DOKUZ EYLÜL UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

A DEEP REINFORCEMENT LEARNING APPROACH FOR PATHFINDING IN COMPUTER GAMES

by
Doğaç EKİCİ

February, 2023
İZMİR

A DEEP REINFORCEMENT LEARNING APPROACH FOR PATHFINDING IN COMPUTER GAMES

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
in Department of Computer Science

by
Doğaç EKİCİ

February, 2023
İZMİR

M.SC THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**A DEEP REINFORCEMENT LEARNING APPROACH FOR PATHFINDING IN COMPUTER GAMES**" completed by **DOĞAÇ EKİCİ** under supervision of **ASSOC. PROF. DR. METE EMİNAĞAOĞLU** and we certify that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

……

Assoc. Prof. Dr. Mete EMİNAĞAOĞLU

Supervisor

……	……
Assoc. Prof. Dr. K. Ulaş BİRANT	Assoc. Prof. Dr. Korhan KARABULUT
(Jury Member)	(Jury Member)

……

Prof. Dr. Okan FISTIKOĞLU

Director

Graduate School of Natural and Applied Sciences

A DEEP REINFORCEMENT LEARNING APPROACH FOR PATHFINDING IN COMPUTER GAMES

ABSTRACT

One of the biggest challenges of game development is to produce a pathfinding algorithm that both produces satisfactory realistic movement results and can solve different scenarios in worlds created by game developers' unlimited imagination. Furthermore, since the games are programs for the end user, it is desired that the systems they contain use as little computer resources as possible and be developed as quickly as possible in terms of cost. Although existing solutions can produce strong answers to the problem, they also contain some chronic problems that take a long development time to solve. Existing solutions work very well on maps that are continuous and can be navigated by moving from start to finish. However, they cannot find a solution in cases where various obstacles must be overcome by various movement mechanics such as jumping, flying are used other than walking. Most of the time developers must manually assign links to meshes, significantly prolonging the game development process on the maps with such features. At the same time, since the connected links are established manually, the movement of the object moving on the link does not seem natural. The focus of this study is to create a system that will generate a node network by using artificial neural networks and deep reinforcement learning to overcome the difficulties of existing pathfinding algorithms. Finally, a system that is fast and uses less resources is aimed for the end user, since artificial neural networks will not be used during the build phase.

Keywords: Pathfinding, artificial neural networks, game development, deep reinforcement learning

BİLGİSAYAR OYUNLARINDA YOL BULMADA DERİN PEKİŞTİRMELİ ÖĞRENME YAKLAŞIMI

ÖZ

Oyun geliştirmenin en büyük zorluklarından biri, hem tatmin edici derecede gerçekçi hareket sonuçları üreten hem de oyun geliştiricilerin sınırsız hayal gücünün yarattığı dünyalarda farklı senaryoları gerçekleştirebilen bir yol bulma algoritması üretmektir. Ayrıca oyunlar son kullanıcıya yönelik programlar olduğu için içerdikleri sistemlerin mümkün olduğunca az bilgisayar kaynağı kullanması ve maliyet açısından mümkün olan en kısa sürede geliştirilmesi istenmektedir. Mevcut yol bulma çözümleri, soruna güçlü yanıtlar üretebilse de çözülmesi uzun bir geliştirme süresi gerektiren bazı kronik sorunları da içermektedir. Bu çözümler, baştan sona gezinilebilen ve sürekli haritalarda çok iyi çalışmaktadır. Ancak yürüme dışında zıplama, uçma gibi farklı hareket mekaniklerinin de kullanıldığı çeşitli engellerin aşılması gereken durumlara çözüm getirememektedir. Çoğu zaman geliştiricilerin yol bulma örgülerini birbirine linklerle manuel olarak bağlaması gerekmektedir. Bu da farklı hareket mekaniklerine sahip haritalarda oyun geliştirme sürecini önemli ölçüde uzatmaktadır. Aynı zamanda bağlanan linkler manuel olarak kurulduğu için link üzerinde hareket eden cismin hareketi doğal görünmemektedir. Bu çalışmanın odak noktası, mevcut yol bulma algoritmalarının zorluklarını aşmak için yapay sinir ağları ve derin pekiştirmeli öğrenme kullanarak bir düğüm ağı oluşturacak bir sistem oluşturmaktır. Son olarak, yapım aşamasında yapay sinir ağları kullanılmayacağından son kullanıcı için hızlı ve daha az kaynak kullanan bir sistem hedeflenmektedir.

Anahtar Kelimeler: Yol bulma, yapay sinir ağları, oyun geliştirme, derin pekiştirmeli öğrenme

CONTENTS

Page

M.Sc THESIS EXAMINATION RESULT FORM ii

ABSTRACT iii

ÖZ iv

LIST OF FIGURES vii

LIST OF TABLES ix

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 LITERATURE REVIEW 4

2.1 Graph Generation from Topology 4

2.2 Path Search 7

2.3 Deep Reinforcement Learning 7

2.4 Convolutional Neural Networks 10

2.5 Unity ML Agents Package 13

CHAPTER 3 PROPOSED MODEL 14

3.1 Generating Node Matrix 16

3.2 Creating Character 23

3.3 Machine Learning Agent Training 31

3.4 Node Training 35

3.5 Pathfinding 38

3.6 Results 44

CHAPTER 4 CONCLUSION 49

REFERENCES 52

LIST OF FIGURES

Page

Figure 1.1 Navigation mesh islands and links 2
Figure 2.1 Waypoint pathfinding structure in a dynamic environment 5
Figure 2.2 Visibility graph 6
Figure 2.3 Different representations of an environment 6
Figure 2.4 Reinforcement learning cycle 8
Figure 2.5 Gradient descent 9
Figure 2.6 Learning rate 9
Figure 2.7 Extracting features from image class 10
Figure 2.8 Convolution 11
Figure 2.9 Pooling 12
Figure 2.10 Convolution architecture 12
Figure 3.1 General architecture of namespaces 16
Figure 3.2 Cage point path generated by final cage points 17
Figure 3.3 Node Creation tab on Nav Ai Main Manager 18
Figure 3.4 Before and after the generation of final cage points 18
Figure 3.5 Generated final cage points 19
Figure 3.6 Two to one right-angled triangle object tile set 19
Figure 3.7 Created tile palette 20
Figure 3.8 Ground tile map object 20
Figure 3.9 Before and after the composite collider 21
Figure 3.10 Before and after generation of nodes 23
Figure 3.11 Character selection tab on main manager class 24
Figure 3.12 Character object hierarchy 25
Figure 3.13 Player node check object 25
Figure 3.14 Ground check rays and ground direction rays 27
Figure 3.15 Wall check collider 28
Figure 3.16 Character collider 29
Figure 3.17 Character state machine and movement scripts 30
Figure 3.18 Scene view vs. what agent camera sees 32

Figure 3.19 Default Unity hyperparameters..... 33
Figure 3.20 An agent target placement that is unlikely to be successful in a short time 34
Figure 3.21 Character origin and stable nodes..... 36
Figure 3.22 Different paths for different ending nodes..... 37
Figure 3.23 Node link 38
Figure 3.24 Path to target..... 39
Figure 3.25 Possible paths between red node groups 41
Figure 3.26 Selected path for wall jumping character; red rectangle nodes are stable node groups, blue line nodes are node group links 42
Figure 3.27 Character following the path 43
Figure 3.28 Fifty thousand steps of training on the main environment 44
Figure 3.29 Node link training using neural network after training, each percent represents about 500 nodes 46
Figure 3.30 The orange zone is the area that the agent can never reach, and the blue zone is the areas that are difficult to reach unless target object is specially located to target that area 47
Figure 3.31 Difficulty map..... 47

LIST OF TABLES

Page

Table 4.1 Agents' reward values 45

CHAPTER 1
INTRODUCTION

Pathfinding is one of the most fundamental problems encountered in game development and robotics. Whether it is a game character or a robot, pathfinding is used to calculate the path that all computer-controlled objects will reach from one point to another. For these objects that usually need to find their way on complex maps, it must be able to find solutions to many problems. How the object collects data on obstacles and roads around it, and in which data format the map data of this world should be stored are the first problems to be solved. The nature of the problem is also important. Is a system envisaged that will travel only through known routes between some known stops, or is a system that can provide free access to all locations on the map is intended? What data structure should the entire map be converted to in order to find a path, and how should the path between two points be determined in this structure? And finally, what forces must the object be subjected to, to follow this path?

As might be expected, pathfinding is a multi-layered problem. It is necessary to determine the nature of the problem correctly and to determine how a pathfinding strategy will be developed, specific to the problem. In cases where the environment is static or dynamic, applications with different features may be needed.

Pathfinding systems are generally using a lot of computer resources because of their multi-layered structure and often searching large data sets (Krafft, 2021). Therefore, optimization is of great importance. For pathfinding applications using machine learning, there are also studies that aim to achieve lower cost applications with supervised learning methods instead of reinforcement learning for pathfinding optimization in real world robotic systems (Abreu, 2022).

One of the difficulties of pathfinding applications in games is that game maps usually contain many obstacles and discontinuities. To go from one place to another on the game maps, the agents sometimes need to use different mechanics such as jumping, teleporting, dashing, and using ladders. One of the best examples of this is

platformer genre games. It can be said that pathfinding is quite challenging in the platformer genre, which usually has much more movement mechanics than other game genres (Iskandar, Diah, Marina & Ismail, 2020). If the map is fragmented, the agent will not be able to apply pathfinding between separate navigation islands. To circumvent this situation, the links connecting these islands are used as shown in figure 1.1.

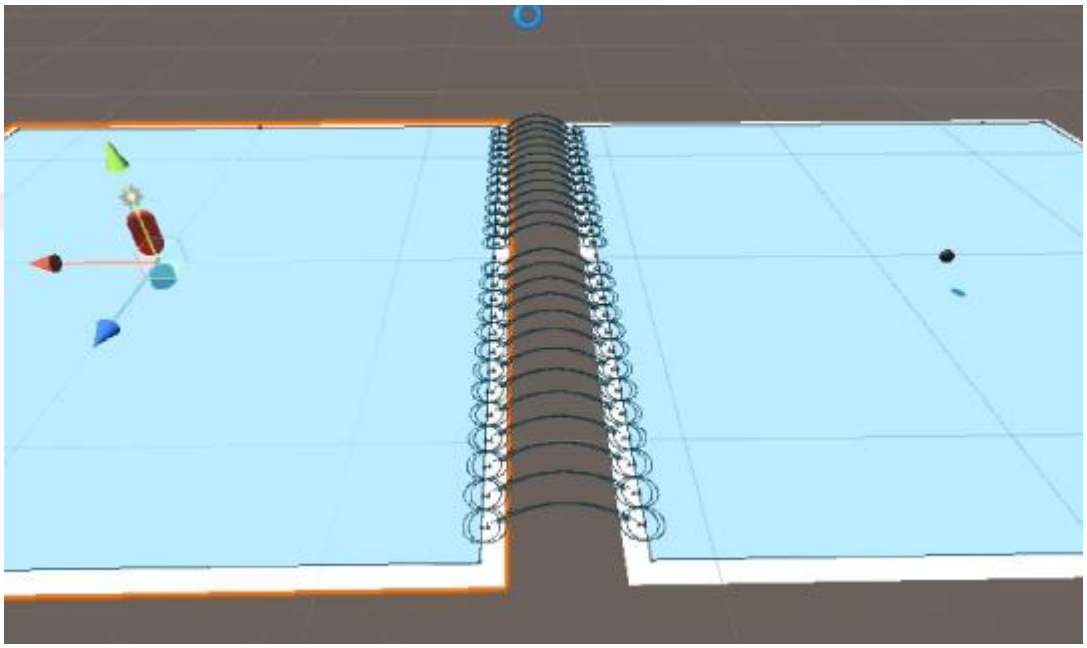


Figure 1.1 Navigation mesh islands and links

Although the links connecting the islands are a good solution, they bring many problems. It is possible to automatically assign links, but in this case, most of the times movement they create seems unnatural. On the other hand, when the links are connected manually, it requires a serious workforce. Furthermore, since it is a separate structure detached from normal navigation islands, the behavior of a character on this link and its interaction with the environment must be programmed from scratch. Therefore, for a link structure that is intended to look natural, the character's movement, interaction with the environment, and animations must be tailored made to make it look like the character is using talent, which also causes an increase in the workload (Alonso, Peter, Goumard & Romoff, 2020).

This study focuses on developing pathfinding in a side-scroller, platformer type game by making use of machine learning methods to speed up the process during the development phase. In the study, the pathfinding system is not designed as islands and links, but as a system of nodes connected to each other by links. Islands will be created on nodes in the process, thanks to the relationship of the nodes with the character. In this way, since the whole system will consist of the same objects, it is aimed to prevent an unnatural looking movement behavior. In addition, it is aimed to place the links between nodes by a trained machine learning agent, thus saving the time to create the links. Finally, it is aimed to create a structure in which artificial neural networks are not used for the final product and make the final product solvable by classical pathfinding algorithms. In this way, it is aimed to load the majority of the workload on the development stage and to produce a structure that consumes the lowest possible resource for the final product. To narrow the scope of the study, a dynamically structured map was not used, and pathfinding is made for a single agent at the same time on the map.

CHAPTER 2
LITERATURE REVIEW

In this part, it is aimed to summarize the literature background of the layers required for the targeted application. To create a pathfinding application, the available map must be simplified and stored as a meaningful data structure. Simplified maps are represented by various graph structures. Some graph models will be examined in the graph generation from topology section. In the second section, path search algorithm used on this study will be discussed roughly. In the third and fourth sections, reinforcement learning, and convolutional neural networks algorithms will be summarized as they were used in the study, respectively. In the last part, Unity game engine and its machine learning add-on package will be summarized.

2.1 Graph Generation from Topology

For pathfinding, the game map needs to be simplified and made into a workable data structure. For this purpose, many different data structures have been derived. These structures can be examined as two different types as waypoints and grids (Graham, McCabe, & Sheridan, 2003). The waypoint system is a simplified version of the map converted into graphs with nodes and edges between nodes. The object moves from the point where it is located to the point it wants to reach, by finding the shortest path with various search algorithms. A waypoint system produced for a dynamic environment can be examined from figure 2.1.

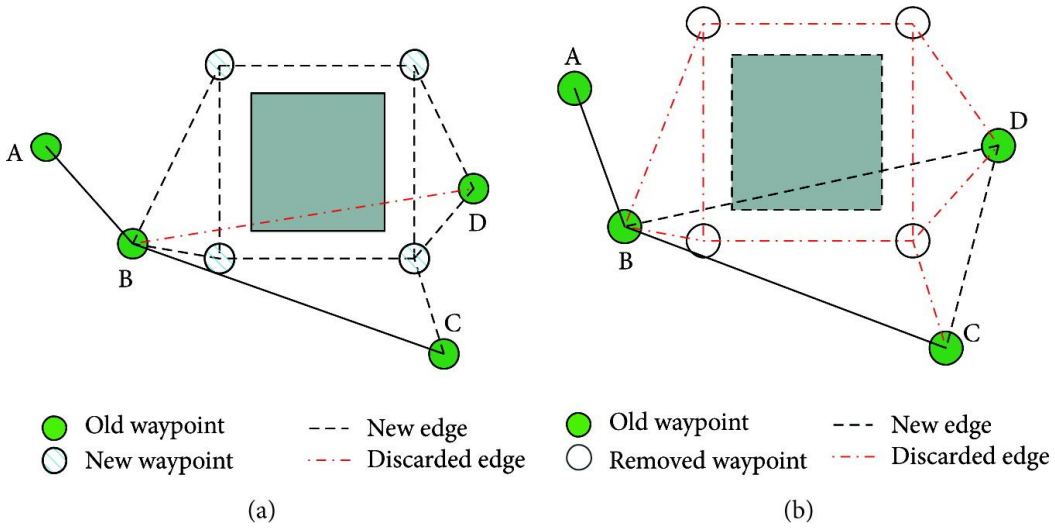


Figure 2.1 Waypoint pathfinding structure in a dynamic environment (Zhu et al., 2015)

Although the waypoint system continues to be used in the game industry, especially in simpler games, it creates a computational load as the size of the game grows and the freedom of movement increases. Therefore, grid systems have been used mostly in games industry in recent years (Alonso, Peter, Goumard & Romoff, 2020).

