# Razvoj modela strojnog učenja koji igra video igru žanra platformer u alatima Unity i TensorFlow

**Alilović, Dario**

**UNIVERSITY OF ZAGREB**
**FACULTY OF ORGANIZATION AND INFORMATICS**
**VARAŽDIN**

**Dario Alilović**

# DEVELOPING A MACHINE LEARNING MODEL FOR PLAYING A PLATFORMER VIDEO GAME IN UNITY AND TENSORFLOW

**MASTER'S THESIS**

**Varaždin, 2022**

**UNIVERSITY OF ZAGREB**

**FACULTY OF ORGANIZATION AND INFORMATICS**

**V A R A Ž D I N**

**Dario Alilović**

**Student ID: 0016130019**

**Programme: Information and Software Engineering**

# DEVELOPING A MACHINE LEARNING MODEL FOR PLAYING A PLATFORMER VIDEO GAME IN UNITY AND TENSORFLOW

**MASTER'S THESIS**

Mentor:

Bogdan Okreša Đurić, PhD

**Varaždin, September 2022**

*Dario Alilović*

**Statement of Authenticity**

Hereby I state that this document, my Master's Thesis, is authentic, authored by me, and that, for the purposes of writing it, I have not used any sources other than those stated in this thesis. Ethically adequate and acceptable methods and techniques were used while preparing and writing this thesis.

*The author acknowledges the above by accepting the statement in FOI Radovi online system.*

---

**Abstract**

This thesis deals with game development and the use of artificial intelligence in the domain of video games, especially artificial neural networks. It covers some of the basics of game development in the Unity game engine, as well as some basics of machine learning and artificial intelligence. Basically, it follows the process of developing a platform game using the Unity game engine and applying a deep q learning algorithm, developed using the Python library TensorFlow, to teach the agent how to play said game. The thesis concludes that Unity is an excellent reinforcement learning tool since it provides a secure and modular environment in which agents can learn.

**Keywords**: AI; Machine Learning; Unity; TensorFlow; Game development; Platformer; Neural networks

# Table of Contents

# 1. Introduction

Video games have been around for a while, improving in gameplay and graphics along the way. Today there are many games in different genres and styles to choose from and they all have their own setting and rules. We as humans observe these environments using our senses such as sight and hearing, and take appropriate actions based on those senses. While in the past it was extremely difficult to develop a game and often required an entire team, today one person can make a game of their choice thanks to advances in programming languages and more importantly game engines that provide the building blocks for such tasks.

This begs the question: can a computer learn to play a game by itself, using and reacting to its own senses? The answer to this question is yes and this thesis explores this question. This is where artificial intelligence comes into play. This area of research is one of the fastest growing areas today and opens the door to many possibilities. Learning how to play a game is just one example of using artificial intelligence. Although the goal of video games is to provide entertainment, they can also be used as an environment for teaching some machine learning algorithms. For example, a good car driving game simulation can act as an environment for teaching algorithms for self-driving cars since they cannot be put into traffic to learn. Because AI is so versatile, many other fields can benefit from this type of learning because learning in simulations is a much safer alternative to real-life trial and error. A great example of this is probably in the medical field, where an AI powered surgical assistant could not learn to operate on real people.

There are a lot of other uses for AI, but this thesis will be focusing primarily on its use in video games. The thesis will cover some basics of video game development, specifically 2D platformers, some of the basics of artificial intelligence and machine learning and examples of some of the algorithms. Finally, to conclude everything, a practical example of a platformer game is presented together with a neural network brain that learns how to behave in the game world.

# 2. Work Methods and Techniques

The first part of the thesis refers to game development and the description of the platformer game genre. This part mainly covers the basics of the Unity engine which is the chosen tool for the game development part of the thesis, along with Visual Studio for scripting. Most of the information needed to develop a game comes from Unity's documentation, which is extremely user-friendly and easy to navigate. Because of this, most tasks from creating simple objects to animating those objects are easy to implement. The visual part of the game (i.e. level tiles, obstacles, pickups, ...), except for the player character and the parallax background, is hand-drawn using GIMP.

The second part of the thesis deals with artificial intelligence and machine learning. This section covers some of the basics of artificial intelligence and four approaches to it. It also dives deeper into machine learning, a subfield of artificial intelligence, describing all types of machine learning and several algorithms that belong to a particular type. All information on this topic comes from various books on artificial intelligence and machine learning, along with some scientific journals from Google Scholar and various websites that focus on this topic. The tools used to develop the AI part are the Python programming language whose code was written in Visual Studio Code, along with several python libraries, notably mlagents, TensorFlow and Keras.

Finally, to connect these two parts, Unity mlagents toolkit package is used. The approach taken for the whole project is as follows. First, the learning agent is provided with sensors that observe the world in which it is located. This is important because a neural network that will adapt to that world needs concrete observations as input to be able to react to them. These observations are then sent to a Python program that generates an action for the agent. The agent then performs that action and gives the network an appropriate reward or penalty depending on the quality of that action. The network then calculates the value of the loss and adjusts its weights accordingly. This process does not stop until the player dies or a set time expires (so the player does not get stuck) and spans multiple iterations.

# 3. Game Development

Game development is a process of developing a game. The whole process of developing a video game can be done by a single person or by a huge game making company with teams dedicated to a specific part of the whole process. A single developer today can develop a whole game in reasonable amount of time thanks to game development software, or game engines, that became easily accessible and widespread in the recent years [1].

Game engines make the whole process of developing a game much easier, which is why many developers choose to use them for their games. Most of them have built-in features that handle rendering for 2D and 3D, collision detection and physics, sound and other features along with the ability to write custom scripts [2].

Some of the more popular game engines are:

- Unity Engine
- Unreal Engine
- CryEngine
- Game Maker
- Construct

For the purposes of this paper, the Unity Engine will be used, which will be discussed in the following chapter.

## 3.1. Unity Engine

Unity is a game engine that supports development of video games on most platforms including desktops, computer consoles, virtual reality, augmented reality, mobile devices (android, iOS), TV platforms, Web platforms and others [3]. David Helgason, Nicholas Francis and Joachim Ante developed the first version of Unity, which was released in 2005. They continued improving the engine until 2008 when Unity began to grow in popularity, which allowed the company to expand. That same year, Apple introduced its App Store, which caught the attention of Unity developers. They soon developed support for Apple's iPhone, and since they were the first in the industry to do so, Unity's popularity skyrocketed. Large companies, such as Cartoon Network, began using Unity to develop their own games, which further boosted the success of the engine. Today, Unity is among the leading, if not the largest, environments for video game development [4].

Although it is famous for its 3D capabilities, Unity can also be used to create 2D games, for which it has some specific tools, along with tools that are used for both the 3D and 2D development. The first difference that can be seen is the *2D view mode* button located in the top **toolbar** of the **scene view** which, when enabled, sets up the orthographic view that locks

the camera to look along the Z axis while the Y axis is the one pointing upwards [5]. This thesis will focus exclusively on 2D development, and will not discuss any of the 3D specific features, or any features that will not be used in developing a platformer game.

In the following few sections the Unity interface is addressed, and explained in detail the purpose of all of its components that are used for the purposes of this thesis, specifically for the process of developing a game.

### 3.1.1. Unity Interface

**The project window**, shown in figure 1, is a part of the interface that contains all the files and directories related to the current project and it is the main way of finding files like assets, images, sounds and other files that would be used. When starting a project, this window is open by default and is typically located at the bottom of the screen. It consists of two columns, one showing the file hierarchy and the other showing the contents of the currently selected folder as a visual preview [6].

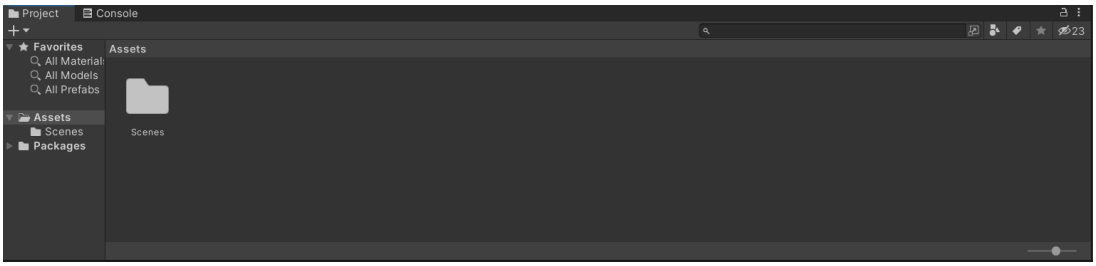


Figure 1: Project window

Top edge of the project window is the location of the browser toolbar. This toolbar has a few tools related to the project window and its functions can be seen in table 1.

Table 1: List of tools in browser toolbar [6]

| Property | Description |
|---|---|
| Create menu | Displays a list of Assets and other sub-folders you can add to the folder currently selected. |
| Search bar | Use the search bar to search for a file within your Project. You can choose to search within the entire Project (All), in the top level folders of your Project (listed individually), in the folder you currently have selected, or within the Asset Store. |
| Open in Search | Opens the Unity Search tool to refine your search. |
| Search by Type | Select this property to confine your search to a specific type, for example Mesh, Prefab, Scene. |
| Search by Label | Select this property to choose a tag to search within. |
| Save Search | Saves your search under Favorites in the left panel. |
| Hidden packages count | Select this property to toggle the visibility of the packages in the Project window. |

**The Scene view**, shown in figure 2 is the interactive part of the interface, which is used while creating a world. It can be used to select and position elements like characters, lights, cameras

or other game objects [7]. This view is essentially the main part of the interface as it is used to create most of the visual parts of the game.

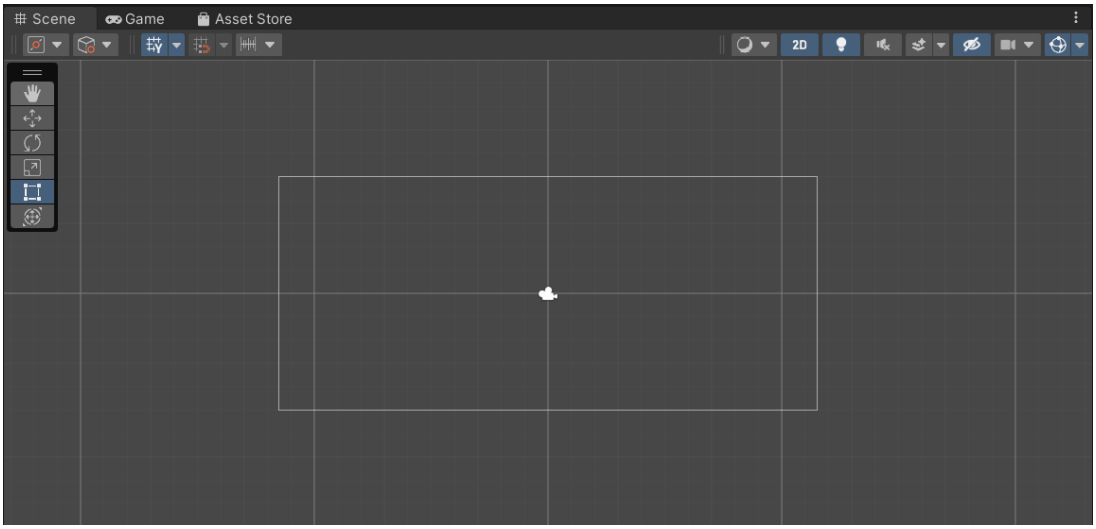


Figure 2: Scene view

To navigate the scene view, arrow keys can be used, as well as the mouse. Arrow key navigation is pretty much how it would be expected as in up and down keys move camera up and down and left and right keys move it sideways. The mouse navigation works on a drag principle, which is implemented in two ways: Alt + middle-click + drag or Alt + Control + left-click + drag. Holding a right mouse click allows user to pan around scene view. To zoom in inside the scene view the user can choose between two options: scroll wheel or Alt + right-click + drag [8].

Game objects placed inside the scene view can be selected individually or as a group. The selected object is highlighted with a orange outline in the scene view along with all its children which are highlighted in a blue outline. To highlight the object, the user can click on it in the scene view (in the case of multiple stacked objects, highlight will be cycled between all the objects in the stack with every click) or select it in hierarchy window, which is described in a later section. To select multiple objects in the scene view click and drag with a mouse will draw a rectangle that selects anything in its boundaries, or a shift + left-click method can be used. To select/deselect a specific object to/from selection there is control + left-click option [9].

To manipulate game object in the scene view, there are five transform modes that can be selected in the toolbar or using a keyboard shortcut and each of them is shown in figure 3 in order as listed below:

- W for Move
- E for Rotate
- R for Scale
- T for RectTransform
- Y for Transform

In the center of the move gizmo, there is a small rectangle that can be used to drag the object on X and Y axis, or the arrows can be used to move in only up/down or sideways. When the rotate tool is selected, the rotate gizmo appears. It consists of circles that are used to rotate the object along a specific axis. The scale tool is used to resize the object evenly on all axes by clicking the rectangle in the middle of the scale gizmo, or to resize the object on one axis individually by clicking the rectangles at the ends of the scale gizmo. The RectTransform tool is commonly used in 2D to place or resize the game objects. By clicking and dragging the circle in the middle of the gizmo, the user can move the object on one plane or by clicking and dragging the corner circles the user can resize the object. Finally, the transform tool combines Move, Rotate and Scale tools into one [10].

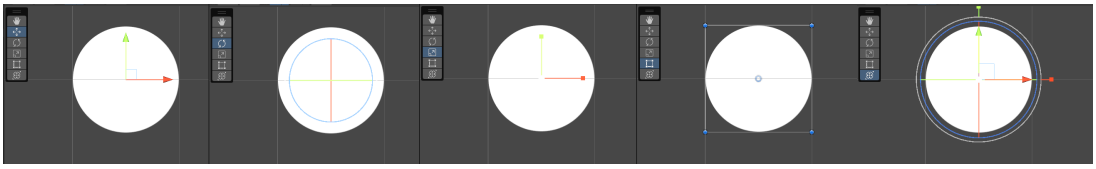

Figure 3: Object transformations

**The Game view**, shown in figure 4 is the part of the interface that is rendered from the camera(s) and it represents the final look of the game. To control what the player sees, there must be one or more cameras placed in a scene view that will then render the game in the game view (or in a final, published game).

This window is used to test the game during development. For this purpose, there is a play mode that can be started using the toolbar in the game view. All the changes made during play mode will not persist after the reset.

At the top of the game view, there is a control bar that is used to control how the game is rendered and played in the Unity environment and its tools along with their functions are listed in table 2.

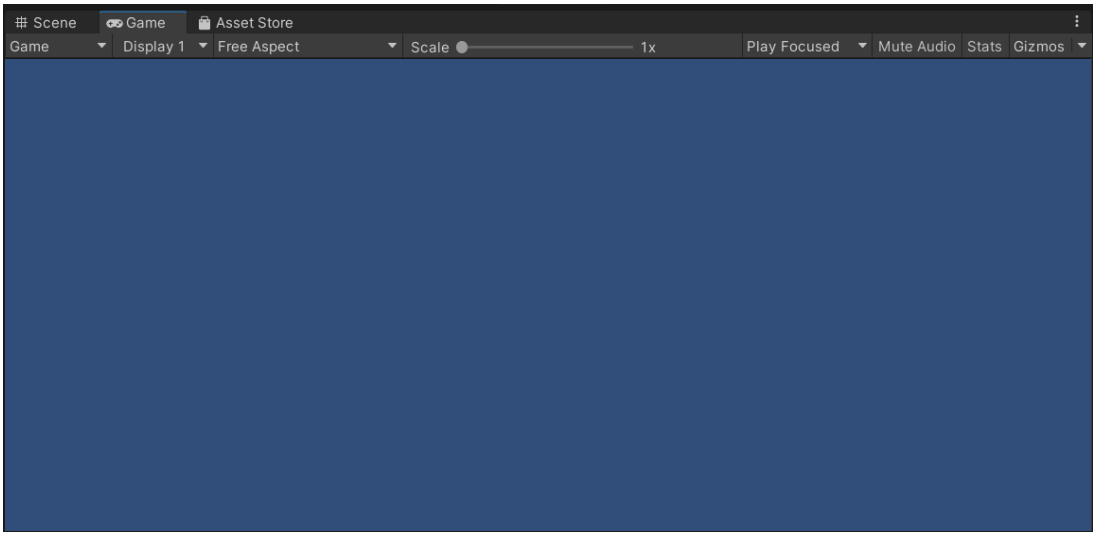


Figure 4: Game view

Table 2: List of tools in the control bar [11]

| Button | Function |
|---|---|
| Game/Simulator | Click to enable the Game or Simulator view from the drop-down menu. |
| Display | Select this to choose from a list of Cameras if you have multiple Cameras in the Scene. This is set to Display 1 by default. You can assign Displays to Cameras in the Camera module, under the Target Display drop-down menu. |
| Aspect | Select different values to test how your game looks on monitors with different aspect ratios. This is set to Free Aspect by default. |
| Scale slider | Scroll right to zoom in and examine areas of the Game screen in more detail. This slider lets you zoom out to see the entire screen where the device resolution is higher than the Game view window size. You can also use the scroll wheel and middle mouse button to do this while the application is stopped or paused. |
| Maximize on Play | Click to enable: use this to maximize the Game view (100% of your Editor window) for a full-screen preview when you enter Play mode. |
| Mute audio | Click to enable: use this to mute any in-application audio when you enter Play mode. |
| Stats | Click this to toggle the Statistics overlay, which contains Rendering Statistics about your application's audio and graphics. This is very useful for monitoring the performance of your application while in Play mode. |
| Gizmos | Click this to toggle the visibility of Gizmos. To only see certain types of Gizmo during Play mode, click the drop-down arrow next to the word Gizmos and only enable the Gizmo types you want to see. |

**The Hierarchy window**, shown in figure 5 displays all game objects placed in a scene. It can be used to select, sort or group objects inside a scene, as well as add/remove them to/from the scene. The Hierarchy window can also display other scenes, if they exist, with each scene being displayed as a parent to objects currently placed inside it [12].

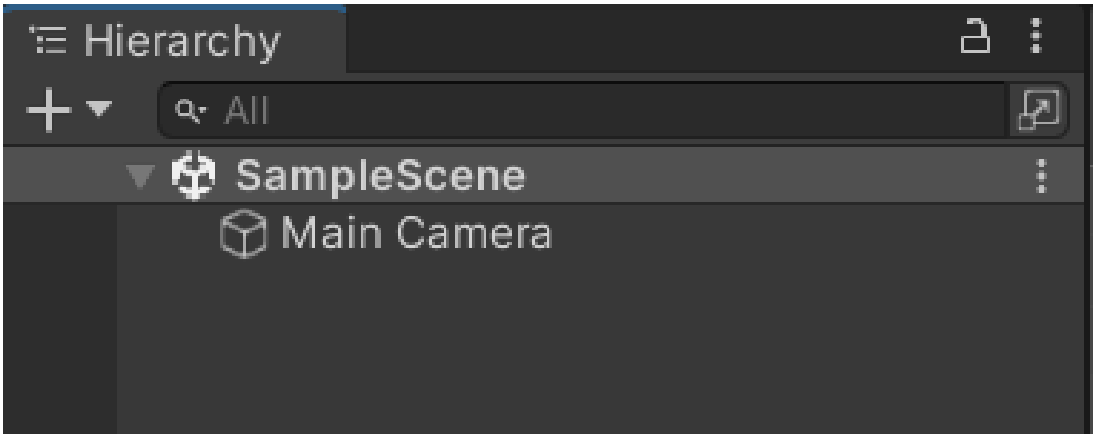


Figure 5: Hierarchy window

Parenting in Unity works in a way that all objects that are in a parent-child relationship are linked together so if the parent gets moved, rotated or scaled, the child gets transformed proportional to its parent. Also, as most of the parent-child hierarchies work, it is possible to collapse all the child objects into one parent [12]. An example of grouping objects in a parent can be seen in figure 6.

