OAMK
OULU UNIVERSITY OF
APPLIED SCIENCES

Yu Li

ARTIFICIAL INTELLIGENCE IN UNITY GAME ENGINE

ARTIFICIAL INTELLIGENCE IN UNITY GAME ENGINE

Yu Li
Bachelor Thesis
Spring 2017
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology

Author: Yu Li
Title of Bachelor´s thesis: Artificial Intelligence in Unity Game Engine
Supervisor: Veikko Tapaninen
Term and year of completion:Spring 2017 Number of pages:38

This thesis was conducted for Oulu Game Lab. The aim of this bachelor thesis was to develop in Oulu Game Lab a game called the feels good to be evil. The main purpose of the project was to develop a game and learn game development focus in the artificial intelligence area.

This thesis has explained the theory behind Artificial Intelligence. The game was developed in Unity Game Engine with C# language, and also Panda Behavior Tree was used in this project as an asset.

The result was the game has a finished demo build. Unity will be my first choice for game development in the future and Behavior Tree will be the solution for artificial intelligence.

Keywords: Unity, Game Design, Finite State Machine, Artificial Intelligence, Behavior Tree

PREFACE

During the training period in Oulu Game Lab, I developed this game using Unity Game Engine and Panda BT. After that I continued to expand this project with my own interests. And I learned a lot about how to design and implement AI in Unity.

I would like to thank my supervisor Veikko Tapaninen. He guided me in this thesis and helped me a lot. I would also like to thank Kaija Posio for the language correction.

Oulu 11.04.2017
Yu Li

CONTENTS

ABSTRACT 3
PREFACE 4
TABLE OF CONTENTS 5
VOCABULARY 7
1 INTRODUCTION 8
2 ARTIFICIAL INTELLIGENCE IN GAME INDUSTRY 9
2.1 Game Industry AI Progress 9
2.2 Introduction to Artificial Intelligence 10
2.2.1 Introduction to state machine 10
2.2.2 Introduction to behavior tree 11
2.3 Comparison Between different AI Model 14
2.4 Introduction to Agent Awareness 15
2.5 Introduction to Fuzzy Logic 16
3 AN INTRODUCTION TO UNITY GAME ENGINE 17
3.1 History of Unity Engine 17
3.2 Comparison of Unity and other Game Engine 17
3.3 Basic Feature of Unity Engine 18
3.3.1 Mecanim Animation 19
3.3.2 Navigation System 20
3.3.3 2D Physics 21
3.3.4 Scripting 22
3.3.5 UI 22
4 GAME MECHANICS INTRODUCTION 23
4.1 Introduction 23
4.2 Concept Statement 23
4.3 Combat 23
4.4 Summon Mechanic 24
4.5 Level Design 24
5 COLDEN DARK GAME AI DESIGN 26
5.1 Introduction 26

5.2 Game AI Logic Design 26
5.2.1 Introduction to PandaBT 26
5.2.2 AI design of the summoned minion 27
5.2.1 AI design of the summoned minion 29
5.2.1 AI design of the summoned minion 30
5.3 Design sensor of the game 32
5.3.1 Add colliders to each agent 32
5.3.2 Make target type based detection system 33
5.4 Choose the navigation system 34
5.4.1 Introduction to A* algorithm 34
6 GAME AI IMPLEMENTATION 36
6.1 Core task for Boss 36
6.2 Core task for minion 38
7 CONCLUSION 40
REFERENCES 41

VOCABULARY

Term	Meaning
AI	Artificial intelligence
IDE	Integrated development environment
2D	2 dimensional
3D	3 dimensional
FSM	Finite state machine
OGL	Oulu game lab
API	Application programming interface
Unity	Unity 3D game engine
NavMesh	Navigation Mesh

1 INTRODUCTION

When talking about AI in the game industry, it is a really in-depth topic. AI includes machine learning, agent behavior, and decision making. It is really important in modern game development.
Making a good AI can have a huge influence on game play (Wexler ,S, 2002, p.3). The idea for this thesis came up with the development of our new game in OGL.

The learning curve of AI is really deep, thus what is covered in this thesis will start from the beginning to intermediate, the major topic will cover the theory and some examples of State Machine and Behavior Tree,it also includes how AI can influence a game and some basic concepts of the Unity game engine.

The aim of this thesis was to discover the mysterious AI and learn how to implement a real AI in a real game with the Unity game engine.

2 ARTIFICIAL INTELLIGENCE IN GAME INDUSTRY

This chapter will explain how AI has been used in game industry. Also this chapter will explain two different ways to achieve the modern AI design, and also the long progress history of AI used in game industry. The purpose of this chapter is to give a brief explanation on the theory behind the modern AI design.

AI has been used in game industry since the 1970s (Wikipedia, Cited 01.01.2017). A good game can sometimes be decided by a good AI. Nowadays AI has more and more impact not only in video games but also on the fast developing mobile game industry.

2.1 Game Industry AI Progress

During the early state of game industry, AI has mostly consisted of some simple rules. It was not yet the core part of game development (Wikipedia, Cited 02.01.2017). The first game that introduced the public a truly advanced AI is a game called Half Life. The enemy AI of this game uses a schedule-driven state machine which makes the enemy show different behavior under some critical situation(Woodcock,S. 1998, p.1).

The state machine is widely used in game industry to achieve a common AI behavior, but there are some significant problems coming with the state machine. First of all, although the state machine is easy to design, it became really hard to maintain when adding more state to it, because every time when adding a new state, all the transition linked to this state and another state had to be considered. Nowadays developers have found a new way to implement those behaviors. It is called the behavior tree. It has a encapsulated logic in a hierarchy structure, From then a behavior tree has been a standard industry form. Compared to the state machine, it provides a scalable solution for adding logic to AI. It is definitely a better solution for a large project (Pereiro,R. 2015, p.1),

2.2 Introduction to Artificial Intelligence

AI has a relatively long history and it has still been changing in recent years. The most well known breakthrough was when the Google alpha beated the world's top Go player. It includes machine learning and tree search techniques (Reese,H. Cited 12.14.2016). AI is always associated with computer science, but the algorithms used in AI came from many other fields such as Maths, and Physics. The aim of AI is to create a more human-like agent to help people solve different tasks (Wikipedia, Cited 01.03.2017).

