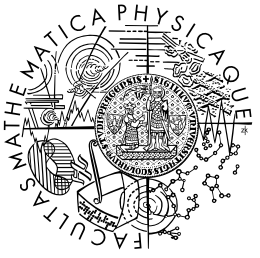

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Cyril Chudáček

Generating levels in a computer stealth strategy game using evolutionary algorithms

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Vojtěch Černý

Study programme: Visual Computing and Game Development

Prague 2024

I declare that I carried out this master thesis on my own, and only with the cited sources, literature and other professional sources. 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

This thesis is dedicated to my parents, who love and support me at any cost and any time. I am grateful to have them mentally and physically covering my back whilst reminding me of the path I chose to walk. Thank you dad, thank you mum.

I would also like to thank my supervisor Mgr. Vojtěch Černý, who gave me helpful advice whenever I asked. He did not hesitate to dedicate his time for me, no matter if he had it or not. Thank you.

Lastly I very appreciate my colleagues and friends, who held my head up, wished me luck and gave me hope. I cannot understate how much their mental support meant for me. Thanks a lot to you too.

Title: Generating levels in a computer stealth strategy game using evolutionary algorithms

Author: Cyril Chudáček

Department: Department of Software and Computer Science Education

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

Abstract: Games of the stealth strategy genre require precise and time demanding creation of game content in the form of individual level layout. For that reason it is desired to automate this process to decrease its high production cost. One of the alternatives of such automatization are evolutionary algorithms. Within this thesis we are going to show how it is possible to use them to create game levels. We will also introduce the test results of the created levels and compare them to levels created manually. Our results suggest, that evolutionary algorithms are one of the good options for creation of levels for the games of the genre.

Keywords: computer games, stealth strategy, evolutionary algorithms

Název práce: Generování úrovní do počítačové hry žánru stealth strategy pomocí evolučních algoritmů

Autor: Cyril Chudáček

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Vojtěch Černý, Katedra softwaru a výuky informatiky

Abstrakt: Hry žánru stealth strategy vyžadují precizní a časově náročnou tvorbu herního obsahu ve formě rozvržení jednotlivých herních úrovní. Je proto žádoucí automatizovat tento proces, abychom snížili jeho vysokou cenu.. Jednou z možností takové automatizace jsou evoluční algoritmy. V rámci této práce ukážeme, jak je s jejich pomocí možné tvořit herní levely. Také si představíme výsledky testování vytvořených levelů a porovnáme je s levely vytvořenými manuálně. Naše výsledky napovídají, že evoluční algoritmy jsou jednou z dobrých alternativ na tvorbu her v rámci tohoto žánru.

Klíčová slova: počítačové hry, stealth strategy, evoluční algoritmy

Contents

Introduction **6**

1 Related studies **7**

2 Background **9**
2.1 Evolutionary algorithms 9
2.2 Stealth strategy 10

3 Implementation **14**
3.1 Game library 14
3.1.1 Game data structure 14
3.1.2 Game actions system 16
3.1.3 Game simulation 19
3.1.4 Level solving 21
3.2 Playable demo 30
3.2.1 Level generating 30
3.2.2 Game 34
3.3 Evolutionary algorithms library 36

4 Methodology **38**
4.1 Evolutionary algorithms 38
4.2 Survey 42

5 Results **46**

Conclusion **49**

Bibliography **50**

List of Figures **52**

List of Tables **53**

Introduction

Game content is one of the most important factors of player engagement, with the exact importance dependent on the given game. Still, it can be estimated given the genre. As games grow more and more complex over time, the cost of development of the game content increases as well [1]. Furthermore, the demand for game content continues to rise. Manual content creation is becoming very expensive and may not be sustainable [2]. With this in mind, it only makes sense to research the territory of procedural game generation.

When choosing to use procedural generation, there are many methods available. This thesis will demonstrate the use of evolutionary algorithms. Those are powerful optimization tools that do not require any properties of the function they are optimizing, which proves to be very valuable. The thesis also describes a generic implementation of an evolutionary algorithm.

There are many different genres of games and their content very much differs. And so when aiming to research procedural generation within game content, it is logical to choose a specific group of games that would benefit from the given generated content. This thesis focuses on level layout generation within the stealth strategy genre. As mentioned above, game content's importance for engagement can be estimated by genre. It is very high for stealth games in general [3].

A generic abstraction of the genre will be discussed since its games are very niche and there is just a thin line between it and other genres. Then we will show the implementation of such an abstraction and a whole playable demo will be shown as well. The thesis also includes a description of a level solver above the created stealth strategy abstraction. We will also show how such a solver can help while using the evolutionary algorithms to generate levels during design time.

Finally, within the thesis a research was conducted which captures the relation between the procedurally generated levels and the manually created ones regarding player experience. The goal of the research is to find out whether playing levels generated by evolutionary algorithms can result in a similar player experience as playing levels created manually.

This comparison is created based on Flow [4] measurements conducted on participants playing the playable demo game.

Thesis structure

We will look into what other work was done on themes similar to this one 1. Then it is necessary to describe what are evolutionary algorithms and stealth strategy games and how we interpret them within this thesis 2.

In the next section, we will see how the respective parts of the software created within the thesis were created. Mainly the game core library, evolutionary algorithms library, the level generation, and the playable demo 3.

Another section will be about the methods used within the thesis experiment, mainly the concepts regarding the survey and how the whole experiment structure is layout 4.

Considering the completed survey, we also have to discuss the data we got 5.

Lastly, we conclude what the data suggest and what are the potential directions for future work 5.

1 Related studies

Procedural content generation within games dates thousands of years back. Since analog games often make the players create some content for themselves through dice throws and pictures [5]. Automated digital generation of game content dates back to the 1980s. The manual game content creation in digital games is suggested not to be scalable [6] and finding materials on this topic is not hard. For example, the book by *Tanya X. Short and T. Adams* [7] covers the topic very extensively. Another example of a study of this topic is *Procedural Generation of Dungeons* [8] which focuses on level creation.

There even exist studies regarding procedurally generated game content by evolutionary algorithms, such as *Evolutionary generation of game levels* [9]. This study focuses on the generation of discrete levels. The implementation of an individual within the study is however a binary string representing tiles within the discretized level. This factor alone signifies how loosely related the works are.

The focus of this thesis also lies in the world of the stealth strategy genre. Stealth games are a popular genre and are also researched extensively. For example, the *Examining the Essentials of Stealth Game Design* [10] covers the essentials of what they are and how they are separated from other games from the design perspective. It mostly covers what challenges and means players have within the genre and how important the artificial intelligence of enemies is to create a well-designed game.

When combining procedural content generation with stealth games, even there we can see many studies. For example, an enemy placement that creates interesting puzzle-like games [11]. In the work, they split the levels into segments and then place enemies, enemy routes, and cameras on the segment-splitting points. There is also a subsequent study that introduces a way to create interesting guard paths in a given vector [12].

Another example study is about enemy patrols which look natural and cover the right amount of area [13]. They achieve it by storing a value for each place in the level indicating how long it was not seen by any enemies. For any combination of enemies and their paths, they compute how well the level is covered.

Another study is about dynamically placing enemies so that they obstruct some of the paths to the level goal using grammar-based approaches [14]. They create a graph of points that are crucial for enemies to spot big areas and then find player paths to the goal and block them by placing guards at the previously computed spots.

There is also an example of a study using grammar-based approaches within stealth games to create discrete level layouts [15]. They use cyclic generation to create a graph, which represents sectors of the game. They also combine it with lock and key puzzles, which describe which sector should contain what element.

Another study discusses the ways both players and non-playable characters can move stealthily and how to calculate such paths [16]. They detect spaces, where the player can never be spotted by enemies and use rapidly exploring random trees(RRT) on a discretized level.

Lastly, we mention a study, that tries to incorporate dynamic enemy behavior based on player actions within stealth games [17], this study also depends on

RRTs and heavily focuses on the usage of distractions, which is typical for stealth games.

All of the above-mentioned studies deal only with generic stealth games and are not more specific. No studies found mention the stealth strategy genre, also called real-time tactics or commandos-like [18] games, directly. We can also see, that evolutionary algorithms are a very novel introduction into the procedural content generation within the stealth genre in general.

Most examined studies about level layout generation either work on a discrete world or discretize a continuous world. In both cases, their abstraction and algorithms work in a discrete environment, which is also not the focus of this thesis.

2 Background

When asking whether the evolutionary algorithms are a suitable tool for generating stealth strategy levels, it is crucial to split the question into multiple different sections before we attempt to answer it. In this chapter we will look into different parts of the question and what approach to them was used within the project.

2.1 Evolutionary algorithms

Evolutionary algorithms are a class of generic purpose algorithms. Their strength lies in a stochastic iterative search within the solution space. In this section, we will explore how they work and what advantages and disadvantages they bring. To see the explanation of the implementation of the used evolutionary algorithm, see section 3.3, to see the actual implementation, you can visit the related GitHub repository as mentioned in section 3.

When we want an evolutionary algorithm to solve a problem, it is crucial to choose what the potential solutions to the problem look like. We refer to those solutions as individuals. When considering a set of solutions, it is referred to as population. The evolutionary algorithm starts by creating some initial population. For the next step, we need to provide a function, that can assign a score to a solution (individual), this score reflects how good the solution is. Such a scoring function is known as a fitness function.

The evolutionary algorithm scores all individuals with the fitness function and then, depending on those scores, selects which individuals will be used to create new individuals. It takes a group of them and combines their solutions into new solutions. In most cases, the count of the newly created solutions is the same as the count of the input individuals, but it is not necessary. This process of combining solutions is referred to as crossover and the new individuals are called offsprings. The total number of offsprings created may vary, but it is mostly equal to or larger than the amount of individuals in the population.

In the next step, those offsprings are changed, mostly referred to as mutated, individually with or without connection to their score, but always totally separately to all other individuals. This mutation typically only occurs on some of the individuals given by mutation probability P.

Then we again score the new individuals and lastly, we select which individuals are going to continue to the newly created population within a process called selection. There were some offsprings created and it is necessary to reduce their number back to the size of the population. It is also possible to include the previous population in this process. When the new population is selected, the iterations continue by creating new offsprings. We can see the whole process portrayed in figure 2.1.

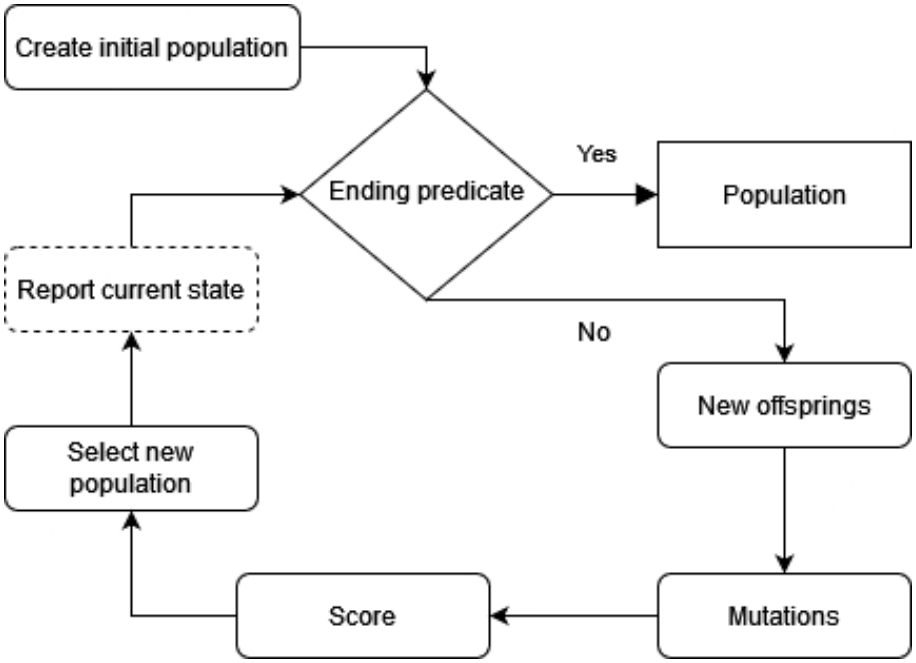

Figure 2.1 Evolutionary algorithm main loop

Evolutionary algorithms are very a powerful tool for finding complex solutions with functions too complex to define or simulate. Computer games within continuous space are exactly that type of virtual environment.

However, defining a fitness function for a procedural level generation using this method is proving to be difficult. This thesis aims to overcome this barrier within the stealth strategy game genre.

2.2 Stealth strategy

The first game of the *Commandos series* [18] is the *Commandos: Behind enemy lines* [19], which is, to the best knowledge to the author, also the first game of the genre. And since the game was very successful [20], a few other titles of the genre were created during the upcoming years, such as *Commandos: Beyond the Call of Duty* [21], *Desperados: Wanted Dead or Alive* [22] or *Robin Hood: The Legend of Sherwood* [23]. The last game of the genre for nearly a decade was *Helldorado* [24].

The silence within the genre was however broken by *Satelitte Reign* [25] and later *Volume* [26]. Many titles have emerged since, for example *Shadow Tactics: Blades of the Shogun* [27], *Desperados III* [28], *Commandos 3: HD Remaster* [29], *Frigato: Shadows of the Caribbean* [30], *Shadow gambit: the Cursed Crew* [31] and more. There are also announced not yet released games of the genre such as *Sumerian Six* [32].

As the genre becomes more known and more games of the type arrive, it only makes more sense to conduct research upon this new territory (see 1).

