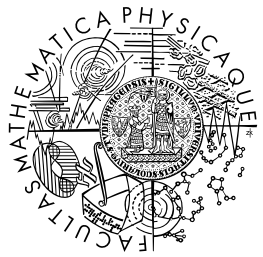

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Filip Sedlák

Procedural generation of levels for a realtime stealth game

Department of Software and Computer Science Education

Supervisor of the master thesis:	Mgr. Vojtěch Černý
Study programme:	Computer Science
Study branch:	Visual Computing and Game Development

Prague 2024

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I dedicate this thesis to my family and friends that helped me and provided me with mental fortitude to finish this journey. I would also like to thank my supervisor Mgr. Vojtěch Černý for his uplifting positivity and invaluable advice.

Title: Procedural generation of levels for a realtime stealth game

Author: Filip Sedlák

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Vojtěch Černý, Department of Software and Computer Science Education

Abstract: Levels in realtime stealth games are often tightly interconnected structures with unique challanges and lock & key puzzles. However, there are no well known instances that attempt to generate these levels procedurally. We implement a small generic 3D realtime stealth game and a level generation algorithm for it. Our game is composed of mechanics commonly found in most stealth centric games. Our generated levels resemble levels from modern stealth games in complexity and interconnectedness. They contain unique challenges for the player. Some generated level sections have shortcomings, but are always playable. In summary, we believe our algorithm succeeded as a proof of concept and can be used in actual stealth games with additional content. Moreover, we contributed a concrete implementation of the cyclic generation algorithm, where the original source is vague on implementation details. Our algorithm can be used to generate levels for stealth games, RPGs, and other genres that make use of lock & key puzzles.

Keywords: procedural content generation level design computer game stealth game

Contents

Introduction **3**

1 Background & Analysis **5**
1.1 Stealth Games . . . 5
1.2 Avatar Challenges . . . 6
1.3 Avatar Means . . . 7
1.4 Level Analysis . . . 8
1.4.1 Bank . . . 9
1.4.2 Room Structure . . . 11
1.5 Cyclic generation . . . 15
1.5.1 Locks & Keys . . . 17
1.5.2 Graph & Patterns . . . 18
1.5.3 Drawing a graph into a grid . . . 18
1.5.4 Tilemaps . . . 19

2 Design **20**
2.1 Procedural Generation . . . 20
2.2 The Game . . . 22
2.2.1 Player Camera . . . 22
2.2.2 Guards . . . 23
2.2.3 Security measures . . . 23
2.2.4 Environmental hazards . . . 24
2.2.5 Movement . . . 24
2.2.6 Cover Systems & Visibility . . . 25
2.2.7 Concealment . . . 25
2.2.8 Tools . . . 27
2.2.9 Locks & Keys . . . 27
2.3 Our Algorithm . . . 28
2.3.1 Graph Drawer . . . 28
2.3.2 Graph Generator . . . 31
2.3.3 Map Builder . . . 35

3 Implementation **39**
3.1 Project 39
3.2 Rules and Patterns 43

4 Results & Discussion **46**
4.1 Level Complexity 46
4.2 Pattern Application 49
4.3 Map Builder 51

Conclusion **53**

Bibliography **54**

Introduction

The stealth genre of videogames revolves around hiding from enemies and achieving a goal undetected. Levels in such games are characterized by interconnectedness and multiple ways in which the player can traverse them. Furthermore, the player is often presented with locked doors and other 'lock & key' style puzzles on the way. On a lower level, stealth mechanics divide these spaces between safe zones and danger zones manifesting in unique challenges.

In summary, levels for this genre are complex and unique. Therefore, they are an interesting subject for procedural generation. Very few stealth games generate their levels though. The most known game is *Invisible, Inc.* [1], however, that is a turn-based game. No popular 3D real-time game does this (such as *Tom Clancy's Splinter Cell®* [2], *HITMAN* [3] or *Dishonored* [4]). So we have chosen to attempt to create an algorithm to generate levels for a 3D real-time stealth game.

As a part of this thesis, we build our own small game with stealth mechanics. This gives us complete control over the generated levels and available mechanics. The game is also implemented in a generic way, so the generated levels may serve as a good base for any stealth game.

The generation itself is based on the cyclic generation algorithm presented in "Procedural generation in game design" [5]. While this source provides the basis for an algorithm that exactly generates interconnected levels with lock & key puzzles, there are a lot of unspecified parts where we needed to complete the design for our purposes. Most notably, extending the concept into 3D space.

In the first chapter, we discuss what exactly characterizes stealth games and we dissect some of the popular levels. Then, we introduce the idea of procedural generation and available algorithms. In the second chapter, we describe the design of the generation algorithm and the game we created for it. The third chapter breaks down implementation details of the game and the algorithm. In the final chapter, we conclude our work and summarize the results.

Chapter 1

Background & Analysis

In this chapter, we look at what stealth games are and what characterizes them from the point of gameplay mechanics and level structure.

1.1 Stealth Games

Videogame genres are often difficult to define due to their equivocal nature. [6] The word stealth can be defined as "a cautious, unobtrusive, and secretive way of moving or proceeding intended to avoid detection" [7], and games centered around stealth mostly focus on achieving a goal while avoiding detection by enemies.[8] Khatib [9], however, provides us with a good, in depth analysis of stealth centric games. He mentions and explains important aspects of these games, namely **avatar means** and **avatar challenges**. We rely on these concepts in the following sections.

This thesis focuses on 3D real-time stealth games, meaning 3D stealth games that don't divide their gameplay into turns or rely on pausing the game as a core game mechanic. The most popular games of this description include titles: *Tom Clancy's Splinter Cell®* [2], *Thief™: The Dark Project* [10], *Dishonored* [4], *HITMAN* [3] and *METAL GEAR SOLID* [11].

1.2 Avatar Challenges

Avatar challenges are dangers and obstacles in players' path. They are a vital part of the level structure in a game.

- **Guards** are the main danger players encounter. [12] They can usually detect players via visual or auditory senses. Depending on the game, they mostly attack the player, trigger game over or engage in a more complex AI behavior. In *Tom Clancy's Splinter Cell®*, they seek an angle to shoot the player. In *Dishonored*, they can run towards an alarm to trigger it.
- **Environmental Hazards** range from deadly traps (*Dishonored*) to broken glass on the ground that makes loud sound while stepping on it (*Thief* [13]) or even hungry rats, like in Figure 1.1.
- **Security Measures** trigger an alarm, alerting the entire level or make the mission harder. These include laser beams (*Tom Clancy's Splinter Cell Chaos Theory®* [14]) or security cameras (*METAL GEAR SOLID*).
- **Alarm State** is triggered by security measures or guards. This state is a result of players' failure at being stealthy. This ranges from triggering game over to transitioning into the action genre. As we are focusing on the level generation, this part is not important for us. Although, in *Tom Clancy's Splinter Cell Chaos Theory®* guards move objects around and block paths to dynamically change the level based on the alarm state, but that is beyond the scope of this thesis.

Figure 1.1 Rats in *Dishonored* that try to eat the player if he gets too close.

1.3 Avatar Means

Avatar means describe various actions and options players' avatar has at their disposal. Levels are often designed around these means, giving players opportunities to take advantage of them.

- **Avatar Characteristics** mostly describe aesthetic 'flavor' of the avatar and are not important for this thesis.
- **Movement** is very important, as the whole point of these games is to move throughout the level without alerting enemies. To this end, all of the aforementioned games have at least 2 modes of movement. Slow and stealthy or fast and loud. Some games have more modes, like prone movement (*METAL GEAR SOLID V: THE PHANTOM PAIN* [15]), continuous spectrum between slow and fast movement (*Tom Clancy's Splinter Cell Chaos Theory®*), or even teleportation (*Dishonored*).
- **Cover systems** allow players to hide behind objects to avoid visual detection. In older games, players can simply position themselves behind these objects, *Thief™: The Dark Project* for instance. Some newer games implement it as a standalone mechanic, allowing players to move from cover to cover with a press of a button, like in *Tom Clancy's Splinter Cell Blacklist* [16].
- **Visibility** of the player is determined by the implementation of enemies' vision. In *Dishonored* [4], players can peak around corners without being seen. In *Tom Clancy's Splinter Cell Blacklist*, enemies cast rays to different joints on avatar's body to compute how much of the body they can see.
- **Concealment** determines how easy it is for enemies to see the player in plain sight. While hiding behind a cover breaks the line of sight, stealth games often implement other mechanics, such as light and shadow (*Thief™: The Dark Project*), camouflage (*METAL GEAR SOLID 3: Snake Eater* [17]) or disguise (*HITMAN*). These are often accompanied by places in the level that increase their effectiveness. For example, dark areas, thick bushes or groups of people to blend in.
- **Tools** are different items and abilities that players can utilize. One of the most common items is a gun. From killing enemies to shooting out lights, guns have often many different uses. More specific gadgets include sticky cameras (*Tom Clancy's Splinter Cell Chaos Theory®*), teleportation and magic (*Dishonored*), or rope arrows (*Thief™: The Dark Project*).

1.4 Level Analysis

In this section, we take a look at levels in various games and analyze how they are structured from the perspective of stealth.

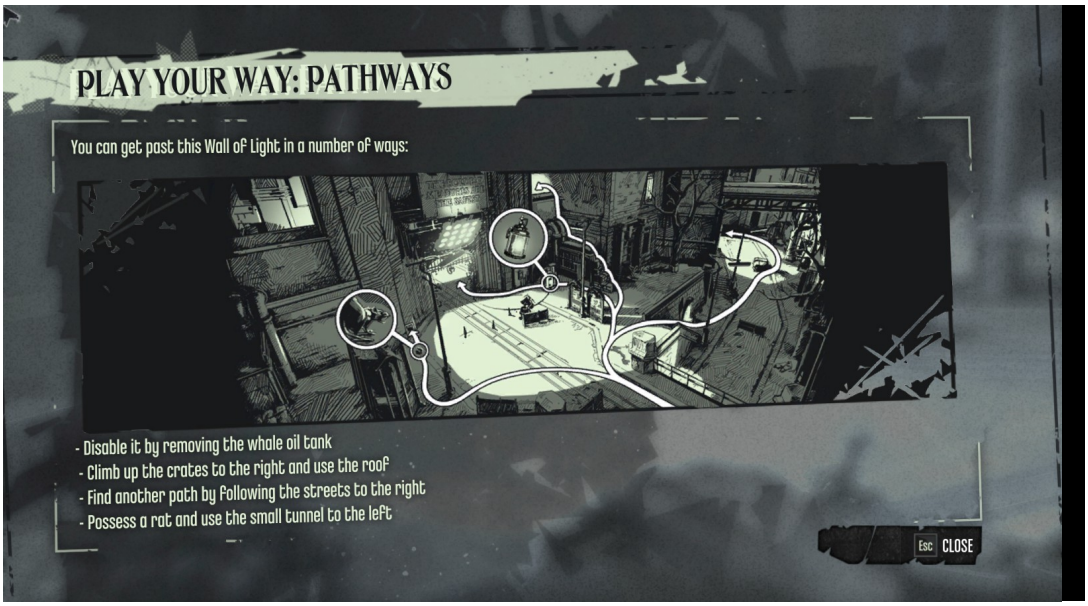

Figure 1.2 An example of different paths players can take in one of the starting levels of *Dishonored*.

Figure 1.2 nicely shows the base concept of stealth levels. Players are met with a guarded obstacle in their path. They are also presented with alternative routes that are less obvious or require skill or a tool to proceed. Perception, skillfulness, and clever thinking are often rewarded by revelation of unique ways of progression.

In this case, a wall of light kills everything that tries to pass through it unauthorized. The first and the most obvious choice is to unplug its power source and pass through it directly. That is dangerous, however, as enemies are guarding the area. The second, less obvious option is to climb the boxes to get onto higher ground, remaining undetected. The third option is to take a completely different path altogether through the streets. The final option is to use the small tunnel to the left. This requires an ability to possess a rat that players may or may not have unlocked.

1.4.1 Bank

One of the most popular levels from the Splinter Cell games is **Bank** from *Tom Clancy's Splinter Cell Chaos Theory®*. We choose Splinter Cell due to the game's sole focus on classic stealth, where as many other titles mix their gameplay with action, or focus on social stealth. The map in the game is not very detailed or comprehensive, so we have redrawn this level into a graph form to better analyze the layout, as shown in Figure 1.3.

The first thing to note is, that this map has multiple floors and an outside garden. From the player's spawn, the bank can be entered through the reception (very dangerous path) or through the roof (less obvious path). The player also encounters a fire exit, which can be opened only from the inside.

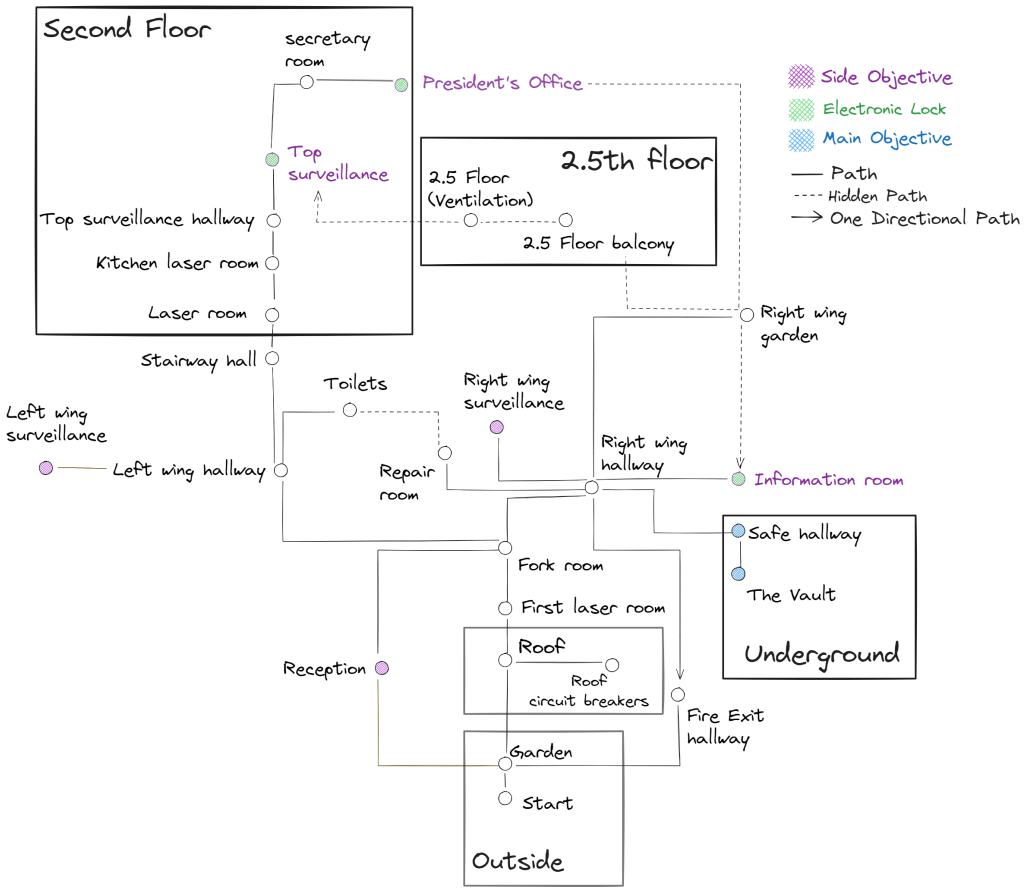

Figure 1.3 A graph drawing of the Bank level from *Tom Clancy's Splinter Cell Chaos Theory®*

When inside, the player is tasked to seek three electronic locks in three separate rooms to unlock the path to the main objective, the vault. The inside of the bank can be divided into three main parts. The **left wing**, the **right wing**, and the **upper floor**. There is a staircase connecting the two floors. In addition, there are two hidden one-way paths, one leading to the upper floor and the other back to the lower floor. One objective is in the right wing and two objectives lay on the upper floor.

Starting from the Fork room, the player has a choice to go to the right wing or the left wing. They are connected via a hidden path in a form of a vent that the player can crawl through. The three rooms containing the electronic locks are themselves blocked by locked doors. The player can either take a chance and hack the keypads, or find the key codes in different parts of the level. Figure 1.4 illustrates what this lock & key relationship looks like.