In grid systems, the usable parts of the map are divided into parts in an orderly way. Here, the rule may be to divide the map into equal parts so that each part is the same size, or irregular sub-parts can be created based on the map geometry. In this sense, these data structures can be divided into regular and irregular (Algfoor, Sunar, & Kolivand, 2015).

Regular grids divide the map into sub-sections according to a certain geometry. Square, triangular, and hexagonal grids are mostly preferred. There are studies in the literature that compare these grids with each other and measure their response to different search algorithms (Bjornsson, Enzenberger, Holte, Schaeffer, & Yap, 2003).

Visibility graphs are the first of the irregular grid types. Obstacle geometries on the map are considered when creating visibility graphs. This data type will subdivide the map by dividing it at its corners. These parts are obtained by combining all the corners

with the other corners in their field of view. (Ghosh, & Goswami, 2013). An example of a visibility graph is shown in figure 2.2.

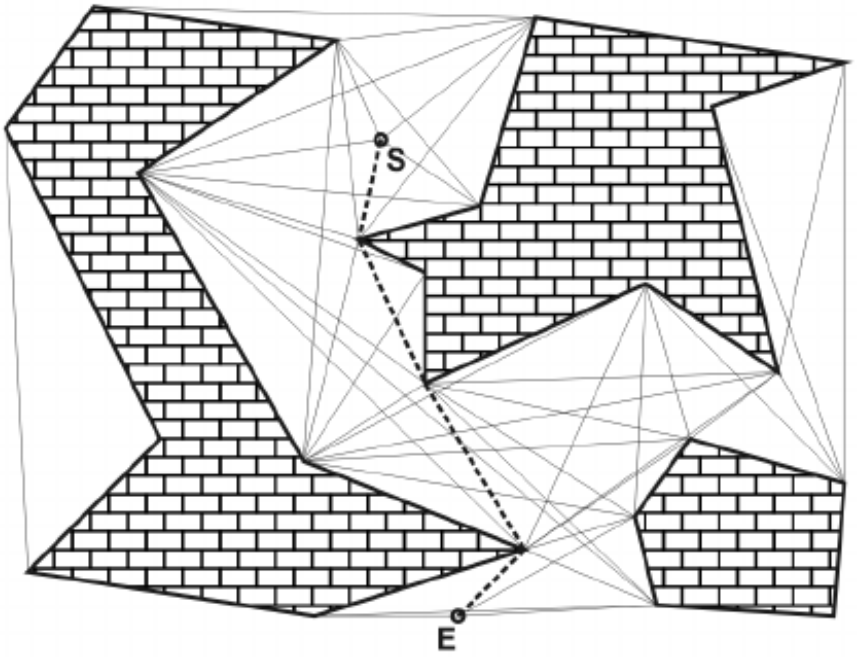


Figure 2.2 Visibility graph (Brož, Kolingerová, & Apu, 2007)

On the other hand, navigation meshes are used to divide a map into convex polygons. This mesh type is often highly preferred due to its low memory and computational requirements (Cui, Harabor, & Grastien, 2017). Navigation meshes are produced from a three-dimensional environment. Its production can be examined from figure 2.3.

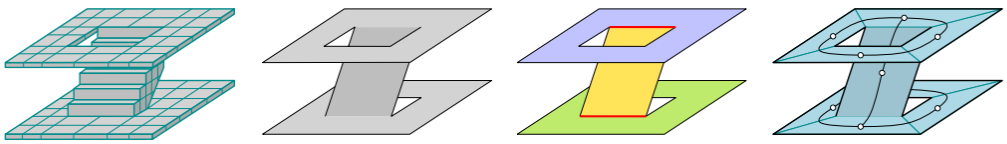


Figure 2.3 Different representations of an environment (van Toll et al., 2016)

In the model proposed in this study, the map is divided into square grids. However, grids are connected to each other by logical links. Therefore, the model proposed in the study shows both waypoint and square grid features.

2.2 Path Search

In this study, an adapted version of a greedy best first search algorithm for a two-level nested model, is used to reveal the path among the trained nodes. In greedy best-first search algorithm, the path is unfolded by selecting the locations with the lowest heuristic function value. The heuristic function can be determined in different ways according to the content of the problem and will directly affect the speed and optimization of the algorithm. The algorithm, which selects between the points that can be reached directly from the starting point in the first step, adds the newly selected point to the list of selected points at each step, and includes new points that can be reached from the last added point in the search (Lawande, Jasmine, Anbarasi, & Izhar, 2022).

2.3 Deep Reinforcement Learning

Reinforcement learning is a method in which the agent, who has not been given any prior knowledge, learns through the method of reward and punishment by trying to reach a higher reward in the environment it is in. The agent decides on an action (a) in the environment. As a result of this decided action, the environment and the agent move to a new state (s). In the transition to the new state, the reward can appear as positive or negative. The agent will choose a new action based on the estimated reward values and the new state. (Kaelbling, Littman, & Moore 1996). This cycle is shown in figure 2.4.

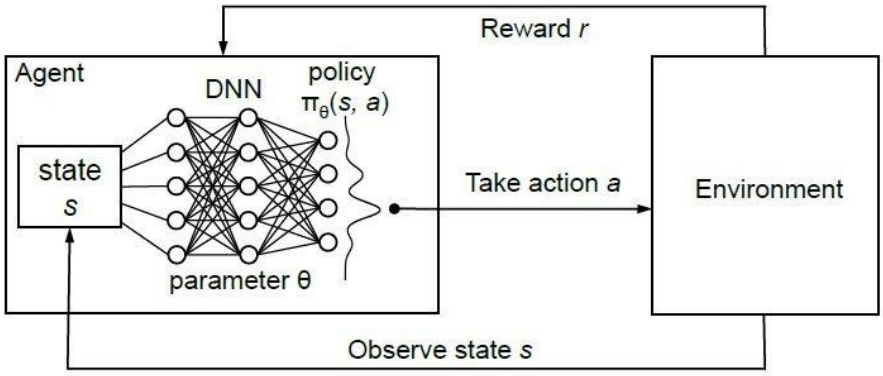


Figure 2.4 Reinforcement learning cycle (Koo, Mendiratta, Rahman, & Walid, 2019)

The agent resides in an environment of which it has only partial knowledge. The agent must therefore make uncertain future reward predictions. Therefore, the expectation of reward for the future is inversely proportional to the distance of the reward from the instantaneous time. The reward that can occur in each state and action pair is indicated by r and the reduction factor is indicated by γ.

$$Q^{\pi}(s,a) = E\lceil r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_t = s, a_t = a \rceil \tag{2.1}$$

Not every action taken by the agent will lead to the desired result. Therefore, the relevant value of the target policy (π) to be reached for each state and action pair is the maximum action values.

$$\pi^*(s) = arg\max_a Q(s,a) \tag{2.2}$$

For the agent to learn, it is not desirable in every situation to pursue the maximum benefit. With a certain probability ε, the agent must take a random action.

$$\pi^{\varepsilon}(s) \triangleq \begin{cases} argmax_a Q^{\pi}(s,a), & 1-\varepsilon \\ UniformRandom(A), & \varepsilon \end{cases} \tag{2.3}$$

The difference between reward values formed after the action and the target will reveal the loss function (L). (Mnih et al., 2015).

$$L = E\left[\left(r + \gamma \max_{a'} Q(s', a')\right) - Q(s', a')\right] \quad (2.4)$$

The sum of the loss functions is the cost function. Training the neural network is the process of minimizing the cost function. The process of finding the global minimum point is done with the gradient descent algorithm. A visualized version of the gradient descent algorithm can be viewed from figure 2.5.

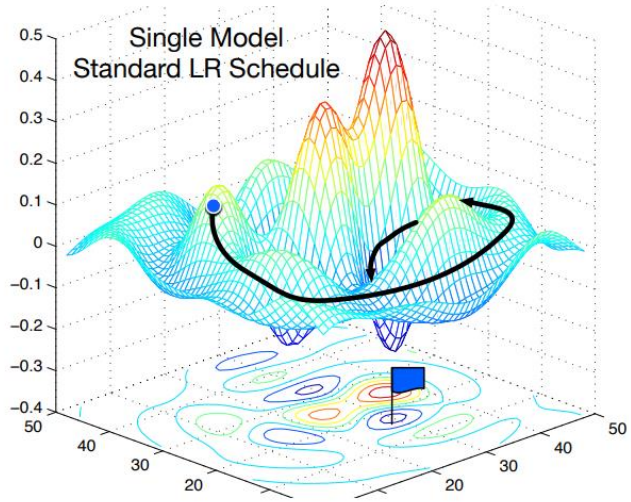


Figure 2.5 Gradient descent (Huang et al., 2017)

It is important to use an optimized learning rate when trying to find the minimum point. Low learning rate can lock learning to local minimum points. The large learning rate can ensure that the desired minimum point cannot be reached as shown in figure 2.6.

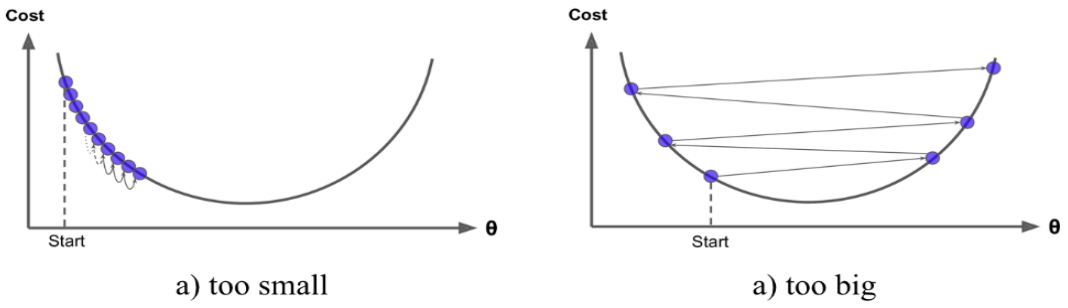


Figure 2.6 Learning rate (Géron, 2017)

2.4 Convolutional Neural Networks

For a machine learning agent to process the camera data, it is not right to feed these images directly as input to the neural network. To reach the meaningful sub-parts of the images, a structure that can reveal the relationships between the pixels is needed. Convolution process solves this need in image processing. Convolution is a method of extracting features from an image by multiplying the image matrix with various reformed filters. The example of the feature extracted from the image can be examined in figure 2.7.

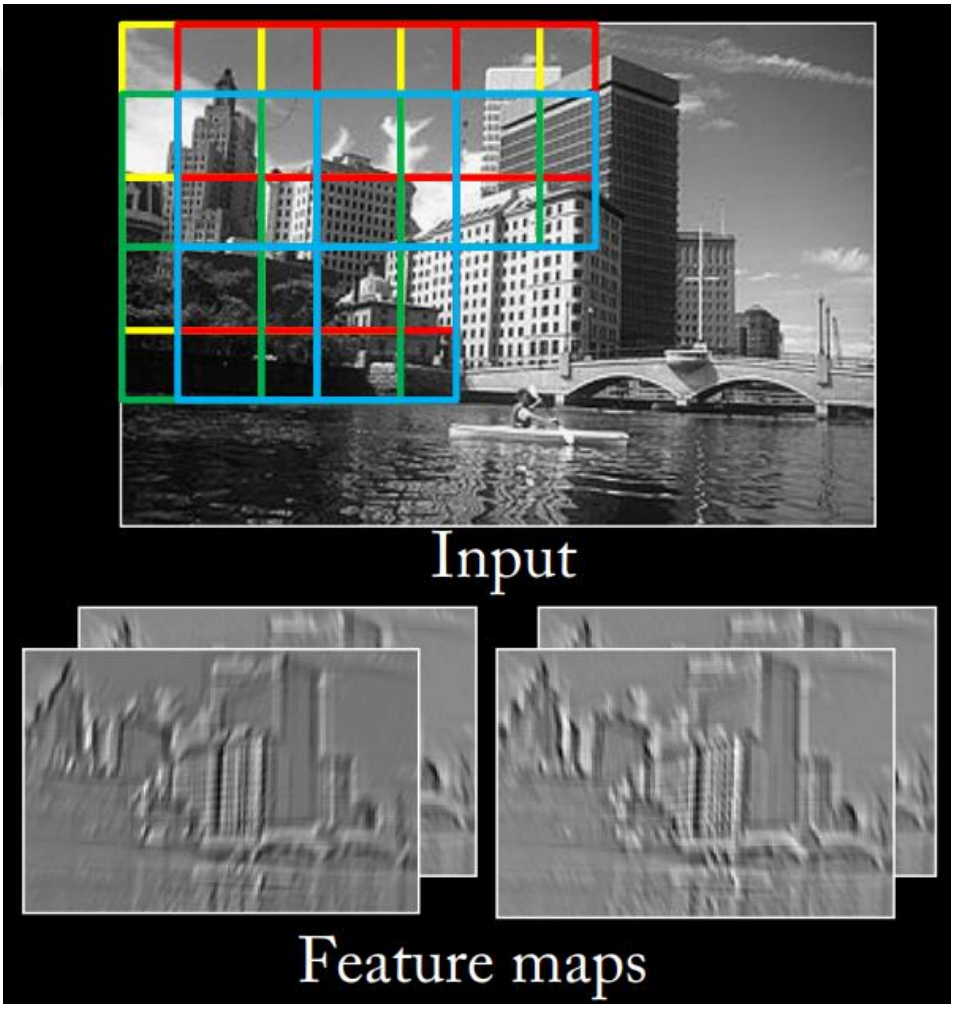


Figure 2.7 Extracting features from image class (Fergus, 2022)

Filters called kernels, navigate through the image, and multiply their corresponding blocks with the number of colors in their corresponding pixels as shown in figure 2.8. After all the elements on the kernel are multiplied by their visual pixels, the results are collected. The result is the new number of the pixel block where the kernel center is currently located (Yamashita, Nishio, Do, & Togashi, 2018).

