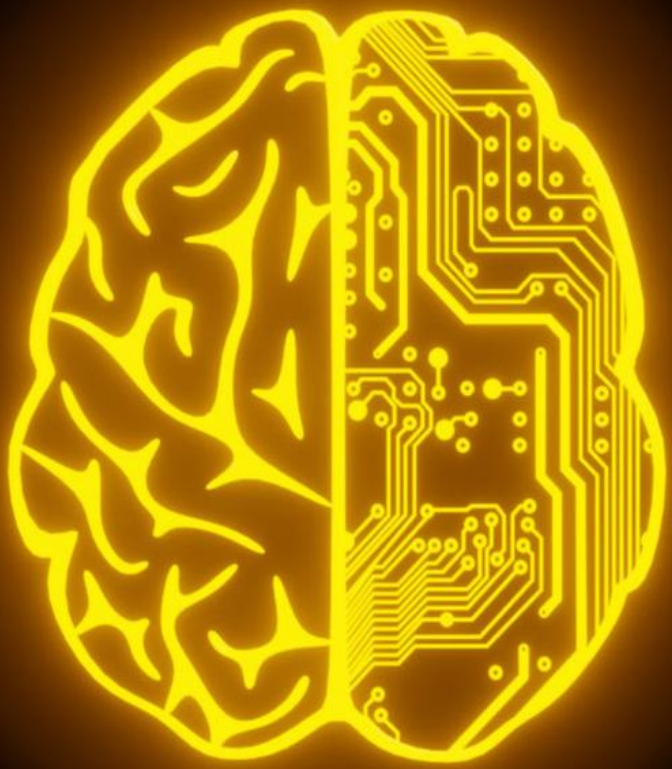
Boost your Unity/C# AI Programming

Mina Pêcheux

2023



GIVE YOUR MINIONS SOME BRAINS!

“Boost your Unity/C#: AI Programming”

Copyright April 2023 © by Mina Pêcheux All rights reserved.

While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

AI PROGRAMMING

First edition. April 2, 2023.

Copyright © 2023 Mina Pêcheux.

Written by Mina Pêcheux.

0 - Preface

This book aims at giving you an all-around knowledge of the most common AI techniques, from finite state machines to behaviour trees, planners and utility-based AI. As we progress in our journey, we'll go through each tool one by one, and we'll talk both about the *why* and the *why* use that technique instead of this one? did game developers come up with this architecture? *how* do you implement such a model in a Unity/C# project?

By the end of your read, you should have a good grasp of where those famous AI tools come from, what well-known games leverage them today, and how you can use them in your future game projects. And you'll also have several complete demo scenes to tweak and expand to your liking, so as to deepen your understanding of the topic.

So, ready? Then let's get started :)

Who is this book for?

Boost your Unity/C#: AI Programming is written for game developers who want to learn more about artificial intelligence in the context of video games, and who are looking for a hands-on approach to AI programming.

Readers should be familiar with Unity – in particular, the book will not detail all the steps to setting up game objects in a scene, or adding and editing components. Also, as all code samples use C#, a good knowledge of C# is a definite plus (although some core C# principles will be recalled when needed).

What is this book about?

Chapter 1: AI in games gives an overview of the topic by discussing what artificial intelligence is (and most notably its various specificities for games), and by going through a brief history of the field.

Chapter 2: Designing a single-script robot AI offers a first dive into game AI programming through the basic example of a 2D exploration game. In this chapter, you will see that game AI doesn't have to be complex, and that a single C# script can be enough for modelling simple behaviours.

Chapter 3: What are FSMs? marks the beginning of our study of the most common AI programming tools, starting with the finite state machines (FSMs). This chapter introduces the basics of the technique, as well as its strengths and weaknesses.

Chapter 4: Making a simple guard AI continues on the topic of finite state machines and explores how to apply this architecture to give life to various units in a 3D scene.

Chapter 5: Upgrading your finite state machines wraps up the discussion on state machines by browsing through some famous extensions of the classical FSM model, and highlighting how they can mitigate some of the issues we usually face with state machines.

Chapter 6: Understanding behaviour trees introduces the next important element in a game AI developer's toolbox – behaviour trees (BTs). This chapter explains the fundamentals of BTs, how they differ from finite state machines and why they are an interesting alternative.

Chapter 7: Creating a behaviour tree toolbox then goes through the step-by-step implementation of a custom BehaviorTree C# package to start and apply the theory from *Chapter* This library is an ensemble of utilities for making behaviour tree-based AI in a Unity/C# project, and it contains several handy base objects for creating such structures.

Chapter 8: Implementing a RTS collector AI builds upon the previous chapters and shows how to gradually implement a collector AI for a real-time strategy game (RTS) thanks to behaviour trees. This chapter puts in practice the various notions studied during *Chapter 6* and relies on the BehaviorTree C# package written in *Chapter*

Chapter 9: The reverse-thinking of planners shifts the focus to another type of less-structured AI programming tool called the AI planners. The chapter offers an in-depth summary of automated planning and AI planners in video games, plus a list of interesting implementations.

Chapter 10: Discovering utility-based AI continues this discussion on less-rigid AI techniques by focusing on a more recent tool: the utility-based AI (UBAI). This introductory chapter presents the core principles of this technique, a simple example to understand the thought-process behind it and a quick run-through of the most notable advantages and drawbacks of UBAI.

Chapter 11: Designing a utility-based wizard AI expands on the knowledge from the previous chapter and shows how to apply UBAI to a single-player magic duel game to create a worthy AI opponent for the player.

Finally, *Chapter 12: Expanding your horizons* takes a step back and reflects upon the tools studied throughout the book and the possible extensions or improvements. This final chapter also contains a list of extra resources and creators to check out to continue your exploration of the topic.

Checking out the Github demo repository!

As a complement, I have also prepared a Unity/C# demo project with the scripts and assets for the different examples studied throughout the book.

The Github repository is available for free over here:

https://github.com/MinaPecheux/Ebook-Unity-AIProgramming

USING A COMPATIBLE UNITY EDITOR VERSION

> This Unity project was made in the **2021.3.11f1 LTS** Unity editor. To avoid any incompatibilities, it is best to open it in the same version on your end – or at least, one that is roughly equivalent, like another 2021 Unity editor.

In this repository, you will find the assets and the code for *Chapters 8* and plus all their dependencies. You're more than welcome to download, modify and play around with this project to your liking **for a personal usage** :)

Otherwise, the contents of the Github repository are licensed under the **CC BY-NC 4.0 license**

(The Github repository contains the full license file for more details.)

PART 1

GETTING STARTED

1 - AI in games

Artificial intelligence (AI) is one of those big topics that everyone hear of, but don't always understand completely. The definition tends to vary from one person to another, and over time. For video games, though, it's usually about bringing your non-playable characters to life by simulating some human-like behaviour.

AI has become an essential component of games because, nowadays, both single-player and multiplayer titles are expected to offer rich and dense worlds, with credible characters that act naturally enough to maintain the illusion and keep you immersed. Except that breathing life into a video game character is by no means an easy task – especially since, as humans, we are quite picky about what we consider "believable".

Game AI is therefore a complex subject that is fundamental to providing a good gameplay experience to the players, and AI developers rely on quite the number of techniques to do so. Throughout our journey in the world of game AI programming, we will study the most common ones and work on several use

cases to better understand the strengths and weaknesses of each.

But first of all, let's start by discussing the basics of artificial intelligence and see what we mean by "game AI"; then, we'll talk about the modern machine learning-based AI tools, and how those differ from the AI techniques we use as game developers; finally, we'll have a quick look at the history of the field to get a glimpse of how those various AI techniques appeared.

"AI": a catch-all term?

Over the years, the term "artificial intelligence" has evolved quite a lot, and it has grown to slowly encompass a significant amount of concepts. From coding tic-tac-toe opponents to developing Go players, or designing image classifiers, or making trading assistants, or creating video recommendation algorithms, AI has become a somewhat ubiquitous and yet hard-to-grasp tool of our everyday lives. We hear of expert systems, machine learning, neural networks and predictive algorithms, still it's sometimes difficult to really understand what's hidden behind those words.

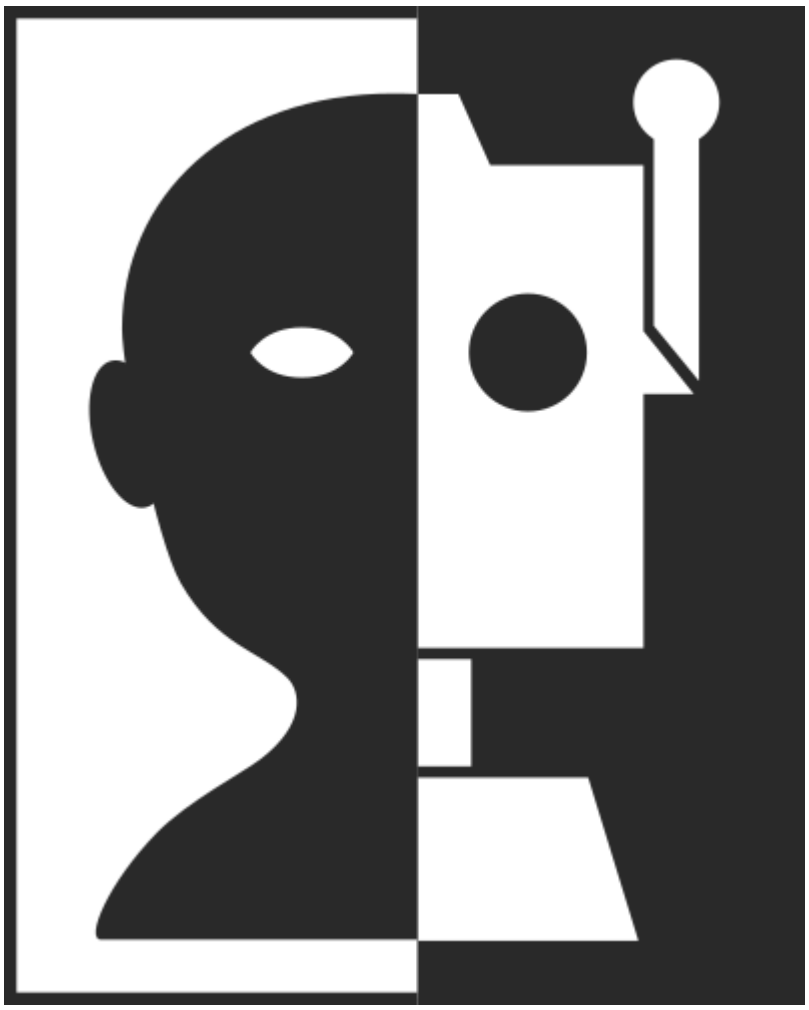


Figure 1.1 – Artificial intelligence is everywhere in our everyday lives and yet, it's sometimes hard to understand how it works exactly!

Yet, it feels like the AI we're talking about in the news is not really about giving a personality to an on-screen enemy character anymore. Which is why, before going further and talking about the "how", I would like to discuss the "what"

and the "what for" of AI, and more particularly AI within the context of video games.

What does "artificial intelligence" really mean?

The notion of "artificial intelligence" is notoriously hard to define.

First, because the concept of "intelligence" is, in itself, quite an umbrella term for a lot of things, and we still don't know exactly how our own brains work. And second, because copying real-life to make an "artificial" version of it in our computers is just as fuzzy a process.

In the context of video games, however, we can refer to Wikipedia's definition, at least as a starting point:

In video games, artificial intelligence (AI) is used to generate responsive, adaptive or intelligent behaviours primarily in non-player characters (NPCs) similar to human-like intelligence.

Wikipedia, "Artificial intelligence in video games", 2023
https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

Again, this is just one possible definition among many. But it describes our topic pretty well, and it highlights a few important points:

Generating **intelligent** Even though artificial intelligence tools can be applied to various game systems, our focus throughout this book will be on the most common acceptation of the word, which is how to use AI to model the behaviour of our non-playable characters (NPCs). In the upcoming chapters, "game AI" will therefore refer to this idea of giving on-screen mobs and characters brains capable of perceiving and reacting to their environment, with a more or less advanced decision logic, to emulate lifelike behaviours.

Generating behaviours *similar* to **human-like** Game AI is very different from academic AI, in that it doesn't aim at replicating our cognitive architecture and crafting an intelligence like ours, but rather focuses on creating adaptive behaviours for NPC entities, so as to face players with challenging enough opponents.

In games, we don't really care how scientifically accurate our brain model is – if it feels right and it reacts right (meaning if it creates the intended experience for the players), then it is good AI logic.

It's just like that famous saying that was supposedly coined by Walt Disney back in the days: our goal as AI designers is not to recreate life, but the **"illusion of life"** (see *Figure*

Figure 1.2 – Video game AI is about creating the "illusion of life", with all its surprises and quirkiness… like in The Last Guardian (Japan Studio and GenDesign, 2016), where the creators gave Trico a behaviour similar to the one of a pet to create an emotional bond with the player (image from:

Actually, all things considered, we have to admit that video games are a very particular kind of artwork. They are an **interactive media** that mixes visuals, sounds, inputs, real-time

simulations and many more systems – and on top of that, their primary goal is to **immerse** you in a fictitious world by appealing to your **willing suspension of** and having you gladly enter and discover this new universe. As such, a crucial idea in game development is that every part of your game has to be crafted *for* the players, for their fun. It might sound obvious, but it actually impacts heavily your whole design process and decisions, because everything has to be done with them in mind.

And, of course, game AI is no different. So whereas academic AI tries its best to understand our cognition and reproduce it on a computer, artificial intelligence in games is devoted to providing players with an entertaining gameplay experience at all costs.

And yet, despite having these opposing visions, it is interesting to see how game AI and academic AI complement one another. In fact, games and AI have always been quite intertwined, the research in one domain pushing forward our knowledge in the other.

In his 2019 book

Playing Smart

, the game researcher and professor Julian Togelius states that these fields depend on each other in three ways:

First, games offer well-scoped and detailed use cases for AI. They are often simplified reproductions of our world with clear rules that can help us train and test AIs in a sandboxed and yet fairly complex environment.

Plus, a video game contains a lot of sub-systems that may use artificial intelligence in different ways, thus allowing us to focus on one specific subdomain and improve our AI tools in this particular area (e.g. dialogue creation, entity behaviour modelling, procedural generation...).

Second, Julian Togelius argues that "AI is the future of video games". For him, giving life to worthy opponents is just one small piece of the puzzle, and artificial intelligence could also be leveraged to create richer "AI-augmented games", for example with procedural story lines or dynamic gameplay experiences.

Third, he believes that studying AI and games together could help us understand intelligence itself – meaning our own cognition. Wanting to replicate human behaviour obviously requires studying it, and Togelius thinks that in our quest for realistic artificial intelligence, we might get a deeper insight on natural intelligence as well.

AI and games are therefore an inseparable pair that ought to lean on each other to expand and grow. And despite game AI being about faking life, it may still teach us quite a lot on our own reasoning and decision making.

With all that said, I don't want to bog down this chapter by spending too much time on searching for the perfect definition for artificial intelligence in games, 'cause that's an unsolvable issue. So let's assume that, for now, we have a first working basis for the notion of game AI. (Don't worry, though: we'll actually come back to this idea of mimicking our brain architecture later on, in *Chapter 9: The reverse-thinking of* when we're a bit more familiar with common AI techniques!)

Instead, I now want to shift the focus slightly and discuss why game AI is also quite different from the "artificial intelligence algorithms" we hear about in the news.

Favouring versatility over expertise

When you think of the modern AI tools we are offered nowadays, be it the famous

Midjourney
image maker or OpenAI's
ChatGPT

language model, you'll notice that while they are really good at what they do, they are also fairly limited.

ChatGPT

, for example, is only able to concoct texts. Those texts may be extremely diverse, ranging from cooking recipes to code snippets or video scripts, but they are still all just texts. My point being that

---

ChatGPT

---

won't ever be able to make you a coffee, or recognise a cat in a picture.

That's because those artificial "intelligences" are, in truth, nothing but a fancy mathematical formula. These amazing tools are powered by machine learning-based deep networks which, at their core, are simply a way to approximate a function to map an input to the right answer – and, because this task is usually too difficult to solve with a written-out formula, we instead design an ensemble of little gauges and variators that altogether compute a somewhat valid result. All those factors are tweaked ever so slightly during the training phase, where we have both the input and the correct answer, and we're thus able to do a feedback loop on the system once it has tried some calculations to have it fit the expected result better.

However, by nature, this architecture implies that we are only able to solve a very specific task, that we can only comprehend situations very similar to the initial training context. Neural networks are therefore first and foremost **experts in a single** sometimes with impressive accuracy and capabilities, but nonetheless a narrow field of competence.