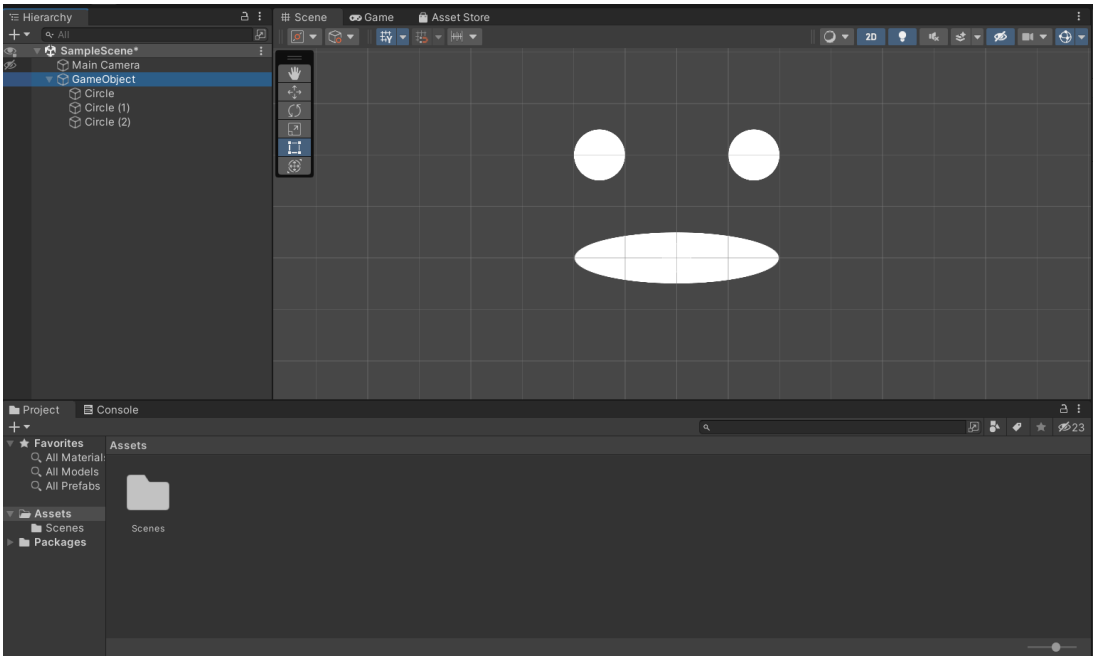


Figure 6: Parenting example

The Hierarchy window can also be used to create a new game object by right-clicking on the empty space in it and selecting the game object that needs to be created, as shown in figure 7 [12].

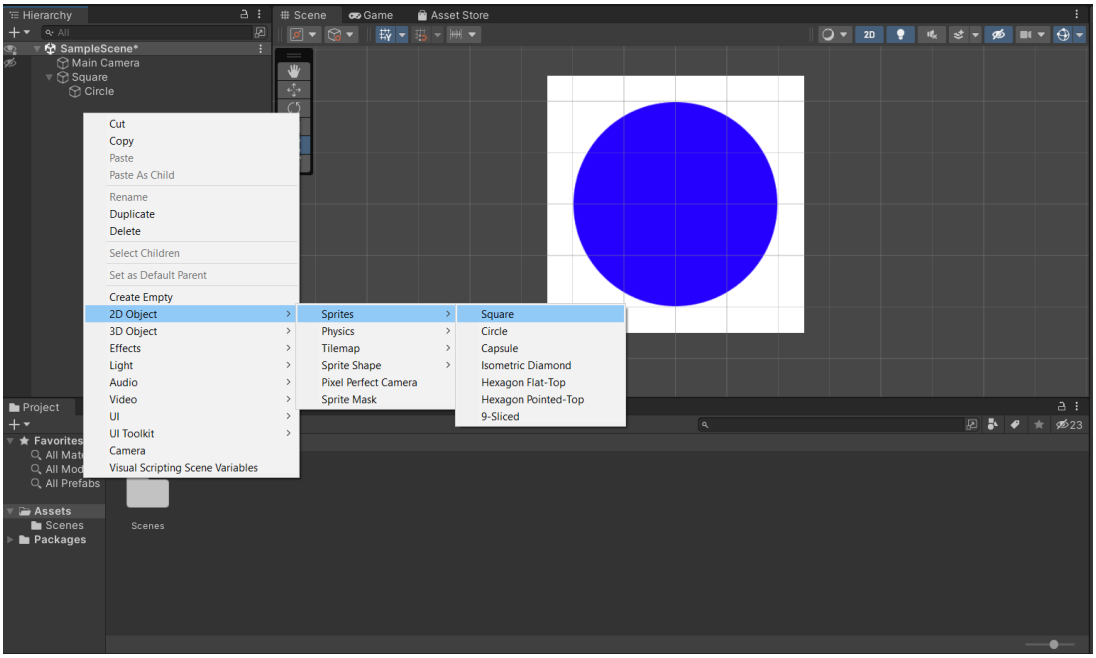


Figure 7: Creating a game object

To sort and group game objects through the Hierarchy window, the user can simply use drag and drop technique. Alternatively, if the user wants to group multiple game objects into one new parent, they can right-click selected objects and select "create empty parent" option which creates an empty object (not visible in a scene) that acts as a parent to the selected objects [12]. An example of creating an empty parent can be seen in figure 8.

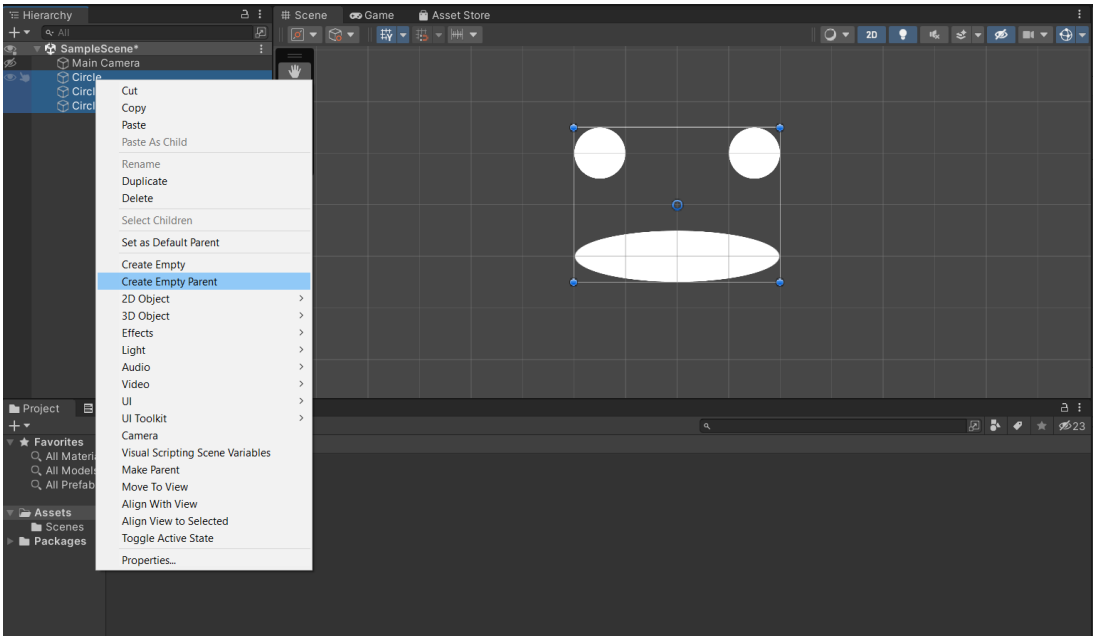


Figure 8: Creating an empty parent

Through the Hierarchy window, it is also possible to duplicate, copy and paste game objects, as well as disable visibility and pickability of the object. Enabling/disabling of the object can be done by clicking on the eye icon left of the object in the Hierarchy window and the same can be done for pickability only by clicking the hand icon next to the eye icon, as shown in figure 9 [12].

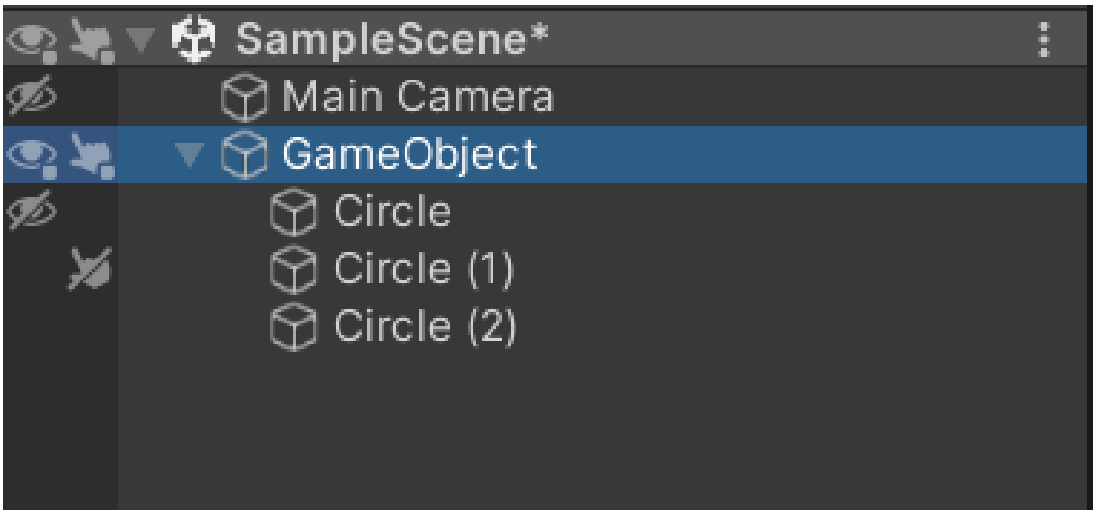


Figure 9: Disabling visibility and pickability

**The Inspector window** is also a very important tool in Unity. It is used to view and edit properties and setting for almost everything in the editor, including game objects, Unity components, Assets, Materials and even editor settings and preferences. The inspector window shows different options based on the currently selected items. If the game object is selected, the Inspector window displays properties of all the components and materials of that object, which can then be edited or reordered through it. This is especially useful when working with scripts, as the public variables can be edited directly through the Inspector window without changing the value in the script itself. This interaction of a script and the inspector window is represented in figure 10 [13].

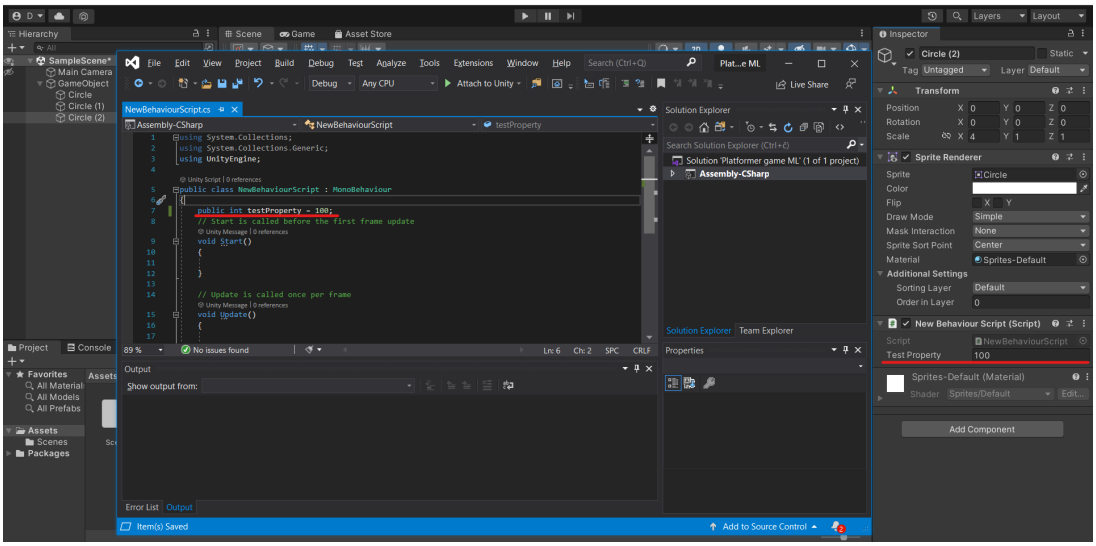


Figure 10: Public property showing up in inspector window

Like any other development environment, Unity engine has its own **Console window** (shown in figure 11) which displays errors, warnings and other messages the editor generates. This window is used to find issues in the project, such as script compilation errors. A Debug class can be used inside the script to print the value of a specific variable at the certain point in the code. The console window works pretty much the same as most other development environment console windows [14].

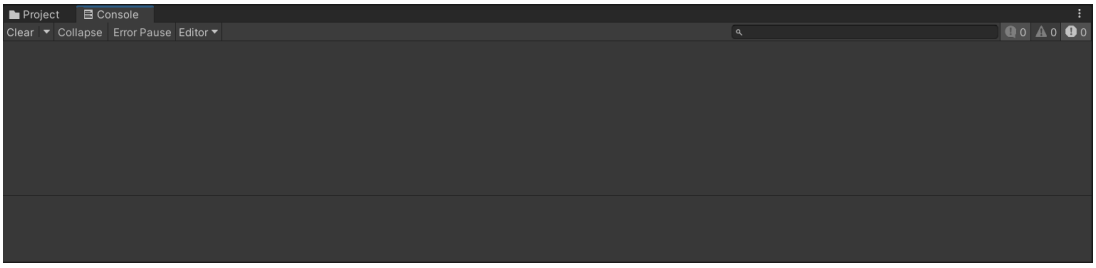


Figure 11: Console window

### 3.1.2. 2D Game Development Support

When developing a 2D game, the user must be familiar with the term "sprite". A sprite is a graphic object [5], that is essentially a texture, but there are special techniques that are used for combining and managing those textures which improves efficiency and is more convenient during development faze.

Unity engine has a built in Sprite Editor which lets the user extract sprite graphics from a single larger image. This method is useful when creating characters, for example. The larger image could contain all the body parts of the character as separate elements.

Unity also has a physics engine specifically made for 2D games. The sprite can be equipped with specialized components that give the sprite a specific feature [5].

**Rigidbody 2D** is a component that enables the physics engine to manipulate the object [15]. This component is used for communicating with the editor's transform component, which defines how the object is positioned, rotated and scaled, to calculate how different objects' colliders interact with each other. An example of this component can be seen in figure 12.

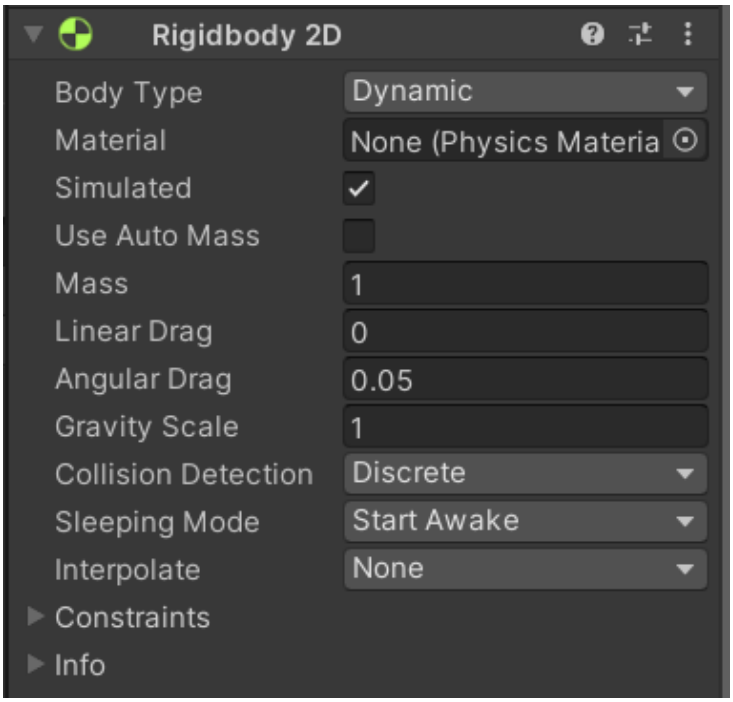


Figure 12: Rigidbody 2D comnponent

**Collider 2D** is a component that defines the shape of a game object and how that object handles collisions. There are multiple different collider shapes but, the collider does not have to be the shame shape as the object [16]. For example, a circle object can have a box collider (square shape), and it would behave as a square. An example of this combination is shown in figure 13.

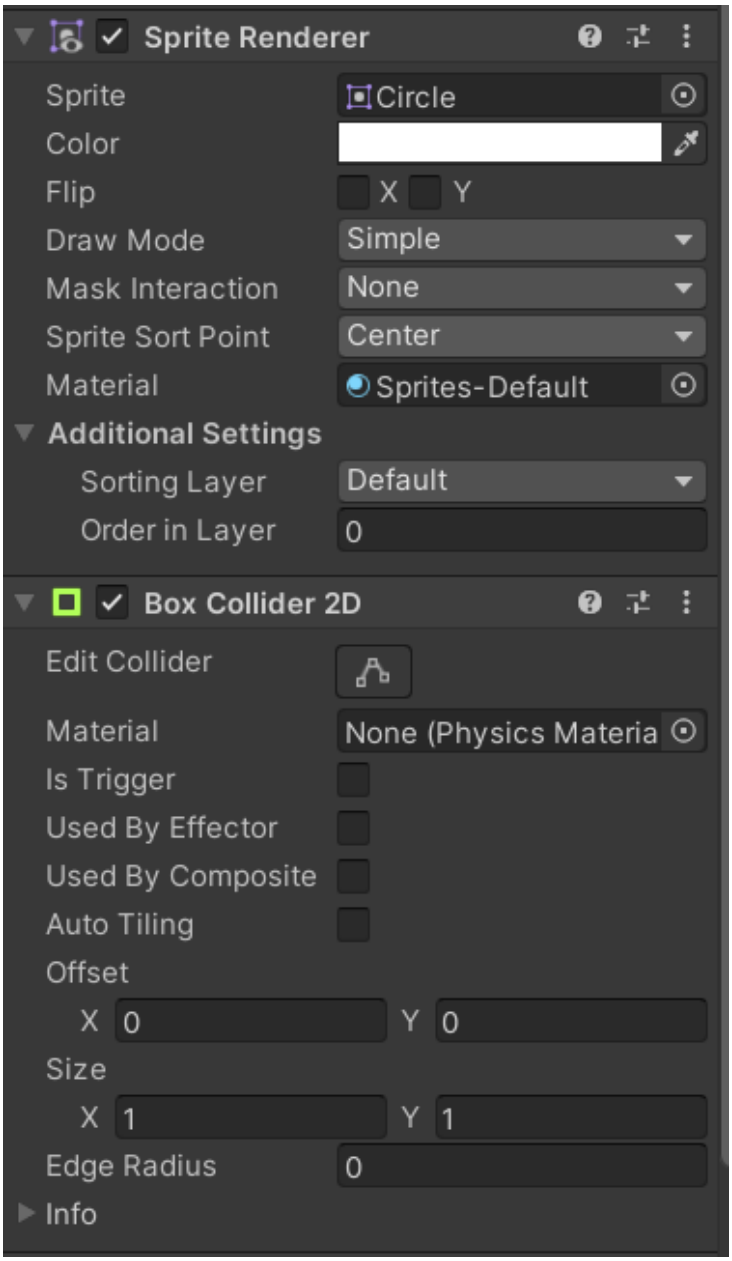


Figure 13: Circle with Box Collider 2D component

**Physics Material 2D** is a component that is used to adjust the friction and bounce between 2D physics objects upon collision. This material is added to the collider component under the "Material" field [17]. An example of this component can be seen in figure 14.

