Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Technologies (FEM)

Bachelor Thesis

Use of AI in game development

Illia Holovko

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Illia Holovko

Informatics

Thesis title

Use of AI in game development

Objectives of thesis

The goal of this thesis is to assess the possibilities of using different AI approaches for purposes of game development. The main objective is to propose and develop an experimental solution for a specific scenario and evaluate the suitability of AI integration.

Partial objectives are:
- Conduct a theoretical review of literature and online sources with a focus on AI implementation in game development
- Assess and compare existing approaches using AI in a selected model scenario
- Propose a solution using a suitable AI approach and implement it in a game prototype

Methodology

Bachelor's thesis will consist of two parts. The theoretical part of the work will be based on study and analysis of available literature and online sources in the areas of AI and game development. In the practical part, a scenario involving a specific usage of AI will be defined. A solution will be proposed and experimentally implemented using a game prototype application. The suitability of the chosen approach will be evaluated and discussed. The conclusions of the thesis will be based on the results of both the theoretical and practical parts.

The proposed extent of the thesis

40-50

Keywords

game development, Unity, AI, C#, software tools, prototyping, pathfinding, automated environment generation

---

Recommended information sources

Bennett, C., & Sagmiller, D. V. (2014). Unity AI Programming Essentials. Birmingham, UK: ISBN 978-1-78355-355-6.

Hendrikx, M., Meijer, S., Van Der Velden, J., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. ResearchGate. [Online]. Available at: https://www.researchgate.net/publication/262327212_Procedural_Content_Generation_for_Games_A_Surve [Accessed: 28.05.2023].

Shaker, N., Togelius, J., & Nelson, M. J. (2016). Procedural Content Generation in Games. Switzerland: ISBN 978-3-319-42716-4.

Spronck, P., André, E., Cook, M., & Preuß, M. (2017). "Artificial and Computational Intelligence in Games: AI-Driven Game Design." Dagstuhl. [Online]. Available at: https://drops.dagstuhl.de/opus/volltexte/2018/8672/pdf/dagrep_v007_i011_p086_17471.pdf [Accessed: 29.05.2023].

Yannakakis, G. N., & Togelius, J. (2018). Artificial Intelligence and Games. Cham, Switzerland: Springer International Publishing AG. ISBN 978-3-319-63519-4.

Expected date of thesis defence

2023/24 SS – PEF

The Bachelor Thesis Supervisor

Ing. Jan Pavlík, Ph.D.

Supervising department

Department of Information Technologies

Electronic approval: 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Head of department

Electronic approval: 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 13. 03. 2024

Declaration

I declare that I have worked on my bachelor thesis titled “Use of AI in game development” by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 14.03.2024 _____

Acknowledgment

I would like to thank Jan Pavlik, for his support and given advice throughout my work on this thesis.

Use of AI in game development

Abstract

The main objective of this thesis is to explore different AI approaches that are being used in Game Development. In the theoretical part of the thesis, three main AI implementations will be analyzed, specifically: Pathfinding, Procedural Content Generation, and Player Behavior Analysis.

In the practical part of the thesis, an environment for development will be selected and a scenario will be proposed with an experimental solution that will involve some of the AI technologies discovered earlier. The feasibility of the application of AI will be discussed with the ensuing pros and cons.

Keywords: game development, Unity, AI, C#, software tools, prototyping, pathfinding, automated environment generation

Použití umělé inteligence ve vývoji her

Abstrakt

Hlavním cílem této práce je prozkoumat různé přístupy umělé inteligence, které se používají ve vývoji her. V teoretické části práce budou analyzovány tři hlavní implementace umělé inteligence, konkrétně: hledání cesty, procedurální generace obsahu a analýza chování hráče.

V praktické části práce bude vybráno prostředí pro vývoj a navržen scénář pro experimentální řešení, které bude zahrnovat některé ze zkoumaných technologií umělé inteligence. Bude diskutována proveditelnost aplikace umělé inteligence a stanoveny její výhody a nevýhody.

Klíčová slova: Vývoj Her, Unity, AI (Umělá Inteligence), C#, Softwarové Nástroje, Prototypování, Hledání Cesty, Automatická Generace Prostředí

Table of content

1 Introduction

In today's world full of rapid technology advancements, we, human beings, can access many facilities that sounded like science fiction some years ago. The Internet, smart devices, video games, etc. With the enlarging spreading of such facilities and the Earth's population, more and more people are involved in video games.

According to statistics almost half of the Earth's population is playing video games. It means that the realm of video gaming has shifted from a niche hobby for certain ages (mostly for teenagers) into today's global phenomenon. The economic and cultural aspects of this sector of IT can now be reckoned with.

Moreover, another rapidly growing aspect is Artificial Intelligence. The world has already faced the helpfulness of implementing AI, for example: AI chats for copywriting, AI algorithms that help with improving PC's performance in gaming, and AI algorithms that are being used for generating content (pictures, textures, descriptions, etc.).

Since the first video game, the development of this product has improved hugely, nowadays programmers are using special Gaming Engines, such as Unity, Unreal Engine, CryEngine, Crytek, etc. Those environments allow to spread of development between different people and have a lower entry threshold for developers as some engines can even provide visual scripting(f.e. Blueprints in Unreal Engine, Unity Visual Scripting, etc.). This is so much different from the first video games that were written manually and intricately.

To sum up, everything that was written above can lead to the obvious conclusion - we can count the video games segment as a viable business, the barrier to becoming a developer has increasingly lowered through time, so implementing AI during video game development is very important as it saves money and time for companies and indie developers, some examples of implementation were already mentioned above.

My thesis will focus on discovering ways of implementing AI technologies during video game development. I will define some problems and provide solutions, which will involve AI, and an analysis of the solution will be provided.

2 Objectives and Methodology

2.1 Objectives

The goal of this thesis is to assess the possibilities of using different AI approaches for purposes of game development. The main objective is to propose and develop an experimental solution for a specific scenario and evaluate the suitability of AI integration.

Partial objectives are:
- Conduct a theoretical review of literature and online sources with a focus on AI implementation in game development
- Assess and compare existing approaches using AI in a selected model scenario
- Propose a solution using a suitable AI approach and implement it in a game prototype

2.2 Methodology

Bachelor's thesis will consist of two parts. The theoretical part of the work will be based on study and analysis of available literature and online sources in the areas of AI and game development. In the practical part, a scenario involving a specific usage of AI will be defined. A solution will be proposed and experimentally implemented using a game prototype application. The suitability of the chosen approach will be evaluated and discussed. The conclusions of the thesis will be based on the results of both the theoretical and practical parts.

3 Literature Review

3.1 Game Development

Every video game is complex software, its production requires at least some knowledge in game designing, programming or scripting, sound designing, UI/UX interface creation, etc. Nowadays, game development has grown into an essential and serious business, there are two major "players" in this kind of business: developer companies and publisher companies, however, not every video game has enough resources for its development, so not every video game will have publisher or a whole team for development. Usually, there are three main types of video games based on its budget, development team, etc. and those types are AAA projects, AA projects, and Indie projects:

1. AAA projects: these are some kinds of videogames that has high budgets (approximately 50 million dollars or more), developer has different dedicated departments, such as: the marketing department, the programming department, the art department etc. AAA-Projects are usually being made on company's own game Engine as their budgets allow them to do so. Such games have a vast number of developers working on them and their marketing includes digital, targeted and physical advertisement. We, as players, can see physical copies of AAA titles in almost every specialized retail shop, also, those games are available on big digital gaming platforms like Steam, Epic Games Store, etc. Sometimes, a company can have its own platform that is dedicated only to its produced or published games, for example, Battle.net, which includes only Activision Blizzard games, or Origin which includes only Electronic Arts games. Examples of projects are the GTA series (Rockstar Games), the Forza Horizon series (Playground Games), and The Witcher 3: Wild Hunt (CD Projekt Red, 2015).
2. AA projects: usually, these kinds of projects have a medium budget (approximately between 1 million and 50 million dollars), developer company can have some departments, but, unlike AAA titles, AA titles are not very often distributed through physical copies, but sometimes publisher can decide to make a limited run of physical copies of the game, especially if it's some special or collector's edition. Their marketing usually includes digital and targeted advertisement. Also, AA projects are sometimes made by subsidiaries of some big companies, for example, Child of Light (Ubisoft Montreal, 2014) which was made by a subsidiary company of Ubisoft.

These titles are usually built on open-distributed game engines like Unity or Unreal Engine.

3. Indie projects: usually, these projects are made by a very small group of people or even by only one person, they have low budgets (approximately up to 1 million dollars). Almost every indie project is distributed through digital stores that were mentioned above. While AAA and AA titles are considered highly manageable products that don't quite often manipulate with some niche mechanics or settings, indie developers usually experiment with new and not very well-known gameplay features, game settings, etc. Example of indie projects are: Stardew Valley (ConcernedApe, 2016), Ori and the Blind Forest (Moon Studies, 2015) (1). Despite AAA and AA projects, indie projects usually don't have publisher. Of course there are a lot of them that help indie developers with deploying their project into digital stores and financing it, but usually, those companies require some royalty from game sales. If developers don't want to share their income with the publisher, they usually find financing in some special crowdfunding platforms, like Kickstarter or itch.io.

However, its worth mentioning that such distinctions are conditional, in some cases, AA projects can be classified as either AAA or Indie project. Also, as the topic of this thesis is implementation of AI during game development, it's vital to say that AI can be implemented in almost every stage of game development, it can be useful for providing some gameplay features for the programming department, generating content for the art department or even for generating some ideas for the marketing department, this means that AI is suitable for either AAA or Indie projects. I will be mainly focused on some AI implementations that are aimed at developing video games, not for their distribution, more specifically it will be discussed below.

3.2 Game Engines

At the beginning of the video game development era, every game has been written manually by using different programming languages. It really differs from today's development as the first video games were implemented in a way, that no intermediary was involved between the game and hardware, which means that every rendering "engine" could work only for one game only. Now it changed, you don't have to do it manually without any GUI, now you can see what you are actually doing, and how your changes reflect interaction with the game, this all is possible, thanks to game engines.