On the other hand, think about what we call AIs in video games. Usually, those mobs, bosses and NPCs move around the game's world and act as if they were real living creatures, switching from one action to the other over and over again to accommodate the current situation. Of course, depending on the genre and the role of the AI, this behaviour might be more or less realistic (just look at the difference between the mobs in

---

World of Warcraft

, and the city people in

GTA V

). Still, overall, the goal of AI in games is always to mimic animal or human reactions to some extent.

A direct consequence of this philosophy is that those game AIs cannot be as specialised as the neural networks: if your game contained soldiers that were able to walk smoothly to a given spot with the perfect trajectory, but couldn't fire their gun or utter a line of dialogue, it would properly feel a bit weird.

Here, it is more interesting to have the unit be average in multiple areas, than exceptional in just one.

Basically, in most video games, AIs need to be way more **versatile** than what classical machine learning can offer – which is why neural networks aren't that common a tool for game AI. Instead, we prefer to rely on more controlled and well-understood techniques, such as the ones that we will study throughout this book: finite state machines, behaviour trees, planners...

"Clever predictability" is the watchword

Alright, so – we know that game AI doesn't share the same goals as academic AI, and that because its whole objective is to please the players, it is more about creating human-like behaviours than truly replicating the exact human brain architecture.

To put it another way: in video games, like in movies, everything's fake. Half of your job as a game AI designer is to **dupe the players** into believing that your entities are more clever than they really are... even if that means intentionally limiting their brains so that the gamers feel more sympathetic, or manually writing some code to teach a boss a really nifty trick.

In fact, if you want your game to be truly enjoyable and interesting for the gamers, there are a few key things you should keep in mind when designing your AIs:

Control & Above all, you need to be able to control how your entities act and predict their behaviour, to anticipate the possible situations and craft a real experience for the players. This is typically something that would be hard to do with machine learning algorithms, since those data science processes are **black boxes** that inherently offer responses tainted with a high degree of

Ensuring that your AIs don't go haywire and express a rational and reasonable behaviour is paramount to making a cool and entertaining slice of fun for your gamers.

And by the way, this notion of predictability is not limited to just the creators – players also like to be able to guess and learn some of the ropes behind your AI, to develop a certain mastery of your game and feel more at ease as they play through it (the countless

---

Elden Ring

YouTube videos showing a perfect boss fight are a good example of that urge for mastery!).

Power In most cases, it is also essential to gently give the players the upper hand, and not just try to crush them with an insanely brilliant AI. A common problem with poorly balanced games is that the players think "the computer cheats".

That's because there is a somewhat tacit rule in the gaming community that video games are the one place where you (the player) can be in control. Multiplayer games are of course a bit different in that regard, since you are confronted to other human agents – but focusing solely on human-VS-AI scenarios, it sounds quite reasonable to assume that the human player should usually win.

Now, this is not to say every game should be casualised and dumbed down; the point is not to insult the intelligence of your community. But you have to maintain a certain power hierarchy that prioritises the players above the machine, and ensure you're not simply imposing an unbeatable adversary on them. Never forget that a game should remain a nice and amusing artwork!

Despite expecting all these common rules and hierarchies (sometimes without even realising it), players are also always

eager to discover brand new entity behaviours that “feel real” and surprise them like a real agent would.

The trick is therefore to find the right balance between lifelikeness and predictability. An enemy that continuously repeats the same action isn’t that good an opponent, but a creature that frantically bursts into ever-changing reactions each second is no better.

Truth is: there is no perfect recipe for this, and it’s more about tweaking and refining your logic until it feels somewhat credible. Randomness can often be a nice way of introducing some “natural quirks” in your entities’ behaviour... but remember that there’s a whole spectrum from pure robots to completely random decision makers, and good AI usually lies somewhere in the middle.

Flaws & Continuing on these ideas of respecting the power hierarchy and creating realistic AIs, it is interesting to note that flaws can actually be an advantage in this case. Although your units should have overall rational behaviours, introducing some errors in their reasoning can make them more credible in the eyes of the players since, as the saying goes: “to err is human”...

For example, having a character get anxious and slowly take more and more bad decisions can be a clever way of making sure the player eventually wins, while covering this little gameplay imbalance with a touching emotional one.

Gameplay Finally, always keep in mind that your AI design should match your gameplay, and be consistent with your game's philosophy. Like all the other game systems, the point is not to have it shine on its own and break the immersion; rather, it has to tie into the narrative, the genre, the players' expectations and the unique mechanics of your artwork.

For example, in the 2016

Doom

reedition, the game creators wanted to emphasise movements and speed. Thus, they made the enemies less likely to hit you when you're running, to encourage this style of gameplay.

Similarly, in the

Assassin's Creed

series, the developers didn't want to overwhelm the gamers with endless waves of enemies all piling up and attacking at the same time. Because, the story revolves around you, the lone and ferocious assassin, you need to remain the main character and overcome most situations with ease. That's why there is a queue system that has the enemies "wait in line"

before entering the fight, and limits the risk of a huge pack flooding you.

Of course, this is just a brief peek at important concepts that we will discuss more thoroughly as we study various use cases and AI design examples.

But anyway – now that we have a rough idea of what game AI is and what it implies with regard to gameplay experience crafting, let's wrap up this introduction by having a quick look at some of the big milestones in the field to learn where the tools we'll work on in this book come from.

A brief history of the domain

Going back to the roots

The concept of artificial intelligence has been around for quite a while: from the Greeks and their mythical automata powered by some strange energy, to Mary Shelley's

Frankenstein

or Fritz Lang's

Metropolis

, the idea of having a manmade creature suddenly live and breath is an old dream that is still vivid in our day and age.

Yet these studies of formal reasoning and mathematical logic couldn't really be applied to artificial entities before the mid-1950s because, well, we didn't have that many artificial entities before that date!

In the second half of the century, however, AI research boomed and led to the invention of various algorithms, structures and architectures that allowed programmers to create ever-more realistic behaviours in their machines.

And, as far as game AI goes, there are two important milestones that marked the minds:

In 1997, both AI specialists and video game creators joined in awe when IBM's

---

Deep Blue

AI managed to beat Garry Kasparov at chess. The computer relied on a simple minimax algorithm that, today, would probably be laughed at for being too basic and naive. But, at the time, it wasn't about inventing finesse and re-enacting subtle reactions – having a program triumph over a (human) world champion at a game as complex as chess was already an extraordinary achievement.

Twenty years later, in 2015, Google repeated the performance when

AlphaGo

managed to defeat Fan Hui and Lee Sedol at Go. This time, not only had the computer beaten professional human Go players, but it had won at an extremely complex board game with way more branching scenarios than chess. A task that would have been impossible to solve with a crude algorithm like the minimax, but that the Google engineers managed to overcome thanks to modern deep learning and neural networks.

These two events are essential because they are proof that artificial intelligence is a powerful tool that can solve very complex tasks, and that as we develop new systems we open the door to taking on new challenges.

Now, this is not to say that nothing happened in between, or that game AI just spontaneously appeared in 1997. The modelling of entity behaviour in games has its own history and key moments.

An evolutive toolbox

Back in the 1970s and 1980s, arcade games already faced the players with computer-controlled opponents. (It was actually the

only way of having endless runners and shoot 'em ups, which would force people to put another coin in the machine and finance the industry!)

The behaviours were simple and fully scripted, showing a mix of **static patterns** and **breaks** to react to specific actions from the players, but they were still basic game AIs. For example, Mario's infamous monkey enemy in the 1981

Donkey Kong

---

game didn't have a very realistic behaviour, but it was a challenging opponent nonetheless.

Game creators knew that this hand-authored behaviour scripting would soon be too limited and hard to scale for multiple creatures, though. That's why, at the same time, games like

---

Pac-Man

---

introduced one of the founding blocks of game AI to this day: the **finite state** This allowed the developers to power the ghosts in the maze and have them go from one behaviour to another depending on the current game state (see *Figure*

(We'll study FSM in detail in *Chapters 3 to* but in short they're about chopping down the behaviour of your unit into several states, or contexts, and transitioning between each of those states when certain triggers are activated to have the entity switch to another action.)

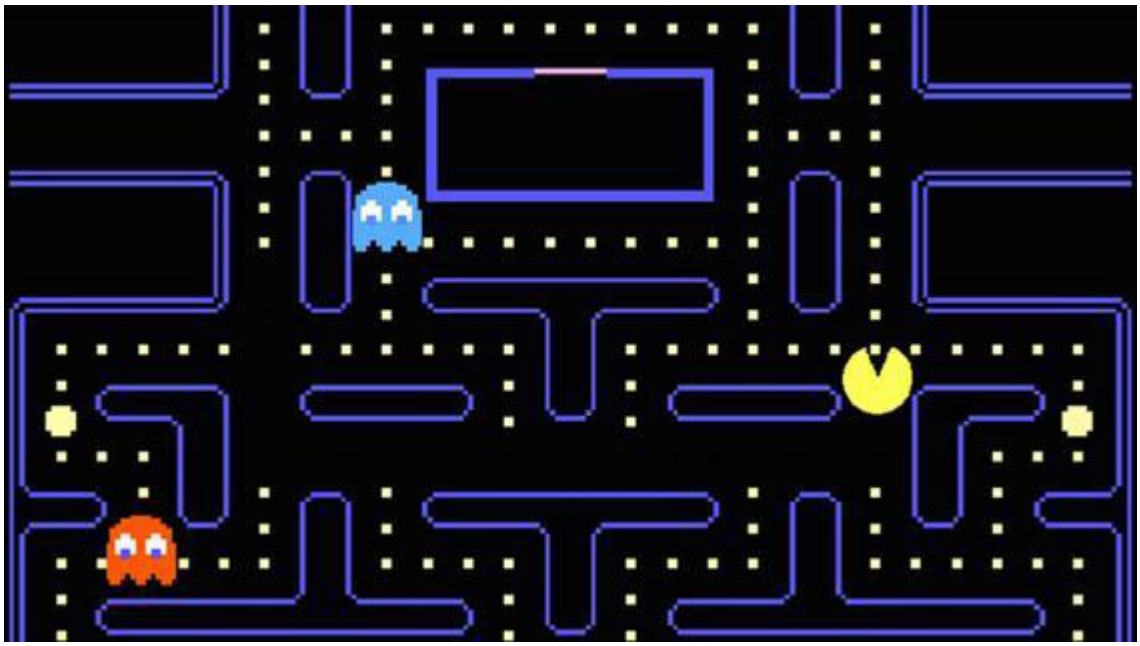


Figure 1.3 – Many video games such as Pac-Man rely on finite state machines to model the behaviour of enemies (image from:

In spite of its apparent simplicity, the FSM architecture can actually be used to model quite advanced behaviours, and it quickly became an essential tool in the game AI developer's toolbox. So much so that renowned games like Metal Gear Solid , Halo or even

Batman: Arkham Asylum

relied on state machines to model the behaviour of the enemies.

But then, in the early 2000s, the

Halo

team decided to level up from finite state machines for their new game:

Halo 2

. Instead, they used a new type of architecture for their entities, called **behaviour** This allowed them to populate wide-opened areas with multiple enemy types that all responded in unique ways and exhibited more robust and flexible behaviours... and as we'll explore in *Chapters 6 to* using behaviour trees also provided the team with a modular easy-to-debug and easy-to-maintain architecture that could create more evolved artificial brains.

From that point on, behaviour trees would become one of the go-to techniques for creating game AIs and numerous famous titles rely on those, such as

Just Cause 3

or

Spore

.

However, parallel to the rise of behaviour trees, game AI developers continued their relentless search for new ways of modelling human-like behaviour.

And in 2005, the developer behind the AI of

F.E.A.R.

, Jeff Orkin, proposed yet another system for modelling entity behaviour: **Goal-Oriented Action Planning** (or GOAP). The idea was to try and decouple the decision logic from the actual action execution by giving the AI a long-term objective, and then having it plan a series of actions to reach this desired world state step by step. This was the first real instance of an AI planner being used for a video game; since then, other teams like the Guerilla Games developers behind the

Transformers

series of games have worked on this AI technique, and even offered an evolution in the form of **hierarchical task networks** (or HTNs). (We will discuss all of this in depth in *Chapter*

Nowadays, game AI is a fascinating field teeming with new ideas all over the place, as designers try and push the limits of the existing models and potentially invent brand new ones. This tenacious research has led to alternative architectures like the **utility-based AI** (that we will study in *Chapters 10 and* or the more narration-oriented at the heart of

Left 4 Dead

and

Alien: Isolation

that try to dynamically author the game's rhythm and adjust the pacing on-the-fly to maximise the "drama" for the player.

What's the future of game AI?

It is impossible to predict how game AI will evolve, and many people in the community wonder what new architectures and structures we might come up with in the upcoming years.

Some think that the existing tools will eventually constrain us too much. The AI researcher and professor Pedro Domingos pushed this idea forward in his 2015 book

The Master Algorithm

, saying that we shouldn't rely on static manually-authored AI systems anymore, but instead find ways to have the AIs learn by themselves and craft their own decision logic.

Others prefer to keep their old pans because that's where they cook their best broths, and they are exploring possible improvements or evolutions of existing tools, just like the HTNs that were a follow-up on Orkin's GOAP planner.

All in all, we'll simply have to wait and see how the field develops in the years to come – but if the past is any indication, we are in for quite the surprises!

Summary

In this first chapter, we've discussed some base principles of artificial intelligence for games.

We began by focusing on the term "artificial intelligence" in itself, and in particular the specificities of AI in games compared to the modern meaning of the word (that more often refers to machine learning and neural network-based tools).

Then, we drafted a quick history of the field to understand where the tools we'll study in the rest of the book come from.

In the next chapter, we'll take the plunge and explore how to design a basic robot AI using just a single C# script.

2 - Designing a single-script robot AI

There is this fantasy with game AI that it has to be really hard and difficult in order to be a tad valid. That it's only by using the most cutting-edge tech and the most recent algorithms that you stand a chance at creating a credible behaviour. That it's only by making your models bigger, and larger, and more detailed, and more precise, that you will be able to design good AIs.

And yet, in software design, there is an interesting design principle called **"Keep It Simple,** or KISS. In a nutshell, it's the idea that, when you're faced with a problem, you try and find the most straight-forward solution. Typically, in computer science, it's about using a basic but easy-to-understand algorithm, rather than a complex and intricate one, when possible.

So, in this chapter, I want to mitigate this mythical vision of AI, and discuss how we can create a basic but credible AI for a little explorer robot with just a single C# script.

Preparing our base AI logic

Going blind into the implementation is always a bad idea in software design but, with AI, it's particularly detrimental because you can fake quite a lot before you realise the underlying architecture is not robust and could suddenly backfire.

That's why to start off, we're going to do a quick overview of what it is we're trying to achieve.

A run-down of the situation

For this first use case, our goal will be simple:

We will assume that we have a little robot in a scene, that can move forward in either one of four directions: up, right, down or left. Its movement will be constrained to a 2D grid, meaning that the robot takes discrete steps from cell to another.

This robot will be placed in a closed room where there is also a hidden gem; its objective will be to explore the room and find this reward.

Whenever the robot is faced with a wall, it will turn to the side to unlock and continue its route.

To try and locate the reward, the robot will have a very dumb routine: it will just move at random until it stumbles upon the cell with the gem.

As you can see, this logic isn't very complex. It clearly doesn't necessitate we dive into high-end techniques and code up some mysterious chain of commands to analyse everything and react in some extraordinary way. We only need to implement these specific checks and actions, in order to get this controlled and predictable behaviour.

Once again: Keep It Simple, Stupid.

We'll code the logic of our robot in a C# class called We're also going to assume that the scene has already been prepared, and that we have a pre-existing GameManager script that handles three tasks:

spawning the reward at a random position in the room

executing the robot's AI logic every X seconds (to let us visualise its progression step-by-step), by calling the public

Execute() function on its referenced RobotAI component

checking if the robot has reached the gem's tile, in which case we show a victory panel

WANNA CHECK OUT THE ASSETS?

> Don't forget that all the code and assets for this chapter are available in the Github repository, over here: https://github.com/MinaPecheux/Ebook-Unity-AIProgramming. Feel free to have a look if you want to check out how the scene is built, or you want to compare your own work with the provided RobotAI script!