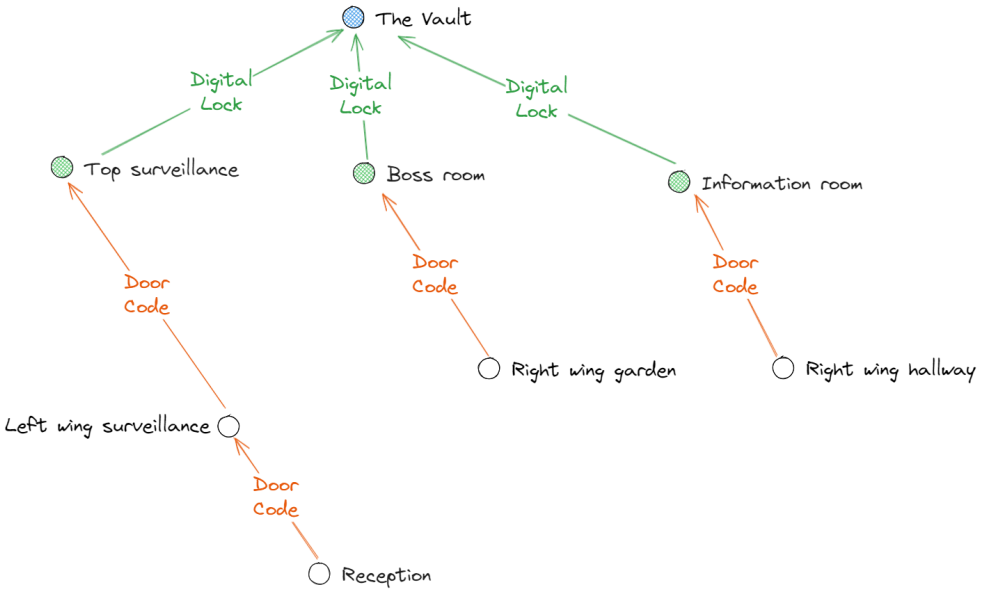

Figure 1.4 A graph representing lock & key relationships in the Bank level.

The key takeaway here is the interconnectedness of the level. We can see cycles in the graph that imply different possible routes of traversing this mission. There are patterns, however, that make these choices more interesting. This includes hidden paths devoid of danger or locked rooms requiring a key to proceed. For example, the player is presented with a choice to test his skills and go through the guarded reception or be observant and take the hidden path to the roof.

When going to the roof, the player encounters a locked fire exit. With the main objective accomplished, he may use this exit to get out of the building safely to the extraction. The player can discover the hidden vent connecting the right and the left wing. This helps with faster and safer traversal when searching for keys to unlock the keypads. Players are stopped by obstacles and therefore are motivated to use these connections in search to remove or bypass the obstacles.

1.4.2 Room Structure

Now we look at the structure of the rooms themselves. Another useful concept Khatib [9] mentions is the idea of **refuge** and **prospect** spaces. Or safe and dangerous spaces respectively. In summary, the game flow of stealth games is often divided into 2 states. The first state is planning, where the player observes the environment and forms a plan on how to proceed. This involves surveying guard patrol paths or noticing environmental hazards. The second state is execution, where the player executes the plan, moving to the next location. [18]

Whether avoiding danger or killing guards, this part challenges player's skills. Consequently, the level can be also divided into the safe spaces for the planning state and danger spaces for the execution state. Figure 1.5 shows a simple implementation of this concept. In practice, we can see a similar hallway in *Thief™ II: The Metal Age* [19] in Figure 1.6. Only, instead of shadowy alcoves, there are small dark rooms on the sides with some curiosities inside.

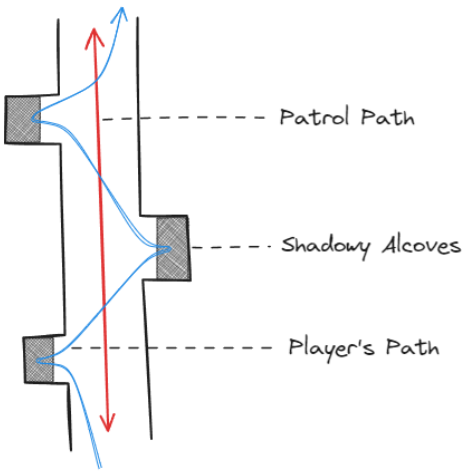

Figure 1.5 This is an illustration of a very simple stealth section. It demonstrates alternation of safety and danger or planning and execution.

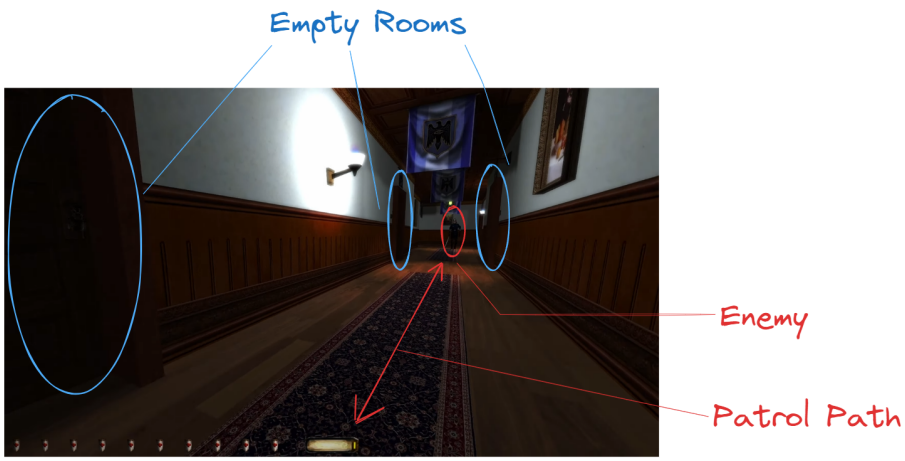

Figure 1.6 The level section from *Thief™ II: The Metal Age*. A simple hallway with rooms on the sides for refuge. The guard is patrolling back and forth.

Figure 1.7 shows a room from the Bank level. It consists of lasers in the middle and a guard patrolling through them. The guard carries an electronic beacon disabling the lasers around him as he walks. Figure 1.8 highlights the division of this room into safe and dangerous spaces. The player has multiple possibilities. First, follow the guard closely in his electronic shadow. Second, kill the guard and carry his body. Third, the player can slip to the right cover for easier traversal. Fourth, it is possible to avoid the danger completely by climbing the ledge to the left in the darkness. The player needs to use night vision to see this.

Figure 1.7 A screenshot of a room from the Bank level. The lasers are dynamically disabled around the patrolling guard.

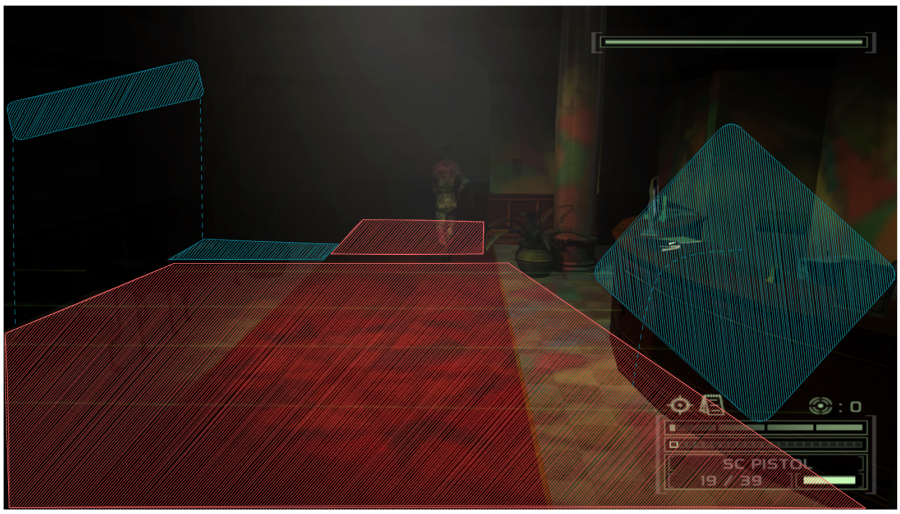

Figure 1.8 A screenshot of a room from the Bank level. This image highlights safe (blue) and dangerous (red) zones. On the left, there is a hidden ledge in the darkness. It is visible with nightvision.

However, refuge spaces can take many forms. Figure 1.9 displays an example of a room with pillars as refuge. In Figure 1.10 we can see guards being preoccupied by their computer screens. The player does not alarm them unless he walks very close to them or makes a bunch of noise. This makes a big part of the room a refuge space in spite of it being lit up and without cover. Another interesting thing to note here is that one of the computers holds a side objective. While the player's avatar has a tool for remote hacking at his disposal, it is not uncommon that the player has to distract an enemy to access an objective.

Figure 1.9 An example of a *Dishonored* room using cover of pillars for refuge.

Figure 1.10 An example of a *Tom Clancy's Splinter Cell Chaos Theory®* room with guards preoccupied by staring into computer screens.

1.5 Cyclic generation

With a better understanding of stealth games, let us look at the algorithm we use as the basis for our generation. Chapter 9 of "Procedural generation in game design" [5] describes the cyclic generation algorithm. The author describes the algorithm in three steps. The first step is applying patterns on a graph via transformational graph grammars, as seen in Figure 1.11. The second step is mapping the graph onto a grid. The third step is transforming the grid into tiles and populating the space. This is illustrated in Figure 1.12. Finally, Figure 1.13 displays an example of a generated level from *Unexplored* [20].

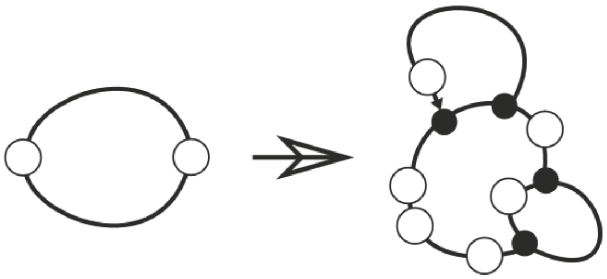

Figure 1.11 The first step of the cyclic generation algorithm, the graph transformation.

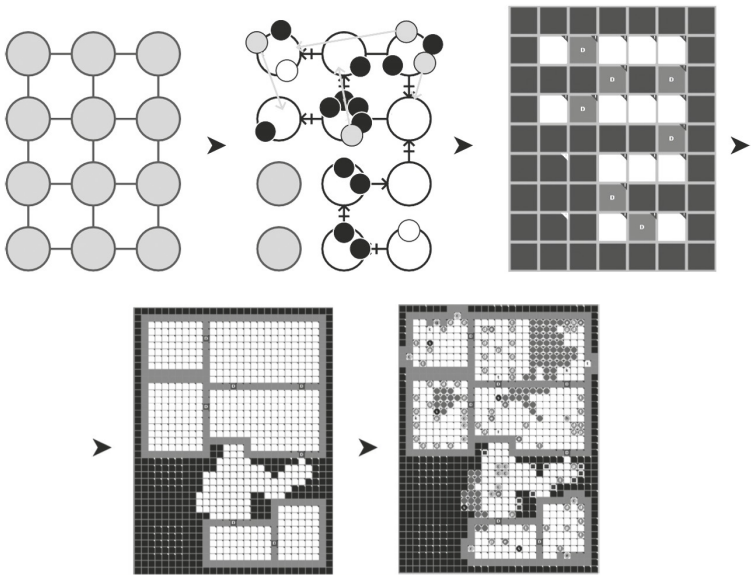

Figure 1.12 The second and the third step of the cyclic generation algorithm, graph drawing and tile building.

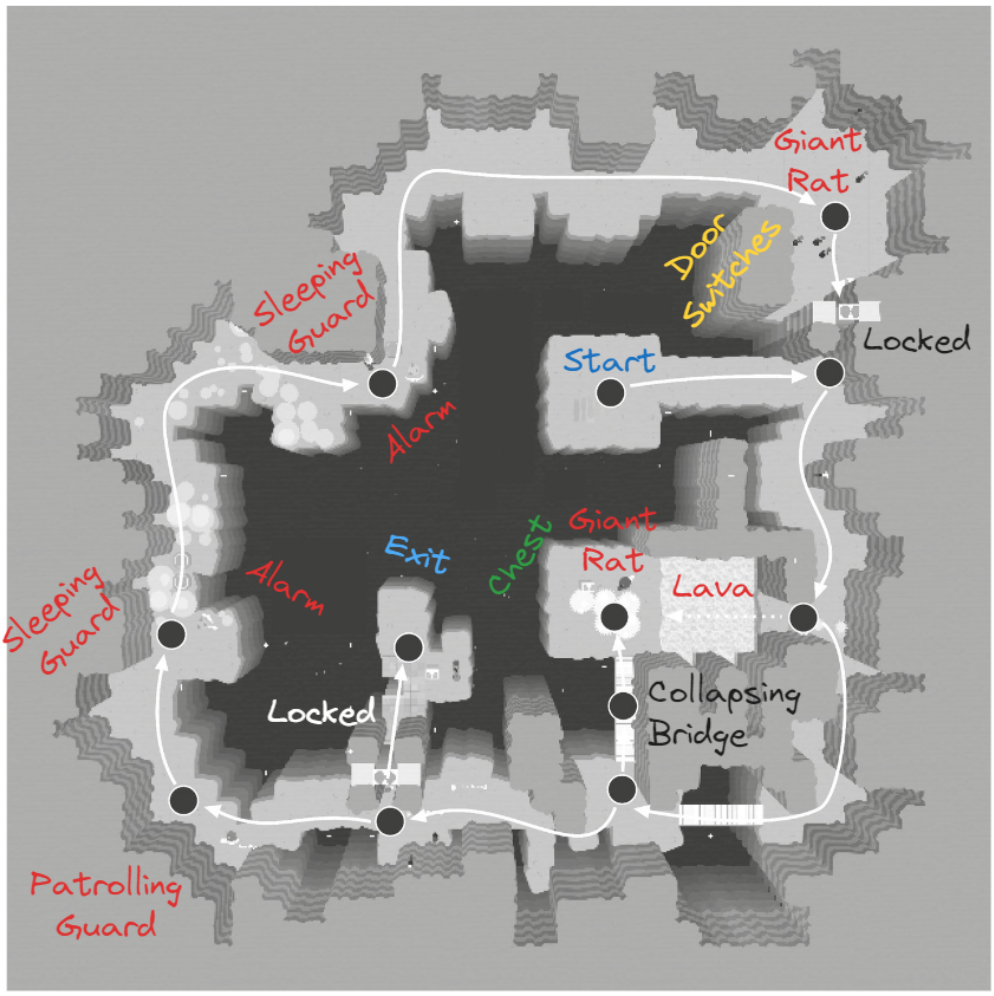

Figure 1.13 An example of an *Unexplored* level generated by the cyclic generation algorithm.

1.5.1 Locks & Keys

An important part of this process is the placement of locks & keys. As the name suggests, locks prevent players from progressing and keys remove these locks. The most obvious implementation are locked doors and literal keys. Having said that, this concept governs much more abstract implementations. For instance, a lock can be anything that prevents progression, like deadly traps, and a key can be a trap disarming kit. Or a lock can be a group of enemies and a key can be ammo for the player's gun. Locks also don't have to completely prevent the player's progress, only make it much more difficult. The book dedicates an entire section to describing types of locks and keys.

