how the rooms are connected, first through a Delaunay triangulation, then followed by *Kruskal's algorithm* (Kruskal 1956) to minimise the connections. The next step described is the conversion of connections into corridors. Lastly, we present how we fill the level with content.

Creating rectangles

The generator begins by generating a fixed amount of rectangles in different sizes, placing them on an empty area and then separating the rectangles until none overlap. When this is done, all rectangles with a size above a certain threshold are decided to be called *rooms*. We call the smaller rectangles *nooks* and save them for a later purpose.

Delaunay triangulation

A Delaunay triangulation takes a set of points and links the points together, creating triangles in a graph. Every triangle has the property that when circumcircled, no point is within the circle (Lee & Schachter 1980). The most simple way to do the triangulation is by inserting the points one by one and removing triangles which have a circumscribed circle encompassing the point. Then new triangles are formed between the inserted point and the corners of the triangles that were removed. This is called the incremental implementation (Lischinski 1994).

To get a playable map that is in line with our initial condition of looking man-made, an organised way of connecting the rooms is needed. What makes

Delaunay triangulation appealing, is that it can connect points in a structured manner with no intersections. To maximise diversity, a random point in each room is chosen, and our algorithm calculates a Delaunay triangulation on the set of points. As seen in Figure 4.1, all nodes are connected and no edges cross each other.

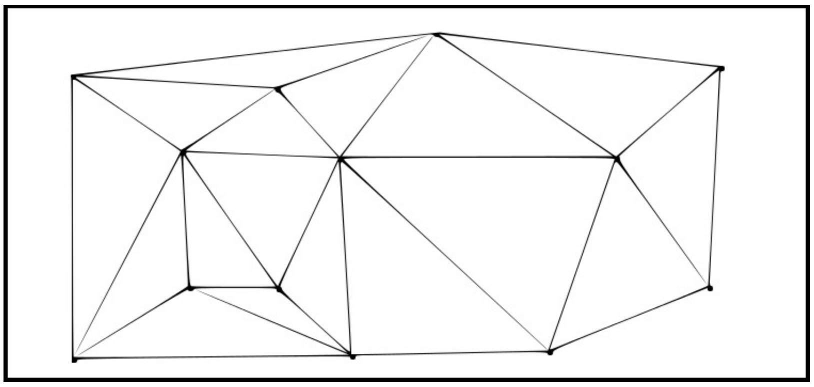


Figure 4.1: Here is an example of how our Delaunay triangulation algorithm connects all nodes with no edges crossing each other. A node represents an arbitrarily chosen position in a room.

Kruskal's minimum spanning tree

We do not want our levels to be cluttered with corridors. We also want all of our rooms to still be connected. Using Kruskal's algorithm can solve this problem. Kruskal's algorithm takes a connected graph where the edges all have a weight, and creates a *minimum spanning tree* (MST), keeping the edges with a minimum total weight between them that still connect all nodes. The edges are sorted by weight in increasing order. When creating the MST, the algorithm starts with any of the edges with lowest weight. It then proceeds with iterating through the sorted list of edges and checks if adding an edge creates a cycle; if not, the edge is added to the MST (Kruskal 1956).

The generator uses Kruskal's algorithm to create a MST. The MST is the minimum possible representation of the map. Figure 4.2 shows what the graph from Figure 4.1 looks like after Kruskal's algorithm has been applied; this is the minimal graph where all nodes are connected. However, we want the player to explore as much new content as possible and we do not want to force the player to backtrack if they encounter a dead end. Therefore, a fixed percentage of the edges returned by the triangulation is added back to the MST.

Connecting the rooms in an interesting way

The last part is to create a good visual representation of the rooms and the connected graph. The rooms in the graph simply translate to rooms in the level. The edges translate to corridors and if they intersect any nook, that nook is also added to the map. All other nooks are discarded. This is done since, in our opinion, linear corridors are less interesting. Figure 4.3 shows an example of

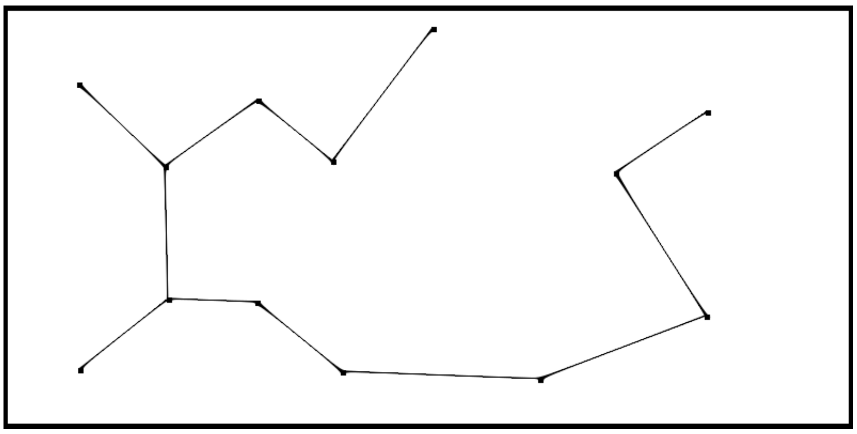


Figure 4.2: An example of how Kruskal's algorithm takes the connected graph in Figure 4.1 and only keeps the edges with a minimum total length between them that still connect all nodes.

how the rooms can be positioned and how the final version of that map would look with the rooms connected with corridors.

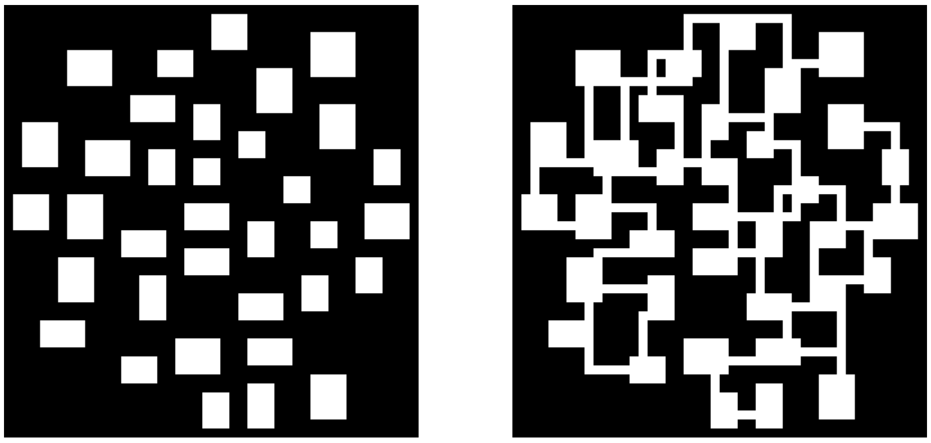


Figure 4.3: These images show how our first level generator connects rooms using corridors. To the left we see the rectangles that are kept as rooms. To the right we see the rooms connected with corridors.

Filling the map with content

When the basic map exists, the generator starts to fill the level with content. Traps, treasures, and enemies are placed randomly according to certain rules: for example, some types of content are solely placed in corridors, while other types are placed in rooms. The algorithm then calculates where doors would fit in and places doors in these spots. A door can only be placed where a room would otherwise have a wall, but now has a corridor connected instead. Finally, stairs are created. There is a chance that every level may contain a flight of stairs that lead down to a floor below. The chance of a floor containing stairs is reduced each time a new floor is created, which is done because we want the player experience to be as varied as possible. When limiting the size of the levels, the player is forced to visit new planets. Figure 4.4 shows an example of a generated level with connected rooms, used in the game.

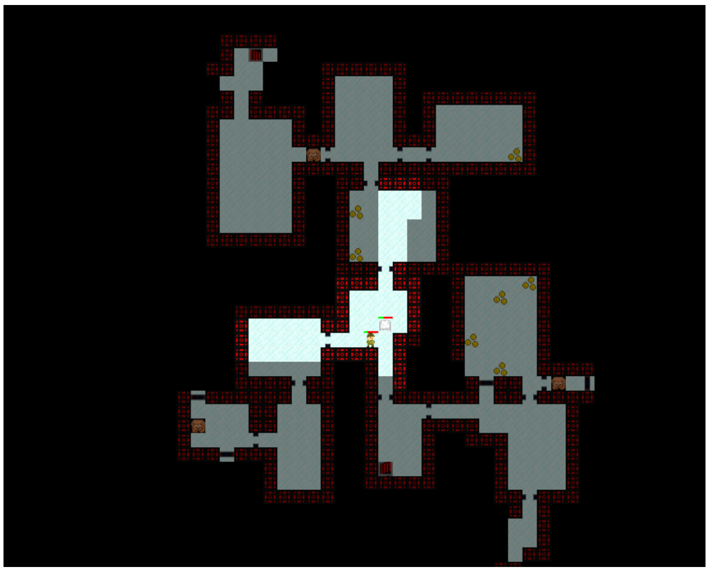


Figure 4.4: The final result of a level created by the generator that connects rooms. Entities such as enemies and gold are also seen in the picture.

4.2.2 Generating cave levels with cellular automata

For the second algorithm we wanted something that generates more organic, cavelike levels. For this purpose we have chosen cellular automata, which is discussed in this section. First, we describe the cellular automaton rules for our algorithm, then how we fill *pockets* giving us a single connected cave. Finally, we explain how we fill the level with content.

The cellular automaton

The algorithm begins with a square grid of a randomly chosen size, with 45% of the cells, chosen randomly, initialised as walls. The rest of the cells are initialised as floor. This can be seen to the left in Figure 4.5. These are the two states in this cellular automata: a cell is either a wall or a floor. Over a number of iterations, cellular automata rules are applied on each cell. The rules are as follows: if the close neighbourhood (the eight neighbouring cells) contains at least five walls, the cell becomes a wall, otherwise it becomes a floor. However, this leads to caves that often have one huge open space or several disconnected spaces. This was solved by adding a second rule: if the number of walls within two steps (the closest 24 neighbours) contain two or fewer walls, then the cell becomes a wall. This rule creates walls in open spaces and thus makes sure that there are no huge open spaces. An example of the first four iterations is visualised in Figure 4.5, Figure 4.6 and Figure 4.7.