2.2.1 Introduction to state machine

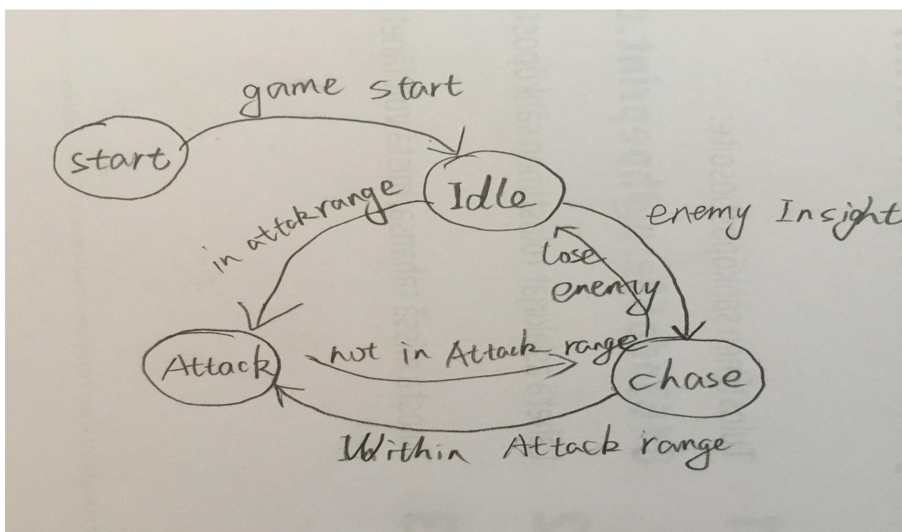

FIGURE 1. State machine example

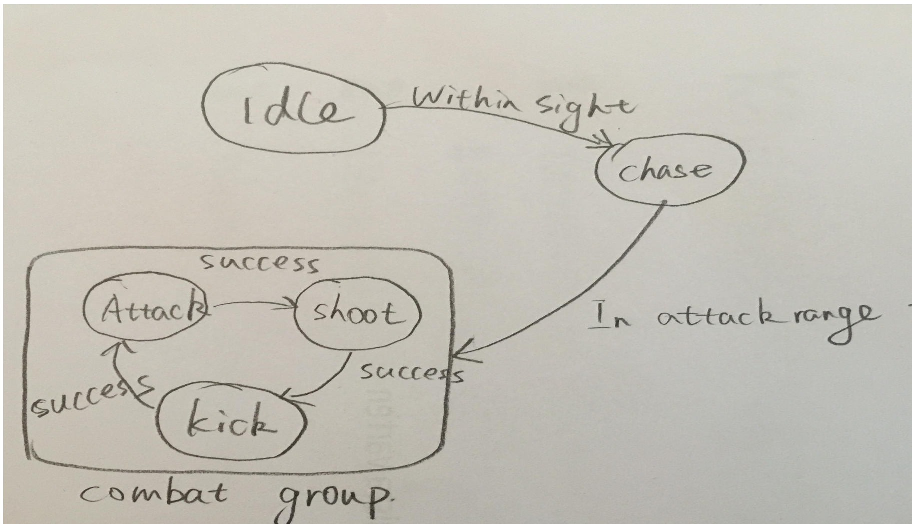

FIGURE 2. Hierarchical state machine example

The state machine is a model used to control the execution flow, the state machine only needs two simple components, a state and a transition.

The state represents the currently executed flow. The transition is used to translate into another state when it meets the condition. The state machine is like the brain of the enemy, each state is like a task for the enemy to finish. In Figure 1, the image shows a common example of the state machine. There are four states: start, idle, attack, and chase, Each of them has a transition to another state.

The state machine has an upgraded version which called hierarchical state machine(Jorge, P. 2016. p.98), it is similar to the state machine. The difference is that the developer can make a group of certain state, and all the group members can share the same transition to other groups which will provide more reusability. In Figure 2, the image shows an example of a hierarchical state machine. There are three states in a combat group, they all share the same transition from a chase state. It is like a looping system inside the combat group while the condition succeed.

2.2.2 Introduction to behavior tree

The behavior tree was developed in game industry. It is purely served as an AI model to achieve common behavior, The behavior tree is similar to the hierarchical state machine. The main difference is that the behavior tree is

made with a single block of a task rather than a state (Renato,P. Cited 29.03.2017).

The behavior tree is made with different nodes. It runs the specific node from the top to the bottom. These nodes have a different name and functionality. It can be divided into three major categories: composite, decorator and leaf. A common example of these nodes can be seen in Figure 3.

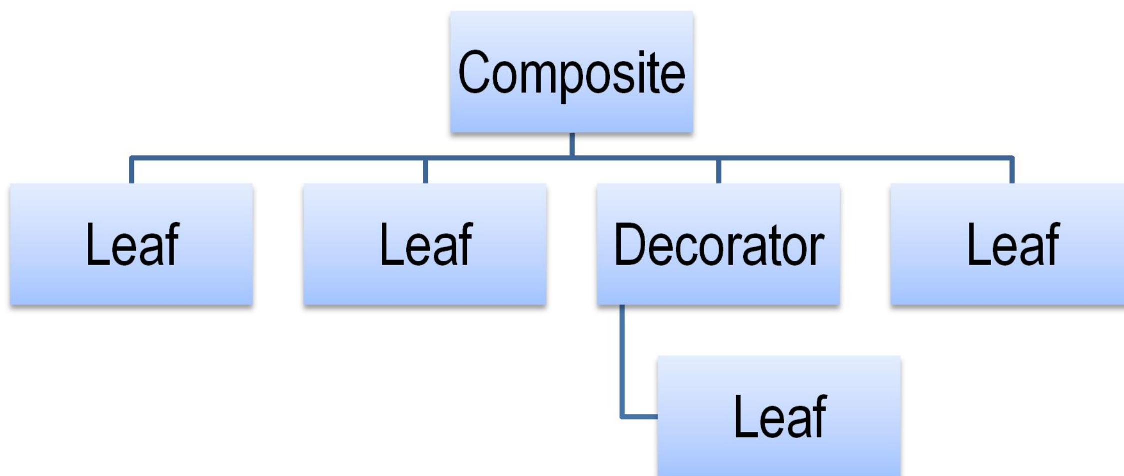

FIGURE 3. Common behavior tree example

- Composite node: This is the base structure of these hierarchical nodes. It gives the identification of the node structure. It can have several child nodes, lsuch as a sequence node, a selector node, and a parallel node. A sequence node is a self-explanatory term. It runs child tasks one by one as long as they all succeed, A selector node runs child tasks which return the first succeeded task. A parallel node will run all the child tasks at the same time. We could explain how to use a parallel node as follows. When an agent tries to attack someone while walking, we consider attacking and walking as two separate actions, the only way to perform both actions is to put them in a parallel node.

- Leaf node: A leaf node is like a leaf in the tree. It is the most low-level node yet it has the most function that actually performs a certain action.
- Decorator node: Sometimes referred to a not, mute, or repeat node, it serves as a decorator as the same in the computer language which can invert, mute, or repeat the returned status of their child. For instance, I want to make an agent shoot four times when he encounters a target, in order to do that, I can use a repeat decorator which will repeat the shooting four times.

All these nodes above are used as a decision making state for a more robust behavior purpose. These tree nodes can be really deep in a certain large project. In Table 1, the left side is a typical selector node, it is a list of question and it will evaluate each action in the order. If an attack task is succeeding, it will return to the parent node, which means that the selector task has been done. If it fails, it will go on with the second one until the last one succeeds.

TABLE 1. List of questions and actions

Player within sight?	Attack
Player died	Chase
Player reaches the attack range	Detect

In Figure 4, the image shows the use of the sequence node. It will run each sequence one by one. It runs like a test task. If the target in sight returns succeed, it will run the nested test task. If it fails, it means that the whole sequence node will fail. Thus in order to perform an attack action, these two test tasks must succeed before it.

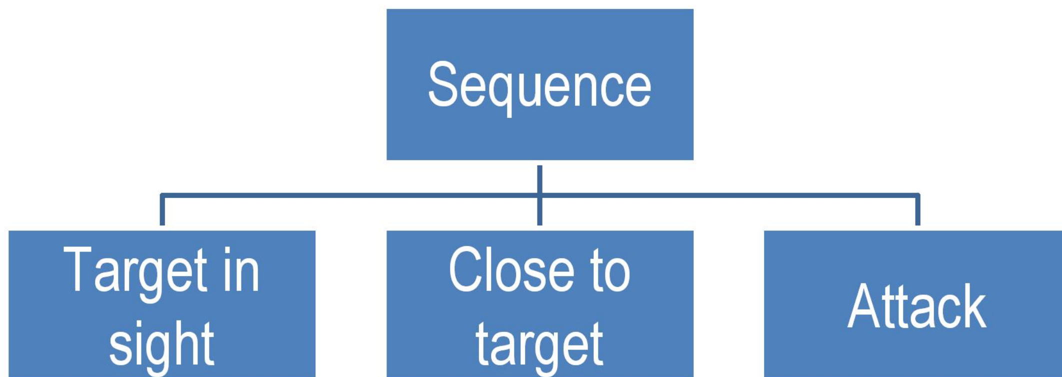

FIGURE 4. Sequence node example

2.3 Comparison Between different AI Model

In this section, I will compare some common AI design patterns. All the pros and cons will be listed below in the Table 2.