To summarize, locks can be:

- **Permanent** - stays unlocked
- **Reversible** - can be locked again
- **Temporary** - unlocked only for a time
- **Collapsing** - allows passage only once
- **Valves** - traversable in one direction
- **Asymmetrical** - unlocked from one direction, but traversable in both
- **Safe** - has a solution
- **Unsafe** - does not have to have a solution

Keys can be:

- **Particular** - unlocks specific locks
- **Nonparticular** - multipurpose
- **Consumed** - destroyed upon use
- **Persistent** - can be used multiple times
- **Fixed in Place** - does not change location
- **Not Fixed in Place** - can change location

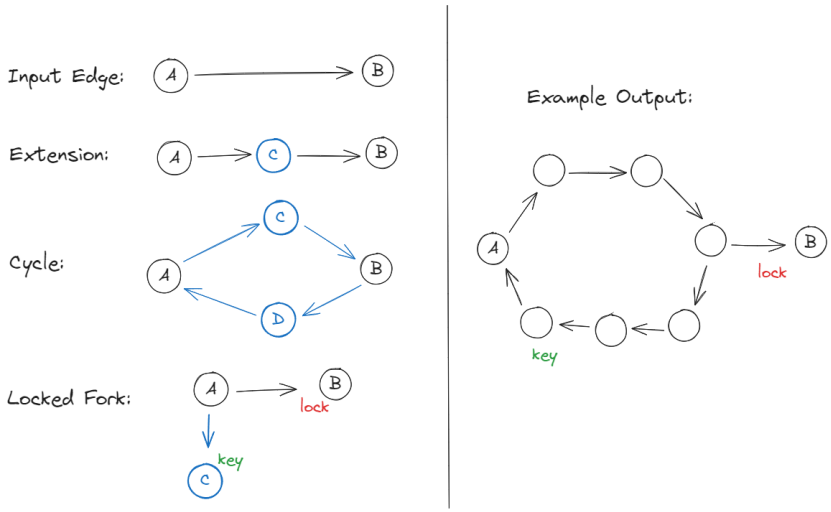

Figure 1.14 An example of graph transformation in the cyclic generation algorithm.

1.5.2 Graph & Patterns

The algorithm for *Unexplored* starts with a simple graph. Then, it uses different rules to transform the graph, simlarly to Graph-free graph grammars [21]. Examples are displayed by Figure 1.14. These rules take a simple edge and simply extend it or create a cycle or a fork. But on a higher level, these rules are used to implement more complex patterns shown in Figure 1.15. The author is unfortunately vague on what is the exact relation between rules and patterns.

1.5.3 Drawing a graph into a grid

For the tilemaps to be constructed, the graph is first drawn into a grid. This is not very well explained, however. The author mentions:

> *"Unfortunately, there is no standard solution to this problem. For Unexplored, we solved the issue by first creating a graph that is laid out as a grid and stored the topological information in the nodes' attributes. All graph transformations are then performed on this graph."* [5]

The Figure 1.12 also implies this approach. But this is far from a specific solution. It raises more questions then it does answer. What is 'topologial information'? How are cycles added into a grid? How are edges extended and new paths created? What if a rule fails and can't add side path to form a cycle?

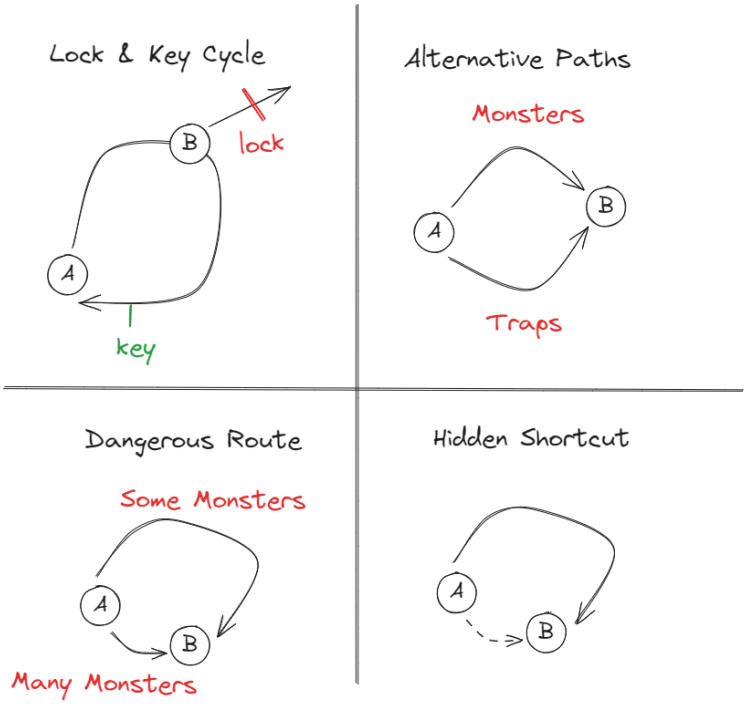

Figure 1.15 Instances of patterns in the cyclic generation algorithm.

1.5.4 Tilemaps

The level is constructed in tiles. Having a graph drawn into a grid, the algorithm starts with creating a tilemap with the same resolution. Then, it increases the resolution to create more detailed structure, as seen in Figure 1.12. Finally, the level is populated. In author's words:

> *"During subsequent steps in the generation process, transformation rules, much like the graph transformation rules discussed in an earlier section, are used to expand rooms, add features as dictated by the graph, and decorate dungeons."* [5]

These 'transformation rules' for expanding rooms are an important detail, yet they are not explained further. So we had to fill in the gaps ourselves.

Chapter 2

Design

2.1 Procedural Generation

There are many algorithms to procedurally generate levels for games. For instance, "Procedural generation of dungeons" [22] gives us a synopsis for some useful algorithms. As we have discussed in the previous chapter, stealth levels are well represented in a graph form highlighting interconnectedness. For this purpose, we will be using ideas from another good source: "Procedural generation in game design" [5]. The stealth levels are to some point reminiscent of metroidvania levels and roguelike levels with their lock & key mechanics. The difference here being that a stealth level is usually a 3D tightly connected structure instead of distant loosely connected parts. And of course, the different implementations of locks, keys and other challenges.

To this end, we choose to use the idea of the cyclic generation algorithm [5]. It gives us exactly what we need. It creates a level in a graph form containing cycles and lock & key patterns. This algorithm was originally used for *Unexplored*. The authors explain that it was meant to be used for roguelikes, but has the potential to be used in variety of different genres. In many traditional roguelikes levels are created in branching tree-like form. This causes a big amount of unnecessary backtracking. Some developers tried to resolve this issue by adding random edges connecting branches of the tree, like in *Brogue* [23]. The cyclic generation proposes to start working with cycles from the beginning, incorporating more interesting and varied patterns for lock & key puzzles.

As for the stealth games, there are not many titles that attempt procedural generation. Most notably, *Invisible, Inc.* [1] implements procedural generation, but the game is 2D and turn based. The level design has often linear or branching form, suffering from the same issues as the aforementioned roguelikes. An example of a level can be seen in Figure 2.1 along with its graph structure and the branching paths in it.

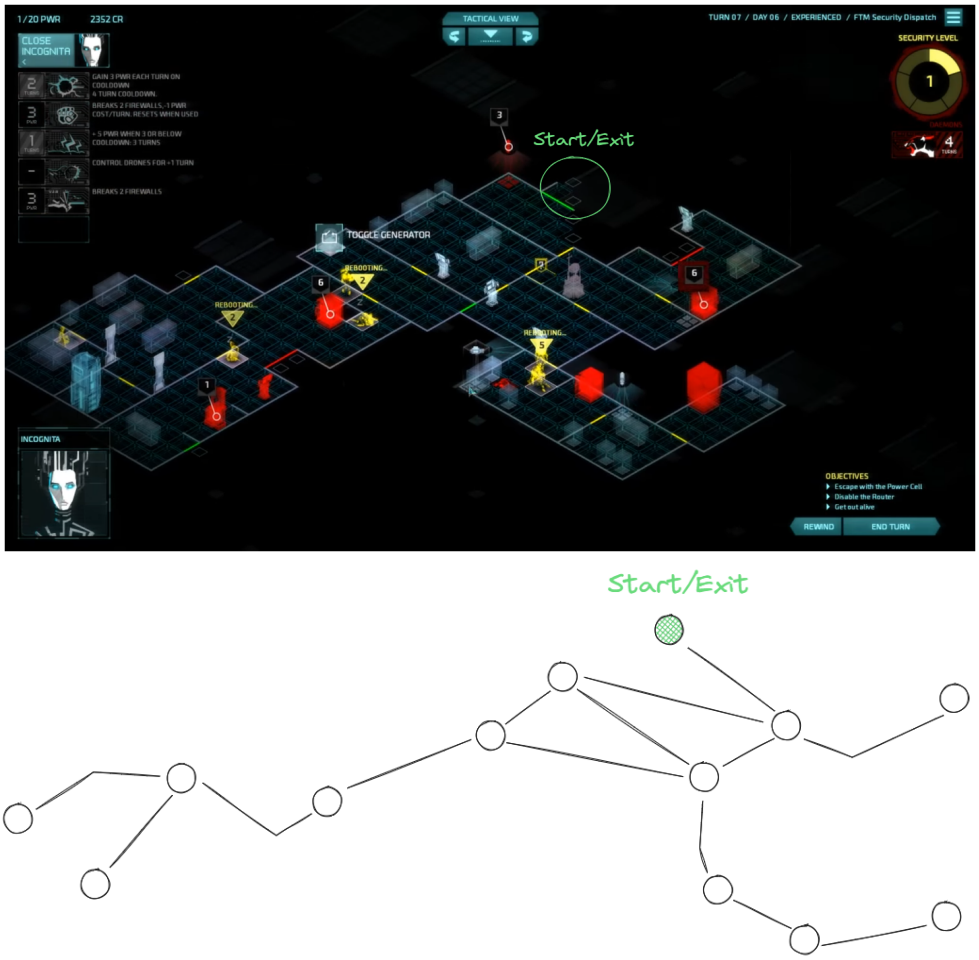

Figure 2.1 An example of a level section from *Invisible, Inc.* and its graph representation.

2.2 The Game

Before we start generating levels, we need to choose a game we are generating for. The obvious requirements are for the game to have an option of importing levels or have API for level building. Many of the popular titles do not have this, as their selling point are the complex carefully crafted levels made by the developers. One game that has such option is *The Dark Mod* [24], a fan made project based on the Thief series. But even so, we choose to develop our own game for multiple reasons. Firstly, we avoid any issues and constraints that might be caused by relying on a third party fan-made project. Secondly, we have full control over the game's mechanics and can make it generic enough to be mappable to many other stealth centric games.

2.2.1 Player Camera

As the game is supposed to be generic, we choose to implement first person, third person and top-down view, so the levels can be seen from any perspective. Players can switch camera mode on the fly. Figure 2.2 demonstrates these camera modes.

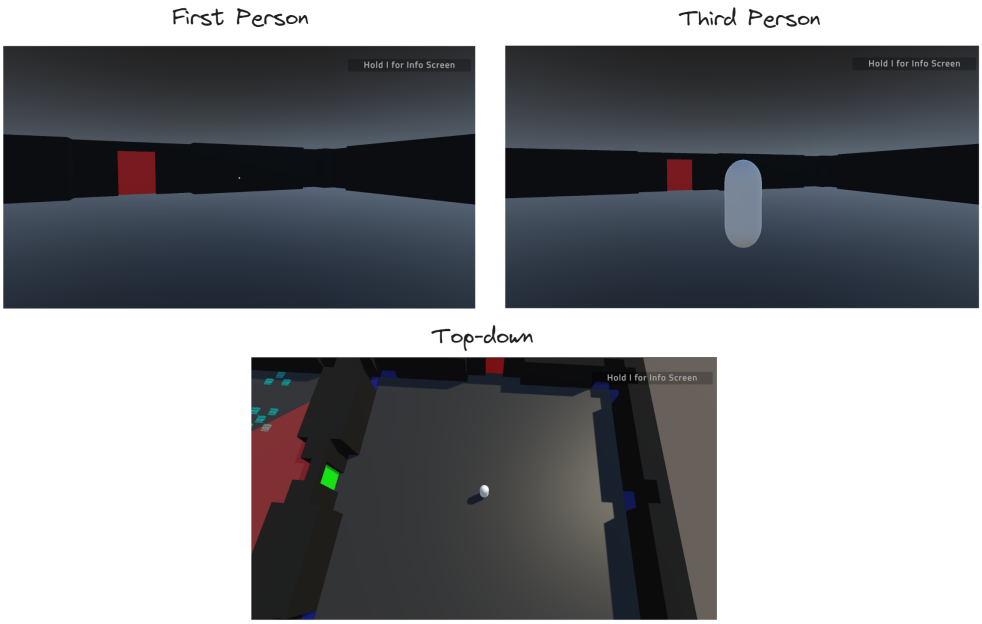

Figure 2.2 Camera modes implemented in our game.

2.2.2 Guards

Enemies are implemented as simple guards patrolling or guarding an area. As with most stealth games, they have implemented visual and auditory senses. Perceiving sounds makes the guards go investigate the area, then return to their usual routine. Visual perception causes guards to run at the player, killing him when they are close. We choose not to implement any complex AI and alarm states, as this is beyond the scope of this thesis and does not influence level design that much. Figure 2.3 shows a guard with a view-cone representing his sight.

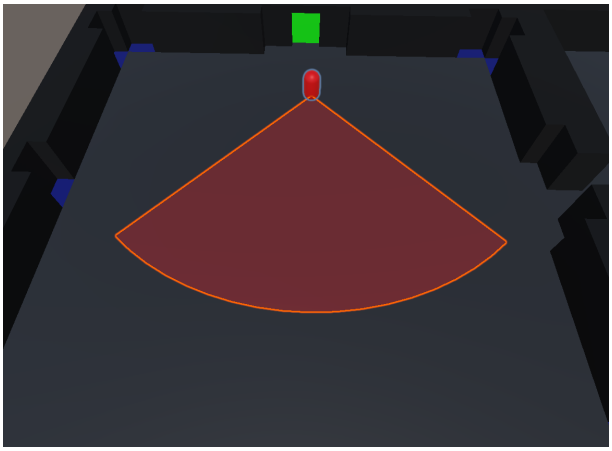

Figure 2.3 A guard patrolling an area. The guard's sight is represented by the view-cone.

2.2.3 Security measures

One of the most common security measure in stealth games is a security camera. [12] We implement cameras that have visual sensors and respond to the player by sounding a loud alarm, alerting surrounding guards. If standing in the camera's vision for too long, it will take behavior of a turret and shoot the player. Figure 2.4 shows two security cameras at the exits of an area.

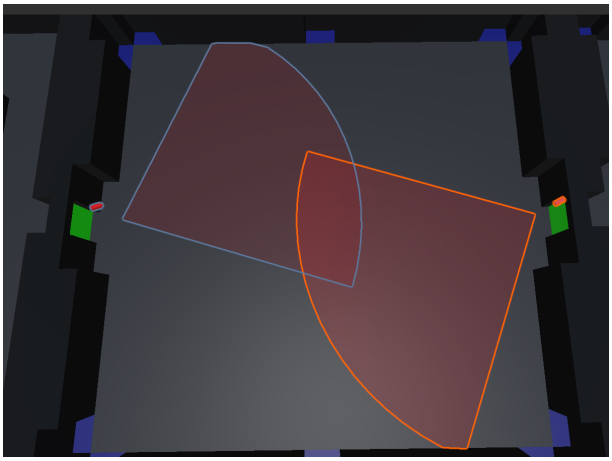

Figure 2.4 Security cameras at the exits of a room. Vision cones represent their sight.

2.2.4 Environmental hazards

Here, we take inspiration from *Thief* and *Dishonored*, implementing sound and death traps. Many stealth games have loud flooring or motion sensors that make a loud sounds in response to players presence. We implement traps triggered by stepping on them, causing a loud noise. Death traps will kill the player instantly when stepping on them. These can be seen in Figure 2.5.

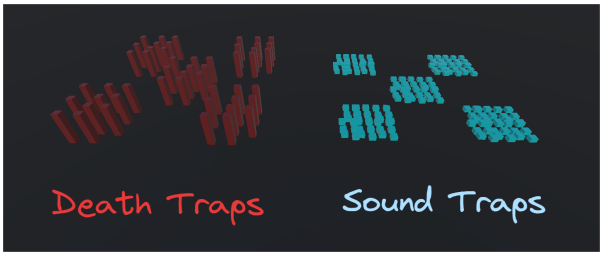

Figure 2.5 Traps implemented in our game.

2.2.5 Movement

We mentioned movement being a very important part of the avatar means. Most of the games implement at least 2 movement modes, slow and stealthy or fast and loud. The player in our game normally walks slowly and quietly. Fast running is accompanied by loud steps that can be heard by guards, as seen in Figure 2.6. Here, we take inspiration from *Dishonored* and chose to implement sliding to give to the

player ability for some skill expression. While running, players can slide on the ground, retaining the fast momentum in a fixed direction without making a sound.