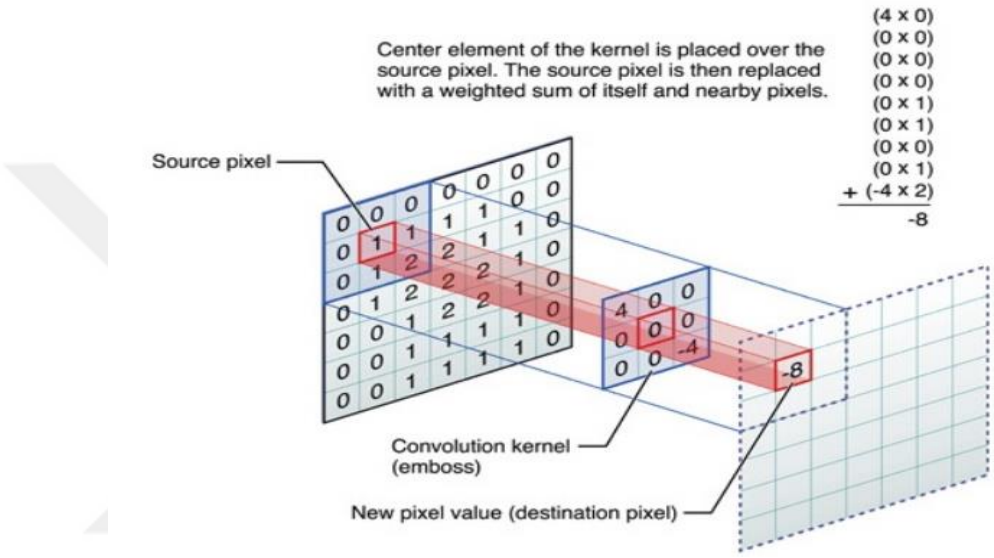


Figure 2.8 Convolution (Agapitos et al., 2015)

The resulting images that are smaller than the original are called maps. Each map is an image that has lost some data from the original. At the same time, each map is a version of the original image, highlighting various features. As a second operation, pooling is applied to reduce the size of the resulting maps and thus increase performance. It can be said that pooling is a structure that summarizes the important details on the map (Albawi, Bayat, Al-Azawi, & Ucan, 2018). Two different types of pooling, max pooling, and average pooling are shown in figure 2.9.

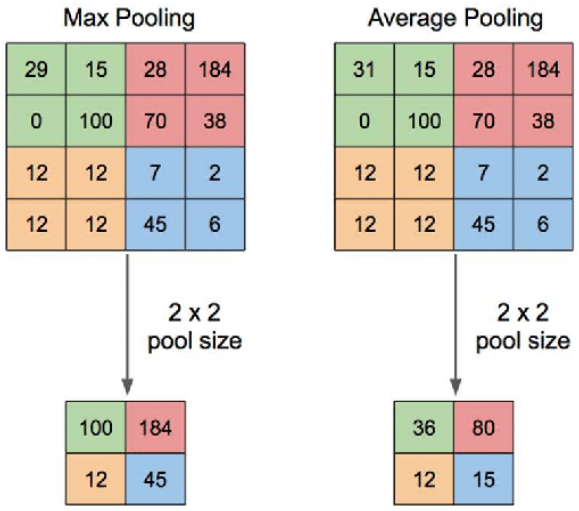


Figure 2.9 Pooling (Yani, Irawan, & Setiningsih, 2019)

The general convolutional neural network application as shown in figure 2.10 is as follows. The image to be processed is passed through many filters in the convolution process. It is then normalized with the activation function. After this stage, maps that are smaller in size than original are revealed. Although the size is reduced, the number of images to be processed increases as a result of this process. Before the maps are put into a new convolution, pooling is done. Thus, the size to be processed is reduced by keeping the number the same. These processes are continued until the desired depth. The result is flattened and fed as an input to the neural network (O'Shea, & Nash 2015).

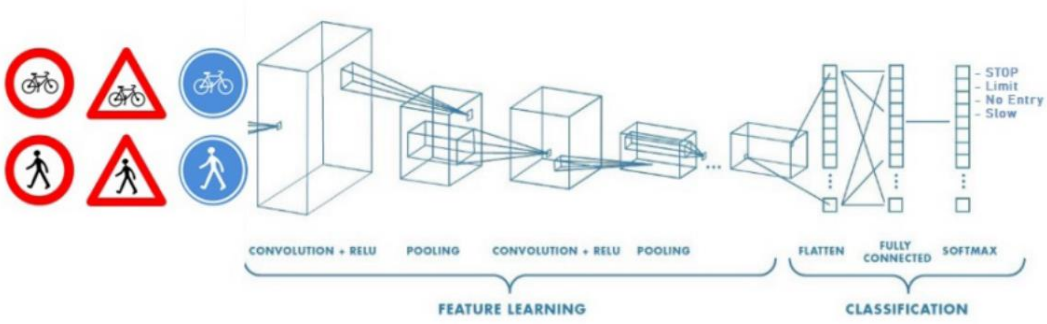


Figure 2.10 Convolution architecture (Narejo, Talpur, Memon, & Rahoo, 2020)

2.5 Unity ML Agents Package

Unity is a multi-development environment and game engine for developing games and simulations. Unity published the ML-Agents Toolkit project in 2020. ML-Agents Toolkit is an open-source project. It allows researchers and developers to develop machine learning applications in simulations they want to implement (Juliani et al., 2018).

In this study, default Unity hyperparameters are used in agent training. The Unity version used is 2021.3.5f1 and the ML Agents package version is 2.0.1. The Default ML-Agents Toolkit trainer is PPO (Proximal Policy Optimization) learning technique. PPO is a policy gradient type method that predicts the best reward action based on the agent's observations and optimizes the reward (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017).

CHAPTER 3
PROPOSED MODEL

The proposed model consists of five subsections. These are respectively; node matrix generation for the map, character creation, character machine learning training (optional), node training and pathfinding.

Since video games are programs developed for the end user, it is preferred that the systems they contain do not strain the computer resources too much and can be run by an average computer. Therefore, one of the main purposes of this study is to transfer the required workload to the development phase as much as possible and to leave a minimal workload for the product to work as a result. In addition, since each game will reveal different problems that need to be solved in terms of pathfinding, a structure that can be applied to different scenarios as much as possible has been tried to be revealed. The following paragraphs will summarize how these factors were evaluated at each stage of the study.

To narrow the scope of the study, this study was developed considering platformer, side-scroller style games, since it is the genre with the most different movement mechanics among the game types and there are not many pathfinding applications produced for this type of games in the sector. Adapting the work to other two-dimensional game genres or even three-dimensional game genres by changing some factors and calculations may be the subject of further study.

The first of the determined subsections is to create the node matrix for the map. At this stage, the nodes that will record the motion data are created based on the map and their basic parameters are assigned. While developing this section, attention has been paid to working with objects that are close to the content of an average platformer game. Secondly, a versatile structure that can be used in differently designed maps has been tried to be put forward. The processes of creating maps and generating nodes within the map have been made applicable in every scenario as much as possible.

A new platformer game emerges every day and introduces players to new character mechanics. New mechanics can be on combat mechanics, as well as the trademark of the genre, on character mobility. This mobility features makes pathfinding quite difficult in platformer games. Therefore, in the character creation section, which is the second subsection, more than one character movement type has been tried to be considered in a way that will be suitable for computer-controlled characters that can move in different ways in the game. Initially, a character that makes a wall slide and a wall jump and a character that does a double jump are created.

Node training and character machine learning training stages can be done separately or together. For this study data, they have been determined to work separately. The purpose of this section is to enable the trained character to produce information by navigating the map and to ensure that this information can be used in the final product without the need for artificial neural networks. Neural networks are computationally expensive, and a game can have many computer-controlled characters on the scene at the same time. Therefore, controlling each of these characters by neural network will create a serious computational load. Artificial neural networks will not be used after this part to obtain a program that is lighter in terms of computer resources.

The pathfinding phase is where all this information is used. It is the stage where the shortest path from one desired point to the other on the map is found and the character is directed. A counterpart character that is not connected to the artificial neural network is produced for each character, and when this stage is started, the machine learning agent is deleted from the scene and replaced by its counterpart. Using previously generated knowledge, this character can navigate the map by following the footsteps of the machine learning agent. In this way, creating a system that is much faster and uses much less computer resources is aimed. As a second case, a system that will eliminate nodes and reduce the calculations they make in each physics cycle across the entire map is being considered, work continues in this issue.

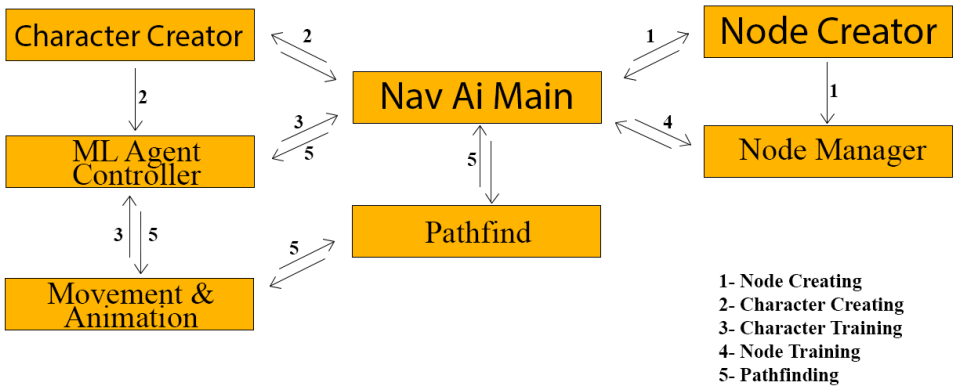


Figure 3.1 General architecture of namespaces

The general namespace dependence map can be seen in figure 3.1. The NavAiMain namespace, which is the main manager, in the center undertakes the task of controlling all stages in general and transferring information between stages. Main manager and NodeCreator namespaces work together in the first stage, creating the node matrix. Main manager and CharacterCreator namespaces undertake the character creation stage together. During the character neural network training phase, with the information coming from the CharacterCreator namespace, the MLAgentController, main manager and Movement&Animation namespaces work together. Node training takes place because of the collaboration of NodeManager and main manager namespaces. Ultimately, pathfinding is possible thanks to the collaboration of main manager, MLAgentController, Pathfind and Movement&Animation namespaces.

3.1 Generating Node Matrix

While creating nodes, it is aimed to be created depending on the map geometry. In this way, it is aimed to create a versatile system that can work on different maps or easily adapt to the changes made on the map. For this purpose, first, create a lattice structure that limits the area where the nodes will be placed on the map is done. Then, the final cage point object was created to determine the boundaries of the usable space.

Each final cage point object holds the border object it is included in with the name path. It also recognizes final cage point objects that come before and after it on this border path. In this way, all elements of the border object and the relations between them can be seen visually as shown in figure 3.2.

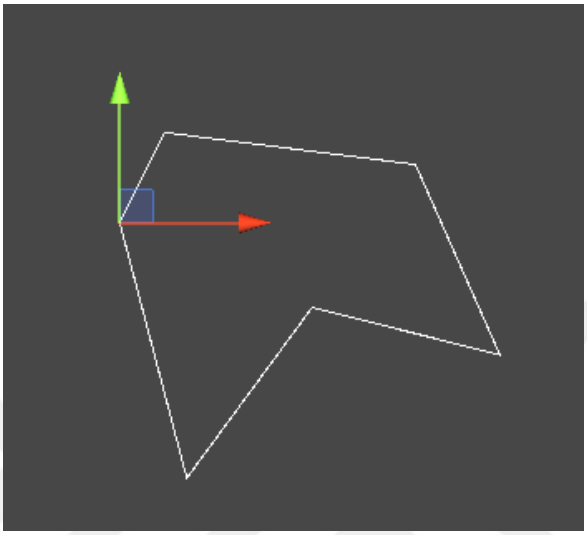


Figure 3.2 Cage point path generated by final cage points

Cage point path objects were created to be used in maps where more than one path is required. Each cage point path maintains a list of all final cage points it owns and a bool of whether they are created automatically or not.

To reduce the number of nodes and to obtain more detailed information about which nodes are at the border, it is aimed to determine the borders of the map and all objects with hitboxes. For this purpose, the map borders are drawn from the geometries formed by the tiles in the ground layer in Unity. And some boundaries can also be added manually so that characters are not wanted to pass through cannot navigate and can limit areas that do not have a hitbox or close open spots on the map. Final cage point and cage point path creation processes are done through the main manager class editor window. Editor view of main manager class is as in figure 3.3.

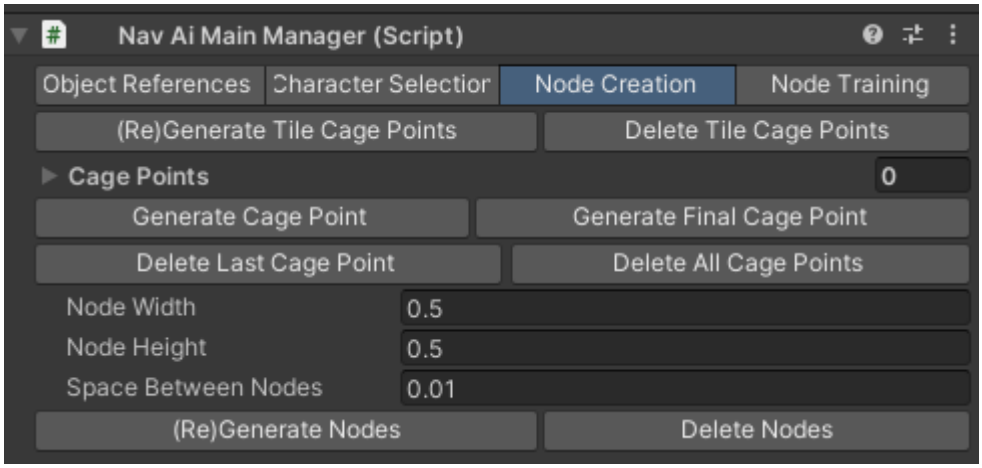


Figure 3.3 Node Creation tab on Nav Ai Main Manager

As the first step of creating a cage point, it is planned to extract information from the map and place the final cage point object at the required points and establish the relationships between them. (Re)generate tile cage points and delete tile cage points buttons are used for this operation. The (re)generate tile cage points button reads the data from the created map and deletes the previously created final cage point objects, if any, and creates new ones. The process of creating final cage points from tiles can be viewed in figure 3.4, and the images remaining after the tiles are deleted can be viewed in figure 3.5.

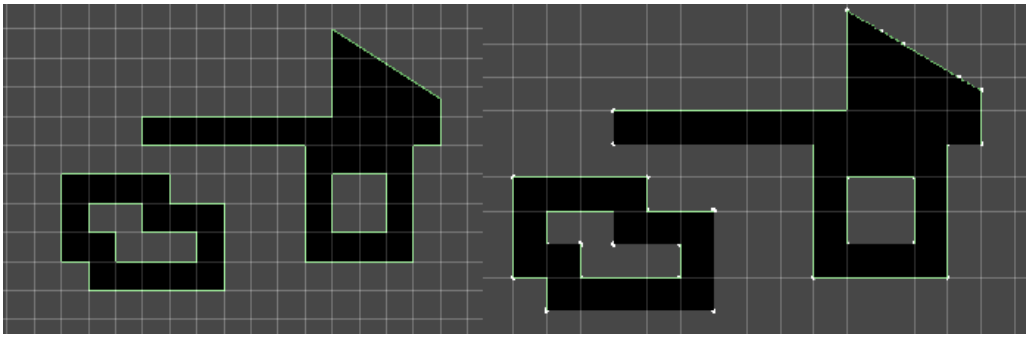


Figure 3.4 Before and after the generation of final cage points

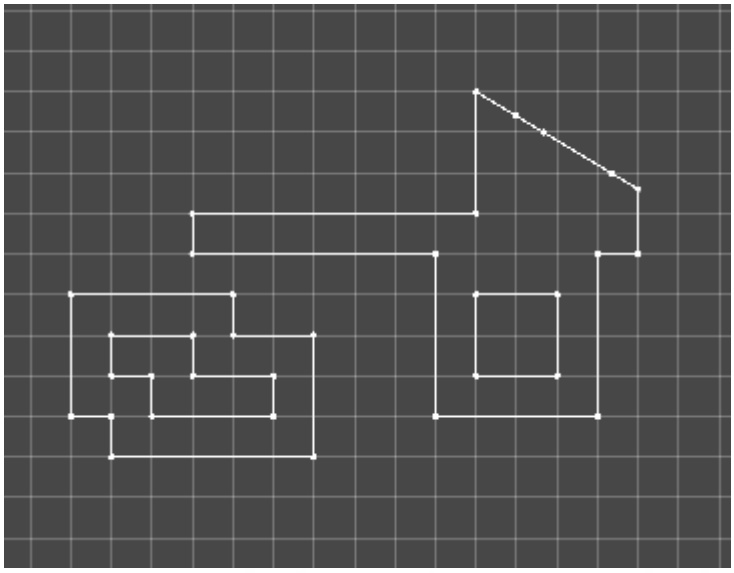


Figure 3.5 Generated final cage points

Unity's tile map feature was used when extracting data from the map. First, right triangles and rectangles with different proportions were created to create different geometries. Physics boundaries were determined on the created images and saved in Unity as a tile object as shown in figure 3.6.