TABLE 2. Compare State Machine, Behavior Tree and Utility AI

Name	Pros	Cons
State Machine	• Easy to learn and understand • Easy to implement	• Very old style to achieve AI • Logic is limited • Can become really complex when the project scale goes up.
Behavior Tree	• Provides more flexibility • Easy to make changes when the project	• Can be hard to learn at the beginning

	goes on • Custom tasks to fit different design patterns. • Strong reusability	• Not worth for a small scale AI behavior
Utility AI	• Ultra-high performance • easy to extend	• Much more complex than the Behavior Tree • Really deep learning curve

2.4 Introduction to Agent Awareness

An agent is anything that can perceive its environment through sensors and acts. Sometimes it needs to perform some human-like behavior (TutorialPoints. Cited 28.02.2017). In order to modify the human sense, I need to create something called sensor. This sensor may include a common human sense, such as seeing, hearing or smelling. In order to simulate this kind of sense in Unity, a collider-based system, which is a component attached to an agent that has a physically shaped range, is always used to simulate the sense(Jorge, P. 2016. p.154). In Figure 5 the green box outside shows the range of sight. Anyone within the sight will be noticed by using unity’s built in physics.

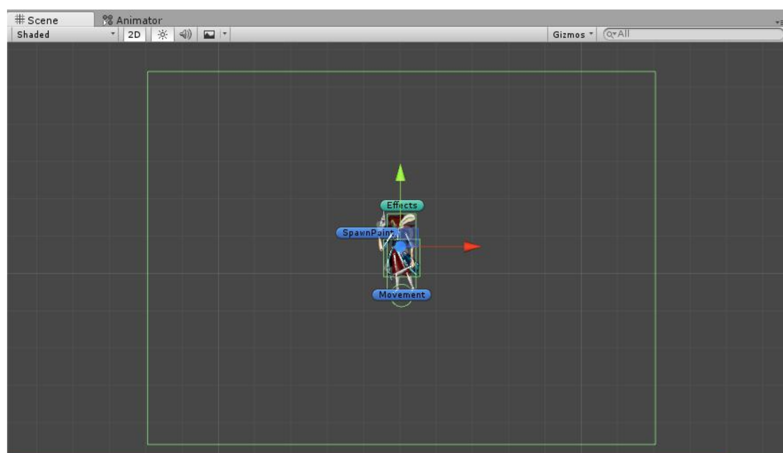

FIGURE 5. Sensor in-game use

2.5 Introduction to Fuzzy Logic

Fuzzy logic is an essential part when designing more human-like AI, unlike the true or false in the regular boolean statement, fuzzy logic uses the amount of true or false to evaluate a fact(Wikipedia. 05.02.2016.).The Figure 6 image shows how the fuzzy logic is used to evaluate temperature.When it comes to AI whether it is behavior tree or state machine, a decision always has to be made.Instead of using a boolean statement to evaluate an agent is died or alive, fuzzy logic provides a lot more fuzzy solution like healthy, almost died, almost healthy etc. It makes an AI agent more unpredictable and interesting.

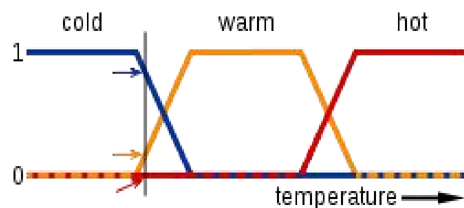

FIGURE 6. Fuzzy Logic Image(Wikipedia, Fuzzy Logic)

3 AN INTRODUCTION TO UNITY GAME ENGINE

3.1 History of Unity Engine

Unity nowadays is commonly known as Unity3D. It is a game engine with IDE for creating video games and mobile games. The first version of Unity was published on 3 October 2008. The initial idea was to create a game engine for junior developers with professional tools. It has an easy to use interface and workflow (Wikipedia 2016, Cited 02.01.2017).
With years of development, Unity has became what it is now, a great game engine for every developer.

3.2 Comparison of Unity and other Game Engine

Unity has been here for 8 years, but what makes Unity so special? What makes it different from other game engines? Unity is free to use, for the personal edition, it is totally free and a user can still use most of its functionality. Unity really opens the border between game development and common students. Table 3 contains a specific comparison between Unity and some other Game Engines.

TABLE 3. Compare Unity3d, GameMaker, Unreal Engine, Cocos2d

Name	Pros	Cons
Unity3d	• Unity personal edition is free • Has an easy Assets work flow • Uses C# which has an easier learning curve compared to C++ • Cross platform almost available for the most platform. • Supports both 2D and 3D • A lot of tutorials and a good	• Buying the assets could take a lot of money • No Source code available. • Takes a bit time to learn

	community	
GameMaker	• Simple to use • Good for small and personal project • No need to understand the deep level programming term such as multi-treading.	• Only Support 2D game • Use its own language GML • A little bit expensive
Unreal Engine	• Has a blueprint visual scripting, no writing of code is needed. • Open source • Completely free for universities • Really stunning Graphics compared to other engines	• C++ can be hard to learn • Less documented • Not enough learning material compared to Unity
Cocos2d	• Open source and free • Lots of learning resources from the community • Lots of extensions	• Does not support a console such as Xbox • No direct graphics tools to use • Only Supports 2D

3.3 Basic Feature of Unity Engine

Unity contains many features that developers can use to create a game. Also it has lots of third party integrated frameworks. In this chapter, I will cover some basic features in Unity and also demonstrate how to use them.