So, at this point, we just have a top view of our room showing: a visible grid of walkable white cells, a border of blue wall cells that block the robot, and the robot and the gem positioned on this grid (see *Figure*

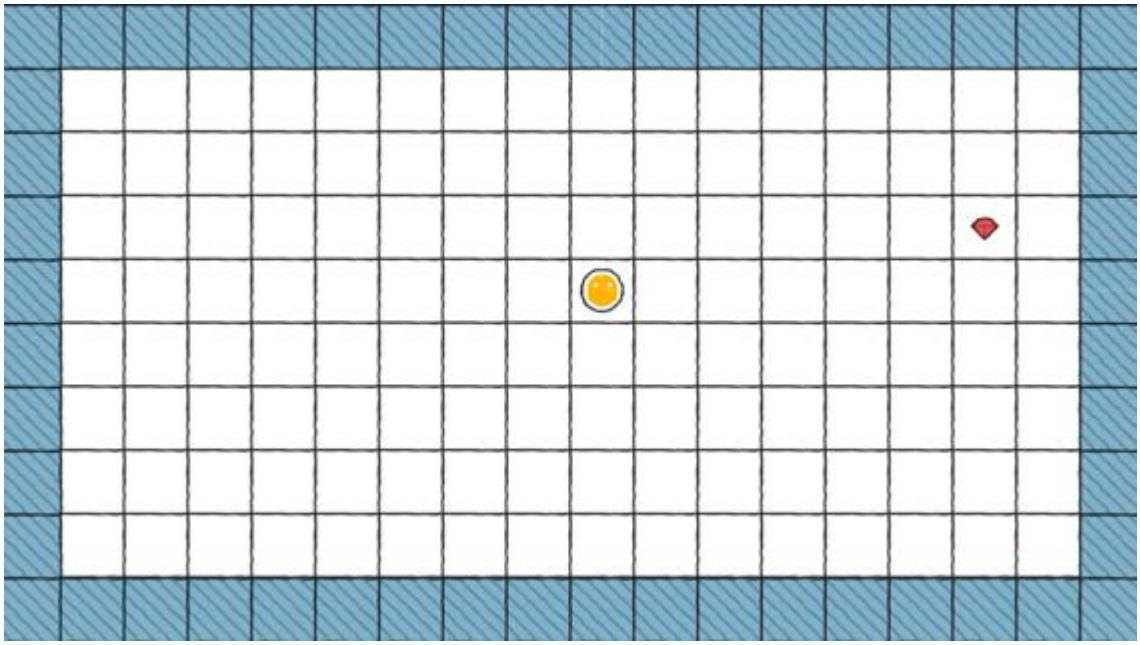


Figure 2.1 – Screenshot of the current game state, without any logic. White cells are walkable by the robot, blue cells are blocking walls, the robot is located at the origin of the grid and the gem is spawned at a random position inside the room.

We'll consider the initial position of the robot is the origin of the grid (i.e. the cell with coordinates (0, and that the gem is spawned on a random walkable cell with an X coordinate between -8 and and a Y coordinate between -4 and As an example, in *Figure* the gem is located in (6,

For now, our explorer robot of course doesn't do anything – it just stays at its starting position. The first step is therefore to give it a basic movement logic, to have it translate forward according to its current direction.

Setting up our move logic

To do this, let's create our AI C# script, and place it on our robot game object. Then we'll open it and remove the template code from Unity to get just an empty class:

RobotAI.cs

```
1  using UnityEngine;
2
3  public class RobotAI : MonoBehaviour {}
```

We know that we want our robot to move forward in its current direction. So, to begin with, we need to store this direction. For that, we're going to define:

a private _direction variable that contains the current direction of the robot as an integer being "up", 1 being "right", 2 being "down" and 3 being "left")

some static data to easily match a direction to its corresponding offset along the X and Y axis – for example, for the “up” direction, we get an offset of 0 along the X axis, and of +1 along the Y axis

---

RobotAI.cs

```
1  using UnityEngine;
2
3  public class RobotAI : MonoBehaviour {
4      private static Vector2[] _OFFSETS = new Vector2[] {
5          new Vector2( 0,  1), // up
6          new Vector2( 1,  0), // right
7          new Vector2( 0, -1), // down
8          new Vector2(-1,  0), // left
9      };
10     private int _direction;
11 }
```

Here, I’m using a C# array to easily store the offset for each direction, stored as a Unity

Now, the nice thing is that just by calling I could get the offset along the X and Y axis to apply to the current position of my robot to move it in the "up" direction.

We've said in the previous section that our robot's logic needs to be defined in a public Execute() function so that the GameManager can call it regularly and update the game state. We also want this GameManager to have an idea of where the robot is currently, to check whether it has found the reward or not. So we'll solve both problems at once by creating a new Execute() function in our script that returns a tuple of integers that are the current coordinates of the robot:

RobotAI.cs

```
1  using UnityEngine;
2
3  public class RobotAI : MonoBehaviour {
4      private static Vector2[] _OFFSETS = ...
5      private int _direction;
6      private int _x, _y;
7
8      public (int, int) Execute() {
9          return (_x, _y);
10     }
```

```
11 }
```

That's great but, for now, the robot still isn't moving. We need to properly initialise its coordinates at the start, and then make sure that, each time the GameManager refreshes the game state and calls the Execute() function of our RobotAI script, the robot takes one step in its current direction.

To implement all of this, we'll proceed in three steps:

1. First, we'll create a private method in our RobotAI class to set the position of our robot at a given couple of (x, y) coordinates. This function will both update the _x and _y variables, and actually move the sprite on the board by setting the position of its Transform component:

RobotAI.cs

```
1  using UnityEngine;
2
3  public class RobotAI : MonoBehaviour {
4      private static Vector2[] _OFFSETS = ...
5      private int _direction;
```

```
6     private int _x, _y;
7
8     public (int, int) Execute() { … }
9
10    private void _SetCoordinates(int x, int y) {
11        _x = x; _y = y;
12        transform.position = new Vector3(x, y, -1);
13    }
14 }
```

(Note that I'm setting the Z coordinate of my robot sprite to -1 to make sure it is closer to the camera and thus appears in front of the grid.)

2. Then, in the Awake() method of our script, we'll initialise our robot's coordinates to the origin of the grid:

RobotAI.cs

```
1  using UnityEngine;
2
3  public class RobotAI : MonoBehaviour {
4      private static Vector2[] _OFFSETS = ...
```

```
5     private int _direction;
6     private int _x, _y;
7
8     private void Awake() {
9         _SetCoordinates(0, 0);
10    }
11
12    public (int, int) Execute() { ... }
13    private void _SetCoordinates(int x, int y) { … }
14 }
```

---

3. Finally, in our Execute() method, we'll use our static _OFFSETS array to get the offset to apply to our robot – considering its current direction –, and use our new _SetCoordinates() method and this offset to truly move the robot on the board:

RobotAI.cs

```
1  public (int, int) Execute() {
2     Vector2 d = _OFFSETS[_direction];
3     _SetCoordinates((int) (_x + d.x), (int) (_y + d.y));
4     return (_x, _y);
```

```
5 }
```

If we re-run the game now, we see that the robot indeed moves one cell at a time at regular intervals in its initial direction (which is 0 by default, meaning "up").

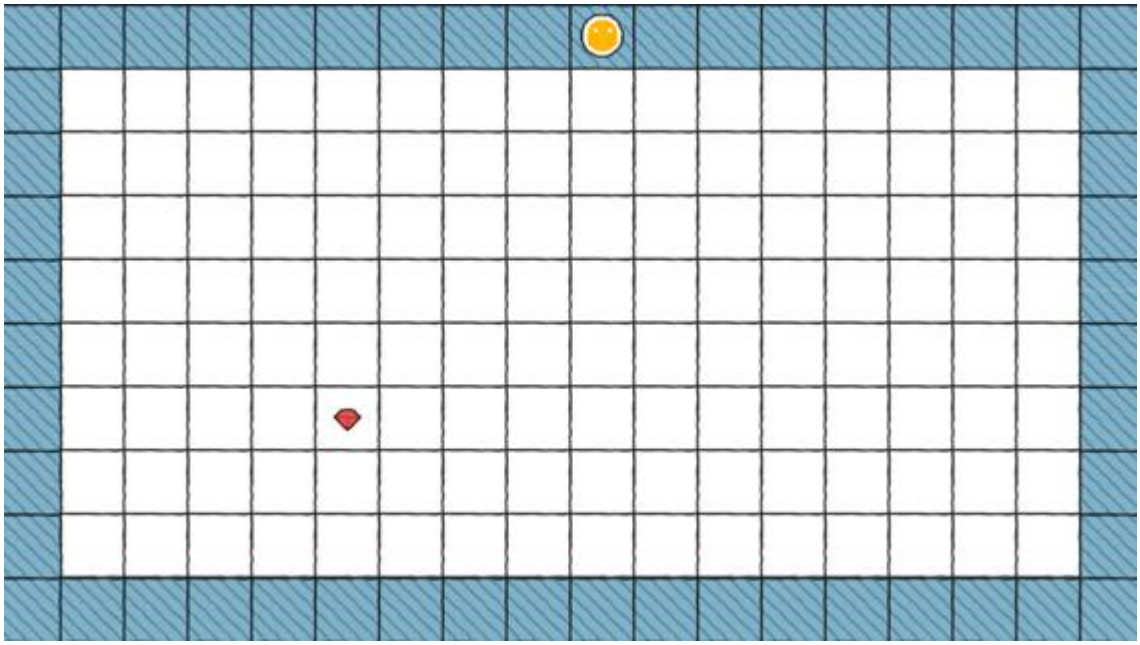


Figure 2.2 – Screenshot of the game state after a few iterations, with the robot moving up one cell at a time... and ignoring walls!

Except that, of course, our robot completely ignores walls, and doesn't really stand a chance at finding the gem unless it happens to be right in front of it. That's why we have to make sure that the robot checks for blockers and learns how to turn.

Adding the walls check

Now that our robot can move around, we need to ensure it stays inside the room. And, for that, we have to ensure it can detect and react to a wall in front of it.

When I prepared my scene, I made sure to give each of my wall tiles a collider (those colliders are shown as green squares around the blue tiles on *Figure*

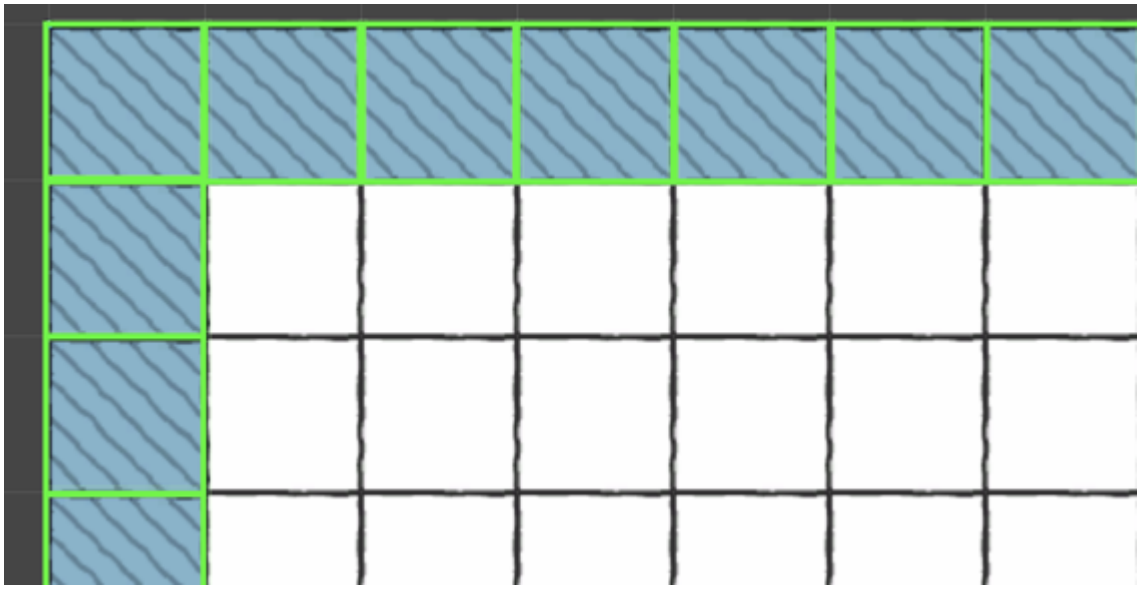


Figure 2.3 – Visualisation of the wall 2D colliders in the scene view

This means that we can use Unity's 2D Physics tools to check for nearby blockers – more precisely, we can use 2D

To put it simply, a raycast is like an infinite beam fired from a given position, in a given direction. Any object that has a collider and is on the path of this beam can then be detected and reported in our script. We can also force the length of the raycast to a specific value to create a sort of proximity sensor for objects in a certain direction.

In our case, because we're working in 2D, the beam will ignore the depth and we will therefore detect collisions in the XY plane, thanks to the built-in Physics2D.Raycast() function. This method returns a RaycastHit2D object which contains a collider reference to the first object with a collider it hit in the scene, if any; else the collider reference is

Here, we want to check for a wall in front of our robot. So we're simply going to create a 2D raycast with a length of 1 that starts at the position of our robot and is oriented in the same direction as the entity, by re-using our Vector2 movement offset:

RobotAI.cs

```
1  public (int, int) Execute() {
2      Vector2 d = _OFFSETS[_direction];
```

```
    var hit = Physics2D.Raycast(transform.position, d, 1f);
    if (hit.collider == null) { // no wall block: move forward
        _SetCoordinates((int) (_x + d.x), (int) (_y + d.y));
    }
    return (_x, _y);
}
```

Thanks to this additional check, our robot will now be able to stop whenever it is faced with a wall, which is nice... but also makes for an even shorter ride!

What we'd like is for walls not to block our AI but rather have it change its direction and try another path. We can do this quite easily by:

1. Creating a new method in our class, that updates the _direction variable and the sprite's actual rotation to get matching visuals:

RobotAI.cs

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

```
3
4  public class RobotAI : MonoBehaviour {
5      private static Vector2[] _OFFSETS = ...
6      private int _direction;
7      private int _x, _y;
8
9      public (int, int) Execute() { ... }
10     private void _SetCoordinates(int x, int y) { … }
11
12     private void _SetDirection(int direction) {
13         _direction = direction;
14         transform.rotation = Quaternion.Euler(0, 0, -90 * _direction);
15     }
16 }
```

---

2. Calling this function in the Execute() method if we hit a wall – here I'll arbitrarily force my robot to turn right if it hits a wall, by incrementing its _direction variable by one and optionally looping back to 0 with a modulo if need be:

RobotAI.cs

```
1  public (int, int) Execute() {
```

```
2     Vector2 d = _OFFSETS[_direction];
3     var hit = Physics2D.Raycast(transform.position, d, 1f);
4     if (hit.collider == null) { // no wall block: move forward
5         _SetCoordinates((int) (_x + d.x), (int) (_y + d.y));
6     }
7     else { // else turn right
8         _SetDirection((_direction + 1) % 4);
9     }
10    return (_x, _y);
11 }
```

Our robot can now stop and turn when confronted with a wall, which means it won't ever get out of the room. Instead, if it is faced with a wall, it will turn right and continue its adventure:

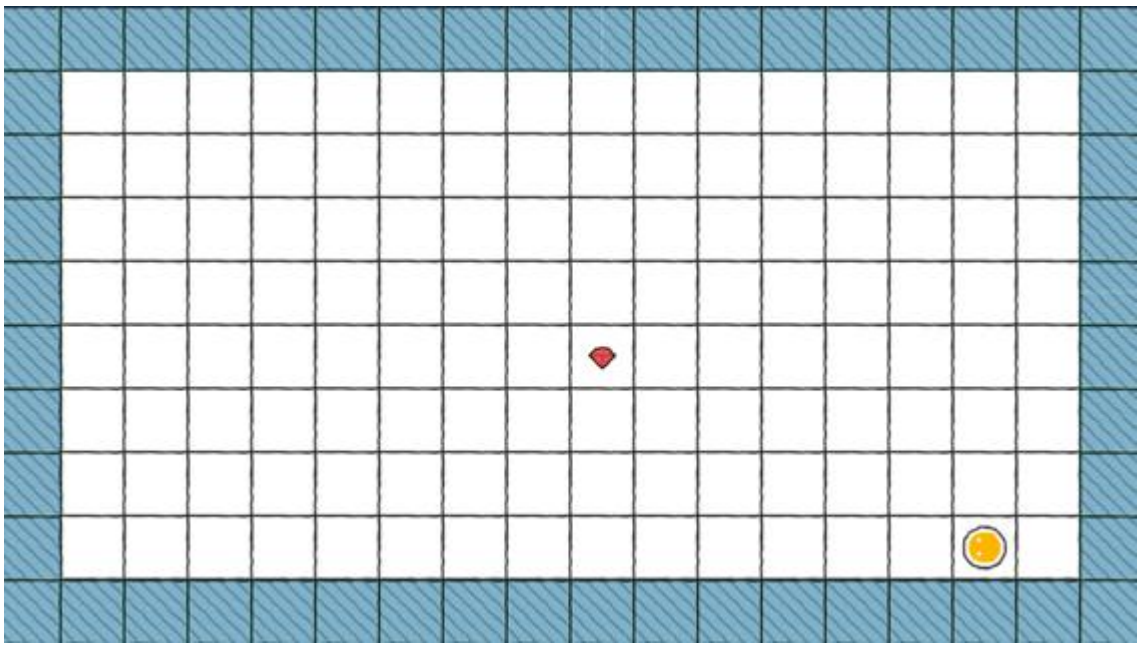


Figure 2.4 – Screenshot of the game state after a few iterations, with our robot AI now able to avoid walls and turn right whenever it encounters one