Figure 3.6 Two to one right-angled triangle object tile set

Objects are produced in 256 * 256-pixel dimensions so that they can be used in a 2D platformer game. A tile palette has been created to paint with all objects as shown in figure 3.7.

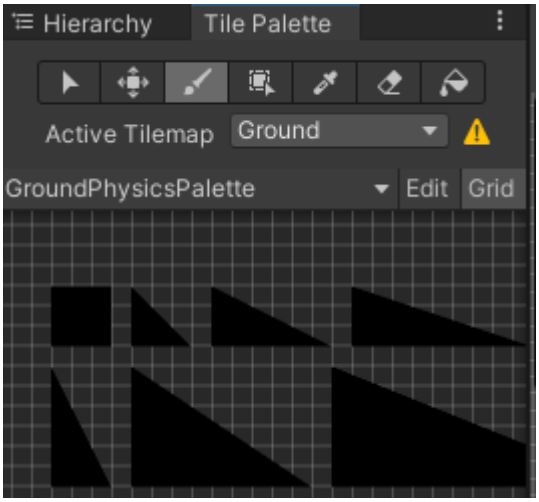


Figure 3.7 Created tile palette

Painted tiles are located under the ground object in the scene hierarchy and their layers are marked as ground. Editor view of ground object is as in figure 3.8.

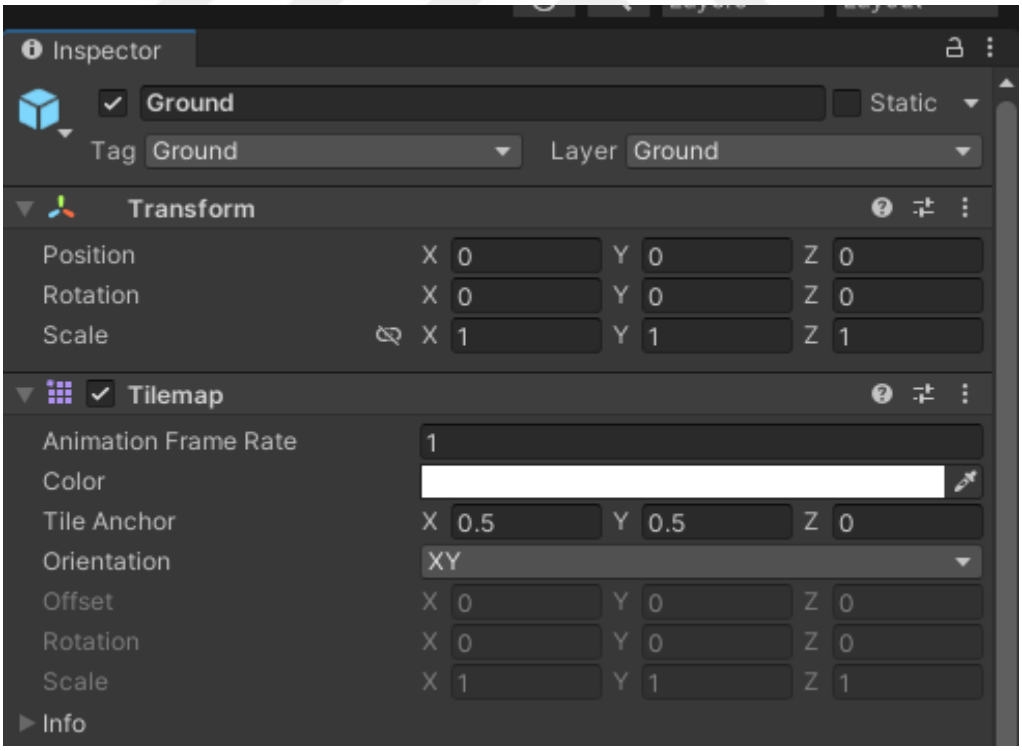


Figure 3.8 Ground tile map object

In Unity, tile map collider component is used as collider in tile map objects. This component creates a separate collider for each tile. Because this, it creates collider

borders in some unnecessary regions. As a solution, Unity has the composite collider component by default as shown in figure 3.9. By using the two components together, the map borders can be easily determined.

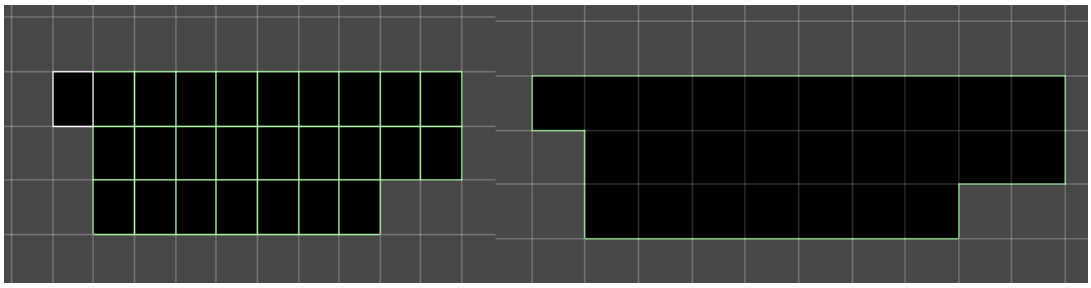


Figure 3.9 Before and after the composite collider

The composite collider component contains a method that can export the boundaries of the object it creates. The locations of final cage points are determined by receiving this data.

After placing the final cage point objects to be placed with the data from the map ground layer, extra final cage points can be added manually to cover the gap areas and areas of the map that are not wanted to be accessed, if any. For this, the generate cage point, generate final cage point, delete last cage point, delete all cage points buttons in figure 3.3 are used. The generate cage point button creates a new final cage point object and connects it to the previous object, if any. The generate final cage point button also creates a new final cage point object, but this time both connects it to the previous object and connects it to the point at the top of the list. Thus, a closed shape is obtained. The closed shape example obtained by this method can be examined in figure 3.2.

After the cage point paths are created, the process of creating the node objects can be started. For this, the (re)generate nodes button in figure 3.3 can be pressed. The method that works as a result of pressing this key will first check whether all final cage points produced have created a closed shape. If all cage point paths are created correctly, it will navigate through all points, revealing the maximum and minimum points to be used in scene coordinates. Until now, cage point paths have been kept only as a set of points, but to determine the coordinates of the nodes, the line segments

formed by these points must be determined. Thus, it is aimed to prevent creating a node at the coordinate with another object. For this purpose, line segments are created and temporarily saved over all final cage points and their network of relations. Finally, the nodes are placed by scanning between the determined maximum and minimum points.

All line segments are kept with start and end points, gradient and y-intercept values. In addition, line type is recorded as horizontal, vertical, or oblique for ease of operation.

Nodes are placed by scanning from the determined minimum x and y coordinates. When scanning, node placement starts or ends at each line segment transition. In this way, no nodes will be placed in the closed zone areas, but only in the playable part. Scanning is done by skipping a node size at each step, considering the node width, node height and space between nodes values. The node creation process can be observed from figure 3.10.

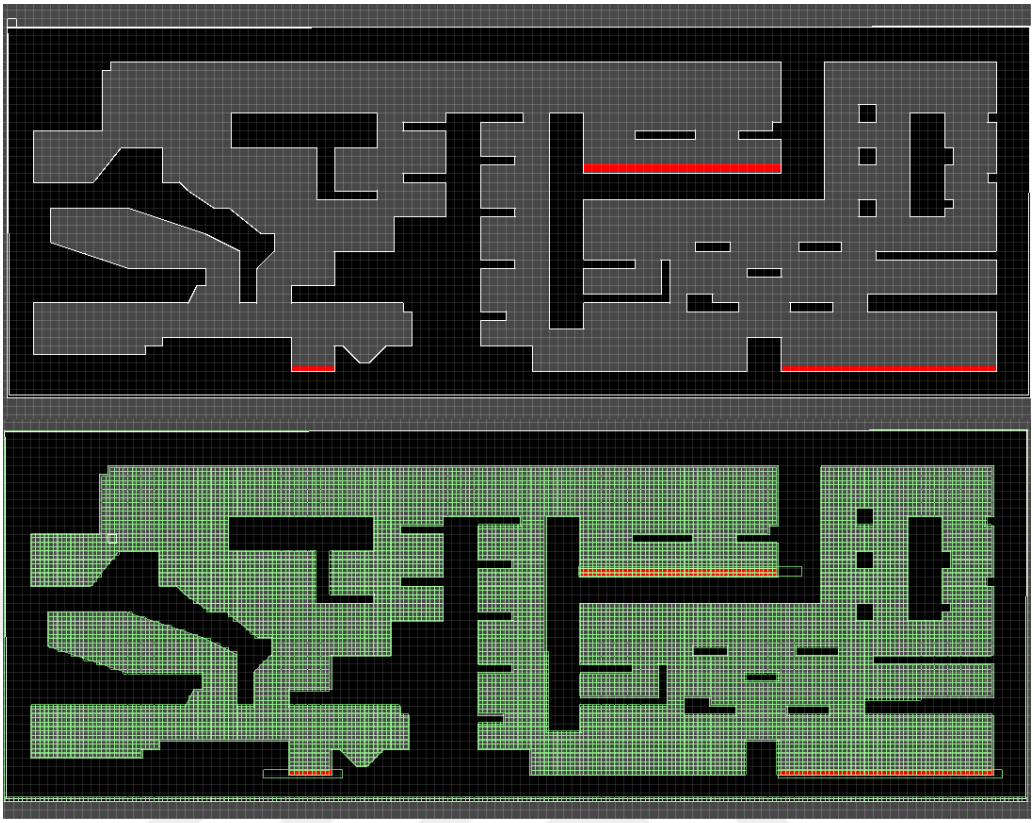


Figure 3.10 Before and after generation of nodes

While the nodes are created, their positions on the matrix and their distance to the nearest boundary node are recorded. The distance to the boundary node value will be used in further calculations so that the hitboxes of the characters do not go beyond the desired area.

3.2 Creating Character

One of the aims of this study is to create a model in which different characters with different movement mechanics can be trained. The list of selectable character traits can be viewed from the character selection tab of the main manager class, as can be seen in figure 3.11. Work is still in progress for most of the character types. Initially, it was worked with characters with wall-jump, wall-slide and double-jump mechanics, which are common in platformer games. In addition, final character versions that are not connected to any artificial neural network, which will be controlled by pathfinding after the artificial neural network is eliminated, were also created.

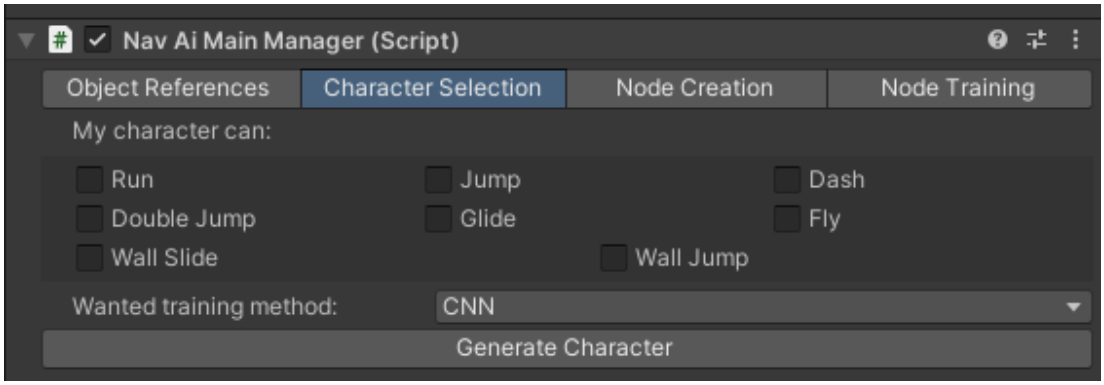


Figure 3.11 Character selection tab on main manager class

After the desired character traits are specified, when the generate character button is clicked, the character in the scene, if any, is deleted and the desired character is added to the scene. Only one character can be trained at a time in each environment. In order to train more than one character over more than one environment at the same time to train a single character faster, all calculations were made over local coordinates.

Almost all objects of characters are used in common to reduce data usage. However, some objects that will enable to distinguish the character have been created specific to the character. In short, all content is the same, except for the state machine classes that determine which mechanics the characters use during movement. Figure 3.12 can be viewed for a summary of all the objects used on the characters.

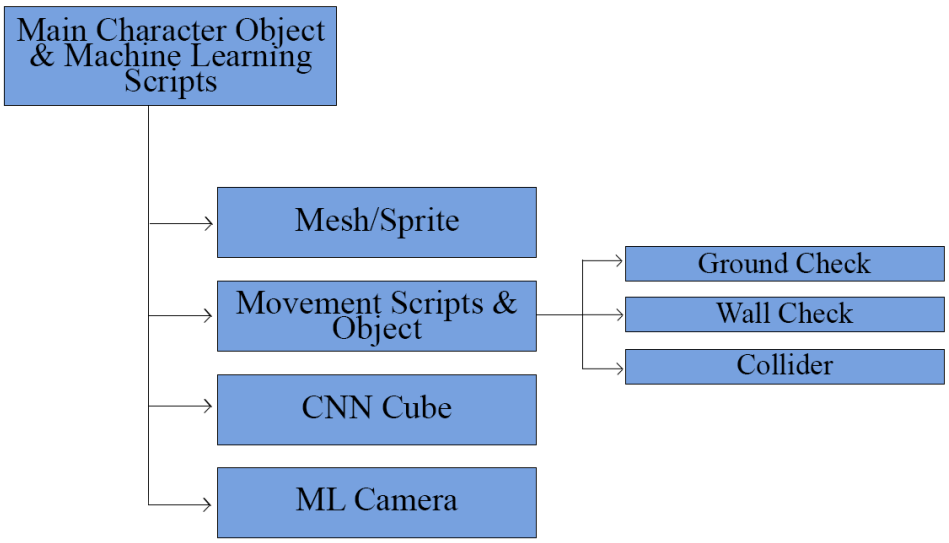


Figure 3.12 Character object hierarchy

The character main object is used to connect with the main manager class and to have machine learning scripts according to the character's needs. In addition, the node the character is on is tracked from the main object. In each physics cycle, the circular object in the character center shown in figure 3.13 controls which node it is on.