3.3.1 Mecanim Animation

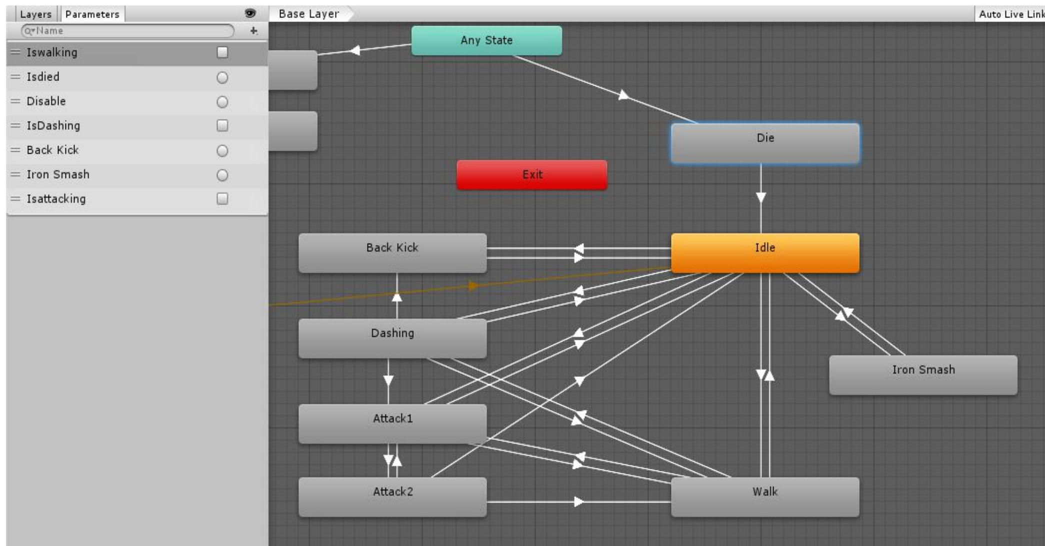

FIGURE 7. Mecanim Animation

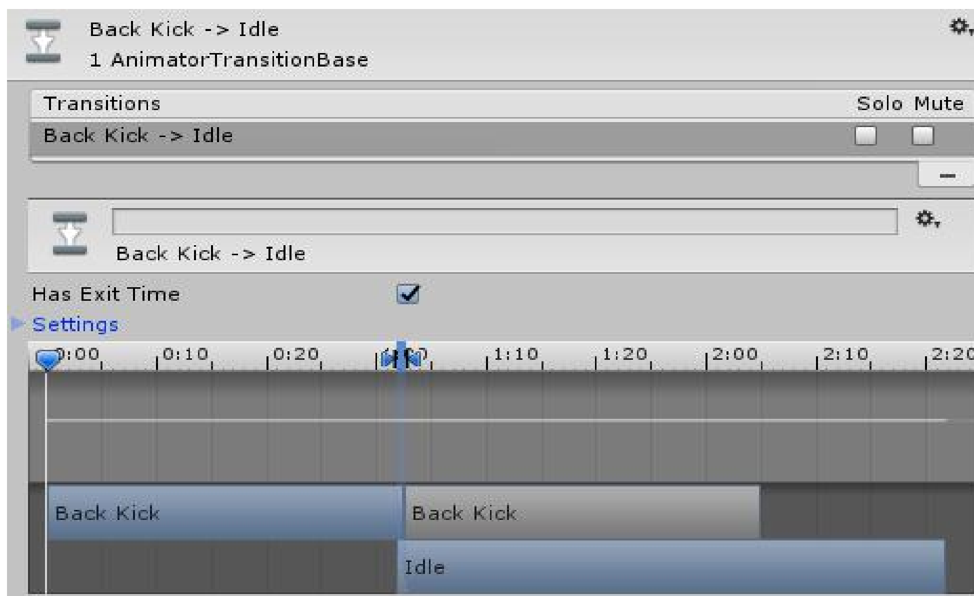

FIGURE 8. Mecanim Animation Transition

Unity uses its own mecanim animation system. This animation system uses a state machine system which has transitions between different animations. The Figure 7 shows how a basic mecanim animation system looks like. Each animation is a state in the animation system. There are lines with an arrow to link them together in order to control the transition. The blue box named as Any State is a default state when creating the mecanim system, It is used as a

default state which can transit to another state by ignoring the current state. In the left box, it contains the parameter used as a conditional checker between that transition. It includes Boolean, trigger, string and Int as a different input. The Figure 8 shows how the transition works between different animation states. By setting up the transition time between each state, the animation transition will become much smoother.

3.3.2 Navigation System

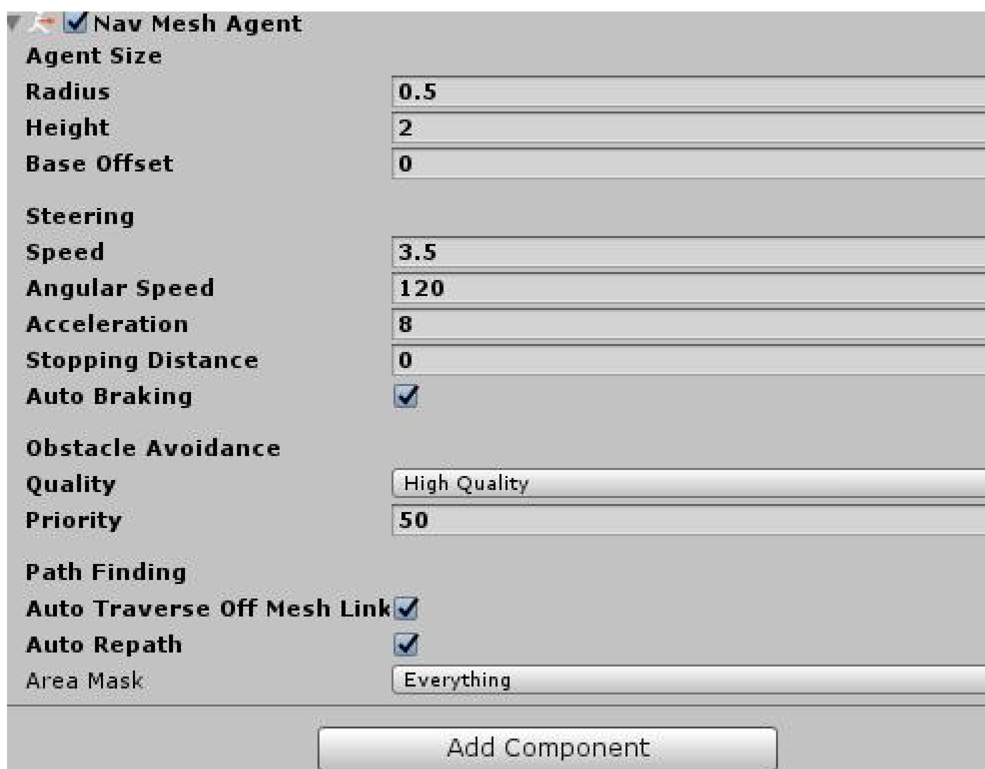

FIGURE 9. Nav Mesh Agent

Unity has its own built-in navigation system. It only supports the 3D game.
Navigation is made with two basic components: navmesh and navmesh agent. Navmesh is used for backing a walkable and non-walkable area, The navmesh agent is a component attached to a game object, which will help the object lead to the destination. Usually, a non-walkable path is some objects with colliders that are higher than the platform surface. In Figure 9 the navmesh agent is required to be added to the agent. The user can add it with the Add Component

button in the Figure 9. There are lots of parameters to adjust in the navmesh agent in order to get a better result. When it comes a real usage, the SetDestination() is always called whenever an agent needs to set its destination.

3.3.3 2D Physics

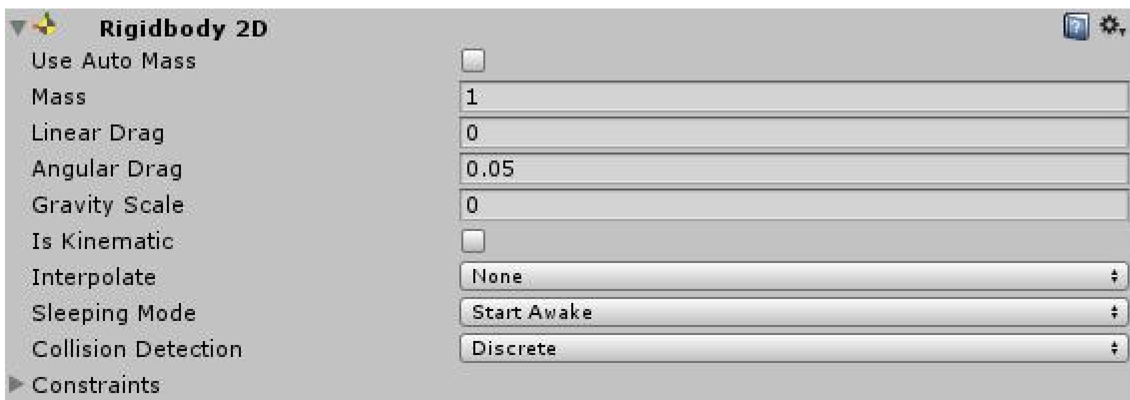

FIGURE 10. Rigidbody 2D

Unity also has a great built-in 2D physics with powerful API. In the Figure 10 a rigid body 2D is a component attached to the game object. It allows this game object to interact with other game objects by using 2D physics.

The user can set the gravity, mass and other parameters to simulate a different physical situation.

Here are some example APIs that can be used when this two game object collides with each other.

OnCollisionEnter2D: This is sent when an incoming collider collides with this game object.

OnCollisionExit2D: This is sent when the collider stop collides with this game object.

OnCollisionStay2D: Called each frame when two object collides with each other.

Those built-in functions can be used in different situations which provides a really flexible usage.

3.3.4 Scripting

Scripting is the essential part of all game engines. Unity has its own built-in class called MonoBahavior. It is attached to a certain game object to control it. Unity supports two different programming languages: C# and Unityscript. The Mono class usually contains two default functions: Start() and Update(). Start() is called when the game starts and the Update() function is called with every frame, Update() usually contains code for each frame update.
Unity has a lot of powerful APIs which makes the development of game much more easier than pure hand coding. All of those APIs can be checked online at the power Unity documentation website.

3.3.5 UI

A good game also needs a good UI system, Unity provides some really cool feature: to make the UI development process much quicker. It has a drag and drop system to simply drag anything to the canvas, By adding a different component to those UI objects, the functionality could never be easier to use.

4 GAME MECHANICS INTRODUCTION

4.1 Introduction

This chapter will be used as an introduction to the game project which I have been working on. It will include some basic concepts and gameplay features of the game.