Moreover, the interesting thing is that this behaviour isn't limited to simple room geometries like the one in *Figure* Even if we have blocked out chunks, or small corridors, our robot will still be able to navigate on the walkable cells and avoid the blocked ones:

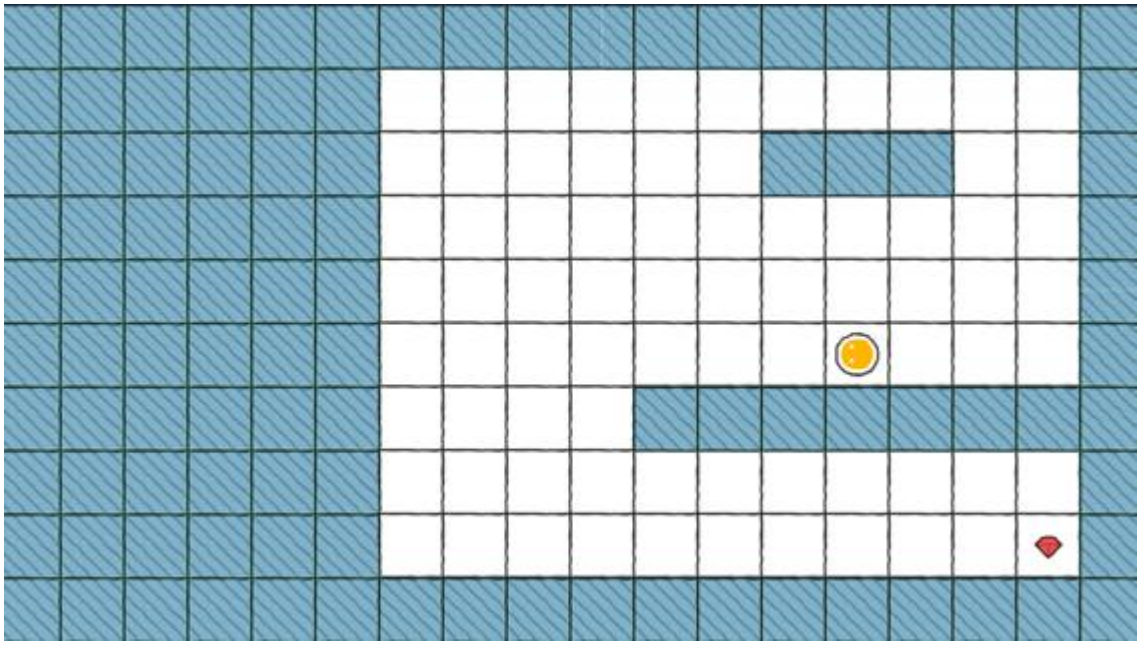


Figure 2.5 – Screenshot of the game state after a few iterations in a more complex room

However, with this logic, the robot will still miss the reward most of the time!

Consider the basic room shown in *Figure* Well, at this point, if the robot hits a wall, it will then turn and move until it hits the next one; then it will turn again and move until it hits a third wall; and so on. It will thus end up doing an endless loop on the outmost ring of cells of the room. And if the gem isn't in one of those cells, the robot has absolutely no chance of exiting this pattern and finding the reward.

The last thing to implement in our robot's logic is therefore the random move mechanics...

Implementing some randomness

Until now, we've gradually built a simple but efficient behaviour for our robot, that allows it to move and turn in the room without ever hitting a wall. But this is not enough to actually find the hidden gem.

In truth, for this specific use case, there are two very different reasons we can add randomness in our logic for: completeness, and realism. Let's see why in more details!

Exploring thanks to randomness

If you're anything like me, hearing that we're going to solve our exploration problem by pouring in randomness will feel very weird at first. I remember when I was still a bit new to coding and I ran into this idea a long time ago; I just had a hard time processing how moving about in an erratic manner could ever help my AI find its target.

And yet, think about it this way: in this example, we haven't given our robot any sensor or radar. It doesn't have any tool to track the reward and directly turn to the right spot. Instead,

its only hope is to cover as much ground as possible to eventually stumble upon the gem.

To be fair, a random walk is not always the most optimal technique, and it's not the only solution. Typically, going to the top-left corner of the room and then systematically going down and up each column could yield a better (and faster) result.

But using this methodic process could also be longer than a random search in some cases. If the reward happens to be in the bottom-right corner, then the robot would need to walk through the entire grid until it finally reaches the right cell. A random search, on the other hand, could potentially bring the AI to the gem quite rapidly, especially if it spawned somewhat close to the robot at the start. Using randomness is therefore kind of a gamble on the average case.

ALGORITHMS & COMPLEXITY

> In computer science, there is a specific subfield dedicated to studying what is called **computational complexity**. This notion refers to the amount of resources (usually, mostly execution time) required to run a specific algorithm.

> It is thus possible to evaluate the complexity of an algorithm to determine how long it would take to complete the task in the worst-case scenario, the average-case scenario and the best-case scenario. Though, oftentimes, we focus on the worst-case scenario, since this gives us an upper bound on the algorithm's complexity.

> In our case, we see that a naive approach like the systematic grid search has a very high complexity in the worst-case scenario, that is proportional to the size of the grid. The random approach has a complexity that is harder to evaluate, but that may sometimes vastly improve the execution time – in particular for large rooms.

Also, the cool thing is that, for us, adding this random behaviour isn't too hard. We just have to create a new function in our RobotAI class to allow the entity to pick a random direction among the valid ones (i.e. ignoring the directions that would lead the robot into a wall):

RobotAI.cs

```
1 using System.Collections.Generic;
2 using UnityEngine;
3
```

```
4  public class RobotAI : MonoBehaviour {
5      private static Vector2[] _OFFSETS = ...
6      private int _direction;
7      private int _x, _y;
8
9      public (int, int) Execute() { ... }
10     private void _SetCoordinates(int x, int y) { … }
11     private void _SetDirection(int direction) { … }
12
13     private void _ChooseRandomDirection() {
14         List<int> possibleDirections = new List<int>() { 0, 1, 2, 3 };
15         for (int i = 0; i < _OFFSETS.Length; i++) { // check invalid dirs
16             var hit = Physics2D.Raycast(transform.position, _OFFSETS[i], 1f);
17             if (hit.collider != null) possibleDirections.Remove(i);
18         }
19         int dirIndex = Random.Range(0, possibleDirections.Count);
20         _SetDirection(possibleDirections[dirIndex]);
21     }
22 }
```

(Note that we need to import the System.Collections.Generic package in order to use the C# List data structure.)

And then, we'll simply use this function in our Evaluate() logic. To avoid the robot changing direction too often, we can wrap

the call to our _ChooseRandomDirection() method in a 50-50% chance if-check, thanks to Unity's Random library:

RobotAI.cs

```
1  public (int, int) Execute() {
2      Vector2 d = _OFFSETS[_direction];
3      var hit = Physics2D.Raycast(transform.position, d, 1f);
4      if (hit.collider == null) { // no wall block: move forward
5          _SetCoordinates((int) (_x + d.x), (int) (_y + d.y));
6          if (Random.Range(0f, 1f) < 0.5f) // 50-50% chance
7              _ChooseRandomDirection();
8      }
9      else { // else turn right
10         _SetDirection((_direction + 1) % 4);
11     }
12     return (_x, _y);
13 }
```

And here we are! Our robot AI is now able to change direction at random to explore the grid and, ultimately, find the reward. It may take some time, but it will eventually happen, and we have coded up a simple artificial intelligence that,

given a situation and a set of behaviour rules, reacts to its environment and executes various tasks to solve the problem the best it can...

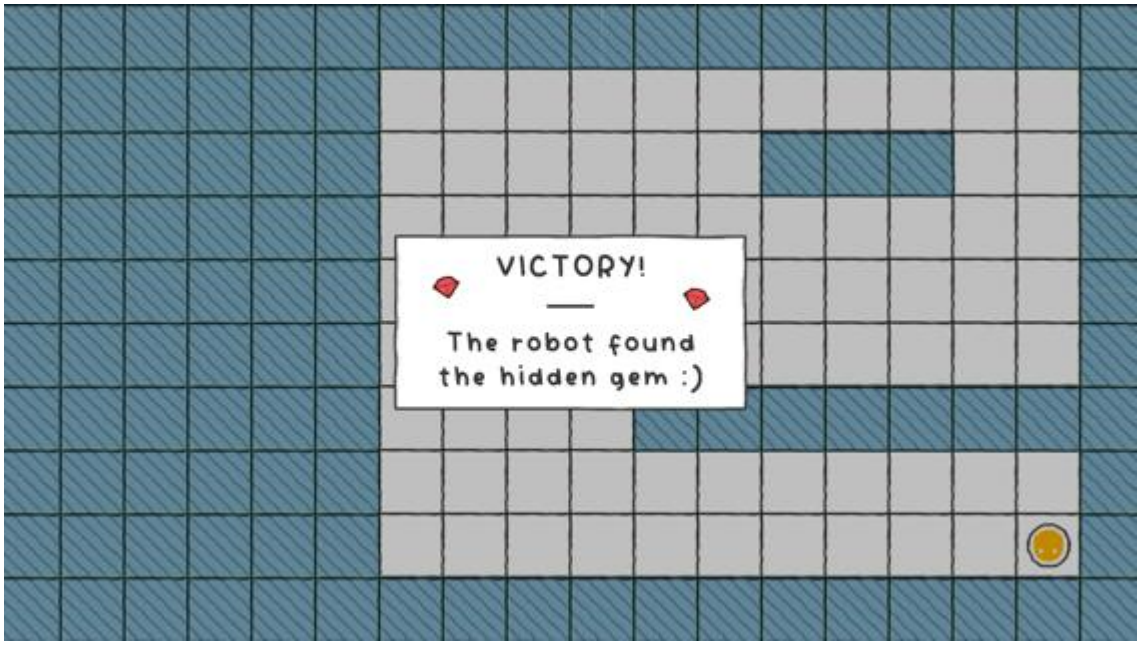


Figure 2.6 – Screenshot of the final game state with the victory panel, once the robot has reached the cell of the hidden gem

But wait – what about this "randomness for realism" idea?

Predictability VS randomness

Just before we end this chapter, I want to quickly discuss a fundamental idea in game AI programming, which is the **balance** of control and lifelikeness.

As we've said in *Chapter 1: AI in* for AI developers, it is important to always keep in mind two core principles: predictability and accountability. Contrary to the "modern AI", with its machine learning algorithms and its black-box neural networks we keep reading it about in mainstream press, game AI has to integrate into the rest of the gameplay and, as such, it requires more control over the results. When you implement the behaviour of an entity, you need to be able to assert that it will stay within certain bounds and react appropriately to the player's actions.

But this minimum level of predictability is an objective for you as an AI creator, with respect to your team and leads. From the player's standpoint, ideally, it would be best if the AI didn't just loop with the same reaction over and over again... otherwise, it will become boring very quickly.

For example, at the moment, our robot will always turn right when it is faced with a wall. This is not bad *per* we could decide that it's just the way it goes in our game, but it's totally predictable.

For the current use case, this predictability has no real consequences, apart from us being able to anticipate little chunks of our robot's path. However, if it was a turn-based game where we controlled our one robot on the board and

played against the AI to collect the reward as quickly as possible, knowing that this behaviour logic always turns right when faced with a wall would be an advantage for us.

Wouldn't it be better if, instead, the robot could "surprise" us a little?

A quick way to introduce this layer of "realistic randomness" in our RobotAI script would be to slightly modify its wall avoidance response. Namely, rather than always going right, the robot could pick between turning left and turning right at random, like this:

RobotAI.cs

```
1  public (int, int) Execute() {
2      Vector2 d = _OFFSETS[_direction];
3      var hit = Physics2D.Raycast(transform.position, d, 1f);
4      if (hit.collider == null) { // no wall block: move forward
5          _SetCoordinates((int) (_x + d.x), (int) (_y + d.y));
6          if (Random.Range(0f, 1f) < 0.5f) // 50-50% chance
7              _ChooseRandomDirection();
8      }
9      else { // else turn right or left
10         bool turnRight = Random.Range(0f, 1f) < 0.5f;
```

```
11          _SetDirection(turnRight
12              ? ((_direction + 1) % 4)
13              : ((_direction - 1 + 4) % 4));
14      }
15      return (_x, _y);
16  }
```

This is obviously not a big dose of randomness, and it doesn't impact the robot's behaviour overall, but at a micro-level, it helps give it a bit more personality.

Finding the right balance between "predictable" and "surprising" is not easy, and yet it's one of the key to making AIs that are both enjoyable and challenging for the players. As an AI designer, you should always try to aim for entities that feel as real and spontaneous as possible, and stop just before they become too crazy to debug and analyse.

Summary

In this chapter, we've worked on a very basic robot logic and implemented our first AI, using only a single C# script.

We've discussed how this idea of looking for easy-to-understand solutions follows the KISS principle, and how it can be a good

starting point for easy use cases like this one.

On the other hand, it's of course important to understand that the KISS principle has limitations and that you cannot expect to reproduce any behaviour with the simplest script. Here, we've seen that we can fairly quickly setup a movement logic for a robot in Unity and C# that doesn't require any human intervention to survive and adapt to the world, and is therefore an example of artificial intelligence. But it is clearly not scalable to more complex AI logics.

To be honest, I would understand that you feel slightly underwhelmed right now. This book was supposed to teach you about AI techniques and design patterns, and up to this point I've basically said that you should relinquish your high hopes and cram everything in a single C# script. Pretty bad, huh?

Well – fear not!

Now that it's clear that AI doesn't *have to* be complex, let's see how it *may* be. And, to begin our journey in the world of real AI programming, the next chapters will be about one of the most common tools in the AI developer's toolbox: the finite state machine...

PART 2

FINITE STATE MACHINES

3 - What are FSMs?

In the previous chapter, we've discussed how we could model a basic robot behaviour using just a single C# script. This highlighted how AI doesn't have to be complex and all high-end tech... but it also quickly showed the limitations of this naive process.

So now that we have this first reference in mind, it is time to go deeper into the world of AI programming and study real structures and techniques for giving your game entities a bit more brains.

And to start things off, we're going to talk about finite state machines. In this chapter, we'll see how they work, why they can be useful for game AI, and when they are the best fit.

Learning the basics

We've said it before: modelling the behaviour of an on-screen character so that it feels natural and intelligent is a complex task. And while simplistic if-else code can handle some very

precise and constrained reactions, they lack the flexibility and adaptiveness we usually want for our game AIs.

That's why, over the years, people have developed more evolved structures for representing behaviour and creating more lifelike characters – among which finite state machines.

Understanding the finite automaton structure

Finite state machines (or are a specific kind of mathematical model that can be used in game development to design and build the behaviour of an entity. Specifically, they are what we call an or in other words a self-operating machine capable of following a sequence of operations and, to a certain extent, respond to inputs with pre-determined reactions.

The core idea behind an FSM is that:

The system has a finite number of possible

It can only be in one of these states at a time – this state is called the active, or **current**

It starts in a given **initial**

It also contains **transitions** that allow it to switch from one state to the other, and thus update its current state. These transitions are triggered by various

These transition-triggering inputs can be internal or external.

That's quite a lot of terminology, but the concept is actually fairly intuitive. To sum up, you start in some given state, a specific context that makes your entity perform a particular set of actions or checks, and then your entity may react to one or more events to trigger a transition to another state, and therefore change its current context.

Figure 3.1 shows a simplified and theoretical **finite state machine** with just hypothetical names and values, to give us a first idea of the overall workings of this structure:

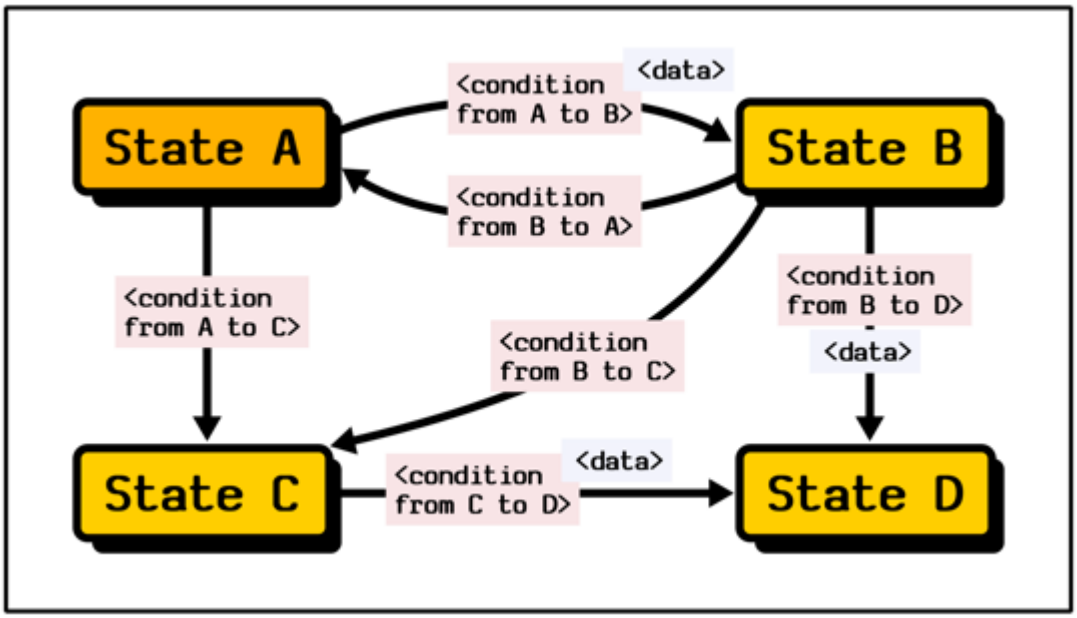


Figure 3.1 – Diagram of a simplified and theoretical finite state machine with four states

In this image, you see the different conventions I'll use in this book to represent FSM diagrams:

States are shown as yellow or orange rectangles.