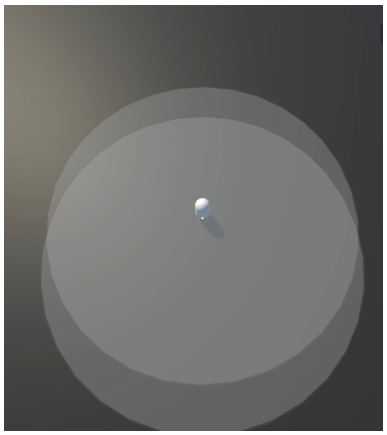

Figure 2.6 The visual representation of sound emitted by loud footsteps in our game.

2.2.6 Cover Systems & Visibility

Visibility will be handled in simple binary manner. The player can be either seen or unseen. Enemies' line of sight is broken by walls and and objects. We do not implement a cover system as a standalone mechanic like in *Tom Clancy's Splinter Cell Blacklist*, as many stealth games do not implement it either. It is not necessary addition, as players can simply stand behind a cover and so it does not greatly influence the level design. However, we grant the player ability to peak around corners while in first person without being seen, similarly to *Dishonored* or *Thief™ II: The Metal Age*. Figure 2.7 shows a player hiding behind cover in our game.

2.2.7 Concealment

Concealment can be implemented in many various ways ranging from camouflage to darkness masking players' presence. These are usually accompanied by specific places in a level in a form of thick bushes or shadowy alcoves. We choose to implement a generic refuge space, which makes the player invisible by standing on it. This can be mapped to other forms by a specific game. Figure 2.8 demonstrates a player hiding in a refuge space from a guard.

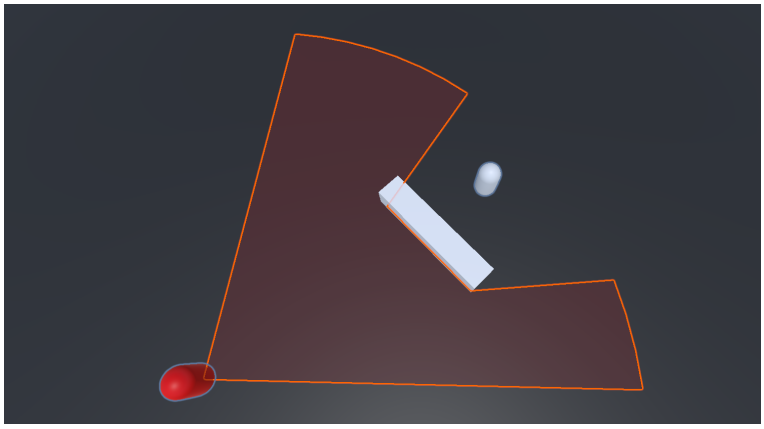

Figure 2.7 An instance of a player standing behind cover, so the enemy does not see him.

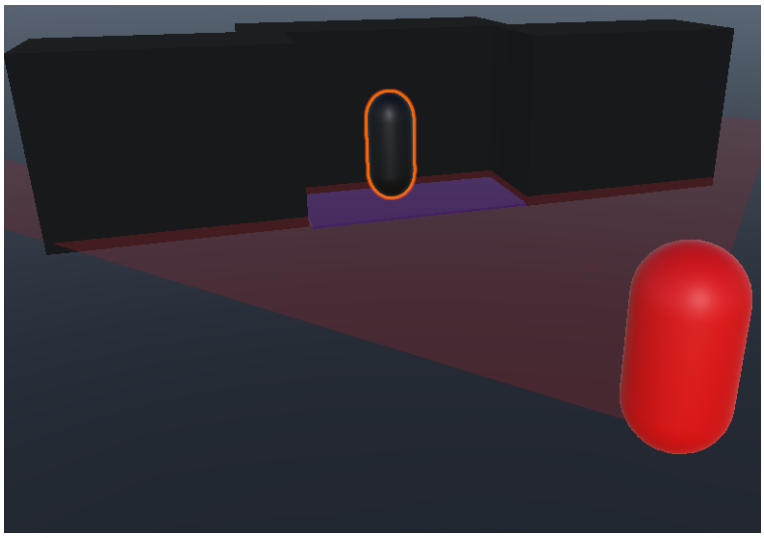

Figure 2.8 The player is standing in a refuge space, invisible to the guard.

2.2.8 Tools

One of the most common tools in stealth videogames is distraction. [25] Commonly, it takes the form of throwable rocks and bottles on the ground (*Dishonored*) or a part of the avatar's firearm (*Tom Clancy's Splinter Cell Chaos Theory®*). We choose to give players a gun that shoots loud objects that can draw guards' attention. However, tools can be often picked up and serve as a bonus found in a level. For this reason, we implement two pickupable items. Trap disarming kits to counteract the environmental hazards, and an invisibility camo, similar to the thermo-optic camo in *Deus Ex* [26]. Using this camo will make the player invisible for a certain duration, counteracting the vision of enemies and security cameras.

2.2.9 Locks & Keys

We talked about types of locks and keys in Section 1.5.1. The main type of lock we implement are locked doors. These are permanent safe locks. Door keys are particular and persistent. Aforementioned traps can be also considered as unsafe locks, while trap disarming kits are nonparticular consumed keys. It is the same case for guards and invisibility camo. Finally, we implement a special locked door to be used as an asymmetrical lock shown in Figure 2.9. The inspiration comes from walls of light in *Dishonored*. Some keys can be guarded as depicted in Figure 2.10.

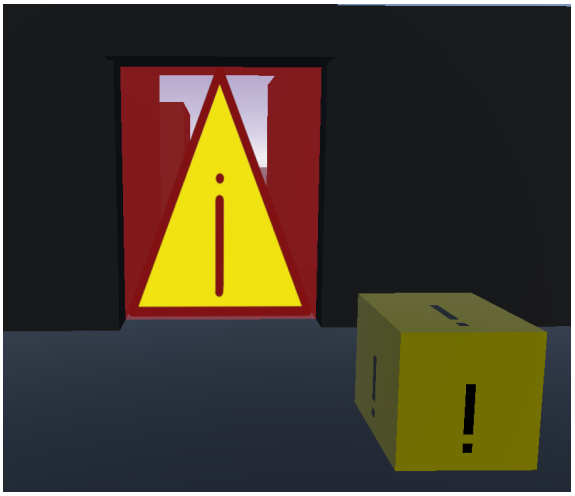

Figure 2.9 An asymmetrical lock in our game, inspired by the walls of light in *Dishonored*. The yellow box serves as a power source for the door. Disabling the box disables the door.

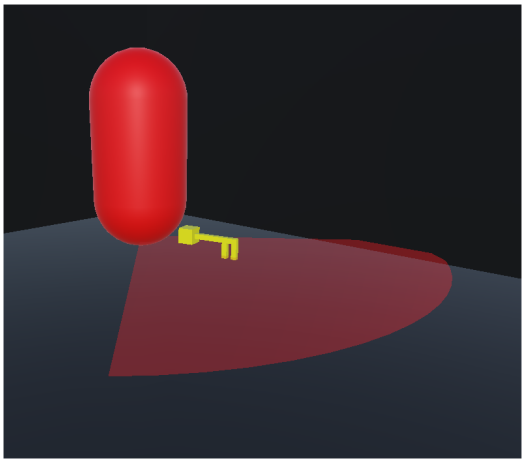

Figure 2.10 En example of a guarded key.

2.3 Our Algorithm

We construct our algorithm based on the idea of cyclic generation. The main part of the algorithm is split into three core classes. **Graph Generator**, **Graph Drawer**, and **Map Builder**.

2.3.1 Graph Drawer

We discuss Graph Drawer first, as it is the most obscure part of the cyclic generation algorithm. We considered three options for drawing a graph into a grid.

The first option is to **generate a graph into predetermined grid**, as the authors suggested. Cyclic generation suggests that we start with a graph embedded into a grid. Then, applying rules on the graph would maintain and work with this topological information, placing new vertices and edges straight into the grid. That said, we have already mentioned that this produces many questions.

Suppose we have a grid. The book mentions starting out with a graph with a single cycle. But how does this look like in the grid? Let's try one implementation. Figure 2.11 illustrates a starting graph along with the uncertainties. How do we perform an extension on an edge? Would we simply promote one of the grid nodes into a graph vertex? But what if the edge is already full? Then, there is the question of adding new paths to form cycles. How to determine what nodes will this path go through? Will it be some search algorithm like A*? What happens

when it can't add another path due to the lack of space? In summary, other than the fixed size of the level, the result of this step seems very vague and uncertain. Therefore, we chose to abandon this approach.

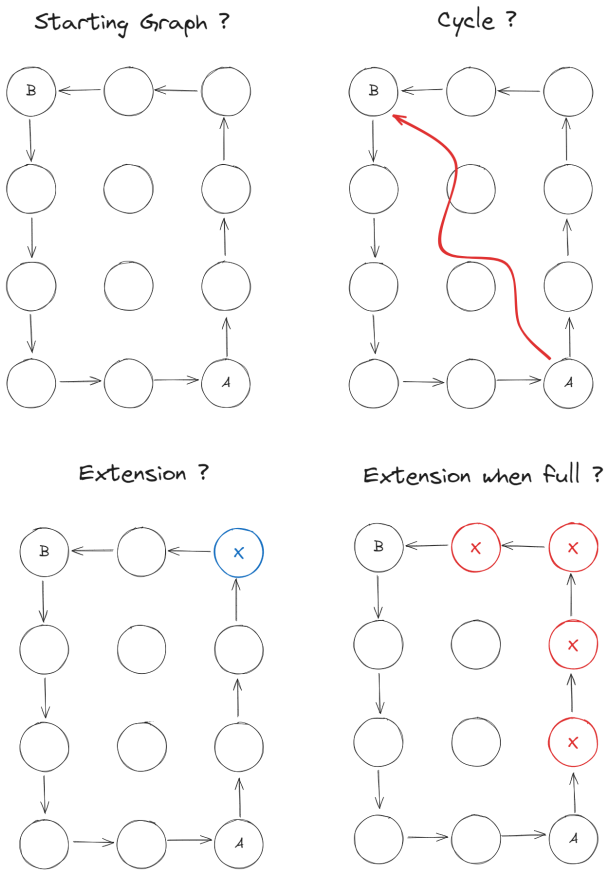

Figure 2.11 The problems we encountered when trying to implement the grid graph representation in the cyclic generation algorithm. Namely the implementation of rules such as cycle and extension.

The second option is to **generate a graph first and then draw it into a grid**. This problem is called orthogonal graph drawing. There are multiple algorithms that try solving this problem. [27]. Heuristics are usually used to establish what constitutes a good result. For example, total edge length, number of bends, or resulting area of the drawing. One fitting approach we encountered is Topology-Shape-Metrics algorithm. [28] Nevertheless, as many other similar algorithms, it is difficult to implement. And even more so, if we consider extending such algorithm into the 3D space. In the end, we leaned towards the third, more straight forward and extendable approach.

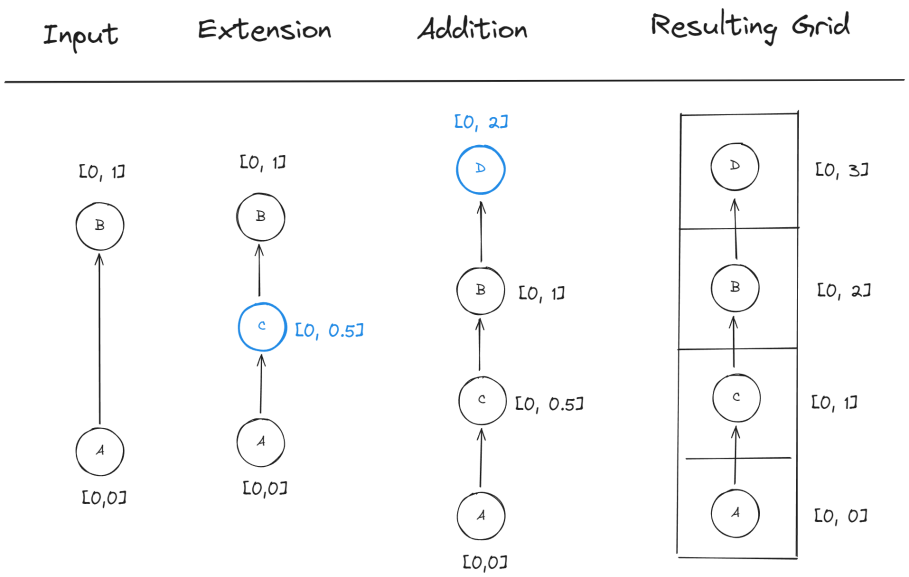

Figure 2.12 The basic idea behind our algorithm.

The third option is to **build a grid during the generation process**. The basic idea is simple. Let's say we have an edge that we want to extend. Both vertices have coordinates. Then, we simply add a node in the middle of that edge. We can do this as many times as we want. Finally, we take the sorted coordinates, their count setting the size of the grid and their order the place in the grid. Figure 2.12 demonstrates this idea. We can do this both in vertical and horizontal direction. The challenge is adding a new vertex without any crossings. Suppose we want to add a new vertex to the right, but there is something already there. We then have to loop through all the vertices and edges right to our location and pick the closest one that could be in the way of adding our vertex. If there is none, we simply add it with coordinate + 1. If there is an obstruction, we add it at half of the difference. Such process of addition is shown in Figure 2.13.

This approach is much simpler to implement then graph drawing algorithms and we have much better idea of what happens then in the first approach. The downside opposed to the first approach is that we can't control the final size of the grid. In general, we choose the certainty of the results of applied patterns rather than control over the exact size of the level. The second downside, together with the first approach, is the sacrifice of modularity. Graph Drawer depends on Graph Generator for precalculating most of the topological information.

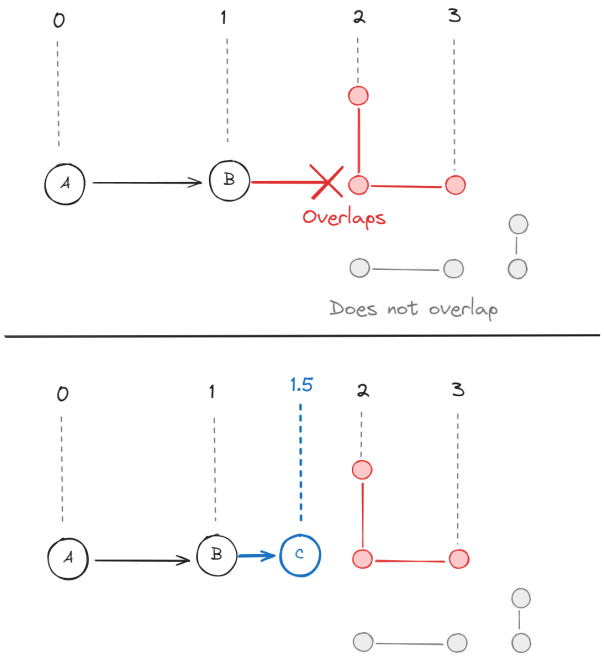

Figure 2.13 An addition of a new vertex. It cannot be added to the position 2 because of the overlapping edges and vertices. So it is added in-between.

2.3.2 Graph Generator

The generation can begin with a simple graph, like those in Figure 2.14. In regards to the transformational rules, we implement three rules inspired by the cyclic generation algorithm: **Extension**, **Fork**, and **Cycle**, as shown in Figure 2.15. It is important to realize that the resulting graph contains only edges with at most one bend. We begin with such graph and application of every rule holds this invariant. Another important realization is that these rules don't have to be deterministic. Applying Cycle onto an edge means, that we want to transform the graph so there is an additional path towards the target vertex. In general, we focus on the number of paths between vertices rather than building the exact structure. We go into more detail about these rules in Chapter 3.

Using these rules we implement patterns displayed in Figure 2.17. Patterns also add locks and keys to the resulting vertices. As a result, we care about the directions of edges, as they describe intended routes for the player. Applying a locked pattern in an opposite direction could cause a deadlock, preventing the player's progression altogether. Figure 2.16 shows an example of what can be generated with just these rules.