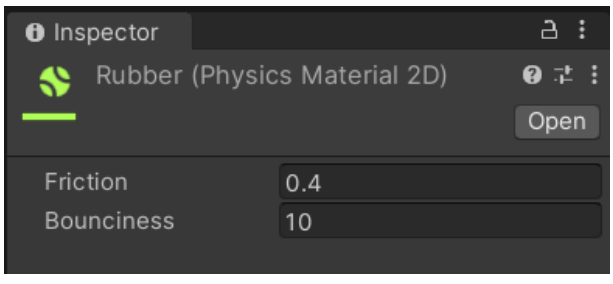


Figure 14: Physics Material 2D component

**2D Joints** are components that attach game objects together. They can be attached only to objects that have a rigidbody 2D component applied to them, or to a fixed position in the world. There are multiple different types of joints and they all have their own constraints that they apply to the rigidbody [18]. For example, setup like the one in figure 15, with the hinge joint, would make the circle behave a pendulum.

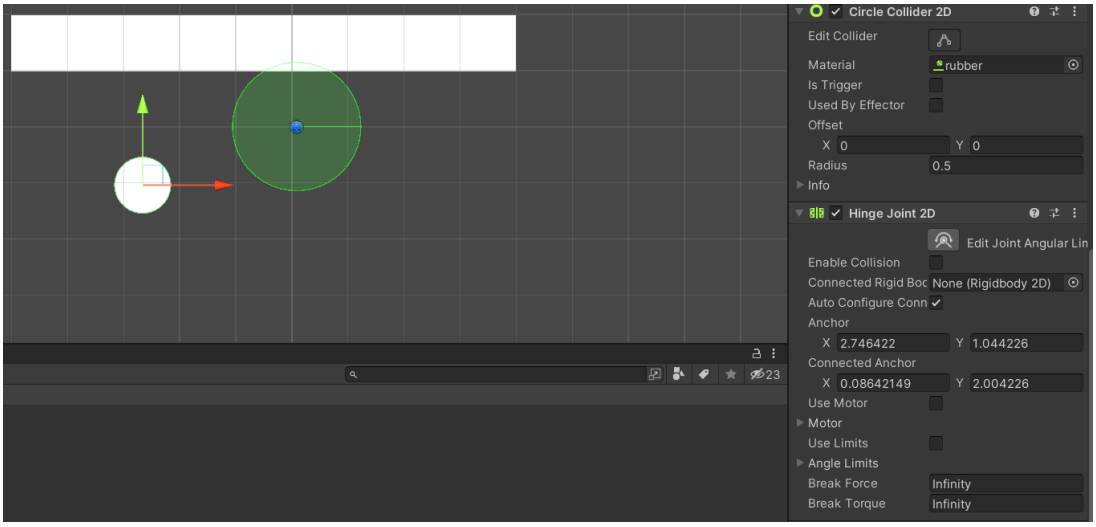


Figure 15: Joint component example

**Constant Force 2D** is a simple component that applies a constant force to a rigidbody making them, for example, accelerate in one direction over time rather than the object starting with a large velocity. This component applies both linear and torque forces continuously to the rigidbody [19]. An example of this component is shown in figure 16.

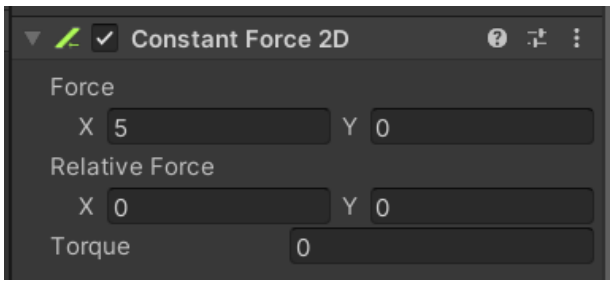


Figure 16: Constant Force 2D component

Finally, **Effectors 2D** are components that, when used with collider component, direct the physics forces in a specific way when colliders get in contact with eachother [20].

When talking about 2D game development, it is also worth mentioning tilemaps. Tilemap [21] is a tool that uses tile assets for creating 2D levels. When the tilemap is added to the scene, the grid component is added as a parent as well and it acts as a guide for tiles to be placed onto the tilemap. To create or modify existing tiles that are placed in the tilemap, the Tile pallete window [22] is used. Textures and sprites can be drag and dropped inside the tile pallete and they become new tile assets that can then be used to create levels on the tilemap.

### 3.1.3. Animations

Unity has its own integrated animation system that can be used to create animated game objects. This paper primarily focuses on sprite animations since the game being made is a 2D game. Sprite animations [23] are animated clips made for 2D assets. The easiest and the most efficient way of importing sprites for animations is using a sprite sheet [24], which is a single file that contains multiple graphics in a grid formation. This way only one file needs to be loaded for a single or even multiple animations. Unity has a system that can handle these types of files, that is, it can slice them into separate animation frames. To do this, in the inspector the sprite mode must be set to multiple [23] and the sprite editor can be opened for that file. The sprite editor can then slice the image either automatically, or manually by selecting the grid size of the sprite sheet [23]. An example of a sliced sprite sheet can be seen in figure 17.

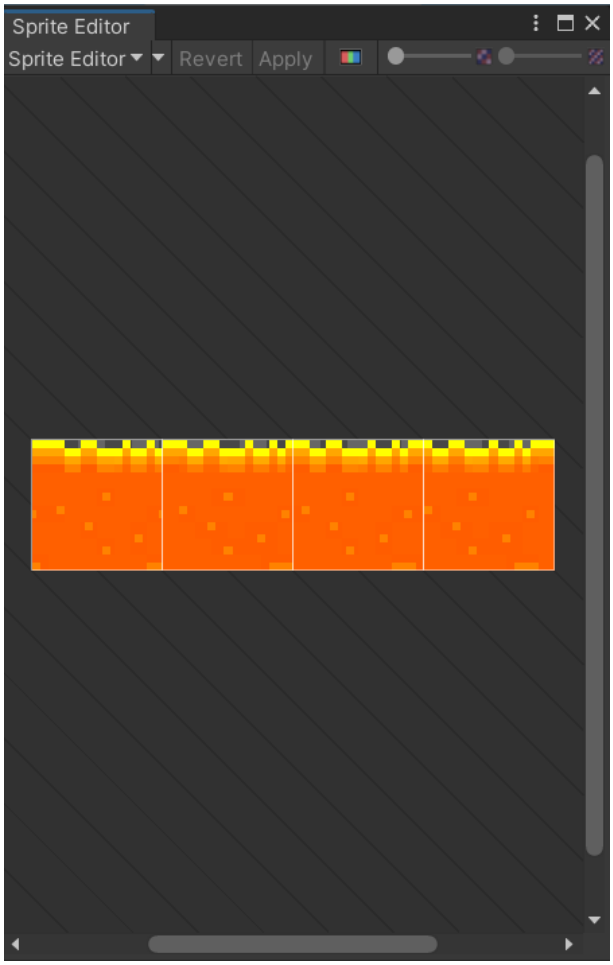


Figure 17: Sliced sprite example

After the sprite has been sliced, an animation can be created through the animation window. The newly created sprites can be dragged into the animation window and be arranged as keyframes of the animation [23]. An example of a sprite animation keyframes can be seen in figure 18.

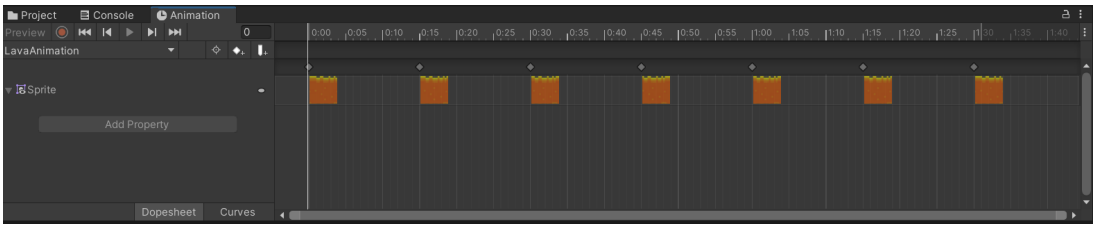


Figure 18: Animation clip keyframes

For simple game objects like an animated tile, this would be enough, but for more complex game objects, like a player sprite, which requires multiple animations for different states, an animator controller is needed. Animator controller [25] is a tool that allows the creation of animation transitions between multiple animation clips using a state machine. Every link between animations can have different parameters [26] that need to be met to actually transition to that animation clip. These parameters can be of four basic types: integer, float, bool or a trigger (a bool that is reset upon transition); and can be modified using scripts. An example of an animator controller with multiple clips and transitions can be seen in figure 19.

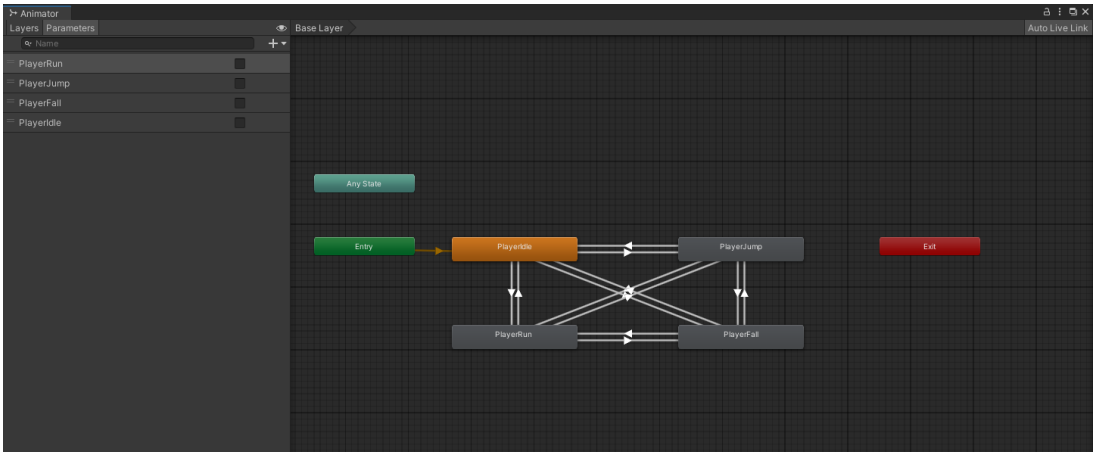


Figure 19: Animator controller with clips and transitions

### 3.1.4. Scripting

Scripting is essential to building any application in Unity. It is used to control many aspects of a game, such as player input, graphical effects, GUI, sound and many more [27]. Standard IDE in Unity is Visual Studio, but it can be manually set to any other editor. In case of a programming language, the default one in Unity is C#, although it will support any .NET language that can compile a compatible DLL [28]. When the script is created the initial skeleton is set and it cen be seen in listing 1.

Listing 1: Default script skeleton

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class ExampleScript : MonoBehaviour
```