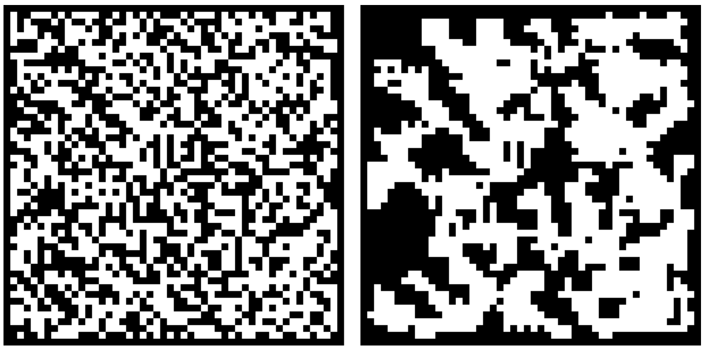


Figure 4.5: An example of the process behind our cellular automata level generator. To the left the initial grid of randomised cells can be seen, with 45% walls (black) and 55% floor (white). To the right is the first generation of the algorithm.

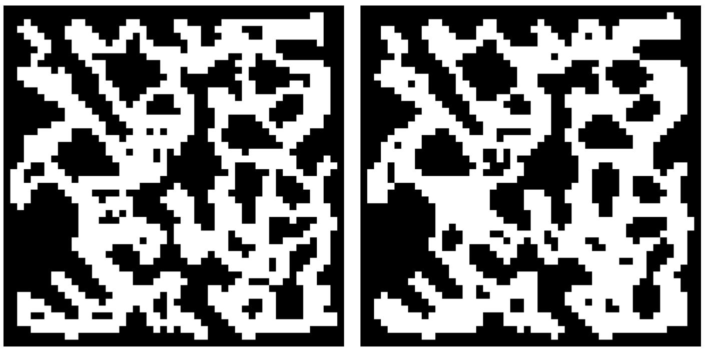


Figure 4.6: The second and third generations of our cellular automata level generator.

Filling pockets

After we have generated this cave there is a possibility that there exist so called *pockets* that are not connected to the main cave. This problem is solved by finding all separate caves and filling the smaller ones with walls so that the only cave left is the largest one, as shown in Figure 4.8. This guarantees that the whole level is reachable and playable. One other solution would be to, instead of filling the smaller caves, connect all the separate caves with corridors; however, this could make the cave look unnatural.

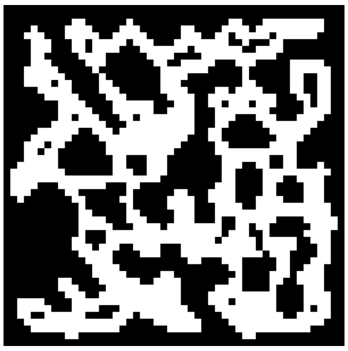


Figure 4.7: The fourth and final generation of our cellular automata level generator.

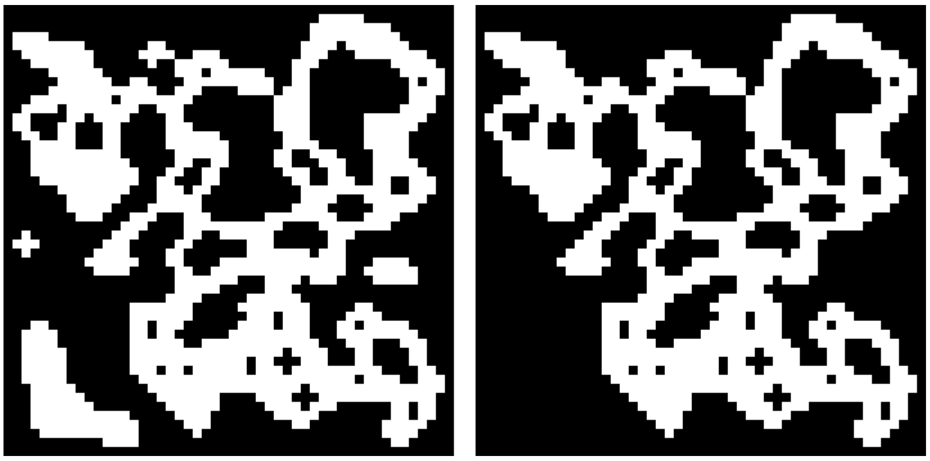


Figure 4.8: This figure shows the removal of *pockets*. Before (left) and after (right).

Filling the cave with content

Once the map is generated, all that is left is the simple task of filling the cave with content such as enemies, stairs, and treasure on appropriate tiles. Appropriate tiles are those not in the immediate vicinity of a wall, thus limiting the spawning tiles. The player's start position is also considered so that enemies cannot spawn close to the player. It would be frustrating if monsters could spawn right next to the player when they just entered a dungeon.

The finished cave

The end products of the cellular automaton algorithm are varied levels that are cavelike. Example levels can been seen in Figure 4.9. With the removal of the potential pockets it is guaranteed that there is only one connected cave. Figure 4.10 shows a screenshot of what a generated cave looks like in the game.

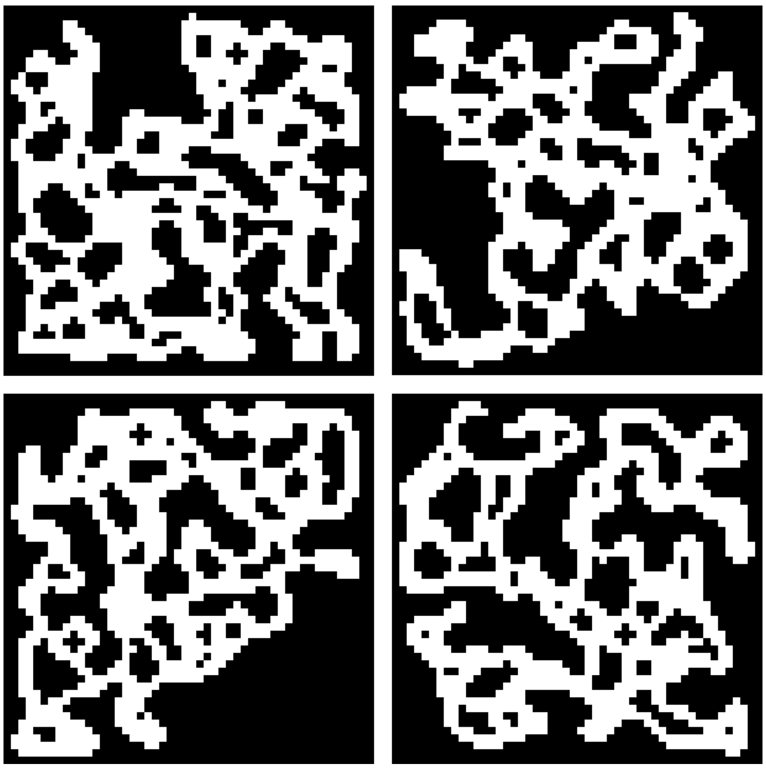


Figure 4.9: Here are four different examples of generated levels, showing that the algorithm can generate different caves while still having a cohesive style.

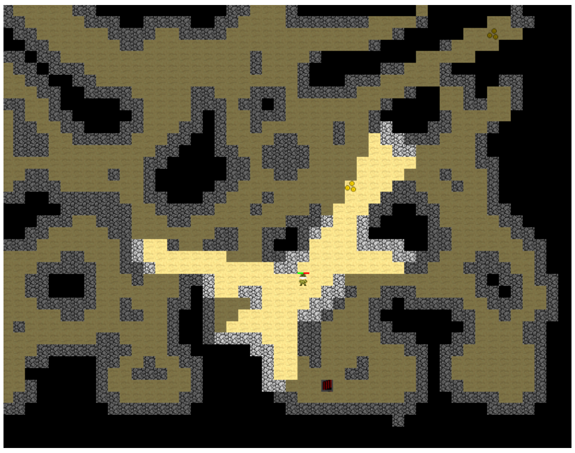


Figure 4.10: An example of a level generated by the algorithm used in the game.

4.3 Discussion

In this section we will discuss the results of our level generators. We look at benefits and drawbacks of our implementations, and what we could have done differently.

We think that the level generator that connects rooms is functional and elegant. All rooms are reachable and the placement of the doors is done in such a way that they affect the gameplay. No door is placed in an illogical space according to us. Enemies, treasure and traps also have thought behind their placement and have impact on the game. We find that the generator can generate a sufficient amount of levels, since we use seeds that are represented as Java Longs, and each seed is likely to give a different level. On top of this, it is possible to give the generator other parameters such as corridor density and number of rooms, which leads to an even greater amount of viable levels.

One aspect we did not like as much, was that even though the generator creates levels with some kind of indoor resemblance, the levels are not realistic enough. However, one could argue what constitutes as realistic in a science fiction game. We are not happy with the fact that when trying to achieve a specific style of level, it is hard to know which parameters to adjust and how much.

We consider our other algorithm, the cellular automaton, to have turned out really well. It is an algorithm with relatively simple rules, but it is capable of generating complex, naturally looking caves. This algorithm also gives more variation to the game than if we had only used our first level generation algorithm, and variation was one of our main goals. Similar to the other algorithm, the cellular automaton can generate a nearly endless amount of levels, which we are proud of. The way content is spawned within the levels, mentioned in section 4.2.2, is also an aspect we believe to be sound.

If we had begun the project with our current knowledge, we would probably have explored genetic algorithms to aid in the generation of more interesting levels. Furthermore, we would have liked to combine several algorithms to create levels that are bigger and with much more variation.

Although the game already have some variation, we really would have liked to implement even more level generating algorithms. In addition, the currently implemented algorithms could be extended to generate more interesting levels. One could add, for example: several types of traps; aesthetics such as trees, rocks and water; crates and other objects that can act as cover in combat; boss encounters; and more.

One feature that we would have liked to have explored, is how to realise procedurally generated puzzles within our level generation algorithms. One such puzzle is a simple locked door placed within the level, which requires the player to find the correct key to progress. As discussed by (Dormans 2010).

Chapter 5

Plot generation

This chapter covers our work on procedurally generating names and plots in Astrogue. First, the previous works in the two fields are presented and discussed. The results section explains our Markov chain name generator and the plot generator implemented for Astrogue, *AstroPG*. AstroPG generates plots in an AI simulation with multiple autonomous agents. How generated names and plots are integrated within the game is explained. Finally, our results are reviewed and our implementations' advantages and disadvantages are discussed in relation to previous works.