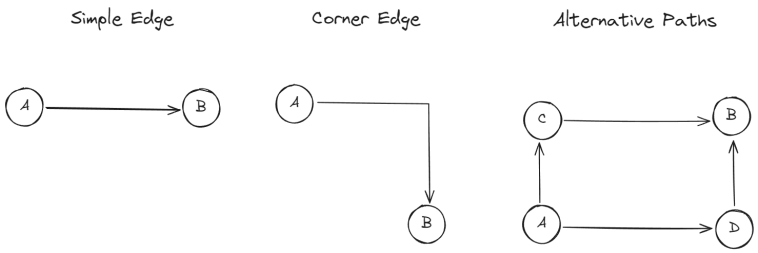

Figure 2.14 Examples of starting graphs for generation.

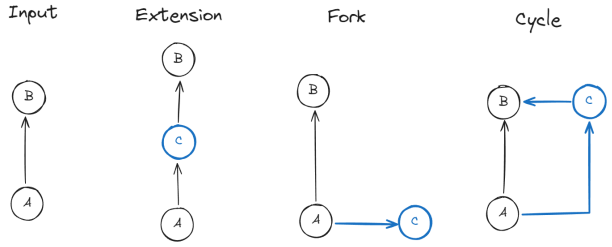

Figure 2.15 Graph transformational rules used in our algorithm.

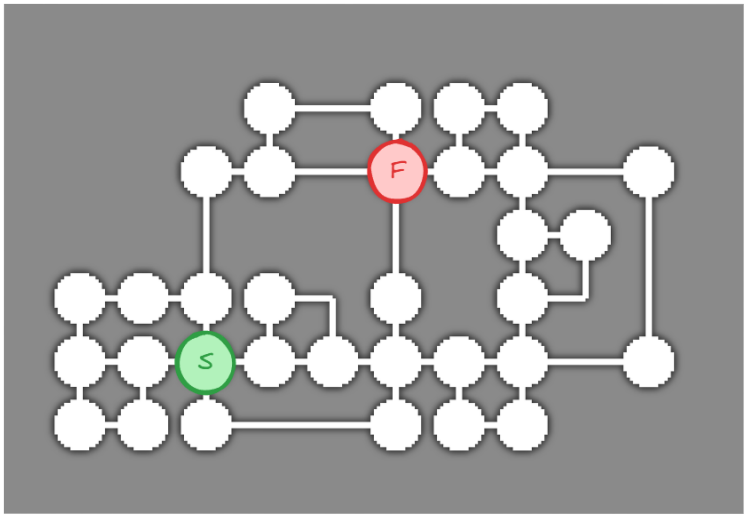

Figure 2.16 A generated graph from the corner starting graph. This was done by a repeated application of the cycle rule on a random edge and the extension rule on a longest edge.

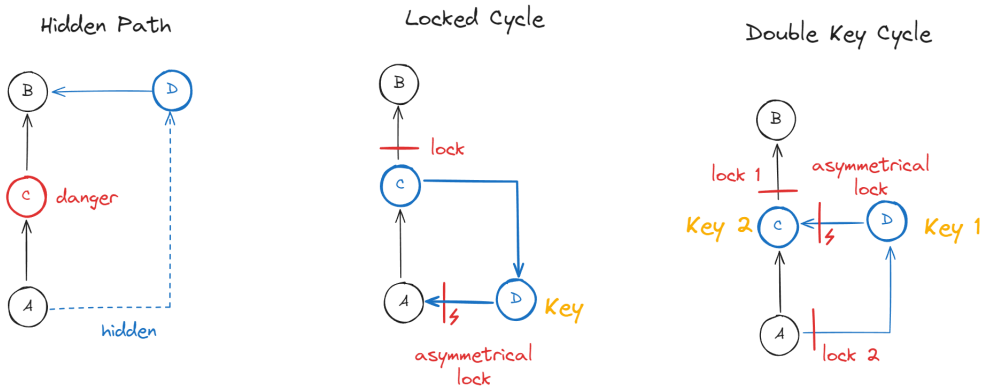

Figure 2.17 Patterns used in our algorithm.

We use combination of patterns from the cyclic generation algorithm in accordance to the existing levels from the games we analyzed. Figure 2.17 displays **Hidden Path**, **Locked Cycle** and **Double Key Cycle** patterns. The Hidden Path pattern was inspired by the situations in both the Bank level (roof entrance) and Figure 1.2 level section (box climb). The Locked Cycle pattern was inspired by the fire exit situation in the Bank level. The Double Key Cycle pattern was simply taken as an example of a more complex pattern.

But now, we need to extend this algorithm into 3D. We considered multiple options. The easiest way would be to work with a predetermined graph with multiple floors. Then, only use patterns on each floor individually. But this wouldn't result in any patterns extending through multiple floors. We want for it to be possible to find a locked door on one floor and a key on another floor. Also, there would be too much work on the side of creating a preexisting graph, taking away from the idea of procedural generation. The second consideration was for the patterns to dynamically extend or create a new floor. But we thought that would make the result very random and unpredictable.

We settled on an approach that requires only a simple graph with two floors. Figure 2.18 shows the graph we use for generated levels in our project. The generation is split into two phases. In the first phase, we use patterns that are specifically altered to work on edges connecting two floors. This creates puzzles and relations spanning multiple floors. It also creates additional floors. Then, only in the second phase we apply the normal patterns on each floor as we intended. We call these altered patterns **floor patterns**, depicted in Figure 2.19.

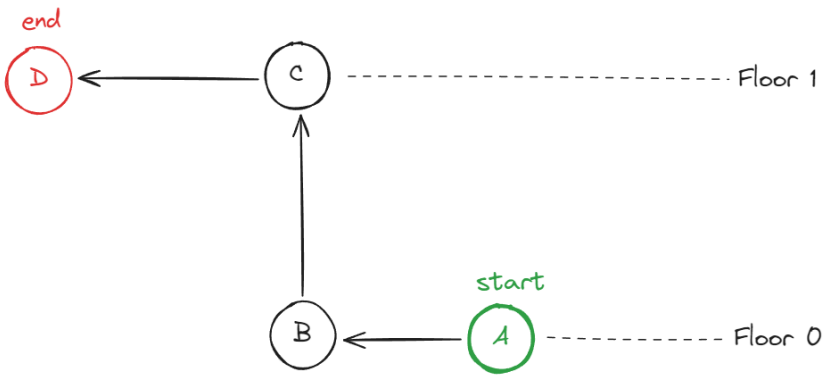

Figure 2.18 The starting graph we use in our level generation.

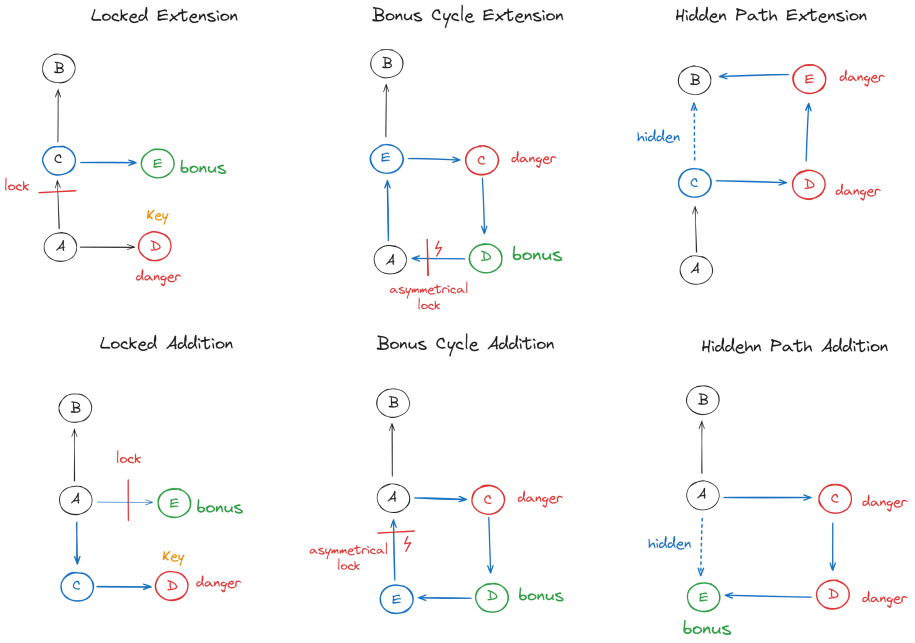

Figure 2.19 Floor patterns are patterns that are applied on edges that connect two different floors. When there is a possibility to add a new floor on top or bottom, pattern can perform an addition instead of an extension.

2.3.3 Map Builder

The third step of our algorithm is responsible for transforming the graph drawing into actual objects in the game. "Procedural generation in game design" mentions starting out with a low resolution grid, subsequently increasing the resolution, resulting in a highly detailed populated tilemap. As the author of the cyclic generation algorithm does not elaborate on the concrete process of this transformation, we have chosen to do it in the following way based on the few bits of information there are.

We implemented three resolutions. As an input, we get a set of vertices and edges described with coordinates in a 3D grid. We construct a grid of **Supertiles** of the same size. These are then deconstructed into **Tiles** for higher resolution. Finally, Tiles are deconstructed into **Subtiles** at the highest resolution. Figure 2.20 demonstrates this process. Supertile represents a room layout and Figure 2.21 shows what layouts we implemented into our game. Some of these are straight forward, like the hallway with shadowy alcoves or the simple room. But some of our layouts are inspired by the Splinter Cell games. For example, Figure 2.22 shows the inspiration behind the wide hallway layout and Figure 2.23 is inspiration for the recursive room layout.

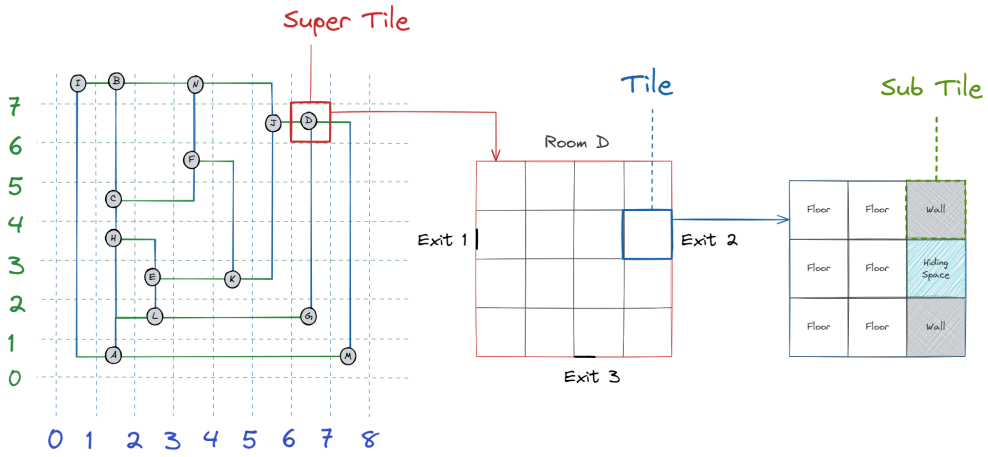

Figure 2.20 A demonstration of map building from a grid graph drawing. Supertile is a room. Tile is a smaller building block. Subtile is a spawnable object in the game of size $(1m \times 1m)$.

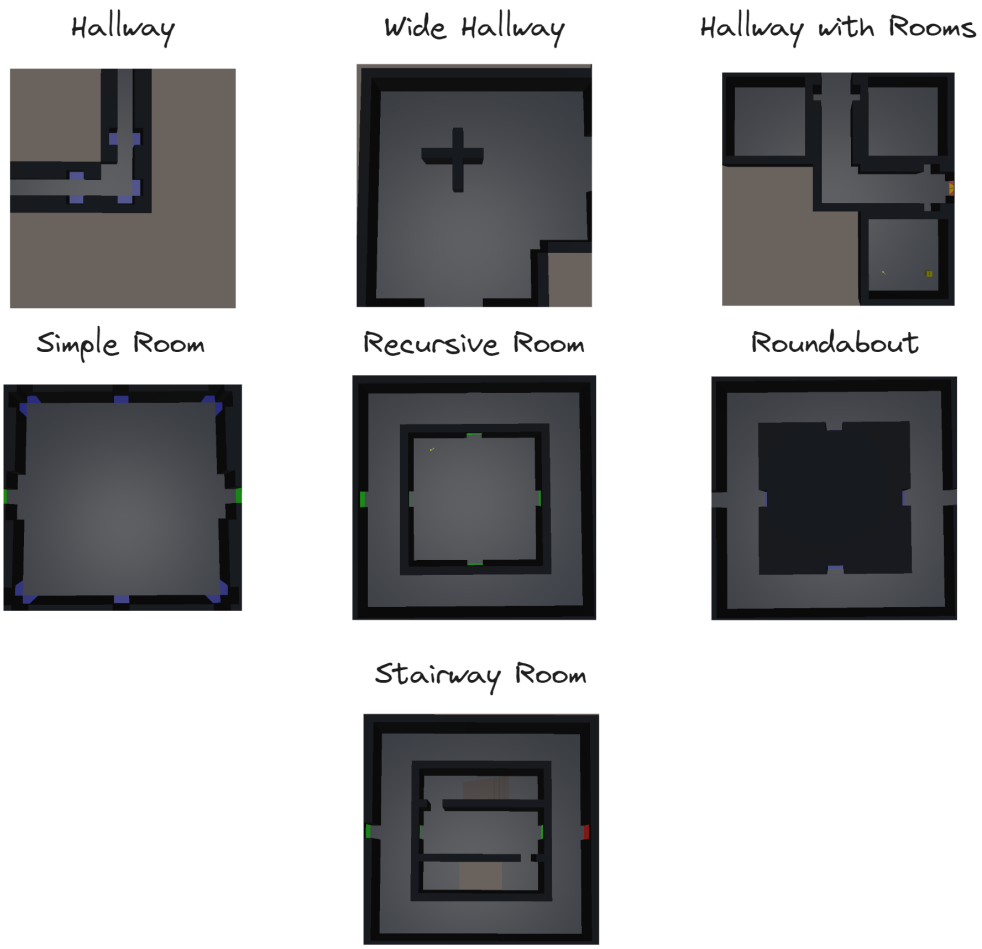

Figure 2.21 Supertile layouts we implemented into our game.

Figure 2.22 A wide hallway with covers from *Tom Clancy's Splinter Cell®*.

Figure 2.23 Recursive room from *Tom Clancy's Splinter Cell Chaos Theory®*.

The room layouts have three main parts. A patrol path is reserved for guards to patrol. Exit tiles contain doors or openings to another room, so they can be used for special things like security cameras. Free tiles are reserved for any objects, keys, or additional preoccupied guards. Figure 2.24 shows this composition. This approach of placing items into slots is a general concept in procedural generation, used in *Spelunky* [29] for example. Figure 2.25 shows the layouts from the point of refuge and prospect spaces. Note that not all of the refuge spaces have to be literal refuge tiles. In Wide Hallway, players can take cover behind the pillar in the middle. In Recursive Room, the inner room does not spawn anything right behind those inner doors. Furthermore, guards that are possibly spawned inside are preoccupied with their vision reduced. It is the same case for Hallway with Rooms. We have discussed the concept of different refuge spaces in Section 1.4.

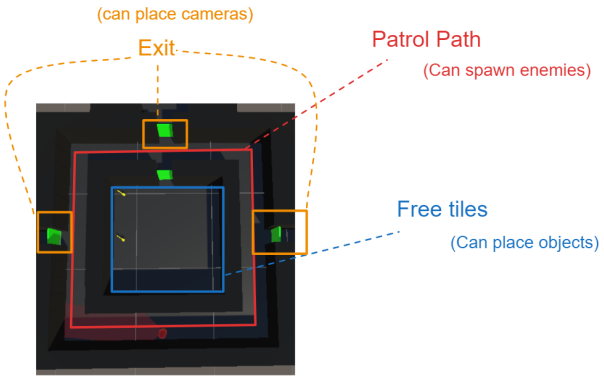

Figure 2.24 The layout composition of a recursive room.