```
6  {
7      // Start is called before the first frame update
8      void Start()
9      {
10
11     }
12
13     // Update is called once per frame
14     void Update()
15     {
16
17     }
18 }
```

Scripts in Unity are basically just a custom component that can be added to the game object. Like many other components, its variables can be edited through the inspector by simply making a variable *public* or by adding an annotation *[SerializeField]* above the variable. The annotation method can also be used to hide the public variable in the inspector by annotating the variable with *[HideInInspector]* [29]. Instead of working like a standard program, Unity passes control to the script by calling functions defined in it and regains control when those functions finish their tasks. These functions are called **event functions** [30] and, as the name implies, they are activated by events that happen during runtime. Unity supports many of these functions, but the most important and most used event functions are the following:

- **Update()** - This function is called before each frame is rendered and before the animations are calculated, which is why it is used to change the behavior of game objects in the game (like position, rotation, state, ...).
- **FixedUpdate()** - This function is called before each physics update, and since they occur at a different frequency than frame updates, it is used to write any physics code to make it more accurate.
- **LateUpdate()** - This function is called after the Update and FixedUpdate functions have been called. This is useful for things like moving camera position after all game object transformations have been made.
- **OnGUI()** - This function is used when working with GUI elements in the game.
- **OnCollisionEnter()** - This function is called whenever a collision happens between the object that has the script component and any other game object with a collider. Two different but similar functions exist: **OnCollisionStay()** which is the one that is called repeatedly while the collision is held and **OnCollisionExit()** which is the one that is being called whenever the collision is broken. Alternatively, if the collider is configured as a trigger **OnTriggerEnter()**, **OnTriggerStay()** and **OnTriggerExit()** functions are called.

While talking about scripting it is important to mentions some of the most used and most important built-in classes [31]. These classes are following:

- **GameObject** - This class represents all the objects that can exist in the scene and provides methods to manipulate those objects.
- **MonoBehaviour** - Default base class that every script extends. It provides a framework that allows scripts to be added as components to a GameObject and provides hooks for event functions.
- **Object** - The default class for all objects that Unity can reference in the editor such as GameObjects, Components, Textures, Sprites and others.
- **Transform** - This class provides methods that allow manipulation of GameObject position, scale, and rotation, as well as the hierarchical relationship between parent and child GameObjects.
- **Vectors** - Multiple classes used to express and manipulate 2D, 3D, and 4D points, lines, or directions.
- **Quaternion** - This class represents absolute or relative rotation and has methods that allow their creation and manipulation.ž
- **ScriptableObject** - This class represents a data container that is designed to store large amounts of data.
- **Time** - Allows time manipulation and time measurement, as well as the ability to control the number of frames per second of the project.
- **Mathf** - A class that contains common math functions that are needed in game development.
- **Random** - Provides methods for generating common types of random values.
- **Debug** - Enables visualization of information from scripts in the editor.
- **Gizmos and Handles** - Allows lines and shapes to be drawn in the scene view and game view.

## 3.2. Platformer Games

Platformer games have played a huge role in evolution of games and they are considered to be one of the first game genres [32]. The core of this game genre is a character that is being controlled by the player, which has to avoid obstacles and defeat enemies while trying to progress in game. These games can be divided into two groups: single screen platformers and scrolling platformers [32].

Single screen platformer games are displayed as a single screen per level, with multiple levels in game that progressively become more difficult. On the other hand, scrolling platformers have levels that scroll based upon characters' location and movement. Just like single screen platformers, they have multiple levels that increase in difficulty as the player progresses [32].

Depending on the technology, there are four different types of platformers [32]. First type is pure tile-based platformer. This type of platformer is the easiest one to implement because the character position is limited to a position on a grid [32]. This makes implementing collisions easy, but at the cost of precision when controlling movement. This problem could be solved by investing more time in making good animations to cover it up. Movement is implemented by copying the character to a tile that is adjacent to the current tile and if that tile is not an obstacle, the movement is considered valid and it is rendered. Jumping in this type is limited to only vertical or horizontal movements, so instead of being physics-based, these jumps are purely visual effects.

Second type of platformer is smooth tile-based. This type of platformer is similar to pure tile-based, with the difference being that the character is not bound to the grid [32]. To implement collisions, smooth tile-based platformer uses an axis-aligned bounding box (AABB) which is a rectangle that surrounds the character and is aligned with x, y and z axis of the world. The movement here is calculated separately for each axis.

Third type is bitmask. Bitmask type [32] is similar to smooth tile-based type with the difference being the grid tile size, which is composed of only one pixel. This method uses more memory than the previous two methods, and the levels are made from whole images as opposed to tiles combined into one whole. This method is rarely used in platformers and when it is, it is usually because dynamic environment becomes an option.

Final type of platformer is vectorial. Vectorial platformers [32] use lines and polygons to implement collision. Although, this method is very hard to implement, it is popularity is on the rise because of commercial physics and game engines like Unity.

# 4. Artificial Intelligence

## 4.1. Introduction

Intelligence has been defined in many ways over the years and is a very controversial topic today. To keep it simple, intelligence can be defined as the “ability to understand and adapt to the environment by using inherited abilities and learned knowledge.” [33]

Artificial intelligence (AI) is regarded as one of the most interesting and fastest growing fields today. The main goal of AI as a field is to create machines that can compute how to behave in certain situations effectively and safely [34, str. 29]. There are two dimensions that are used when considering AI [34, str. 31]: human vs. rational and thought vs. behaviour. These dimensions produce a total of four combinations that have been researched separately and using very different methods.

The first approach is acting humanly or better known as "The Turing test approach" [34, str. 32]. The Turing test [35] is a famous test of intelligence proposed by Alan Turing that tests the computer by giving it a set of written questions and seeing whether a human interrogator can determine if the answers were written by a computer or a real person. If the interrogator cannot tell whether the answers were submitted by a person or by a computer, the computer passes the test. This type of test does not really determine if the machine is intelligent or not so to make it a bit more complete and challenging, some researchers proposed a so called total Turing test [35], that requires the machine to interact with objects and people in the real world to pass.

The second approach is thinking humanly or "The cognitive modeling approach" [34, str. 33]. This approach tries to develop a program that not only completes its task successfully, but also compares its "thought process" and steps to a person that is solving the same problem. This approach is closely related to cognitive science and by working together, these fields have been able to develop more rapidly.

The third approach is thinking rationally or "the laws of thought approach" [34, str. 35]. This approach uses logic or irrefutable reasoning processes to develop AI. The example of such a way of thinking was presented by the Greek philosopher Aristotle and it reads [36]: if Socrates is a man and all men are mortal, that would conclude that Socrates is mortal. For a computer to "think" logically, it requires knowledge of the world that is certain which is in reality rarely achievable. The theory of probability fixes this problem in a way by allowing the machine to come to a conclusion based on uncertain information.

The fourth and final approach is acting rationally or "the rational agent approach" [34, str. 36]. An agent is a thing that acts or does some work. Computer agents are a bit different from regular agents in a way that they are required to operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. To expand that even further, a rational agent is the agent that acts to achieve the best outcome or, when the outcome is uncertain, the best expected outcome.

## 4.2. Machine Learning

Learning is a skill of improving performance based on observations about the world [37]. When the learning agent is a computer, it is called machine learning. Machine learning is a subfield of AI that studies the ability to improve performance based on experience [34, str. 31]. To learn something, a computer first collects data, then builds a model based on that data and uses the model as a hypothesis of the world and as a software that can solve problems [34, str. 1201].

There are three types of feedback that determine the main types of machine learning. The first type is **supervised learning** which is the type of learning that observes input-output pairs and creates a function that would give a certain output for a specific input [34, str. 1205]. It is used when there is a pre-established and labeled data that can be used for learning [38, str. 8]. An example of this is image recognition, where the computer first learns about an object on multiple images and then can predict that object on a different image.

The second type is **unsupervised learning** which is a type of learning that learns from the input, but without an explicit feedback [34, str. 1206]. This type attempts to discover associations of inputs by searching for patterns without having access to a target output [39, str. 77].

The third type of learning is **reinforcement learning** which, as the name implies, learns from reinforcement (rewards or punishments) [34, str. 1206]. The algorithm gets its feedback from the environment based on how it is interacting with it, meaning that the agent will be in a specific state at a specific time and decide to take action from all the available actions at that state, for which it will receive an appropriate reward from the environment [39, str. 71].

### 4.2.1. Neural Networks

Neural networks are a machine learning algorithm that fall under the category of supervised learning. It is an approach to AI and ML that is based on the model of a biological brain. It consists of a set of interconnected processor units called neurons [39, str. 59]. The artificial neuron resembles its biological counterpart in a way that it has a number of **inputs (x)** that all have their own **weight (w)** [39, str. 59]. The neuron also has a processing unit that calculates a weighted sum of all the inputs and adds a bias **weight (b)** to that. The calculated value is then fed to an **activation function (g)** that then calculates the output of the neuron [39, str. 59]. Neural networks that have two or more hidden layers are called **deep neural networks** [38, str. 7] and from this terminology arises the name **deep learning** [39, str. 65]. Figure 20 shows a visual representation of the artificial neuron.

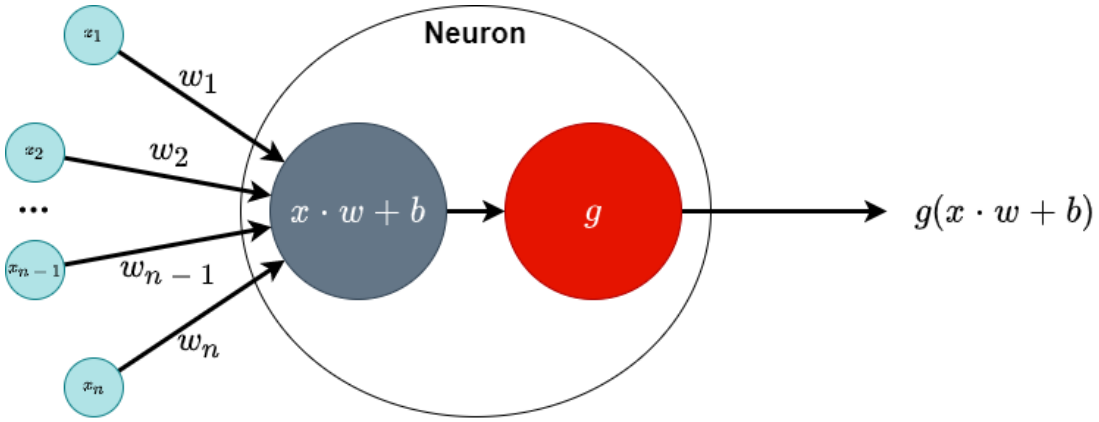


Figure 20: Artificial neuron model; based on [39]

The original neuron model [39, str. 60] had a Heaviside step activation function whose purpose is to determine if the neuron should fire or not. The limitation of this function is that, when connected to a network, the neurons could only calculate linear problems. That is where the **backpropagation** algorithm [39, str. 60] comes into play. The invention of this algorithm allowed a neural network to solve nonlinear problems. Today, there are several different activation functions that, when used, yield a different type of a neural network.

To construct a neural network, a number of neurons need to be connected. The most common structure used is the **multi-layer perceptron** [39, str. 60] (MLP) which structures its neurons in multiple layers in such a way that no neurons on the same layer are interconnected. Each neuron has its output connected to all the neurons on the next layer, becoming their input. This continues until the last layer, whose outputs become the outputs of the entire neural network, which is why this layer is also called the **output layer**. Accordingly, the first layer is called the **input layer**, and all layers in between are called **hidden layers** [39, str. 60, 61]. Figure 21 shows an example of an MLP network.

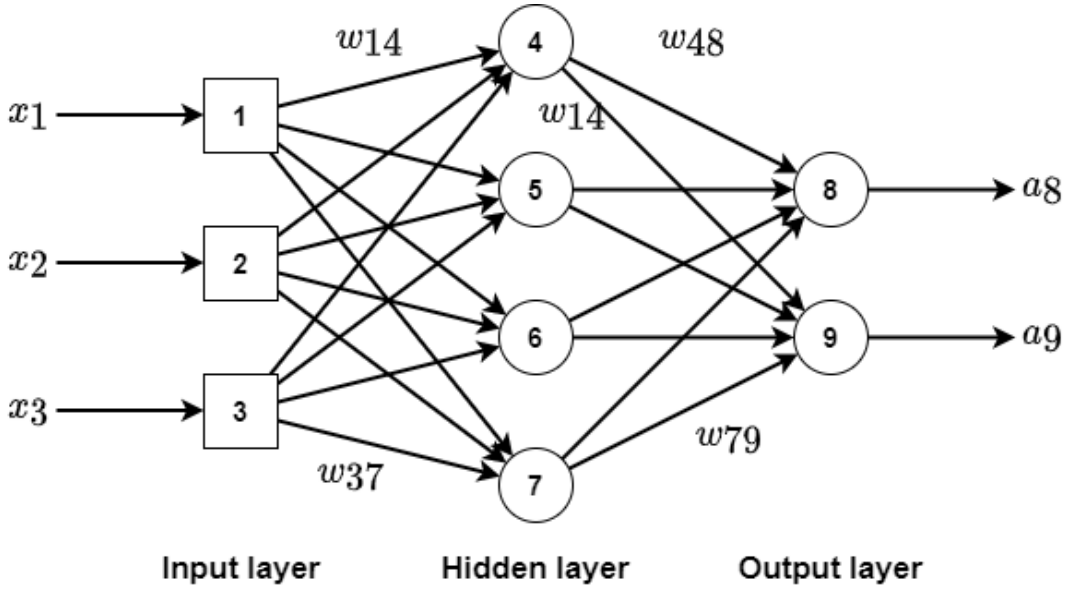


Figure 21: MLP network example; based on [39]

To calculate the output of the network based on a given input, a process called **forward operation** is applied, which sends inputs through the layers of the network with the ultimate goal of generating outputs.

" The steps of this process are as follows:

1. Label and order neurons.We typically start numbering at the input layer and increment the numbers towards the output layer. Note that the input layer does not contain neurons, nevertheless is treated as such for numbering purposes only.
2. Label connection weights assuming that $w_{ij}$ is the connection weight from neuron $i$ (pre-synaptic neuron) to neuron $j$ (post-synaptic neuron). Label bias weights that connect to neuron $j$ as $b_j$.
3. Present an input pattern $x$.
4. For each neuron $j$ compute its output as follows: $a_j = g(\sum_i \{w_{ij}a_i\} + b_j)$, where $a_j$ and $a_i$ are , respectively, the output of and the inputs to neuron $j$ (n.b. $a_i = x_i$ in the input layer); $g$ is the activation function (usually the logistic sigmoid function).
5. The outputs of the neurons of the output layer are the outputs of the ANN.

" [39, str. 61, 62]

To "teach" the neural network to return the desired outputs based on the input, the training algorithm needs to be implemented [39, str. 62]. The training algorithm adjusts the weights (**w** and **b**) so that function $f(x; w, b)$ matches the input dataset $y$, or $f : x \to y$.

The training algorithm requires a **cost (error) function** [39, str. 62] which is used to determine the quality of any set of weights. The most common perfomance testing function for training neural network is the squared Euclidean distance (error) between the vectors of the output of the neural network ($a$) and the desired outputs from the dataset ($y$).

$$E = \frac{1}{2} \sum_j (y_j - a_j)^2$$

To calculate weight updates that minimize this error function, the most common algorithm used is the **backpropagation** [39, str. 63] or *backward propagation of errors*. This algorithm calculates the partial derivative (gradient) of the function $E$ with respect to each weight of the neural network and adjusts the weights of the neural network following the calculated gradient.

As the Euclidian error [39, str. 63] depends on the weights of the neural network, gradient of E can be calculated with respect to any weight ($\frac{\theta E}{\theta w_{ij}}$) and any bias weight ($\frac{\theta E}{\theta b_j}$), which will determine the of error change if the weight values are changed. To determine how much of that change will be implemented, a parameter $\eta \in [0, 1]$ called **learning rate** [39, str. 63] is used.

“ The basic steps of the backpropagation algorithm are as follows:

1. Initialize **w** and **b** to random (commonly small) values.
2. For each training pattern (input-output pair):
   (a) Present input pattern **x**, ideally normalized to a range (e.g., [0;1]).
   (b) Compute ANN actual outputs $a_j$ using the forward operation.
   (c) Compute E according to its formula.
   (d) Compute error derivatives with respect to each weight $\frac{\theta E}{\theta w_{ij}}$ and bias weight $\frac{\theta E}{\theta b_j}$ of the ANN from the output to the input layer.
   (e) Update weights and bias weights as $\Delta w_{ij} = -\eta \frac{\theta E}{\theta w_{ij}}$ and $\Delta b_j = -\eta \frac{\theta E}{\theta b_j}$ respectively.
3. If E is small or you are out of computational budget, stop! Otherwise go to step 2.

” [39, str. 63]

The problem with backpropagation is that it is not guarantied to find the global minimum of the $E$ because of the possibility of the plateaux areas the error function landscape which has a near zero gradient which in turn results in near zero weight updates. There are a few solutions to this problem [39, str. 64]:

- Random restarts: Rerunning the algorithm with new weight values. This is not good for networks that are luck based.
- Dynamic learning rate: Modifying the learning rate parameter or creating a dynamic learning parameter that increases when convergence is slow or decreases when convergence is fast.
- Momentum: Adding a momentum amount to the weight update rule as follows:

  $$\Delta w_{ij}^{(t)} = m \Delta w_{ij}^{(t-1)} - \eta \frac{\theta E}{\theta w_{ij}}$$

  where $m \in [0, 1]$ is the momentum parameter and $t$ is the iteration of the weight update.

The backpropagation can be implemented in two different learning modes, batch and non-batch mode [39, str. 64]. Non-batch mode updates the weights every time the a training sample is presented, which makes it unstable, but it can also be useful to avoid local minimum convergence. Batch mode updates the weights only after all training samples are presented. This feature makes it more stable than its counterpart, but there is a problem with the local

minimum convergence. To best utilize the better qualities of both approaches, it is common to apply batch learning of randomly selected samples in smaller sizes.

Neural networks in the gaming industry are most commonly used for path navigation due to their ability to adapt and learn [40]. For example, it can track the player's movement and adjust its behaviour accordingly making the game more challenging with time. To put it simply, it allows the game to adapt to the player making their decisions influence the environment.

### 4.2.2. Support Vector Machines

Support vector machine [39, str. 66] (SVMs) is a supervised learning algorithm that is trained to maximize the margin between the training examples of separate classes. By introducing the attributes of new unseen example, the algorithm tries to predict which class it belongs to. It is mostly used for text categorization, speech recognition, image classification, hand-written character recognition and other similar areas.

Similarly to neural networks, this algorithm defines a function $f$ that maps between input and target outputs, but instead of trying to minimize the difference between the actual output and the desired output, SVMs construct a hyperplane that maintains the largest distance between the nearest data point of any other class, called a **maximum-margin** [39, str. 66]. That margin divides the points of a class with label 1 from those with label -1 in a dataset, so the distance between the derived hyperplane and the nearest point from eather class is maximized. The hyperplane can be defined mathematically as $w \cdot x - b = 0$, where $w$ represents the weight vector, $x$ the input attributes and $\frac{b}{||w||}$ determines the offset or the weight bias. The job of a SVM is to predict the output while trying to:

- minimize $||w||$,
- subject to $y_i(w \cdot x_i - b) \geq 1$, for $i = 1, ..., n$

the above problem is solvable only if the data points are linearly separable, which is also called a **hard-margin** [39, str. 66], and if the data is not linearly separable, also called a **soft-margin** [39, str. 66], the algorithm attempts to:

- minimize $[\frac{1}{n} \sum_{i=1}^{n} max(0, 1 - y_i(w \cdot x_i - b))] + \lambda ||w||^2$

This approach is more efficient in finding solutions for large, but sparse, datatsets. It is also efficient in dealing with large feature spaces because the task complexity does not depend on the dimensionality of the feature space [39].

### 4.2.3. Decision Tree Learning

Decision tree learning [39, str. 68] is another supervised learning algorithm which, as the name implies, uses decision tree representation to map the data to its target value. The

way it is set up is that the inputs are represented as nodes and the outputs are represented as the leaves of the tree. All possible values of the inputs are represented as branches that originate in that node. The example of the decision tree can be seen in the figure 22.

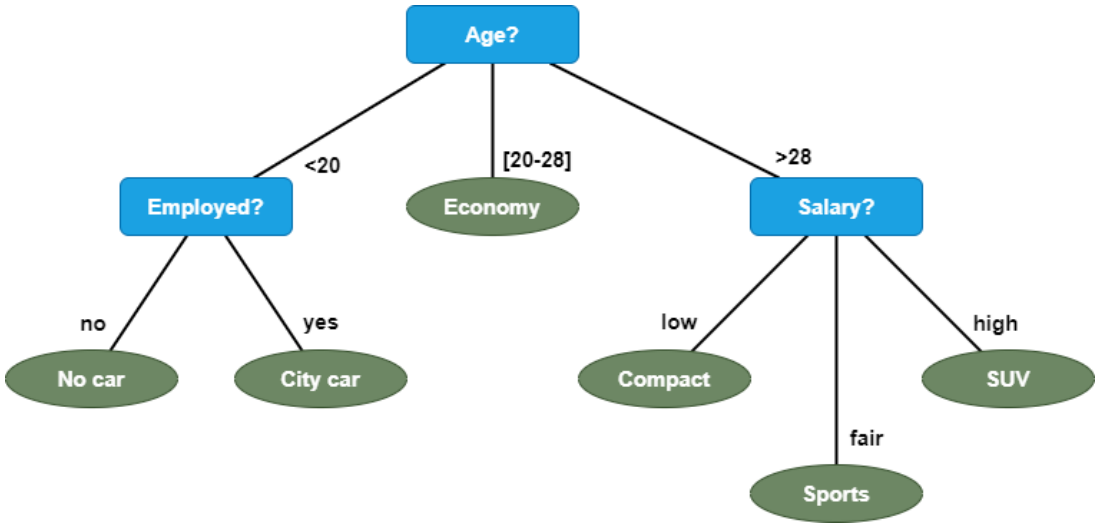


Figure 22: Decision tree; based on [39]

The decision tree learning works in a way that it constructs a tree model that predicts the value of target outputs based on a number of inputs. To construct a decision tree, the dataset is split into subsets based on the selections made for the attributes of the dataset. This process is repeated until the tree is fully constructed.

### 4.2.4. Q-learning

Q-learning [39, str. 74] falls in the category of reinforcement learning. It is a model free learning algorithm that relies on tabular representation of $Q(s, a)$ values, where it got its name from. $Q(s, a)$ is a representation of how good the action $a$ is in state $s$. This algorithm learns from experience, meaning it picks actions and receives rewards based on those actions. Its goal is to maximise the expected reward in each state by picking the right action. The reward in question is a weighted sum of the expected values of the discounted future rewards. Q-learning works in a way that it updates the Q values, that are initially set by the designer, in a iterative fashion.

> " Each time the agent selects an action $a$ from state $s$, it visits state $s'$, it receives an immediate reward $r$, and updates $Q(s, a)$ value as follows:
>
> $$Q(s, a) \leftarrow Q(s, a) + \alpha\{r + \gamma max Q(s', a') - Q(s, a)\}$$
>
> where $\alpha \in [0, 1]$ is the learning rate and $\gamma \in [0, 1]$ is the discount factor. The learning rate determines the extent to which the new estimate for $Q$ will override the old estimate. The discount factor weights the importance of earlier versus later rewards; the closer $\gamma$ is to 1, the greater the weight is given to future reinforcements. " [39, str. 74]

Since it is represented in a tabular form [39, str. 75], Q-learning has some limitations with the size of the table and memory requirements. Also, since the learning is exponential to the size of the table, it could also take a lot of time to process information. To bypass this limitation, artificial neural networks are used to approximate the Q-value function.

### 4.2.5. Clustering

Clustering [39, str. 77] is the algorithm that falls under the category of unsupervised learning. Its task is to find unknown groups of data so that data within a group (**cluster**) is similar to each other but different from the data in other clusters. Clustering is important for games, as it is used player modeling, game play and content generation [39, str. 77].

This algorithm places data into classes for which the labels are unknown prior to running the algorithm and are discovered iteratively. The quality of generated clusters [39, str. 77] is determined using two properties:

1. high intra-cluster similarity or high compactness
2. low inter-cluster similarity or good separation

Clustering can be realized as a number of different algorithms [39, str. 78], like hierarchical clustering, k-means , k-medoids, DBSCAN and self-organizing maps. These algorithms are different in a way they define a cluster and form it. Every algorithm has its own purpose and selecting a proper one for a specific task is essential.

K-means is considered the most popular clustering algorithm as it has a good balance between simplicity and effectiveness. It works as follows:

" Given $k$:

1. Randomly partition the data points into $k$ nonempty clusters.
2. Compute the position of the centroids of the clusters of the current partitioning. Centroids are the centers (mean points) of the clusters.
3. Assign each data point to the cluster with the nearest centroid.
4. Stop when the assignment does not change; otherwise go to step 2.

" [39, str. 78]

Although it is simple and effective, k-means has its weaknesses [39, str. 78]. It can be used only on data in a continuous space, number of clusters, $k$, needs to be defined in advance, it can only find hyper-spherical clusters and it is sensitive to outliers so extreme values can distort the distribution of the data and affect the performance of the algorithm. This is where hierarchical clustering comes into play.

Hierarchical clustering [39, str. 79] is the algorithm that attempts to build a hierarchy of clusters. It can be realised by two strategies [39, str. 79]:

- agglomerative - constructs hierarchies from the bottom up by gradually merging data points together.
- divisive - constructs top-down hierarchies by splitting the data set.

This method does not require the explicit number of clusters defined beforehand, but it needs a terminating condition [39, str. 79]. The steps of the agglomerative clustering algorithm are as follows:

" Given $k$:

1. Create one cluster per data sample.
2. Find the two closest data samples—i.e., find the shortest Euclidean distance between two points (single link)—which are not in the same cluster.
3. Merge the clusters containing these two samples.
4. Stop if there are $k$ clusters; otherwise go to step 2.

" [39, str. 79]

The divisive approach works in a way that the data is all in the same cluster by default, and is split until every data has its own cluster which is defined by a split strategy [39, str. 79].

Once created, clusters can be visually represented as a tree diagram called dendrogram [39, str. 79].

### 4.2.6. Frequent Pattern Mining

Frequent pattern mining [39, str. 80], as the name implies, is a set of techniques that try to find frequent patterns and structures in provided data and falls under the category of unsupervised learning. There are a few different types of frequent pattern mining, but only two are being used in game AI. First one is frequent itemset mining [39, str. 80], which tries to find structure in data that has no particular internal order, and the second one is frequent sequence mining [39, str. 80], which tires to find structure in data based on its inherent temporal order.

One of the more popular algorithms for itemset mining is apriori [39, str. 80]. It is used for mining datasets that contain sets of instances (transactions) each of which contains a set of items, or an itemset. It is a very simple algorithm and it can be described as follows: given a predetermined threshold named support ($T$), the algorithm detects the itemsets that are subsets of at least $T$ transactions.

General sequential patterns (GSP) [39, str. 80] is one of the more popular sequence mining algorithms. It tries to find frequently occurring subsequences in a sequence or a set of sequences or, more formally, in a dataset that contains samples that are sequences of events (data sequence), a sequential pattern, that is defined as a subsequence of events, is a frequent sequence if it occurs in samples of data regularly.

# 5. Practical Example

This chapter describes the steps involved in developing a platform game in Unity as well, as developing and connecting the "brain" that will learn to play said game. First step is to actually develop a game, followed by making a machine learning model that will play it.

Parts of the code used in this example, particularly for the player controller and the neural network, are based on pre-existing code from [41] and [42].

## 5.1. Making a Game

This section shows the steps in creating a simple platform game that was developed from scratch. An overview of the finished game can be seen in figure 23.

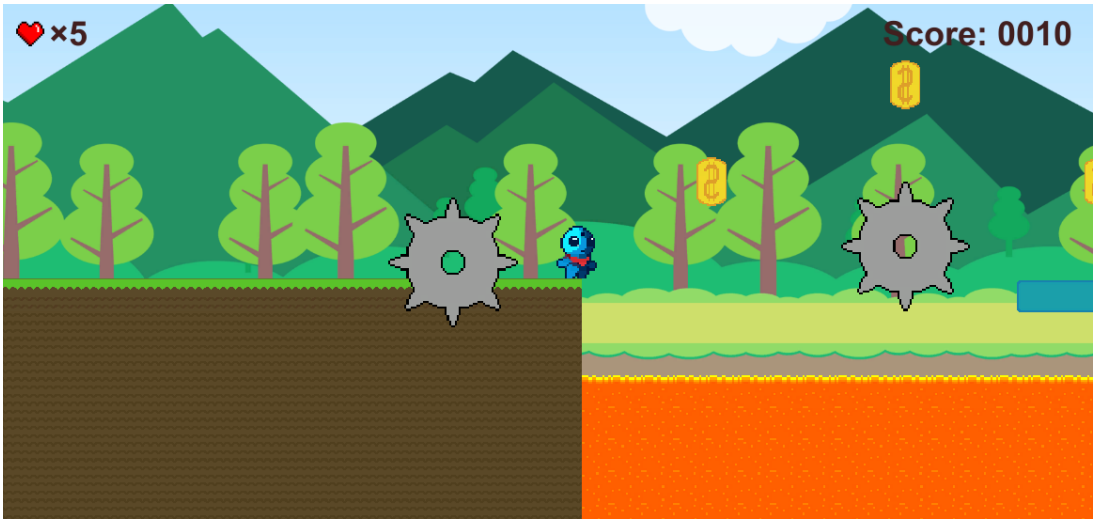


Figure 23: Preview of the finished game

### 5.1.1. Implementing the Player

For the purposes of this thesis, the physics and collisions, along with several other components, are developed from scratch as opposed to using Unity's built-in components as it is a more flexible and reliable way of handling player movement, assuming it is implemented correctly.

First step is to implement a sort of distance detection for the player. Ideal way to do this is to use raycast. Rays originating from the player would need to be slightly recessed into the player so they would never measure the distance of zero. Initially, the player is a simple square and the sprite is added later. To create the desired behaviour, a script is created and later added to the player. The controller script is created separately and is linked to the player script as a required component. To know the bounds of the player sprite, a BoxCollider2D is a required component in the controller.

The first script in line is the *Player* script. This script handles the user inputs, sends the

movement information to the *Controller* script, plays the appropriate animation for a specific movement and toggles the menu popups. The full script can be seen in listing 2:

Listing 2: Player script: based on [41]