5.1 Previous work

This section describes and examines the previous research that has been done in procedural generation of names and plots.

5.1.1 Name generation

In this section we will present numerous ways of generating random names and try to shed light on the different methods' advantages and disadvantages. All of the methods require some sort of database, either a set of names or some statistics of how characters or syllables are distributed.

The problem of generating names for persons, places, and objects can be solved by using Markov chains. Actually, any content that essentially is a series of components can be generated using Markov chains. A Markov chain is a memoryless random process with a finite amount of states (Norris 1998). The only knowledge needed is the current state and the probabilities for the transitions to the next state. Markov chains have numerous application areas. They are e.g. used in biology when predicting epidemics, in queuing networks, and in computer science Markov chains can be used when analysing linguistics in texts (Dunning 1994) (Norris 1998). A Markov chain allows for use of different *orders*. The order of the Markov chain decides how many instances the current state holds, i.e. when building a word, the current state can hold from one character up to any number of characters. In Figure 5.1, an example of a third-order Markov chain building a word is shown. An advantage with this method is that far more names can be generated than what could be had in a database.

The disadvantages are that controlling the result is difficult and names can be unpronounceable or profane.

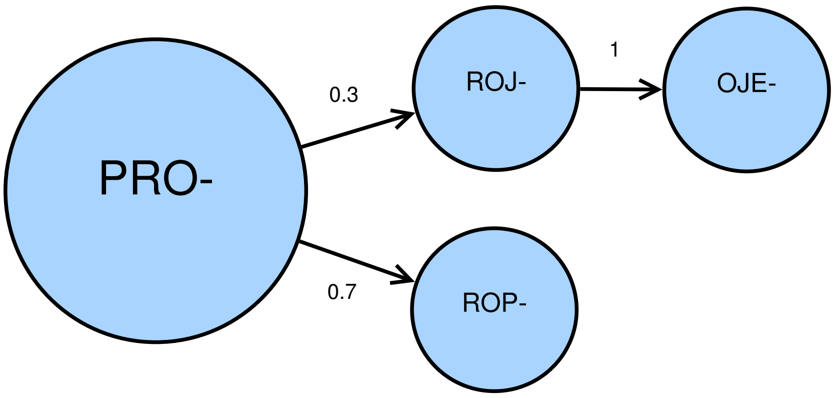


Figure 5.1: Here we see a third-order Markov chain, building a word, in the current state *PRO*. There is a probability of 0.3 that the next letter is *J*, which will make the next state *ROJ* and a 0.7 probability that the next letter is *P*, which will make the next state *ROP*.

Setting the transition probabilities is possible in multiple ways. One way is to enter them by hand, offline, to control the results to some extent. Another way is to use N-grams, which are N-character long strings. First, define the N-gram strings be as long as the order of the Markov chain that is used. Then, analyse the list of names as a training set when running the program and save the occurrences of successors of the N-grams to get the probabilities (Cavnar et al. 1994). The latter way is more flexible and gives the potential of using different training sets to produce names. That makes it possible to generate a name based on e.g. all Finnish names and a name based on all American names, where the two will likely be very divergent, since there are letter combinations in Finnish names that do not exist in American ones and vice versa.

Another way of naming entities is to have a database filled with names and simply pick an arbitrary one. A set of names can either be created or taken from an external source. The benefits of using this method is that you can control which names are generated and that it is easy to implement. The clear drawback is that the amount of names that can be generated is the same amount that you have in your database.

A third way of approaching the problem could be to divide the names into segments and then construct new names out of those segments. To decide where to split the names, one can analyse the database of names according to *letter successor variety*, which essentially says that a string of letters is a segment if the string has enough successors (Hafer & Weiss 1974). Another viable way to split the names into segments is to divide by syllables. A syllable consists of a vowel or a vowel-like consonant that can be prefixed and suffixed by consonants (Jones et al. 1997). The profit with the segmentation approach is that the names created are pronounceable. One weakness with the letter successor variety implementation is that the amount of names that could be generated is less than the names in the database. In the case of dividing by syllables, the drawback is that implementing a refined method that recognises syllables in more than one language is time consuming.

Since one of the fundamental ideas was to vary as much as possible, our implementation of the name generator uses Markov chains. As mentioned earlier, this method could result in names that are unpronounceable, but that is seen as a feature, since the game takes place in space on planets that surely have linguistic rules that differ from Earth's. The segmentation when dividing by syllables was also interesting for this project, but since the implementation requires several methods to recognise syllables in different languages, the approach fell short. When creating the transition table using N-gram analysis, it makes no difference to use e.g. Swedish or English. The implementation stays the same.

5.1.2 Procedural generation of plots

Being able to automatically generate plots has for a long time been a dream in the entertainment industry (Díaz-Agudo et al. 2004). One of the first automated plot generators is TALE-SPIN, created and described by James Meehan in his dissertation from 1977 (Meehan 1977). TALE-SPIN simulates a forest world with talking animals, that try to reach goals such as quenching their thirsts. By creating sub-goals from these goals, the animals interact with each other and the world. A story is created during this AI simulation, which is told with generated natural language (Meehan 1981).

TALE-SPIN has, since then, inspired others to create their own programs for automatically generating plots (Lang 1999). A number of different approaches to generate plots have emerged, most focusing on artificial intelligence, such as the character-based plot generation in TALE-SPIN and similar projects (Cavazza et al. 2002), using reinforcement learning (Nelson et al. 2006) and using case-based reasoning (Díaz-Agudo et al. 2004) on a case base of already existing stories. Essentially, there are two different paradigms in plot generation; a *structuralist* approach and a *transformationalist* approach (Peinado & Gervás 2006). Typical for the structuralist approach is that stories are generated from multiple smaller parts into more complex structures, akin to the DEFACTO story generator (Sgouros 1999). Defining for the transformationalist paradigm is to generate the plot from beginning to end, as with for example TALE-SPIN (Meehan 1981). Our implementation would fall into the transformationalist category.

One of the most common methods for plot generation is based on the work of Russian formalist Vladimir Propp (1895 – 1970). Vladimir Propp describes in his work *Morphology of the Folktale* how the content of Russian fairy tales can be analysed to a more abstract level. Propp describes the different kinds of characters that you will find in a Russian fairytale (Propp 1928, pp 79-80) and he breaks up the fairytales into different parts which he calls *functions.* These proppian functions describe different events in a story, such as when the hero departs from home or the villain is punished (Propp 1928, pp 25). A number of story generators are based on Propp's work (Fairclough & Cunningham 2003) (Peinado & Gervás 2006) (Imabuchi & Ogata 2012).

The output of a procedural plot generator varies between different research projects. In a plot generator like TALE-SPIN, the output is a readable text, generated by means of natural language processing (Meehan 1981). In many projects, the output is an abstract plot that is interpreted by an application, such as a game or a simulation, and presented to the user graphically (Ciarlini et al. 2005) (Crawford 2012). Procedural plot generation can also be used to generate plots for a number of fields. Generated plots can for example be used as help for television writers (Díaz-Agudo et al. 2004). The field most known for generating plots is called *interactive storytelling*, a type of storytelling where the story adjusts to the user's interactions. Interactive storytelling is based on many different fields, such as games, cinema, narratology, mathematics and artificial intelligence (Crawford 2012).

Of course, procedurally generated plots can also be used in games. Some examples, such as the ones related to interactive storytelling, use computer games as case studies for their plot generating systems. There is also research that focuses on the gaming industry and how plot generation can be used in computer games (Onuczko et al. 2006). With Astrogue, the focus has been on making a game and see how procedural content generation, such as plot generation, can be used within it, rather than focusing on the plot generation itself.

Our approach to plot generation is similar to the one used in TALE-SPIN. Our plot generator is character-based; autonomous agents are assigned goals, that they try to reach by performing different actions, interacting with each other and their environment. A similar approach, where the main problem becomes a *planning* problem for the autonomous agents, has been used in various plot generators (Cavazza et al. 2002) (Charles et al. 2003). As mentioned earlier, this approach would fall into the *transformationalist* category. One reason for originally choosing a transformationalist approach over a structuralist approach is that the transformationalist approach can be done on-the-fly, meaning the story can be generated while running game based on the player's actions. Later on, however, we decided to not create the plot on-the-fly, but generate it before the game starts. This decision was made, because generating the plot beforehand lets the generator test the plot for properties such as length and discard it if the plot is not satisfactory. The reason for choosing a character-based plot generator is because it seemed easier to create a simple algorithm, compared to using for example machine-learning techniques that would require a lot of knowledge to be usable, making it less appropriate for a bachelor's thesis project. We also preferred to not use a case-based reasoning approach, since it would need a large case base of pre-written stories to be usable, which our plot generator does not need.

5.2 Results

In this section, our implemented name generator and the plot generator for Astrogue, *AstroPG*, are explained. We describe how the plots are represented and also present how the generated names and plots are integrated into Astrogue.

5.2.1 Markov chain name generator

The name generator, based on Markov chains, is first taught the transition table from a list of names using N-grams. Afterwards, the generator can build names with orders between one and four. A lower order will more often produce a result that is more random, since the transitions are made with limited knowledge of the past. The initial state is randomly chosen among all starting sequences in the training set of names. The names generated are restricted in the way that they have a maximum length. Names also tend to be at least as long as the order entered; however, this is not always the case. Generated names can be profane, but we are not concerned about that. In Table 5.1, all words that can be generated from Astrogue, Bear, and Treasure with a second-order Markov chain are shown. A word starts with the same amount of letters as the order of the Markov chain. The sequence of letters within brackets can be repeated indefinitely. The amount of words that can be generated rises quickly when expanding the database.

Table 5.1: Here we list all possible words that can be generated, using a Markov chain of order 2, from a database with the words *Astrogue*, *Bear*, and *Treasure*. The initial two-letter sequence is arbitrarily chosen.

As-	Be-	Tr-
Astrear	Bear	Trear
Astreasure	Beastrogue	Treastrogue
Astreasure[reasure]-ar	Beasure	Treasure
Astrogue		Treasure[reasure]-ar
Asure		Trogue

5.2.2 Examples of generated names

Table 5.2 illustrates how our name generator can produce names with distinct nationalities when given different sets of name data. The left column presents names generated with a database of American male names. The column to the right consists of names generated from a database with Icelandic names and it is possible to immediately see the difference, since there are letters not used in the English language within the Icelandic names.