The initial state is shown as an orange rectangle.

Transitions between states are shown as bold arrows, oriented from the current state (at the time of the transition) to the new state.

On each transition, we show the trigger condition in pink, and, if need be, the data that is transmitted from the current state (at the time of the transition) to the new state in grey.

We see that this data transfer is optional: some transitions have extra data attached to them, others don't.

Figure 3.1 also contains an example of a **final state** – here, State D is final because there is no transitions to go back to another state in the machine once you've reached it. So, basically, any path that passes through this state will end up stuck at this point.

DETERMINISTIC VS NON-DETERMINISTIC STATE MACHINES

> For now, we'll keep things relatively light and only discuss **deterministic** state machines. In other words, we'll consider that the transitions between our states always use the exact same condition, and result in the exact same outcome for the exact same situation. This is nice to get the gist and learn the fundamentals, because it ensures that the transitions (and thus the entity's behaviour) are **predictable**.

> However, as we've peeked at in *Chapter 2: Designing a single-script robot AI*, randomness can be useful in making the logic more "realistic"

> – so we'll definitely dive into **non-deterministic**, *aka* **stochastic** FSMs in more details in *Chapter 5: Upgrading your finite state machines*, and see what it means exactly to introduce randomness in a state machine.

We've now got a bit of knowledge on FSMs, on what they look like and what kind of objects we find in them. But, to really get the gist, let's take a little example.

Studying a basic use case

Finite automata are quite an intuitive behaviour modelling technique, yet designing a state machine from scratch can be a challenge when you're new to the concept. In particular, properly segmenting your entity's actions into states and understanding the conditions that trigger a transition from one state to another can be difficult when you're trying to model a really clever minion.

For this first example, we're not going to aim for clever. We'll just take a simple AI logic, and walk through the design process step by step to learn some tricks to "coding intelligence" with FSMs.

Namely, here, we'll model the AI of a dragon guarding a treasure.

An overview of our AI logic

Whenever you want to build an AI for a game, it is important to precisely define the different behaviours of this AI, and also its scope. You need to know exactly what the entity can and can't do, in order to implement enough logic but also enough barriers.

For a real game, this task is usually handled by several people in the team: the game designers, the AI designers, the character artists and many more collaborate to give an identity to the different units, and include them seamlessly in the rest of the gameplay.

Since we don't have this whole team of experts here, we'll decide for ourselves what AI we want to implement, and we'll keep things simple. The idea will be to invent a behaviour for our dragon that sounds reasonable from a gamer's point of view, even if it is not too evolved; and then gradually transform this description into a list of states and transitions to see how a finite state machine could represent this behaviour.

So let's say that:

Our dragon is a static creature sitting on its pile of gold and treasures. Its objective is to guard this fortune at all costs by defending it against all the heroes coming its way. We'll thus consider the creature has a fixed field of vision, and that any living thing that enters this radius becomes a target.

If the dragon has a target, then it tries to roast it with regular bursts of flame. In gameplay terms, the dragon has an infinite timer that loops and, at the end of each cycle, it runs some damage logic on the target.

However, because it is an old beast, the dragon also cannot stay awake all the time, and it sometimes sleeps for a while. During these phases, it doesn't actively look for threats... but its predatoristic nature will wake it up if an enemy stands really close, and in that case the dragon will start to attack as described before!

Clearly: not the most intelligent creature. But, in the context of a video game, I would argue it is enough, and that it meets the common expectations players may have for a dragon guarding its treasure.

If we analyse this description, we see that we can decompose the AI's behaviour in three states:

Guarding: The dragon is awake and looking around for potential threats.

Attacking: The dragon is awake and currently has an enemy in range – so it's trying to eliminate it to protect the treasure, and does so by dealing some damage at regular intervals.

Sleeping: The dragon is unconscious and isn't ready to fight (but it can still react to the presence of enemies if they are close to it).

Initially, the dragon will be in its Guarding state (though, technically, we could also decide the initial state is the Sleeping state).

At this point, we therefore have the beginning of a state machine, with three disconnected states:

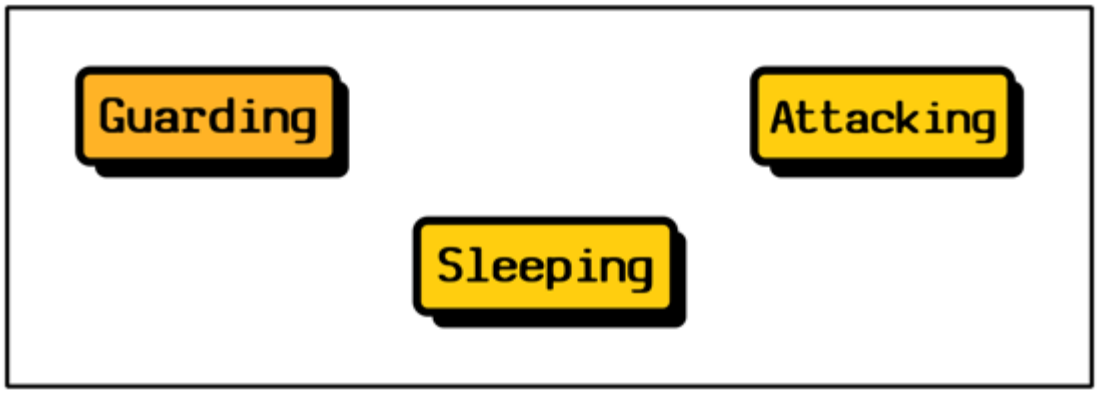


Figure 3.2 – Diagram of our (incomplete) dragon AI finite state machine

The next question we need to answer to complete the state machine of our dragon AI is: what triggers the transition from one state to the other? Or, to put it another way – what is the **decision logic** of this entity?

Remember that these transitions may happen between any two states in the diagram, so in order to be thorough, we have to check all the possible pairs of states – even though some transitions might ultimately be impossible.

We'll proceed methodically:

Guarding-to-Attacking: This transition is totally possible. It should occur if a unit enters the field of vision of the dragon, and designate this unit as the new target for the AI.

Attacking-to-Guarding: Conversely, if the current target of the dragon leaves its field of vision, we'll consider that the beast instantly forgets about this enemy and goes back to looking for threats.

This is typically something that could be improved to make a more realistic behaviour, for example by extending the field of vision for that transition to mimic an **attention** but we won't go into these considerations for this first example.

Guarding-to-Sleeping: This transition is also valid. The idea would probably be to have a global counter in the Guarding state that checks how long it's been since the last fight – and if this "idle time" is too long, then the dragon transitions to its Sleeping state.

Sleeping-to-Guarding: Similarly, we could have a timer in the Sleeping state to automatically return to the Guarding state after a while.

Sleeping-to-Attacking: As we've mentioned in our behaviour description above, although the dragon is less reactive when asleep, it should still be able to spot close dangers. So this transition is possible, except it relies on another radius check than the Guarding-to-Attacking one.

Attacking-to-Sleeping: This transition is invalid for our AI. It wouldn't make sense for our dragon to be able to instantly go to sleep in the middle of the fight. Thus we won't add this arrow to our diagram, and we shouldn't implement any logic to trigger this transition.

All in all, the diagram of the finite state machine for this AI could be the following:

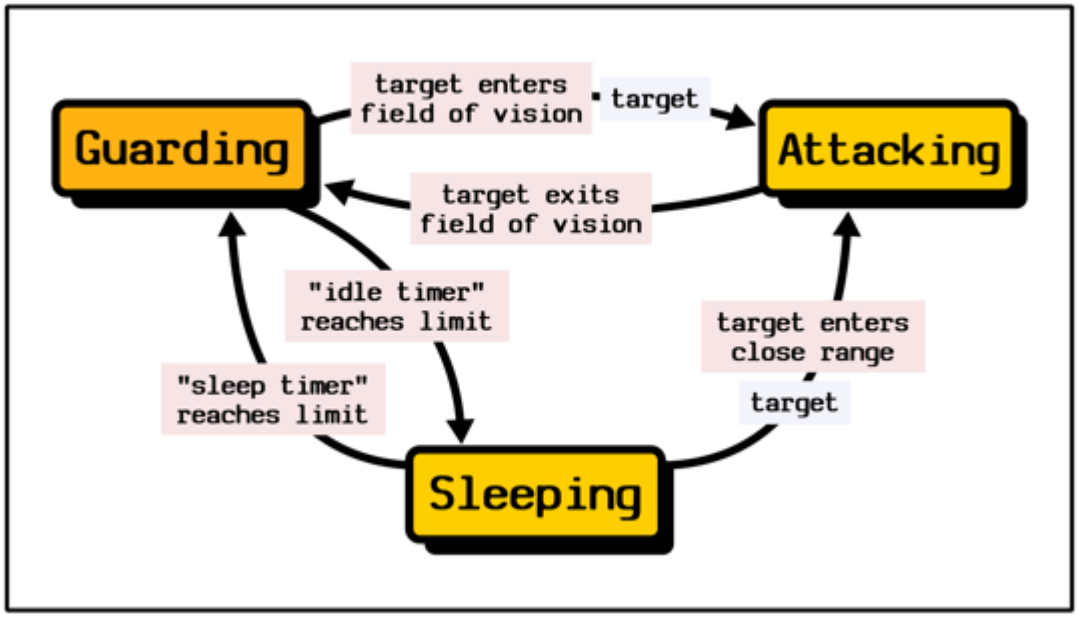


Figure 3.3 – Diagram of our final dragon AI finite state machine

As you can see on *Figure* even a behaviour as simple as the one of our treasure-guarding dragon already makes for quite a lot of states and transitions. On the other hand, it is pretty readable and intuitive; someone who doesn't have any prior knowledge of the intended dragon's behaviour would probably be able to guess and outline most of it just by looking at this diagram.

So we've successfully designed our very first FSM-based AI! It's crude and quick, but it shows the basic process for building a finite state machine: identify the states, and then identify the valid transitions and their triggers for each pair of states.

Pretty straight-forward, right?

Well, now that we have this basic example in mind, let's put on our developer hat and see how these states and transitions could actually be coded up in C#.

A peek at the implementation

Although we'll discuss all the ins and outs of making a finite state machine AI in the upcoming chapter, *Chapter 4: Making a simple guard* it can still be interesting to get a bird's-eye view of the matter at hand and draw a big picture of how FSMs can be implemented in a Unity/C# project.

So, in this subsection, we'll focus a bit more on what these "states" and "transitions" could be from a programmer's point of view, and what C# tools we'll need to use to create them...

Inheritance and shared behaviours

After this quick study of an application of finite state machines, you might be thinking that the logic of each state is pretty unique to each situation. That there is no way of abstracting commonalities between two state machines and that, apart from having states and transitions, two FSM-based AIs don't share any similarities.

Well, that's absolutely true.

The logics don't have anything in common, *apart from this base*

If you're somewhat versed in C#, and in object-oriented programming, you probably see where I'm going with this... Sure, we can't devise a grand overarching logic that fits every entity behaviour, but we can still abstract some FSM mechanics and global tools, typically thanks to inheritance.

Inheritance is one of the core pillars of **object-oriented programming** (or OOP). It is the idea of building a structure or a class based on another object to re-use, extend and possibly specialise the logic of this base and adapt it to your own use case. We typically say that we create **derived classes** that inherit from a **base**

With inheritance, you thus get a **hierarchy** of objects, with the most generalist at the top and the most expert at the bottom.

Each level further specialises the behaviour and interface of the structures but potentially also shares part of the implementation with the rest of the objects.

In C#, inheritance usually relies on two main keywords: abstract and

'Cause oftentimes, you'll want to declare your base classes as abstract, to prevent one from instantiating those (incomplete) blueprints and to tell the compiler some of the logic will be defined in the derived classes. And then, inside, you'll define either abstract, virtual or normal methods.

Abstract methods just declare a prototype, but they don't have any logic. They therefore enforce that derived classes implement this logic, which is a good trick to avoid programmers forgetting to code up their own logic!

Virtual methods have both a prototype and a default implementation at the base class level. This means that they allow us to share some logic between all derived classes by default, for this method, unless one of the derived classes overrides this logic with its own.

Normal methods (i.e. methods that don't have neither the abstract nor the virtual keyword) are declared and defined at the base class level, and cannot be overridden by the derived

classes afterwards. They're a way of defining a fixed shared behaviour for all instances of classes at this level or below in this inheritance scheme.

Inside your derived classes, you can then use the override keyword to implement the logic of an abstract method, or to replace the logic of a virtual method that you inherit from your base parent class.

The typical example is to have an Animal base class that defines a MakeSound() method; and then to create the Cat and Dog derived classes that adapt this MakeSound() function to either print "miaou" or "woof":

Animal.cs

```
1  public abstract class Animal {
2      public abstract void MakeSound();
3  }
```

Cat.cs

```
1  using UnityEngine;
2
3  public class Cat : Animal {
4      public override void MakeSound() => Debug.Log("miaou");
5  }
```

Dog.cs

```
1  using UnityEngine;
2
3  public class Dog : Animal {
4      public override void MakeSound() => Debug.Log("woof");
5  }
```

Here, we have no way of directly instantiating the Animal class; we can only make instances of the derived classes, Cat or And if we call the MakeSound() method on our instances, the debug will depend on the type of derived class we've instantiated, since the logic is implemented per-derived class with overrides.

IMPORTANT DISCLAIMER

> These few paragraphs are a very (very!) light overview of huge object-oriented programming concepts, and they definitely shouldn't be considered a full introduction to the topic. There are many more subtleties to take into account, and many more tools to discuss to really grasp the reach, advantages and shortcomings of OOP.

> I've made my best to discuss what will be useful for us in this book, but if you're not familiar with these principles, feel free to have a more in-depth look!

Now, that's great – but why is all of that interesting to us?

Because, in short, by using abstract classes, and virtual or abstract methods, we'll be able to prepare some agnostic objects that encapsulate the essence of a FSM state, and that integrate some fields and functions that can be used in any state machine.

We won't fill the functions with any real logic, obviously, since we saw in the *Studying a base use case* section that the logic depends on the behaviour you're modelling currently... but we'll "prepare" those functions, we'll declare them in these abstract objects so that the C# compiler knows any instance in our inheritance chain *necessarily* has these methods. This will allow us to define a basic toolbox for FSM programming that takes care of the core state machine system (for example, setting up an initial state, or properly updating the machine when we switch from one state to the other) and provides us with easy-to-use and ready-to-be-derived classes.

I'm aware that this is a lot to take in – so we'll continue this discussion in the next chapter, when we get into the nitty-gritty and really code up this FSM toolbox, because I think working on a hands-on example will help clear things out.

For now, let's just leave it at that and say that we have this "abstract state class". There is still a question we need to ask ourselves: what exactly should this class contain? Or, in other words: what is this abstract structure that is common to all FSM states?

Defining entry points

In the first section of this chapter, *Learning the* we've seen that the states of a finite automaton are like contexts – they tell

the entity what actions and checks it should do right now, and what conditions it should test to potentially transition to another part of the machine.

If we analyse this logic, we see that any FSM state can therefore be chopped down in three phases:

Initialisation: The logic that is run when the state machine transitions to this state, for example to setup some variables, or even update other components on the entity to have it match this new context.

Update: The logic that is run continuously while the state machine is in this state (i.e. while this state is the current state of the FSM).

Shutdown: The logic that is run when the state machine transitions from this state, for example to clean up variables, or reset other components on the entity to have it "forget" this context.

Again, here, we're not talking about implementing a specific logic for the initialisation phase, or a particular clean up behaviour. We're just saying that, from a purely theoretical standpoint, the logic of an automaton state can be fully specified by injecting the right commands in one, two or three of these phases.