4.2 Concept Statement

Colden Dark is a 2D hack-n-slash game situated in a medieval-themed fantasy world, which explores a different perspective on the usual portrayal of the “good guys” being the stars of every story they appear in. Colden Dark gives the player an opportunity to embrace the evil side to the fullest, burning villages, killing innocents, and spreading chaos and fear across the country.

The player finds themselfs in the shoes of an evil champion, who has been summoned by the dark prophet to protect the values of evil and oppose the egoistic and narcissistic heroes of good. His behavior in the game will directly influence the gameplay.

4.3 Combat

- Fast paced, hack-n-slash combat.
- Main means of fighting for the main character – melee weapons, including various abilities respecting the evil theme like sacrificing minions, fueling abilities by killing enemies, and so on.
- Skills have cooldown. There is no need for mana.
- Summoned minions that will fight alongside and support the main character as long as they can.
- As the player progresses in the game, they will be continuously confronted by enemy minions and minion waves, the levels themselves will end with a boss fight or an important event.

- The enemies may appear from any direction – front, behind, or the sides of the battlefield, depending on the environment.

4.4 Summon Mechanic

- Minions will be summoned next to the player with a short animation. Summoning doesn’t require currency, it relies on cooldown. Summoning a certain minion might limit using others simultaneously.
- Each minion has unique abilities that either deal damage to enemies or support the main character.

4.5 Level Design

Level design is the core part of our game, the list below contains some ideas for the level design.

- A lot of focus is being put on the environment and fluent transitions between environments. A parallax background and battlefield is essential to immerse the player in the story and its progression.
- The game should support even elevation, The levels do not have to be plainly flat.
- Various dynamic features can be supported. such as obstacles, traps, environmental destruction animation.
- A dialogue system should be seamlessly intertwined with the game. Its correct functionality and fluency is vital for the feel of the game (dialogue timing, correct event handling, perhaps choice mechanic to some extent.).
- Most of the decisions are made directly by actions in the game not by dialogue boxes (aka. Burn the village, break something, kill someone).
- Boss fights appear at the end of some levels and they are scripted to provide a challenge for the player – their design can use e.g - the hack-n-

slash combat and movement, minion mechanic, rage mechanic, abilities mechanic, or environmental obstacles.

5 COLDEN DARK GAME AI DESIGN

5.1 Introduction

This chapter will focus on the real AI usage in the Colden Dark game. The core AI concept in this game will be described in this chapter. This is a heavily code based chapter which will contain a lot of code through the entire behavior tree implementation. This chapter serves the core part of this thesis that demonstrates the game AI use in the Unity game engine.

5.2 Game AI Logic Design

In the project there are three specific individual needs for AI: the basic Enemy, the Basic summoned minion and the Boss. Each of them needs a specific design to achieve the goal.

5.2.1 Introduction to PandaBT

Building a behavior tree is a heavy task. PandaBT is a Unity free assets used to create a behavior tree. The user can create custom tasks through code and make hierarchical nodes use a pure text editor.

5.2.2 AI design of the summoned minion

```
1  tree("OrgeBehaviour")
2      fallback
3          tree("CastFirstAbility")
4          tree("Attack")
5          tree("ChaseEnemy")
6          tree("Follow")
7          tree("Idle")
8  tree("Idle")
9      while
10         sequence
11             not HasEnemy
12             InPlayerRange
13         parallel
14                         sequence
15                             StopMoving
16
17                     repeat
18                         sequence
19                             UpdateEnemyTarget
20 tree("Follow")
21     while
22          sequence
23             not HasEnemy
24             not InPlayerRange
25         parallel
26                     repeat
27                         sequence
28                             SetDestinationFollow
29                     parallel
30
31                                                 repeat
32                                                     sequence
33                                                             UpdateEnemyTarget
34                                                 repeat
35                                                     sequence
36                                                             MoveToDestination
54 tree("Attack")
55     sequence
56             HasEnemy
57             InSAttackRange
58             not CastingSpell
59             StopDashing
60             InitialTimer
61             while
62                 sequence
63                                 HasEnemy
64                                 not CanCastFirstAbility
65                                 InSAttackRange
66
67                                 not CanCastSecondAbility
68                 parallel
69                                 sequence
70                                                     AttackEnemy
71                                 sequence
72                                                     TurnOnTarget
73
74 tree("CastFirstAbility")
75     //When the enemy is visible, attack it,
76     //Otherwise forget about it.
77     sequence
78             HasEnemy
79             InSAttackRange
80             StopDashing
81             fallback
82                         sequence
83                             CanCastFirstAbility
84                             while
85                                 sequence
86                                                     InSAttackRange
87                                 sequence
88                                                     UseFirstAbility
89                                                     Wait(1.5)
90                         sequence
91                             CanCastSecondAbility
92                             while
93                                 sequence
94                                                     InSAttackRange
95                                 sequence
96                                                     UseSecondAbility
97                                                     Wait(2.0)
```

FIGURE 11. Hierarchy node

For this one I use a Behavior tree to implement AI. The PandaBT plugin is used in this individual object.

The summoned object uses 5 tree nodes to complete the task (Idle, Follow, Attack, CastFirstAbility and Chase enemy)

In the Figure 11, which is the designed tree structure for the summoned minion, at the root tree which is named the OrgeBehavior, it will run through the top to bottom as long as one of the child nodes succeeds. All of the tree nodes are placed based on the priority of execution.

- In the Idle tree, the while statement has the same functionality as a common programming language, It will run the child sequence as soon as it has met all the condition in the while statement therefore, the summoned minion will be idle if there is no enemy around and he is in the following range, below the while condition. The parallel node is the actual action node that will perform a certain action. The StopMoving is used to stop the movement whenever this node starts, and the Update EnemyTaget is used for detection so that it will be repeating every second.
- The Follow tree is a little different from the idle tree, because the player must not in the following range meet this condition, while the tree node is running, First of all, it must set the destination around the player as a following mark, and in the parallel node, the UpdateEnemyTarget node and the MoveToDestination node will run at the same time so that the object will move to the destination while continuing to update its target list.
- The Attack tree is more complex than others. At the top sequence of this tree, it will check through all the conditions one by one. In the while statement it has all the conditions that will be checked in each frame, because the object needs to make sure that he is in the attack range and also all the abilities are on cooldown. This is not like one time condition therefore it has to be in a while loop.
- The last one, the Ability tree is similar to the attack tree. The difference is that the last node is a fallback node. The fallback node is like a selector, it will run the children until one of them is succeeded. the ability tree has a separate cooldown so that we need that selector to choose which one to cast at a certain condition.

5.2.1 AI design of the summoned minion

Considering the topics of the thesis, I chose to use a variety of AI design. For the easy AI, such as the enemy without a further extension, a state machine can be a good choice because it is easy to implement and understand.