The stealth strategy game genre has a few characteristics that separate it from other games. Because the borderline is very thin, it is crucial to unify our expectations for this thesis. For this research, we consider games with the following characteristics

to be in the stealth strategy genre:

- Top-down view of the world
- Real-time
- Single player
- Continuous world space
- Up to a few playable characters in a given level, each with different abilities
- Levels include enemies
- Enemies can detect player by predetermined mechanisms
- Until the player interferes with enemies, their actions are mostly expectable
- The level has some goal(s)
- The player can control the character by setting a destination, mostly not by giving a directional input

The properties of stealth strategy games are also visible within the figure 2.2.

Figure 2.2 Using a skill on an enemy in Desperados III

For algorithmic purposes within this thesis, it is essential to make an abstraction of such games, that encompasses as many of them as possible but does not include any of the games outside of the genre. It is however also necessary to remove some of the constraints to allow the work to remain in the intended time scope. With this in mind, the abstraction used in this thesis is the following:

- The level is represented in a **two dimensional** space

- The level only takes place within a predefined **small area**, meaning area of 10 full enemy viewcones or less
- Levels may contain **obstacles**, which are areas of concave polygonal shape restricting movement or vision of the enemies or the playable character.
- **Enemies** have starting position and rotation, they move on a predefined path the whole game
- Enemies can **detect** the playable character via their **view**, which is always in the direction they are facing
- When the player is detected, the levels ends unsuccessfully
- The level contains **one playable character** which starts at a given position
- The level is successfully completed when the playable character reaches some predefined **goal location**
- The playable character can have some **starting skills** which can each be used a given number of times
- The level can contain some **skills** that could be **picked up** by the player

We shall elaborate on why were these constraints selected and how they relate to a stealth strategy game as we defined it above. The ones that fully satisfy the definition portrayed above, are going to be omitted.

Two dimensional Even though most of the existing stealth strategy games are in a 3-dimensional environment, they can mostly be transformed into two dimensions without losing any major gameplay elements. Furthermore creating a three-dimensional playable game would be outside of the time scope of this study.

In small area Stealth strategy levels are typically taking place on larger maps. Mostly 200 enemy viewcones and more. But the levels can mostly be broken down into small sections containing just a few enemies. For this reason, we can assume levels in small areas without loss of generality.

Detect via view In a typical stealth strategy, the enemies can not only see but also hear the playable character. The definition also allows sensing via other means. However, the main way to catch the player is in most games the view. So even though we do not cover the whole possible scale, the main point remains covered and while adding other senses would improve the gameplay experience and may introduce some interesting challenges that could be discovered by the procedural generation, the feature is minor enough not to be implemented for the sole reason of the study scope.

One playable character In the thesis we opted to only focus on creating a game for one singular playable character. That is even though in all existing stealth strategy games, the player can play as multiple different characters. By our choice of including only one character we do not let the player appreciate

the interesting synergies that can arise from using multiple characters and the level generation cannot discover those synergies. However, since every character has its own skill set, creating a setup that is targeted for that particular skill set is very useful as well. Also creating a level solver for multiple characters would increase the complexity of the study far beyond the available time frame.

It is also worth noting, that stealth strategy games typically offer the possibility to save and load the game. This feature is however fully supported by the library, only the demo does not include it. That is a decision by design. In a typical stealth strategy the levels contain many different groups of enemies (as mentioned above), so saving and loading in between interacting with these different groups is crucial. However, in the demo game, there is only one group of enemies so the saving feature becomes redundant.

3 Implementation

Now that we have set the terminology and have the background knowledge, we can proceed to the explanation of three major pieces of software, that were developed within this project. Those are:

- the game library, which contains the game core as well as level solvers,
- the evolutionary algorithms library,
- the example game demo created in the Unity software, which also contains the level generating part.

All three of them are written in C#. In this chapter, we will dive into their implementations and mention their respective algorithms, both novel and settled ones. We will also look into the software architecture of those parts and how they are interconnected. All of those 3 parts are available to view in a public GitHub repository at https://github.com/CyrChudac/DiplomaThesis.

3.1 Game library

The game library contains stealth strategy game rules abstraction, level abstraction as well as level state abstraction, navigation pathing within the abstraction, scalable player and enemy action repertoire, level solver on a given level and more. Within this section, we will introduce the algorithms, concepts, classes and architecture behind the library.

3.1.1 Game data structure

To be able to understand all of the algorithms and processes in the library, it is important to first understand what data structures lie behind the game itself. The most rigid data are the game rules, which are within the *GameCreatingCore.StaticSettings* namespace. They form the rules of the game, such as characters' speed or the enemy's viewcone lengths. Their data are stored in the form of classes, which then get processed to their form which is actually used by other parts of the software. The used classes have the *Processed* suffix compared to their storing counterparts. In the figure 3.1 we see which class holds which data and how they are structured, the diagram however omits the class duality for clarity.

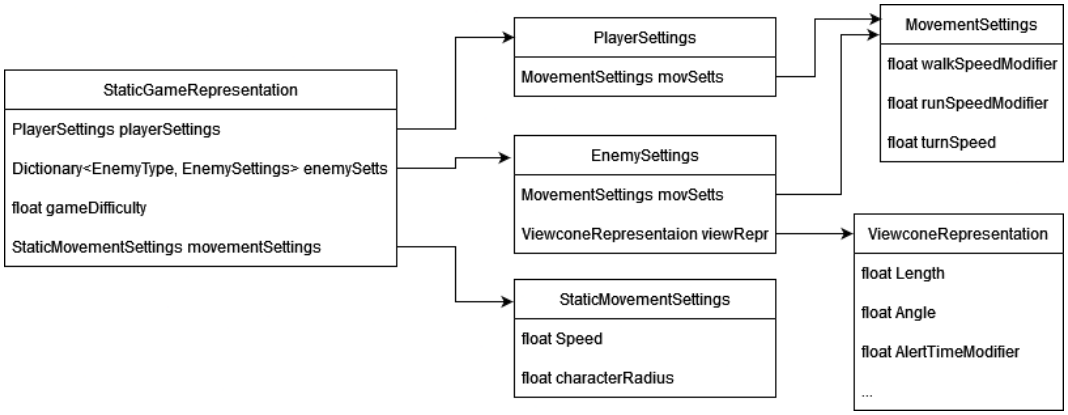

Figure 3.1 The data classes of the game rules

The next data structure to understand is the level representation data, which are in the namespace of *GameCreatingCore.LevelRepresentationData*. They hold the positions and shapes of obstacles as well as the starting positions of the player and enemies, for which they also hold their patrol routines. We can see the data structure displayed in the figure 3.2.

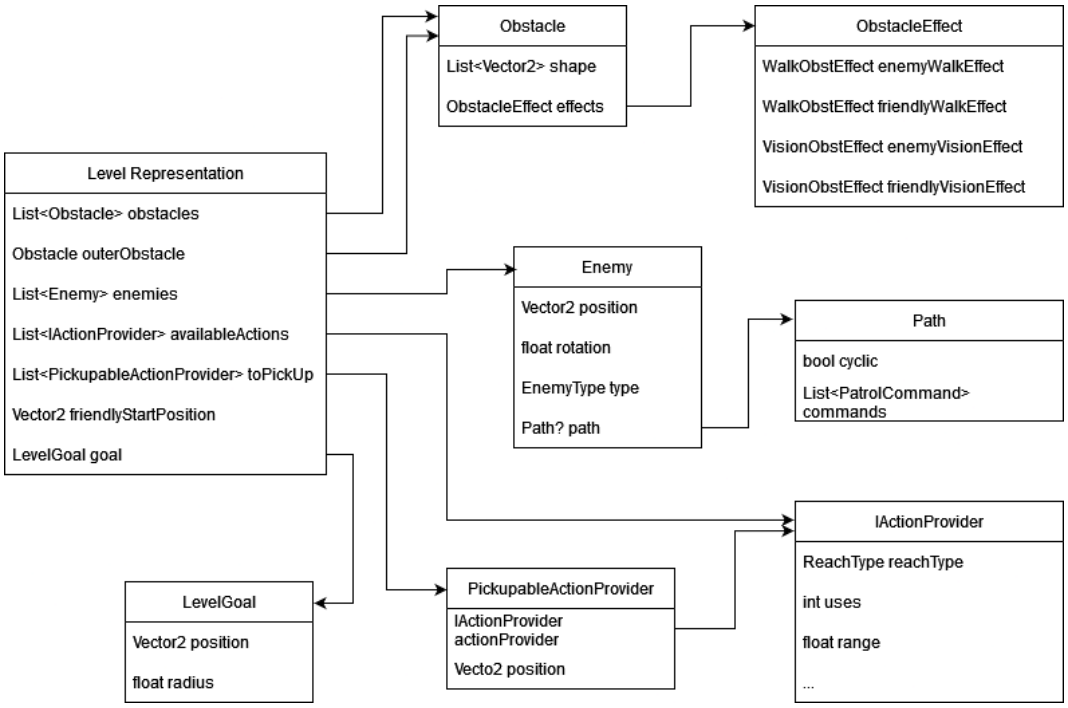

Figure 3.2 The data classes of a level representation

The last data structure regarding the game data is the *LevelState* in the *GameCreatingCore.LevelStateData* namespace. Those hold the data of the current state of the level, mainly the player and enemy positions, which skills are available to the player and what actions are the enemies currently doing. Also if there are any actions, which have a longer-lasting effect without occupying the character who initiated them, such as projectiles, the Level state remembers those as well. We can see the structure in the figure 3.3.

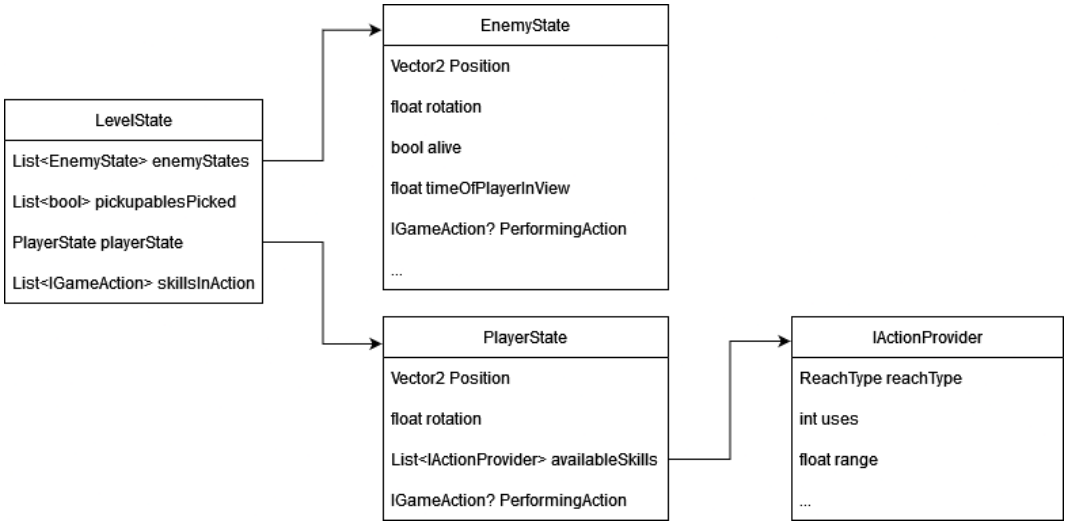

Figure 3.3 The data classes of a level state

3.1.2 Game actions system

Almost every change from game state to game state is made through game actions, specifically through classes implementing the *IGameAction* interface. These changes happen from the *GameSimulator* class, which can simulate the game for a given time window of arbitrary length. The game actions have two separate stages, both of which can be omitted while defining an action. In the figure 3.4 we can see what methods and properties game actions have.

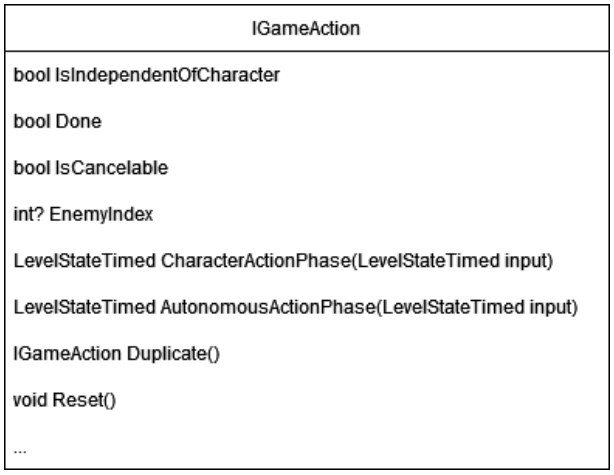

Figure 3.4 The properties and methods of game actions

The two methods of the game actions, which can alter the game state are the *CharacterActionPhase* and the *AutonomousActionPhase*. They both can change any data within the level state. The character action phase is meant to signify what a character is doing, whilst the autonomous action phase is the result of his actions doing something. A good example would be a character drawing a bow as the action phase and the arrow flying as the autonomous phase. The actions also know which character initiated them, which is the *EnemyIndex* property, where null signifies the player. Some actions require the character to finish them,

which would for example fit a potential jumping ability. Such actions have the *IsCancelable* sign set to true.

Since the library is meant to support level solving, it is important that any given level state can be duplicated and branched into multiple different scenarios. For this reason, the game actions need to implement the duplicate function, which creates an exact copy of the action including its internal state. Without this feature, the action's internal state would change while searching one potential branch of level states and the action would not finish properly in the other branches.