What is a gaming engine? Basically, this software provides a development environment, usually with GUI, so that it can be useful not only for programmers but also for people who are working on the art style of the game (creating models, level designing, etc.), also, despite those obsolete rendering "engines", nowadays gaming engines(like Unity, Unreal Engine, etc.) can provide multiplatform and can be used for different kind of games, you can create shooter or platforming, or either implement 2D or 3D graphics(most of the modern engines provide both dimensions, however, 2D is usually implemented as 3D graphics but with a static camera so that the game can render only 2 dimensions) (2). There are different types of licenses for this sort of software it can be either free or paid or hybrid versions (you can either use a free version or buy a subscription). Also, game engines can be only for in-company use, which means that technologies of those engines can be accessed only within certain companies and/or their subsidiaries, for example, it can be RAGE, Crytek, Duna, etc. A detailed overview of the 2 most popular game engines, namely: Unity Game Engine, and Unreal Engine, that are being used by both big and indie companies will be provided below.

Moreover, nowadays, despite the rendering aspect, game engines are also capable of providing audio, animation tools, AI implementation, and physics logic. It means that "Game" engines can also be used not only for gaming but for example: storyboards for film production, visualization of any data, etc. This means that this software grew from niche usage for game developers to a worldwide powerful tool for different purposes.

3.2.1 Unity Game Engine

A Unity Game Engine was developed by Unity Technologies in 2005 and originally was intended to be a Mac OS-specified game engine, however, later it rapidly changed and transformed into a multi-platform environment for development. Nowadays, Unity supports the creation of games for different platforms from desktop to mobile ones. The basic features of this engine involve real-time rendering, physics simulation, audio system, scripting support, asset management, and GUI.

However, developers can use different programming languages to create scripts in Unity, the main one is C# (3). Also, sometimes JavaScript can be used for some web interactions if the game is developed for a web browser or if it includes an in-game browser, but, it's not quite widely being used nowadays, the game engine shifted in the way that developers usually use only C# in their products, however, Unity WebGL is currently being used for

some 3D interactions on the website, it can be, for example, demonstration of the car within car configurator on the manufacturer's website.

Moreover, Unity Game Engine provides real-time rendering, which means, that the developer can see what he is doing right now in dynamics (4). Whatever will be changed whether game assets, scripts, or environment the result can be instantly seen on the developer's screen. Also, Unity provides a drag-and-drop feature, which means that there is no need to manually write coordinates to place something, you can place models or apply scripts on some characters, environment aspects, etc. just by dragging that over your game scene. This feature pushes forward development speed, as changes can be seen instantly, so you have space for experimentation and adjusting, without any need to write everything directly with the programming language.

Unity also supports the implementation of AI technologies, for example, Procedural Content Generation and Pathfinding. Different AI implementations will be discussed below. Unity Game Engine's AI-integrated tools can be very helpful for different ranges of developers, as it doesn't involve high-tech knowledge to create some basic AI integration methods. For example, the NavMesh system that provides a user-friendly approach to simplify the creation of complex pathfinding processes (5), NavMesh is a helpful system that helps indie and mid-range developer studios perform pathfinding and walkability tests.

In conclusion, nowadays Unity Game Engine is a very powerful tool and not even for game development as it was written above, WebGL provides the ability to create some 3D interactions on the website. It has evolved from being a niche instrument for one specific platform into the cross-platform powerful game engine with a user-friendly interface, that can be used by different members of a development team.

3.2.2 Unreal Engine

Despite Unity, Unreal Engine was developed by Epic Games in 1998 and was released with the "Unreal" shooter. The main features of this game engine crosshairs with Unity. The main programming language of this game engine is C++, which makes the development entry threshold higher than in Unity. However, despite Unity, Unreal Engine has a built-in blueprint system, which allows developers to create gaming scripts and render rules without deep knowledge of C++, it provides a graphical interface with nodes that can be connected and tuned. Unity has a similar feature, but it was implemented much sooner

than in Unreal Engine, which makes it more complicated to find some documentation about that.

Unreal Engine is more likely to be used for triple-A projects, while Unity can compete in the indie segment, due to its larger community, also, Unreal Engine is more capable of dealing with hyper-realistic graphics (especially after the release of UE 5.2 with PCG, Substrate, Nanite and Lumen (6)).

In contrast to Unity, Unreal Engine can be used for graphical representations as it provides better rendering features, so that, it can be used for creating rendered advertisements or even some movie special effects, unfortunately, Unity can't compete with Unreal Engine in such segment, due to lack of built-in rendering features.

Unreal Engine also provides AI implementation, as was written above, with the release of UE 5.2 developers can now create procedurally generated content, so that, a game scene can be very diverse, thanks to this built-in technology.

In conclusion, Unity implementation can vary, from developing a video game to implementing some 3D representations on the website. However, Unreal Engine also provides web interaction, but due to its higher demands on the hardware – it isn't being used that much in a such segment. However, due to complexity and rendering features, Unreal Engine is more suitable for example: rendering scenes for advertisements, representation of real-life architecture projects, movie storyboards, etc.

3.3 AI Technologies during Game Development

There are a bunch of AI implementations that can be used in game development, some of them can enhance the gaming experience by simulating unique behavior of Non-Playable Characters(NPC) or generating seemingly "random" environmental interactions. On the other hand, AI implementation can be also beneficial for developers themselves, because some content can be generated autonomously, so that, there is no need to spend time and resources on creating assets.

However, some of the AI implementations can save time and resources, but they most probably won't have the same quality as something that has been created by human beings or they can consume a large amount of the hardware power, those drawbacks will be provided below.

3.3.1 Pathfinding

Let's assume that you have to create a character in your video game. One of the solutions for such a task is to create paths so that the character can take instructions on where and how it must go. However, the main problem occurs when there are large locations, not a small room, or if we want to create variety, so that, users won't have a feeling that everything is very predictable and our characters on the scene can walk in different directions that will create a feeling of having "life alike" behavior. To solve this problem, the main solution can be AI, to be more precise - pathfinding.

To begin with, the main drawback of pathfinding, that most researchers are pointing at is the big usage of CPU and Memory power (7). But, it's worth mentioning, that nowadays hardware can handle most of the pathfinding implementations and almost every modern game that includes NPCs uses this AI technology. How does it work? Well, the main thing here is to realize, that pathfinding can be implemented in different ways that vary from the algorithm that has been cored in certain pathfinding methods. On the "high level" of this technology lays a very simple principle: we just have to define how to get the shortest path from point A to point B. Let's see different algorithms that can lay on the "low level" of pathfinding.

3.3.1.1 A* algorithm

This algorithm is one of the most used by many developers (8). It is perfect for games that depend on the proper pathfinding, such as racing, strategy, etc. How does it work? It faces challenges of pathfinding by systematically navigation through nodes in a graph to find the most optimal path from a start node to the end node.

The main two factors that lie on the basis of A* are cost and heuristic estimation. It operates by simultaneously considering both the actual cost of the path and an estimation of the remaining cost to reach the end node. By this action, A* is capable of narrowing down search space while still optimizing trajectory.

A* uses a heuristic function(f(n)) to serve, and it is $f(n) = g(n) + h(n)$. The function g(n) keeps track of all the costs incurred while traveling from the starting point to the destination, thus representing the total effort involved in the journey. On the other hand, h(n) acts as a guide providing an estimate of how far you are from your goal at any given point. In addition, when there are obstacles on your path, f(n) comes into play. It evaluates and selects the most cost-effective route to ensure that you reach your destination with minimal hassle (7).

However, this algorithm can be seen as the ideal solution, but there are some drawbacks. First of all, due to the fact that it's node-based, it can consume a lot of memory resources during computations. Also, this is a heuristic method, so it isn't very accurate and depends on well definition of the remaining cost, it can result in an inability to define the most optimal path.

3.3.1.2 Dijkstra's algorithm

Dijkstra's algorithm is another approach to pathfinding. Its implementation can be seemingly similar to the A* algorithm, but it isn't a heuristic method. So, it requires more resources than A*, but it's more accurate. The function(f(n)) that is being used in Dijkstra's algorithm looks like this: f(n) = g(n). The definition of the g(n) is similar to the one that was in A*, but, as you can see, we don't have heuristic value, this is why Dijsktra's algorithm can't be considered as a heuristic one (9).

So why choose Dijkstra's algorithm instead of A*? To begin with, it's accuracy, it is better than in A*. Also, it is simpler to implement. However there are some disadvantages, and the main one is that Dijkstra's algorithm can't really estimate optimal value in grids as it doesn't consider diagonal shortcuts, also, due to its specific work, it won't reduce its search space, as A* do, and it examines all reachable nodes which can lead to long computational time and big memory usage. But it is a perfect solution for small levels, with few amount of NPCs or any else entity that will require pathfinding.

3.3.1.3 Depth-First Search

Depth-first search is a search algorithm that explores each branch as far as possible before backtracking and trying another branch (10). But despite A* and Dijkstra's algorithms, this algorithm is usually being used for decision-making rather than navigation, however, it is still a viable approach for pathfinding.

The main disadvantage of this algorithm is that it tries every possible path/variant, so that it crossfires with Dijkstra's algorithm. Despite A* and Dijsktra's developers can use this algorithm to define different variations, after which entity in the game can make a decision about which option is better, based on what was defined as a good decision by a programmer. Because it's not that sufficient to find an optimal way and in big games, it can even lead to infinite loops.

But from first sight Dijkstra's algorithm and DFS are very similar and do the same job, the main difference is that in DFS there is no need to make sure that all paths that were checked should be best, this is the point why DFS isn't that popular in Pathfinding.

3.3.1.4 Breadth-First Search

Breadth-First Search is the opposite of Depth-First Search. Rather than diving deep into one option, it sorts them into layers and starts exploring from a single point, moving on to neighboring nodes. This method ensures a thorough and organized exploration of possible paths (10). This makes it more suitable for Pathfinding than DFS because it can compare and choose which explored layer is better.