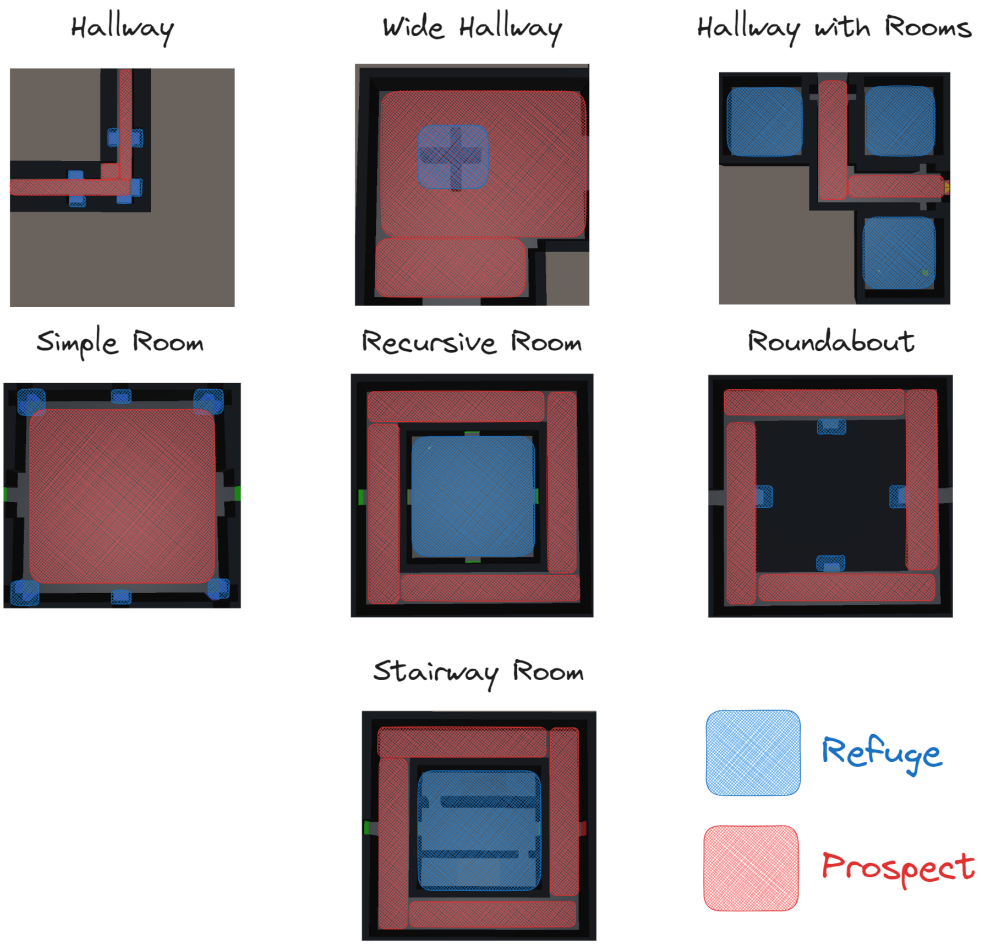

Figure 2.25 Layouts represented from the point of refuge (safe) and prospect (dangerous) spaces.

Chapter 3

Implementation

In this chapter, we will discuss implementation details of our game and the level generation.

3.1 Project

Our project is made in Unity, as it is a reputable free 3D engine that we have experience with. Godot and Unreal were also considered, but the former was still is unproven infancy at the time of starting this project and the latter is a big and complex tool with powerful graphics capabilities unnecessary for our project.

The project is divided into two main parts. The game logic part and the level generation part. In the game logic part we can most notably find implementations of the player controller, guard and camera behaviors, sight and audition senses, and all of the objects that spawn into the world. Figure 3.1 shows the general structure in the game scene, while Figure 3.2 shows the composition of live objects spawned in the game, with 'live' meaning that they have some independent behavior.

The flow of information in the level generation part is described in Figure 3.3. We implemented a special directed adjacency graph called Grid Graph that holds topological information that patterns use. Concrete patterns are derived from an abstract Pattern class. They have to override the `Apply(Edge, GridGraph)` function to transform the graph and then place locks and keys on appropriate vertices. They can use transformational rules `ApplyExtension(Edge, GridGraph)`, `ApplyCycle(Edge, GridGraph)`, or `ApplyFloorExtension(Edge, GridGraph)` that are implemented in the Pattern class.

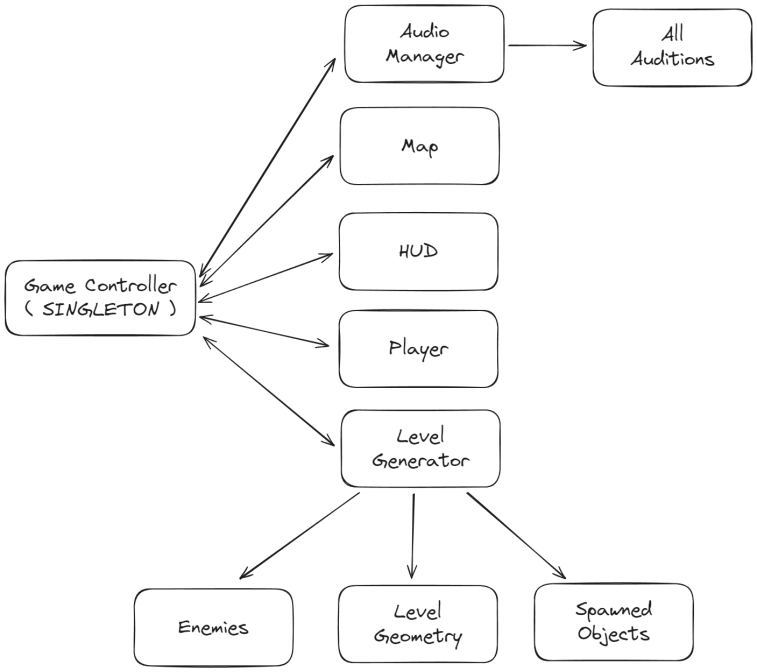

Figure 3.1 The general structure of the game scene.

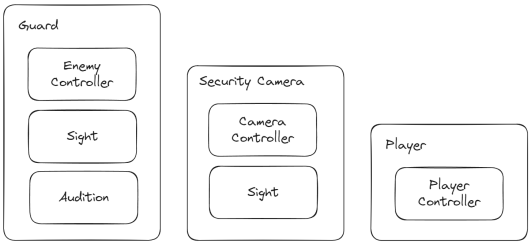

Figure 3.2 The component composition of live objects in the game.

`IKey` and `ILock` are interfaces for classes representing locks and keys placed on vertices during generation. Their relationships are illustrated in Figure 3.4. Both interfaces also contain the `Implement(SuperTile)` function. When a lock or a key is placed onto a vertex, Map Builder simply calls that function and they implement themselves onto a given Supertile. Finally, objects and enemies are spawned onto the appropriate tiles. Figure 3.5 shows what objects can be spawned and what they inherit from. `ILockObject` contains `Unlock()` function that allows for unlocking the object. For example, camera controller is disabled when power source is cut.

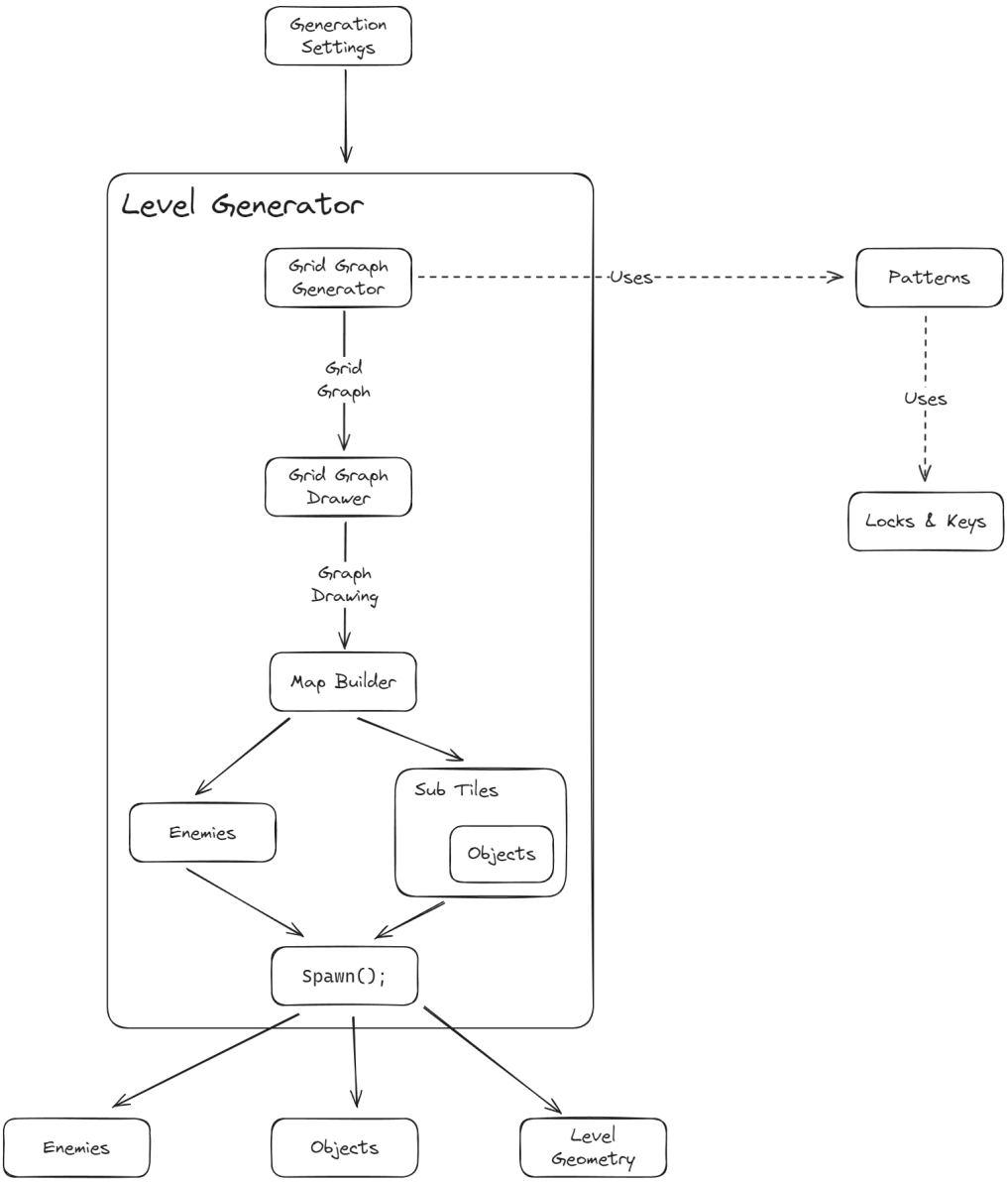

Figure 3.3 Level generation information flow.

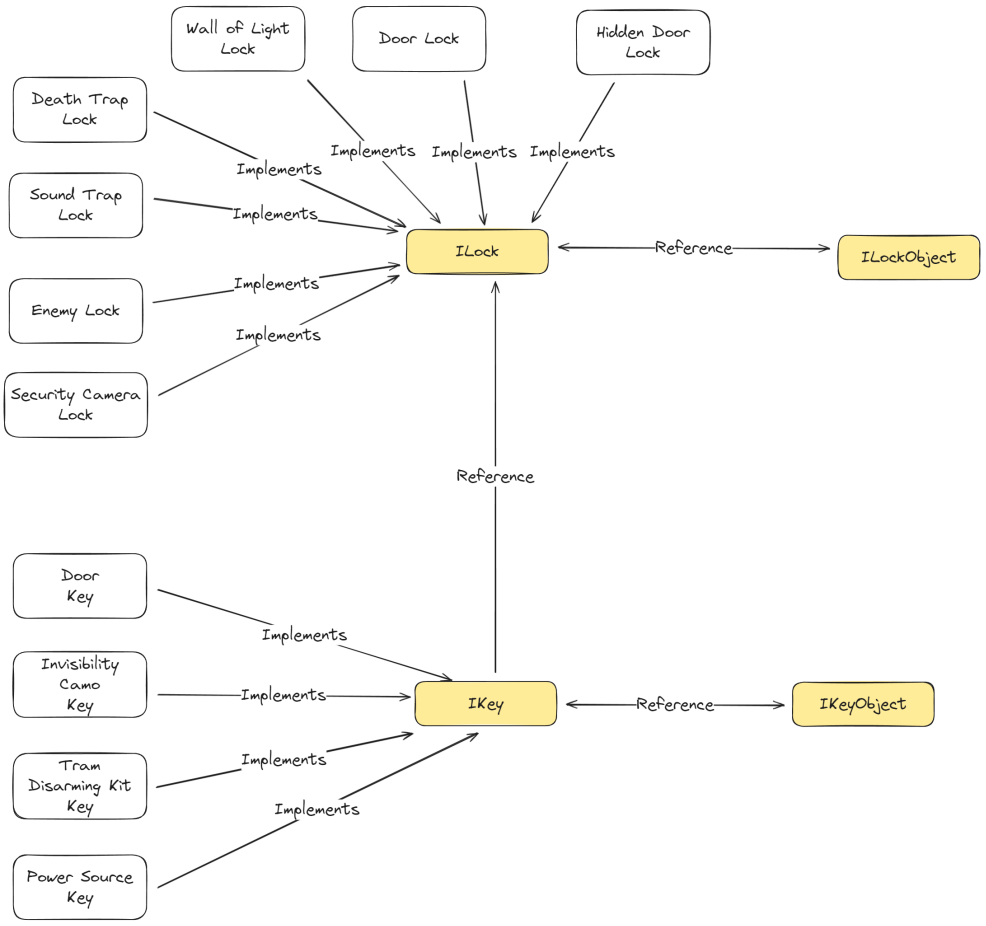

Figure 3.4 Relationships between locks and keys.

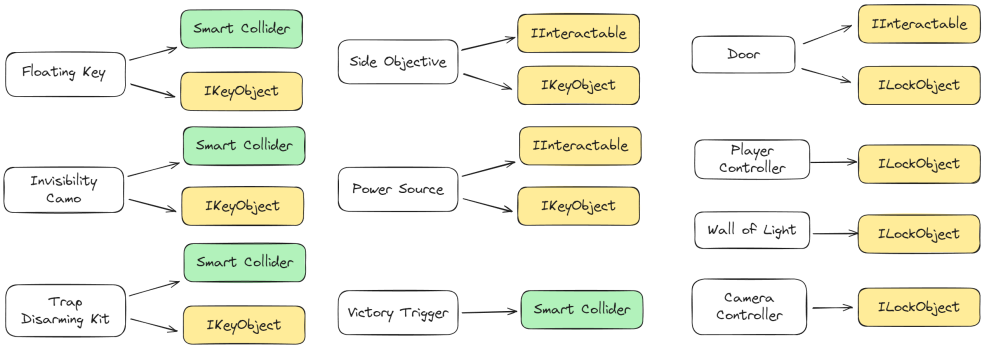

Figure 3.5 The inheritance of spawnable objects in our game.

3.2 Rules and Patterns

In Chapter 2 we mentioned that rules and patterns don't have to be deterministic. We also discussed how we prefer the certainty of the results of applied patterns rather than the control over the exact size of the level. The main point of a pattern is to create a scenario for the player. For example, the player finds a dangerous path, so he can go back and look for a safer (or hidden) path. The exact length and positions of the rooms may differ.

This brings us back to the cycle rule. The problem is that it can fail in adding a new side path. This happens when rooms do not have free exits to attach a new path onto. As a result, we thought of alternative implementations of creating a cycle between two points, so we have other options when the default one fails. Figure 3.6 shows the concrete implementation of the cycle pattern. This means the cycle is always successful. Fork is much easier. As rules are always applied on an edge, if an exit is not available in the source vertex, we can simply extend the edge and apply a fork on the newly created vertex in the middle.

Another problem we needed to solve with patterns is overlapping of locks. Imagine a scenario where a pattern locks a door. Then, another pattern can be applied to the edge that leads to that locked door and tries to place another lock on it. While we could implement locks dynamically stacking on each other, we believe it's not a good design, as not all types of locks can be stackable. As a solution, we designed the patterns so they always place locks and keys only on the new vertices or new exits. New means they are still empty, without any locks. You can see this in previous figures: Figure 2.17 and Figure 2.19.

There is one more detail we want to mention about rules and the intermediate grid coordinates that are computed for the vertices. In Figure 2.12 we implied there are decimal numbers. However, the implementation oftentimes requires comparison or equality checking, which is not reliable with floating point numbers. To create a reliable grid we use 64-bit integers instead. We mention this, because this puts a theoretical maximum on the size of the grid.