Because of the level-solving feature, it is also necessary to support the reset function. When the level-solving actions for the player are found and are to be returned to the user, they were just used to alter the game state and so their internal state has to be reset so that they function properly again.

It is also worth noting, that various actions implement the *IWithInnerActions* interface, which means that they themselves do not change the level state, they only provide a wrapper for other actions. The already implemented ones are:

ChainedAction which chains a list of actions of a single character one after another

TimeRestrictedAction which executes the underlying action for a given time-frame and then ends

StartAfterAction which wait for a given time and then executes the underlying action

This design allows any action nesting, which proves very convenient as we will see in section 3.1.4. The *ChainedAction* is also used for walking. Since the walk action moves a character in a straight line from A to B, the *WalkAlongPathAction* inherits from the chained action and assigns the underlying action to be multiple walking and turning actions that make the given character walk along the given path.

Another aspect of the game actions part of the software is the *IActiveGameAction-Provider* interface. It represents an action someone can do, whilst *IGameAction* represents an action they are doing. For example "someone can throw a rock" would be an action provider, which can create the game action of "the player is throwing a rock at the position (0,0)". The action provider can also have a limited number of uses.

These action providers are used as player starting skills, player available skills, and as part of the skills the player can pick up within the level.

The last part of the game actions are the *PatrolCommands*, which represent a part of an enemy routine. They are basically a middle ground between action and action providers, inside them, it is precisely known what the action is, they are just not connected to any enemy character. Returning to the previous stone-throwing example, the patrol commands can be represented as "some enemy can throw a rock at the position (0,0)".

This may be counter-intuitive since every command is created for a specific enemy since they belong to one specific routine of one specific given enemy.

However, as much as the enemy is an integral part of the commands, also the commands are an integral part of the enemy. So it is an arbitrary choice which one will contain the reference and which one an index. When we understand this, we find out that it is much more convenient to index enemies from commands than commands from enemies.

Pathing

As the player defines the target location of the playable character, as well as the enemy has some arbitrary position set for their commands, it is necessary to be able to find the shortest path from point A to point B within the game level quickly. It will also be very handy to have a quick way of creating the navigation graph necessary for path-finding. We will get to the reason in section 3.1.4. For all those reasons a dedicated path-finding system was created within the library, which also includes the navigation graph creation.

The path-finding algorithm of choice was the Dijkstra's algorithm. This same algorithm is used for all traveling objects, however, the graph underneath is different since various obstacles can be walkable and unwalkable for enemies and the player respectively.

As the vertices on the graph inputting the Dijkstra's algorithm, we use the corners of the obstacles, which the given graph is to be created from. We also have to keep in mind, that if we do not want the characters to visually interfere with the obstacles, we cannot use the obstacles as they are stored in the Level representation, they have to be inflated beforehand. This inflation happens within the *ObstaclesInflator* class in the *GameCreatingCore.GamePathing* namespace. The implications for the resulting game are described in section 3.2.2. We can see the resulting navigation graph in figure 3.5.

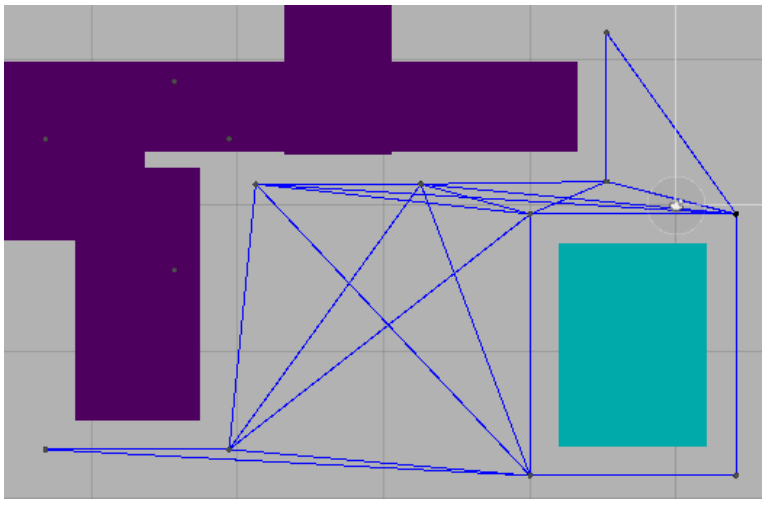

Figure 3.5 A navigation graph on an example level

As the starting and ending positions are in a generic case not on the graph, we have to find all the reachable points to the starting point and check reachability to the ending point from every vertex within the graph. There are no further deviations from the regular Dijkstra's algorithm.

When creating the graph, we check for every pair of vertices and if they are reachable in a straight line. To do that we have to find out whether the line between them has an intersection with any of the obstacle perimeters and also if the line is not inside any obstacle, which would happen if one of the obstacle corners was inside another obstacle. That is allowed within the library.

The graph itself is a generic class, which contains a list of edges and a list of vertices. It also computes the adjacency matrix if prompted to and holds a sign indicating, whether it is computed or not. The figure 3.6 shows different types of edge information and nodes and what data they contain.

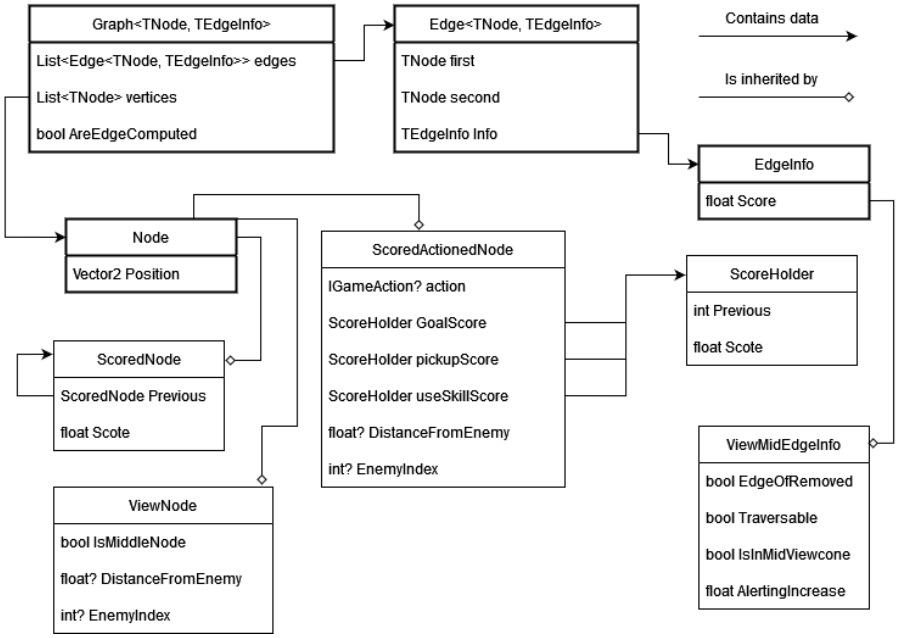

Figure 3.6 The inheritance of Graph classes and their data

3.1.3 Game simulation

One of the most important features of the library is to simulate the run of a stealth strategy game. That includes enemy movement, player actions, autonomous actions, and determining enemy alerting. This all happens within the *GameSimulator* class in the *GameCretingCore* namespace. Since it is a very crucial part of the game, we will go through the whole process in this section. It is illustrated in pseudocode 1. The respective parts are then described below.

Algorithm 1 SIMULATE(state, playerActions)

```
1:  state ← DUPLICATEACTIONS(state)
2:  actions ← []
3:  if state.player.noncancallableAction ≠ null then
4:      actions.ADD(state.player.noncancallableAction)
5:  actions.ADD(playerActions)
6:  actions.ADD(longNoAction)
7:  allStates ← [state]
8:  for all act in actions do
9:      newState ← act.CHARACTERPHASE(state)
10:     usedTime ← state.time − newState.Time
11:     newState ← REMOVESKILLSWITHNOUSESLEFT(newState)
12:     newState ← UPDATEENEMIES(newState, usedTime)
13:     for all autoAct in state.ActionsInPlay do
14:         newState ← autoActAUTONOMOUSACTION(newState, usedTime)
15:     newState.ActionsInPlay.REMOVE(a ⇒ a.Done)
16:     newState ← PLA(newState, usedTime)
17:     allStates.ADD(newState)
18:     if newState.time == 0 then
19:         if notact.Cancellable then
20:             newState.player.noncancallableAction ← act
21:         break
22:     else
23:         if act.IsAutonomous then
24:             newState.ActionsInPlay.ADD(act)
25:         state ← newState
26: return UPDATETIMEOFPLAYEINVIEW(allStates)
```

As mentioned above, we are aiming to allow level solving so before any action takes place, we need to duplicate all actions within the level state (line 1).

We have to add one long empty action into player actions (line 6) so that enemies and autonomous actions are updated even when the player does nothing.

For enemy update (line 12) we again have to consider if it has some non-cancellable action from the previous state if the enemy is going forwards or backwards within the patrol and more.

It is important to note, that the autonomous actions(lines 13-15) are part of the level state and can be updated from other actions. For example, when a character starts turning, it removes all previous autonomous turning actions. For that reason, we have to be careful to run all actions we should and none that we should not.

We also have to compute for every enemy what time the player has been in its view for and update the value within the enemy state (line 26).

Even though in classical stealth strategy being caught by an enemy typically only triggers some alarm and lets you try to escape, in the case of our game it does end the game. For that reason, there is no mechanism implemented, that would allow the enemy to set the alerted sign back to false. However, the game core is very scalable in that regard and some features of that kind may be added.

Viewcones

When computing whether the playable character is within the player view, it is necessary to compute the shape of the enemy viewcone. This process happens within the *ViewconeCreator* class in the *GameCreatingCore* namespace.

As a constructor parameter for this class, we pass the number of rays to cast inside of the view. Such a ray is of a given length since we know the maximal length of the viewcone. So we can consider it just a straight line. Then we compute if this line has any intersection with any obstacles the enemies cannot see through. When all rays are cast, the resulting points of intersection or potentially the points at the max distance are the ending points of the view. To get the full shape of the viewcone, we also add the enemy position.

To save computational resources, we remember this computed viewcone in a dictionary, where the keys are enemy states. We only need to consider where the enemy is, what rotation it has, and if it is alive. This optimization is very useful while playing since many enemies do not move and so their viewcone shape is the same the whole time they are alive.

3.1.4 Level solving

The library has a build-in level-solving feature. This section will look into how level-solving is executed on the implementation level. There are two different path-finding systems implemented. One very trivial one, which finds the path to the goal disregarding all enemies, the other one is much more complex, since it factors the enemies in. They both implement the *IGamePathSolver* interface. Any new solver that may be potentially added is expected to use this interface as well.

At the start, we can easily say, that in a level without any enemies, the path can be found by path-finding on the graph with obstacles that obstruct player movement. To improve this method, the navigation graph is computed from the goal and the score to the goal is kept within the nodes. This way the navigation only finds which reachable point has the lowest score after adding the distance to the score and the rest is already covered. This technique uses the *ScoredNode* class to create the navigation graph and it is used by the *NoEnemyGamePather* class.

The other level solver includes enemies and time into the equation. It is within the *FullGamePather* class. In short, it creates a tree of level states, takes the best of the states in the tree leaves, finds possible actions out of that state, simulates them on the current state, and adds the new states into the tree. We can see the algorithm in psedocode 2.

Algorithm 2 GameSolve(*LevelRepresentation* level)

```
1: if inflateObstacles then
2:     level ← ObstaclesInflater.InflateObsts(level)
3: noEnemyPath ← NoEnemyGamePather.GameSolve(level)
4: if noEnemyPath == null or level.Enemies.Count == 0 then
5:     return noEnemyPath

6: viewconeGraph ← new ViewconeNavGraph(level)

7: simulator ← new GameSimulator(level)
8: state ← WaitInitial(initialWaitTime)
9: return GameSolveRecursion(state, 0)
```

At line 6 we initialize the *ViewconeNavGraph* class. For now, let us consider it a black box, that can create a graph that includes the enemy viewcones and possible actions.

And at line 8, we do the initial wait. Basically, if an enemy sees the playable character at the level start, we want to give the player a time window bigger than what the solver can do. So we force the solver to wait before it starts the actual solving.

The algorithm returns the result of the *GameSolveRecursion*. This recursion goes to a deeper level whenever an enemy dies or whenever some of the pickupable skills is picked up. This way we separate the scoring in the previous recursion level from the one with some of the objectives achieved.

Now let us see what the recursion does internally. It creates a priority queue of *StateNode*s (see figure 3.7) and puts in the input state.

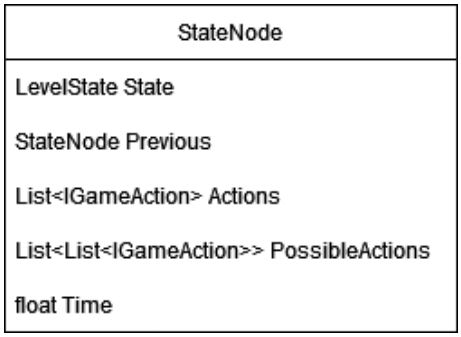

Figure 3.7 *StateNode* class with its internal data

We can see the whole algorithm illustrated in the pseudocode 3.