Actually, this approach is faster in most cases than Dijsktra's algorithm and can be chosen as an alternative, however, A* is still the best option among the other four. Despite the fact, that A* can be not that accurate, it is still able to lower the scope of the search, so that, it is much better for optimization of the game project. But, of course, for some game scenarios where precision is very important, A* won't be a good choice, so it depends on which game scenarios the developer is realizing.

3.3.1.5 Pathfinding implementation in different game genres

Let's discuss how pathfinding implementation in some video game genres is useful. For example, in strategy games - pathfinding is a cornerstone technology. It plays a crucial role in enabling Non-Playable Characters (NPCs) to navigate complex environments, which is essential for creating realistic and challenging gameplay. For example, in real-time strategy (RTS) games like StarCraft (Blizzard Entertainment, 1988) and Age of Empires (Ensemble Studios, 1997), efficient pathfinding algorithms are vitally important. They allow for the nuanced movement and management of multiple units, each navigating through varied terrains and reacting dynamically to enemy structures and movements. This not only enhances the tactical depth of these games but also ensures that NPC movements are logical and contribute to a strategically engaging experience.

Another genre that is a good example of pathfinding implementation is stealth and action games. In those video games, the NPC behaviour of enemies is a very important aspect of creating a proper game challenge and properly immersing players in the atmosphere of the specific scene or game's world itself. Effective pathfinding enables NPCs to perform complex tasks, such as patrolling areas, searching for the player, and navigating complex

environments. This is evident in games like Metal Gear Solid (Konami, 1998) and Splinter Cell (Ubisoft, 2002), where enemy NPCs use sophisticated pathfinding algorithms to create a realistic and immersive stealth experience. The AI's ability to realistically imitate human search patterns and responses significantly enhances the strategic depth and realism of these games.

3.3.2 Procedural Content Generation(PCG)

One of the greatest AI technologies that can be used during video game development is Procedural Content Generation. It aims to create in-game content within algorithms, without any need for manual labor. Potential drawback is similar to pathfinding as this technology requires a lot of hardware power, and, unlike pathfinding, modern hardware can face a real challenge with rendering such content, it depends on the complexity of the algorithms that lay on the "low level" of the PCG and on the amount of the content that has been generated.

Where can PCG be implemented in the video game project? Well, there are a lot of ways of the implementation. First of all, with PCG you can generate landscapes, dungeons, and every other environment. One of the greatest examples is dungeon generation for 2D "rogue-like" genre games, like Pixel Dungeon (Retronic Games and Watabou, 2012) or The Binding of Isaac (Group of indie developers, 2011), however, levels or even worlds can be generated within 3D graphics, great examples are Minecraft (Mojang Studios, 2011) and Diablo (Blizzard Entertainment, 1977), both of them use noises for generating their worlds. What is "noise" in the scope of PCG? Basically, it's a mathematical function or algorithm that generates "pseudo-random" patterns, those patterns are so-called "noises", they can be then used to generate a height map of the landscape, that will provide a unique-looking environment without any manual implementation from the developer's side. Or it can create realistic textures or normal and specular maps for them so that fewer hardware resources will be used. There are several patterns that are being used to create noises, because, otherwise, even creating a terrain on the game scene by just filling its height map with random numbers will result in random spikes. All of those noises are usually represented as arrays filled with pseudo-random numbers (11), however, the approach to this filling varies from pattern to pattern.

3.3.2.1 Perlin Noise

So, let's look at some examples of noise generation patterns. One of the most popular ones is Perlin Noise. I will consider it in terms of the generation of the terrain. The basic principle of this noise creation involves directly producing slopes and deriving height values from them, rather than generating height values and subsequently interpolating slopes. The random numbers generated serve as interpretable random gradients, capturing both the steepness and direction of the slopes (12). This pattern is ideal for creating landscapes, I have already written that filling random values into arrays of height maps results in rather multiple spikes than smooth and "realistic" terrain, but with this gradient approach, we will avoid spikes and create smooth transitions heights of our terrain (13).

The main advantage of this principle is that we can avoid some abrupt changes in the elevation, however, in game development, using just one noise-generated map can lead into not very sufficient results. Usually, developers use different noises so that, we can see diverse results. If we are talking about terrain generation – Perlin Noise will just provide us a basic look of the terrain, so it can be considered as a starting point, after implementation of the gradient height map we can proceed to the diversification of our generated terrain. Developers are typically using Fractals noises or Cellular Automata to add some micro-details to the generated landscapes or some natural surface irregularities.

But it was just a defined implementation of the Perlin Noise, can it be implemented in different aspects of the video game? Well, of course, there are some examples:

1. Texture Generation: Perlin Noise can be used for generating different types of textures, usually it isn't some complex textures, but it can handle materials like wood, marble, and stone.
2. Fluid Simulation: Perlin Noise can also handle basic simulation of fluids which is pretty enough for game development, it will be inaccurate according to the physics, but it will be enough for decorative elements in video games.
3. Clouds generation: Perlin Noise is a good choice for cloud generation, but it won't generate a whole skybox, it is more likely to be used for adding depth to the middle graphical games, but it won't be enough for AAA projects.

3.3.2.2 Fractal noises

What if we want to somehow complex our Perlin Noise data? In this case, Fractal noises can be a useful tool. They can help to add some details to the noise pattern. However, it is still a

self-sufficient pattern, a combination of Perlin and Fractal noises is a popular technique, that is being used by various video games today.

How do Fractals work? Well, the main principle of fractals is that it's a pattern that consists of smaller patterns so that by "zooming in" you will see that they just repeat themselves. This process is achieved by continuously replicating iterations in which specific patterns can be assigned, by proceeding with such iterations a Fractal can be created (14). This is why it is considered a less complex technology than Perlin Noise, but the combination of both can add to the base that was generated based on Perlin Noise data some minor details and won't look repetitive to the user.

Some examples of Fractal Noises implementation:

1. Texture mapping: Fractal Noises find application in the creation of normal and specular maps for the textures. While Perlin Noise provides only a base for mapping, additional integration of Fractal Noises can impart depth and intricacy, for example, developers can easily add porosity to the wood texture by the synergy of those technologies. Also, such implementation will be beneficial for the CPU and Memory resources of the hardware, as noises for texture mapping can be created only during the development stage of the game without any 3D model user will see that texture isn't a solid picture on the wall. But this approach is not being used quite often, as it is much better to add dynamical generation to make game textures feel different from each other, however, it is still a better option than the manual creation of an enormous amount of normal and specular maps for textures.
2. Vegetation Placement: Fractal noises can provide pseudo-random placement for the vegetation on the game scene. However, this implementation is scale-sensitive, it works perfectly fine with smaller vegetation, for example, flowers or grass. It is still can be applied to larger vegetation(for creating forests), but the end result for the user will strongly vary on the playable character's vantage point, because repeatability is more visually noticeable.

In conclusion, there are a lot of ways to create noises(diamond square algorithm, simplex noises, etc.), so that they will be later used for Procedural Content Generation, but those two that were described above are the main ones that are being used by most developers.

3.3.2.3 Cellular Automata

Another technology that is being involved in PCG is Cellular automata. It is a good alternative to noise-based technologies, however, the results of its implementation can be seen as similar to noises, but the approach is quite different.

It comprises an n-dimensional grid along with sets of states and transition rules. Typically, Cellular Automata exist in either one or two dimensions. Each cell within this framework can exist in various states; in its simplest form, cells can either be on or off. The arrangement of cell states at the commencement of an experiment (at time t0) constitutes the initial state of the cellular automaton. Subsequently, the automaton progresses in discrete steps, adhering to the specific rules defined for that automaton. At each time step t, every cell determines its new state by considering not only its own state but also the states of all neighboring cells at the preceding time step (t −1) (12).

This technology is more complex than noises, but, on the other hand, developers can define rules, so that it is more adjustable than Fractal or Perlin noises. Moreover, the base(ground), can be created by noise-based technologies, and with Cellular Automata developer can supplement it with different features, like rivers, caves, etc. While synergy of Fractals and Perlin noises was intended to be a micro-feature addition, synergy with both those noises and Cellular Automata results in a smooth and clean Procedurally Generated level.

On the other hand, there is no need to use Cellular Automata with noises as it's standalone technology too. So that, it is capable of creating content without any noise techniques. Also, Cellular Automata can be used for different simulations, not just for models, for example, resource distribution in strategy games, and pseudo-random population simulation. Some practical examples of Cellular Automata implementation:

1. Cave generation: this is a key feature and what developers use this technology mainly for. Due to its ability to define rules between neighborhood cells, this tool is very powerful for generating dungeons with different entries and exits (15).
2. Vegetation distribution: I have already mentioned that Fractal noises can handle this task, but it has some boundaries. With Cellular Automata it's much easier to create a distinctive spread of different fauna on the game level without any repetitive pattern as it was with Fractals.

Overall, this technology is more complex than noises, but the outcome is more fascinating.

3.3.2.4 PCG implementation in different game genres

I have earlier described that PCG is more likely to be used in “rogue-like” and “open-world” game genres, but, now when we have discussed different noise-based technologies I can be more specific in my examples. The “Open-world” game genre is a great example of where should we implement PCG. By leveraging algorithms like Perlin Noise and Fractal Noises, these games can generate vast, evolving landscapes, offering players a unique experience with each playthrough. In Minecraft (Mojang Studios, 2011), for instance, PCG creates endless terrain variations, ensuring that no two players’ experiences are the same, the developers used different approaches, Noises are among them, alongside unique seeds to generate the world, ore distribution algorithms etc. (16). Similarly, No Man’s Sky (Hello Games, 2016) uses PCG to generate an entire universe, complete with diverse planets and ecosystems, pushing the boundaries of exploration and replayability in the game. This dynamic environment creation is not just about visual diversity, it significantly impacts gameplay, as players encounter unique challenges and discoveries in each session. What about a “rogue-like” game genre? In “rogue-like” games, PCG is fundamental for creating varied and unpredictable gameplay experiences. Games like The Binding of Isaac (Group of Indie Developers, 2011) and Spelunky (Mossmouth, LLC, 2008) rely on PCG to randomly generate levels, enemies, and item placements. This randomness is a defining feature of the roguelike genre, ensuring that each playthrough presents new challenges and environments. The unpredictability fostered by PCG not only enhances replayability but also tests players’ adaptability and strategy-making skills in ever-changing scenarios.