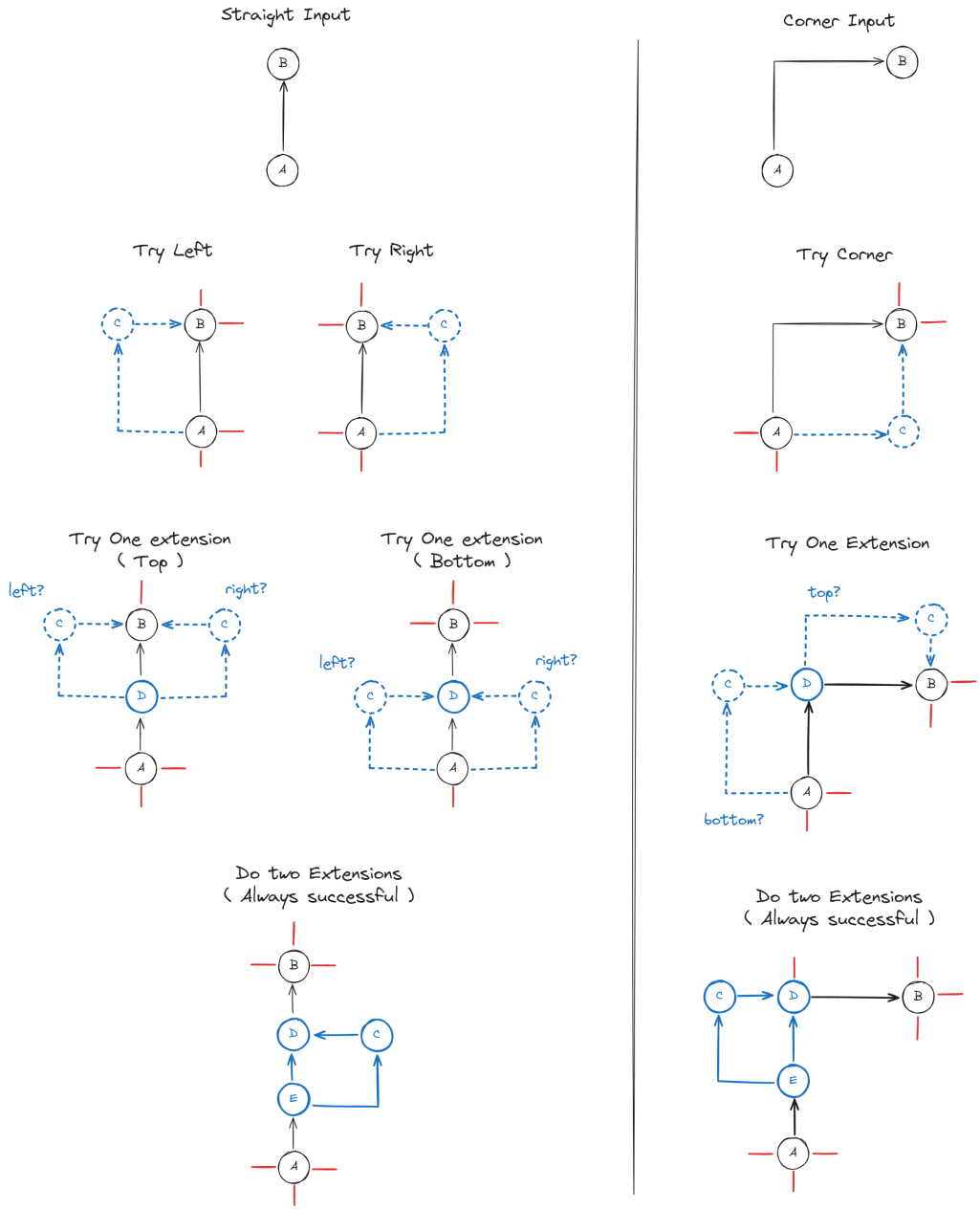

Figure 3.6 Different implementations of the cycle rule, when vertex exits are obstructed.

When appending a new vertex without obstruction, instead of adding 1.0 as in Figure 2.12, we add 2^{57}. This means we can append a maximum of $\lfloor \frac{2^{63-1}}{2^{57}} \rfloor = 63$ times. And in the worst case we can perform extension at most 57 times if we always happen to choose the shortest edge. Every pattern we have adds at most one vertex onto another, or extends at most three times. Figure 3.7 demonstrates this on the Locked Cycle pattern. The worst case is that the shortest edge is always chosen and it is extended three times for each pattern. This can happen at most $\lfloor 57/3 \rfloor = 19$ times. However, we found that even application of ten rules on a single floor results in levels far more complex than those from the analyzed games. We place five times as the maximum number of rule applications in our project due to performance reasons.

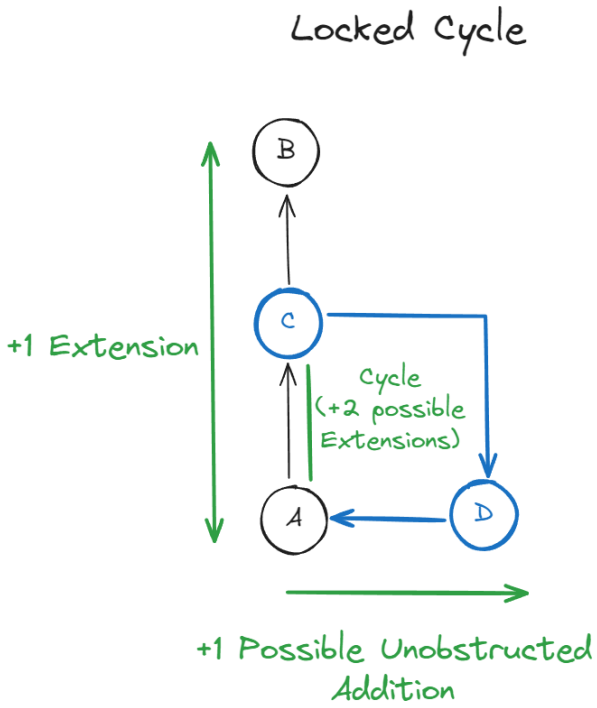

Figure 3.7 An illustration of the Locked Cycle pattern and its additions and extensions.

Chapter 4

Results & Discussion

In this chapter, we present levels generated by our algorithm and discuss their quality. Our project can be found on Github [1].

4.1 Level Complexity

Figure 4.1, Figure 4.2, and Figure 4.3 showcase levels that our project can generate. Their generation took no more than few seconds. Let's look at the first level. It was generated with 1 floor pattern and 2 normal patterns per floor. The player starts at the ground level and the finish is on the first floor. Two optional side objectives are located on the first and the second floor. There is one path connecting the ground and the first floor and two paths connecting the first and the second floor.

	Bank	**Level '0'**	**Level '1'**	**Level '2'**
Rooms	27	19	20	18
Locked Doors	5	6	7	3
Side Objectives	6	2	3	2
Walls of Light	-	5	3	2

Table 4.1 Difference in complexity between the Bank level from *Tom Clancy's Splinter Cell Chaos Theory®* and our generated levels. Hallways on the edges are excluded from our levels when counting rooms. The seeds for generating these levels were chosen sequentially as '0', '1', '2' and therefore should be unaffected by selection bias.

[1] `https://github.com/Ermith/MasterThesis/tree/main`

Table 4.1 shows comparison of our levels to the Bank level from Figure 1.3. Bank has two main floors with small exceptions of the roof, the underground and the intermediate floor for the hidden path between the 'right wing garden' and the 'top surveillance'. In terms of raw complexity, the levels are comparable. Our levels were generated by one floor pattern and two normal patterns per floor (except the third one). We can generate vastly more complex levels if we wanted to.

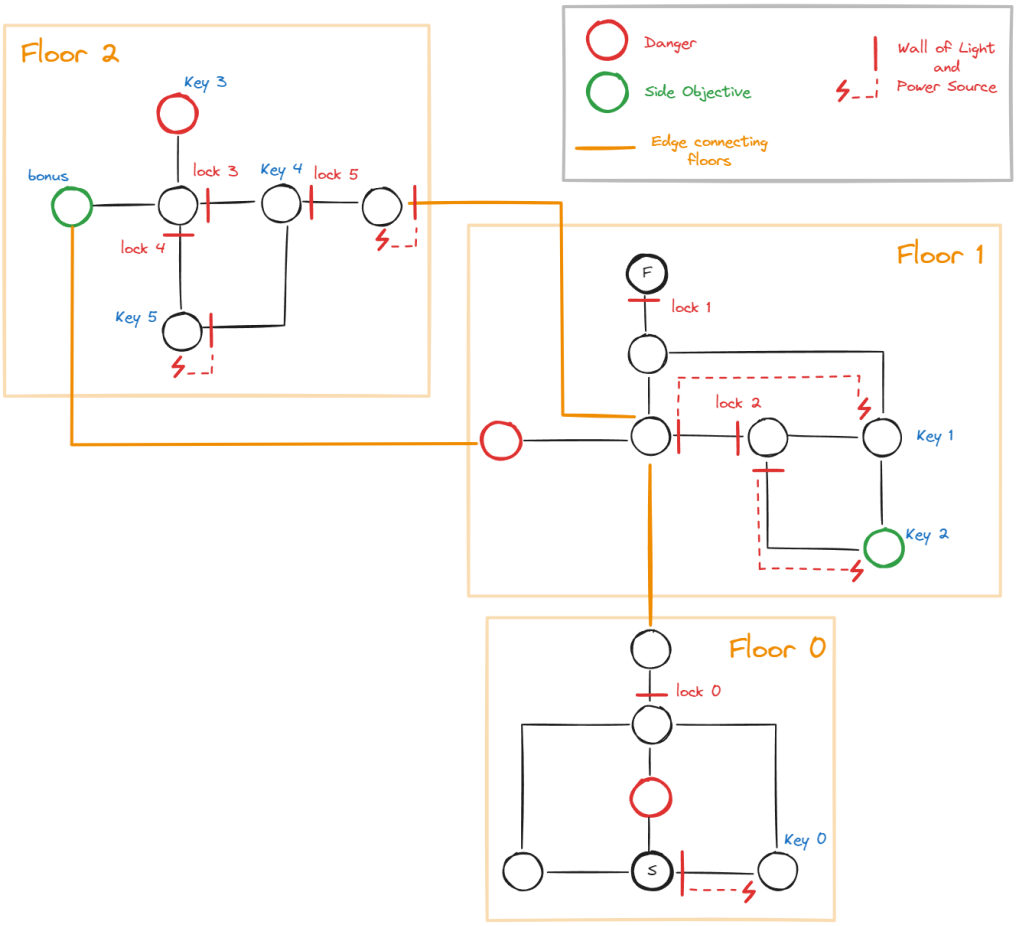

Figure 4.1 A level generated in our game with seed '0'. We used 1 floor pattern and 2 normal patterns per floor.

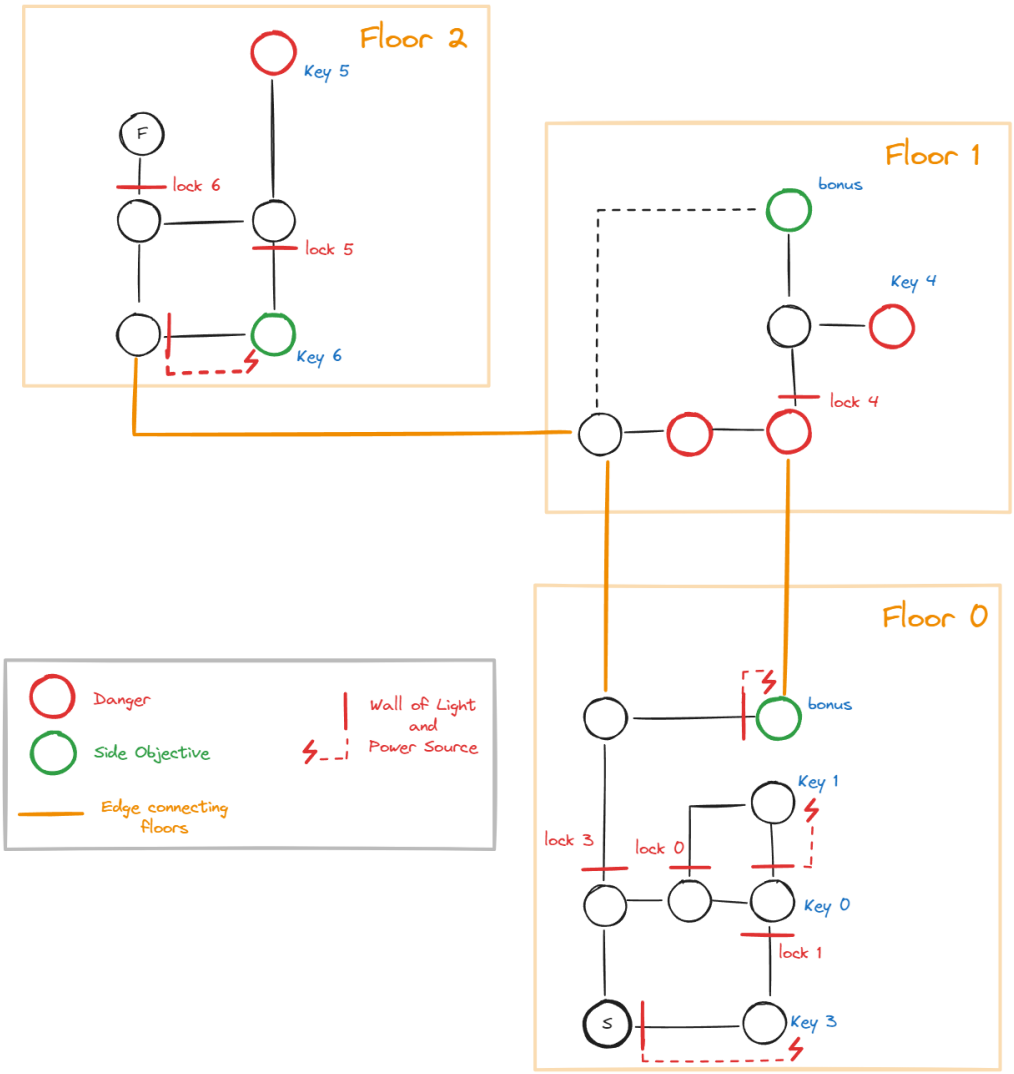

Figure 4.2 A level generated in our game with seed '1'. We used 1 floor pattern and 2 normal patterns per floor.

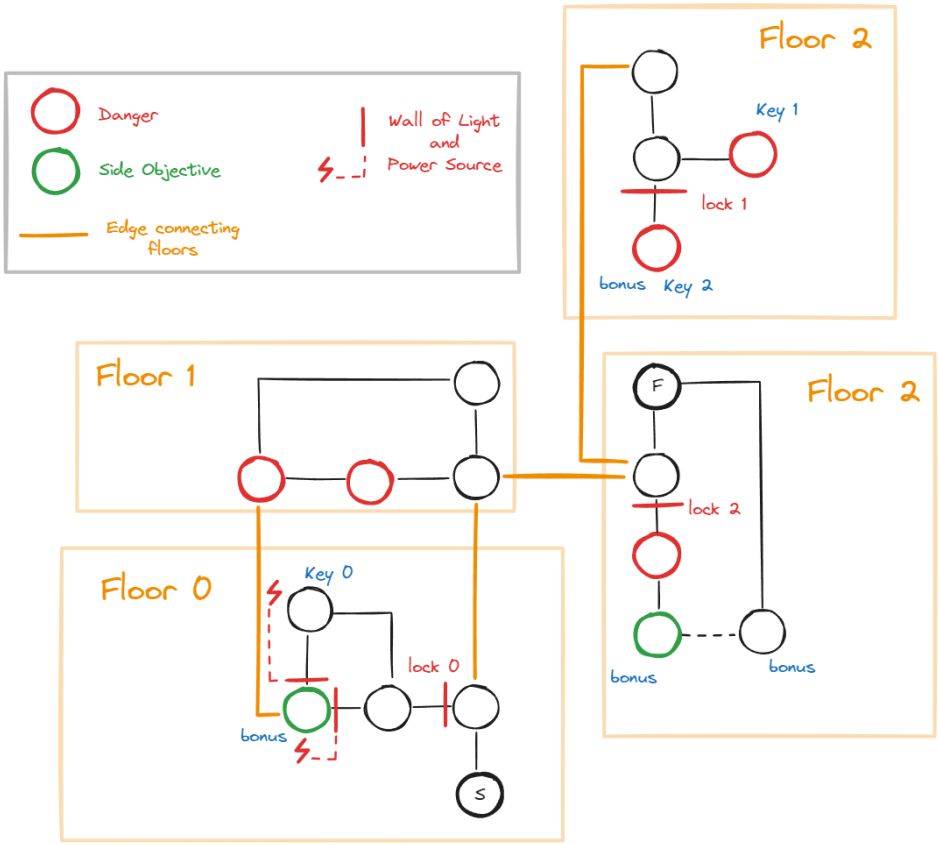

Figure 4.3 A level generated in our game with seed '2'. We used 2 floor patterns and 1 normal pattern per floor.