Algorithm 3 GameSolveRecursion(inputNode, iterations)

```
1: queue.Enqueue(0, inputNode)        /* We initialize the priority queue. */
2: while queue.Any() do
3:     currentNode ← queue.DequeueMin()
4:     if IsGoalAchieved(currentNode) then
5:         return currentNode
6:     currentTime ← currentNode.Time + timeStep
7:     if currentTime ≥ maxTime then
8:         return null
9:     iterations ← iterations + 1
10:    if iterations > maxIterations then
11:        return null
12:    for all actions in currentNode.PossibleActions do
13:        newState ← Simulate(currentNode.State, actions, timeStep)
14:        time ← currentTime
15:        /* If the player has some non-cancellable action, we run the simulator
           to finish it, since the next states would have to wait for it anyway. */
16:        if newState.PlayerNoncancellableAciton ≠ null then
17:            actions ← newState.PlayerNoncancellableAciton
18:            timeAdd ← actions.TimeLeft
19:            newState ← Simulate(newState, actions, timeAdd)
20:            time ← time + timeAdd
21:        /* We compute the graph including viewcones using the black box
           defined above */
22:        graph ← ComputeGraph(newState)
23:        possibleNodes ← GetReachableNodes(graph, newState)
24:        /* We have to limit the branching factor to some user defined value,
           since the tree is huge */
25:        LimitBranching(possibleNodes, maxBranching)
26:        score ← ComputeScore(possibleNodes)
27:        newPossibleActions ← NodesToActions(possibleNodes)
28:        newNode ← new StateNode(score, newPossibleActions, time)
29:        if ShouldStartNewRecursion(newNode) then
30:            result ← GameSolveRecursion(newNode, iterations)
31:            if result ≠ null then
32:                return result
33:        else
34:            queue.Enqueue(score, newNode)
35: /* We have not found any actions from the given input state (or all states
    created from the input state), so we return null. */
36: return null
```

The described algorithm searches through the tree of level states and if it has enough iterations and in-game time available, it finds the path to the goal, if it exists. It does not guarantee the best path. That is because we go into a deeper level of recursion as soon as the node which should go into the deeper level is created. Going back to the pseudocode 3, lines 29-32 are at their place referencing *newNode* and not between lines 3 and 4 referencing *currentNode*.

That is however necessary. The sole purpose of the recursion is to not let the scores after doing an important action interfere with the scores before doing it. If we added the node into the queue, its score would represent the score after the important action took place and so the invariant would break.

To fully show that the algorithm works, we also have to show that the branching does not take up all of the iterations before it gets into the next recursion, or to the goal respectively. That is very much dependent on the score that the nodes have in the priority queue (line 26). We compute the score to be very slightly dependent on the time stored in the node and very heavily dependent on the score of the possible actions. We will answer this one later. Let us first focus on a different question.

That is, why it is necessary to separate the scores before and after the important action. The explanation will be possible after we look into the creation of the graph with viewcones, which is the black box we established earlier in this chapter.

It is important to note, that the algorithm requires a different graph with viewcones every tick because the enemies can move. However, since the enemy actions are predefined and the time step is almost fixed, we only need to compute the graph for most timestamps once. Only when an enemy dies or a skill is picked up, it has to be recalculated. Moreover, the black box also stores the viewcone shape for enemies given their location and rotation, so when an enemy is standing on one point multiple ticks, his viewcone shape is only computed once.

When considering the viewcone as a shape, it is not strictly non-traversable. The playable character can step inside, it just has to be there for a short enough time. For this reason, we cannot consider the viewcone as some kind of an obstacle.

In short, we discretize the viewcone to a new separate graph with directed edges based on where it is possible to traverse. Then we take all viewcones and cut off what common parts they have with each other, we add the pickupable skills as vertices into the graph, we add the available skills as vertices into the graph, we run scoring of all those vertices and then we get the graph. Now we will look into how all of these steps are implemented.

Viewcones discretization

We can consider the viewcone shape as a set of ending points and one point at the position of the enemy. If we take the first and last ending points, there will be a graph edge towards the position of the enemy. We can compute, how far could the player go on this edge if it was inside the view so that they wouldn't be caught by the enemy. At this distance, we add a point on each side of the viewcone. Then we take a set distance and cut the viewcone end including these two new points by this distance. This process is shown in figure 3.8.

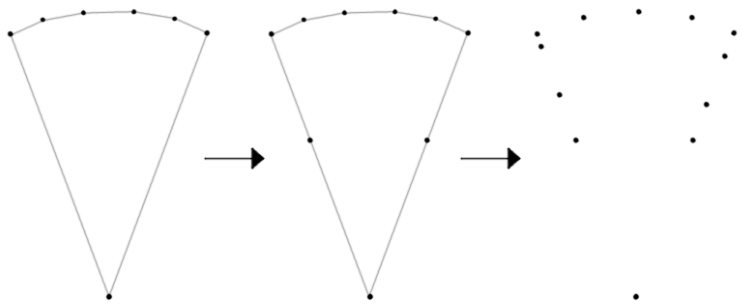

Figure 3.8 Discretization of an enemy viewcone

Even though this process may seem pointless when the number of rays is not large, it has multiple advantages. We can set a high number of rays, which creates a very precise shape of the viewcone, and then reduce the number of vertices in the resulting graph. We can also point a small number of rays and still have fine discretization. We basically separate the complexity of the path-finding from the precision of the viewcone shape.

On the newly created set of points, we create one-directional edges between vertices, where the playable character has time to walk through without being caught by the enemy. We need to remember, that all edges on the viewcone perimeter are walkable since the enemy cannot see there. This way it is possible to go around the viewcone.

Intersecting viewcones removal

If there was only one single enemy in the level, then the viewcones would be ready now to be merged with the navigation graph. However since it is possible, that multiple viewcones intersect each other, we cannot merge them yet. It would possibly add some edges inside the viewcones, which are not walkable. In figure 3.9 we can see that adding two viewcones one on top of the other possibly creates an illegal path (marked in red) within the navigation graph.

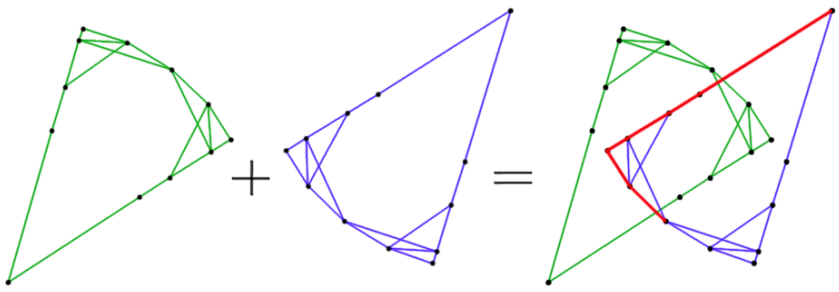

Figure 3.9 Combining viewcones with illegal path visualization

For this reason, it is necessary to adjust both viewcones. Going back to the figure 3.9 we can see, that there were actually two illegal paths created, one on the blue viewcone and one on the green one. So that proves that adjustments are possibly necessary for both viewcones.

The adjustments however have to be very delicate, to let all edges which are possible to travel there and remove the nontraversable ones. It is also important to consider cases that are more complex than the simple one portrayed above. However, let us now stick with this simple case and deal with the more complex one later. We need to find the intersections of the viewcone perimeters and remove all nodes between such intersections. Then possibly add an edge between them if the distance between them is short enough for the player not to get caught. We can see the desired result in the figure 3.10.

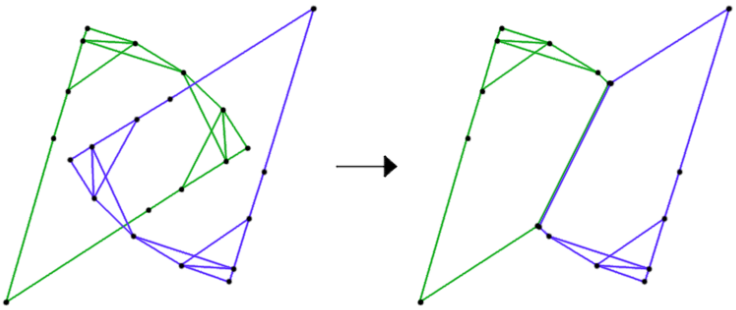

Figure 3.10 Desired result of viewcone combination

Now we get to the basic implementation principle. Let us consider a pair of two viewcones. We iterate over the vertices of one of them and if we find out that some of the vertices are inside the other viewcone, we store their index and find out the first next index that is outside the viewcone. This way we find the two edges, which have to intersect with the other viewcone. Let us call this edge *Es*. In this step, we also compute the intersection points.

Then we iterate over the perimeter edges of the second viewcone and find out which of them have an intersection with *Es*. This way we find the two edges in the second viewcone, which contain the intersection.

Next, we consider the viewcones separately, let us focus on the first one now. We simply remove the vertices, that are inside the other viewcone, remove edges that end or start in them, and add to vertices in the intersection points. We also add edges from and to these points and find out, if it is possible to travel on the edge between the two intersections.

When focusing on the second viewcone, it is more complex. The first complication is, that we know the two edges which have the intersection on them, but we do not know which part of the perimeter is contained in the first viewcone. Checking if the vertices on the perimeter are inside the first viewcone will not help, because the viewcones can actually cross each other, not just have some part of themselves in the other viewcone. Two such situations are illustrated in the figure 3.11.

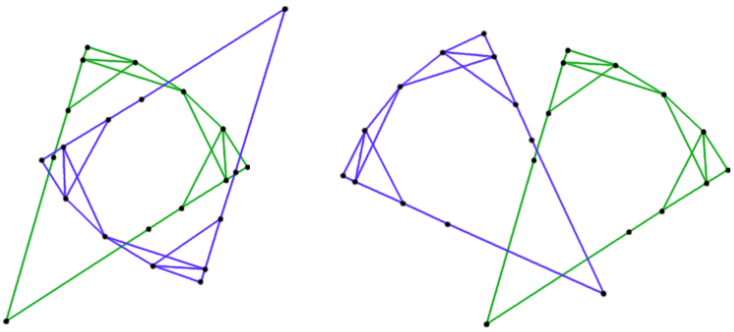

Figure 3.11 Viewcones crossing each other

To overcome this challenge, we look in the direction of the ending and starting points of the edges with intersections. This way we find out in which direction the first viewcone really is. However, we cannot simply remove all vertices between the two edges. It is very apparent from the figure 3.11 above. Some of the vertices outside of the viewcone can be removed.

So firstly, we check if all the vertices in between the intersections are inside the first viewcone. If so, we simply remove them just as we did with the first viewcone. If not, we found two sections that are inside the first viewcone. Then we remove those two separately.

We also should consider what happens if more than two viewcones intersect somewhere. However, we find out by simple observation, that if we do the pairwise viewcone intersecting area removal, the final result will work for any number of intersecting viewcones.

Pickupables and available skills

To form the final graph, we will also need the points where pickupable skills can be picked up and points where available skills can be used. It is very simple with pickupable skills since they have their position stored within the *LevelRepresentation* data. However available skills are different. Every available skill has its specific reach type, use range, and target type. The target type has four options, which the skill can target:

- The playable character himself (for example healing, speed...)
- An enemy (for example melee kill, shortening viewcone...)
- The ground, including obstacles (for example some distraction)
- The ground, excluding obstacles (for example some teleport)

Every skill has a *Start* and a *Target*. The *Target* is the location or the character, which the skill will affect. The *Start* is the location where the player will use the skill from. Both of them depend on the target type, we can see how they are calculated in the following list based on the target type.

Player - the *Target* is the player, the *Start* is the current player location

Ground - we form a circle around the player with radius equal the range of the skill. Then we create a user-defined number of *Targets* evenly distributed on the circle. The *Start* is the current location of the player. We can see the process illustrated in figure 3.12.

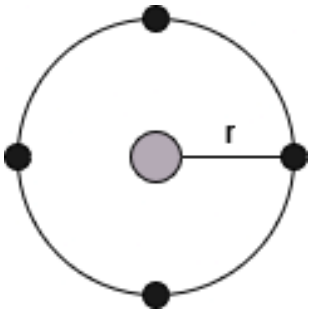

Figure 3.12 Created targets around the player with use range r

Enemy - We create the circle as shown in figure 3.12 around every enemy, but the locations on the circle are now *Starts* and the *Target* is the given enemy.

The reach type only describes through what the skill can be used. It either can be used through obstacles or not. The following code describes whether a *Start-Target* pair will be added to the graph:

Algorithm 4 Reach Type requirement to add a pair

```
if ReachType == CanGoThroughObstacles then
    return True
else
    return not IsPathThroughObstacle(Start, Target)
```

This concludes the possible target types and how they influence the *Start* and *Target* selection, as well as the reach types and what they require from a pair to be added to the graph.

Combining the navigation graph and the viewcones

Now that the viewcone has no more common areas, and we have the pickupable and available skill points, we can unify them all with the navigation graph. We have all the vertices that will be in the final graph, however, there are many edges missing and possibly many excess edges as well.

First, we find out if there are any obstacle corners, which are inside of enemy views, this is especially probable, when the obstacles are inflated, because obstacle inflation only affects the unwalkable obstacles, the vision is not affected and so the corners of walkable obstacles naturally poke inside the viewcones. The vertices inside viewcones are going to have all edges from and to them removed and will be treated as an integral part of the viewcone, so they will only have edges to other vertices inside the given viewcone, nowhere else.

Then we select such edges from the navigation graph, that are not obstructed by any viewcones, only those are going to be passed to the final graph from the navigation graph.

Next, we add the pickupable and skill use points into the new vertices. From now on they will be treated just if they were an integral part of the previous graph. Only we store their indices within the final vertices list so that we can access them later.