```
[RequireComponent(typeof(Controller))]

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Controller))]
public class Player : MonoBehaviour
{
    public GUIController gui;
    public GameObject checkpoint;
    public int lives = 10;
    public int score = 0;

    public float jumpSpeed = 5;
    public float movementSpeed = 5;
    public float gravity = -5;

    float currentJumpSpeed;
    float currentMovementSpeed;
    float currentGravity;

    bool menuOpen = false;

    Vector2 playerMovement;
    Controller controller;

    SpriteRenderer sprite;
    Animator animator;

    void Start()
    {
        animator = GetComponent<Animator>();
        sprite = GetComponent<SpriteRenderer>();
        controller = GetComponent<Controller>();
        updateGuiLives();
        resetToCheckpoint();
        ResetPlayerMovement();
    }

    private void Update()
    {
        if (controller.collisions.top || controller.collisions.bottom)
        {
            playerMovement.y = 0;
        }

        if (Input.GetKey(KeyCode.Space) && controller.collisions.bottom)
```

```
50          {
51              playerMovement.y = currentJumpSpeed;
52          }
53
54          if (Input.GetKeyDown(KeyCode.Escape))
55          {
56              if (!menuOpen)
57              {
58                  gui.ToggleMenu();
59                  TogglePlayerMovement();
60              }
61          }
62
63          Vector2 movementDirection = new Vector2(Input.GetAxisRaw("Horizontal"),
             Input.GetAxisRaw("Vertical"));
64
65          if (movementDirection.x == -1) sprite.flipX = true;
66          if (movementDirection.x == 1) sprite.flipX = false;
67
68          if (movementDirection.x != 0) animator.SetBool("PlayerRun", true);
69          else animator.SetBool("PlayerRun", false);
70
71          if (!controller.collisions.bottom)
72          {
73              animator.SetBool("PlayerIdle", false);
74
75              if (playerMovement.y > 0) animator.SetBool("PlayerJump", true);
76              else animator.SetBool("PlayerJump", false);
77
78              if (playerMovement.y < 0) animator.SetBool("PlayerFall", true);
79              else animator.SetBool("PlayerFall", false);
80          }
81          else
82          {
83              animator.SetBool("PlayerIdle", true);
84              animator.SetBool("PlayerFall", false);
85              animator.SetBool("PlayerJump", false);
86          }
87
88          playerMovement.x = movementDirection.x * currentMovementSpeed;
89          playerMovement.y += currentGravity * Time.deltaTime;
90          controller.Move(playerMovement * Time.deltaTime);
91      }
92
93      internal void resetToCheckpoint()
94      {
95          transform.position = checkpoint.transform.position;
96      }
97
98      internal void incrementScore(int increment)
99      {
100         score += increment;
101         gui.SetScoreText(score);
```

```
102     }
103
104     internal void removeLife()
105     {
106         lives--;
107         resetToCheckpoint();
108         updateGuiLives();
109         if (lives == 0)
110         {
111             showGameOverDialog();
112         }
113     }
114     private void updateGuiLives()
115     {
116         gui.SetLivesText(lives);
117     }
118
119     internal void levelComplete()
120     {
121         if (!menuOpen)
122         {
123             menuOpen = true;
124             gui.completeLevel();
125             TogglePlayerMovement();
126         }
127     }
128
129     private void showGameOverDialog()
130     {
131         if (!menuOpen)
132         {
133             menuOpen = true;
134             gui.GameOver();
135             TogglePlayerMovement();
136         }
137     }
138
139     private void TogglePlayerMovement()
140     {
141         if (currentGravity != 0) FrezePlayer();
142         else ResetPlayerMovement();
143     }
144
145     private void FrezePlayer()
146     {
147         currentGravity = currentJumpSpeed = currentMovementSpeed = 0;
148         playerMovement = Vector2.zero;
149     }
150
151     private void ResetPlayerMovement()
152     {
153         currentGravity = gravity;
154         currentJumpSpeed = jumpSpeed;
```

```
155             currentMovementSpeed = movementSpeed;
156         }
157     }
```

Next up is the *RaycastController* script which, as the name implies, implements raycasting to the game object it is attached to. Its job is to calculate the starting point of each ray depending on the ray distance variable that is previously defined. Since it is defined that the BoxCollider2D is a required component, the corners are found using the *bounds* function. Listing 3 shows the raycast controller script and the result of this code can be seen in figure 24. The rays are shown in the scene view using the *Debug.DrawRay* function.

Listing 3: Raycast controller: based on [41]

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   [RequireComponent(typeof(BoxCollider2D))]
6   public class RaycastController : MonoBehaviour
7   {
8       public LayerMask collisionMask;
9       public const float rayDistance = 0.2f;
10      public const float rayOffset = 0.02f;
11
12      [HideInInspector]
13      public int horizontalRays;
14      [HideInInspector]
15      public int verticalRays;
16      [HideInInspector]
17      public BoxCollider2D collider2d;
18      [HideInInspector]
19      public float horizontalRayDistance;
20      [HideInInspector]
21      public float verticalRayDistance;
22      [HideInInspector]
23      public RayOrngins rayOrigins;
24
25      public virtual void Awake()
26      {
27          collider2d = GetComponent<BoxCollider2D>();
28      }
29
30      public virtual void Start()
31      {
32          calculateRayDistance();
33      }
34
35      public void calculateRayDistance()
36      {
37          Bounds bounds = collider2d.bounds;
38          bounds.Expand(rayOffset * -2);
39
40          horizontalRays = Mathf.RoundToInt(bounds.size.y / rayDistance);
```

```
41          verticalRays = Mathf.RoundToInt(bounds.size.x / rayDistance);
42
43          horizontalRayDistance = bounds.size.y / (horizontalRays - 1);
44          verticalRayDistance = bounds.size.x / (verticalRays - 1);
45      }
46
47      public void updateRayOrigins()
48      {
49          Bounds bounds = collider2d.bounds;
50          bounds.Expand(rayOffset * -2);
51
52          rayOrigins.tl = new Vector2(bounds.min.x, bounds.max.y);
53          rayOrigins.tr = new Vector2(bounds.max.x, bounds.max.y);
54          rayOrigins.bl = new Vector2(bounds.min.x, bounds.min.y);
55          rayOrigins.br = new Vector2(bounds.max.x, bounds.min.y);
56      }
57
58      public struct RayOrngins
59      {
60          public Vector2 tl, tr, bl, br;
61      }
62  }
```

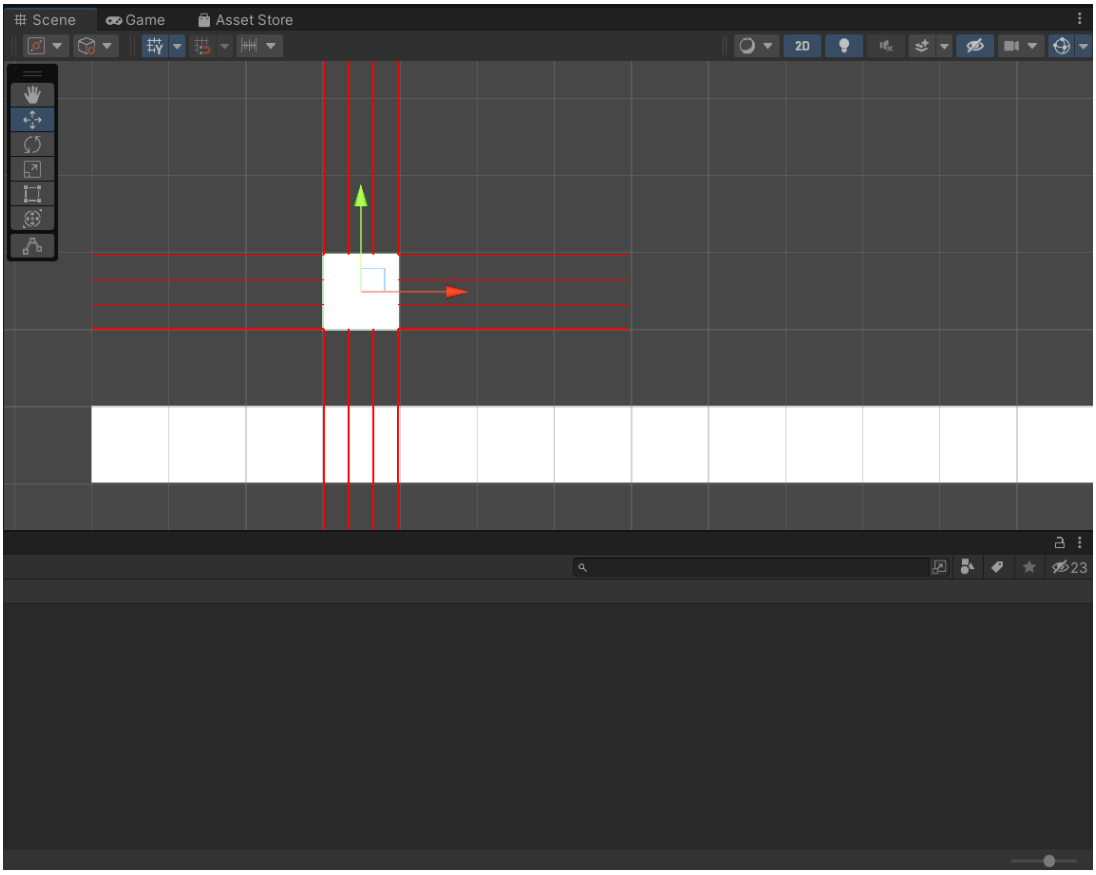


Figure 24: Result of the controller code above

To actually move the character, the controller script needs to be implemented and it

has to extend the raycast controller script to add a method for moving the player. This scripts job is to take the movement information passed form the player script and move the player if there are no obstacles in the way, which is determined using the functions defined in the raycast controller script. Every time the *Move* function is called, the ray origins (where the first ray in each direction is positioned) are calculated and new rays are cast depending on the ray distance defined previously. If the player touches the obstacle on any side, the movement in that direction is disabled. The *Controller* script can be seen in listing 4:

Listing 4: Player controller script: based on [41]

```
1
2  using System;
3  using System.Collections;
4  using System.Collections.Generic;
5  using UnityEngine;
6
7
8  public class Controller : RaycastController
9  {
10     public Collisions collisions;
11
12     public override void Start()
13     {
14         base.Start();
15
16     }
17     public void Move(Vector2 movement)
18     {
19         updateRayOrigins();
20         collisions.Reset();
21
22         if (movement.x != 0)
23         {
24             calculateHorizontalCollisions(ref movement);
25         }
26         if (movement.y != 0)
27         {
28             calculateVerticalCollisions(ref movement);
29         }
30
31         transform.Translate(movement);
32     }
33
34     private void calculateHorizontalCollisions(ref Vector2 movement)
35     {
36         float direction = Mathf.Sign(movement.x);
37         float rayLength = Mathf.Abs(movement.x) + rayOffset;
38
39         for (int i = 0; i < horizontalRays; i++)
40         {
41             Vector2 rayOrigin = (direction == -1) ? rayOrigins.bl : rayOrigins.br;
42
```

```
            rayOrigin += Vector2.up * (horizontalRayDistance * i);
            RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.right *
             direction, rayLength, collisionMask);

            if (hit)
            {
                movement.x = (hit.distance - rayOffset) * direction;
                rayLength = hit.distance;

                collisions.left = direction == -1;
                collisions.right = direction == 1;
            }
        }
    }
    private void calculateVerticalCollisions(ref Vector2 movement)
    {
        float direction = Mathf.Sign(movement.y);
        float rayLength = Mathf.Abs(movement.y) + rayOffset;

        for (int i = 0; i < verticalRays; i++)
        {
            Vector2 rayOrigin = (direction == -1) ? rayOrigins.bl : rayOrigins.tl;

            rayOrigin += Vector2.right * (verticalRayDistance * i + movement.x);
            RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.up * direction,
             rayLength, collisionMask);

            if (hit)
            {
                movement.y = (hit.distance - rayOffset) * direction;
                rayLength = hit.distance;

                collisions.bottom = direction == -1;
                collisions.top = direction == 1;
            }
        }
    }



    public struct Collisions
    {
        public bool top, bottom, left, right;

        public void Reset()
        {
            top = bottom = left = right = false;
        }
    }
}
```

The final step in creating a player is to move the camera along with the player. Instead of nesting a camera inside the player object to move with it, a small delay is added to prevent

sudden camera movements. This is achieved by placing a player inside a rectangle border (camera window [43]) and when the player "pushes" onto a certain side of that border, the camera moves in that direction. To put it simply, the player can freely move inside the border without the camera moving. The script that controls this behaviour can be seen in listing 5, and the visual example can be seen in figure 25.