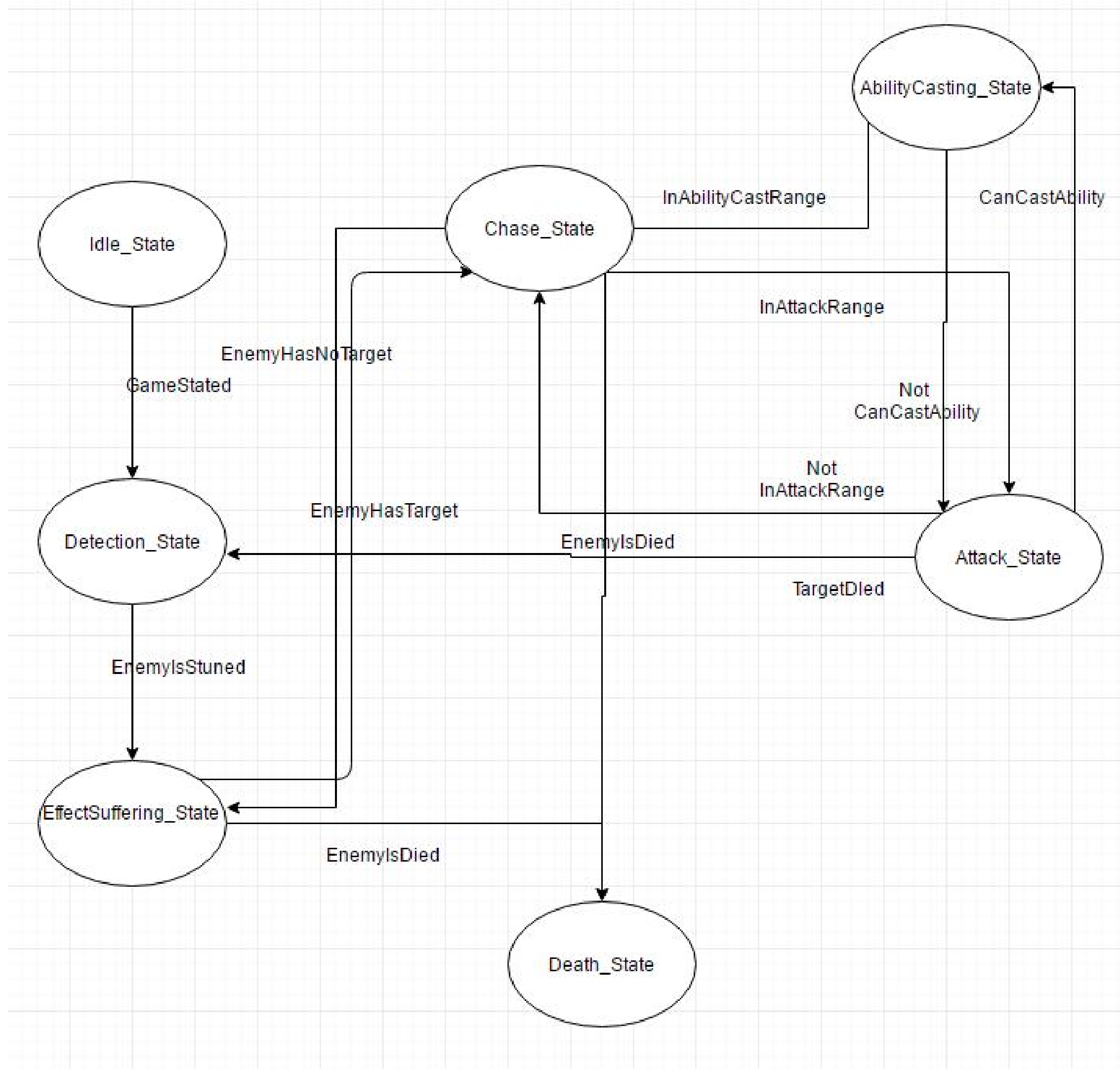

FIGURE 12. State Machine Diagram

The Figure 12 shows the State Machine diagram which is used for the enemy minion. It has totally 7 states, Each state has a transition to another state when it meets the condition. The death state is a state that every state should have a transition on it.This state machine is a basic design and it can also be translated to a behavior tree. In order to use the state machine easily, I will keep the transition as simple as possible:

- Idle State: From the idle state. It will be translated to a detection state when the game starts.
- Detection State: The detection state is used to evaluate a target. When the target is detected, it will translate to a chase state.
- Chase State: In the chase state, the minion will chase the target until it reaches the attack range.it can be translated into to two different states, depending on whether it reaches the attack range first or the ability cast range first.
- Attack State: In this state, the minions will attack the target, it will translate back to chase state when the target is out of attack range, or translate to detection state when the current target dies, or translate to ability casting state when the ability is not on cooldown.
- Ability Casting State: This state is similar to attack state, but it has a long time cool down, so it will call only a few times and translate back to previews state.
- Death State: this state is so called any state that every state has a transition to it. It has a basic condition that the minions die, and all the other state will translate to this state while muting all the other states.

5.2.1 AI design of the summoned minion

For the last one, it comes to the hardest part, the Boss AI design. A boss fight is suppose to be a challenge for the player. It should be funny and well balanced. in this particular game, our boss character is a ranger with a knife, thus the boss should have two different attack style which depends on the distance between the boss and the character. On the other side the boss will have the melee ability and the ranged ability. The Figure 13 shows a boss behavior tree where there are 7 tree nodes based on the Behavior node.
The general function of each tree:

- SummonMinions: this tree is used to handle the boss to summon minions. It is very simple. There is only one sequence in this tree therefore it will

check all the conditions first and summon the minion and then wait for an animation time to finish.

- UseBackKick: this tree is used for the boss to do a special melee attack.
- UseAbility: this tree is used for the boss to use a ranged ability. This tree is very similar to the CastAbilityTree in OrgeBehavior but it is more complex. In the selector node there is two sequences represents two abilities, but inside the sequence there is another selector which is used to check current ammo. Thus it combines the reload and the ability together to make a more complex behavior.
- FireWithBullets: this tree is used for the boss to do a basic range shot. It will check all the conditions one by one in the sequence node, And while the boss is in the shooting range, it will check the ammo first and ensures that the current ammo is more than 3, otherwise the boss will recharge the ammo and then do the basic shooting. After the basic shooting, a selector node with three children will be executed. Thus the boss will either do a special attack or just continue on the melee chase.
- Attack: this tree is used to do a basic melee attack. It is very similar to the attack tree in OrgeBehavior.
- Chase: this tree is used to Chase a character. It is very similar to the chase tree in OrgeBehavior.
- Idle: this tree is used as idle for the boss.

```
tree("BossBehaviour")
        fallback
            tree("SommonMinions")
            tree("UseBackKick")
            tree("UseAbility")
            tree("FireWithBullets")
            tree("Attack")
            tree("Chase")
            tree("Idle")
tree("Idle")
    while
            not HasTarget

          repeat
              sequence
                    UpdateTarget
tree("Wondering")
   sequence
       Wait 1.0
tree("SommonMinions")
    sequence
        HasTarget
        CanSommonMinion
        not IsLoadingAmmo
        not CastingSpell
        Stop
        SommonMinion
        Wait(1.5)
        tree("FireWithBullets")
tree("Chase")
   while
         sequence
            HasTarget
            not InShootingRange
            not IsLoadingAmmo
            Wait(1.0)

        parallel
                repeat
                        sequence
                            MoveToDestination

                repeat
                        sequence
                            SetDestinationTarget
                repeat
                        sequence
                            UpdateTarget
```

```
tree("MeleeChase")
   while
         sequence
            HasTarget
            not InAttackRange
        parallel
                repeat
                    sequence
                        MoveToDestination

                repeat
                    sequence
                        SetDestinationTarget
                repeat
                    sequence
                        UpdateTarget

tree("FireWithBullets")
    sequence
        HasTarget
        InShootingRange
        not CanUseIceBullets
        not CanUseBackStab
        not CastingSpell
        not InAttackRange
        Stop
        while InShootingRange
            sequence
                fallback
                    sequence
                        AmmoLessThan(3)
                        tree("Reload")
                    AmmoMoreThan(3)
                BasicShoot
                Wait(2.0)
                fallback
                    sequence
                        CanUseIceBullets
                        UseIceBullets
                        Wait(2.0)
                        tree("MeleeChase")
                    sequence
                        CanUseBackStab
                        UseBackStab
                        Wait(2.0)
                        tree("Chase")
                    tree("MeleeChase")
```

```
tree("UseAbility")
  sequence
        HasTarget
        InShootingRange
        not IsLoadingAmmo
        Stop
        fallback
                    sequence
                        fallback
                            sequence
                                                AmmoLessThan(2)
                                                LoadAmmo
                                                Wait(4.0)
                            AmmoMoreThan(2)
                        CanUseBackStab
                        while
                            sequence
                                                InShootingRange
                            sequence
                                                UseBackStab
                                                Wait(2.0)
                    sequence
                        fallback
                            sequence
                                                AmmoLessThan(1)
                                                LoadAmmo
                                                Wait(4.0)
                            AmmoMoreThan(1)
                        not InAttackRange
                        CanUseIceBullets
                        while
                            sequence
                                                InShootingRange
                            sequence
                                                UseIceBullets
                                                Wait(2.0)

Tree("Attack")
    sequence
        HasTarget
        InAttackRange
        not CastingSpell
        Stop
        while
            sequence
                            not CastingSpell
                            InAttackRange
                            not CanSommonMinion
            sequence
                            BasicAttack

Tree("UseBackKick")
    sequence
        HasTarget
        InAttackRange
        CanUseBackKick
        Stop
        while InAttackRange
            sequence
                UseBackKick
```

FIGURE 13. First part of behavior tree

5.3 Design sensor of the game

In this chapter, I will use the Unity engine built in physics to simulate a sense stimuli on a different agent. In this game every agent needs to have a sight to decide if he can see their opponent. I will use colliders as a sight box to simulate the length and width of the sight.