Now we add vertices and edges from all the viewcones. It is important here, that we combine vertices that are at the same points, which happens a lot in the case of viewcone intersection, where the intersection points create two vertices each - one in both viewcones. We also combine edges that go from the same vertex to the same vertex. This way we find out if an edge in two intersecting viewcones was traversable in one but not in the other. Such edges are not propagated into the final graph.

Finally, we add edges between the viewcone vertices and navigation graph vertices. There we have to again check, whether they are not obstructed by any obstacle and also by any other viewcone.

Combined graph scoring

Now that we have a graph that contains all the viewcones, pickupable skills, and usable skills, we have to score every vertex to know how good it is to go there. However, we actually do not create one score for every vertex, we create three of them. One for reaching the goal, one for reaching skill use and one for skill pick up.

For scoring, we again use the Dijkstra's algorithm. However, here we cannot simply take the distance of the edge's start and end as the edge score, the edge may be inside of a viewcone, and in such cases, we raise the score so that passing through viewcones is penalized.

To find the goal reachability score, we start at the goal and find out all vertices that are reachable and what score they have. We store those scores. Then we do the same for the vertices that the playable character can pick up objects at. Finally, we do it for the points it is possible to use skills at. This way we get a graph, where each vertex has up to three scores and for each of these scores the edge also stores the previous vertex on the best path.

Final notes

Now that we understand how the graph is created and how the final data looks, we can come back to the previously asked question: Why is it necessary to separate the scores between recursions? (see the question)

Well, it is because we have to choose the score of vertices as one number. So when scoring a vertex during the level solving, we take preferably the score to goal, then the score to skill use, and finally the score to skill pick up. This is defined within a single function which can be very easily changed to be an input of the user.

When we consider this, we can see, that after an enemy dies or after a skill is picked up, we have either picked up or used a skill and so other state nodes that are close to executing that exact skill are close to a score of 0. So if some new goal arose somewhere far, the scores within the current queue would be close to zero and it would take a long time until the newly discovered goal would be

explored. For this reason, we go into deeper levels of recursion and use a new queue to not let the previous low scores make the current new goal wait.

Now we can also answer the question about branching: Why does the algorithm go into a deeper level of recursion or to goal instead of branching until it has no more iterations left? (see the question)

We have three scores and when any of them gets to 0, we either go into a deeper level of recursion or we got to the goal. So when the algorithm gradually lowers the score inside the priority queue, it will inevitably get to a node with a score of 0. That is because the character speed is linear and so the downwards tendency of the score is linear as well. However, a score of 0 will lead to a deeper level of recursion or the goal as already said above.

3.2 Playable demo

Even though the game library (3.1) represents a full stealth strategy game core, building it into a game without a game engine would pose many graphical, logistical, time, and other challenges. With this in mind, it was chosen, that the game core library will be used in an existing game engine. Specifically Unity 2022.3.11f1. Unity already handles game the loop and graphics. The code for Unity is also written in C# which further simplifies its interconnection with the rest of the software created within the project.

In this section, we will dive into the implementation of the game. It was created to demonstrate the capabilities of the game core library and also to conduct research (4.2) focusing on the stealth strategy game genre. It is important to consider the game with this in mind.

There are two very distinct parts of software, that were created within the Unity game engine:

The first one is the part that creates game levels using evolutionary algorithms. This one interconnects the evolutionary algorithms library and the game core library. It uses game simulation and the automated level solution finder both implemented within the game core library. This part of the software is only used in design time and does not change the user gameplay experience.

The second part is the playable game itself. It takes either procedurally or manually generated levels and turns them into a playable demo game.

3.2.1 Level generating

The software part within Unity that generates levels is not large. Its most important functions are the mutation function and the fitness function. Those are very crucial for the evolutionary algorithm to successfully create good results.

In this section, we will see a brief overview of the classes that combined create the procedural generating functionality. Followed by the implementation of the crucial parts mentioned above.

In the level-generating part of the game, there are only 8 classes, so this overview is going to be very brief. The main class is the *EvolAlgoGenerator* class as apparent from the figure 3.13.

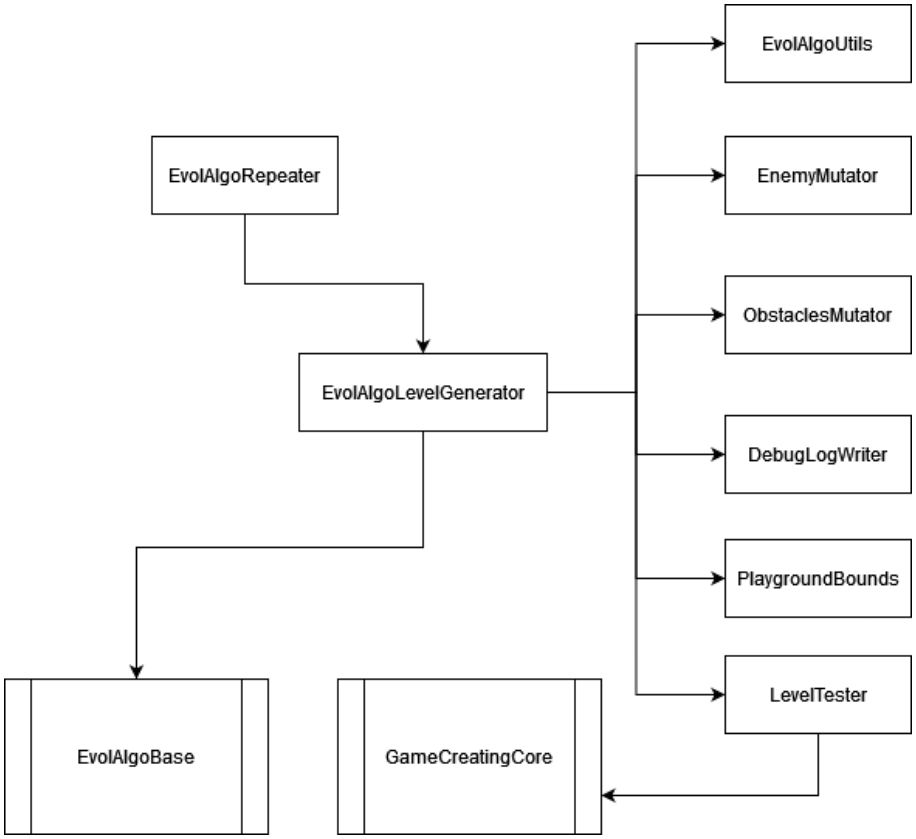

Figure 3.13 Class dependencies within level generating part of Unity game

EvolAlgoGenerator calls the evolutionary algorithm itself while creating all of its necessary parameters. It uses the *LevelRepresentation* class from the *GameCreationCore.LevelRepresentationData* namespace as the individual. It also uses the mutation classes (*ObstaclesMutator*, *EnemyMutator*), which define how obstacles and enemies are changed from generation to generation within the evolutionary algorithm.

The *EvolAlgoGenerator* class also sets the initial population. However, the implementation within the software creates empty levels without any obstacles or enemies. The reason is that creating a valid starting level is a hard problem and implementing another new function for the initial population would be nontrivial.

The crossover function is also defined by the *EvolAlgoGenerator* class, but it is empty. In the case of complex levels with enemies and obstacles in continuous space, the interchange of data between individuals would lead to a huge downward leap in scores for both resulting individuals. For that reason, no crossover was implemented.

To have the level bounds set by the user, there is the *PlaygroundBounds* class which graphically shows the user where the level generation will occur. This is very useful to create levels that feel natural with their spacing given player speed and size.

Since the evolutionary algorithm gives the option to reflect the current state after each generation, there is the *DebugLogWriter* which writes to the Unity debug console. It is however not updated until the full generating is done. Because of that reason, the main purpose of this class is purely for debugging.

The *LevelTester* class lets the user see a layout of a chosen level without

entering the Unity play mode and it also scores the levels, which is used as a fitness function by the *EvolAlgoGenerator* class.

The *EvolAlgoRepeaterClass* only runs multiple sessions of the level generating each with a different random seed.

Fitness function

The fitness function located within the *LevelTester* class scores a given level. It calls the level-solving part of the *GameCreatingCore* library, specifically the *FullGamePather*'s *GetPath* method (see 3.1.4) and processes the outputted actions into a single floating point number representing the level. Within this function, it is crucial to capture the goal of what we expect to see in the level.

Now lets carry through the scoring function visible in pseudocode 5.

Algorithm 5 Score(level)

```
1:  playerActions ← LevelSolver.GetPath(level)
2:  if playerActions == null then
3:      return 0                                   /* We found no path */
4:  divisor ← level.Obstacles.Count * 1
    +level.enemies.Count * 3
    /* A user-defined count is added to the divisor so that levels with obstacles
    are not punished too much compared to levels without them. */
    +minimalDivisor
5:  /* Path length is almost equivalent to the time spent in level as shown above. */
6:  pathScore ← playerActions.Count/divisor

7:  actionBonus ← 0
8:  for all action in playerActions do
9:      if not action.IsCancellable() then
10:         if actionisKillAction then
11:             actionBonus ← actionBonus + killBonus
12:         else if actionisWalkThroughViewcone then
13:             actionBonus ← actionBonus + throughViewconeBonus
14:         else
15:             /* We do not know what action it is, but since it is non-cancellable,
                it is probably interesting. */
16:             actionBonus ← actionBonus + genericBonus

17: multiplier ← 1
18: if notplayerActions.IncludeKillAction() then
19:     multiplier ← killNotUsedMultiplier
        /* This lowers the score for the levels, which allow the player to kill
        enemies even while it is not necessary, since the path does not contain any
        kill action. */
20: return actionBonus + pathScore * multiplier
```

The approach of adding fixed values(lines 7-16) creates some very strange behavior within the evolutionary algorithm. Levels that have the solved path

with some specific interesting action naturally get much higher scores and so they propagate within the whole population very quickly making most of the levels within one evolutionary algorithm run similar.

Even though it is logical to do the approach on lines 17-19, it also poses a similar problem to the evolutionary algorithm as described above. If the level allows the player to kill someone, it is punished heavily for not killing anything and as soon as one level appears which solved path contains killing, its score gets much higher than the score of the other levels and it spreads through the population. For this reason, even though the software was created to support it, the value of the score modifier for not using killing was set to 1 within the experiment.

Mutation function

The mutation function is actually in the case of this software a set of functions. It provides a system of how to randomly change a level and create a different level. It aims for slight changes, which result in slight score shifts so the score can gradually increase with the right mutations applied.

This aim is fulfilled with the Obstacles mutations, however, changes to the enemies can have a drastic impact on the score so changes to the enemies are in essence smaller just to lessen their impact on the score.

The enemy mutations can change enemy position and rotation slightly, also add or remove points on a given enemy path and change those points slightly. It can also remove and add enemies, with probability provided by the user with respect to the current enemy count in the level. This approach gives the user full control over enemy count in the level. It also however makes it more probable that multiple levels will have the same count of enemies.

The obstacles mutation function can add and remove obstacles, it can also change their position slightly and change their effect. The obstacles can be walkable and unwalkable for both enemies and the player character and see-through or non-see-through for enemies, which this mutation function can change with probability given by the user.

When new obstacles are created, their area has to be in some given range (provided by the user) and they cannot be too thin in any dimension. Even though the game creation core library supports any concave obstacle shapes, this mutation only operates over rectangle shapes, but in the full continuous scale. Outer obstacle - the level bounds - can also be mutated by this function.

The mutation function also only makes valid levels. This is done by providing the obstacle mutation function with the current position of the player, goal, enemies and enemy path points so that obstacles are never mutated in a way that they would contain some of the points they should not. Similarly, the enemy mutation function is provided with the current shape of obstacles, so that enemies are never mutated inside the obstacles.

The player character's starting position, the level goal position, skills available to the player at level start and skills that the player can pick up within the game are not mutated within this function. Even though it could be implemented, they change the level too fundamentally and so their mutation is not desired. Moreover, they are often to be set as starting parameters for the generator, since they are

an important integral part of the overall designing process, possibly driven by the game story.

3.2.2 Game

Within the thesis, a playable demo of a stealth strategy game was implemented. The resulting gameplay can be seen in figure 3.14.

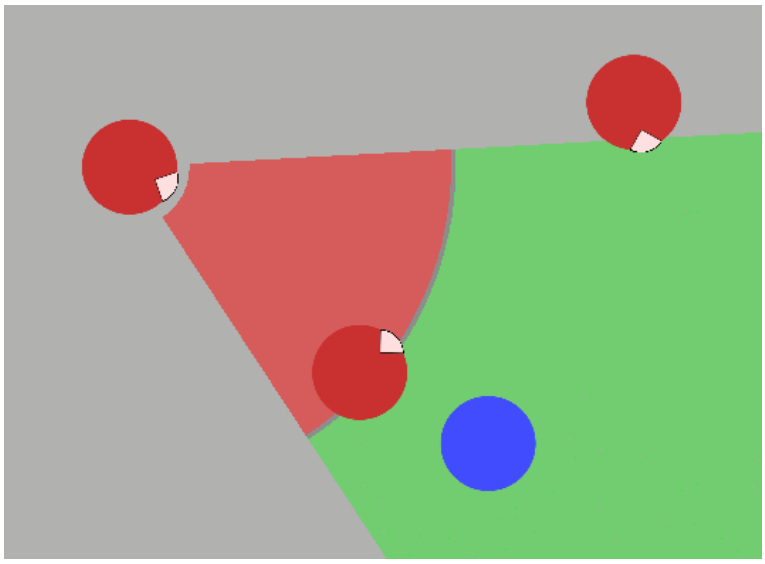

Figure 3.14 Example gameplay from the game demo

Within a level, the player controls the playable character(blue) with their mouse. If the kill option is available within the level, they can choose enemies(red) to kill. They can also display or hide viewcones of enemies. Their goal within a level is to reach a given destination without being detected by the enemies. The whole game is playable and downloadable at https://cyrdach.itch.io/masterthesisresearch.