Our “abstract state class” will therefore rely heavily on the concept of entry points – roughly put, they’re those “injection points” where programmers will be able to integrate their own logic to specialise the state and adapt it to the current use case. More precisely, we’ll need to define three methods that match each of the aforementioned phases:

an Enter() function for the state’s initialisation

an Update() function the state’s update

an Exit() function for the state’s shutdown

Also, it’s important to note that these three methods will be virtual since, technically, each could be optional. There are cases where you won’t need to do anything specific to initialise the state; others where you don’t have any continuous logic to run during the update, etc. So, by making the methods virtual, we’ll ensure that programmers can override them if they want to, but they can also simply leave a default empty behaviour for this specific phase if they prefer.

By the way, this idea of having the derived classes “hook themselves” in the right spot to inject their logic is actually a fundamental technique in Unity game development, since it’s

partially what gives their power to classes. Defining an a an a LateUpdate() or even an OnCollisionEnter() method in a Unity MonoBehaviour script is just a way to tell the engine that you want this particular piece of code to be executed at a precise moment during runtime.

Even though it doesn't use abstract and virtual methods, this process of defining methods is therefore a way to inject your custom logic into Unity's

UNDERSTANDING UNITY'S LIFECYCLE

> Roughly put, Unity's script lifecycle is the pre-determined order in which event functions are executed in the editor and/or at runtime. Typically, that's why the Awake() method of a Unity C# script will *always* execute before its Start() method – and that's also why your Physics-related logic should usually be done in the FixedUpdate() rather than the Update().

> For more info, you can have a look at the official Unity docs over here: https://docs.unity3d.com/Manual/ExecutionOrder.html.

Advantages & drawbacks of FSMs

With this overview of finite state machines in mind, let's now examine some of their benefits and some of their limitations to get a better grasp of when they are most useful.

Centralising the logic

One of the major strengths of finite state machines is that, by definition, each state class contains all the logic of its state. Contrary to the single-script AI we coded in *Chapter* where the Update() function had to mix all the actions and checks together, finite automata rely on better-scoped C# files.

For example, the state classes encapsulate all of the relevant component references and local variables to describe, maintain and even shutdown the state. They'll also directly handle the proper initialisation and clean-up of everything – and the nice thing is that all of this happens in one clearly defined C# class. (We'll see some applications of this in the next chapter, when we implement our simple guard AI with FSMs, since some of our code will be about starting the right animations on our 3D model by referencing and communicating with its Animator component.)

This makes for a nicely **centralised** logic, and therefore a cleaner codebase. If you want to know how the Walk state of

an entity works, just look up its Walk C# class and you should find everything you need in this one place!

A readable and yet *relatively* scalable model

A consequence of this centralisation is that finite state machines are quite readable and easy-to-understand at first glance for you, and for your teammates. As opposed to more evolved models like the behaviour trees that we'll study in *Chapters 6 to* or the planners and utility-based AI we'll explore in *Chapters 9 to* state machines are straight-forward to conceptualise and implement.

The fact that we use states goes well with our intuitive mental representation of an entity's "activity", and similarly the fact that we use transitions and triggers matches our concept of an agent that "observes and reacts". This model is therefore pretty logical to most people, and we can quickly get experienced enough to design advanced logics with this tool.

And even better: despite FSMs being somewhat easy to learn, they can still model complex behaviours when need be! Scaling up the "intelligence" of your FSM is mostly about adding more states and handling the new transitions in your system – don't get me wrong, this can require some brainstorming and hardcore design process when the machine gets really big, but at least the process is mapped out.

However... this advantage is also one of the FSM's biggest flaw – because having that clear a structure also means it is rigid, and constrained.

Can you be "too structured"?

State machines are great for quickly tidying up your AI logic and making a real architecture, with well-defined and logical chunks, because they inherently impose this strict structure of a finite set of states and transitions.

But this strictness is not all roses...

A frequent problem with FSMs is when you have to model the behaviour of an entity that has states that are almost identical, but not exactly the same. Picture a little fighter that can run empty-handed or run with a gun (as long as its energy hasn't depleted), and that keeps an eye out for various bonuses around him.

On the one hand, depending on whether the unit has a gun or not, the animations will be different, the move speed might change, the possible interactions with the enemies and the environment could be modified as well... basically, it's like you have two states.

On the other hand, in both cases, you want to move in the given direction if there is a target location, you want to check for pickups and bonuses around the unit, and you want to stop if the energy is exhausted... so, it's like you have kind of the same state?

With FSMs being so rigid, you don't really have a choice here but to code both states side-by-side, and copy-paste a lot of logic. This implies **code** which is always risky cause it can quickly lead to

Although there are some improved state machine models that try and mitigate this kind of issues, as we'll discuss in *Chapter* we see that FSMs also enforce constraints on the developers and, in the worst case scenario, severely limit the re-usability of the logic. You *may* luck out and get states that are agnostic enough of the context to transfer to another entity, or to another part of your unit's AI, but more often than not you'll need to write a new state class to perfectly describe this exact state, and differentiate it from the pre-existing similar-but-unfit ones.

And, unfortunately, this proliferation of states is tightly linked to one last drawback of finite state machines: the coupling issue.

The coupling issue

A key thing to keep in mind when designing finite state machines is that, as we've already mentioned, the logic of a state is centralised in one specific C# class. This is excellent in terms of codebase readability, but it also means that all the transitions from a given State A have to be handled and checked for in the code of the matching StateA class. So a significant chunk of the state's logic will actually be for checking if, when and how the state machine should transition to another state.

With FSMs, the decision logic and the execution logic are all bundled together in the state's class, and you have to manage the two together while maintaining a consistent state context.

The problem is that as we add states to our FSM, the number of possible transitions will increase too, and very rapidly – at an exploding rate. For math lovers, we can easily deduce from a FSM diagram like *Figure 3.4* that the number of transitions in a fully connected state machine containing **N** states is **N • (N –**

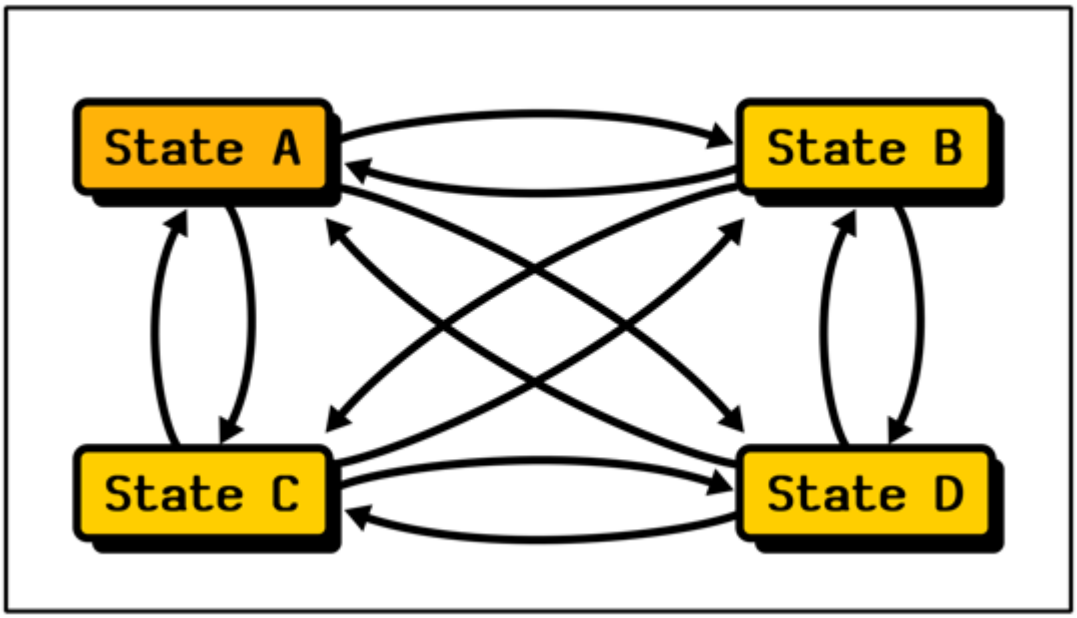


Figure 3.4 – Diagram of a fully connected finite state machine with 4 states and 12 transitions

(If we go from 4 to 5 states, the maximum number of transitions will therefore go from 12 to 20; and then going from 5 to 6 states will bring the maximum number of transitions up to 30.)

So, coming back to our "the state logic has to handle all the transitions" concept from before; we see that if the behaviour you're modelling with your state machine becomes too complex and the number of states gets too large, then this transition-related chunk of the state's logic will also grow uncontrollably.

Moreover, in order to tell the state machine where to transition to, the state C# class actually needs a reference to the new state at the other end of the transition... which means that the states in your FSMs have to be, to a certain extent, aware of the rest of the system. They obviously don't need to know the exact logic inside each other, but they have to know what other states exist around them. This idea that you have some strong dependencies between two objects in your code, that one cannot function properly if the other isn't there, is called It's usually something we try to avoid because it makes the overall architecture harder to develop and maintain. Most notably, it hinders unit testing and self-contained iterations, since we basically need to keep the whole system packed together to properly satisfy all the dependencies.

The exploding growth of the machine's size therefore combines with the high coupling of the state logics to produce a very bad combo. If we ever decide to add a new state to the system (along with some transitions to actually get to it of course), then we'll potentially need to re-visit the logic of every other state to handle their transition to this new state – and have these new checks fit with the rest of the previously written behaviour!

To conclude, let's re-iterate: finite state machines are an amazing tool for designing and building simple behaviours, especially if the entity has clearly defined and distinct action

contexts, but it can also get out of hand if the number of states and transitions grows too much.

Summary

In this chapter, we've talked about the base principles of finite state machines.

We began by studying the core objects this model relies on (the states, the transitions and the inputs), before taking a basic example to really understand the architecture. Then, we briefly discussed some C# tools and concepts that will be useful for implementing state machines in our Unity project, as we'll study throughout *Chapter*

Finally, we highlighted the essential pros and cons of FSMs, and in particular we discussed how this structure is easy to start with, but also limited in terms of scalability.

In the next chapter, we will apply this theory to a real game development example by first implementing our own FSM C# toolbox, and then creating a simple finite state machine-based guard AI.

4 - Making a simple guard AI

In *Chapter 3: What are* we introduced the fundamentals of finite state machines, and we discussed how this common AI programming technique can help us design and create pretty powerful behaviours for our entities thanks to states and transitions.

In this chapter, we're going to continue on this topic and discover how to use this architecture to model a simple AI for a guard (and its enemies) to have him patrol along a path, spot invaders, chase after them, fight, and eventually eliminate the menace before resuming his patrol.

Before we dive into our actual use case and start implementing our guard's AI using finite state machines, however, let's first take a bit of time to prepare some utilities and base classes to build this kind of structure.

Creating the FSM building blocks

Our goal here will be to put the theory we've seen in the last chapter in practice to make our own FSM C# package with

two abstract classes, BaseState and a that any developer will then be able to inherit from to code their own state machine-based AI logic.

USING NAMESPACES

> In this chapter, we will use the concept of C# namespaces in several places. If you're not too familiar with them, C# namespaces are a handy way of isolating some scripts as a package, here typically our base FSM tools as an aptly named FSM module, so that they can then be easily re-imported and used elsewhere in a Unity/C# codebase with the using keyword.

> Furthermore, in the Unity project shared in the Github repository of this book (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming), I've also used C# Assembly Definition assets to separate the different parts of the codebase and speed up compilation. If you're curious about what Unity Assemblies are, you can have a look at the repo, or you can check out this page from the official Unity docs: https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html.

Setting up our base structures

Because our BaseState and StateMachine classes will be just blueprints for our states and FSMs without any real AI logic inside, we will write these classes as **C# abstract** As we saw in *Chapter* this will prevent the users of our library from using these classes directly – instead, developers will have to

implement their own behaviour and adapt the BaseState and StateMachine to the current context.

So we can start slowly by setting up two C# files in our Unity project, BaseState.cs and with the following content:

BaseState.cs

```
1 namespace FSM {
2     public abstract class BaseState {}
3 }
```

StateMachine.cs

```
1 namespace FSM {
2     public abstract class StateMachine : MonoBehaviour {}
3 }
```

In these two code snippets, you see that:

We wrapped our two classes in the FSM C# namespace thanks to the namespace keyword.

We made sure to add the abstract keyword to our class declarations.

The StateMachine class derives from Unity's built-in MonoBehaviour C# class, because we'll want this script to use the usual Unity lifecycle hooks like etc.

The BaseState class is fairly easy and quick to code – as discussed in the last chapter, we simply need to specify the Exit() and Update() entry points. In this example, we'll also add one entry point for the LateUpdate() to get a bit more granularity into Unity's scripts lifecycle. And as a bonus, let's add a name to our state for debugs, if need be, and a reference to the StateMachine instance the state lives in to make it easier to call state transitions further down the line:

BaseState.cs

```
1  namespace FSM {
2      public abstract class BaseState {
3          public string name;
```

```
4       protected StateMachine _fsm;
5
6       public BaseState(string name, StateMachine fsm) {
7           this.name = name;
8           _fsm = fsm;
9       }
10
11      public virtual void Enter() { }
12      public virtual void Update() { }
13      public virtual void LateUpdate() { }
14      public virtual void Exit() { }
15  }
16 }
```

You'll notice that here, I've marked my Update() and LateUpdate() methods with the virtual C# keyword. This way, this methods will necessary exist for all class instances, and we'll be able to call them from the StateMachine class – which is key to actually running our FSM logic during the game's lifecycle.

Indeed, to properly execute the AI's behaviour and have the entity really act, our StateMachine script should do the following:

1. First, the StateMachine has to keep a list of all the states in this specific FSM instance. Here, I'll actually use a **C# Dictionary** to easily map the states to unique string codes – this will make it easier to retrieve and use these states later on:

StateMachine.cs

```
1  using System.Collections.Generic;
2
3  namespace FSM {
4      public abstract class StateMachine : MonoBehaviour {
5          protected Dictionary<string, BaseState> _states;
6      }
7  }
```

Don't forget that, in order to use C# we need to import the System.Collections.Generic package. Also, you see I made our _states variable protected so that this data is and only available to instances of a class.

A nice thing with using a Dictionary is that, now, we can quickly get a state by its unique code, like this:

StateMachine.cs

```
1  using System.Collections.Generic;
2
3  namespace FSM {
4      public abstract class StateMachine : MonoBehaviour {
5          protected Dictionary<string, BaseState> _states;
6          public BaseState GetState(string code) => _states[code];
7      }
8  }
```

2. Then, our class should also have a reference to the current state of the FSM, as well as a method to set up the initial state of the machine upon start.

So we'll declare a variable called _currentState and make it private because, this time, even our classes don't need access to it – it's just used here inside the StateMachine class itself.

Parallel to that, we'll also prepare an abstract method, to get the initial value for the _currentState variable in the Start() lifecycle hook. Since it is abstract, GetInitialState() will be

implemented in our derived classes and taken care of by the users of our FSM package. But we still have to declare its prototype, and here we know that it should return a BaseState instance and take no parameters.

All of this gives us the following code:

StateMachine.cs

```
 1 using System.Collections.Generic;
 2
 3 namespace FSM {
 4     public abstract class StateMachine : MonoBehaviour {
 5         protected Dictionary<string, BaseState> _states;
 6         private BaseState _currentState;
 7
 8         void Start() {
 9             _currentState = GetInitialState();
10         }
11
12         protected abstract BaseState GetInitialState();
13         public BaseState GetState(string code) => _states[code];
14     }
15 }
```

Of course, when we set a new state as the active one in a FSM, we should always call its Enter() entry point in case it requires some initialisation. So the Start() method of our StateMachine class actually needs to also call the Enter() method of our initial state, if there is one:

StateMachine.cs

```
1  using System.Collections.Generic;
2
3  namespace FSM {
4    public abstract class StateMachine : MonoBehaviour {
5      protected Dictionary<string, BaseState> _states;
6      private BaseState _currentState;
7
8      void Start() {
9        _currentState = GetInitialState();
10       if (_currentState != null) _currentState.Enter(null);
11     }
12
13     protected abstract BaseState GetInitialState();
14     public BaseState GetState(string code) => _states[code];
```

```
    }
}
```

---

3. The next step is to use this _currentState variable to execute our FSM behaviour and run the state's logic at each frame in our Update() and LastUpdate() lifecycle hooks:

StateMachine.cs

```
using System.Collections.Generic;

namespace FSM {
    public abstract class StateMachine : MonoBehaviour {
        protected Dictionary<string, BaseState> _states;
        private BaseState _currentState;

        void Start() { ... }

        void Update() {
            if (_currentState != null) _currentState.Update();
        }

        void LateUpdate() {
```

```
15            if (_currentState != null) _currentState.LateUpdate();
16        }
17
18        protected abstract BaseState GetInitialState();
19        public BaseState GetState(string code) => _states[code];
20    }
21 }
```

---

4. Last but not least, we need to implement the state transition logic for our FSM, which consists in exiting our current state, updating the _currentState variable to the new value, and entering the new state:

StateMachine.cs

```
1  using System.Collections.Generic;
2
3  namespace FSM {
4    public abstract class StateMachine : MonoBehaviour {
5        protected Dictionary<string, BaseState> _states;
6        private BaseState _currentState;
7
8        void Start() { ... }
```

```
 9        void Update() { … }
10        void LateUpdate() { … }
11
12        protected abstract BaseState GetInitialState();
13        public BaseState GetState(string code) => _states[code];
14
15        public void ChangeState(BaseState newState) {
16            if (_currentState != null) _currentState.Exit();
17            _currentState = newState;
18            if (_currentState != null) _currentState.Enter();
19        }
20    }
21 }
```

Populating our state transitions with data

As a little extra, there is an improvement we can make on the ChangeState() method of our StateMachine class, which is to optionally accept additional input parameters. This will allow us to easily transfer data from one state to another.