5.3.1 Add colliders to each agent

Each agent has a different sight, The melee minion should have a shorter range compared to the ranged minion, Thus I adjusted a different sight range for them to fit this purpose. By using the Unity built-in colliders we do not have to write

our own physics detection when the colliders collide with any game object. An OnTriggerEnter() function is called to ensure that this agent is detected and added to the target list.

5.3.2 Make target type based detection system

When the agent detects multiple targets, I want that this agent can choose the desired target based on the distance and the target type of the opponent. I will use a list to store all the targeted opponents. The Figure 14 shows how to fetch the best target in the list. I use an individual number in different types of minions by multiplying it with the current distance. The final result will be fetched in the list by comparing the current target with the highest priority target in order to find those targets in the same priority. And within the if function, another if function will be used to find the closest target, which shares the same type of priority. The result came out with the highest priority target with the closest distance.

```
public Transform GetHighestPriorityTarget()
{
    Vector3 position = transform.position;

    /* Currently evaluated target in the loop */
    MinionType target_type;
    float target_distance;
    float current_target_priority = 0.0f;
    float max_distance = Mathf.Infinity;
    float highest_priority_target_priority = 0.0f;

    float x_distance, y_distance;
    for (int i = 0; i < detected_enemies.Count; ++i)
    { // Check if the enemy isnt null or if he died.
        if (detected_enemies[i] != null && (detected_enemies[i].GetComponent<Collider2D>().enabled))
        {
            /* Get the enemy type */
            if (detected_enemies[i].GetComponent<Unit>())
            {
                target_type = detected_enemies[i].gameObject.GetComponent<Unit>().stats.Type;
                if (target_type == MinionType.FLY && stats.Type == MinionType.MELEE && detected_enemies[i].gameObject.GetComponent<Minion>().flying_state == FlyingState.WALK)
                {
                    target_type = MinionType.MELEE;
                }
            }
            else {
                target_type = detected_enemies[i].gameObject.GetComponent<Castle>().GetMinionType();
            }
            /* Get the distance between the targeting minion and target, emphasising the x axis as it is more important. */
            x_distance = 2 * (detected_enemies[i].transform.position.x - position.x);
            y_distance = (detected_enemies[i].transform.position.y - position.y);
            target_distance = Mathf.Pow(x_distance, 2) + Mathf.Pow(y_distance, 2);

            /* Get the target priority */
            current_target_priority = GetTargetPriority(stats.Type, target_type, target_distance);
            /* Evaluate if it is the biggest priority */
            if (current_target_priority > highest_priority_target_priority)
            {
                highest_priority_target_priority = current_target_priority;
                if (target_distance < max_distance && current_target_priority==highest_priority_target_priority) {
                    max_distance = target_distance;
                    target = detected_enemies [i].transform;
                }
            }
        }
    }
    return target;
}
```

Figure 14. Function to fetch targets

5.4 Choose the navigation system

In this chapter I will cover the navigation system used in Colden Dark, Unity has its built-in 3D navigation system, but our game mainly uses 2D physics. Thus, I needed to find a 2D navigation system which fits our game. I found that the polynav2D is a great asset in the Unity assets store and it uses the A* algorithm to find the closest path.

5.4.1 Introduction to A* algorithm

A* is a computer algorithm used in pathfinding. It is one of the most popular methods used in finding the closest path (Wikipedia. Cited 07.02.2017.)
A* has an equation to determine the closest path: F = G+ H (Wikipedia, Cited 07.02.2017.). The Figure 15 shows each value of a nearby square. The result F shows the closet to move on. The nearby square will be the original spot to evaluate the value. By simply choosing the lowest F value a path will be drawn and it is the closest path to the destination.In the Figure 16 the blue square shows the final path and all the calculated squares from the start position.

- G stands for the movement cost from the starting square to the nearby square.
- H stands for the movement cost from the nearby square to the final destination.

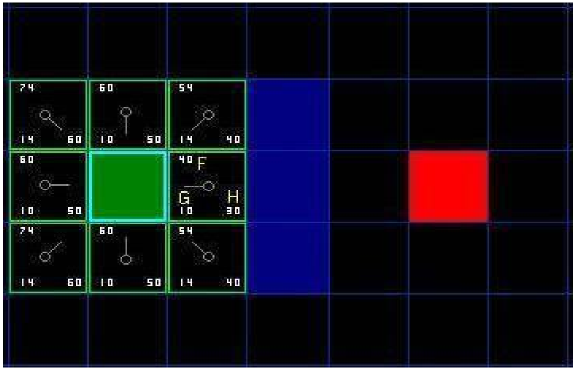

FIGURE 15. A pathfinding grid(A* Pathfinding for beginners)*

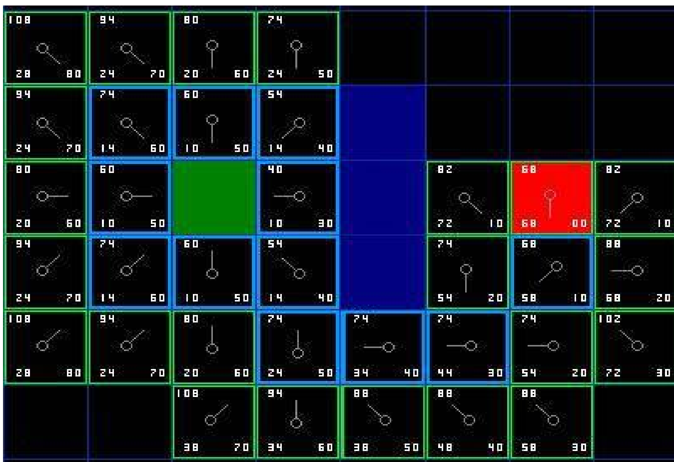

FIGURE 16. Final path to the destination(A Pathfinding for beginners)*

6 GAME AI IMPLEMENTATION

In this chapter, I will demonstrate the implementation of core task for the agent to do the actual behavior.

6.1 Core task for Boss

The Core mechanic of Boss Includes a basic attack, a special attack, a reload, a movement and summon minions. The basic attack includes shooting with a gun and slashing with a dagger. In the Figure 17 the code shows how to implement a melee attack. I use a swing_progress_timer as an interval in each attack. It will reset to 0 every time the boss attacks. In the Figure 18 the shooting function will be called every time when the shooting animation is played. It will create a projectile on the correct position and the projectile will move to a certain direction where the boss is facing.

```
[Task]
public void BasicAttack()
{
    if (swing_progress_timer > 0) {
        swing_progress_timer = swing_progress_timer - Time.deltaTime;
        isAttacking = false;
        anim.SetBool ("Isattacking", isAttacking);
    }
    if (swing_progress_timer <= 0) {
        swing_progress_timer = 0;
    }
    if (swing_progress_timer == 0) {
        isAttacking = true;
        swing_progress_timer = stats.SwingTimer;
        anim.SetBool ("Isattacking", isAttacking);
    }
}
```