The rest of this section will focus on the implementation of the portrayed playable demo. A very brief picture is: Every frame, the game calls the game creation library, which moves and turns the enemies accordingly and also moves the playable character towards the location desired by the player. It computes which enemies see the player and returns all of that information to the game as a level state. All of the game objects are then changed to match that state.

To represent a game level, there are various scriptable objects, such as *UnityLevelRepresentation*, *UnityObstacle*, *UnityStaticGameRepresentation*, and more. Those hold the data necessary to run the game within the *GameCreatingCore* library. We can see all of those classes, what other data classes they hold, and which of those classes are in the Unity game and which are in the game creating core library in figure 3.15.

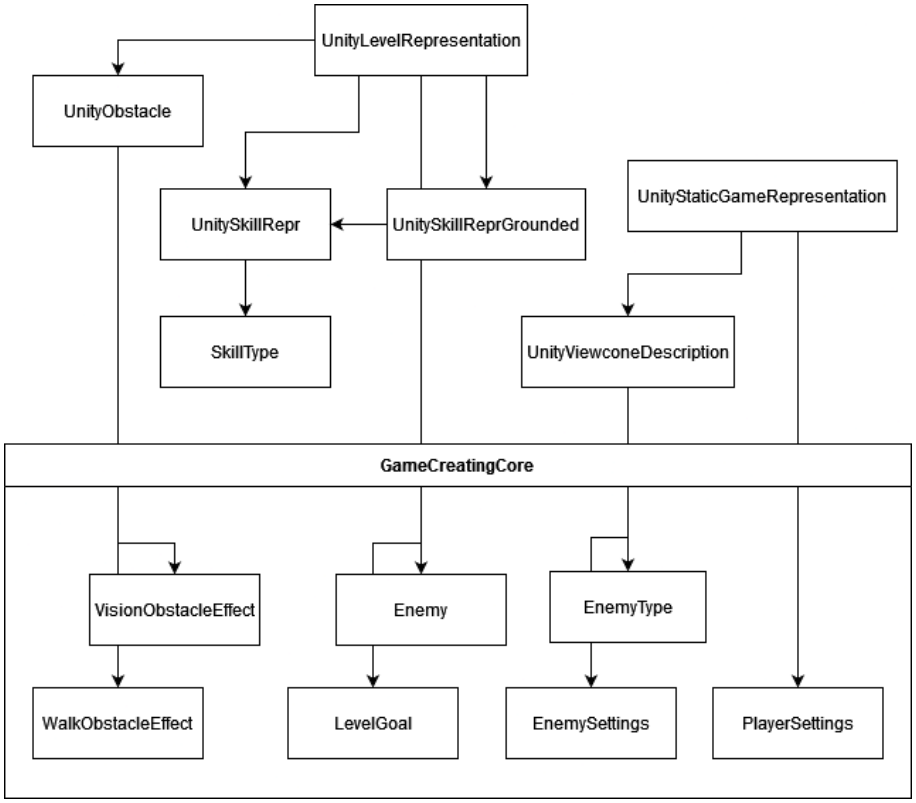

Figure 3.15 Class dependencies within level generating part of Unity game

When the game starts, the *LevelInitializor* class calls the associated *LevelProvider*, which either takes one of the scriptable objects above and creates a *LevelRepresentation* out of it, or it can also generate the level representation itself, just as the *LevelProviderDummy* class does.

The *LevelInitializor* then spawns obstacles and enemies provided by the *EnemyProvider* to the places set by the *LevelRepresentation*. The obstacles are created as new separate objects each with a custom *MeshRenderer*, the enemies are instantiated from prefabs as is.

Then the *GameRunner* class takes the level representation, that the *LevelInitializor* got, and creates static navigation meshes, one with obstacles enlarged(also called inflated) so that characters do not visually interfere with obstacles, and one straightly from the unchanged obstacles. It then calls the *GameCreatingCore* library, specifically the *GameSimulator* class, which at each frame simulates the game. And moves the player and enemies to the right positions. It also finds out if the level goal is achieved and if the game is lost. And lastly is sets all enemy viewcones to display how alerted the respective enemy is.

The *GameRunner* also looks into the *HumanPlayerController* class, which supplies places where the player clicked and possible enemy targets to be killed. Whenever a player makes such input, the *GameRunner* finds out if it was outside of the inflated obstacles, inside them, or inside the unchanged obstacles. If inside the unchanged, no location is assigned to the player, if it is inside inflated, the nearest location outside is assigned. This process is shown in figure 3.16.

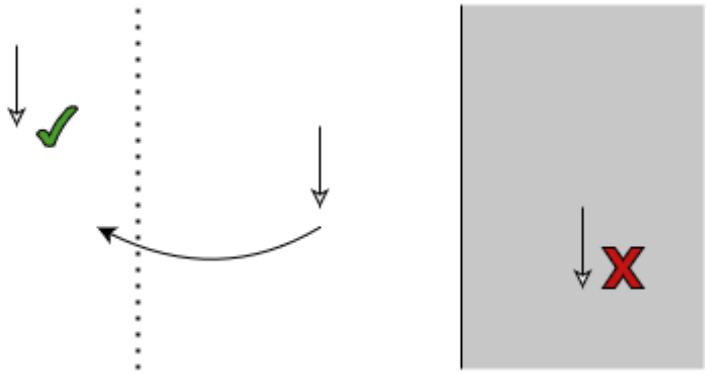

Figure 3.16 Behaviour of player clicking on obstacles

Even though the game creation core library computes the shape of the enemy viewcones every frame, it only creates one continuous shape, but the visualization of the viewcone has to create multiple shapes based on how long the player is inside of that enemy's viewcone. For this reason, the viewcone creation and maintenance are handled solely from the Unity game demo by the *ViewconeCreator* and *ViewconeMaintainer* classes. The length of the viewcone as well as how big portion of it is turned red, which corresponds to how long time the player has spent in the enemy view, is both computed within the library.

3.3 Evolutionary algorithms library

When asking the question if evolutionary algorithms can perform some specific task, it is crucial to have a stable library that can run generic evolutionary algorithms. To ensure its stability, to have full control over the library's internal functionalities, and to mitigate the overhead of unused evolutionary algorithm parts, we created a whole new standalone library for evolutionary algorithms within this project. This chapter will look into its implementation. We will not however explain the evolutionary algorithm basics here, for that see 2.

Creating an evolutionary algorithms library requires to have a main loop. In the case of the library created within this project, it is in the *EvolutionaryAlgo* class. The main loop creates the initial population and then loops over creating new offsprings, mutating and crossovering them, scoring them, and selecting a new population. This process is shown in figure 2.1 in the Background chapter.

With this approach we do not deviate from the basic functionality or implementation of evolutionary algorithms [33], however, we still have full control over the parameters. Those are all gathered within the *EvolAlgoParameters* class. We can see which ones are available and what were their values for the level-generating in the following list:

Population size - how many individuals are in one population

- was 10 individuals for a very long time because most of the individuals were very similar. However then we found out that increasing the population size creates more variety for the given run and creates better results, so for that reason we ultimately used 100 for all levels

Offspring count - how many offsprings are created between populations

- was in 1:1 ratio with population size for all runs

Parent selector - before the offsprings are created, it is necessary to select their parent from the population

- selected every individual once in each generation - this guarantees no unintentional data loss, only works, because offspring count is 1:1 to population size

Breeding - when parents are selected, this function defines how the new offsprings are created from them (includes crossover)

- from 2 parent to 2 offsprings with no crossover, not changing any individual - this is not ideal, but no functional crossover method was discovered

Mutating function - with some probability, the new offspring is mutated to have a bit different data

- it is described in detail in section 3.2.1, however some of its details were changed in between runs as described in section 4

Selection mechanism - after the offsprings are mutated, we have to pick the individuals that will pass to the next generation

- we used roulette wheel selection for all runs, however one of its parameters was changed as described in section 4

Elitism - when the selection occurs, this says how many of the best individuals will be kept from the previous generation

- always kept exactly 1 best individual

Previous population bias - Within the selection, the previous population may be used as well, this signifies how much is their score lower compared to the new offsprings

- was changed in between runs, for details see section 4

Ending predicate - after each generation, we ask whether the algorithm should end and this function gives us the answer

Initial population - when the algorithm starts, it gets the initial population here

- provided empty levels with no enemies and no obstacles

Scoring function - a function that can compute score for any individual

- it is described in detail in section 3.2.1

Plotter - between every generation, this function gets the opportunity to report the data to some outside source

- used to show the time, when each generation was finished
- later also stores the scores of all generations to produce data (see 4)

4 Methodology

In this section, we will look into what methods were used in this project and why. We will also discuss the method specifics and their parametrization within this thesis. First, we will discuss the evolutionary algorithms, their parametrization, and the used techniques. Then we will dive into the survey used to compare the manually created levels to the procedurally generated ones. In both cases, we will also discuss the potential flaws with the used methods and their executions.

4.1 Evolutionary algorithms

In this thesis, evolutionary algorithms were used to generate levels. An evolutionary algorithm is a very broad term, so to describe the method properly, further specification is needed. However, the algorithm implementation is described in section 3.3 and the basic structure in section 2, so no more specifics are needed in that regard.

With this in mind, this section will mostly cover what results the genetic algorithms yielded and what potential changes can be made in further research in this direction. We will also cover the parameters, whose value was changed in between runs. The values which stayed the same are specified in section 3.3.

Even though the unchanged parameters surely can be changed and it could lead to better results, the parameter space is simply too huge to optimize manually on all of its axes. Moreover, the generating itself is a time-consuming process so it would not be possible to fit within the time frame available.

There were more than 50 successful recorded runs of the evolutionary algorithm, each resulting in 10 to 100 created levels. It has to be admitted, that some of those were pure test runs.

The time each run took varied a lot depending on the parameters. Most runs, which had a population of 100 individuals took approximately 15 minutes, but when we changed the iterations (see code of level solver 3) from 500 to 5000, the whole run took approximately an hour. This holds for the following processor: *Intel(R) Core(TM) i7-4710HQ CPU, 2.5GHz 2501 Mhz, 4 cores.*

It is also important to note, that since this thesis aims to show whether the evolutionary algorithms are yielding good results and not what to optimize their time, we did not implement any parallelization. The evolutionary algorithms can benefit greatly there since most of the time spent is within the scoring function, which could be parallel for the whole generation.

Now we can discuss the parameters changed between runs of the algorithm. We will first describe them in a list if necessary, then we will present a table with the actual value changes.

Mutating function - the mutation function is very complex and it cannot be described as one value parameter, the internal mechanism remained the same the whole time and is described in section 3.2.1. The following of its parameters were changed in between runs:

Mutate probability - the probability that a given level representation will be changed by the mutation function

Enemy versus obstacle mutation probability - when mutating, either obstacles or enemies are mutated, this is the probability of one versus the other, we used many different values, but found out that enemies have a much bigger impact on the level score and so their probability should be low, we settled on 1:6 in favor of obstacles

Enemy mutate probability - when mutating enemies, what is the probability for every single enemy to be mutated

Add enemy probability curve - a curve that gives probability of adding enemy given the number of enemies in level, it was changed, but we did not find a way to display the values in the table

Remove enemy probability curve - this is the same as the previous value, just for removing enemies

Enemy mutate max position change - when mutating an enemy, what is the maximal change on position on a given axis, we shall call it *Enemy pos change*

Selection mechanism - we only used roulette wheel selection, where the better score an individual has the bigger chance it has to be selected, however in this type of selection we can set some baseline, so that low scoring individuals have a chance to advance, we will call it *roulette base*

Previous population bias - the previous generation has just slightly worse chances to get to the next generations than the new offsprings

Ending predicate - as ending predicate, we used number of generations, which changed between runs, we are going to call this number *generations*

Plotter - When some of the runs were already concluded, we discovered, that it could be beneficial to store all of the scores of all generations. Those can later be plotted into graphs. Those graphs will be presented later in this chapter.

For space and clarity reasons it is not possible to include all runs in the table. Since the actual research was conducted on a game including 5 procedurally generated levels, the table 4.1 only focuses on parameters from those 5 runs, which are represented as columns. The order of the columns is the order in which they were generated. The last row shows the order of levels in the game.

Created order	1	2	3	4	5
Mutate probability	0.2	0.15	0.35	0.35	0.35
Enemy pos change	20	20	7	7	7
Roulette base	0.001	0.001	0.001	0.001	0.001
Previous pop bias	0.9	0.9	0.99	0.99	0.99
Generations	100	100	100	100	130
In game order	3	2	1	5	4

Table 4.1 Changed parameters of evolutionary algorithm

Even though columns 3 and 4 have the same parameter values, the results were radically different, which is normal within the realm of genetic algorithms. Now let us dive into which parameter changes had which reasoning and how they potentially influenced the results. For visualization, we are going to use graphs generated by *matplotlib.pyplot* python library. First such graph is to be seen in figure 4.1.