For example, if we have some state dedicated searching for a target, it can be interesting to convey the reference of the target we found to the next state (as we'll see in more details

in the next section, when we implement our guard FSM-based AI).

To do this, a cool trick is to mix together the params keyword and the C# object "lazy" type.

The **params keyword** is a way to tell the program to bundle all the arguments passed at the end of the call in an array to retrieve them more easily – it's just like the variadic arguments in C or C++. For example, the following code snippet shows how to use the params keyword to create a sum function that can take in a varying number of integer inputs:

ParamsExample.cs

```
1  public class ParamsExample : MonoBehaviour {
2
3      public void Start() {
4          Debug.Log(_SumIntegers(1));        // 1
5          Debug.Log(_SumIntegers(1, 2));     // 3
6          Debug.Log(_SumIntegers(1, 2, 3));  // 6
7      }
8
9      private int _SumIntegers(params int[] inputs) {
10         int sum = 0;
```

```
11        foreach (int input in inputs) sum += input;
12        return sum;
13    }
14
15 }
```

The **object type** is the base type for any C# value, and it's an alias to the System.Object .NET type. So, to put it simply: any type, be it a pre-defined built-in type or a user-defined custom struct or class, inherits directly or indirectly from This in turn means that any variable can be converted to and from the object type into its "real" type, and that the object type can act as a global agnostic type to store values of different types in the same container.

BOXING/UNBOXING

> By the way, the process of converting a value from its real type to object is called **boxing**, and the reverse process of converting from object to the real type is called **unboxing**.

In our case, to quickly allow the users to pass in any number of arguments of any type to a call to we can simply modify the function as follows:

StateMachine.cs

```
1  public void ChangeState(BaseState newState, params object[] transitionArgs) {
2      if (_currentState != null) _currentState.Exit();
3      _currentState = newState;
4      if (_currentState != null) _currentState.Enter(transitionArgs);
5  }
```

Of course, we need to propagate this update to the Enter() method of the BaseState class:

BaseState.cs

```
1  namespace FSM {
2      public abstract class BaseState {
3          public string name;
4          protected StateMachine _fsm;
5
6          public BaseState(string name, StateMachine fsm) { … }
7
```

```
8         public virtual void Enter(params object[] transitionArgs) { }
9         public virtual void Update() { }
10        public virtual void LateUpdate() { }
11        public virtual void Exit() { }
12    }
13 }
```

Even though the base implementation of the Enter() virtual method is empty and doesn't do anything with these extra input parameters, it's a nice-to-have for the real classes because it will help us to share data between our states without having to maintain strong references to a global data source.

And with these two base FSM tools ready, we are now all set up to dive into coding our own AI logic thanks to finite state machines...

Designing a guard AI with finite state machines

Ok – with this FSM package at our disposal, it is time to apply the theory we've discussed in *Chapter 3* and implement our very own finite state machine-based AI! In the rest of this chapter, we'll thus use our BaseState and StateMachine abstract

classes to model the behaviour of a couple of units in a 3D scene.

A quick peek at the AI's behaviour

As usual, the very first step when taking on a new AI modelling challenge is to make sure you've properly understood the entity you're implementing the behaviour of. So, let's mark a little pause and discuss what our AIs should do exactly.

Ultimately, our objective in the rest of this chapter will be to give some brains to two types of units in a scene:

a little Egyptian guard who's doing a routine patrol, and who's attacked by Roman invaders on the way,

and the aforementioned Roman invaders, who will be quite static to begin with but will react if the Egyptian guard gets close.

These two entity behaviours will actually be fairly similar, and they will be a good opportunity to see how, sometimes, the components of a finite state machine may be abstracted enough to be re-used from one FSM to another. Thus even if we're technically going to code up two AIs, most of what we'll

learn for the first one (the Egyptian guard) will be directly transferrable to the second one (the Roman invader).

Figure 4.1 – Let's bring those units to life thanks to AI!

WANNA CHECK OUT THE ASSETS?

> For this example, I'll use some very cool 3D models for my units from the "Free Ancient Models" pack by *markinhofaci* (see *Figure 4.1*, this pack is available on CG Trader over here: https://www.cgtrader.com/free-3d-models/character/fantasy-character/free-ancient-warriors).

> Of course, don't forget that all the code and assets for this chapter and the rest of this book are available in the Github repository, over here: https://github.com/MinaPecheux/Ebook-Unity-AIProgramming. So feel free to have a look if you want to check how the scene is built, or if you want to check out the FSM AI scripts!

Now, to really grasp all the subtle details of these two AIs, we're going to dive deeper into the states and transitions each FSM should implement.

The Egyptian guard state machine

For our player unit, the guard, the FSM will be composed of three states: Patrolling, Chasing and Attacking. The transition between those states will be triggered when an enemy (i.e. a Roman invader unit) is spotted at a certain distance from the guard.

This FSM can be represented by the following diagram:

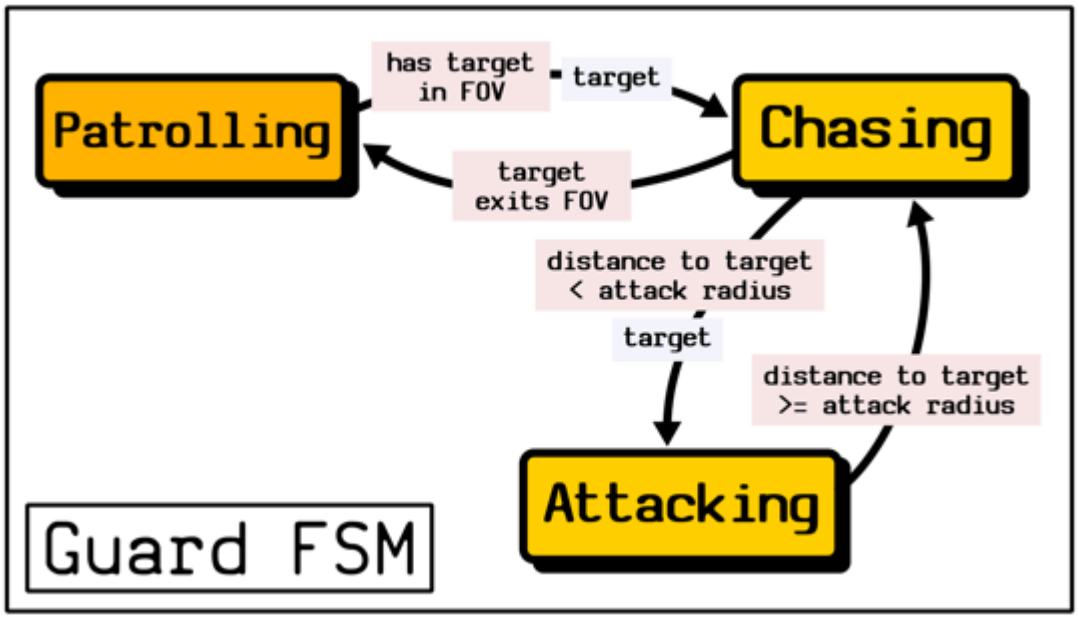


Figure 4.2 – Diagram of the finite state machine for our guard AI

More precisely, the behaviour logic works like this:

The Egyptian guard initially starts in the Patrolling state.

In this Patrolling state, the unit follows a pre-determined path by cycling through a series of waypoints endlessly. At the same time, it keeps an eye out for nearby enemies – if one is spotted inside the field of vision (FOV) of the guard, then the guard stops patrolling and transitions to the Chasing state.

In the Chasing state, the unit simply runs towards its target to get close enough to attack. If the target gets inside the attack

radius of the guard, then the guard stops the chase and transitions to the Attacking state.

In the Attacking state, the unit regularly swings its sword to attack and damage the enemy. We will see at the end of this section how to quickly setup some healthpoints and death mechanics so that the fight eventually ends and the guard can return to patrolling.

The FSM also needs to implement the “reverse” transitions, or in other words: if the guard is in Chasing mode and the target gets further away than its field of vision, the guard has to transition back to the Patrolling state; if the guard is in Attacking mode and the target gets further away than its attack radius, the guard has to transition back to the Chasing state; if the guard is in Attacking mode and the target dies, the guard has to transition back to the Patrolling state.

When the guard transitions back to the Patrolling state, it will go to the last waypoint it recorded as its target position, before resuming its patrol.

In order to implement this state machine, we will need a GuardFSM C# class that inherits from our StateMachine base object, and three classes: Chasing and

The Roman invader state machine

The enemies will have an FSM that resembles a lot the guard's:

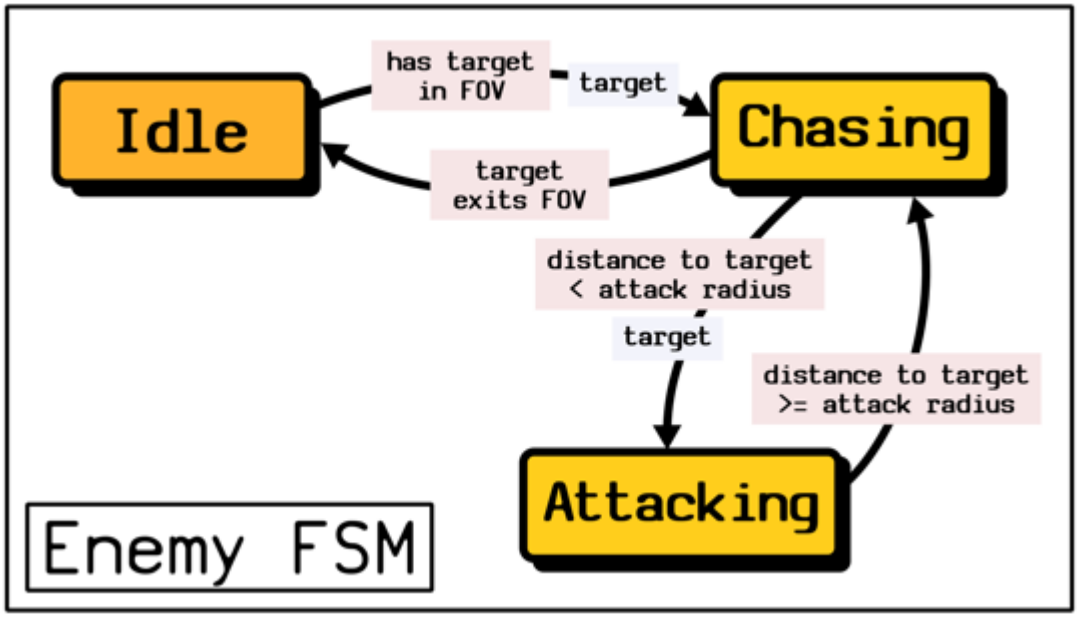


Figure 4.3 – Diagram of the finite state machine for our enemy AI

Basically, the main difference is that, instead of a Patrolling state, the Roman invaders have an Idle state because we want them to stay put until the guard comes in their field of vision.

Once the guard enters this FOV radius, the enemy enters its Chasing state, and then its the same logic as with the guard: if our player gets inside the Roman invader's attack radius,

then the unit will switch to the Attacking state; or else, we have all the same reverse transitions.

Again, this implies we need four classes: an EnemyFSM that derives from StateMachine and three state classes – Chasing and Attacking – that derive from

However, this brief analysis shows something important, which is that parts of our FSMs are shared between the two types of units. Namely, both the guard and the invader will be able to chase and attack another unit, and both will have some base properties like a movement speed or a field of vision.

For example, the Chasing and Attacking script will be identical for both the Egyptian guard and the Roman invader – and it would definitely be a shame to copy-paste the same code just because of some misaligned variable types, right?

That's why, before getting to the implementation, we should think ahead and plan for these shared variables and behaviours!

Shared variables and behaviours

Because the guard and the invader have such similar behaviours, we know that we can probably abstract some of

the data and logic into some generic FSM class, and then use this new intermediary as a base for our actual GuardFSM and EnemyFSM classes.

For example, all these units have the same properties: the move speed, the field of vision, the attack radius and the attack rate. The exact values may vary from one unit type to the other, but they should be available on both so that we can properly execute the shared Chasing and Attacking state logics.

Therefore, let's make a new C# script in our Unity project named UnitFSM that inherits from and that contains these properties as public variables (to make them easily editable in the inspector):

UnitFSM.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public abstract class UnitFSM : StateMachine {
5      public float speed = 2f;
6      public float fieldOfVision = 3f;
7      public float attackRadius = 1f;
8      public float attackRate = 2.4f; // in seconds
```

```
9 }
```

Here, I'm importing the FSM C# package we prepared in the first section, *Creating the FSM building* so as to access our StateMachine type, and then I'm defining a new abstract C# class called UnitFSM that represents this intermediary behaviour shared between the guard and the invader.

Now, we can use this class as a base for two new classes, GuardFSM and

---

GuardFSM.cs

```
1 using System.Collections.Generic;
2 using UnityEngine;
3 using FSM;
4
5 public class GuardFSM : UnitFSM {
6     private void Awake() { /* setup the states Dictionary */ }
7
8     protected override BaseState GetInitialState() => null;
9 }
```

EnemyFSM.cs

```
1  using System.Collections.Generic;
2  using FSM;
3
4  public class EnemyFSM : UnitFSM {
5      private void Awake() { /* setup the states Dictionary */ }
6
7      protected override BaseState GetInitialState() => null;
8  }
```

What's cool is that, because our StateMachine class already does most of the heavy-lifting, we actually don't need to add anything in the UnitFSM – 'cause what we need to do now is define our actual state classes and then declare them in the lower-level GuardFSM and

Preparing the state classes and filling the low-level FSMs

The idea is to fill the _states C# Dictionary we prepared in our StateMachine class with specific instances of classes. Specifically, we want to prepare the four possible state classes

for our two unit types Attacking and and then instantiate them in the Awake() function of either the GuardFSM or the EnemyFSM script to fill the _states Then, we'll just use this Dictionary to define the initial state of our unit's state machine.

For now, let's just create our C# classes with an empty behaviour. We'll make four new C# files, Attacking.cs and with the following content:

Patrolling.cs

```
1  using FSM;
2
3  public class Patrolling : BaseState {
4      public Patrolling(UnitFSM fsm) : base("Patrolling", fsm) {}
5  }
```

Chasing.cs

```
1  using FSM;
```

```
2
3 public class Chasing : BaseState {
4     public Chasing(UnitFSM fsm) : base("Chasing", fsm) {}
5 }
```

Attacking.cs

```
1 using FSM;
2
3 public class Attacking : BaseState {
4     public Attacking(UnitFSM fsm) : base("Attacking", fsm) {}
5 }
```

Idle.cs

```
1 using FSM;
2
3 public class Idle : BaseState {
4     public Idle(UnitFSM fsm) : base("Idle", fsm) {}
5 }
```

At this point, these four classes don't define any logic, so the unit won't have any real behaviour. However, they have introduced all the types we need in our C# codebase to finalise our GuardFSM and EnemyFSM scripts.

All that's left to do in those classes is to prepare our _states Dictionary by instantiating the right class, and then setting the initial state by returning a specific state from the Dictionary in the GetInitialState() overridden method:

GuardFSM.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using FSM;
4
5  public class GuardFSM : UnitFSM {
6      private void Awake() {
7          _states = new Dictionary<string, BaseState>() {
8              { "patrolling", new Patrolling(this) },
9              { "chasing", new Chasing(this) },
10             { "attacking", new Attacking(this) },
```

```
        };
    }

    protected override BaseState GetInitialState() => _states["patrolling"];
}
```

EnemyFSM.cs

```
using System.Collections.Generic;
using FSM;

public class EnemyFSM : UnitFSM {
    private void Awake() {
        _states = new Dictionary<string, BaseState>() {
            { "idle", new Idle(this) },
            { "chasing", new Chasing(this) },
            { "attacking", new Attacking(this) },
        };
    }

    protected override BaseState GetInitialState() => _states["idle"];
}
```

We've now prepared everything we need to assemble our various states together – it's now time to dive into the meat of this chapter and gradually implement each state (along with the transitions) to really give our units some behaviour!