Table 5.2: Examples of names that are generated by our name generator with third-order Markov chains, using two different databases.

Database with American names	**Database with Icelandic names**
Waite	Dís
Jiracle	Kir
Soland	Ædísafold
Zareddiells	Liði
Ferdian	Þormar
Rtz	Heiðar

5.2.3 The plot representation

One of the first problems to solve when making a plot generator is the question of how a plot should be represented. This section covers our solution to this problem.

The generated plots are represented as sets of *scenes*, *actors* and *props* (words borrowed from the theater, here collectively called *SAPs*), and a list of *actions* called the *plotline*. The SAPs represent all the different entities in a story. Scenes are the environments where a story takes place, actors are the characters in a story and props are the different items that appear in the story. The actions are represented by sentences containing SAPs, with verbs decided by five different action types; VISIT, MEET, GIVE, TAKE and KILL. These action types are chosen, since we believe they are typical interactions in an action game. The actions take the following forms:

- Actor VISITS Scene
- Actor MEETS Actor
- Actor GIVES Prop to Actor
- Actor TAKES Prop
- Actor KILLS Actor

The SAPs in these actions are in the plotline replaced by their respective names. Examples of actions are:

- Brian VISITS Alcygol
- Brian MEETS Ming
- Ming GIVES Bog to Brian
- Brian TAKES Hairt
- Brian KILLS Ming

A plot is hence, together with the sets of SAPs, a list of these actions — the plotline. An example of a hand-written plot can be seen in Figure 5.2. Note that there can be SAPs in the plot that are never used in the plotline, for example the actor Grandma in the figure. This does not have much relevance in a hand-written plot, but is possible with the plots generated by our plot generator.

In the game, actions are also accompanied by pre-written narrative texts. The narrative texts are written to be read by the player, while the actions are written to be understood by a computer application. The narrative texts are based on their corresponding actions and subsequent actions in the plotline, which makes the story coherent.

5.2.4 The Astrogue Plot Generator (AstroPG)

We call our plot generator *AstroPG*, *The Astrogue Plot Generator*. AstroPG is created as a stand-alone module that can generate abstract plots by itself, independently of the Astrogue game. It is, however, mainly created for and used as a part of Astrogue, where it generates the plots for the game. This section covers how AstroPG generates a plot by creating its scenes, actors, and props and then generating the plotline in an AI simulation.

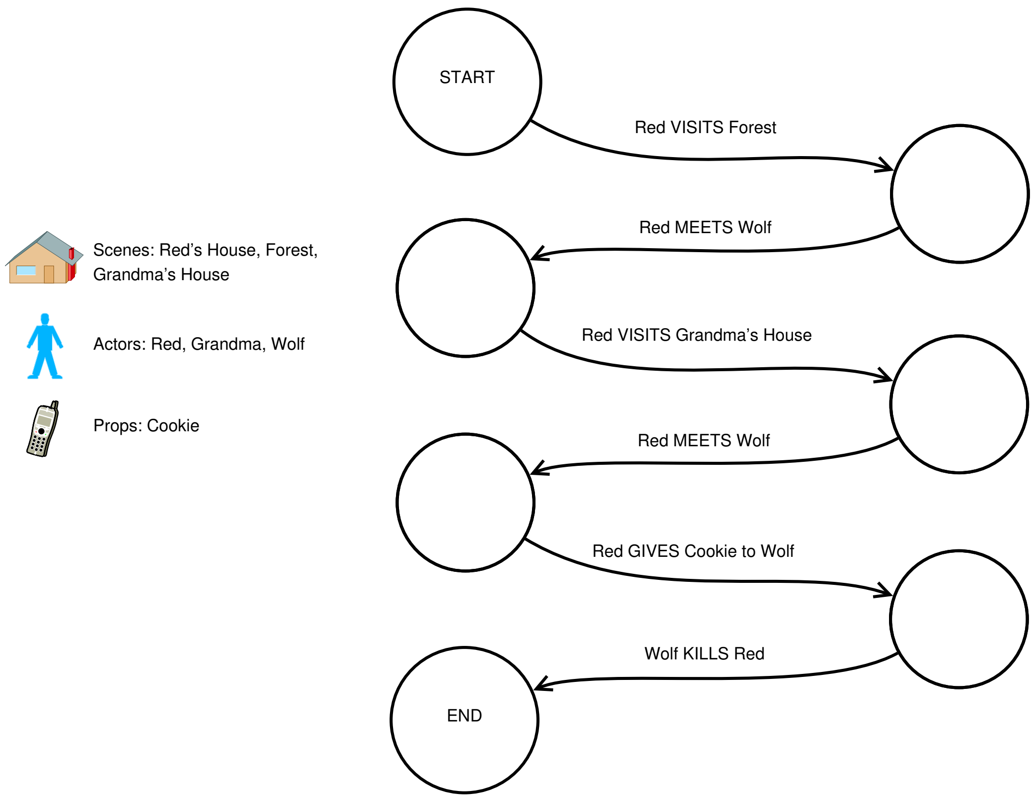


Figure 5.2: This figure shows an example of what a plot can look like using our plot representation. This example is a hand-written version of the classic *Little Red Riding Hood* story.

AstroPG generates the plot in a series of different steps. First, it generates scenes, actors and props for the story, randomising their names. Agents corresponding to actors are then created and assigned different goals that they want to reach. The story is simulated, where the agents perform different actions, until the main agent's goals are reached. The resulting plotline is then represented as a list of actions that include the main actor. A check is performed: if the plotline is too short or long, it is discarded and re-generated. Lastly, for the plotlines that are used in the game, all actions are assigned a corresponding narrative text, where the text for each action also depends on subsequent actions in the plotline. An overview of the process can be seen in Figure 5.3.

Creating scenes, actors and props (SAPs)

AstroPG starts by creating scenes, actors, and props, that represent the different entities and objects in the story. A randomised amount of each type of SAP is generated, with 5-10 scenes, 5-10 props and 3-6 actors; these numbers were chosen because they seemed reasonable for the story of a short game. Each actor is then placed on a scene and each prop is placed either on a scene or in an actor's inventory. This is illustrated in Figure 5.4.

The names for the SAPs are generated using our Markov chain name gener-

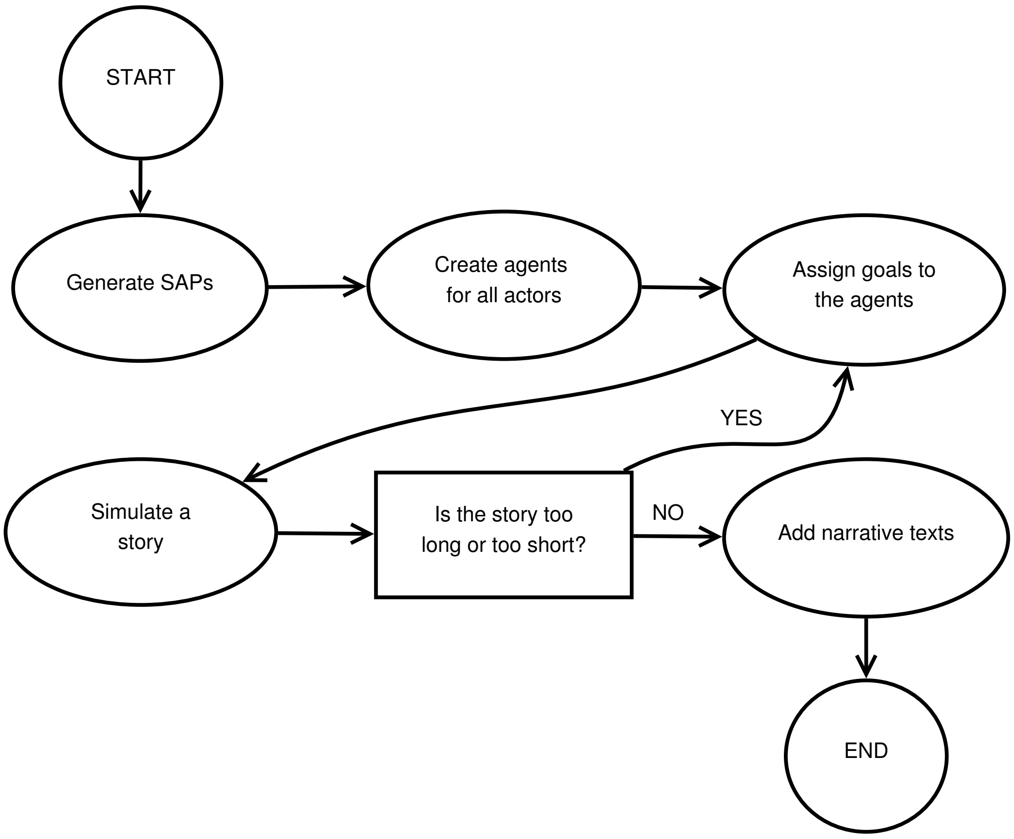

START
Generate SAPs
Create agents for all actors
Assign goals to the agents
YES
Simulate a story
Is the story too long or too short?
NO
Add narrative texts
END

Figure 5.3: A flowchart showing an overview of the process of generating a plot. Note that the story simulation is the main part and where the plotline is actually generated.

ator. The names of the scenes are based on a list of real star names, since the scene equivalents in the game are star systems. The names for the actors are based on a list of American male names. The names of the props are based on a list of everyday objects.

Creating agents and assigning their goals

After the SAPs are generated, autonomous agents corresponding to each actor are created. When the agents are initialised, they are assigned different goals that they will try to achieve during the story simulation. These goals are represented as *conditions*, which are described below. The goals can either be to own a certain prop or that a certain actor is dead. Other goals, such as being at a certain scene, or a set of multiple goals, could also easily be assigned but were deemed to not be relevant for the plot of a simple action game.