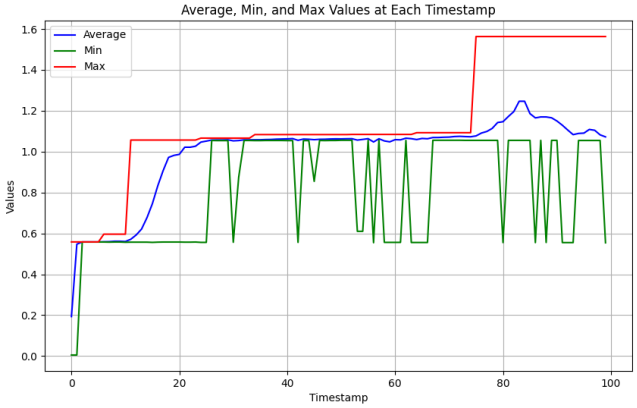

Figure 4.1 The scores of all generations in the third level

In the graph, it looks like there is not enough variety for individuals, since the maximum and minimum are very close and so most individuals are the same, for that reason we lowered the mutation probability so that fewer individuals have mutations and so they have a higher chance of remaining in the population. The result can be seen in figure 4.2.

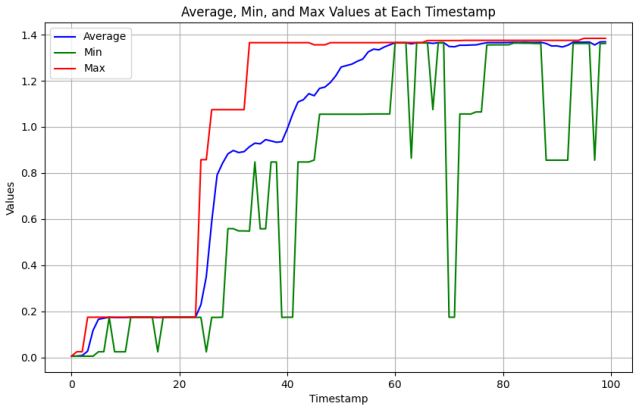

Figure 4.2 The scores of all generations in the second level

We can see that there is a bigger section without a bottleneck, but since there is a serious bottleneck at the very start, most of the levels will not have much variety and the last third of the generations were very stagnant. At this point, we figured, that if we increase the previous population bias, the old populations will have a high probability of retaining in the population so then we can increase the mutation probability without the risk of losing much level diversity. We can see the results of that change in figure 4.3.

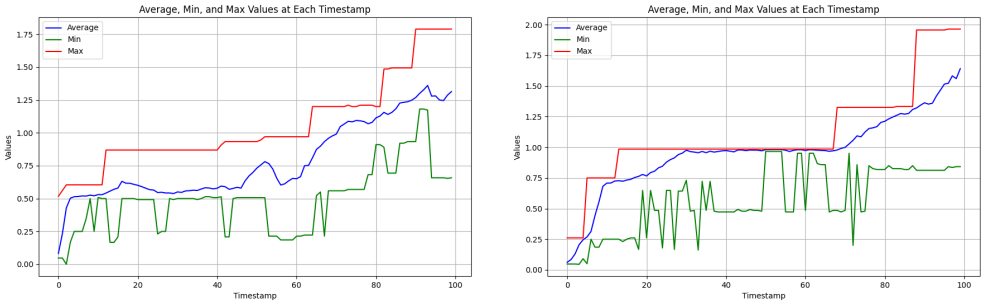

Figure 4.3 The scores of all generations in the first and fifth levels

As we can see the diversity within the generations got much better. And even though the parameters are the same, the fifth level still suffers from some serious bottle-necking. However, when looking at the graph, we can see that the average score increases at the graph end in both examples. From this, we concluded, that increasing the generation count can be beneficial. We can see what that resulted in in figure 4.4.

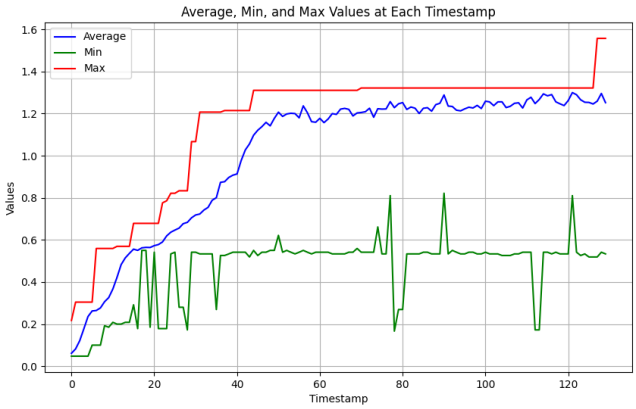

Figure 4.4 The scores of all generations in the fourth level

Even though the bottle-necking problem did improve, we can see that the average is close to the maximum and so it is very probable, that most of the individuals have very similar scores which signifies them being nearly identical, even more so after the slight bottleneck we can see around generation 15.

We did not manage to arrive at a result, which would look like the parameters are set ideally. However, we also saw the levels created and concluded, that they are interesting enough to conduct the research on them.

This creates a very good opportunity for further research since there are many possible parameters to change and the results may improve significantly upon doing so.

4.2 Survey

As the hypothesis of the study is that stealth strategy game levels generated by evolutionary algorithms can result in a similar experience of players as manually created levels, it is necessary to measure the player experience. For this purpose, a number of respondents were contacted and they completed a survey directly after playing the game.

The demo contains 4 tutorial levels, 5 procedurally generated and 5 manually created levels in this order. Between each of those sections, the game asks the player to fill out one part of the research survey. The last part is then filled out when the player finishes the game.

To measure the player experience, we decided to use the concept of Flow [4]. The flow of an activity signifies how much the person at hand is immersed in the action they are taking. Since the definition allows the measurement of any action, flow is a very universal tool to measure immersion. It is also commonly used in the context of video games [34]. We can see the visualization of flow within video games in figure 4.5.

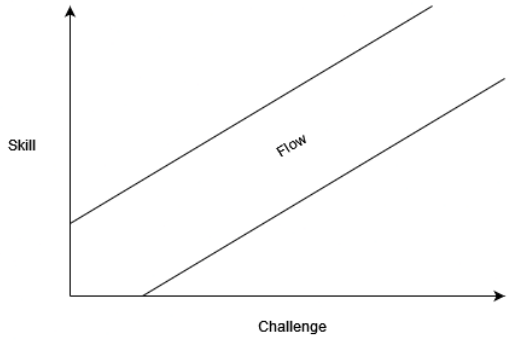

Figure 4.5 The flow state within video games

The questionnaire used was the standardized Flow Short Scale(FSS) from 2003 [35].

To compare the experiences of the players when playing the procedurally generated levels and the manually created ones, it is necessary to discuss the level properties and how they relate to the flow measurements first.

As we have already discussed, the levels within the game are short. When played optimally, they take just under 12 seconds on average to complete (excluding the tutorial levels).

It is however necessary to acknowledge, that the participants will not know the levels in advance so it is highly unlikely that they will find an optimal time-saving play. It is also noteworthy, that when the player gets caught by an enemy, the level is restarted. For all those reasons, players are likely to spend significantly more time on each level than the optimal one. Also, the vocal report from several participants is, that they played for more than 20 minutes in total.

The time spent in levels is potentially relevant for the study. Even though neither the creators [35] nor the usage manual [36] mention any requirement on time spent doing the activity before the flow measurement, we need to assume that some minimal time requirement exists. It is for this reason, that we decided to let the participants play all of the levels in a given group and only then measure the flow.

The order, in which the level groups are, is potentially relevant as well. The flow depends on the player's skill as we can see in figure 4.5. So if the participants learn the game within the first level group, the second level group naturally needs harder levels to stay in the flow state which would decrease the flow comparison value.

However, before both of the groups, there is a tutorial part, which is there in large part to unify the skills for both groups. Also as discussed earlier, the time spent in each level is very short so the skill gained for the player within each group should be negligible.

Another point of discussion can be the validity of the manually created levels. If those levels were created badly, the comparison between them and the procedurally generated levels would hold no value. Those levels were created by someone with thousands of hours of playtime in the stealth strategy games. They were created methodically, we aimed to use mechanisms and design patterns from existing stealth strategy games. Those principles were for example:

- incapacitating enemies barely outside of the view of other enemies,

- enemies inside of viewcones of other enemies,
- enemies which give very short time window for action within their usual direction of view,
- small corners, where playable characters are not seen,
- and more.

It is up to debate whether this creative process did yield interesting enough levels. It should also be addressed, that it is possible that the methodically created levels are done well, but their difficulty is not right for the participants group.

With these points in mind, we can critically evaluate the whole experiment structure which is to be seen in figure 4.6.

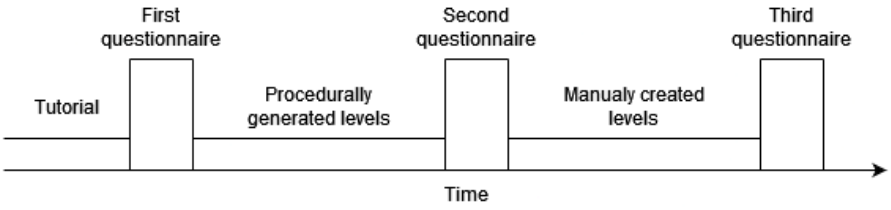

Figure 4.6 The structure of the experiment

It is also important to address the participants group, the conditions the participants were in, the time window, and potential flaws with these points.

In the participant group, there were both men and women with ages ranging between 20 and 30 years. Since no special requirement was set on the group, there were people familiar with the stealth strategy genre as well as ones who did not know it existed. Approximately one-fifth of the participants knew the genre well.

From this point of view, it is important to address that the flow of people with less interest in the genre can be much lower than the flow of prior interested players. For that reason, we can expect the flow values to be relatively low. However since the experiment includes flow comparison, the relative difference of flows is still relevant.

The participants completed the survey online from a location of their choice. It may have skewed the results since some different locations give different amounts of external impulses which may disturb the flow. However, we assume that the amount of disturbance was the same during both groups, so the flow comparison should still hold.

The participants were all asked to use their personal computers for the study. So that the conditions are as similar as possible for all of them.

Since the research was conducted in Czechia, most available participants were Czech-speaking, so it was decided that the survey would be written in Czech as well. It does not have any impact on the results, since they are transferable between languages.

It however may be possible, that concentration on a non-native language is breaking the immersion. This idea was not researched and so has no backing, but there are no downsides to having the survey in Czech since all of the participants

are Czech as well. We can see the survey questions in English and their translation to Czech in figure 4.7.

	anglický originál	česká adaptace
1	I feel just the right amount of challenge.	Hra je tak akorát těžká.
2	My thoughts/activities run fluidly and smoothly.	Moje myšlenky běží plynule a hladce.
3	I don't notice time passing.	Během hraní nevnímám čas.
4	I have no difficulty concentrating.	Nemám problém se koncentrovat.
5	My mind is completely clear.	Mám zcela jasnou hlavu.
6	I am totally absorbed in what I am doing.	Jsem do hry naprosto ponořený.
7	The right thoughts/movements occur of their own accord.	Správné myšlenky přichází jakoby samy od sebe.
8	I know what I have to do each step of the way.	Je mi jasné, co mám dělat.
9	I feel that I have everything under control.	Mám pocit, že mám průběh hry pod kontrolou.
10	I am completely lost in thought.	Jsem zahloubaný a nevnímám své okolí.

Figure 4.7 The English to Czech translation of survey questions

The FSS was also used in other Czech surveys, such as [37] or [38]. These theses also include the Czech translation, however, one of the questions is very ambiguous within them and also they are not targeted at video games. For that reason, the questions were translated to retain their meaning as much as possible while targeting them at games.

There were twenty-two participants, yielding sixty-six measurements in total. We have to acknowledge, that such a number is not sufficient to draw strong statistical conclusions, as the sample-to-item ratio rule suggests that for each question, there should be at least 5 participants. The conclusions however still point in a direction, which could be followed in further research.

The survey was filled out 3 times by each participant, with a random order of questions each time. Since the first survey does not try to target the game, it omitted 5 of the questions. As both the creators [35] and the usage manual [36] suggest, the participants answered each question on a 7-point Likert-scale from ”holds” to ”does not hold”.

5 Results

In this chapter, we will discuss the numerical gathered data and what possible causes there are for their values.

In the flow short scale [35] survey, there are typically 4 factors measured, which are Flow, Fluency, Absorption, and Worry. However, the Worry factor was omitted within this thesis since the activity at hand does not pose any immediate danger to the participants and so the low worry results may skew the overall flow measurement.

To be able to interpret the data at hand, we need to know what is the explanation of the above-mentioned terms and how they are calculated. Firstly, since the measurements were on a 7-point scale, all of the data are from 1 to 7, where 7 corresponds to the best flow (or fluency and absorption respectively) and 1 to the lowest.

For the explanation, let us start with the absorption. It can be interpreted as an indicator of how much the participant is ignoring the outer impulses. With high absorption, the measured person is only focused on one single task and no other ones. The questions which are included in this factor are questions 1, 3, 6, 10.

The fluency describes, how much the person has to concentrate to overcome barriers that obstruct the gameplay. Those would be for example bugs, poor or not enough explanation as well as their mind state. The questions which are included in this factor are questions 2, 4, 5, 7, 8, 9.

The flow factor is somehow deceptive by its name. Even with its name, a high flow factor does not equate to high flow in general. And that is even though the flow factor is calculated based on all of the questions.

To conclude the overall flow, we have to look if the flow factor value is high and also whether both absorption and fluency are not much lower.

In table 5.1 we can see the mean and the standard deviation for all three conducted flow surveys. The values are quite low, however, that was suspected as described above (4.2). We also need to acknowledge, that it may be caused by the potentially low quality of both manually created and procedurally generated levels as also mentioned above (4.2).