3.3.3 AI in Player Behavior Analysis

In the sphere of game development, AI’s role extends beyond creating immersive environments and simulating NPC behavior. So-called Player Behavior Analysis can be helpful in terms of game testing and enhancement of user interaction with a video game, this branch of AI focuses on player modeling. One important aspect to notice is that player modeling isn’t NPC modeling, player modeling tries to imitate a human player (10), while NPC modeling tries to simulate the behavior of a human being.

Moreover, player modeling plays a vital role in increasing the replayability of the game, usually, there are 3 main roles that could be imitated through player modeling:

1. Companion role: Here, the game AI must align with the expectations of the human player. For instance, if a player prefers a stealth approach, AI-controlled

companions should not engage in open combat that contradicts this strategy. The success of the companion AI lies in its ability to accurately predict and complement the player's desires, enhancing the gaming experience rather than disturbing it.

2. Coaching role: In this role, the AI monitors the player's behavior and, depending on the game's objectives, guides or redirects the player's focus. This role is particularly crucial in serious games, where training and personalized coaching are integral. A well-developed player model assists in achieving the game's goals efficiently and effectively. Unlike the companion role, the coaching AI may challenge the player's approach to foster learning or dramatic engagement.
3. Opponent role: As an opponent, the AI must adapt to match the human player's skill level and respond aptly to their playing style. This balance is challenging; overly weak AI opponents may bore the player, while excessively strong ones can lead to frustration and disengagement. The key is to provide a competitive but fair challenge that maintains player interest and satisfaction (17).

Each of these roles demonstrates the versatility and importance of player-modeling AI for game development. By effectively adopting these roles, AI can significantly enhance the player's experience, catering to different needs and preferences, and ultimately contributing to a more engaging and immersive gaming experience.

Also, player modeling through AI is particularly beneficial in the testing phase of game development. For smaller studios or projects with limited budgets, creating sophisticated AI models that can simulate player behavior is a cost-effective alternative to extensive human testing. This approach allows for preliminary testing of gameplay mechanics and features, helping developers identify and rectify potential issues without the need for a large pool of human testers. It also facilitates the refinement of game features in a controlled and iterative manner, which is especially valuable in the early stages of game development.

In summary, AI in Player Behavior Analysis, particularly through player modeling, is a multifaceted tool in game development. Its applications range from being an efficient testing mechanism for developers with constrained resources to enhancing the overall entertainment value and replayability of games (18). The implementation of AI in Companion, Coaching, and Opponent roles not only revolutionizes player interactions but

also demonstrates the evolving capabilities of AI in creating more responsive and immersive gaming experiences.

4 Practical Part

An overview of the two most popular game engines was provided in the theoretical part of the thesis. I will use Unity Game Engine in my practical part. Firstly, Unity has a lot of built-in AI instruments that will be helpful in my thesis. However, Unreal Engine provide more complex toolkit, it is redundant in such scope. Additionally, most indie developers use Unity for creating video games, so it means that Unity has larger community which will be helpful to implement my solutions to the scenario that is described below. Unity has much wider documentation than Unreal Engine does and it requires only basic knowledge of C# to implement some scripting features.

Moreover, Unreal Engine can provide much better rendering features, which will result in more advantageous graphics. However, it is unnecessary in my practical part as the goal is to implement such technologies and evaluate how their implementation can be helpful during the development of video games. Therefore, a basic prototype with worked mechanics will be quite indicative.

Let's imagine that some indie developers need to implement pathfinding and PCG into their games. The game prototype should be aimed to represent the adventurer-exploration genre so it will need basic mechanisms for providing controlling and unique environment generation, the player should be able to control the Agent by mouse-clicking on the generated terrain, so Pathfinding will be used to provide controlling and PCG will be used to provide unique terrain every playthrough. The goal is to achieve 2 demos and demonstrate how pathfinding and PCG can be implemented through Unity Game Engine, after that both demos will be combined to achieve proper navigation on randomly generated terrain, also the outflows will be discussed below. Throughout the implementation of such AI technologies through Unity engine I will use scripting, the book Unity AI Programming Essentials (19) and Unity official online documentation (20) were helpful throughout the whole development process to understand how Unity C# scripting works.

4.1 Implementation of Pathfinding using Unity

4.1.1 Initial setup

As it was discussed above, Unity has a built-in NavMesh toolkit that uses the A* algorithm at its basic level, so I will use it to solve some practical problems that can occur during game development. Before starting let's imagine a scenario in which the developer

needs to implement some pathfinding, the next content is only imaginary but can be helpful in terms of real development:

- The demo needs to have an Agent that will be controlled by the player
- The Agent should find the closest path to the chosen point by the player.
- The Agent should be able to avoid obstacles
- The Agent should be able to easily adapt to new obstacles, terrain, etc, if the developer does so.

Creating an initial project in Unity

First of all, I need to create the project, from the offered templates in Unity (version 2022.3.17f1 was used in this project), I have chosen 3D Core.

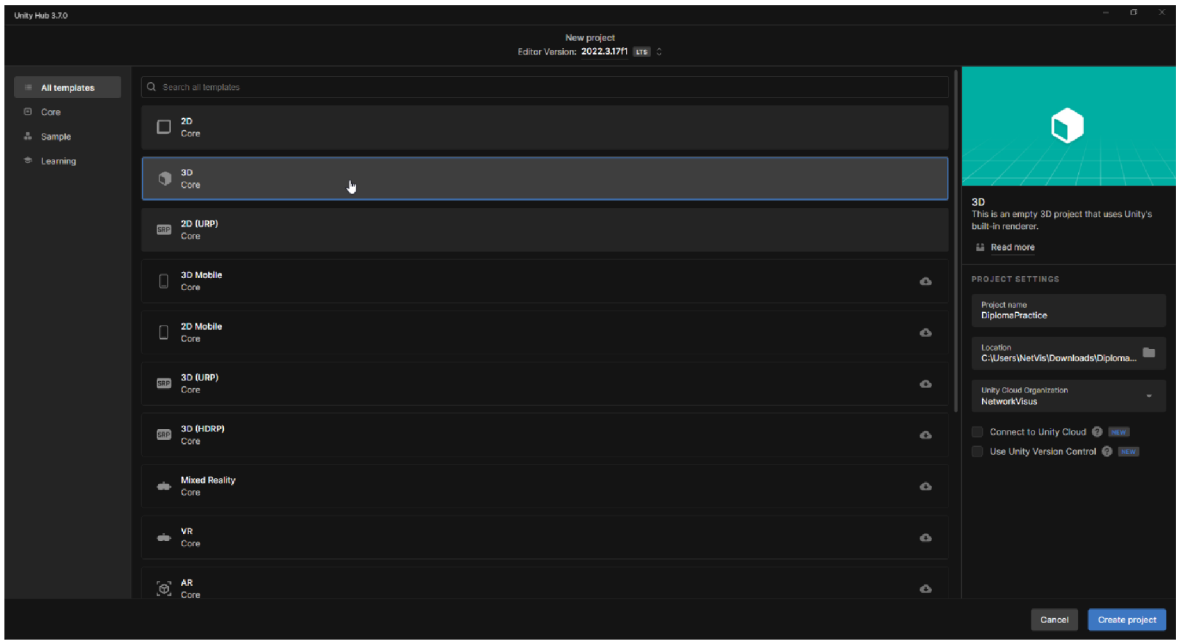

Figure 1 - Unity project creation interface, source: own processing

After the creation of the 3D template in Unity, there is a need to create a basic scene so that I can test the Pathfinding implementation in action. To do so I have used 3D Object -> Cube to create the basic floor of the scene. Also, as it was described above, Unity allows me to do that directly in its interface, also, it allows me to change the sizes of the object without any coding, so I have changed the sizes of my Cube so that it would look more like a plain solid floor for the Agent.

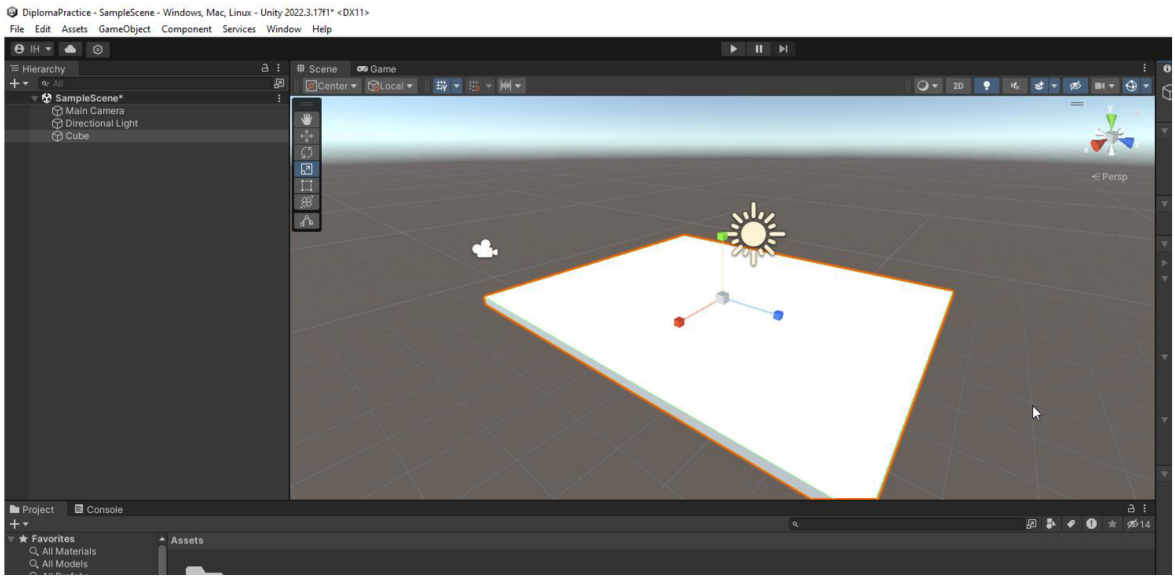

Figure 2 – Plain surface in Unity, source: own processing

To test out that the Agent can actually find a path, I added some other 3D objects so that it will emulate obstacles that can occur on our path. As far as we know, the A* algorithm should be able to navigate even on diagonals instead of Dijkstra's Algorithm, at this point, it doesn't matter how those obstacles will be placed, so I placed them at different points on the floor and in different angles. By doing this, it will be noticeable that Nav Mesh helps the Agent to find the nearest possible path without any limitations according to the closest path being on a diagonal axis instead of a straight one. Also, just for convenience, I made the floor and obstacles in different colors as Unity Game Engine allows to do so. The next step will be to add the Agent that the player can control by mouse, to do so, firstly I created a 3D Object, a sphere.