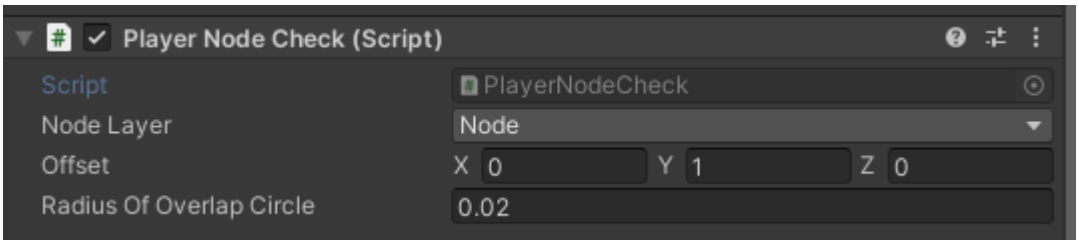


Figure 3.13 Player node check object

Scripts connected to the main object of the character vary according to the character type. Unity machine learning package includes camera sensor component for characters that will perform convolution operation.

The mesh object that can be seen in figure 3.12 contains the three-dimensional image of the character. It creates character animations and visuals. Mesh object can be converted to two-dimensional sprite object upon desire. Madea, character from mixamo.com, is used only visually to better explain the program (Arrebola, 2022). The locomotion animation package from the same source is used for the movement animations of the character (Mixamo, 2022). The program does not need a character image to work. It can be run using any sprite, a mesh, or neither.

Movement object is the basic object that determines the state of the character according to the input values from the artificial neural network or the keyboard and creates the movement. Which states the character can enter are limited by the movement mechanics it can do. For example, a character with double jump, run, idle mechanics has its state value in one of the four states idle, run, fall, jump, depending on its position and movement in the environment. To do all this, the movement object should collect information about whether the character is touching the ground or any wall and should have information about the speed and direction of the character. The object information that the character is in contact with is collected from the child objects of the movement object, the decisions made by the artificial neural network are collected from the main character object, if used, the inputs are collected from the Unity input system.

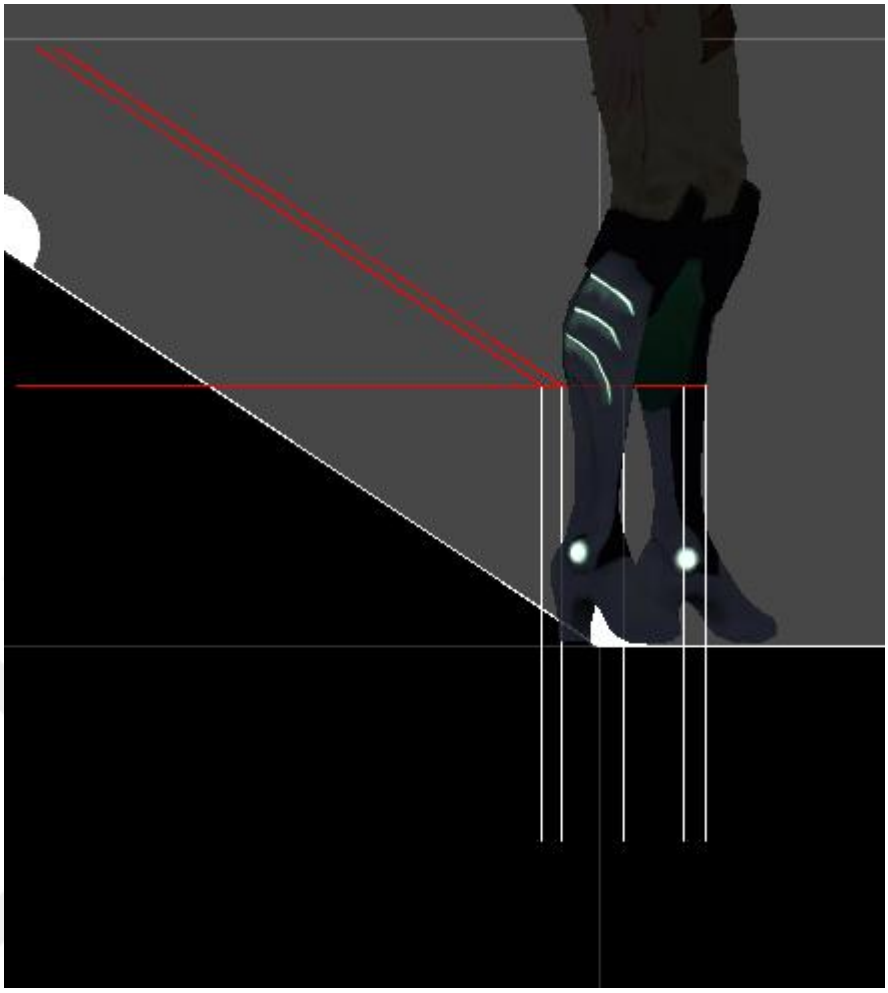


Figure 3.14 Ground check rays and ground direction rays

The character receives the information whether it touches the ground and in which direction the motion vector should be applied while moving, through the rays applied in the y-direction over the ground check object. As can be seen in figure 3.14, when the white colored rays coming out of the character touch the ground, they make the ground contact value true. In addition, for all rays, the normal of the objects they contact are calculated by rotating 90 degrees and shown in red. The reason for applying five beams, two on the right, two on the left, and one in the middle, is to predict the angular change of the contact area in wide and narrow characters. Depending on the direction the character goes, it is calculated with which vector the character should accelerate over the angular change between the normal of the object in contact with the first ray in the relevant direction and the second ray.



Figure 3.15 Wall check collider

As can be seen from the green box in figure 3.15, the character's wall control is provided by a collider. There are two wall check trigger colliders in total on the front and back of the character.



Figure 3.16 Character collider

The basic collider, which enables the character to make physical calculations and interact with other objects, can be seen as the green box around the character in figure 3.16.

Character movements are created with state machine and movement scripts according to the type of character. The motion vector changes depending on what state the character is in and how it wants to move. For example, for a character to enter the wall-slide state and do wall slide, it must be moved in the direction of the wall while in the air and leaning against the wall. In this state, the force to be applied to the character is calculated and applied as one third of the gravity. Rate constants, can be

seen from the figure 3.17, have been created for all characters to control which speed values they can reach under different conditions.

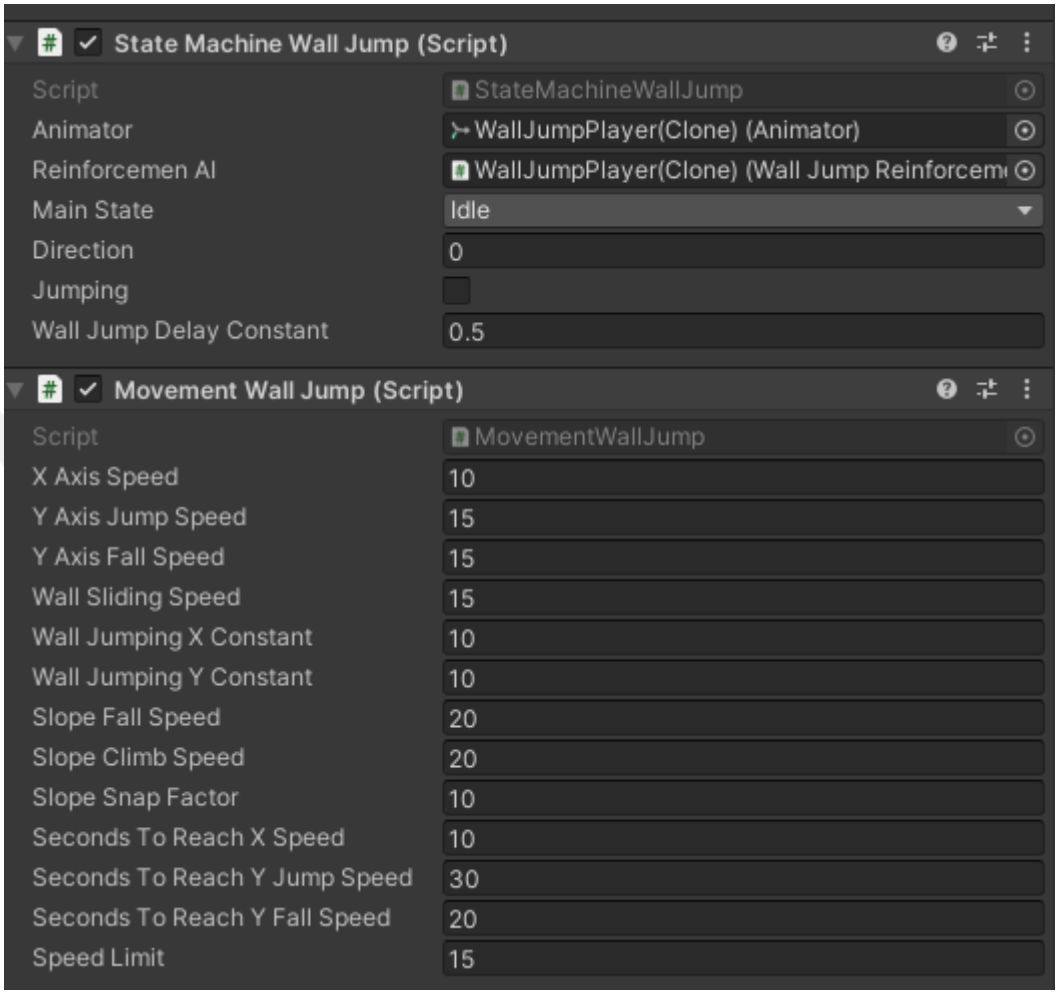


Figure 3.17 Character state machine and movement scripts

After determining in which direction and how fast the character wants to go, the relative vector is calculated according to the instantaneous speed of the character and aimed vector. And the amount of acceleration required for the character to reach the aimed speed is calculated according to Newton's second law. And the seconds to reach constant has been added to the calculations so that the character's acceleration time is reduced to a minimum and the desired speed is reached almost instantly.

In each physics cycle, the state that should be at that moment is determined by the speed and position of the character. According to the determined state, the forces to be applied to the character or other states that it can pass through are revealed.

Convolutional neural network cube, machine learning camera objects that can be seen in figure 3.12 are the objects that connect the agent with the artificial neural network. These objects are discussed in detail in section 3.3.

3.3 Machine Learning Agent Training

Agent training has been done with convolutional reinforcement learning method with receiving input from the in-game camera and convolutional neural network cube which can be seen in figure 3.12. The machine learning camera object is used to create images to be input to the neural network. It moves with the agent and records the agent and the objects around it and sends it to the neural network. Convolutional neural network cube, on the other hand, is a rectangular object that will be displayed to the machine learning camera instead of the character in the character position to reduce the processing size. Since convolution is a costly operation in terms of computer resources and the image of the character is not important to the agent, a white rectangle is used to show only the location of the agent. There are five different colors the agent camera will see; white for the agent itself, black for the ground, default Unity blue for the background that the agent cannot interact with, red for traps, yellow for the target. In addition, to use computer resources more efficiently, the resolution of the agent camera was determined as 160x90 pixels. The difference between the game camera and the visuals seen by the agent can be seen in figure 3.18.

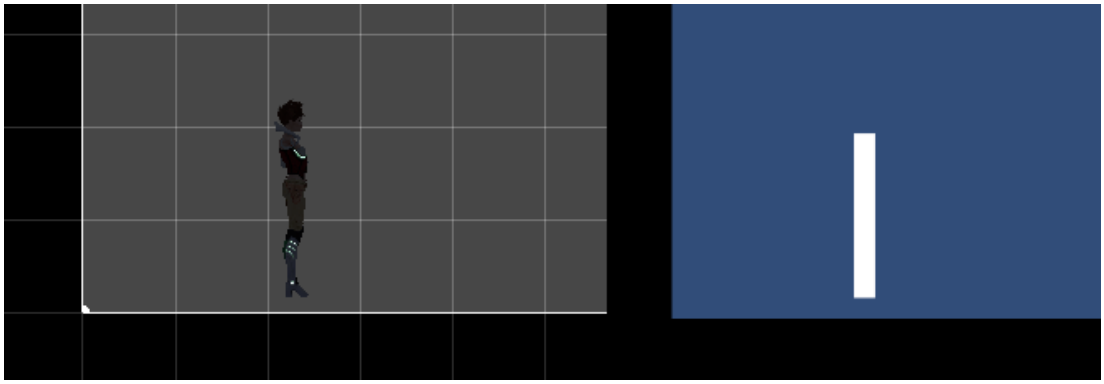


Figure 3.18 Scene view vs. what agent camera sees

All agents also receive input from a vector object between the agent and the target origin locations as an array.

Default parameters specified in Unity machine learning agents' documentation are used as hyperparameter. Default hyperparameters can be seen from figure 3.19. Similarly, default values are used for convolution parameters.

```yaml
behaviors:
  3DBall:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 500000
    time_horizon: 1000
    summary_freq: 12000
```

Figure 3.19 Default Unity hyperparameters (Unity, 2022)

The reward object is placed at a location 5 to 25 nodes away from the agent in each episode. It is ensured that distance of this randomly chosen location is at a distance where the agent can be trained, but close enough to find the reward in a limited time. At the beginning of each episode, the location selection is made at random positions, and it is not checked whether it is impossible to reach the relevant location in a short time. This scenario has been neglected as it is unlikely to occur. An example of an episode that is impossible to reach in a short time can be examined in figure 3.20.

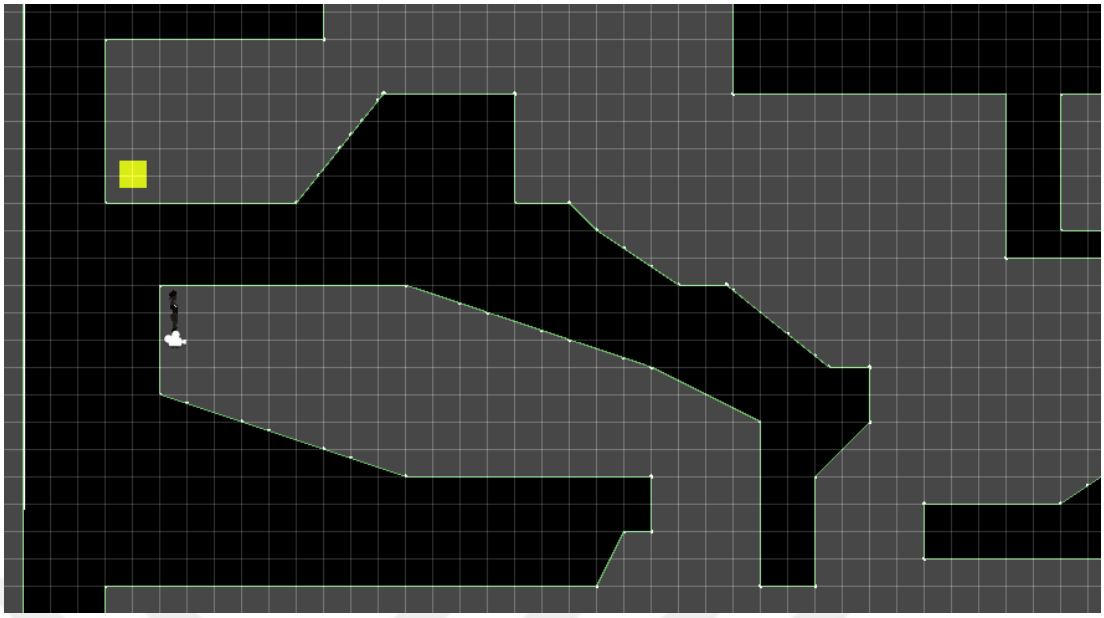


Figure 3.20 An agent target placement that is unlikely to be successful in a short time

There are three different ways an agent can get rewards throughout the tutorials. If the agent reaches the target object, it gets +100 points rewarded. If the agent cannot find the reward within 15 seconds or is caught in any trap, it will receive a reward of -20 points.