Listing 5: Camera controller script: based on [41]

```
1
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class CameraController : MonoBehaviour
7  {
8      public Controller player;
9      public Vector2 focusAreaSize;
10
11     public float verticalOffset;
12
13     FocusArea focusArea;
14
15     void Start()
16     {
17         focusArea = new FocusArea(player.collider2d.bounds, focusAreaSize);
18     }
19
20     private void LateUpdate()
21     {
22         focusArea.Update(player.collider2d.bounds);
23
24         Vector2 focusPosition = focusArea.center + Vector2.up * verticalOffset;
25
26         transform.position = (Vector3)focusPosition + Vector3.forward * -10;
27     }
28
29     private void OnDrawGizmos()
30     {
31         Gizmos.color = new Color(1, 0, 0, .5f);
32         Gizmos.DrawCube(focusArea.center, focusAreaSize);
33     }
34
35     struct FocusArea
36     {
37         public Vector2 center;
38         public Vector2 cameraMove;
39         float top, bottom, left, right;
40
41         public FocusArea(Bounds playerBounds, Vector2 size)
42         {
43             top = playerBounds.min.y + size.y;
44             bottom = playerBounds.min.y;
45             left = playerBounds.center.x - size.x / 2;
```

```
46              right = playerBounds.center.x + size.x / 2;
47
48              cameraMove = Vector2.zero;
49              center = new Vector2((left + right) / 2, (top + bottom) / 2);
50          }
51
52          public void Update(Bounds playerBounds)
53          {
54              float moveOnX = 0;
55              float moveOnY = 0;
56
57              if (playerBounds.min.x < left)
58              {
59                  moveOnX = playerBounds.min.x - left;
60              }
61              else if (playerBounds.max.x > right)
62              {
63                  moveOnX = playerBounds.max.x - right;
64              }
65
66              if (playerBounds.min.y < bottom)
67              {
68                  moveOnY = playerBounds.min.y - bottom;
69              }
70              else if (playerBounds.max.y > top)
71              {
72                  moveOnY = playerBounds.max.y - top;
73              }
74
75              top += moveOnY;
76              bottom += moveOnY;
77              left += moveOnX;
78              right += moveOnX;
79
80              center = new Vector2((left + right) / 2, (top + bottom) / 2);
81              cameraMove = new Vector2(moveOnX, moveOnY);
82          }
83      }
84  }
```

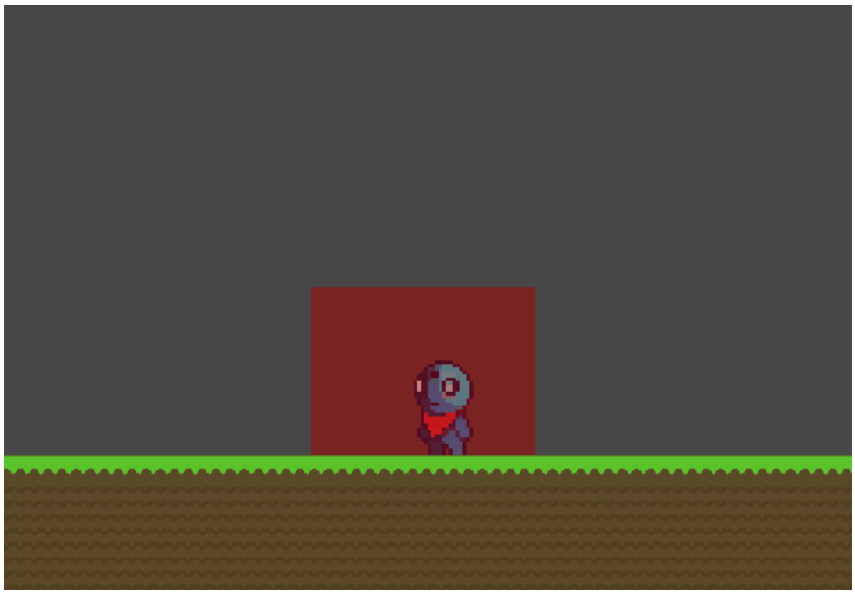

Figure 25: Camera moving border

Finally, the player sprite is added and animated. Since it takes a long time to draw movement for the player sprite and it is not the main focus of the thesis, the player sprites are downloaded from ctaftpix website and can be seen in figures 26 27 28.

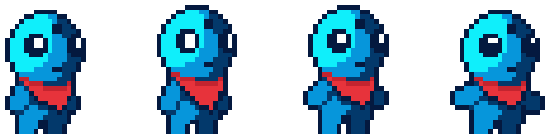

Figure 26: Idle player sprite, downloaded from [44]

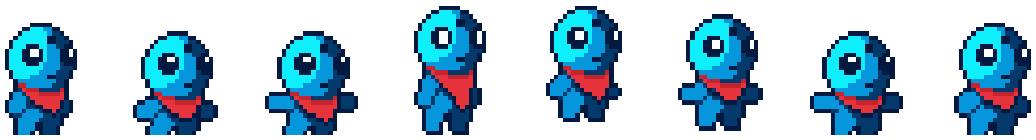

Figure 27: Jump player sprite, downloaded from [44]

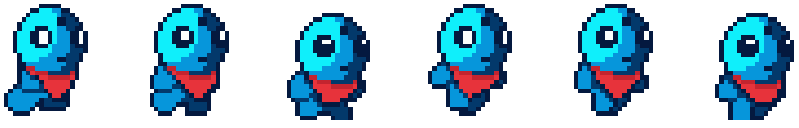

Figure 28: Run player sprite, downloaded from [44]

### 5.1.2. Creating the Environment

The level is build using a rectangular tilemap. The created sprites are inserted into the tile pallete window, which is used to "draw" the selected sprite on the tilemap grid. "2D tilemap extras" package [45] is used to make the animated sprites for the tilemap. Creation of one

animated tile can be seen on figure 29. The previously sliced image can then be inserted in the animation slots in the inspector window in the desired order, after which the speed of the animation is defined along with the collider type for the tile. The example of animating the tile can be seen in figure 30.

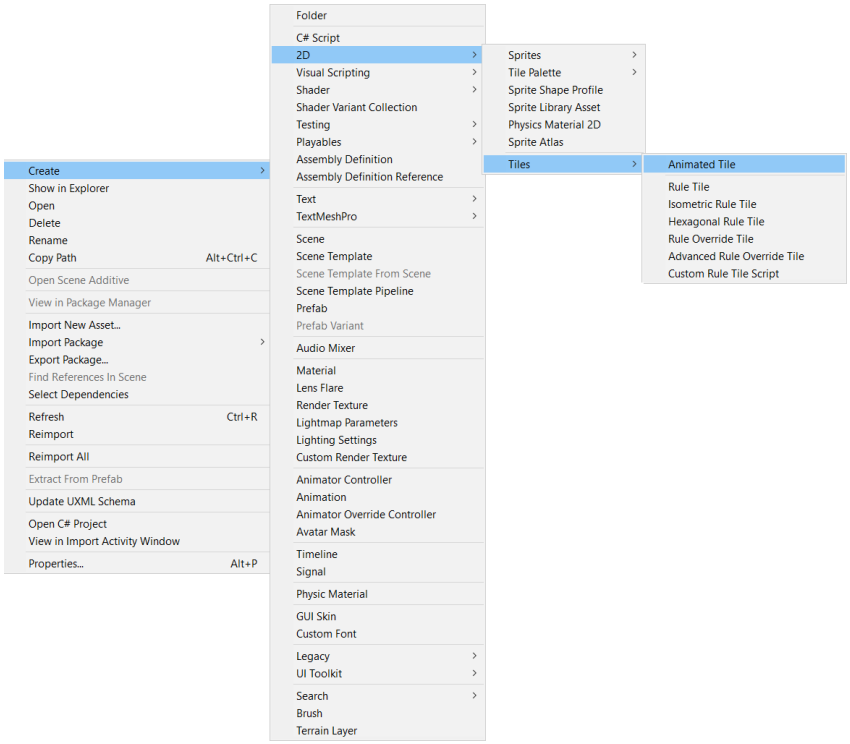


Figure 29: Creating an animated tile


Figure 30: Example of a tile animation

After creating all the elements in the pallete, the level can be easily painted by using the tile pallete tools. The finished product can be seen in figure 31.

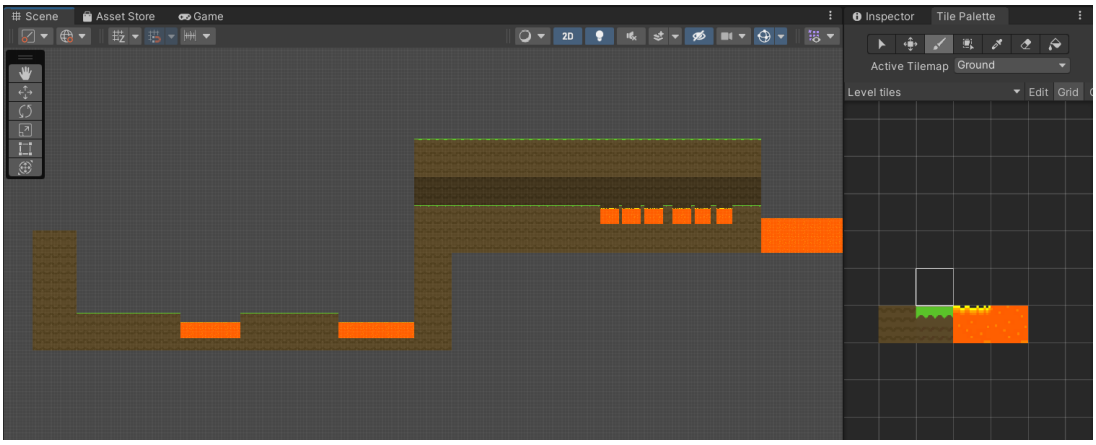


Figure 31: Finished level layout

To add a bit more difficulty to the game, another obstacle is introduced to be placed above ground. The script that controls those obstacles and lava is very simple. All it does is detect if another collider contacted the current collider. It then checks if that collider is a player collider and if it is, sends the kill information to that player. The script can be seen in listing 6:

Listing 6: Player kill script

```
1
2 using System.Collections;
```

```
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class PlayerKill : MonoBehaviour
7  {
8      Player player;
9
10     private void Start()
11     {
12
13     }
14
15     void OnCollisionEnter2D(Collision2D collision)
16     {
17         if(collision.gameObject.layer == 3)
18         {
19             player = (Player)collision.gameObject.GetComponent("Player");
20             player.removeLife();
21         }
22     }
23 }
```

After that, the reward system is added in a form of coin pickups. The script that controls the coins is similar to the killing script in a way that it detects the collider collision and if that collider is player it rewards adds points to that player and disables itself from the scene. The script that controls this behaviour can be seen in listing 7:

Listing 7: Script for adding points to player

```
1
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class AddScore : MonoBehaviour
7  {
8      private int scoreIncrement = 10;
9      Player player;
10
11     private void Start()
12     {
13
14     }
15
16     private void OnCollisionEnter2D(Collision2D collision)
17     {
18         if (collision.gameObject.layer == 3)
19         {
20             player = (Player)collision.gameObject.GetComponent("Player");
21             player.incrementScore(scoreIncrement);
22             gameObject.SetActive(false);
23         }
24     }
25 }
```

To cross the created lava lakes, the moving platforms are added to the level. The script that controls their behaviour also extends the raycast controller script, because to properly carry the player, the platform needs to determine if the player is actually on board. It moves at a calculated speed between two points located in the scene, and if the player is onboard it moves the player in the same direction and at the same speed. The script that handles this behaviour can be seen in listing 8:

Listing 8: Moving platform controller: based on [41]

```
1
2  using System;
3  using System.Collections;
4  using System.Collections.Generic;
5  using UnityEngine;
6
7  public class MovingPlatformsController : RaycastController
8  {
9      public LayerMask passengerMask;
10     public Vector2 move;
11     public float moveSpeed = 10;
12
13     public GameObject startPosition;
14     public GameObject endPosition;
15
16     GameObject currentTarget;
```

```
17
18      public override void Start()
19      {
20          base.Start();
21          transform.position = startPosition.transform.position;
22          currentTarget = endPosition;
23      }
24
25      void Update()
26      {
27          updateRayOrigins();
28
29          CheckMoveArea();
30          Vector2 velocity = (currentTarget.transform.position - transform.position).
             normalized * moveSpeed * Time.deltaTime;
31
32          MovePassengers(velocity);
33          transform.Translate(velocity);
34      }
35
36      private void CheckMoveArea()
37      {
38          if(Vector2.Distance(currentTarget.transform.position, transform.position) <
             0.1f)
39          {
40              if (currentTarget == endPosition) currentTarget = startPosition;
41              else if (currentTarget == startPosition) currentTarget = endPosition;
42          }
43      }
44
45      void MovePassengers(Vector2 velocity)
46      {
47          HashSet<Transform> movedPassengers = new HashSet<Transform>();
48          float directionX = Mathf.Sign(velocity.x);
49          float directionY = Mathf.Sign(velocity.y);
50
51          if (velocity.y != 0)
52          {
53              float rayLength = Mathf.Abs(velocity.y) + rayOffset;
54
55              for (int i = 0; i < verticalRays; i++)
56              {
57                  Vector2 rayOrigin = (directionY == -1) ? rayOrigins.bl : rayOrigins.
                     tl;
58
59                  rayOrigin += Vector2.right * (verticalRayDistance * i);
60                  RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.up *
                     directionY, rayLength, passengerMask);
61
62                  if (hit)
63                  {
64                      if (!movedPassengers.Contains(hit.transform))
65                      {
```

```
                    movedPassengers.Add(hit.transform);
                    float pushX = (directionY == 1) ? velocity.x : 0;
                    float pushY = velocity.y - (hit.distance - rayOffset) *
                     directionY;

                    hit.transform.Translate(new Vector3(pushX, pushY));
                }
            }
        }
    }

    if (velocity.x != 0)
    {
        float rayLength = Mathf.Abs(velocity.x) + rayOffset;

        for (int i = 0; i < horizontalRays; i++)
        {
            Vector2 rayOrigin = (directionX == -1) ? rayOrigins.bl: rayOrigins.
             br;
            rayOrigin += Vector2.up * (horizontalRays * i);
            RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.right *
             directionX, rayLength, passengerMask);

            if (hit)
            {
                if (!movedPassengers.Contains(hit.transform))
                {
                    movedPassengers.Add(hit.transform);
                    float pushX = velocity.x - (hit.distance - rayOffset) *
                     directionX;
                    float pushY = 0;

                    hit.transform.Translate(new Vector3(pushX, pushY));
                }
            }
        }
    }

    if (directionY == -1 || velocity.y == 0 && velocity.x != 0)
    {
        float rayLength = rayOffset * 2;

        for (int i = 0; i < verticalRays; i++)
        {
            Vector2 rayOrigin = rayOrigins.tl + Vector2.right * (
             verticalRayDistance* i);
            RaycastHit2D hit = Physics2D.Raycast(rayOrigin, Vector2.up,
             rayLength, passengerMask);

            if (hit)
            {
                if (!movedPassengers.Contains(hit.transform))
                {
```

```
                            movedPassengers.Add(hit.transform);
                            float pushX = velocity.x;
                            float pushY = velocity.y;

                            hit.transform.Translate(new Vector3(pushX, pushY));
                        }
                    }
                }
            }
        }
```

The last thing added is the goal that the player needs to reach to finish the level. The goal is represented by a simple door that detects the player collision and sends the information that the level is completed to the player. The script that controls that behaviour is shown in listing 9:

Listing 9: The goal controller

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

public class LevelComplete : MonoBehaviour
{
    Player player;

    private void Start()
    {

    }

    void OnCollisionEnter2D(Collision2D collision)
    {
        if (collision.gameObject.layer == 3)
        {
            player = (Player)collision.gameObject.GetComponent("Player");
            player.levelComplete();
        }
    }
}
```

The finished level can be seen in figure 32:

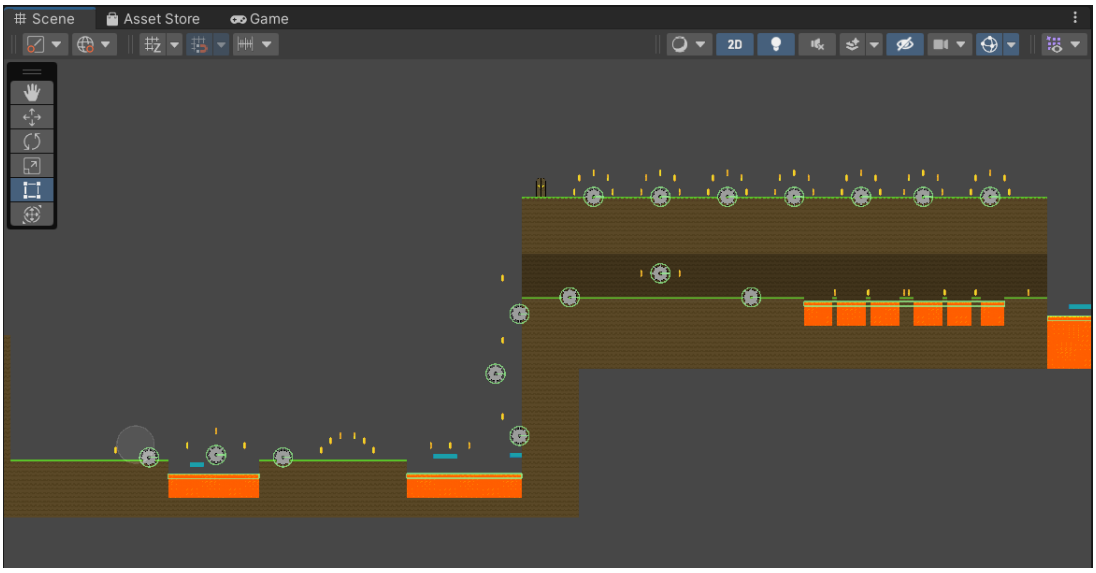


Figure 32: Finished level layout with obstacles and rewards

### 5.1.3. Background

To make the background more interesting, instead of placing a static image behind the level, a parallax background is placed instead. Since it is a lot of work to draw the components of this background and it is not the main focus of this thesis, the parallax background is imported from the Unity asset store. The asset already comes with a example code so it needs to be slightly modified. The script takes multiple images defined beforehand along with moving speed defined for each of the layers and moves them accordingly. The images that are not inside the camera focus area are moved to be in front of it to give the effect of a never ending background. The script can be seen in listing 10:

Listing 10: Parallax background controller, downloaded from [46]

```
1
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  public class ParallaxBackground_0 : MonoBehaviour
7  {
8      public float verticalOffset = 1;
9      [Header("Layer Setting")]
10     public float[] Layer_Speed = new float[7];
11     public GameObject[] Layer_Objects = new GameObject[7];
12
13     private Transform _camera;
14     private float[] startPos = new float[7];
15     private float boundSizeX;
16     private float sizeX;
17     private GameObject Layer_0;
18     void Start()
19     {
```

```
20          _camera = Camera.main.transform;
21          sizeX = Layer_Objects[0].transform.localScale.x;
22          boundSizeX = Layer_Objects[0].GetComponent<SpriteRenderer>().sprite.bounds.
             size.x;
23          for (int i=0;i<5;i++){
24              startPos[i] = _camera.position.x;
25          }
26      }
27
28      void Update(){
29          for (int i=0;i<5;i++){
30              float temp = (_camera.position.x * (1-Layer_Speed[i]) );
31              float distance = _camera.position.x  * Layer_Speed[i];
32              Layer_Objects[i].transform.position = new Vector2 (startPos[i] +
                 distance, _camera.position.y + verticalOffset);
33              if (temp > startPos[i] + boundSizeX*sizeX){
34                  startPos[i] += boundSizeX*sizeX;
35              }else if(temp < startPos[i] - boundSizeX*sizeX){
36                  startPos[i] -= boundSizeX*sizeX;
37              }
38
39          }
40      }
41  }
```

### 5.1.4. GUI

Final thing added to the game is the GUI. It consists of two parts: the visible information and hidden menus that pop up on certain events. The visible part show the info about player lives and the current score which are set through the player script shown in the section before. The same principle works for the hidden menus. The player script sends the information which menu needs to be displayed at a certain time. The script controlling this behaviour can be seen in listing 11:

Listing 11: GUI controller

```
1
2   using System;
3   using System.Collections;
4   using System.Collections.Generic;
5   using UnityEngine;
6   using UnityEngine.UI;
7   using TMPro;
8
9   public class GUIController : MonoBehaviour
10  {
11      public GameObject lives;
12      public GameObject score;
13      public GameObject finalScore;
14      public GameObject gameOver;
15      public GameObject levelComplete;
```