An action language for the agents' planning

During the simulation, the agents corresponding to each actor take turn to try to perform different actions using *operators*. Each operator corresponds to a

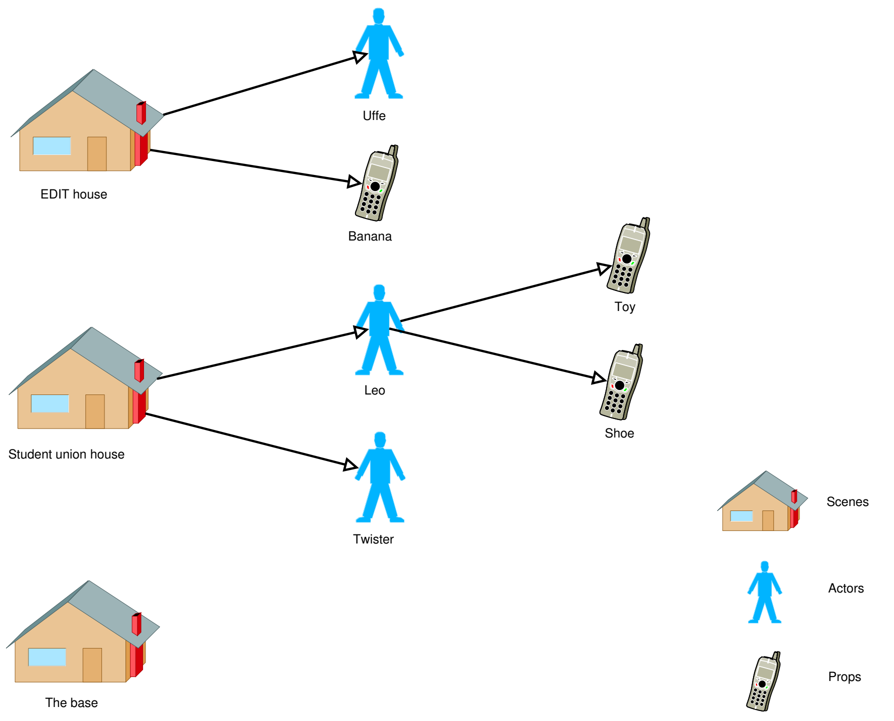


Figure 5.4: This figure shows how actors are placed on scenes, and props are placed both on scenes and in actors' inventories. Note that some generated scenes can be empty, as is the case with *The base* in this figure.

certain action type but is expressed in a different way. The operators are part of a domain-specific *action language* that is used in the simulation.

The action language is inspired by STRIPS (Fikes & Nilsson 1972), a formal language used to express automated planning instances (Fikes & Nilsson 1972) (Bonet & Geffner 1999).

Every agent in our simulation has its own *planning instance*. A planning instance in our action language can be described mathematically as a quadruple:

$$\langle P, O, I, G \rangle$$

The instance's components take on the following meaning:

$\boldsymbol{P}$ is a set of *conditions*, that is, propositional variables of a boolean value.

$\boldsymbol{O}$ is a set of *operators*. Each operator is also a quadruple of the form $\langle \alpha, \beta, \gamma, \delta \rangle$, where the symbols mean:

- **$\boldsymbol{\alpha}$** is a set of conditions that must be true to execute the operation.
- **$\boldsymbol{\beta}$** is a set of conditions that must be false to execute the operation.
- **$\boldsymbol{\gamma}$** is a set of conditions that will be made true when the operation is executed.
- **$\boldsymbol{\delta}$** is a set of conditions that will be made false when the operation is executed.

I is the *initial state*, a set with all conditions that should be set true from the start. All other conditions are set false.

G is the *goal state*, a tuple of the form $\langle N, M \rangle$, where the symbols mean:

> ***N*** is a set of conditions that should be true for the goal to be achieved.
>
> ***M*** is a set of conditions that should be false for the goal to be achieved.

In our action language, there are four different types of conditions:

belongsTo(prop, actor) is true if the given prop is in the given actor's inventory.

lives(actor) is true if the given actor is alive.

isAtLocation(actor, scene) is true if the given actor is at the given location.

isAtSameLocation(actorA, actorB) is true if the given actors are both at the same location.

There are also five operators, all corresponding to the different plot actions described in section 5.2.3: VISIT, MEET, GIVE, TAKE and KILL. As written above, all operators have four sets of conditions: those that must be true and those that must be false for the operation to be performable, as well as those that will be set true and those that will be set false when the operation is performed. The sets of conditions for the different operators are given in Table 5.3. Note that the MEET operation does not have any actual consequences; it is still relevant to the plots, and therefore something that the agents may choose to do.

Simulation of a story

A story is simulated when the agents try to reach their goals taking turn performing operations, using the operators in Table 5.3. Here, we would have preferred to implement an actual planning algorithm, but that would have required significantly more time to develop. Instead, a greedy algorithm is used. During an agent's turn, the agent looks through all the operations it can perform, according to the operations' prerequisites (α and β). The agent will then choose the operation that will fulfill most of the agent's goals (since the agent may only have one goal in our operation, this can at most be one goal). If multiple operations will fulfill the same number of goals, the choice of operation will be randomised among them. This is also true if no operations will fulfill any goals. Pseudocode describing this process can be seen in in Code 5.1.

Table 5.3: This table describes which conditions that are part of the operator sets. The four sets of conditions that belong to an operator are all that is needed to describe that operator.

OPERATOR				
Conditions	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$	$\boldsymbol{\delta}$
VISIT(actor, scene)				
lives(actor)	x			
isAtLocation(actor, scene)		x		
isAtLocation(actor, scene)			x	
isAtLocation(actor, actor.location)				x
MEET(actorA, actorB)				
lives(actorA)	x			
lives(actorB)	x			
isAtSameLocation(actorA, actorB)	x			
GIVE(actorA, actorB, prop)				
lives(actorA)	x			
lives(actorB)	x			
isAtSameLocation(actorA, actorB)	x			
belongsTo(prop, actorA)	x			
belongsTo(prop, actorB)			x	
belongsTo(prop, actorA)				x
TAKE(actor, prop)				
lives(actor)	x			
samePlace(actor, prop)	x			
belongsTo(prop, actor)			x	
isPlacedAt(prop, prop.location)				x
KILL(actorA, actorB)				
lives(actorA)	x			
lives(actorB)	x			
samePlace(actorA, actorB)	x			
isAtLocation(actorB.props, actorB.location)			x	
lives(actorB)				x
belongsTo(actorB.props, actorB)				x

Code 5.1: Pseudocode describing how agents determine which operators are best. An operator is then randomly chosen from these, taking the operators' weights in account.

```
bestOps = new List()
bestOpValue = 0
for op in possible operators:
    opValue = 0
    for cond in op.$\gamma$:
        if cond in agent.$N$:
            opValue++
    for cond in op.$\delta$:
        if cond in agent.$M$:
            opValue++
    if opValue > bestOpValue:
        bestOpValue = opValue
        bestOps = new List()
    if opValue == bestOpValue:
        bestOps.add(op)

return bestOps
```

The operations are also weighted with a pre-determined weight. These make some operations more likely than others when the agent chooses one on random. For example, it is more likely that the agent will meet another actor than go to a new location. A special rule has been added: the main character should not be able to die, so if the active agent would like to perform the KILL operation on the main actor, the weight for that operation is set to zero. There is also the possibility that an agent will perform no action at all during its turn, otherwise the actors would move around too much at random.

For every performed operation including the main actor the corresponding plot action is added to a plotline. This is how the plotline is generated. When the main agent has accomplished all its goals (to kill another actor or get a certain item), the plot is finished. One last check is made before the plot is returned: if the plotline is too short or too long. If it is not in a set interval, the plotline will be re-generated: the SAPs will be kept, but new goals will be generated for the agents. The output of AstroPG is then the generated SAPs and the plotline, which is a list of plot actions.

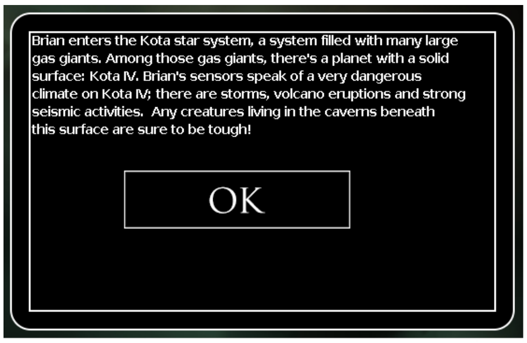
Adding narrative texts to the plot

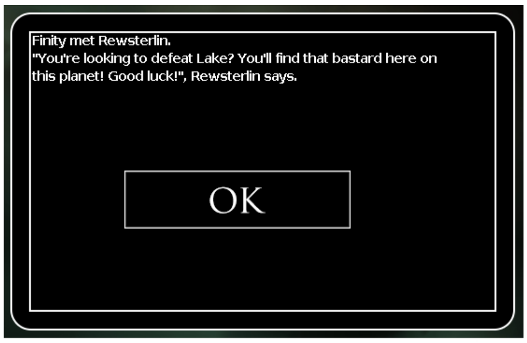
An abstract plot in itself is not of very much interest; the plot also has to be presented in some way. In the game, the plot is presented with narrative texts in popup windows describing the plot actions. Unlike the abstract plots, these narrative texts are not very general, but written for the theme of the game; they mention flying to planets and other space-related topics.

These narrative texts are pre-written. Placeholders in the pre-written texts are replaced with the names of scenes, actors and props. A plot action type is linked to a number of pre-written texts and for each plot action one is randomly picked. The text for a plot action is sometimes chosen based on subsequent actions in the plotline. For example, the text for moving to a certain scene will mention why the main actor moves to that scene, e.g. when the main actor moves there to pick up a prop or kill another actor. Dialogues shown when meeting another actor also mention the main actor's goal and what the main actor has to do next.

5.2.5 Plot integration in Astrogue

In Astrogue, most of the plot is perceived in the overworld mode. AstroPG works as a *state machine*. It keeps track of what action was last performed and it advances the plot when the player performs the next action in the plotline. When the plot is advanced, a popup window with the performed action's corresponding narrative text appears, describing the story to the player. Examples of this can be seen in Figure 5.5. The plot actions seen are *Brian VISITS Kota*, *Finity MEETS Rewsterlin* and *Wells GIVES Hanael to Lint*.


Brian enters the Kota star system, a system filled with many large gas giants. Among those gas giants, there's a planet with a solid surface: Kota IV. Brian's sensors speak of a very dangerous climate on Kota IV; there are storms, volcano eruptions and strong seismic activities. Any creatures living in the caverns beneath this surface are sure to be tough!


Finity met Rewsterlin.
"You're looking to defeat Lake? You'll find that bastard here on this planet! Good luck!", Rewsterlin says.