Two separate action variables have been determined for the agent action as movement and jump. In each physics cycle, the neural network determines the values of these variables and moves the agent. If the agent does not use a neural network, the output values are connected to the same keyboard keys in all agents. The first action variable is the running variable, it can take the values -1, 0 and 1. The value of -1 can be reached by pressing “a” key on the keyboard, 1 by pressing the “d” key, and 0 by not pressing any key, in cases where the neural network is not used. If this variable is 0, the agent will stay stationary, if it is -1, it will accelerate in the -x direction and if it is 1, it will accelerate in the +x direction. The second variable is the jump variable, it can take values 0 and 1. For agents without a neural network, it will take the value 1 when the “space” key is pressed, and 0 when not pressed. A value of 1 order the agent to jump. Motion action output variables and keyboard shortcuts are the same for both agent types. Agent state machines are responsible to create different movement mechanics with agents with the same inputs.

3.4 Node Training

Node training is the process to reveal the relationships and paths between nodes for the relevant agent. Since the pathfinding method suggested in this study is not through the coordinates and neighborhood relations of the nodes, but through the links established with the motion mechanics, it is necessary to train the nodes and reveal the paths by using these motion mechanics. In addition, since the movement mechanics will change from character to character, it is more reasonable to store the revealed paths on the character rather than on the environment. In this study, this topic has been left out of the scope as it is aimed to show that pathfinding is possible with the proposed method with different characters.

Although different paths can be created for each character, the fact that the environment and nodes are constant for the characters will ensure more efficient use of computer resources. In this respect, the node training of all the characters created is designed to be in the same environment with the same nodes.

The first determined objects when node training is done are stable nodes. Stable nodes will be grouped later in the study and will be used to reduce the number of nodes to be searched while pathfinding. At the same time, stable nodes are the basis for finding valid paths. These nodes can be defined as where the character can remain stationary for an indefinite period while in the idle state without the need to hold itself in the same position with any force. Identifying these nodes for each character can be done in different ways. In this study, the stable nodes for the wall jump character and the double jump character are determined on the nodes where the character touches the ground. For characters that have not yet been added, this may need to be handled differently. For example, for a character that can hang on the walls, the node that it hangs on the wall or for a character that can fly indefinitely, all nodes on the map can be determined as stable nodes.

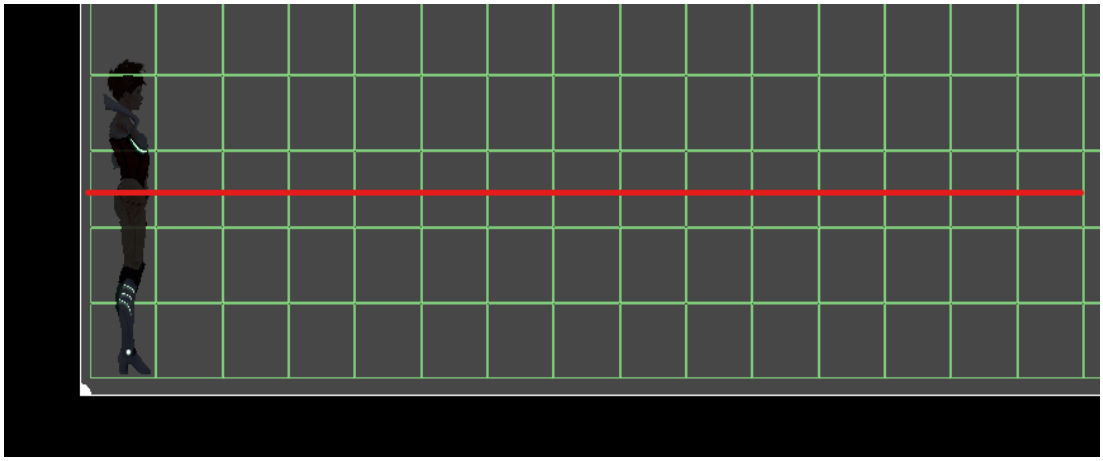


Figure 3.21 Character origin and stable nodes

Since the nodes will be trained while the character is navigating the map, the nodes where the character's node detection center comes across while the character is stationary or walking and touching the ground are marked as stable nodes. It should be noted that these node’s positions may vary for characters with different sizes or mechanics. In figure 3.21, nodes in line with the red line are stable. In this study, determination of stable nodes was done simultaneously with node training. Determining stable nodes and their groups by character before node training may speed up the process quite a bit and may be the subject of a further study.

As stable nodes are determined, paths between nodes can be determined. In this study, for a path to be registered as a path, it is stipulated that it starts from a stable node and ends in a stable node. Thus, the paths taken by the agent, who may enter an unused space, various traps, or an undesirable place, are prevented from being recorded as a path.

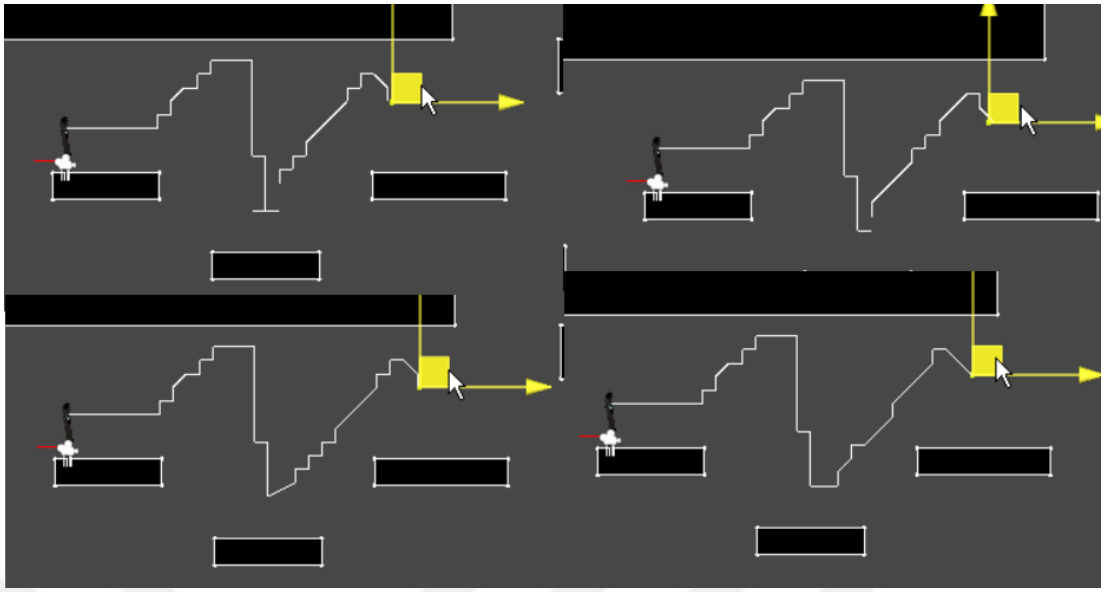


Figure 3.22 Different paths for different ending nodes

As a second condition, only the shorter of the two paths with the same starting and ending stable nodes will be recorded. In this way, it is desired to prevent unnatural movements that the agent may do during or after neural network training. Saving fewer paths will also be positive in terms of performance. The main disadvantage of the second requirement is that it makes difficult to register some unstable nodes. However, given enough time, the agent can process most of the nodes on the map on which it can be physically. For this, constantly manipulating the position of the target object and directing the agent to the less processed nodes can speed up the process. However, since the main purpose of this study is to establish a node network and pathfinding among stable nodes, it is not necessary to train all unstable nodes. Therefore, this study is not programmed to select untrained nodes by manipulating the position of the target object. A structure that can guarantee the training of all nodes may be the subject of further study. Different paths for different unstable nodes can be examined in figure 3.22.

Every time the character changes a node, the node change is detected, and it is recorded which mechanic is used when going from the previous node to the next. Movement connection is recorded as starting node, movement direction, ending node. Paths are recorded sequentially from the beginning to the ending node.

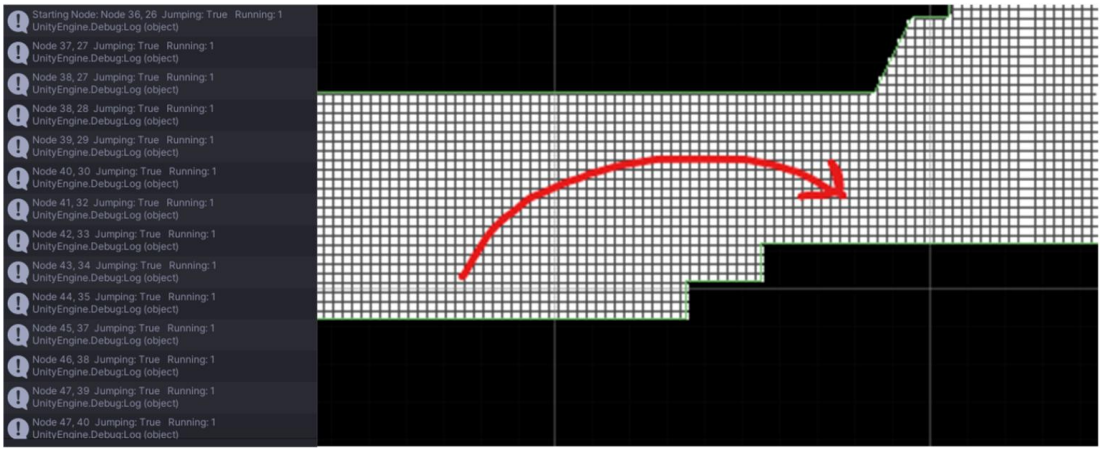


Figure 3.23 Node link

Paths that start with a stable node and end with a stable node and are the shortest known path are saved as node link objects. The node link object, can be seen from figure 3.23, contains the path nodes, including the starting node and the ending node, and the necessary movements for the path. The starting node object maintains a list of all paths originating from it. In this way, the record of which node can be reached from which node with which path is kept throughout the entire map.

3.5 Pathfinding

Pathfinding using only nodes is an expensive system in terms of computer resources. Therefore, in today's systems, coordinates are grouped with geometries called mesh, and pathfinding is done primarily between meshes and then within the mesh. In this study, this system will be imitated.

In the study, a structure has been created that character will travel from the starting node to a stable node and from there to another stable node closest to the target node and finally to the target node. The reason for pathfinding over stable nodes is that the character movement is aimed to be as targeted as possible and to look natural. Therefore, it is aimed that the character will display an image that will go directly to the target along the shortest path. The path revealed by the program over character and target can be examined in figure 3.24.

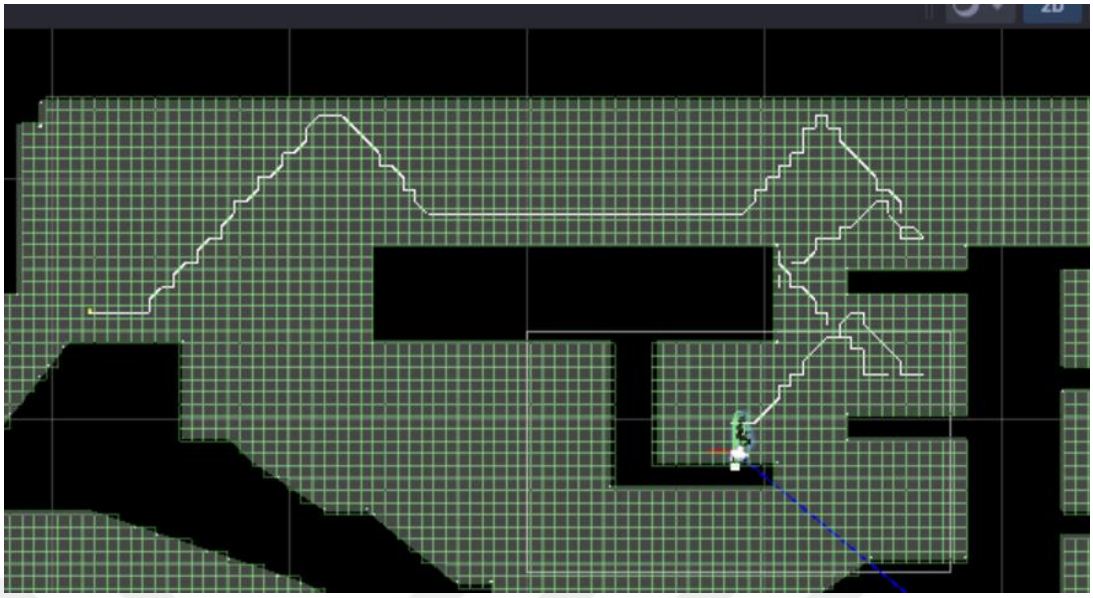


Figure 3.24 Path to target

Before starting the pathfinding stage, stable nodes are grouped as stable node group objects to increase the computational performance of pathfinding. To determine the stable nodes that will be in the same group object, it is stipulated to be able to move from one stable node to another without passing through another node and without jumping the character, and to return from the second node to the first in the same way. The content of this requirement may vary for characters with different movement mechanics. In this study, this condition is considered sufficient for the characters that can wall jump and double jump. In this way, when performing pathfinding in a stable node group, it is made in such a way that it is known that the agent can move freely and path can be calculated only with movement mechanics.

The stable node group object is derived from Unity's data container class, the scriptable object class. Scriptable objects are one of the Unity base classes that provide memory performance by preventing the same data from being loaded more than once when using the same data in the scene.

In this study, in order to reduce the number of nodes searched at once, pathfinding is done gradually, first, if necessary, between groups, then within groups. If the target node is not in a group and there are links it is in, firstly the links that this node reaching

a group with the least number of nodes are determined and stored. After the all calculations done and path between groups is revealed, this stored data can be added to the path and the final path can be found. In this case, the group that are the target of the stored link become target group.

If the starting and target groups are the same group, only in-group pathfinding is done. While calculating the path between stable nodes in the same group, each node is scored according to its distance from the starting and target nodes. This score, which will be used for the heuristic function of greedy search algorithm, is calculated using the node matrix coordinates of the target (x_t, y_t) and starting node (x_s, y_s) and the node matrix coordinates of the node whose score is calculated (x_n, y_n).

$$f(n) = (x_s - x_n)^2 + (y_s - y_n)^2 + (x_t - x_n)^2 + (y_t - y_n)^2 \quad (3.1)$$

Two clusters of nodes are maintained, named selected nodes and potential nodes. Initially, the only element of the selected nodes set is the starting node. The potential nodes cluster's elements are determined as starting node's neighbors that can be traversed by links. At each stage, the node with the lowest score from the set of potential nodes is selected by the greedy algorithm and transferred to the selected set of nodes. Linked neighbors of the newly selected node will be added to the set of potential nodes if they are not in the set of selected nodes. This process will be repeated until the target node is found.

If the starting group and the target group are not the same, pathfinding is made gradually first between groups and then within groups. Pathfinding between groups is almost the same as in-group pathfinding. Groups are used instead of nodes, and node group links are used instead of links. Node group links are no different from node links. While creating node groups after node training, node links that start from one group and switch to another group are pointed by node groups as a list of node group links. Thereby, while searching for a way, only groups and links that can be switched between groups can be focused on. The node group link object is derived from the scriptable object class. Each one keeps the data of the node group it starts and targets,

which nodes it contains within these groups, and the link between these nodes. Finally, the group score for each group is the score of the node with the lowest score in the group. Greedy algorithm will only consider this score when searching between groups.