	Flow	Fluency	Absorption
Tutorial	5.46 (±1.6)	5.85 (±1.3)	3.86 (±2.0)
Procedural	5.62 (±1.7)	6.10 (±1.3)	5.16 (±2.0)
Manual	5.67 (±1.6)	5.90 (±1.5)	5.48 (±1.7)

Table 5.1 Gathered flow means and standard deviations

On one hand, we target the flow comparison, but it is also beneficial to discuss the flow raw values. As described above, all three factors play a role in it. We see that the tutorial flow factor is holding close to the flow factors of the other parts.

however, the Absorption factor is 1.6 lower than the flow factor. That signifies a relatively low flow level.

We also measured the internal consistency of the flow data within the procedural and manual created levels. The tutorial is omitted since the target of the thesis. To measure the consistency, we used the Cronbach's alpha [39]. As we can see from the values in table 5.2, they all exceed the 0.7 mark, which points to acceptable data, and only in one case dropped below 0.8, which marks good data consistency.

	Flow	Fluency	Absorption
Procedural	0.88	0.85	0.83
Manual	0.87	0.79	0.89

Table 5.2 Cronbach's alpha of the flow data

In table 5.3 we can see how the flow values changed from the ones conducted after the tutorial to the ones conducted after playing the procedurally generated levels. It is apparent, that the Absorption raised very highly. We suspect the cause is the tutorial not posing much challenge and breaking the immersion with the instructions.

We can also see, that the fluency did not change much. That is also logical since the players play the same game and the amount of barriers encountered is the same. It also signifies no major bugs compared to the tutorial.

We also have to acknowledge, that Fluency rose by 1.3, shrinking the gap between it and the Flow factor to to only 0.46.

Flow Δ	Fluency Δ	Absorption Δ
0.16	0.25	1.3

Table 5.3 The flow differences from tutorial to procedural levels

In table 5.4 we can see how the flow values changed from the ones conducted after the procedurally generated levels to the ones conducted after playing the manually created levels.

Flow Δ	Fluency Δ	Absorption Δ
0.05	-0.20	0.32

Table 5.4 The flow differences from procedural to manual levels

All these values are together determining the conclusion of the experiment. We can again see almost no change in fluency, which is probably for the same reason as described above. We can also see, that the Absorption factor rose again

so it is now only 0.19 under the Flow factor. That signifies much better flow with very similar values of the Flow factor.

Here we have to take into account, that the interval shrinking does not linearly add to the flow. In other words, shrinking from 5 to 4 is very different from shrinking from 1.5 to 0.5. There also exists a value, where more shrinking is no longer important since the previous data signify good flow already.

We can also see the Flow factor values comparison in figure 5.1.

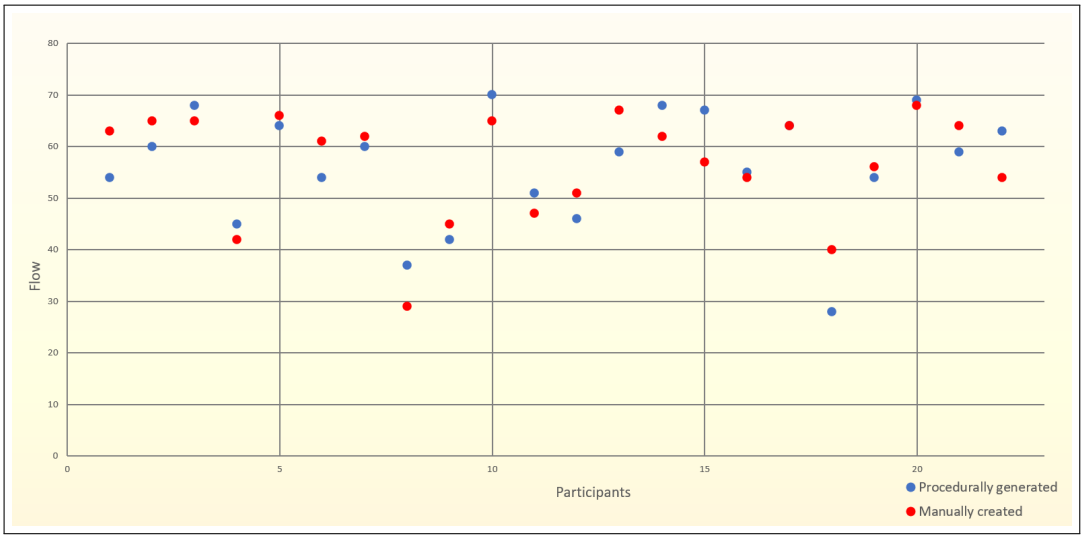

Figure 5.1 Flow factor of manual and procedural levels

Finally, we need to acknowledge, that even though we made the playing sessions relatively long by only having the participants fill the survey three times, it is still possible that the slow Flow factor growth is solely caused by them getting gradually immersed because the sessions were not long enough. However, as mentioned in section 4.2, some participants verbally described the experiment as longer than 20 minutes, which should be sufficient.

Conclusion

We asked the question, whether playing stealth strategy levels generated by evolutionary algorithms can result in a similar player experience as playing levels created manually.

Throughout the thesis, we went through the implementation of the game library including the game solver, which presents a novel algorithm for finding a path within a level with moving enemies. We also saw our implementation of the game demo including level generating. It uses the implementation of evolutionary algorithms, which we went through as well.

Then we looked at how the experiment was conducted. We also explained the techniques used and why they were used in the experiment. Now that we have seen the final data, that the flow[4] survey yielded, we can try to draw some conclusions.

We calculated the paired samples T-test [40] on all three factors where pairing was the survey before and after playing the manually created levels. With a significance level of 0.05, the T-test does not disprove our theory that there is no significant flow difference between playing levels generated by evolutionary algorithms and playing levels created manually.

In the final data, we can see that the Flow factor is high for all three measurements compared to 6 studies presented in the flow manual [36]. On the other hand, these studies also used the worry factor, which we omitted, so the comparison cannot be used to draw any conclusions.

Finally, we can say, that the total flow level is slightly higher for the manually created levels than the procedurally generated ones.

With these points in mind, we conclude, that our data suggest that a similar level of flow can be achieved by playing levels generated using evolutionary algorithms as by playing levels created manually. Moreover, we deduce from the flow comparison, that it is probably possible to get a similar player experience by playing stealth strategy levels generated by evolutionary algorithms as by playing levels created manually.

We however have to acknowledge, that even while we are dealing with a yes-no question, the conclusion is not as binary. The numerical data point in some direction, but further exploration is needed to prove or potentially disprove what they suggest.

We consider the project as an overall success. We managed to prove, that it is possible to conduct research in the area and made a first step in that direction. We also created a full game core, which could be used in future work to create a full-scale playable game. Such a game could also include procedural generation of levels.

That is however not the only direction potential future work can take, another possibility would be to conduct further research investigating for example participant group consisting solely of stealth strategy players, longer time spent playing by the participants, further investigation of the evolutionary algorithm parameter space or researching on a game supporting multiple playable characters.

Bibliography

1. Blow, J. Game development: Harder than you think. *Queue*. 2004, vol. 1, no. 10, pp. 28–37.
2. Köhler, B.; Haladjian, J.; Simeonova, B.; Ismailocic, D. Feedback in low vs. high fidelity visuals for game prototypes. *Second International Workshop on Games and Software Engineering: Realizing User Engagement with Game Engineering Techniques*. 2012, vol. 12, no. 1, pp. 42–47.
3. Smith, R. Level Building for Stealth Gameplay. In: *GDC 06*. 2006.
4. Csikszentmihalyi, M. *Flow*. Portál, 2015. isbn 978-80-262-0918-8.
5. Smith, G. An Analog History of Procedural Content Generation. In: *International Conference on Foundations of Digital Games*. 2015.
6. Roden, T.; Pareberry, I. From Artistry to Automation: A Structured Methodology for Procedural Content Creation. *Entertainment Computing*. 2004.
7. Short, T. X.; Adams, T. *Procedural Generation in game design*. Boca Raton: Taylor & Francis, 2017. Paperback edition. isbn 978-1-4987-9919-5.
8. Linden, R; Lopes, R.; Bidarra, R. Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*. 2014, vol. 6, no. 1, pp. 78–89.
9. Connor, A.M.; Greig, T.J.; Kruse, J. Evolutionary generation of game levels. *EAI Endorsed Transactions on Creative Technologies*. 2018, vol. 5, no. 15.
10. Khatib, Y. *Examining the Essentials of Stealth Game Design*. 2013.
11. Xu, Q; Tremblay, J.; Verbrugge, C. Procedural Guard Placement for Stealth Games. In: *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2014.
12. Na, Hyeon-Suk; Jeong, Sanghyeok; Jeong, Juhong. Improving A Stealth Game Level Design Tool. *Journal of Korea Game Society*. 2015, vol. 15, no. 4, pp. 29–38.
13. Al Enezi, W.; Verbrugge, C. Dynamic Guard Patrol in Stealth Games. In: *Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2020.
14. Miloslavov, I. Desired Path-Dependent Enemy Placement in Stealth Video Games. *comp 400, McGill*. 2018.
15. Sedlák, F. *Procedural generation of levels for a realtime stealth game*. 2024. MA thesis. Charles University.
16. Tremblay, J.; Torres, P; Rikovitch, N; Verbrugge, C. An Exploration Tool for Predicting Stealthy Behaviour. *AIIDE Workshop Technical Report WS-13-20 (Artificial Intelligence in the Game Design Process)*. 2013, vol. 9, no. 3.
17. Borodovski, A. *Pathfinding in Dynamically changing stealth games with distractions*. 2016. MA thesis. McGill University.

18. *Commandos series*. Pyro studios, 1998.
19. *Commandos: Behind Enemy Lines*. Pyro studios, 1998.
20. Suarez, G. Gonzo! What's your game? *PC Zone*. 2001, no. 104, pp. 44–45.
21. *Commandos: Beyond the Call of Duty*. Pyro studios, 1999.
22. *Desperados: Wanted Dead or Alive*. Spellbound, 2001.
23. *Robin Hood: the Legend of Sherwood*. Spellbound, 2002.
24. *Helldorado*. Spellbound, 2007.
25. *Satelitte Reign*. 5 Lives studios, 2015.
26. *Volume*. Mike Bithell Games, 2016.
27. *Shadow Tactics: the Blades of the Shogun*. Mimimi Games, 2016.
28. *Desperados III*. Mimimi Games, 2020.
29. *Commandos 3: HD Remaster*. Pyro studios, 2022.
30. *Frigato: Shadows of the Caribbean*. Mercat Games, 2022.
31. *Shadow Gambit: the Cursed Crew*. Mimimi Games, 2023.
32. *Sumerian Six*. Artificer, [n.d.].
33. Mitchell, M. *Introduction to genetic algorithms*. London: MIT Press, 1996. Paperback edition. isbn 0-262-13316-4.
34. Huang, R. F. The Impact of Flow State and Immersion in Video Games. *Communications in Humanities Research*. 2023, vol. 5, no. 1, pp. 43–48.
35. Rheinberg, F.; Vollmeyer, R.; Engeser, S. Die Erfassung des Flow-Erlebens. *Diagnostik von Motivation und Selbstkonzept*. 2003, pp. 261–279.
36. Rheinberg, F.; Vollmeyer, R; Engster, S.; Sreeramoju, R. R. *Flow Short Scale manual*. 2023.
37. Schönová, E. *Výkonová motivace, prožitek typu flow a tanec*. 2010. MA thesis. Jihočeská Univerzita v Českých Budějovicích.
38. Kverka, V. *Autotelické prožívání při překovánání lanových překážek*. 2011. MA thesis. Univerzita Karlova.
39. Cronbach, L.J. Coefficient alpha and the internal structure of tests. *Psychometrika*. 1951, vol. 16, pp. 297–334.
40. Gosset, W.S. *The application of the "Law of Error" to the work of the brewery*. 1905. Tech. rep., 1. Brewhouse.

List of Figures

2.1 Evolutionary algorithm main loop 10
2.2 Using a skill on an enemy in Desperados III 11

3.1 The data classes of the game rules 15
3.2 The data classes of a level representation 15
3.3 The data classes of a level state 16
3.4 The properties and methods of game actions 16
3.5 A navigation graph on an example level 18
3.6 The inheritance of Graph classes and their data 19
3.7 *StateNode* class with its internal data 22
3.8 Discretization of an enemy viewcone 25
3.9 Combining viewcones with illegal path visualization 25
3.10 Desired result of viewcone combination 26
3.11 Viewcones crossing each other 27
3.12 Created targets around the player with use range r 28
3.13 Class dependencies within level generating part of Unity game . . 31
3.14 Example gameplay from the game demo 34
3.15 Class dependencies within level generating part of Unity game . . 35
3.16 Behaviour of player clicking on obstacles 36

4.1 The scores of all generations in the third level 40
4.2 The scores of all generations in the second level 41
4.3 The scores of all generations in the first and fifth levels 41
4.4 The scores of all generations in the fourth level 42
4.5 The flow state within video games 43
4.6 The structure of the experiment 44
4.7 The English to Czech translation of survey questions 45

5.1 Flow factor of manual and procedural levels 48

List of Tables

4.1 Changed parameters of evolutionary algorithm 40

5.1 Gathered flow means and standard deviations 46
5.2 Cronbach's alpha of the flow data 47
5.3 The flow differences from tutorial to procedural levels 47
5.4 The flow differences from procedural to manual levels 47