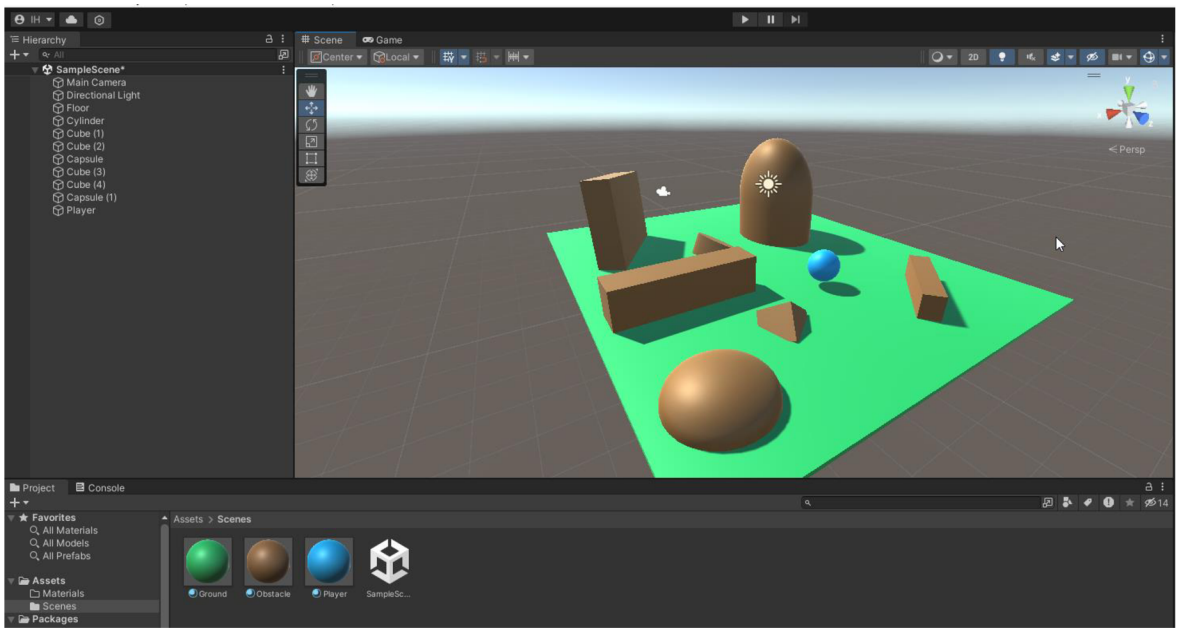

Figure 3 - Basic scene for further Pathfinding testing, source: own processing

4.1.2 NavMesh

To make the scene act like a navigation grid and our Agent navigate through it, I proceeded with settings. First of all, the player agent should be assigned with the "Nav Mesh Agent" component, to do so – Player -> Add Component -> Nav Mesh Agent. After those steps, Unity will identify the Agent as a "Nav Mesh Agent" which means that it can now navigate through the Navigation Area. Then I assigned the Floor a "Navigation Static" property so that it will be counted as a Navigation Area, to do so we have to switch to Nav Mesh Window mode – Window -> AI -> Navigation. To count our floor as a Navigation Area I switched to its properties and checked "Navigation Static" and made it "Walkable". For our obstacles, I did the same but instead of "Walkable" I chose "Not Walkable"

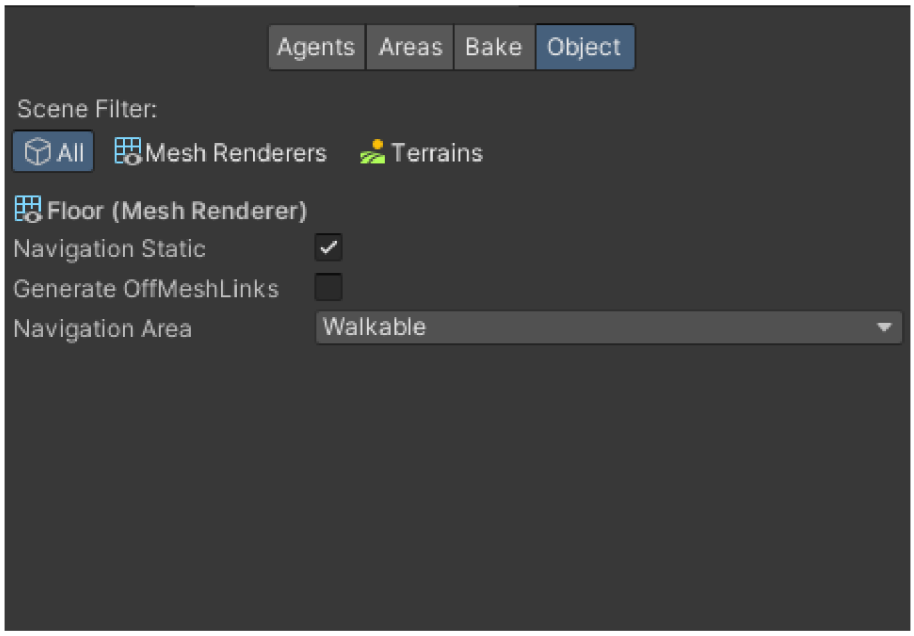

Figure 4 - Settings for NavMesh Surface, source: own processing

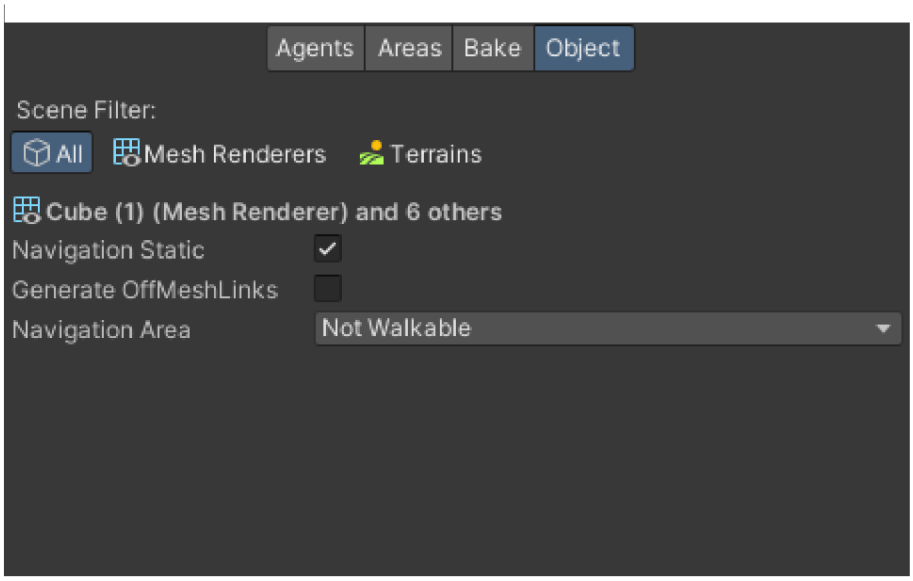

Figure 5 - Settings for NavMesh Obstacle, source: own processing

After that, I baked it so that I could see what my Nav Mesh grid looks like.

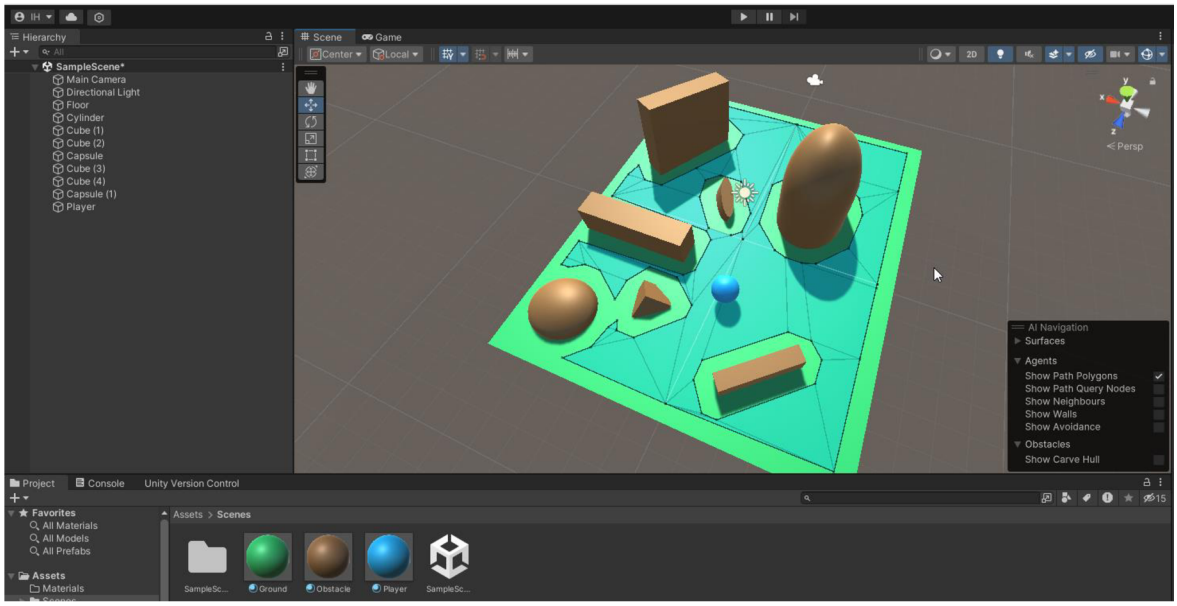

Figure 6 - Generated NavMesh including Obstacles, source: own processing

To test out how the Agent finds the path I wrote a script that will allow the player to control the Agent by clicking the mouse. Also, I implemented a trail (red color) and added visualization for clicked point (pink color) so that it can be visible which path does Agent has. The script looks like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;

public class ControllingOfAgent : MonoBehaviour
{
    public NavMeshAgent agent;
    public TrailRenderer trailRenderer;
    public GameObject clickIndicatorPrefab;

    private GameObject currentClickIndicator;

    void Start()
    {
        trailRenderer = GetComponent<TrailRenderer>();
    }


    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Ray positionToMove = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(positionToMove, out var hitInfo))
            {
                agent.SetDestination(hitInfo.point);