FIGURE 17. Code to implement a melee attack

```
[Task]
public void Shoot()
{
    clone = PoolManager.Spawn(bullet_prefab.gameObject, firepoint.position, Quaternion.identity);
    clone.GetComponent<Projectile>().Init(this);
    ammo--;
    //clone.GetComponent<Projectile> ().unit ddd= GetComponent<Unit> ();
            if (!is_facing_right) {
                if (clone.transform.localScale.y < 0) {
                    clone.transform.localScale = new Vector3 (clone.transform.localScale.x, clone.transform.localScale.y * -1);
                }
                clone.GetComponent<Rigidbody2D> ().AddForce (Vector2.left * 1000);
            } else {
                if (clone.transform.localScale.y > 0) {

                    clone.transform.localScale = new Vector3 (clone.transform.localScale.x, clone.transform.localScale.y * -1);
                }
                clone.GetComponent<Rigidbody2D> ().AddForce (Vector2.right * 1000);
            }

}
```

FIGURE 18. Code to implement gunshot

In order to perform the special attack, I made a countdown for each special attack. The boss will perform certain special attack when the countdown reaches 0. In the Figure 19 the AnimatorControl is used to call the exact animation linked to the spell. The castingSpell is a Boolean type which is used to check if the boss is casting a spell so that it will disable another movement at the same time. All the commands are running in a coroutine, it will be paused at the time yield a statement is called so that the function will wait the exact time for the animation to finish.

```
public void Cast(Spell spell)
{
    StartCoroutine (MinionSpellCast (spell));
}
public IEnumerator MinionSpellCast(Spell spell)
{

        //Play the spell cast animation
        AnimatorControll(spell);
        castingSpell = true;
        yield return new WaitForSeconds(spell.spellCastTime);
        //Set up a spell and cast it.
        //character.invinsible = false;
    castingSpell = false;
    yield break;
    //end.
}
```

FIGURE 19. Code to implement a special attack

The movement function has three steps. It will follow three functions: SetDestinationTarget, MoveToPosition, and WaitForArrival one by one. The

Figure 20 shows the SetDestinationTager function, where the destination is set to the current target position while in the MoveToDestination function, it uses a Boolean to control the walking animation and the Flipturn function is used to ensure that the boss will always face the target position.

```
[Task]
public bool SetDestinationTarget()
{
    bool seted=false;
    if (target != null) {
        destination = target.transform.position;
        seted = true;
    }
    return seted;
}
[Task]
public void MoveToDestination()
{
    if (is_walking == false) {
        is_walking = true;
        anim.SetBool ("Iswalking", is_walking);
    }
    MoveTo(destination);
    FlipTurn ();
    WaitArrival();
}
```

FIGURE 20. Function for movement

The reload and summon function is similar to the special attack, they both have a cooldown timer, which will trigger when the boss has no ammo or the summon cool down is 0 separately.

6.2 Core task for minion

The core task for minions shares some basic function with the boss. Some are introduced below.

Each minion has its own attack range. In the Figure 21 I use a Boolean function to check whether the target is in the attack range, The parameter x_distance represents the distance between the target and the minion on the x axis. The y_distance is calculated depending on a different type of the minion. In the end, the if statement is used to check if the target is in the attack range, it will return true if the target is within the range.

```
public virtual bool InAttackRange()
{
    if (target == null) {
        return false;
    }
    float x_distance = Mathf.Abs(target.position.x - transform.position.x);
    float y_distance = 0;
    if (stats.Type == MinionType.MELEE) {
        y_distance = Mathf.Abs(target.position.y - transform.position.y);
    } else if (stats.Type == MinionType.RANGED) {
        y_distance = Mathf.Abs(target.position.y - (transform.position.y + 1.1f * transform.localScale.y));
    }


        if (x_distance < stats.SwingLength && y_distance < stats.SwingLength)
        {
            return true;
        } else {
            return false;
        }
}
```

FIGURE 21. Function to check attack range

The next function is about flipping the object. In the Figure 22, this function is used to keep flipping the object at run-time. In the beginning the if function is used to check a valid target. If the target is not valid, it will return back, and then I will calculate the distance between the target object and this object to get the relative distance. By checking the relative distance with the current facing state I can easily set this object always facing the right direction.

```
[Task]
public virtual void TurnOnTarget()
{
    if (target != null) {
        if (!target.gameObject.activeSelf)
        {
            target = null;
            return;
        }

        float relativePosition;
        if (target.tag == "Castle")
        {
            if (target.GetComponent <Castle>().GetAllegiance() == AllegianceType.EVIL)
                relativePosition = (target.position.x - target.GetComponent<SpriteRenderer>().bounds.size.x) - transform.position.x;
            else
                relativePosition = (target.position.x + target.GetComponent<SpriteRenderer>().bounds.size.x) - transform.position.x;
        }
        else
            relativePosition = target.position.x - transform.position.x;

        if (stats.Allegiance == AllegianceType.EVIL) {
            if (relativePosition > 0 && !is_facing_right) {
                Flip ();
            }
            if (relativePosition < 0 && is_facing_right) {
                Flip ();
            }
        }
        if (stats.Allegiance == AllegianceType.GOOD) {
            if (relativePosition > 0 && !is_facing_right) {
                Flip ();
            }
            if (relativePosition < 0 && is_facing_right) {
                Flip ();
            }
        }
    }
```

FIGURE 22. Function to turn on target at run-time

7 CONCLUSION

The main advantage of a behavior tree is the flexibility and expandability, even the learning curve is longer than the state machine. But after I started it, it became really easy to implement. Unity has some free behavior tree assets thus I did not have to write my own behavior tree engine which made the development process much quicker.

The task for the development included AI design, game design, and programming. The whole development of the game was also a learning process for me, I was not very familiar with the AI concept at the beginning, therefore this thesis process gave me more understanding of the AI and Unity engine. My overall skill with Unity has increased a significant amount.

There were some problems during the development of the game, I had to spend a lot of time designing the behavior tree, and the logic became really complex as the process went on.

If I want to make a game a with cross-platform in the future, Unity will clearly be my first choice and behavior tree would be the solution for AI.

REFERENCES

A* Pathfinding for Begininers. 2005. Partrick Lester. Cited 31.03.2017 http://www.policyalmanac.org/games/aStarTutorial.htm

Artificial intelligence tutorial. TutorialsPoint. Cited 28.02.2017 https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_agents_and_environments.htm

Chris, S. 2014 Behavior tree for AI: How they work. Cited 07.11.2016. http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

Cocos2d. Chukong Technology. Cited 12.02.2017 http://www.cocos2d-x.org/

Game Maker. Yo Yo Games. Cited 30.03.2017 http://www.yoyogames.com/gamemaker

Hope, R. 2016. The 6 most exciting AI advances of 2016. Cited 12.10.2016. http://www.techrepublic.com/article/the-6-most-exciting-ai-advances-of-2016/

James, W. 2002. Artificial Intelligence in Games. Cited 06.02.2017. https://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf

Jorge, P. 2016. Unity 5.x Game AI programming Cookbook. Mumbai: Packt Publishing.

Panda BT. 2016. Eric Begue. Cited 01.02.2017 http://www.pandabehaviour.com/

Renato, P. 2014. An Introduction to Behavior tree. Cited 25.01.2016.

http://guineashots.com/2014/07/25/an-introduction-to-behavior-trees-part-1/

Steven, W.1998. Game AI the state of the industry. Cited 11.10.2016.
http://www.gamasutra.com/view/feature/131705/game_ai_the_state_of_the_industry.php

Unity technology. Unity Documentation. Cited 02.02.2017
https://docs.unity3d.com/Manual/

Unreal Engine. Epic Games. Cited 31.03.2017
https://www.unrealengine.com/zh-CN

Wikipedia. 2017. Artificial intelligence. Cited 02.01.2017.
https://en.wikipedia.org/wiki/Artificial_intelligence_(video_games)

Wikipedia. 2017. Unity. Cited 02.01.2017.
https://en.wikipedia.org/wiki/Unity_(game_engine)

Wikipedia. 2017. A* search algorithm. Cited 25.01.2017.
https://en.wikipedia.org/wiki/A*_search_algorithm

Wikipedia. 2017. Fuzzy Logic. Cited 05.02.2017
https://en.wikipedia.org/wiki/Fuzzy_logic