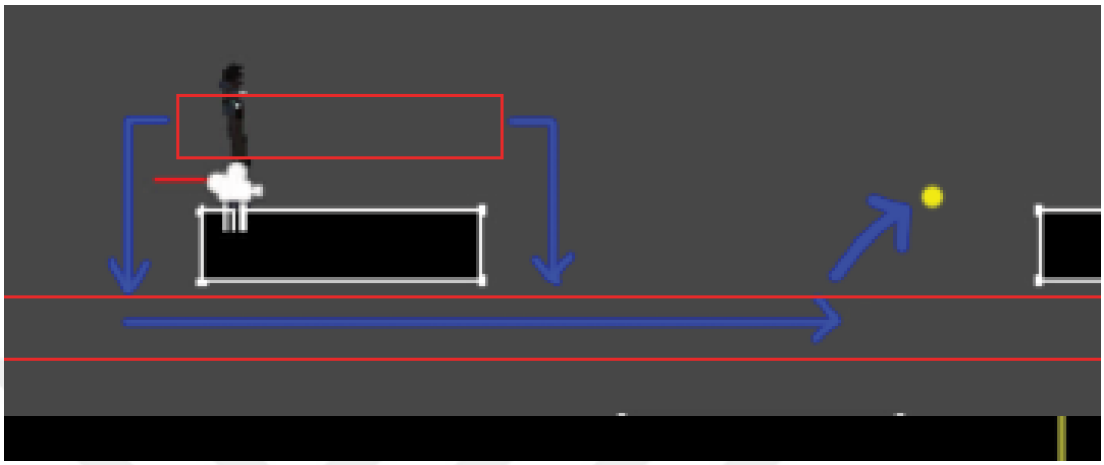


Figure 3.25 Possible paths between red node groups

As can be seen in figure 3.25, more than one option may be possible when finding a path between two different node groups. Since the distance of each option will be different from each other, the actual route to be traveled was not used in the calculation of the route. This is why an estimated value is used for the route taken in greedy algorithm.

After determining which groups to pass through to reach the target, in-group pathfinding is done for each group. The process is carried out by moving from the starting group to the last group, respectively. For each group, pathfinding is made between the first node reached in the group and the starting node of the link that goes to the next group. The length of the links between groups is not included in the pathfinding calculation. When the target node is found, all nodes navigated backwards from the target are recorded as path. The resulting path will be a combination of paths for the group and paths between the groups. An example of this revealed path can be seen in more detail in figure 3.26.

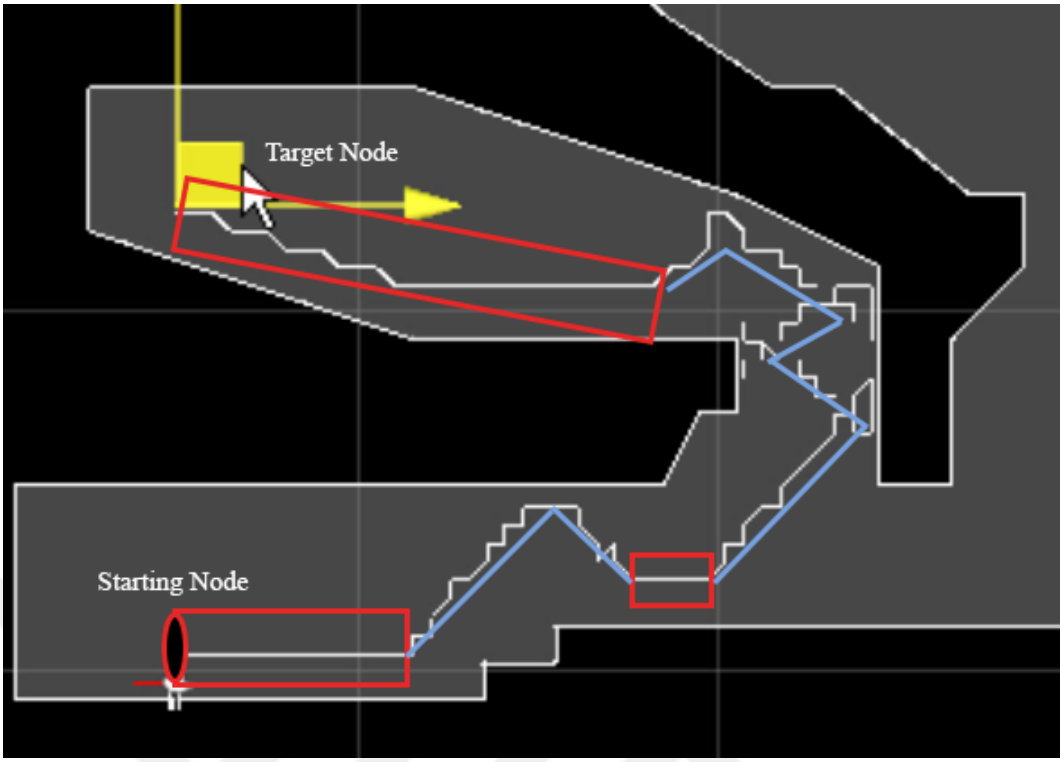


Figure 3.26 Selected path for wall jumping character; red rectangle nodes are stable node groups, blue line nodes are node group links

After the path is revealed, all the character must do is follow the nodes in the path in order and with the movements registered in the nodes. In each physics cycle, the character's node is checked, the next node that needs to reach is pulled from the path, and the actions to reach this node are taken over the node link. When the character repeats its movements, it reaches the aimed node and checks for data for the next node. The progress of the character to the target by following the path can be examined in figure 3.27.

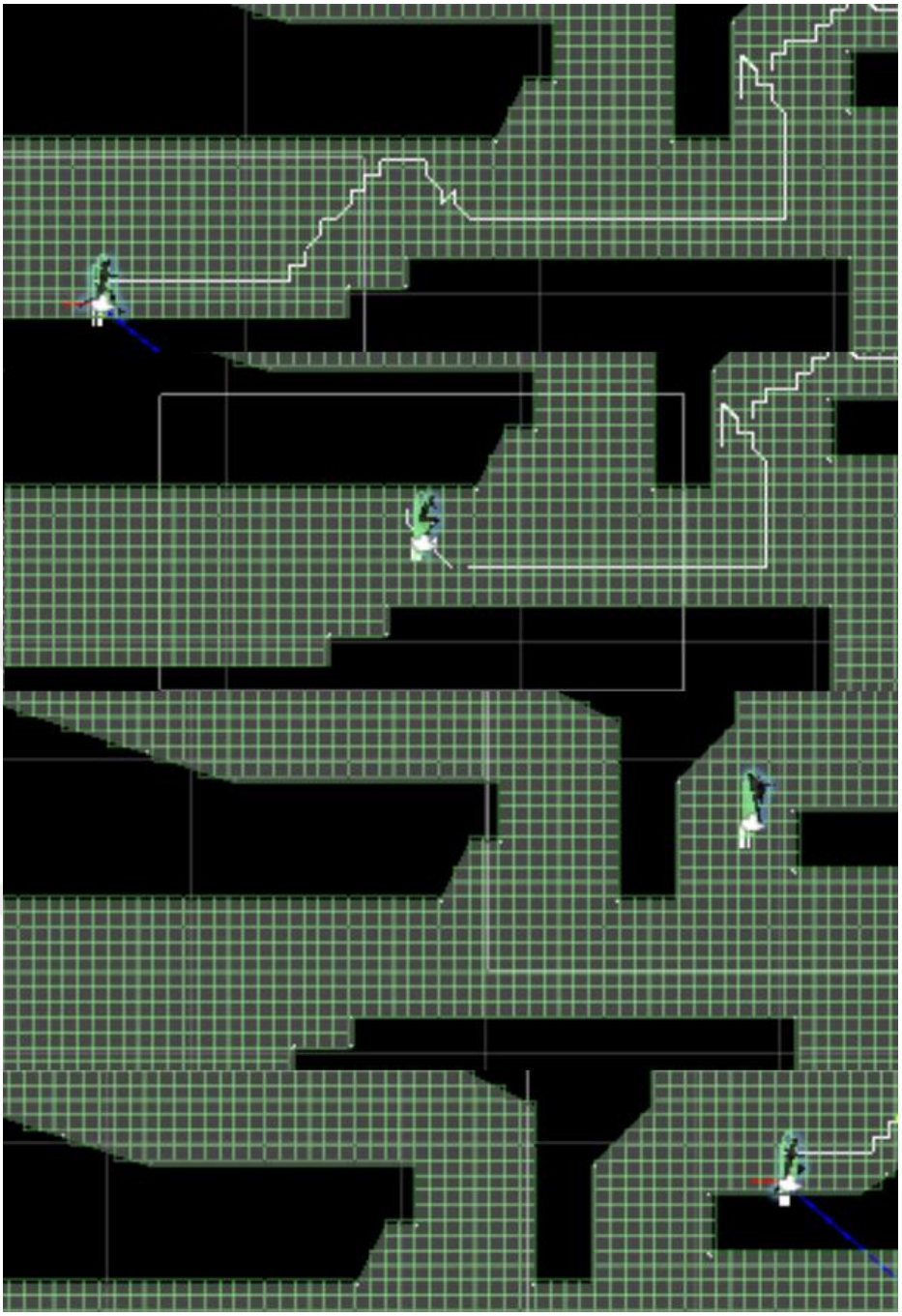


Figure 3.27 Character following the path

3.6 Results

The training was carried out by taking the node width and node height values to 0.2 to reveal the worst-case scenario. There are 25 nodes per Unity unit and training is carried out in approximately 45000 nodes. The training done using the default Unity hyperparameters.

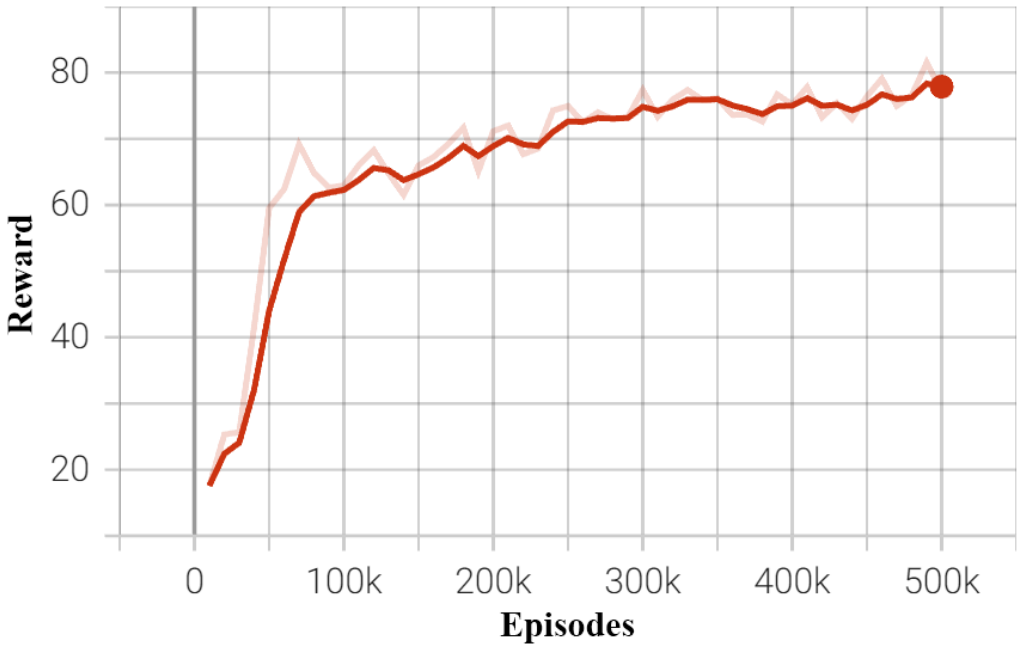


Figure 3.28 Fifty thousand steps of training on the main environment

The training example shown in figure 3.28 was carried out in the environment for the double jump agent using convolutional neural networks. After a training of approximately 500000 steps, which can be considered short for the literature, the node training was started by running the trained agent. The duration of node training may be shortened by keeping the training longer and allowing the agent to learn the movement mechanics better. The aim to measure at this point is to see to what extent the agent can travel the map and create connections between nodes by using newly learned mechanics.

In each episode, the agent and the target are placed in a random location only by considering the distance from each other. This distance is determined as a minimum of 1 Unity unit and a maximum of 5 Unity units. If the agent touches target or one of the traps, or exceeds the allotted 15 second time limit, the target and agent are created at a new location, resetting the loop. Since the location is determined randomly in each cycle, the agent may instantiate directly on a trap so that it cannot escape in some episodes, or as stated earlier the target may be in places that cannot be reached by the relevant agent. No attempt has been made to prevent these.

In this study, mainly the double jump agent was trained and then used. And to see the training performance, the wall jump agent is also trained. The results can be seen in the table 4.1.

Table 4.1 Agents' reward values

Agent	**Minimum Reward**	**Maximum Reward**	**Arithmetic Mean**	**Standard Deviation**
Double Jump	70.035	77.277	74.327	2289.215
Wall Jump	60.723	66.063	63.056	2231.533

The most common problem in the trainings was that the double jump agent reached the same x coordinate of the target but could not make the appropriate moves to reach the y coordinate. For example, in some cases the agent is directly below the target, but the target is out of jump reach. At such times, even though it is located below the target the agent must consciously move away from the target and rise towards the target by making use of the surrounding objects. Especially in such cases, agents who learned to move in the x-axis despite being where it should be and were able to overcome this situation were able to reach higher reward values. In more than half of the trainings, double jump agents trained in the indicated step largely circumvented this problem.

If it is evaluated under more training and different conditions, more precise results can be achieved, but it can be said that the training of the wall jump movement is more difficult than the double jump. The performance of the wall jump mechanic is much

lower in terms of both the average episode length and the maximum reward that can be achieved. However, it should be noted that the nodes that can be reached by wall jump are much less than those that can be reached by double jump. This makes it difficult to compare the two mechanics. Yet, it should be considered that while wall jumping, it is always necessary to consciously move away from the target and unlike double jump, more complex mechanics should be applied as a combination of more than one input.

Node training was done with a trained double jump agent. The results that emerged after a two-hour node training can be examined in figure 3.29.

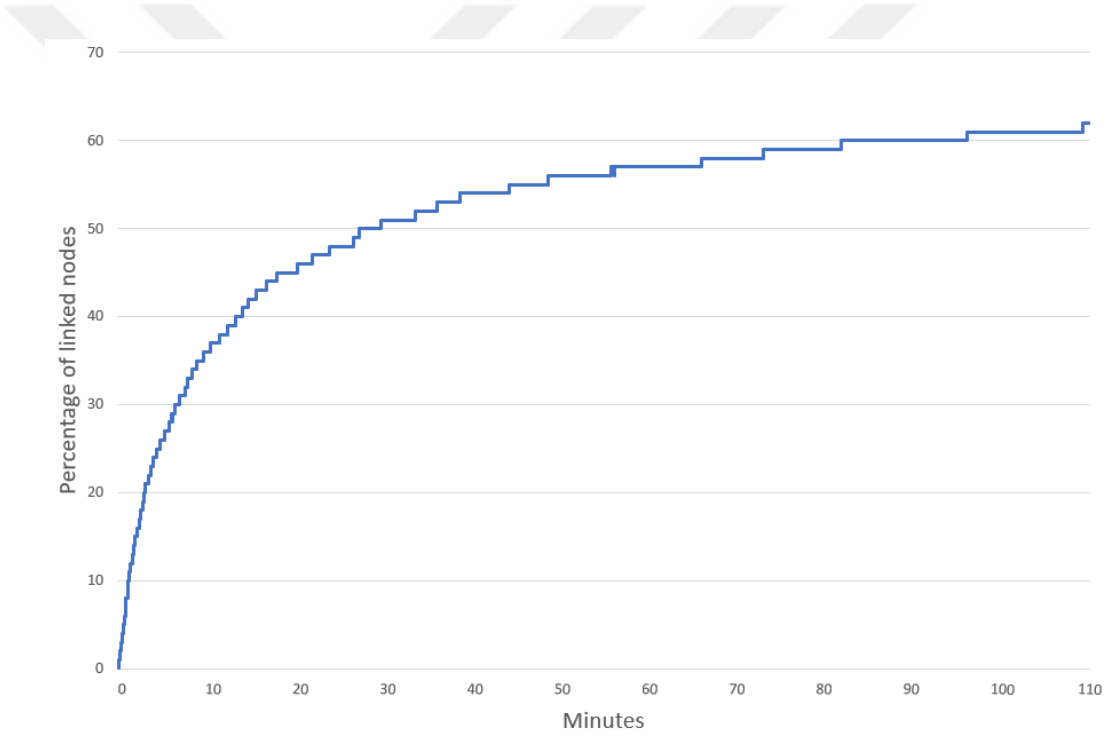


Figure 3.29 Node link training using neural network after training, each percent represents about 500 nodes

The agent enabled more than 60% of the nodes in the environment to connect to another node at least once in about two hours. Considering that the environment accessible node rate for this agent is approximately 80% and given that the main goal of this training is stable intergroup links, it can be said that the node training has been

completed to a large extent. If figure 3.30 is examined, the set of nodes that the agent cannot reach and have difficulty in accessing with this training can be examined.