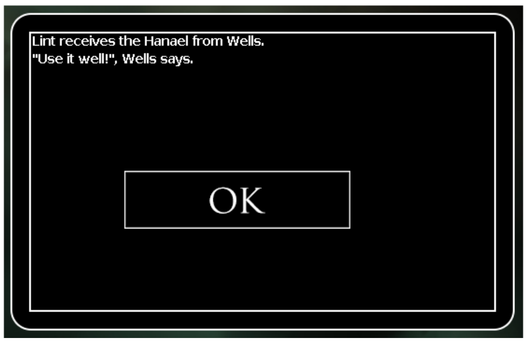

Lint receives the Hanael from Wells.
"Use it well!", Wells says.

Figure 5.5: Screenshots with examples of how the plot is conveyed in the game. Narrative texts, based on the plot actions from the generated plotline, are shown in popup windows during the overworld mode.

The different plot actions correspond to different player actions. To VISIT a scene, the player has to click on a star. When the main actor MEETS, GIVES a prop to, or is GIVEN a prop from another actor, the plot actions are performed automatically. The plot actions TAKE and KILL are both performed in the *dungeon crawl* mode, when landed on a planet. Their respective player actions are picking up a certain item and defeating a certain enemy.

In the overworld mode, neither props nor actors have any graphical representation. The scenes are represented by the stars and all stars correspond to a scene in the plot. In the dungeon crawl mode, both props and actors are graphically represented by item and enemy sprites, which are selected on random from a set of pre-determined sprites. These appear for the player to interact with when the plot is at a TAKE or KILL action. Screenshots showing when an enemy corresponding to a plot actor appears and when an item corresponding to a plot prop appears are shown in Figure 5.6 and Figure 5.7.

5.2.6 Examples of generated plotlines

Below are four examples of generated plotlines from our plot generator AstroPG. They each illustrate different aspects of how the plots can turn out. The first three are only described by their plot actions, while the fourth also has the pre-written narrative texts that are used in the game.

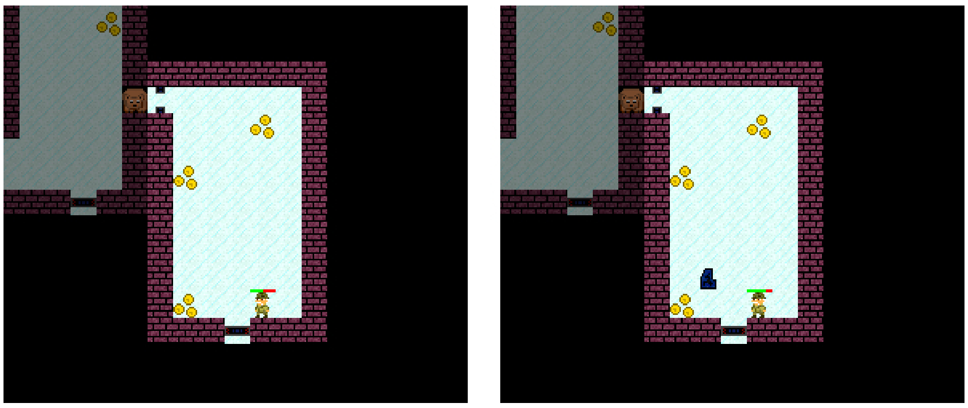


Figure 5.6: Screenshots illustrating when the player can pick up a plot item. The right screenshot shows an item corresponding to a prop in the plot. The left screenshot shows the same room, but without the item. The item only appears when the next action in the plot is a TAKE action.

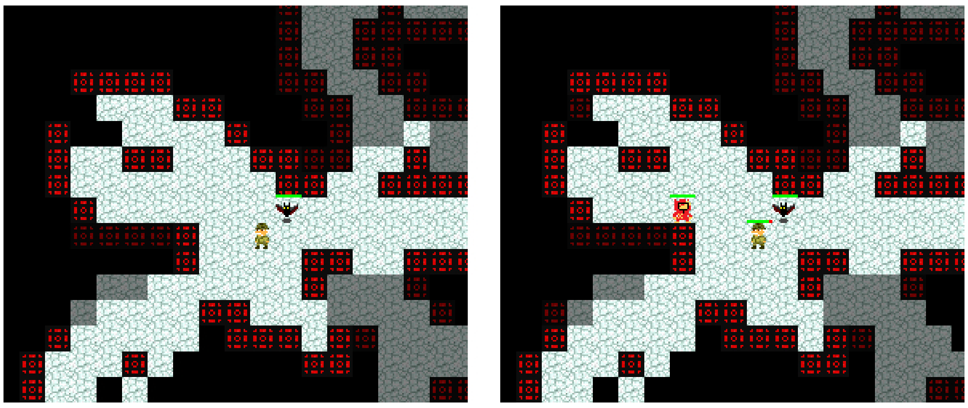


Figure 5.7: Similar to Figure 5.6, these screenshots illustrate how an enemy appears in a dungeon when the next plot action is a KILL action.

- **Plot 1, a short plot:**
  - Brilanden meets Throp
  - Throp meets Mayes
  - Throp kills Mayes
- **Plot 2, an OK plot:**
  - Harring meets Merto
  - Merto takes Shoor
  - Merto visits Kib
  - Merto visits Staban
  - Merto meets Harring

  - Merto gives Shoor to Harring
  - Harring gives Bra to Merto
  - Merto takes Ndool

- **Plot 3, a rather meaningless plot:**
  - Brandar visits Alatrior
  - Brandar visits Ashird
  - Brandar visits Saif
  - Brandar visits Acamalis
  - Brandar visits Amaka
  - Brandar takes Ure

- **Plot 4, an OK plot with added narrative texts:**

  Cleavital meets Elin: Elin met Cleavital. "You're looking to find the Mouscrel? Then, first you'll have to find the Ponglove, hidden somewhere on this planet. Good luck!", Cleavital says.

  Elin takes Ponglove: Deep in the dungeons of the Kaid planet, Elin found the Ponglove.

  Elin gives Ponglove to Cleavital: Elin gives the Ponglove to Cleavital. "Thank you!", Cleavital says.

  Elin visits Matares: Elin enters the Matares star system, a system filled with many large gas giants. Among those gas giants, there's a planet with a solid surface: Matares IV. Elin's sensors speak of a very dangerous climate on Matares IV; there are storms, volcano eruptions and strong seismic activities. Any creatures living in the caverns beneath this surface are sure to be tough!

  Lajos meets Elin: Elin met Lajos. "You're looking to find the Mouscrel? Good luck with that!", Lajos says.

  Kian meets Elin: Elin met Kian. "You're looking to find the Mouscrel? Then you're lucky! It's right here on this planet!", Kian says.

  Elin takes Mouscrel: Deep in the dungeons of the Matares planet, Elin found the Mouscrel.

These plotlines are commented in the discussion section below.

5.3 Discussion

In this section, we discuss the results of our plot and name generator implementations. We examine the quality of the generated plots and names, argue about advantages and disadvantages with our implementations and discuss how things could have been done differently.

As we can see in plot 1 in section 5.2.6, the generated abstract plots are not very exciting. This is mostly due to two factors: abstract plots in themselves are not very interesting and our plotlines are made up of a small number of plot action types. The abstract plots do not feel engaging by themselves, but what is interesting instead is the way they are implemented and presented. In Astrogue, the plots are presented as narrative texts, which correspond to the different plot actions and are added to the plot before it is presented. An example of this can be seen in plot 4 in section 5.2.6.

While we have not performed any real testing, we would still argue that a plot like plot 2 in section 5.2.6 is OK, since what happens is varied and the plot feels appropriate for a simple action game. Still, the plot could be a lot more interesting if more things could happen than the five plot actions; VISIT, MEET, GIVE, TAKE and KILL. On a positive note, the types of plot actions could easily be extended to include a larger number of actions. We believe the action types that were chosen are enough to describe a simple action game, but by adding more actions the plots could be more interesting and many different kinds of plots could be described. Inspiration for these plot actions could be taken from the *proppian functions* (Propp 1928, pp 25) that are often used in plot generators (Fairclough & Cunningham 2003) (Imabuchi & Ogata 2012) (Peinado & Gervás 2006).

Another feature we wanted to add, but did not find enough time for, was relations between actors. These would affect the way the autonomous agents act towards each other during the story simulation in AstroPG. These relations would be similar to the ones in TALE-SPIN where, for example, the animals can be kindly or unkindly disposed against each other (Meehan 1981). Together with these relations, we wanted to add a type of meta-action, where agents could tell each other to do their bidding by performing certain actions, as when a boss asks its employee to perform a certain task. We believe this would make the plots more interesting.

An example of how these relations could work can be seen in Figure 5.8. For example, we can see that Daisy hates Donald. This could in the simulation translate to Daisy being more proponed to kill Donald, or less proponed to give him something. In the figure, we can also see that Smurf is the boss of Woody. This could make Smurf more proponed to use the meta-action described above on Woody, telling Woody to perform another action.

Another thing that could have been added was a better planning algorithm. The greedy algorithm that we implemented is very basic, but all the underlying systems and classes that are needed for it to work took quite some time to get right. This is the main part we would have wanted to do better with AstroPG, but it would have required significantly more time to implement. For example, an algorithm inspired by the original STRIPS planner could have been used (Fikes & Nilsson 1972), or a planning formalism more typically used in interactive storytelling such as using hierarchical task networks (Charles et al. 2003). Hopefully, a better planning algorithm would have helped the problem with plots like plot 3 in section 5.2.6, where too much of the plot is just the main character wandering around meaninglessly.

As AstroPG is implemented now, the main character only has one final goal to achieve. We believe the generated stories would be more interesting if the main character would also have different subgoals to achieve, in order to achieve the final goal. This is typical in many games, for example in The Legend of Zelda (Nintendo Co., Ltd. 1986), where the main character has to collect eight *triforce shards* in order to be able to defeat the final boss.