                Destroy(currentClickIndicator);
                currentClickIndicator = Instantiate(clickIndicatorPrefab, hitInfo.point, Quaternion.identity);
            }
        }
    }
}
```

Figure 7- Script for Agent controlling, source: own processing

Basically, the script does the following things:

- The “Start” method initializes the “TrailRenderer” component by getting a reference to it from the same “GameObject” as the script. This component will be used to create a visual trail behind the moving agent.
- The “Update” method listens for mouse clicks “Input.GetMouseButtonDown(0)”. When a click is detected, a ray is cast from the camera to the clicked position on the screen using “Camera.main.ScreenPointToRay(Input.mousePosition)”.
- The “NavMeshAgent” component (agent) is used to move the “GameObject”. When a valid terrain point is clicked “Physics.Raycast” hits something, the agent's destination is set to the clicked point using “agent.SetDestination(hitInfo.point)”.
- Before setting the destination, the script destroys the previous click indicator “Destroy(currentClickIndicator)” if it exists. It then instantiates a new click indicator “Instantiate(clickIndicatorPrefab, hitInfo.point, Quaternion.identity)” at the clicked position.
- The script uses a “TrailRenderer” to create a visual trail behind the moving agent. The “TrailRenderer” is attached to the same “GameObject” as the script “trailRenderer = GetComponent<TrailRenderer>()”.
- The “trailRenderer” is assigned a reference to the “TrailRenderer” component to allow manipulation of the visual trail during runtime.

As it can be seen, the Agent is now controllable by the player, and without any further instructions or settings it can now go to any point available on the Floor, and it also considers obstacles, so there is no need to create a map of obstacles. I have made several tests and as can be seen in the figure below, the AI does its job perfectly, all I have to do as a player is just click on the point where I need my Agent to go, as a developer I didn’t need to spend so much time on the map of obstacles, implementing some coordinates map for routes so that Agent could find its path and have a large experience with scripting.

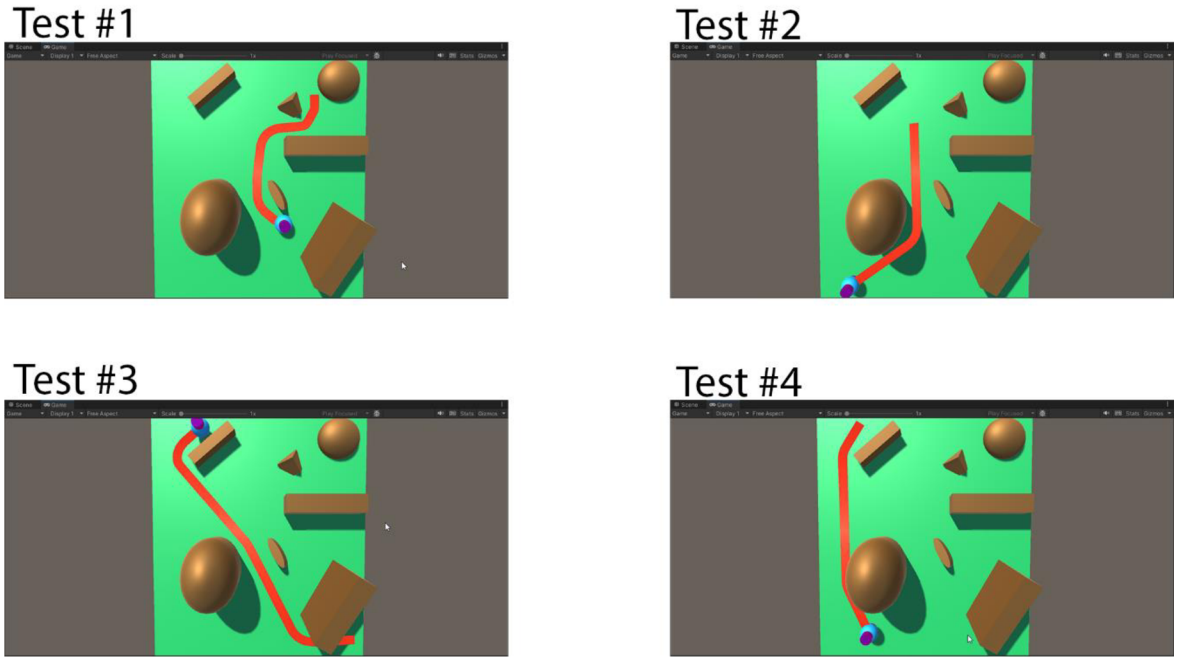

Figure 8- Pathfinding through NavMesh tests, source: own processing

4.2 Implementation of PCG using Unity

Unfortunately, Unity doesn't have a built-in toolkit to generate terrains that are based on noises which is an essential element of PCG. To do so, I will have to write a script that will handle the creation of noise texture and after that implement it on the terrain, so we will see completely new generated terrain. But Unity has a built-in function called Perlin Noise that I chose to create a noise texture that takes 2 float values "x" and "y" that represents the input coordinates for generating Perlin noise and returns a float value from the range between 0.0 and 1.0 representing Perlin noise at the specified input coordinates ("x" and "y" accordingly) (21). The goals are:

- Create an algorithm that will make a 256 by 256 pixels texture of Perlin Noise.
- Implement this texture as a height map of terrain.
- The Perlin Noise texture should allow to generate pseudo-random terrains every time the user starts the game.

4.2.1 Initial setup

To begin with, the creation of the project is similar to what I did in the Pathfinding part of my thesis Practical Part. However, the scene that I will create will now be a little bit different, as to implement Perlin Noise generated height-map texture I will need to apply it

to the Terrain type 3D object instead of a plain cube, as it was before. So now my scene looks like this:

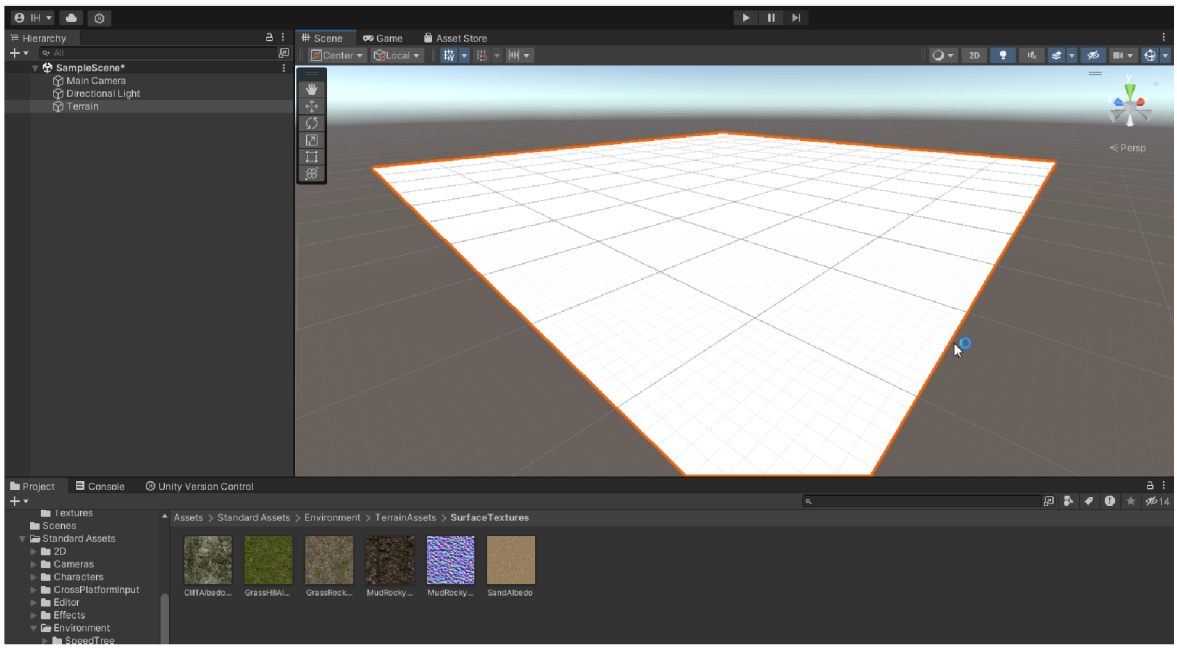

Figure 9 - Initial plain Terrain object in Unity Game Engine, source: own processing

After that I added a component to my Terrain - a script file and named it “TerrainGenerator”, in this file I will write the generation of Perlin Noise texture and immediately apply it to the Terrain on the scene.

4.2.2 Perlin Noise implementation

I have written this script, that generates Perlin Noise data and applies it directly to the terrain:

```
using System.Collections;
using UnityEngine;

public class TerrainGenerator : MonoBehaviour
{
    public int terrainWidth = 256;
    public int terrainHeight = 256;
    public int terrainDepth = 25;
    public float terrainScale = 15f;

    public float randomOffsetU = 1f;
    public float randomOffsetV = 1f;

    void Start()
    {
        randomOffsetU = Random.Range(0f, 9999f);
        randomOffsetV = Random.Range(0f, 9999f);

        Terrain terrain = GetComponent<Terrain>();
        terrain.terrainData = GenerateTerrain(terrain.terrainData);
    }

    TerrainData GenerateTerrain(TerrainData terrainData)
    {
        terrainData.heightmapResolution = terrainWidth + 1;
        terrainData.size = new Vector3(terrainWidth, terrainDepth, terrainHeight);
        terrainData.SetHeights(0, 0, PerlinNoiseHeights());
        return terrainData;
    }

    float[,] PerlinNoiseHeights()
    {
        float[,] heights = new float[terrainWidth, terrainHeight];
        for (int x = 0; x < terrainWidth; x++)
        {
            for (int y = 0; y < terrainHeight; y++)
            {
                heights[x, y] = PerlinNoiseCalcHeight(x, y);
            }
        }

        return heights;
    }

    float PerlinNoiseCalcHeight(int x, int y)
    {
        float xCord = (float)x / terrainWidth * terrainScale + randomOffsetU;
        float yCord = (float)y / terrainHeight * terrainScale + randomOffsetV;

        return Mathf.PerlinNoise(xCord, yCord);
    }
}
```

Figure 10 - Script for terrain generation, source: own processing

Basically, this script does the following things:

- The script exposes several parameters, such as “terrainWidth”, “terrainHeight”, “terrainDepth”, and “terrainScale”. These parameters allow developers to customize the dimensions and appearance of the generated terrain.
- Method Start the initial script every time when the user starts the demo.
- Also, I have implemented offsets (“randomOffsetU” and “randomOffsetV”) so that the data will be pseudo-random at each startup of the demo

- The “GenerateTerrain” method configures the terrain data by setting the heightmap resolution and size based on the specified parameters. It then populates the height map using the “PerlinNoiseHeights” method.
- The “PerlinNoiseHeights” method iterates over the terrain dimensions and calculates heights using the “PerlinNoiseCalcHeight” method.
- The “PerlinNoiseCalcHeight” method converts 2D coordinates to Perlin noise coordinates, considering the random offsets. It then utilizes “Mathf.PerlinNoise” to obtain a Perlin noise value, representing terrain height.
- The script utilizes the Unity “Vector3” size property to define the dimensions of the terrain in the x, y, and z axes.
- The “SetHeights” method is used to apply the generated height map to the terrain. This method sets the heights of the terrain vertices based on the calculated Perlin noise values.

Now every time when the demo starts we can observe that the landscape is always different, I as a developer don’t need to create many variants of terrains so that players can see the diversity of that, because now generated Perlin Noise do it for me.

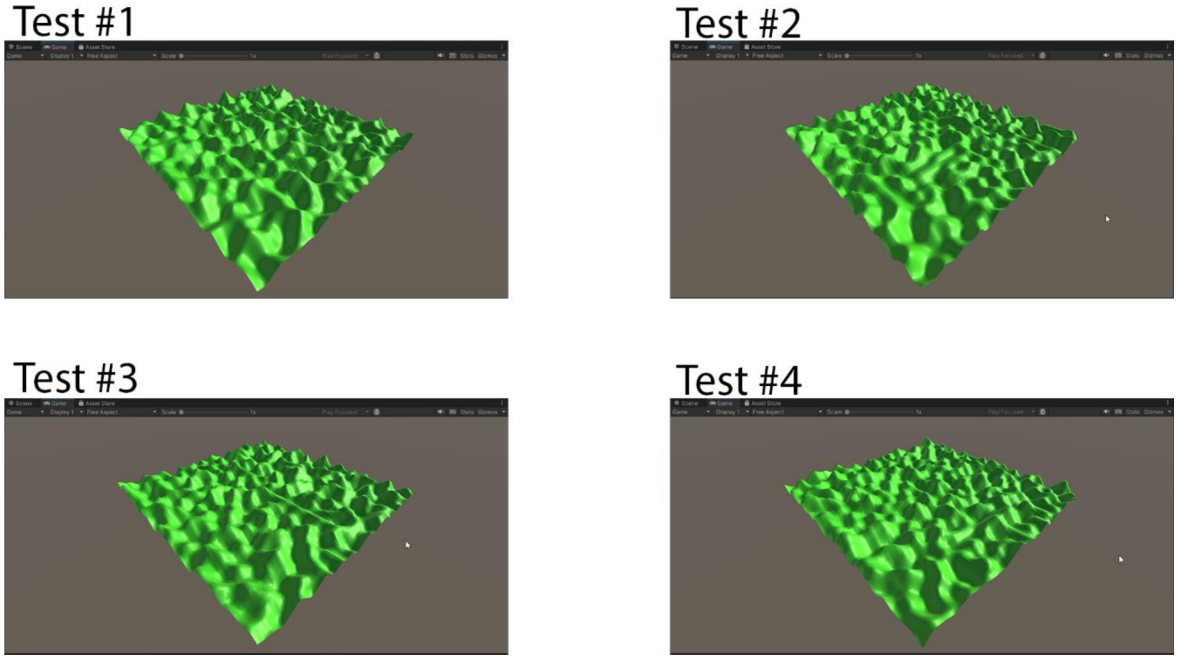

Figure 11 - PCG of terrain through Perlin Noise, source: own processing

4.3 Combination of both PCG and Pathfinding using Unity

The final step will be to implement both PCG and Pathfinding so that the player can explore our demo world which will be always generated randomly. To do so I made some

changes in my script for Perlin Noise terrain generation. I have added two variables "obstaclePrefab" (data type – "GameObject". I have added dependency on the Prefab "Cylinder" so that it imitates an obstacle) and "numberOfObstacles" (data type – int. This variable handles a number of obstacles that will be placed on the terrain). Also, I have added a method, which will place randomly those prefabs:

```
void PlaceRandomObstacles()
  {
    for (int i = 0; i < numberOfObstacles; i++)
    {
      float randomX = Random.Range(0, terrainWidth);
      float randomY = Random.Range(0, terrainHeight);
      float height = PerlinNoiseCalcHeight((int)randomX, (int)randomY);
      Vector3 obstaclePosition = new Vector3(randomX, height * terrainDepth, randomY);

      Instantiate(obstaclePrefab, obstaclePosition, Quaternion.identity);
    }
}
```

After those changes the Terrain now looks like this, all the obstacles are placed randomly each time, when the Player starts the demo:

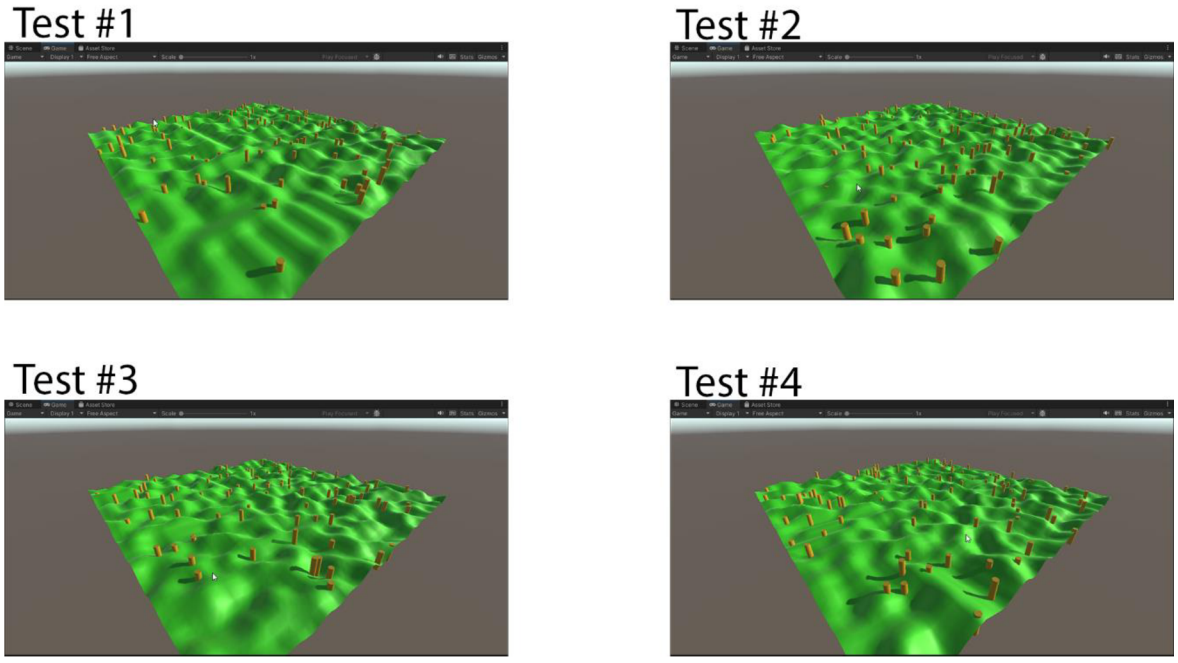

Figure 12 - Randomly placed obstacles on PCG terrain, source: own processing

I used the same script that I had developed in the NavMesh part for my Agent, however, there is one more change that I have implemented in my script for generating the Terrain –

I have added a method to bake NavMesh Surface every time when new Terrain was created:

```
void BakeNavMesh()
{
    NavMeshSurface navMeshSurface = GetComponent<NavMeshSurface>();
    navMeshSurface.BuildNavMesh();
}
```

Now we can finally see how both Pathfinding and PCG Terrain generation work together. Combining both approaches let me to create different terrain with different obstacles, but the Agent still manages to navigate through it:

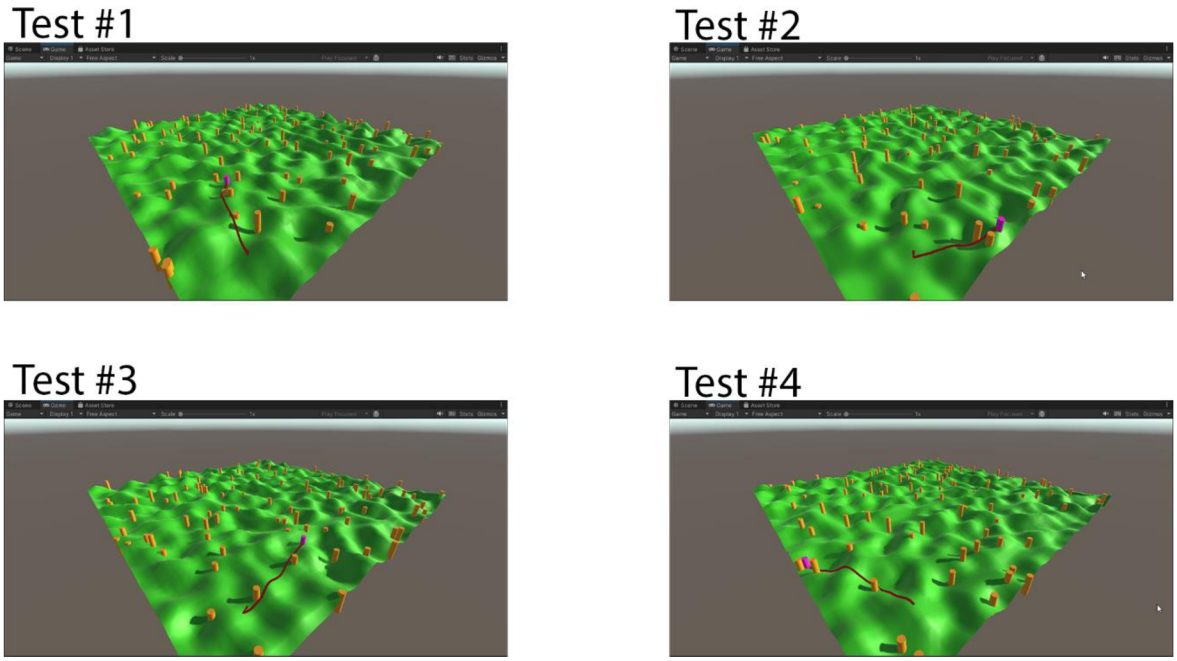

Figure 13 - Pathfinding on PCG terrain with randomly placed obstacles tests, source: own processing

5 Results and Discussion

Based on the comprehensive literature review and the practical implementations detailed in this thesis, the primary objectives of exploring AI's role in game development have been successfully achieved. The practical part was primarily focused on implementing Pathfinding and PCG for specific scenarios.

Pathfinding Implementation: The practical application of pathfinding algorithms, particularly the use of Unity's NavMesh system with A* algorithm that lies in the core of this built-in tool, demonstrated efficient navigation of game characters in complex environments. This implementation showcased the AI's capability to dynamically adapt to varying terrain and obstacles, emphasizing the importance of AI in enhancing in-game realism and interactivity.

Procedural Content Generation: The practical part considered the implementation of PCG with the usage of the Perling Noise algorithm for creating a pseudo-randomly generated terrain. This application showed that PCG can handle the creation of diverse and evolving game landscapes, as a result, a unique player experience can be provided each playthrough. The combination of both AI techniques resulted in interesting gameplay solutions and decrease in development time and human efforts in the creation of video games.

One potential area for further exploration could be the integration of more complex AI models for NPC behavior and player interaction by using Player Behavior Analysis. While the current implementations effectively demonstrate basic AI functionalities in game development, expanding the scope to include more advanced AI techniques could lead to even more engaging and realistic game scenarios. For example, a combination of Cellular Automata and Perlin Noise adds some points of interest for the player, like lakes, rivers, trees, ancient ruins, etc.

6 Conclusion

This thesis aims to investigate and implement AI technologies in game development. The study successfully demonstrated the practical applications of AI technologies, highlighting their significance in modern game development.

The theoretical part of the thesis provided a solid foundation for understanding the various AI technologies available for game development. It also set the stage for the practical implementations, which were key to demonstrating the real-world applicability of the concepts discussed.

In the practical section, the successful implementation of AI in pathfinding and PCG within Unity showcased the potential of AI to revolutionize game development. The use of Unity's NavMesh for pathfinding and the generation of dynamic terrains and obstacles using the Perlin Noise algorithm underscored the efficiency and versatility of AI in creating complex and engaging game environments.

AI implementation showed that it decreases development time and effort. However, AI provides a less detailed environment if it is being made by human beings, it also provides various variations of this environment. AI-driven pathfinding is effective in terms of navigation (especially, previously discussed, A* algorithm that lies on the basis of Unity's NavMesh system) and doesn't require directly written paths from developers.

This thesis contributes to the field by offering a detailed examination and practical demonstration of AI's role in game development. The insights gained could be valuable for both academic research and game development practices.

7 References

1. Hitberry Games. Indie, AA, and AAA Games: The Ultimate Guide. *HitberryGames.* [Online] November 27, 2023. [Cited: March 8, 2024.] https://www.hitberrygames.com/post/indie-aa-and-aaa-games-the-ultimate-guide.

2. DrawAndCode. What is a game engine: an essential overview for beginners. *DrawAndCode.* [Online] DrawAndCode, 25 April 2023. [Cited: 26 August 2023.] https://drawandcode.com/learning-zone/what-is-a-game-engine/.

3. Unity Technologies. Creating and Using Scripts. *Unity Documentation.* [Online] Unity Technologies, March 19, 2018. [Cited: December 19, 2023.] https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.

4. Unity Technologies. Real-time rendering in 3D. *Unity how-to.* [Online] Unity Technologies. [Cited: August 27, 2023.] https://unity.com/how-to/real-time-rendering-3d.

5. Unity Technologies. Unity - Scripting API: NavMesh. *Unity Documentation.* [Online] January 13, 2024. [Cited: January 14, 2024.] https://docs.unity3d.com/ScriptReference/AI.NavMesh.html.

6. Epic Games. Unreal Engine 5.2 is now available! *Unreal Engine.* [Online] Epic Games, May 11, 2023. [Cited: September 23, 2023.] https://www.unrealengine.com/en-US/blog/unreal-engine-5-2-is-now-available.

7. *Pathfinding Algorithms in Game Development.* Abdul Rafiq, Tuty Asmawaty, Abdul Kadir, Siti Normaziah Ihsan. 1, Pahang, Malaysia : IOPSCIENCE, 2019, Vol. 1. DOI 10.1088/1757-899X/769/1/012021.

8. *Procedural Content Generation for Games: A Survey.* Hendrikx, Mark & Meijer, Sebastiaan & Velden, Joeri & Iosup, Alexandru. 1, The Netherlands : ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP), 2013, Vol. 9. DOI: 10.1145/0000000.0000000.

9. *Simulation of artificial intelligence in a computer game.* Elena V Soboleva, Nadezda V Shalaginova. 3, Kirov : IOPSCIENCE, 2019, Vol. 1399. DOI 10.1088/1742-6596/1399/3/033050.

10. G. N. Yannakakis, J. Togelius. *Artificial Intelligence and Games.* Cham, Switzerland : Springer International Publishing AG, 2018. ISBN 978-3-319-63519-4.

11. Ares Lagae, Sylvain Lefebvre, Rob Cook, T. Derose, George Drettakis, David S. Ebert, J.P. Lewis, Ken Perlin, Matthias Zwicker. State of the Art in Procedural Noise Functions.

ResearchGate. [Online] May 2010. [Cited: November 9, 2023.] https://www.researchgate.net/publication/216813586_State_of_the_Art_in_Procedural_Noise_Functions.

12. Noor Shaker, Julian Togelius, Mark J. Nelson. *Procedural Content Generation in Games.* Switzerland : Springer, 2016. ISBN 978-3-319-42716-4.

13. Tatarinov, Andrei. GraphiCon 2008. *GraphiCon.* [Online] 2008. [Cited: October 28, 2023.] https://www.graphicon.ru/html/2008/proceedings/English/S8/Paper_3.pdf.

14. Mandelbrot, Benoit B. *The Fractal Geometry of Nature.* New York : Freeman & Company, W. H, 1982. 0716711869.

15. Andrew Pech, Philip Hingston, Martin Masek, Chiou Peng Lam. Artificial Life and Computational Intelligence. *Evolving Cellular Automata for Maze Generation.* Newcastle : Springer, Cham, 2015.

16. How Minecraft Generates Worlds. *Alphr.* [Online] Alphr, August 1, 2023. [Cited: December 30, 2023.] https://www.alphr.com/how-minecraft-generates-worlds/.

17. *Player behavioural modelling for video games.* Sander C.J. Bakkes, Pieter H.M. Spronck, Giel van Lankveld. 3, The Netherlands : Entertainment Computing, 2012, Vol. 3. ISSN 1875-9521.

18. Hildmann, Hanno. MDPI. *Designing Behavioural Artificial Intelligence to Record, Assess and Evaluate Human Behaviour.* [Online] September 25, 2018. [Cited: December 12, 2023.] https://www.mdpi.com/2414-4088/2/4/63.

19. Curtis Bennett, Dan Violet Sagmiller. *Unity AI Programming Essentials.* Birmingham, UK : Packt Publishing, 2014. ISBN 978-1-78355-355-6.

20. Unity Technologies. *Unity Documentation.* [Online] [Cited: December 30, 2023.] https://docs.unity.com/.

21. Unity Technologies . Unity - Scripting API:Mathf.PerlinNoise. *Unity Documentation.* [Online] January 13, 2024. [Cited: January 14, 2024.] https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html.

8 List of pictures, tables, graphs and abbreviations

8.1 List of pictures

8.2 List of abbreviations

UE – Unreal Engine

PCG – Procedural Content Generation

DFS – Depth-First Search

BFS – Breadth-First Search

NPC – Non-Playable Character

AI – Artificial Intelligence

GUI – Graphical User Interface

CA – Cellular Automata

RTS – Real-Time Strategy