```
16      public GameObject menu;
17
18      public void ToggleMenu()
19      {
20          if (menu.activeInHierarchy) menu.SetActive(false);
21          else menu.SetActive(true);
22      }
23
24      public void SetLivesText(int lives)
25      {
26          this.lives.GetComponent<Text>().text = "×" + lives;
27      }
28
29      public void SetScoreText(int score)
30      {
31          string scoreText = "";
32          if (score > 1000) scoreText = "Score: " + score;
33          if (score < 1000) scoreText = "Score: 0" + score;
34          if (score < 100) scoreText = "Score: 00" + score;
35          if (score < 10)  scoreText = "Score: 000" + score;
36
37          this.score.GetComponent<Text>().text = scoreText;
38          this.finalScore.GetComponent<TextMeshProUGUI>().text = scoreText;
39
40      }
41
42      public void GameOver()
43      {
44          gameOver.SetActive(true);
45      }
46
47      internal void completeLevel()
48      {
49          levelComplete.SetActive(true);
50      }
51
52
53  }
```

An example of the menu popup can be seen in figure 33. The buttons shown in the example are controlled using a simple script that only takes the name of the scene that needs to be loaded, and loads it using the Unity *SceneManagement* library. The script can be seen in listing 12.

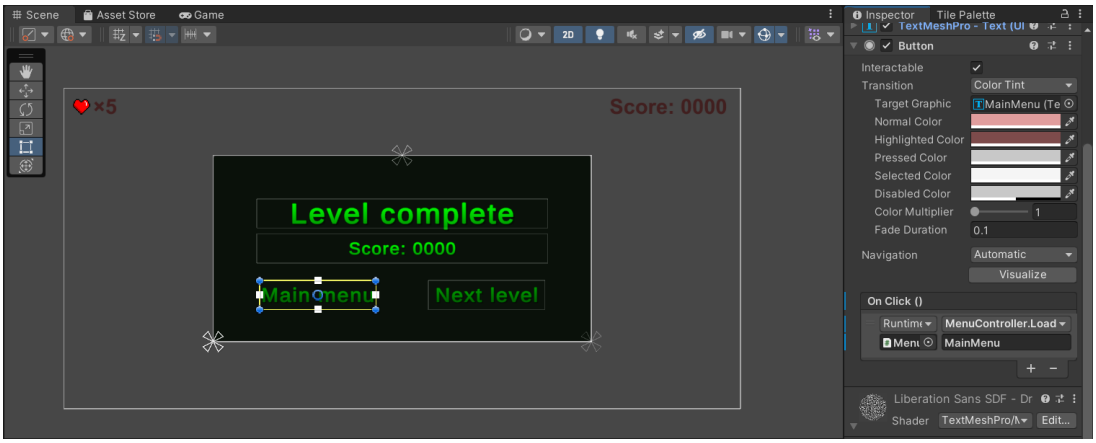


Figure 33: Example of a button that loads the main menu scene

Listing 12: Scene loader

```
1
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5  using UnityEngine.SceneManagement;
6
7  public class MenuController : MonoBehaviour
8  {
9      public void LoadScene(string sceneName)
10     {
11         SceneManager.LoadScene(sceneName);
12     }
13 }
```

## 5.2. Creating a Machine Learning Model

The deep Q learning algorithm will be used to develop the player's "brain". The TensorFlow library will be used to develop this part. TensorFlow [47][p. 265, 266] is a free open source library used in machine learning and artificial intelligence projects. It supports various applications, but mainly focuses on training and inference of deep neural networks. TensorFlow's high-level APIs are based on the Keras [48] API standard, which enables fast creation and training of neural networks. Since the ML part is developed outside the Unity project, the Unity-ML Agents Toolkit [49] package will be used to connect the two parts. It is an open source package that allows games and simulations developed in Unity to act as training environments for intelligent agents. This kit already has machine learning algorithms and models developed [50], which is not what this project needs, but it contains a Python low-level API [50] that allows direct interaction with the Unity environment using Python. This is how the network will be connected to the developed Unity environment.

The first attempt at the development was a simple network with four outputs (move left, move right, do not move and jump) which proved to be extremely slow at learning so the

approach that was finally used is two separate networks that are connected to the same input. One network controls movement (move left, move right and do not move) and the second one controls jumping (jump and do not jump). The model of the described network can be seen in the figure 34 where blue nodes represent the input layer, green nodes the output layers and gray the hidden layers.

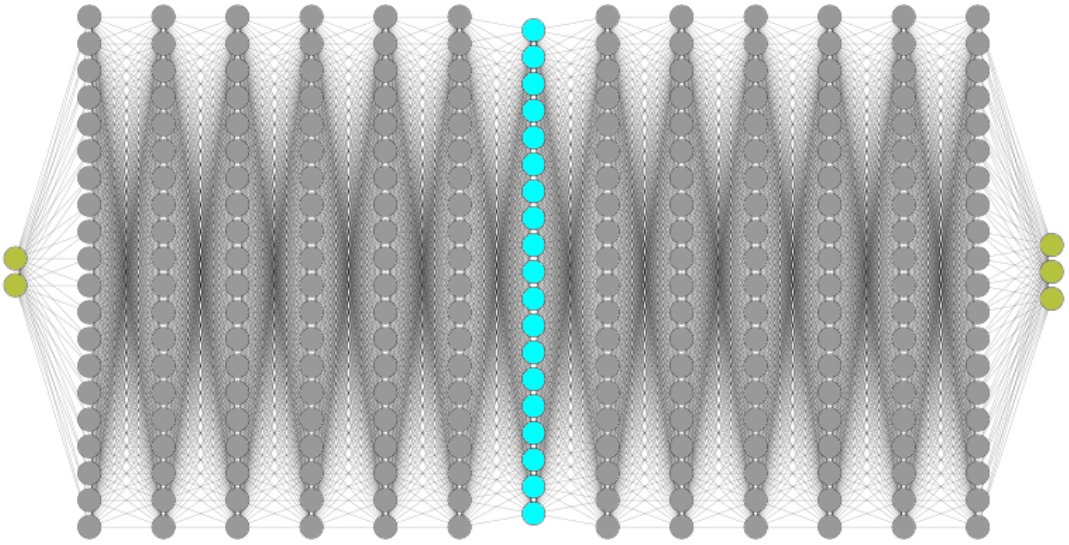

Figure 34: Visual representation of the neural network

The network has nineteen inputs, three of which are X, Y and Z coordinates of the player, and the rest are raycast distances. Eight raycast distances measure the distance from the player to the platforms in the level, and the other eight measure the distance from the player to obstacles. They are arranged in a way that each direction as one ray (up, down, left and right) as well as diagonal rays (top left, top right, bottom left and bottom right).

In order to enable the Python program to interact with the player, it is necessary to make some changes in the player script. Instead of extending the MonoBehaviour script, the Player script should extend the Agent script. Once this is done, another script called BehaviourParameters is automatically added to the player object. This script controls the necessary environment variables, such as the name of the behavior, the number of observations (inputs), actions (how many outputs it receives), and more. An example of a script object can be seen in the figure 35.

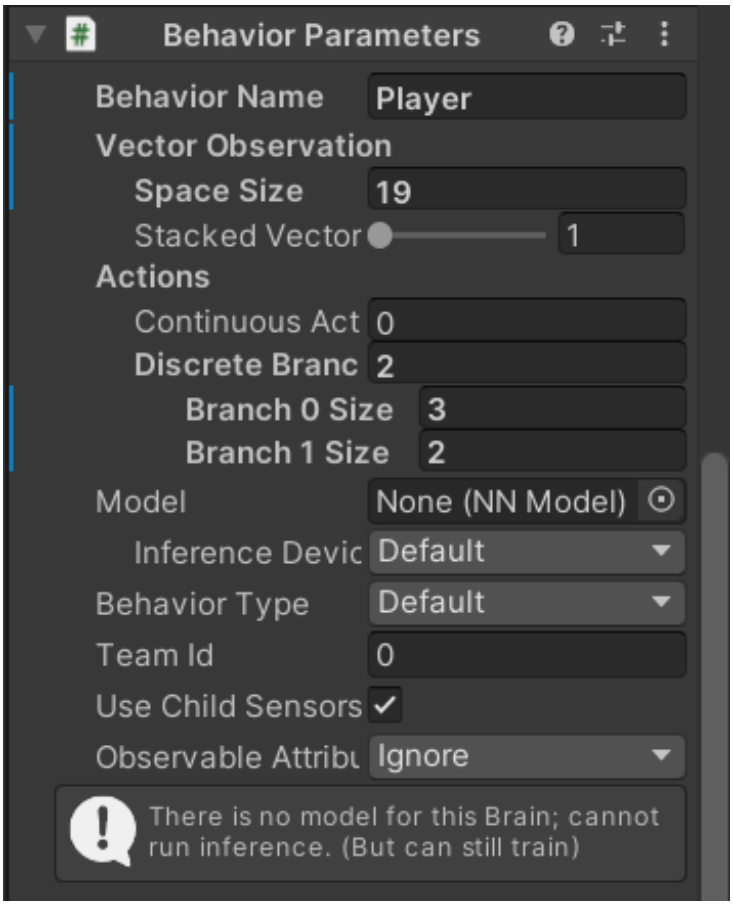


Figure 35: BehaviourParameters script example

To enable the player object to request decisions from a Python program, a DecisionMaker script should also be added to the object. Other changes that needed to be made were: move everything from the Update function to the OnActionReceived function, override the CollectObservations function and define observations (inputs), define rewards/penalties and implement rays that measure distances. After all changes, the script looks as shown in the listing 13.

Listing 13: Modified player script

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Actuators;
using Unity.MLAgents.Sensors;

//public class Player : MonoBehaviour
[RequireComponent(typeof(Controller))]
public class Player : Agent
{
    public GUIController gui;
    public GameObject checkpoint;
    public Transform target;
```

```
16      public int lives = 10;
17      public int score = 0;
18
19      public float jumpSpeed = 5;
20      public float movementSpeed = 5;
21      public float gravity = -5;
22      private bool isHeuristic = false;
23
24      float currentJumpSpeed;
25      float currentMovementSpeed;
26      float currentGravity;
27
28      bool menuOpen = false;
29
30      private collisionDistances obstacles;
31      private collisionDistances level;
32
33      private int actionCounter = 0;
34
35      Vector2 playerMovement;
36      Controller controller;
37
38      SpriteRenderer sprite;
39      Animator animator;
40      BoxCollider2D collider2d;
41
42      void Start()
43      {
44          collider2d = GetComponent<BoxCollider2D>();
45          obstacles.reset();
46          level.reset();
47          animator = GetComponent<Animator>();
48          sprite = GetComponent<SpriteRenderer>();
49          controller = GetComponent<Controller>();
50          updateGuiLives();
51          resetToCheckpoint();
52          ResetPlayerMovement();
53      }
54
55      public override void CollectObservations(VectorSensor sensor)
56      {
57          sensor.AddObservation(transform.position);
58          //sensor.AddObservation(target.position);
59
60          sensor.AddObservation(obstacles.top);
61          sensor.AddObservation(obstacles.bottom);
62          sensor.AddObservation(obstacles.left);
63          sensor.AddObservation(obstacles.right);
64          sensor.AddObservation(obstacles.tRight);
65          sensor.AddObservation(obstacles.tLeft);
66          sensor.AddObservation(obstacles.bRight);
67          sensor.AddObservation(obstacles.bLeft);
68
```

```
69          sensor.AddObservation(level.top);
70          sensor.AddObservation(level.bottom);
71          sensor.AddObservation(level.left);
72          sensor.AddObservation(level.right);
73          sensor.AddObservation(level.tRight);
74          sensor.AddObservation(level.tLeft);
75          sensor.AddObservation(level.bRight);
76          sensor.AddObservation(level.bLeft);
77      }
78
79      private void Update()
80      {
81          obstacles = calculateCollisionDistances(6);
82          level = calculateCollisionDistances(7);
83      }
84
85      public override void OnActionReceived(ActionBuffers actions)
86      {
87          AddReward(-0.3f);
88          //---------------- Movement part ----------------
89          // 0-0: Don't move
90          // 0-1: Right
91          // 0-2: Left
92          // 1-0: Don't jump
93          // 1-1: Jump
94          int movementCode = actions.DiscreteActions[0];
95          int jumpCode = actions.DiscreteActions[1];
96
97          if(actionCounter >= 1000)
98          {
99              AddReward(-20f);
100             restartScene();
101         }
102         actionCounter++;
103
104         if (controller.collisions.top || controller.collisions.bottom)
105         {
106             playerMovement.y = 0;
107         }
108
109         if (jumpCode == 1 && controller.collisions.bottom)
110         {
111             playerMovement.y = currentJumpSpeed;
112         }
113
114         Vector2 movementDirection = new Vector2(0, 0);
115
116         if (movementCode == 1)
117         {
118             movementDirection.x = 1;
119             sprite.flipX = false;
120         }
121         if (movementCode == 2)
```

```
122         {
123             movementDirection.x = -1;
124             sprite.flipX = true;
125         }
126 
127 
128         //---------------- Animation part ----------------
129         if (movementDirection.x != 0) animator.SetBool("PlayerRun", true);
130         else animator.SetBool("PlayerRun", false);
131 
132         if (!controller.collisions.bottom)
133         {
134             animator.SetBool("PlayerIdle", false);
135 
136             if (playerMovement.y > 0) animator.SetBool("PlayerJump", true);
137             else animator.SetBool("PlayerJump", false);
138 
139             if (playerMovement.y < 0) animator.SetBool("PlayerFall", true);
140             else animator.SetBool("PlayerFall", false);
141         }
142         else
143         {
144             animator.SetBool("PlayerIdle", true);
145             animator.SetBool("PlayerFall", false);
146             animator.SetBool("PlayerJump", false);
147         }
148 
149         playerMovement.x = movementDirection.x * currentMovementSpeed;
150         playerMovement.y += currentGravity * Time.deltaTime;
151         controller.Move(playerMovement * Time.deltaTime);
152     }
153 
154     public override void Heuristic(in ActionBuffers actionsOut)
155     {
156         isHeuristic = true;
157         ActionSegment<int> discreteActions = actionsOut.DiscreteActions;
158         int jump = Input.GetKey(KeyCode.Space) ? 1 : 2;
159 
160         int move;
161         if (Input.GetAxisRaw("Horizontal") == 1) move = 1;
162         else if (Input.GetAxisRaw("Horizontal") == -1) move = 2;
163         else move = 3;
164 
165         discreteActions[0] = move;
166         discreteActions[1] = jump;
167 
168         if (Input.GetKeyDown(KeyCode.Escape))
169         {
170             if (!menuOpen)
171             {
172                 gui.ToggleMenu();
173                 TogglePlayerMovement();
174             }
```

```
175         }
176     }
177 
178     void OnCollisionEnter2D(Collision2D collision)
179     {
180         //Obstacles
181         if (collision.gameObject.layer == 6)
182         {
183             AddReward(-50f);
184             if (isHeuristic)
185             {
186                 removeLife();
187             }
188             else
189             {
190                 restartScene();
191             }
192         }
193         //Target
194         if (collision.gameObject.layer == 8)
195         {
196             levelComplete();
197             AddReward(200f);
198             EndEpisode();
199             actionCounter = 0;
200         }
201         //Coins
202         if (collision.gameObject.layer == 9)
203         {
204             incrementScore(10);
205             collision.gameObject.SetActive(false);
206             AddReward(100f);
207         }
208     }
209 
210     internal void resetToCheckpoint()
211     {
212         transform.position = checkpoint.transform.position;
213     }
214 
215     internal void incrementScore(int increment)
216     {
217         score += increment;
218         gui.SetScoreText(score);
219     }
220 
221     internal void removeLife()
222     {
223         lives--;
224         resetToCheckpoint();
225         updateGuiLives();
226         if (lives == 0)
227         {
```

```
228             showGameOverDialog();
229         }
230     }
231     private void updateGuiLives()
232     {
233         gui.SetLivesText(lives);
234     }
235 
236     internal void levelComplete()
237     {
238         if (!menuOpen)
239         {
240             menuOpen = true;
241             gui.completeLevel();
242             TogglePlayerMovement();
243         }
244     }
245 
246     private void showGameOverDialog()
247     {
248         if (!menuOpen)
249         {
250             menuOpen = true;
251             gui.GameOver();
252             TogglePlayerMovement();
253         }
254     }
255 
256     private void TogglePlayerMovement()
257     {
258         if (currentGravity != 0) FrezePlayer();
259         else ResetPlayerMovement();
260     }
261 
262     private void FrezePlayer()
263     {
264         currentGravity = currentJumpSpeed = currentMovementSpeed = 0;
265         playerMovement = Vector2.zero;
266     }
267 
268     private void ResetPlayerMovement()
269     {
270         currentGravity = gravity;
271         currentJumpSpeed = jumpSpeed;
272         currentMovementSpeed = movementSpeed;
273     }
274 
275     private void restartScene()
276     {
277         score = 0;
278         gui.SetScoreText(score);
279 
280         resetToCheckpoint();
```

```
281
282         AddScore[] coins = FindObjectsOfType<AddScore>(true);
283
284         foreach (AddScore coin in coins)
285         {
286             coin.gameObject.SetActive(true);
287         }
288
289         Checkpoint[] checkpoints = FindObjectsOfType<Checkpoint>(true);
290
291         foreach (Checkpoint checkpoint in checkpoints)
292         {
293             checkpoint.gameObject.SetActive(true);
294         }
295
296         MovingPlatformsController[] platforms = FindObjectsOfType<
             MovingPlatformsController>(true);
297
298         foreach (MovingPlatformsController platform in platforms)
299         {
300             platform.Start();
301         }
302
303         EndEpisode();
304         actionCounter = 0;
305     }
306
307     private struct collisionDistances
308     {
309         public float top;
310         public float bottom;
311         public float left;
312         public float right;
313
314         public float tRight;
315         public float bRight;
316         public float tLeft;
317         public float bLeft;
318
319         public void reset()
320         {
321             top = bottom = left = right = tRight = tLeft = bRight = bLeft = 0;
322         }
323     }
324
325     private collisionDistances calculateCollisionDistances(int layer)
326     {
327         collisionDistances dst;
328         Bounds bounds = collider2d.bounds;
329
330         //top ray
331         Vector2 origin = new Vector2(bounds.min.x + (bounds.size.x / 2f), bounds.max
             .y -0.1f);
```

```
332         RaycastHit2D hit = Physics2D.Raycast(origin, Vector2.up, 1f, layer);
333         dst.top = hit ? hit.distance : 1f;
334 
335         //bottom ray
336         origin = new Vector2(bounds.min.x + (bounds.size.x / 2f), bounds.min.y + 0.1
             f);
337         hit = Physics2D.Raycast(origin, Vector2.down, 1f, layer);
338         dst.bottom = hit ? hit.distance : 1f;
339 
340         //left ray
341         origin = new Vector2(bounds.min.x + 0.1f, bounds.min.y + (bounds.size.y / 2f
             ));
342         hit = Physics2D.Raycast(origin, Vector2.left, 1f, layer);
343         dst.left = hit ? hit.distance : 1f;
344 
345         //right ray
346         origin = new Vector2(bounds.max.x - 0.1f, bounds.min.y + (bounds.size.y / 2f
             ));
347         hit = Physics2D.Raycast(origin, Vector2.right, 1f, layer);
348         dst.right = hit ? hit.distance : 1f;
349 
350         //bottom left ray
351         origin = new Vector2(bounds.min.x + 0.1f, bounds.min.y + 0.1f);
352         hit = Physics2D.Raycast(origin, Vector2.down + Vector2.left, 1f, layer);
353         dst.bLeft = hit ? hit.distance : 1f;
354 
355         //top left ray
356         origin = new Vector2(bounds.min.x + 0.1f, bounds.max.y - 0.1f);
357         hit = Physics2D.Raycast(origin, Vector2.up + Vector2.left, 1f, layer);
358         dst.tLeft = hit ? hit.distance : 1f;
359 
360         //bottom right ray
361         origin = new Vector2(bounds.max.x - 0.1f, bounds.min.y + 0.1f);
362         hit = Physics2D.Raycast(origin, Vector2.down + Vector2.right, 1f, layer);
363         dst.bRight = hit ? hit.distance : 1f;
364 
365         //top right ray
366         origin = new Vector2(bounds.max.x - 0.1f, bounds.max.y - 0.1f);
367         hit = Physics2D.Raycast(origin, Vector2.up + Vector2.right, 1f, layer);
368         dst.tRight = hit ? hit.distance : 1f;
369 
370         return dst;
371     }
372 }
```

Once everything is set in unity, the Python program can communicate with the environment. The Python program defines the already described structure of the neural network and enables saving and loading of these networks using the Keras library. Adam's optimizer is used to update the weights using the loss values of the networks. The rest of the program is simply communication with the environment and inserting inputs into the network. To adjust the learning process, the variables learningRate (what percentage of the corrected value will

be applied to the weight) and explorationRate (a number that defines a threshold that decides when to use network output and when to use random output) need to be modified in the range 0-1. The Python program can be seen in listing 14.