In interactive storytelling, it is typical for the user to be able to change the direction of the plot, making it non-linear (Cavazza et al. 2002). This is yet

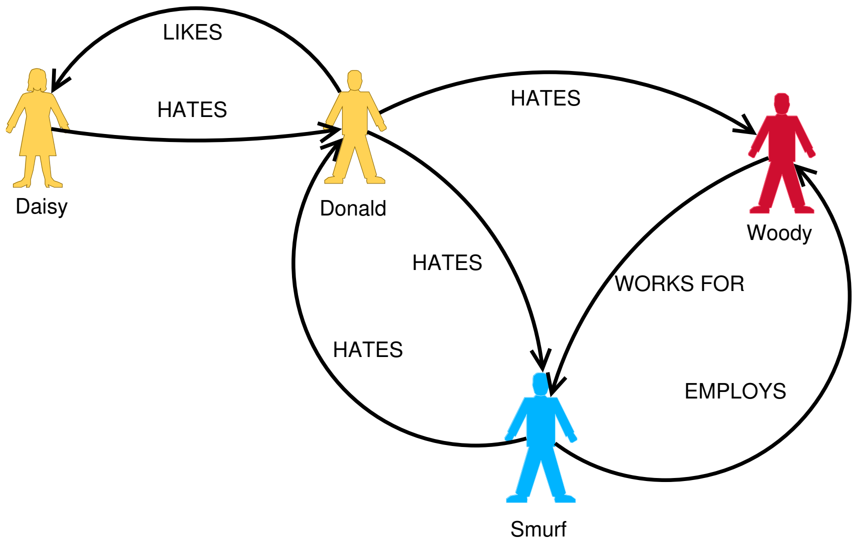


Figure 5.8: This figure shows an example of how relations between characters could work, if implemented. These relations could then affect the way actors act towards each other.

another thing that we wanted to implement, in order to make the game more interesting. In the final version, the plot is represented in the code as a list of plot actions, but the idea was to have it represented as a tree with plot actions as nodes. This is actually how the plot was implemented for a long time, but the iteration of our plot generating algorithm that we finally stuck with only generates linear plots. Creating non-linear plots would probably not require much change in the code, but would of course require more computation time, since all possible paths would need to be simulated.

We are happy with the way our name generator works and we believe it adds much to the plot experience as a whole. Some generated names are quite strange and hard to pronounce such as *Rtz* from Table 5.2, but we believe this works well with the space theme of Astrogue.

Originally, we also wanted to generate the narrative texts that are added to the plot. The idea was to use either Markov chains, as with the name generator, or some sort of natural language processing algorithm. This idea was quickly rejected, since generating coherent texts is hard. The narrative texts that are used now are pre-written. This solution is sub-optimal for a couple of reasons. First, it contradicts one of the purposes with procedural content generation (PCG); you still need to produce a lot of content for it to work well. Secondly, it makes the plots feel less varied, when you can encounter the same wording multiple times in even the same play through, as can be seen in plot 4 in section 5.2.6. The amount of variation linearly depends on the amount of content that is pre-produced, which is not desirable for a PCG algorithm.

Overall, we are still happy with our work, even if the plots are not too interesting. We still believe our system for generating plots is of interest and value, and that the plots are varied enough for our game. We also believe our name generator works well for its purpose and adds to the variety of the plots.

Chapter 6

The final game

In this chapter, we will present the final game as a whole and discuss the results and our choice of methodology.

6.1 Results

This section briefly describes the final game and summarises the results of our project. For further details, see the results sections in chapters 2–5.

The final product of our project is Astrogue: a game within the roguelike genre with procedurally generated levels and plots. At the core of the game is its game engine. The game engine we have implemented uses the entity-component-system design pattern and it is described further in chapter 2. For the game, we have also designed and implemented a menu system and two different game modes: dungeon crawl and overworld. The dungeon crawl mode utilises many systems for game logic, such as a turn management system and a combat system. A method for shadowing the dungeon and a basic enemy AI have also been implemented. These aspects are all essential parts of a roguelike game and they are described further in chapter 3.

Two algorithms have been implemented to generate levels. One creates rectangular rooms and connects them using Delaunay triangulation. It then minimises the amount of corridors using Kruskal's algorithm. The other algorithm uses cellular automata to create cavelike dungeons. We have implemented our own cellular automata rules to achieve this. Both methods ensure that the levels are playable, since every area is reachable. Entities such as enemies and treasures are then placed in the levels according to certain rules. Both level generators accept a seed, which determines which levels will be created. These are described further in chapter 4.

An algorithm for generating plots has also been implemented. It simulates a story using autonomous agents that try to reach individual goals. A language that describes the plots has been implemented. Each plot consists of a list of *plot actions* that describe the events of the plot. The generated plots are ensured not to be too long or too short. We have also implemented a name generator, which utilises Markov chains and lists of already known names to create new ones. These generated names help make the plots feel varied. The plot generator and the name generator are further described in chapter 5.

6.2 Discussion

In this section, we will shortly discuss the final product. We also reflect on what we learned during the project and discuss our choice of methods. For further discussion of our implementations, see the discussion sections in chapters 2–5.

We believe we have implemented all the essential parts of a roguelike game with procedurally generated levels and plots. Our game engine works as it should and we have found it easy to work with, which is what we wanted to accomplish. The feeling of the game is similar to typical roguelike games, with aspects such as turn-based combat and a limited field-of-view, the latter due to our shadowcasting algorithm. We believe all systems that handle game logic have been essential to our game. The engine and the game design is explained further in chapters 2 and 3 respectively.

The most important aspect of our product, however, is its procedural content generators. Our level generators create two-dimensional levels. We believe the levels meet our requirements, such as for playability and the use of a seed. Our plot generator also meets our requirements, such as for length, diversity and coherency. These generators are discussed further in chapters 4 and 5.

Developing a game is very time consuming. While this was expected from the start, just how time consuming the creation of the game itself would be was hard to estimate. Early on in the development phase we had hoped that once the game engine was built, we would only have one person working on the core game during most of the remaining development time. Instead we have had at least two people at any given time working on the core game to be able to implement all the required features for the game to be playable and fun. This has resulted in less manpower avaliable for the content generation aspect of the game than we had originally hoped, even though we always had people working on both the plot and level generators.

One of the hardest parts about the project has been planning for future needs and requirements. Before starting the project we attempted to plan as much as possible, planning what we wanted to see in the game and how much time it would take to implement those things. This, however, has not always worked out perfectly. When developing a new system we have tried to take all the current and future requirements into account. When requirements then change due to unforeseen implementation changes, it can lead to large parts of the previous design being made void, which means extra time needs to be allocated to accomodate for those changes.

Employing agile development techniques has helped us overcome these problems. As mentioned, we planned as much as possible before starting on the implementation. When we could not keep up with our plans, weekly sprint meetings helped us revise our objectives.

We have not encountered any major problems with our choice of programming language, graphics framework or version control system. Java was a good choice of language, since all project members were familiar with it. Lightweight Java Game Library was easy to learn fast and has been a good choice for the simple graphics in Astrogue. Git and GitHub have worked well for version control, especially considering that we wanted our project to be open source.

Overall, we are pleased with our project and we have all learned a lot about procedural content generation and what is needed to create a game. Having many different focuses for the same project, working on the core game, the two level generators and the plot generator at the same time, has been very demanding. In hindsight, we would have preferred a more focused project, since then we would have been able to more deeply examine a certain field, such as level generation. Still, we are satisfied with our work and are very proud of our resulting game.

6.3 Future work

Since our project is implemented using an entity-component-system framework with scalability in mind, the game can be continually worked upon without any major issues. New functionality can be added by adding new systems and components or by improving the already existing systems. Artificial intelligence, for example, is a very broad field and the enemies' AI could be greatly improved.

Future work ideas for game engines, game design, and level and plot generators are mentioned in the discussion sections of chapters 2–5.

Chapter 7

Conclusion

The purpose of this thesis has been to create a game within the roguelike genre, and examine and describe what frameworks and algorithms need to be implemented for such a game. Focus has been on the procedural content generation (PCG) algorithms that are typical for roguelikes. Our game, Astrogue, is a roguelike with procedurally generated levels and plots. Two level generators have been implemented: one that creates rooms and then connects them using Delaunay triangulation and one that creates cavelike levels using cellular automata. Our plot generator, AstroPG, creates a plot from characters' interactions in an AI simulation. We have implemented our own game engine for Astrogue, which utilises the entity-component-system (ECS) design pattern, making the code easy to maintain and modify. We have also implemented various systems for game logic, among them a basic enemy AI, a rendering system and an algorithm for shadowcasting.

Through our case study, we conclude that using ECS suits game development well. We also conclude that our generators work well, but a lot more work could be done to make them more interesting. We believe that PCG serves its purpose in making games more varied and giving them more replay value. We believe more studies on the use of PCG in game development should be done and would be of great value.

Bibliography

Bevilacqua, J. (2013), *Slick2D Game Development*, EBL-Schweitzer, Packt Publishing.

Blizzard Entertainment, Inc. (1996), 'Diablo', PC, Mac.

Blizzard Entertainment, Inc. (2012), 'Diablo III: The world', `http://eu.battle.net/d3/en/game/guide/gameplay/world`. (Accessed: 2014-05-13).

Bonet, B. & Geffner, H. (1999), 'Functional strips: a more general language for planning and problem solving'.

Brownlow, M. (2004), *Game programming golden rules*, Charles River Media, Hingham, Mass.

Cardamone, L., Loiacono, D. & Lanzi, P. L. (2011), Interactive evolution for the procedural generation of tracks in a high-end racing game, *in* 'Proceedings of the 13th annual conference on Genetic and evolutionary computation', ACM, pp. 395–402.

Cavazza, M., Charles, F. & Mead, S. J. (2002), Interacting with virtual characters in interactive storytelling, *in* 'Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1', AAMAS '02, ACM, New York, NY, USA, pp. 318–325.

Cavnar, W. B., Trenkle, J. M. et al. (1994), 'N-gram-based text categorization', *Ann Arbor MI* **48113**(2), 161–175.

Charles, F., Lozano, M., Mead, S. J., Bisquerra, A. F. & Cavazza, M. (2003), Planning formalisms and authoring in interactive storytelling, *in* 'Proceedings of TIDSE', Vol. 3.

Ciarlini, A. E., Pozzer, C. T., Furtado, A. L. & Feijó, B. (2005), A logic-based tool for interactive generation and dramatization of stories, *in* 'Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology', ACM, pp. 133–140.

Compton, K. & Mateas, M. (2006), Procedural level design for platform games., *in* 'AIIDE', pp. 109–111.

Conway, J. (1970), 'The game of life', *Scientific American* **223**(4), 4.

Crawford, C. (2012), *Chris Crawford on Interactive Storytelling*, Pearson Education, chapter Introduction, pp. 26–27.