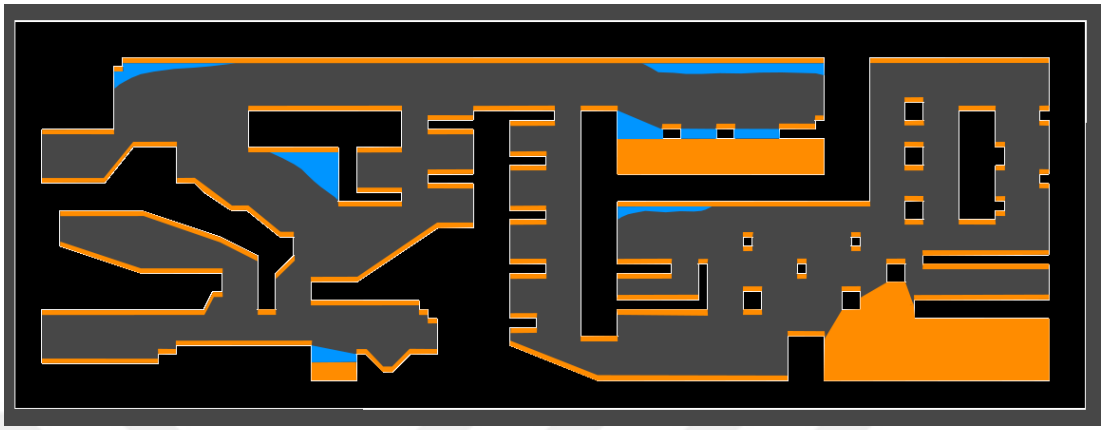


Figure 3.30 The orange zone is the area that the agent can never reach, and the blue zone is the areas that are difficult to reach unless target object is specially located to target that area

It was more difficult for the agent to establish links for stable groups in some areas of the map than in others. The most difficult areas are the areas where the agent must climb directly and consciously move away from the target. The difficulty map of the agent can be seen in figure 3.31. In order of magnitude of which regions are more difficult for the agent from least to most shown in green, orange, red.

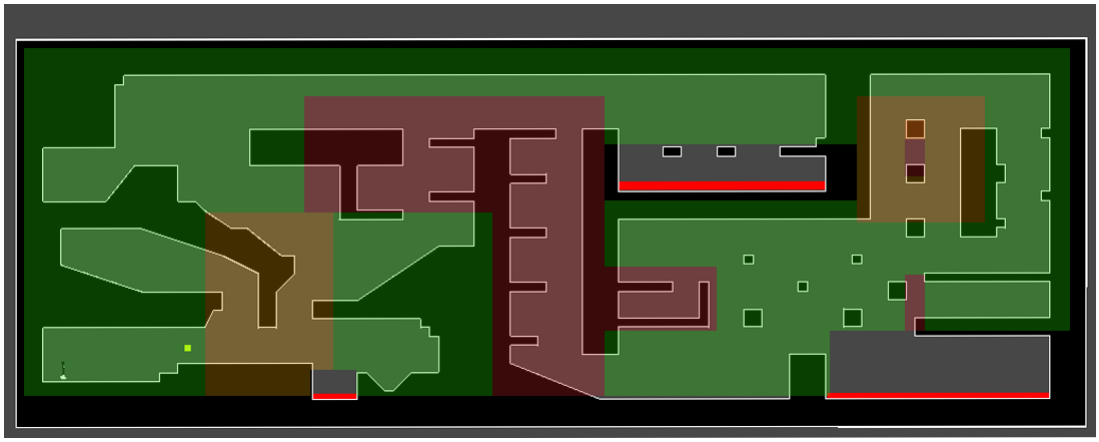


Figure 3.31 Difficulty map

For the proposed pathfinding system, mostly links inside stable node groups which have only two nodes and intergroup links are used. Node links other than these will

most likely not be used during pathfinding. In this respect, training all possible nodes and trying to connect all possible links, or only creating links that will form stable groups and connections between them are choices that will vary depending on the product to be produced. During all this node training, the agent trained all 43 node groups that could be stable node groups on the environment and managed to create 259 different links between groups.

It should be noted that in this study, the node training part was completed with a brute force approach. Equal time has been allocated for each region and situation, and no special effort has been made to reach the difficult groups. For each group, it was deemed sufficient to create at least one link to nearby groups. However, it may not be possible to create all links on different maps and with different training data. For such cases, a smarter structure may be the subject of further study.

In the studied map, the percentage of reaching the node groups in the red region shown in figure 3.31 is much lower than in the other regions. Therefore, it can be said that they are the node groups that determine how long will be the node training time. In map scenarios where there are many hard-to-reach node groups, it can be beneficial in terms of time to keep the agent training as long as possible and to enable the agent to learn the movement mechanics better. Secondly, as stated earlier, a structure can be developed that will select difficult areas more times. In order to do this, it may be useful to determine the node groups before the node training. On maps where these difficult regions are scarce, it may be a better method to roughly complete the node training and to detect the unconnected ones by searching the group list.

CHAPTER 4
CONCLUSION

In this study, a new model and approach for pathfinding has been introduced. A more holistic pathfinding approach is provided with motion mechanics and position data. Since the same system is used for all regions of the map and all movement mechanics, it is easier to create more natural character movements. In addition, there is no need for secondary operations to be performed while obtaining this natural movement. Since the system is a whole, problems encountered in games, such as, what should happen when character getting damaged on the link or somewhat interrupted while on the link, do not apply to this model. After the interrupted movement, the model can create a new path from the point where it is located and continue its movement naturally without the need for a second operation.

Secondly, this whole system is automated. The character and node training time for a character from scratch was completed in about five hours on a model where time moves five times faster and completed without any human effort. The time specified here is the average time for the double-jump agent to complete the map shown in the study. The training was conducted on a computer with an RTX 2060 graphics card, 16 GB of memory and a 9th generation i9-9900K processor. The training of the character neural network is also included in the time. If the same character is wanted to be used on a different map, the time for the second map can be shortened by using the ready trained one. Thanks to the reusability of artificial neural networks, it is also possible to create sandboxes for character training and to perform node training on real maps.

In the last part of the model, the neural network is completely disabled. Machine learning is only used to speed up model creation. Therefore, it is not possible to have a neural network-induced performance loss while pathfinding with the proposed model.

One of the biggest disadvantages of the proposed system is that it needs to keep more data in memory than normal pathfinding applications. A character who can only

use the jump mechanic apart from navigating the map, data on whether this mechanic is applied per link on the map should also be kept. In addition, since the map experiences of different characters will be different from each other, these data must be kept per character. Secondly, development of a data structure for this data to be kept may be beneficial. Even if there are places on the map that the characters cannot reach, filling these places with nodes is done deliberately to create a common node pool for all characters. In this way, inaccessible nodes, and interactions of each character with all nodes could be observed. However, what kind of data structure should be developed for each character on this node pool has been left out of the scope of this study. This data structure, which will determine the character activity regions and store the movement mechanics of the character, may be the subject of further study.

Secondly, it should be noted that the proposed system does not travel between two points as point-and-shoot like normal pathfinding systems. Agents always reach a point close to their target within a certain tolerance value. The factor that determines this tolerance value is the node width and height. This node resolution can be increased for more precise pathfinding. However, it should be noted that every time the resolution is increased, there will be much more nodes to calculate which may affect the performance. In the study, this situation is examined in such a way that there are four nodes for each Unity unit and twenty-five nodes for each Unity unit. For comparison, the size of the characters in the work is about two Unity units. In both cases observed, although the pathfinding part is still at an early stage, there was no significant loss of performance, or the characters not visibly deviated from their goals.

One of the most important issues to be considered on the model is the determination of stable node groups. The criteria by which stable node groups will be determined will vary per character. For the wall-jump and double-jump agents used in the study, it was sufficient to consider stable nodes only when they touched the ground. However, this situation needs to change for a character who can fly indefinitely. For such a character, all nodes it can access will be stable. Therefore, for this character, our model will have turned into a model that does pathfinding over a single group. For a character that can climb walls freely, the points where it can hold may also need to be considered

as stable nodes. In the study, what kind of method should be developed for different characters was left out of scope.

A static map was used in the study and pathfinding was made for only one character at a time. Therefore, the issue of how the model will work in dynamic maps and multi agent situations has been left out of the scope of the study.

REFERENCES

Abreu, N. (2022). Efficient Deep Learning for Multi Agent Pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence,* 36(11), 13122-13123. https://doi.org/10.1609/aaai.v36i11.21697

Agapitos, A., O'Neill, M., Nicolau, M., Fagan, D., Kattan, A., & Curran, K. (2015). Deep evolution of image representations for handwritten digit recognition. *IEEE Congress on Evolutionary Computation (CEC),* 2452-2459. https://doi.org/10.1109/CEC.2015.7257189

Albawi, S., Bayat, O., Al-Azawi, S., & Ucan, O. N. (2018). Social Touch Gesture Recognition Using Convolutional Neural Network. *Computational Intelligence and Neuroscience,* 2018. https://doi.org/10.1155/2018/6973103

Algfoor, Z. A., Sunar, M. S., & Kolivand, H. (2015). A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games. *International Journal of Computer Games Technology,* 2015(2), 1-11. https://doi.org/10.1155/2015/736138.

Alonso, E., Peter, M., Goumard, D., & Romoff, J. (2020). Deep Reinforcement Learning for Navigation in AAA Video Games. *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI-21),* 2133-2139. https://doi.org/10.24963/ijcai.2021/294

Arrebola, M. (2022, June 15) *Madea by M. Arrebola.* Mixamo. https://www.mixamo.com/#/?page=1&query=Arrebola&type=Character

Bjornsson, Y., Enzenberger, M., Holte, R., Schaeffer, J., & Yap, P. (2003). Comparison of different grid abstractions for pathfinding on maps. *IJCAI'03: Proceedings of the 18th international joint conference on Artificial intelligence,* 1511–1512.

Brož, P., Kolingerová, I., & Apu, R. A. (2007). Path planning in dynamic environment using an adaptive mesh. *SCCG '07: Proceedings of the 23rd Spring Conference on Computer Graphics,* 233-239. https://doi.org/10.1145/2614348.2614381

Cui, M. L., Harabor, D. D., & Grastien, A. (2017). Compromise-free Pathfinding on a Navigation Mesh. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, 496-502. https://doi.org/10.24963/ijcai.2017/70

Fergus, R. (2022, December 5). *Introduction to Convolutional Networks*. New York University. https://cs.nyu.edu/~fergus/teaching/vision/7_convnets.pdf

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn & TensorFlow : Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc.

Ghosh, S. K., & Goswami, P. P. (2013). Unsolved problems in visibility graphs of points, segments, and polygons. *ACM Computing Surveys*, 46(2), 1-29. https://doi.org/10.1145/2543581.2543589

Graham, R., McCabe, H., & Sheridan, S. (2003). Pathfinding in Computer Games. *The ITB Journal USA*, 4(2), 57-81. https://doi.org/10.21427/D7ZQ9J

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017). Snapshot Ensembles: Train 1, get M for free. *arXiv*. https://doi.org/10.48550/arXiv.1704.00109

Iskandar, U., Diah, N., & Ismail, M. (2020). Identifying Artificial Intelligence Pathfinding Algorithms for Platformer Games. *2020 IEEE International Conference on Automatic Control and Intelligent Systems (I2CACIS)*, 74-80. https://doi.org/10.1109/I2CACIS49202.2020.9140177.

Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018). Unity: A General Platform for Intelligent Agents. ArXiv, abs/1809.02627. https://doi.org/10.48550/arXiv.1809.02627

Kaelbling, L. P., Littman, M. L., & Moore A. W. (1996). Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4, 237-285. https://doi.org/10.48550/arXiv.cs/9605103

Koo, J., Mendiratta, V., Rahman, M. R., & Walid, A. (2019). Deep Reinforcement Learning for Network Slicing with Heterogeneous Resource Requirements and Time Varying Traffic Dynamics. *15th International Conference on Network and Service Management (CNSM)*, 1-5. https://doi.org/10.23919/CNSM46954.2019.9012702.

Krafft, C. (2021). *Optimisation of pathfinding in a dynamic 3D space.* [Master thesis] University of Applied Sciences Hamburg

Lawande, S. R., Jasmine, G., Anbarasi, J., Izhar, L. I. (2022). A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games. Applied Sciences. 12(11), 5499. https://doi.org/10.3390/app12115499

Mixamo. (2022, June 15). *Locomotion Pack.* https://www.mixamo.com/#/?genres=&page=1&query=locomotion&type=Motion%2CMotionPack

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518, 529--533.

Narejo, S., Talpur, S., Memon, M., & Rahoo, A. (2020). An automated system for traffic sign recognition using convolutional neural network. *3C Tecnología. Glosas de innovación aplicadas a la pyme. Edición Especial,* Noviembre 2020, 119-135. https://doi.org/10.17993/3ctecno.2020.specialissue6.119-135

O'Shea, K., & Nash, R. *(2015). An Introduction to Convolutional Neural Networks.* arXiv. https://doi.org/10.48550/arXiv.1511.08458

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. ArXiv, abs/1707.06347. https://doi.org/10.48550/arXiv.1707.06347

Unity. (2022, June 15). *Unity-Technologies/ml-agents.* https://github.com/Unity-Technologies/ml-agents/blob/develop/config/ppo/3DBall.yaml

van Toll, W., Triesscheijn, R., Kallmann, M., Oliva, R., Pelechano, N., Pettre, J., & Geraerts, R. (2016). A comparative study of navigation meshes. *MIG '16 - 9th International Conference on Motion in Games*, 91-100. https://doi.org/10.1145/2994258.2994262

Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into Imaging,* 9, 611-629. https://doi.org/10.1007/s13244-018-0639-9

Yani, M., Irawan, B., & Setiningsih, C. (2019). Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail. *Journal of Physics: Conference Series*, 1201(1). https://doi.org/10.1088/1742-6596/1201/1/012052

Zhu, W., Jia, D., Wan, H., Yang, T., Hu, C., Qin, K., & Cui, X. (2015). Waypoint Graph Based Fast Pathfinding in Dynamic Environment. *International Journal of Distributed Sensor Networks*, 11(8). https://doi.org/10.1155/2015/238727