Listing 14: Deep Q Network program: based on [42]

```
from argparse import Action
from pickle import TRUE
import re
from select import select
from sre_constants import LITERAL_UNI_IGNORE
from wsgiref.util import application_uri
import numpy as np
import mlagents
from mlagents_envs.environment import UnityEnvironment, ActionTuple
import tensorflow as tf
import keras
from keras import layers
from keras import initializers
from keras import optimizers


def constructQNetwork(stateDim: int, moveActionDim: int, jumpActionDim: int) ->
 keras.Model:
    inputs = layers.Input(shape=(stateDim,))
    hiddenLayer1 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(inputs)
    hiddenLayer2 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(hiddenLayer1)
    hiddenLayer3 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(hiddenLayer2)
    hiddenLayer4 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(hiddenLayer3)
    hiddenLayer5 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(hiddenLayer4)
    hiddenLayer6 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(hiddenLayer5)
    qValuesMove = layers.Dense(
        moveActionDim, kernel_initializer=initializers.initializers_v2.Zeros,
         activation="linear")(hiddenLayer6)

    hiddenLayer7 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(inputs)
    hiddenLayer8 = layers.Dense(
        20, activation="relu", kernel_initializer=initializers.initializers_v1.
         HeNormal)(hiddenLayer7)
```

```
38      hiddenLayer9 = layers.Dense(
39          20, activation="relu", kernel_initializer=initializers.initializers_v1.
             HeNormal)(hiddenLayer8)
40      hiddenLayer10 = layers.Dense(
41          20, activation="relu", kernel_initializer=initializers.initializers_v1.
             HeNormal)(hiddenLayer9)
42      hiddenLayer11 = layers.Dense(
43          20, activation="relu", kernel_initializer=initializers.initializers_v1.
             HeNormal)(hiddenLayer10)
44      hiddenLayer12 = layers.Dense(
45          20, activation="relu", kernel_initializer=initializers.initializers_v1.
             HeNormal)(hiddenLayer11)
46      qValuesJump = layers.Dense(
47          jumpActionDim, kernel_initializer=initializers.initializers_v2.Zeros,
             activation="linear")(hiddenLayer12)
48
49      qNetwork = keras.Model(inputs=inputs, outputs=[qValuesMove, qValuesJump])
50      return qNetwork
51
52
53  def calculateLossValue(qValue: tf.Tensor, reward: tf.Tensor) -> tf.Tensor:
54      loss = 0.5 * (qValue - reward) ** 2
55      return loss
56
57
58  if __name__ == "__main__":
59      selection = 0
60      while(selection != "1" and selection != "2"):
61          print("---------------[Main menu]---------------")
62          print("[1] New network")
63          print("[2] Load network")
64          print("-----------------------------------------")
65          selection = input("Selection: ")
66
67      print("Start Unity environment!")
68
69      learningRate = 0.5
70      explorationRate = 0.05
71
72      movements = np.array([0.0, 0.5, 0.8])
73      jumps = np.array([0.0, 0.0])
74
75      env = UnityEnvironment(file_name=None)
76      env.reset()
77
78      behaviorNames = env.behavior_specs
79      behaviorName = list(behaviorNames)[0]
80
81      obsSpec, actSpec = behaviorNames[behaviorName]
82
83      numOfObservations = obsSpec[0].shape[0]
84      numOfActionsMove = actSpec[1][0]
85      numOfActionsJump = actSpec[1][1]
```

```
86
87      if(selection == "1"):
88          qNetwork = constructQNetwork(numOfObservations, numOfActionsMove,
             numOfActionsJump)
89      else:
90          loaded = False
91          while not loaded:
92              save = input("Save path: ")
93              try:
94                  qNetwork = keras.models.load_model(save)
95                  loaded = True
96              except:
97                  print("Path ", save, " doesn't extist or is not a save file!")
98      number = False
99      while not number:
100         try:
101             iterations = int(input("Number of iterations: "))
102             number = True
103         except:
104             print("Invalid number!")
105
106     optimizer = optimizers.Adam(learning_rate=learningRate)
107
108     for iteration in range(iterations):
109         with tf.GradientTape(persistent=True) as tape:
110             env.reset()
111             decisionSteps, terminalSteps = env.get_steps(behaviorName)
112
113             agent = -1
114             done = False
115             episodeRewards = 0
116             while not done:
117                 if len(decisionSteps) >= 1:
118                     agent = decisionSteps.agent_id[0]
119
120                 spec = behaviorNames[behaviorName][1]
121
122                 observations = decisionSteps[agent].obs[0]
123                 state = tf.constant([observations])
124
125                 qValues = qNetwork(state)
126
127                 epsilon = np.random.rand()
128                 if epsilon <= explorationRate:
129                     actMove = np.random.choice(len(movements))
130                     actJump = np.random.choice(len(jumps))
131                 else:
132                     actMove = np.argmax(qValues[0])
133                     actJump = np.argmax(qValues[1])
134
135                 action = ActionTuple()
136
137                 action.add_discrete(np.array([[actMove, actJump]]))
```

```
138
139                 env.set_actions(behaviorName, action)
140                 env.step()
141
142                 decisionSteps, terminalSteps = env.get_steps(behaviorName)
143                 if agent in decisionSteps:
144                     reward = decisionSteps[agent].reward
145                     episodeRewards += reward
146                 if agent in terminalSteps:
147                     reward = terminalSteps[agent].reward
148                     episodeRewards += reward
149                     done = True
150
151                 qValueMove = qValues[0][0, actMove]
152                 qValueJump = qValues[1][0, actJump]
153
154                 lossValueMove = calculateLossValue(qValueMove, reward)
155                 lossValueJump = calculateLossValue(qValueJump, reward)
156
157                 gradients = tape.gradient([lossValueMove, lossValueJump], qNetwork.
                     trainable_variables)
158
159                 optimizer.apply_gradients(zip(gradients, qNetwork.
                     trainable_variables))
160
161             print(f"Total rewards for episode {iteration + 1} is {episodeRewards}")
162         del tape
163
164     correct = False
165     while not correct:
166         saveModel = input("Save model? (Y/N)")
167         if(saveModel != "Y" and saveModel != "y" and saveModel != "N" and saveModel
              != "n") :
168             print("Invalid option!")
169         else:
170             correct = True
171
172     if(saveModel == "Y" or saveModel == "y"):
173         filePath = input("Save as: ")
174         qNetwork.save(filePath)
175
176     env.close()
```

To train the agent first the Python program needs to be executed. After all is set there, the Unity can start the game simulation and the agent will start to move and observe its environment, slowly learning what is the best course of action in a specific scenario. Generally, the agent learns the correct moving direction within 20 iterations (lives), depending on the learning rate and exploration rate that have been defined, and the rest learns a bit later. It is also worth noting that the moving platforms were greatly slowing down the learning process and, to speed up everything, were removed from the level and level was simplified a bit.

# 6. Conclusion

This thesis explores two different fields of work and combines them to explore how they work together, the first being game development and the second being artificial intelligence.

The Unity game engine was used for the game development part and acted as the environment for the machine learning algorithm. The scripts were written in Visual Studio while the GIMP was used for the visuals of the game. Overall, the game development process was pretty straight forward mainly due to the good documentation by Unity and the very active online community.

Python is used to build on the developed game, creating a Deep Q Network in the Tensorflow and Keras libraries and linking it together using the mlagents library. Two different approaches were tried: one neural network that controls everything and two networks that control movement and jumping. It turned out that the two networks were much more efficient at doing the job, so this approach was chosen for this experiment. To simplify everything, both networks shared the same input layer since they required the same observations.

After everything is set up, simulations are started. Initially, the expectation was that the network would learn fast since the game is not that difficult to complete. This expectation turns out to be false, as the computer doesn't perceive the game the same way we humans do and it takes a lot of trial and error to actually figure out what it's supposed to do. After hours of training, it was concluded that the level was too complicated for a single agent to learn in a reasonable amount of time (because the ML algorithm was developed to support only one agent), and the moving platforms were found to be the culprit. Because they had a changing position, the AI was getting confused and the whole process was taking too long. In order to speed everything up, they were removed from the level, and the level itself was slightly simplified, which greatly improved the agent's performance.

Overall, what proved to be the most difficult was finding a way to connect the Python program to the Unity environment since the Unity MLAgents toolkit already has its own built-in AI algorithms and was not really meant to be an intermediary for custom Python programs. Fortunately, there was a low-level Python API in the package that did just that, although it had a bit of sparse documentation so it took a bit of trial and error to get everything working.

Finally, the whole project is available publicly on the following link: `https://drive.google.com/drive/folders/1nRKQl2ZqAt_jU8xP2BZzPKs1qi-VdOZq?usp=sharing`

# Bibliography


[1] “Video game development - Wikipedia.” (), [Online]. Available: https://en.wikipedia.org/wiki/Video_game_development (visited on 06/07/2022).

[2] “What Is Game Development?” (), [Online]. Available: https://www.freecodecamp.org/news/what-is-game-development/ (visited on 06/07/2022).

[3] “Programming and Scripting with Unity | Unity.” (), [Online]. Available: https://unity.com/solutions/multiplatform (visited on 06/07/2022).

[4] “How Unity3D Became a Game-Development Beast.” (), [Online]. Available: https://insights.dice.com/2013/06/03/how-unity3d-become-a-game-development-beast/ (visited on 06/07/2022).

[5] “Unity - Manual: 2D.” (), [Online]. Available: https://docs.unity3d.com/Manual/Unity2D.html (visited on 06/19/2022).

[6] “Unity - Manual: The Project window.” (), [Online]. Available: https://docs.unity3d.com/Manual/ProjectView.html (visited on 06/19/2022).

[7] “Unity - Manual: The Scene view.” (), [Online]. Available: https://docs.unity3d.com/Manual/UsingTheSceneView.html (visited on 06/19/2022).

[8] “Unity - Manual: Scene view navigation.” (), [Online]. Available: https://docs.unity3d.com/Manual/SceneViewNavigation.html (visited on 06/19/2022).

[9] “Unity - Manual: Pick and select GameObjects.” (), [Online]. Available: https://docs.unity3d.com/Manual/ScenePicking.html (visited on 06/19/2022).

[10] “Unity - Manual: Position GameObjects.” (), [Online]. Available: https://docs.unity3d.com/Manual/PositioningGameObjects.html (visited on 06/19/2022).

[11] “Unity - Manual: The Game view.” (), [Online]. Available: https://docs.unity3d.com/Manual/GameView.html (visited on 06/19/2022).

[12] “Unity - Manual: The Hierarchy window.” (), [Online]. Available: https://docs.unity3d.com/Manual/Hierarchy.html (visited on 06/21/2022).

[13] “Unity - Manual: The Inspector window.” (), [Online]. Available: https://docs.unity3d.com/Manual/UsingTheInspector.html (visited on 06/21/2022).

[14] “Unity - Manual: Console Window.” (), [Online]. Available: https://docs.unity3d.com/Manual/Console.html (visited on 06/21/2022).

[15] “Unity - Manual: Rigidbody 2D.” (), [Online]. Available: https://docs.unity3d.com/Manual/class-Rigidbody2D.html (visited on 06/23/2022).

[16] “Unity - Manual: Collider 2D.” (), [Online]. Available: https://docs.unity3d.com/Manual/Collider2D.html (visited on 06/23/2022).

[17] “Unity - Manual: Physics Material 2D.” (), [Online]. Available: https://docs.unity3d.com/Manual/class-PhysicsMaterial2D.html (visited on 06/23/2022).

[18] “Unity - Manual: 2D Joints.” (), [Online]. Available: https://docs.unity3d.com/Manual/Joints2D.html (visited on 06/23/2022).

[19] “Unity - Manual: Constant Force 2D.” (), [Online]. Available: https://docs.unity3d.com/Manual/class-ConstantForce2D.html (visited on 06/23/2022).

[20] “Unity - Manual: Effectors 2D.” (), [Online]. Available: https://docs.unity3d.com/Manual/Effectors2D.html (visited on 06/23/2022).

[21] “Unity - Manual: Tilemap.” (), [Online]. Available: https://docs.unity3d.com/Manual/class-Tilemap.html (visited on 08/13/2022).

[22] “Unity - Manual: Creating a Tile Palette.” (), [Online]. Available: https://docs.unity3d.com/Manual/Tilemap-Palette.html (visited on 08/13/2022).

[23] “Introduction to Sprite Animations - Unity Learn.” (), [Online]. Available: https://learn.unity.com/tutorial/introduction-to-sprite-animations# (visited on 08/15/2022).

[24] “Export animations for mobile apps and game engines.” (), [Online]. Available: https://helpx.adobe.com/animate/using/create-sprite-sheet.html (visited on 08/15/2022).

[25] “Unity - Manual: Animator Controller.” (), [Online]. Available: https://docs.unity3d.com/Manual/class-AnimatorController.html (visited on 08/15/2022).

[26] “Unity - Manual: Animation Parameters.” (), [Online]. Available: https://docs.unity3d.com/Manual/AnimationParameters.html (visited on 08/15/2022).

[27] “Unity - Manual: Scripting.” (), [Online]. Available: https://docs.unity3d.com/Manual/ScriptingSection.html (visited on 08/17/2022).

[28] “Unity - Manual: Creating and Using Scripts.” (), [Online]. Available: https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html (visited on 08/17/2022).

[29] “Unity - Manual: Variables and the Inspector.” (), [Online]. Available: https://docs.unity3d.com/Manual/VariablesAndTheInspector.html (visited on 08/17/2022).

[30] “Unity - Manual: Event Functions.” (), [Online]. Available: https://docs.unity3d.com/Manual/EventFunctions.html (visited on 08/17/2022).

[31] “Unity - Manual: Important Classes.” (), [Online]. Available: https://docs.unity3d.com/Manual/ScriptingImportantClasses.html (visited on 08/17/2022).

[32] T. Minkkinen, “Basics of Platform Games,” 2016.

[33] C. Ruhl. “Intelligence: Definition, Theories & Testing.” (2020), [Online]. Available: https://www.simplypsychology.org/intelligence.html (visited on 06/28/2022).

[34] S. Russell and R. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Harlow, UK: Pearson Education Limited, 2020, p. 2145.

[35] "The Turing Test (Stanford Encyclopedia of Philosophy/Spring 2022 Edition)." (), [Online]. Available: https://stanford.library.sydney.edu.au/archives/spr2022/entries/turing-test/ (visited on 07/22/2022).

[36] "Logic and natural language." (), [Online]. Available: https://web.stanford.edu/$%5Csim$bobonich/glances%20ahead/III.logic.language.html (visited on 07/22/2022).

[37] "What is Learning?" (), [Online]. Available: https://www.queensu.ca/teachingandlearning/modules/students/04_what_is_learning.html (visited on 07/22/2022).

[38] H. Kinsley and D. Kukieła, *Neural Networks from Scratch in Python*. Harrison Kinsley, 2020.

[39] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. 2018.

[40] J. Qualls and D. J. Russomanno, "Applications of Neural-Based Agents in Computer Game Design," *Evolutionary Computation*, 10/2009. DOI: 10.5772/9601.

[41] "GitHub - SebLague/2DPlatformer-Tutorial: A 2D Platform Controller in Unity." (), [Online]. Available: https://github.com/SebLague/2DPlatformer-Tutorial (visited on 09/10/2022).

[42] "A Minimal Working Example for Deep Q-Learning in TensorFlow 2.0 | by Wouter van Heeswijk, PhD | Towards Data Science." (), [Online]. Available: https://towardsdatascience.com/a-minimal-working-example-for-deep-q-learning-in-tensorflow-2-0-e0ca8a944d5e (visited on 09/09/2022).

[43] I. Keren. "Scroll Back: The Theory and Practice of Cameras in Side-Scrollers." (2015), [Online]. Available: https://www.gamedeveloper.com/design/scroll-back-the-theory-and-practice-of-cameras-in-side-scrollers (visited on 08/12/2022).

[44] "Free Pixel Art Tiny Hero Sprites - CraftPix.net." (), [Online]. Available: https://craftpix.net/freebies/free-pixel-art-tiny-hero-sprites/ (visited on 08/12/2022).

[45] "2D Tilemap Extras | 2D Tilemap Extras | 1.6.0-preview.1." (), [Online]. Available: https://docs.unity3d.com/Packages/com.unity.2d.tilemap.extras@1.6/manual/index.html (visited on 08/11/2022).

[46] "Free 2D Cartoon Parallax Background | 2D Environments | Unity Asset Store." (), [Online]. Available: https://assetstore.unity.com/packages/2d/environments/free-2d-cartoon-parallax-background-205812 (visited on 08/12/2022).

[47] M. Abadi, P. Barham, J. Chen, *et al.*, "TensorFlow: A system for large-scale machine learning,"

[48] "TensorFlow Core | Machine Learning for Beginners and Experts." (), [Online]. Available: https://www.tensorflow.org/overview (visited on 09/09/2022).

[49] A. Juliani, V.-P. Berges, E. Teng, *et al.*, "Unity: A General Platform for Intelligent Agents," 09/2018. arXiv: 1809.02627.

[50] “ml-agents/Python-API.md at main · Unity-Technologies/ml-agents · GitHub.” (), [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Python-API.md (visited on 09/09/2022).

# List of Figures

# List of Tables

# List of Listings