Croft, D. W. (2004), *Advanced Java Game Programming*, Apress, Berkeley, CA, pp. 279–345.

Dechter, R. & Pearl, J. (1985), 'Generalized best-first search strategies and the optimality of a*', *J. ACM* **32**(3), 505–536.

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

Doherty, M. (2003), 'A software architecture for games', *University of the Pacific Department of Computer Science Research and Project Journal (RAPJ)* **1**(1).

Dormans, J. (2010), Adventures in level design: generating missions and spaces for action adventure games, *in* 'Proceedings of the 2010 Workshop on Procedural Content Generation in Games', ACM, p. 1.

Doull, A. (2008), 'The death of the level designer', `http://pcg.wikidot.com/the-death-of-the-level-designer`. (Accessed: 2014-05-02).

Dunning, T. (1994), *Statistical identification of language*, Computing Research Laboratory, New Mexico State University.

Díaz-Agudo, B., Gervás, P. & Peinado, F. (2004), A case based reasoning approach to story plot generation, *in* P. Funk & P. González Calero, eds, 'Advances in Case-Based Reasoning', Vol. 3155 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 142–156.

Fairclough, C. R. & Cunningham, P. (2003), A multiplayer case based story engine, *in* 'In 4th International Conference on Intelligent Games and Simulation (GAME-ON', pp. 41–46.

Fikes, R. E. & Nilsson, N. J. (1972), 'Strips: A new approach to the application of theorem proving to problem solving', *Artificial intelligence* **2**(3), 189–208.

Folmer, E. (2007), *Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines?*, Vol. 4608, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 66–73.

Gregory, J. (2009), *Game engine architecture*, A K Peters, Wellesley, Mass, pp. 711 – 733.

Grey, D. J. (2013), 'Screw the Berlin Interpretation!', `http://www.gamesofgrey.com/blog/?p=403`. (Accessed: 2014-05-14).

Hafer, M. A. & Weiss, S. F. (1974), 'Word segmentation by letter successor varieties', *Information Storage and Retrieval* **10**(11–12), 371 – 385.

Hannemann, J. & Kiczales, G. (2002), Design pattern implementation in java and aspectj, *in* 'ACM Sigplan Notices', Vol. 37, ACM, pp. 161–173.

Hendrikx, M., Meijer, S., Van Der Velden, J. & Iosup, A. (2013), 'Procedural content generation for games: A survey', *ACM Trans. Multimedia Comput. Commun. Appl.* **9**(1), 1:1–1:22.

Hnaidi, H., Guérin, E., Akkouche, S., Peytavie, A. & Galin, E. (2010), Feature based terrain generation using diffusion equation, *in* 'Computer Graphics Forum', Vol. 29, Wiley Online Library, pp. 2179–2186.

Imabuchi, S. & Ogata, T. (2012), Story generation system based on propp theory as a mechanism in narrative generation system, *in* 'Digital Game and Intelligent Toy Enhanced Learning (DIGITEL), 2012 IEEE Fourth International Conference on', IEEE, pp. 165–167.

Johnson, L., Yannakakis, G. N. & Togelius, J. (2010), Cellular automata for real-time generation of infinite cave levels, *in* 'Proceedings of the 2010 Workshop on Procedural Content Generation in Games', ACM, p. 10.

Jones, R. J., Downey, S. & Mason, J. S. (1997), Continuous speech recognition using syllables., *in* 'Eurospeech'.

Kruskal, Joseph B., J. (1956), 'On the shortest spanning subtree of a graph and the traveling salesman problem', *Proceedings of the American Mathematical Society* **7**(1), pp. 48–50.

Lang, R. (1999), A declarative model for simple narratives, *in* 'Proceedings of the AAAI fall symposium on narrative intelligence', pp. 134–141.

Lee, D. & Schachter, B. (1980), 'Two algorithms for constructing a delaunay triangulation', *International Journal of Computer & Information Sciences* **9**(3), 219–242.

Liapis, A., Yannakakis, G. N. & Togelius, J. (2013), Towards a generic method of evaluating game levels, *in* 'Proceedings of the AAAI Artificial Intelligence for Interactive Digital Entertainment Conference'.

Lischinski, D. (1994), 'Incremental delaunay triangulation', *Graphics gems IV* pp. 47–59.

Lord, R. (2012), 'What is an entity system framework for game development?', `http://www.richardlord.net/blog/what-is-an-entity-framework`. (Accessed: 2014-05-14).

LWJGL (2009*a*), 'Keyboard (lwjgl api)', `http://www.lwjgl.org/javadoc/org/lwjgl/input/Keyboard.html`. (Accessed: 2014-05-14).

LWJGL (2009*b*), 'Mouse (lwjgl api)', `http://www.lwjgl.org/javadoc/org/lwjgl/input/Mouse.html`. (Accessed: 2014-05-15).

Meehan, J. (1981), 'Tale-spin', *Inside computer understanding: Five programs plus miniatures* pp. 197–226.

Meehan, J. R. (1977), Tale-spin, an interactive program that writes stories, *in* 'IJCAI', pp. 91–98.

Miller, G. S. (1986), The definition and rendering of terrain maps, *in* 'ACM SIGGRAPH Computer Graphics', Vol. 20, ACM, pp. 39–48.

Mitchell, M. (1998), *An introduction to genetic algorithms*, MIT press, pp. 2–16.

Mitchell, M., Crutchfield, J. P., Das, R. et al. (1996), Evolving cellular automata with genetic algorithms: A review of recent work, *in* 'Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)'.

Mojang AB (2011), 'Minecraft', PC.

Mossmouth (2008), 'Spelunky', PC.

Musgrave, F. K., Kolb, C. E. & Mace, R. S. (1989), The synthesis and rendering of eroded fractal terrains, *in* 'ACM SIGGRAPH Computer Graphics', Vol. 23, ACM, pp. 41–50.

Nareyek, A. (2004), 'AI in computer games', *Queue* **1**(10), 58.

Nelson, M. J., Roberts, D. L., Isbell Jr, C. L. & Mateas, M. (2006), Reinforcement learning for declarative optimization-based drama management, *in* 'Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems', ACM, pp. 775–782.

Neumann, J. v. & Burks, A. W. (1966), 'Theory of self-reproducing automata'.

Nintendo Co., Ltd. (1986), 'The Legend of Zelda', Nintendo Entertainment System.

Norris, J. R. (1998), *Markov chains*, number 2008, Cambridge university press.

Olsen, J. (2004), 'Realtime procedural terrain generation-realtime synthesis of eroded fractal terrain for use in computer games'.

Onuczko, C., Szafron, D., Schaeffer, J., Cutumisu, M., Siegel, J., Waugh, K. & Schumacher, A. (2006), Automatic story generation for computer role-playing games, *in* 'AIIDE', pp. 147–148.

Oracle (2014), 'Rectangle (java platform se 7)', `http://docs.oracle.com/javase/7/docs/api/java/awt/Rectangle.html`. (Accessed: 2014-05-18).

Patel, A. (2011), 'Amit's a* pages', `http://theory.stanford.edu/~amitp/GameProgramming/`. (Accessed: 2014-05-05).

Peinado, F. & Gervás, P. (2006), 'Evaluation of automatic generation of basic stories', *New Generation Computing* **24**(3), 289–302.

Phigames (2013), 'Tinykeep dungeon generation demo', `http://tinykeep.com/dungen/`. (Accessed: 2014-05-05).

Planetside software (2009), 'Terragen', `http://planetside.co.uk/galleries/tg-in-film`. (Accessed: 2014-05-12).

Prechelt, L. et al. (1999), 'Comparing Java vs. C/C++ efficiency differences to interpersonal differences', *Commun. ACM* **42**(10), 109–112.

Propp, V. (1928), *Morphology of the Folktale*, Publications of the American Folklore Society: Bibliographical and special series, University of Texas Press.

Roth, C., Vorderer, P. & Klimmt, C. (2009), The motivational appeal of interactive storytelling: Towards a dimensional model of the user experience, *in* 'Proceedings of the 2Nd Joint International Conference on Interactive Digital Storytelling: Interactive Storytelling', ICIDS '09, Springer-Verlag, Berlin, Heidelberg, pp. 38–43.

Sgouros, N. M. (1999), 'Dynamic generation, management and resolution of interactive plots', *Artificial Intelligence* **107**(1), 29–62.

Sony Computer Entertainment Inc. (2013), 'Gran Turismo 5 Course Maker', `http://us.gran-turismo.com/us/products/gt5/coursemaker/`. (Accessed: 2014-05-13).

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I. & Postma, E. (2006), 'Adaptive game ai with dynamic scripting', *Machine Learning* **63**(3), 217–248.

The Berlin Interpretation (2008), `http://www.roguebasin.com/index.php?title=Berlin_Interpretation`. (Accessed: 2014-05-14).

Togelius, J., Preuss, M. & Yannakakis, G. N. (2010), Towards multiobjective procedural map generation, *in* 'Proceedings of the 2010 Workshop on Procedural Content Generation in Games', PCGames '10, ACM, New York, NY, USA, pp. 3:1–3:8.

Togelius, J., Yannakakis, G. N., Stanley, K. O. & Browne, C. (2010), Search-based procedural content generation, *in* 'Applications of Evolutionary Computation', Springer, pp. 141–150.

Togelius, J., Yannakakis, G. N., Stanley, K. O. & Browne, C. (2011), 'Search-based procedural content generation: A taxonomy and survey', *Computational Intelligence and AI in Games, IEEE Transactions on* **3**(3), 172–186.

Toy, M., Wichman, G. & Arnold, K. (1980), 'Rogue', Unix.

Valtchanov, V. & Brown, J. A. (2012), Evolving dungeon crawler levels with relative placement, *in* 'Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering', C3S2E '12, ACM, New York, NY, USA, pp. 27–35.

Watson, B., Müller, P., Veryovka, O., Fuller, A., Wonka, P. & Sexton, C. (2008), 'Procedural urban modeling in practice.', *IEEE Computer Graphics and Applications* **28**(3), 18–26.

West, M. (2007), 'Evolve your hierarchy', *http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/* . (Accessed: 2014-05-04).

Whitley, D. (1994), 'A genetic algorithm tutorial', *Statistics and Computing* **4**(2), 65–85.

Wolfram, S. (1983), 'Statistical mechanics of cellular automata', *Reviews of modern physics* **55**(3), 601.