4.2 Pattern Application

When looking from the perspective of a player in our level, there is an immediate option to go north or west. Door leading to the east is locked from the other side. Room to the north is filled with danger. Path to the west is a longer alternative path to the north one. The player, however, has to go around to the east and discover the key that will lead him to the next floor.

The finish is on the second floor, but there are two optional objectives to be found. The player can decide to brave the danger to the third floor in search of a side objective. When discovering it, he can grab the bonus (invisibility camo) and use it to get back through the danger or traversing a complex lock filled path to the east. In summary, there are numerous meaningful choices present in our level.

On the other hand, some of the applied patterns are debatable. The third floor was created by the Locked Cycle floor pattern. The intention was for the player to get up through the danger and then get down easily through the wall of light. But extra patterns (Locked Fork and Double Key) were applied to the way leading to the wall of light. This transforms the the pattern in arguably undesirable way, as there is little reason for the player to go through all of those different lock & key puzzles. Figure 4.4 illustrates this issue.

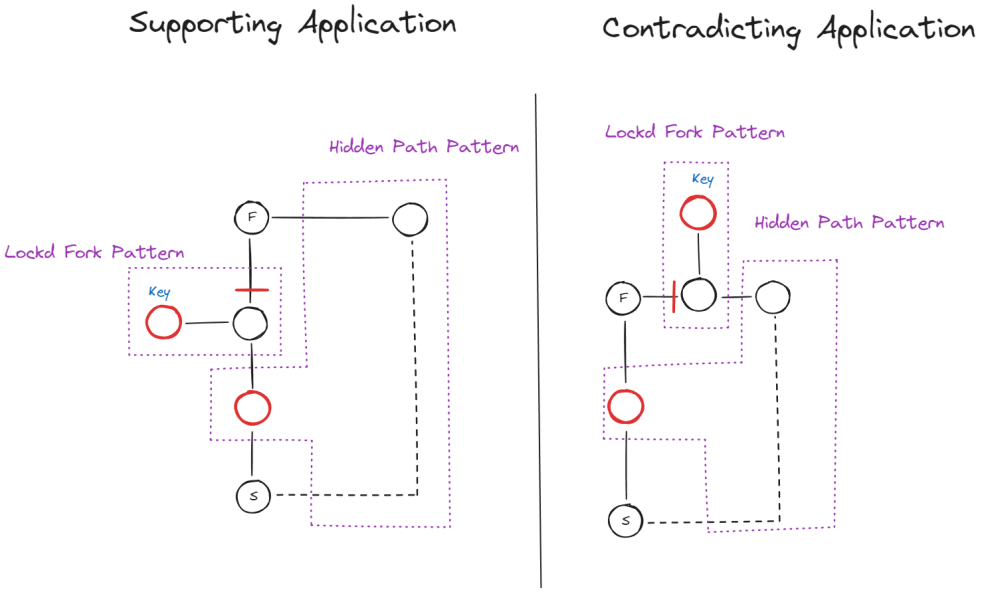

Figure 4.4 A comparison between a supporting and a undesirable application of the Locked Fork pattern onto the Hidden Path pattern. In the first case, the locked door only enhances the danger, making the hidden path more meaningful. In the second case, the fork is applied in a way that makes the the hidden path less worth taking.

For future work, we propose a tagging system for the edges to remedy the issue. We already implement a tag for a hidden edge and we don't apply patterns on them. But we think it would be beneficial to tag the edges whether they are dangerous/safe or critical/unimportant. Based on these tags patterns could make a decision to place a side objective, or the graph generator could filter edges most suitable for a selected pattern.

4.3 Map Builder

The biggest weakness in our algorithm that is to be changed in any specific game that would implement this is the Map Builder. Figure 4.5 shows top-down view of our aforementioned level. While we have built different interesting layouts based on rooms in analyzed games, they determine the moment to moment game-play and there should be ample amount of them. Furthermore, in the Bank level the rooms themselves often have side paths and interesting lock & key challenges in them.

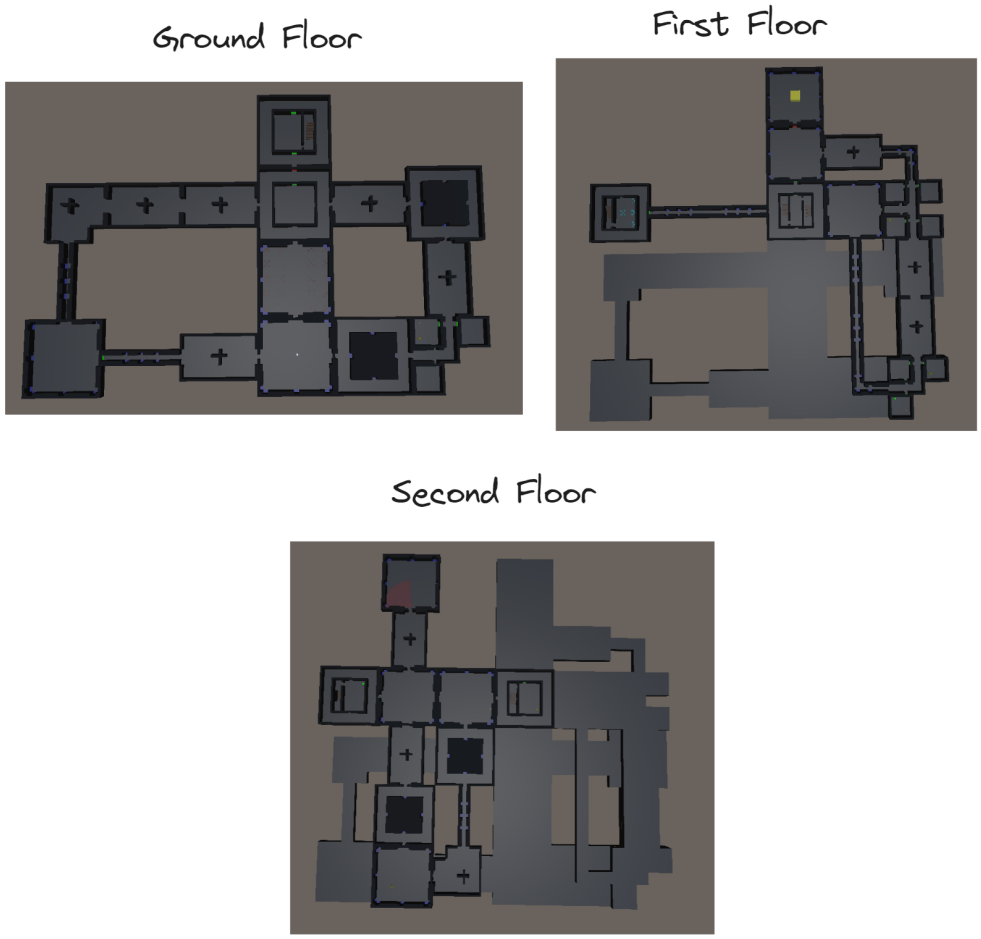

Figure 4.5 A top-down view of the level generated in our game shown in Figure 4.1.

One thing we did not implement that was mentioned in the cyclic generation algorithm were room attributes. The authors suggest giving vertices, and then rooms, different attributes that would shape how they look and what is in them. For example 'armory' or 'lava' attribute. However, we propose using the same approach of grid graph generation and pattern application to create the rooms themselves. It may be difficult, but it could result in very interesting and varied room layouts throughout the level. Take Figure 4.6 for example. We can see graph representation of the room from Figure 1.7. Such graph is not that different from graph that we would generate for the entire level. Only here, a vertex would represent a Tile and not a Supertile.

The final proposition is to implement as much patterns, locks, and keys as possible. Solving the same patterns may get old very quickly.

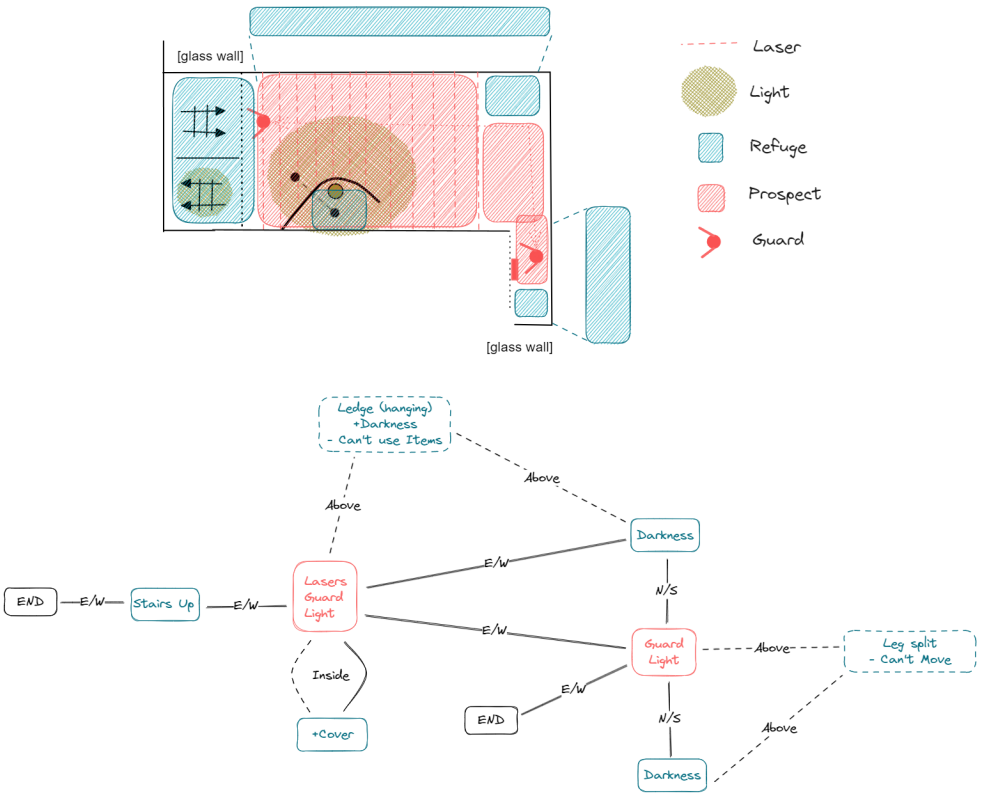

Figure 4.6 Map of the room in Figure 1.7, and its graph representation.

Conclusion

We have designed and implemented a generic 3D real time stealth game and a level generation for it. The level generator is capable of creating large levels with interesting and meaningful lock & key challenges. The levels are composed of number of different room layouts inspired by rooms from existing stealth games. In summary, we believe our algorithm has succeeded as a proof of concept and can be used in actual stealth games with additional content, such as designs of locks, keys, layouts, and patterns. Moreover, we contributed a concrete implementation of the cyclic generation algorithm, where as the original source is vague on implementation details.

As for the shortcomings of the algorithm, some of the level sections consist of undesirable, yet still playable pattern applications. The algorithm could use more guidance to omit these instances. We proposed implementing edge tags for more contextual use of patterns.

For future work, we proposed an improvement to Map Builder. While we have created a number of different room layouts, we proposed to change the map building part to dynamically generate room layouts in similar way as the level itself. Another idea to introduce multiple options for starting graphs, consequently providing more control over the generated level. Furthermore, it would be interesting to implement an algorithm controller, where the user can watch and influence the generation process in step by step fashion.

Finally, we believe this algorithm can be used in different genres, such as RPGs, Immersive Sims, or Horror games. For example, it could be implemented in titles like *Prey* [30] or *Alien: Isolation* [31]. It can be used on roguelikes as well, as it is an extension of the cyclic generation algorithm. It would be interesting to extend a game into 3D space, such as *Brogue* or *Unexplored* itself. Another interesting application could be in the puzzle platformer genre.

Bibliography

[1] Klei Entertainment. *Invisible, Inc.* 2015.

[2] Ubisoft. *Tom Clancy's Splinter Cell®*. 2003.

[3] IO Interactive. *HITMAN*. 2016.

[4] Arkane Studios. *Dishonored*. 2012.

[5] Tanya Short and Tarn Adams. "Procedural generation in game design". In: CRC Press, 2017. Chap. Cyclic Generation, pp. 83–95.

[6] Rachel Ivy Clarke, Jin Ha Lee, and Neils Clark. "Why video game genres fail: A classificatory analysis". In: *Games and Culture* 12.5 (2017), pp. 445–465.

[7] Merriam-Webster. *Abulia*. In: *Merriam-Webster.com dictionary*. URL: https://www.merriam-webster.com/dictionary/abulia (visited on 07/08/2020).

[8] Jonathan Tremblay, Pedro Andrade Torres, and Clark Verbrugge. "Measuring risk in stealth games." In: *FDG*. Citeseer. 2014.

[9] Youssef Khatib. *Examining the Essentials of Stealth Game Design*. 2013.

[10] Looking Glass Studios. *Thief™: The Dark Project*. 1998.

[11] Konami Computer Entertainment Japan. *METAL GEAR SOLID*. 1998.

[12] Qihan Xu, Jonathan Tremblay, and Clark Verbrugge. "Generative methods for guard and camera placement in stealth games". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 10. 1. 2014, pp. 87–93.

[13] Feral Interactive (Mac) Eidos-Montréal. *Thief*. 2014.

[14] Ubisoft Montreal. *Tom Clancy's Splinter Cell Chaos Theory®*. 2005.

[15] KONAMI. *METAL GEAR SOLID V: THE PHANTOM PAIN*. 2015.

[16] Ubisoft Torronto. *Tom Clancy's Splinter Cell Blacklist*. 2013.

[17] Konami Computer Entertainment Japan. *METAL GEAR SOLID 3: Snake Eater*. 2004.

[18] Mark Brown. *The Five Types of Stealth Game Gadget - School of Stealth Part 2*. https://www.youtube.com/watch?v=QLWC081dDpc. 2020.

[19] Looking Glass Studios. *Thief™ II: The Metal Age*. 2000.

[20] Ludomotion. *Unexplored*. 2017.

[21] Joost Engelfriet. "Context-free graph grammars". In: *Handbook of formal languages: volume 3 beyond words*. Springer, 1997, pp. 125–213.

[22] Roland Van Der Linden, Ricardo Lopes, and Rafael Bidarra. "Procedural generation of dungeons". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.1 (2013), pp. 78–89.

[23] Brian Walker. *Brogue*. 2018.

[24] *The Dark Mod*. 2013. URL: https://www.thedarkmod.com/main/.

[25] Alexander Borodovski and Clark Verbrugge. "Analyzing stealth games with distractions". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 12. 1. 2016, pp. 129–135.

[26] Ion Storm. *Deus Ex*. 2000.

[27] Markus Eiglsperger, Sándor P Fekete, and Gunnar W Klau. "Orthogonal graph drawing". In: *Drawing Graphs: Methods and Models*. Springer, 2001, pp. 121–171.

[28] Markus Eiglsperger, Michael Kaufmann, and Martin Siebenhaller. "A topology-shape-metrics approach for the automatic layout of UML class diagrams". In: *Proceedings of the 2003 ACM symposium on Software visualization*. 2003, 189–ff.

[29] Mossmouth. *Spelunky*. 2013.

[30] Arkane Studios. *Prey*. 2017.

[31] Feral Interactive (Linux) Creative Assembly Feral Interactive (Mac). *Alien: Isolation*. 2014.