Coding up the guard patrolling logic

First things first, let's take care of our guard's default logic: walking a patrol route defined by a couple of waypoints. To keep things simple, we will leave the unit alone in the scene for the moment, and we'll focus solely on implementing this endless cycle.

The point will be to get the guard to follow a path like this one (where the waypoints are the green spheres, and we see the unit travels in a straight-line between each) and stop at each waypoint for one second:

Figure 4.4 – A screenshot of our guard walking along a waypoints-based patrol path

DEBUGGING OUR SYSTEMS

> Throughout this chapter, I'll show various screenshots with debug gizmos and extra labels or visualisers. I won't talk about these debugging techniques here 'cause they're a bit out of scope, but as usual don't hesitate to work on it on your own! And remember that you can always check out the Github repository of the book (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) to see how I've done it here :)

The implementation of the logic for this state can be decomposed in three steps:

1. Getting and cycling through the waypoints.

2. Adding the wait logic to have the guard stay at each waypoint for one second.

3. Referencing the unit's Animator component to switch between the "Idle" animation (during the waits) or the "Run" animation (during the moves).

SETTING UP A UNITy ANIMATOR

> Again, I won't get into the details of how to setup a Unity Animator component here because it's not the topic at hand; but if you're curious, you can check out this other tutorial I made on the topic: "Configure & animate Mixamo characters" (https://youtu.be/8Pk7FI629O8), or the book's Github.

Let's go through each step one by one!

Listing and cycling through the waypoints

The core thing currently missing from our Patrolling script is a list of waypoints to actually patrol through.

In this example, we're going to place empty objects in our scene to define the waypoints, and then reference their Transforms to tell our guard what positions to walk to. Our scene therefore looks like this:

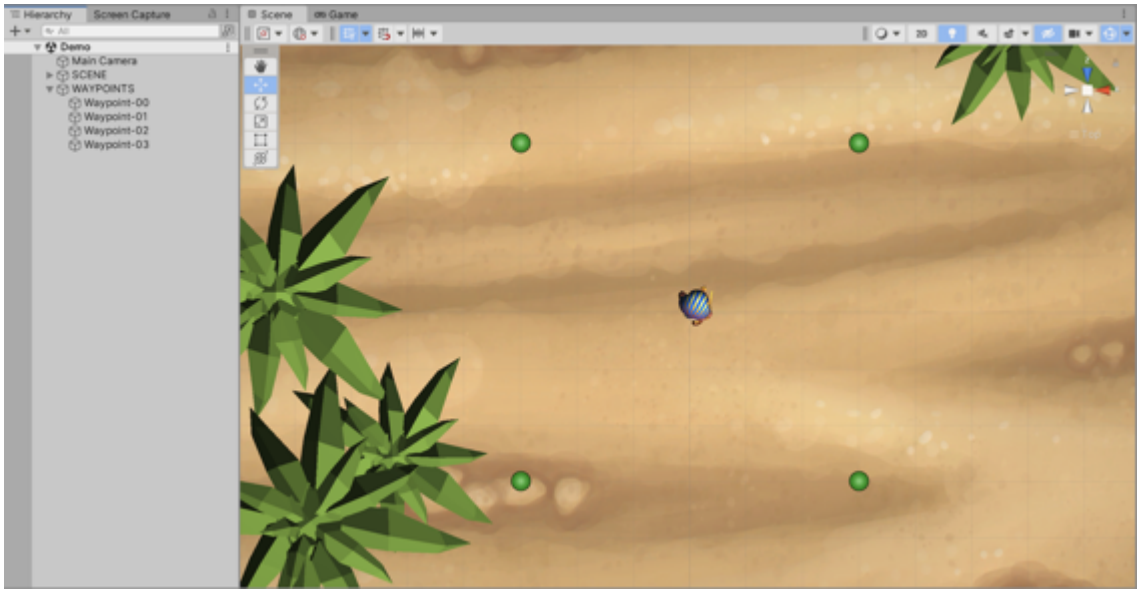


Figure 4.5 – A top-down view of the scene with the guard model in the middle and four waypoints around him (defined using empty game objects)

Figure 4.5 shows the list of four waypoint game objects in the **Hierarchy** panel and the matching gizmos in the **Scene** view.

Because we don't have access to our actual Patrolling object from the Unity we're going to expose a list of Transform references in our and then transfer this info to the Patrolling instance when we create it.

So let's update our GuardFSM script with a new public variable called which is a list of

---

GuardFSM.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using FSM;
4
5  public class GuardFSM : UnitFSM {
6      public Transform[] waypoints;
7
8      private void Awake() { ... }
9      protected override BaseState GetInitialState() => _states["patrolling"];
10 }
```

And then, in the **inspector** panel, we'll drag our four game objects into these slots to get our references. Note that the

order you drag them in will determine the order of the waypoints in the patrol cycle for our guard.

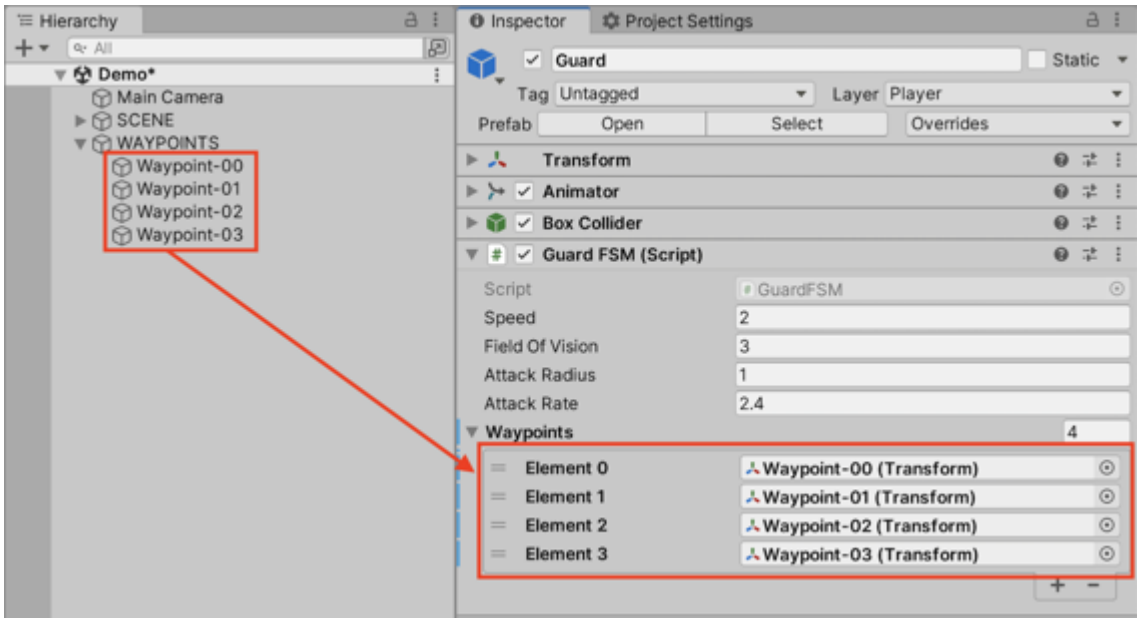


Figure 4.6 – Setup of the waypoint references in the GuardFSM component

Finally, we'll go back to our Patrolling class and update its constructor so that it also expects this list of waypoints; and we'll pass it from the GuardFSM in the Awake() function:

Patrolling.cs

```
1 using UnityEngine;
2 using FSM;
```

```
3
4  public class Patrolling : BaseState {
5      private Transform[] _waypoints;
6
7      public Patrolling(UnitFSM fsm, Transform[] waypoints)
8          : base("Patrolling", fsm) {
9          _waypoints = waypoints;
10     }
11 }
```

GuardFSM.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using FSM;
4
5  public class GuardFSM : UnitFSM {
6      private Transform[] _waypoints;
7
8      private void Awake() {
9          _states = new Dictionary<string, BaseState>() {
10             { "patrolling", new Patrolling(this, waypoints) },
11             { "chasing", new Chasing(this) },
```

```
12            { "attacking", new Attacking(this) },
13        };
14    }
15
16    protected override BaseState GetInitialState() => _states["patrolling"];
17 }
```

Now that we have this list of waypoints, we'll also need to retrieve the Transform and move speed of our unit. These can be extracted directly from our reference to the parent StateMachine instance, passed as the first parameter to our constructor:

Patrolling.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public class Patrolling : BaseState {
5      private Transform _transform;
6      private Transform[] _waypoints;
7      private float _speed;
8
```

```
 9     public Patrolling(UnitFSM fsm, Transform[] waypoints)
10         : base("Patrolling", fsm) {
11         _transform = fsm.transform;
12         _waypoints = waypoints;
13         _speed = fsm.speed;
14     }
15 }
```

Then, we can define the index of our current waypoint, place our unit on the first waypoint at the beginning, and finally use the Update() entry point of our state class to continuously aim for and walk to the current waypoint, or cycle through the list and set the next one as current if we've reached our target:

Patrolling.cs

```
 1 using UnityEngine;
 2 using FSM;
 3
 4 public class Patrolling : BaseState {
 5     private Transform _transform;
 6     private Transform[] _waypoints;
 7     private int _waypointIndex;
```

```
 8    private float _speed;
 9
10    public Patrolling(UnitFSM fsm, Transform[] waypoints)
11        : base("Patrolling", fsm) {
12        _transform = fsm.transform;
13        _waypoints = waypoints;
14        _waypointIndex = 0;
15        _speed = fsm.speed;
16
17        _transform.position = waypoints[0].position;
18    }
19
20    public override void Update() {
21        base.Update();
22
23        Transform wp = _waypoints[_waypointIndex];
24        if (Vector3.Distance(_transform.position, wp.position) < 0.01f) {
25            _transform.position = wp.position;
26            _waypointIndex = (_waypointIndex + 1) % _waypoints.Length;
27        } else {
28            _transform.position = Vector3.MoveTowards(
29                _transform.position, wp.position, _speed * Time.deltaTime);
30            _transform.LookAt(wp.position);
31        }
32    }
33 }
```

To easily loop at the end of the list and designate the first waypoint as the new target when we've reached the last one, we can use a modulo (see *line 22* in the snippet above); and to get a smooth translation from the guard's current position to the position of the target waypoint, we can use the Vector3.MoveTowards() method (see *line 28* in the snippet above).

At this point, if we run our game, we'll see that the guard indeed slides from one waypoint to the other in an infinite cycle!

Figure 4.7 – A screenshot of our guard sliding on its patrol path

Now, we need to implement the wait logic so that the unit also stops for a second when it reaches a new waypoint position.

Adding the wait logic

Having our unit wait at each waypoint is not that hard to code. The main trick is to use a local counter to know how long the unit has been waiting so far and then, whenever this counter reaches the wait time threshold, we know that we've waited the right amount of time.

So, when the unit arrives at its target waypoint, we'll enter the "wait mode" and reset our time counter; then, while in "wait mode", we'll increase this time counter each frame and check to see if we've reached the wait time threshold; if we have, then we'll exit the "wait mode" and resume the movement:

Patrolling.cs

```
1 using FSM;
2 using UnityEngine;
```

```
3
4  public class Patrolling : BaseState {
5      // ...
6      private float _waitTime = 1f; // in seconds
7      private float _waitCounter = 0f;
8      private bool _waiting = false;
9
10     public Patrolling(UnitFSM fsm, Transform[] waypoints)
11         : base("Patrolling", fsm) { ... }
12
13     public override void Update() {
14         base.Update();
15
16         if (_waiting) {
17             _waitCounter += Time.deltaTime;
18             if (_waitCounter < _waitTime) return;
19             _waiting = false;
20         }
21
22         Transform wp = _waypoints[_waypointIndex];
23         if (Vector3.Distance(_transform.position, wp.position) < 0.01f) {
24             _transform.position = wp.position;
25             _waypointIndex = (_waypointIndex + 1) % _waypoints.Length;
26             _waitCounter = 0f;
27             _waiting = true;
28         } else {
29             _transform.position = Vector3.MoveTowards(
```

```
30                 _transform.position, wp.position, _speed * Time.deltaTime);
31             _transform.LookAt(wp.position);
32         }
33     }
34 }
```

And that's it! If we restart the game, we see that our guard player now stops at each waypoint for a second before resuming its patrol.

Updating the unit's animations

The only issue right now is that our unit always plays the same "Idle" animation and slides from one location to the other in quite an unnatural way! To fix this, we're going to use the Animator component on our guard to switch to the "Run" animation when it's moving.

This per-state animation switching logic will also be useful for our Roman invaders, since they'll use a different animation for their Idle, Chasing and Attacking states – so we can add this reference to the unit's Animator component in our abstract UnitFSM class to easily share it between all units in the scene. We'll also assume that the Animator component is actually on the same object as the FSM script, and we'll use Unity's built-in GetComponent() function to retrieve it when the game starts:

UnitFSM.cs

```
1 using UnityEngine;
2 using FSM;
```

```
3
4  public abstract class UnitFSM : StateMachine {
5      public float speed = 2f;
6      public float fieldOfVision = 3f;
7      public float attackRadius = 1f;
8      public float attackRate = 2.4f; // in seconds
9      [HideInInspector] public Animator animator;
10
11      protected virtual void Awake() {
12          animator = GetComponent<Animator>();
13      }
14  }
```

There are a few things to note in this snippet:

Our animator variable should be public because we'll want to access it easily from our other scripts. However, it doesn't need to show in the inspector since we'll auto-assign it with so we can give it the [HideInInspector] attribute.

We're using the Awake() lifecycle hook to assign this animator reference. But since we already had some logic in our Awake() functions down in the GuardFSM and EnemyFSM scripts, we have to make this function virtual... and in the GuardFSM and EnemyFSM classes we need to turn our Awake() functions into

protected overrides and call the Awake() on the base class to keep the entire logic:

GuardFSM.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using FSM;
4
5  public class GuardFSM : UnitFSM {
6      private Transform[] _waypoints;
7
8      protected override void Awake() {
9          base.Awake();
10         _states = new Dictionary<string, BaseState>() { ... };
11     }
12     protected override BaseState GetInitialState() => _states["patrolling"];
13 }
```

EnemyFSM.cs

```
1  using System.Collections.Generic;
2  using FSM;
3
4  public class EnemyFSM : UnitFSM {
5      protected override void Awake() {
6          base.Awake();
7          _states = new Dictionary<string, BaseState>() { ... };
8      }
9
10     protected override BaseState GetInitialState() => _states["idle"];
11 }
```

And now, back in our Patrolling class, we can access this Animator component and set its Running boolean parameter to switch between the “Idle” and the “Run” animations:

Patrolling.cs

```
1  using FSM;
2  using UnityEngine;
3
4  public class Patrolling : BaseState {
5      // ...
```

```
6     private Animator _animator;
7
8     public Patrolling(UnitFSM fsm, Transform[] waypoints)
9         : base("Patrolling", fsm) {
10        // ...
11        _animator = fsm.animator;
12    }
13
14    public override void Update() {
15        base.Update();
16
17        if (_waiting) {
18            _waitCounter += Time.deltaTime;
19            if (_waitCounter < _waitTime) return;
20            _waiting = false;
21            _animator.SetBool("Running", true);
22        }
23        Transform wp = _waypoints[_waypointIndex];
24        if (Vector3.Distance(_transform.position, wp.position) < 0.01f) {
25            _transform.position = wp.position;
26            _waypointIndex = (_waypointIndex + 1) % _waypoints.Length;
27            _waitCounter = 0f;
28            _waiting = true;
29            _animator.SetBool("Running", false);
30        } else { ... }
31    }
32 }
```

As a little bonus, we can also ensure that the entity starts with the right animation when it enters this state, by overriding the Enter() entry point of our Patrolling class and initialising this Running animator parameter depending on the current distance to a waypoint:

Patrolling.cs

```
1  using FSM;
2  using UnityEngine;
3
4  public class Patrolling : BaseState {
5    // ...
6
7    public Patrolling(UnitFSM fsm, Transform[] waypoints)
8        : base("Patrolling", fsm) {
9        ...
10   }
11   public override void Enter(params object[] transitionArgs) {
12       Transform wp = _waypoints[_waypointIndex];
13       float d = Vector3.Distance(_transform.position, wp.position);
14       bool onWaypoint = d < 0.01f;
```

```
15          _animator.SetBool("Running", !onWaypoint);
16      }
17      public override void Update() { ... }
18  }
```

(Of course, here, we don't need to use any transition parameters so we can just ignore the transitionArgs input variable.)

Our Patrolling logic is now mostly complete – our guard properly cycles through the waypoints, it stops for a second each time it reaches a new location and it even switches between its different model animations to get a natural visual with an "Idle" and a "Run" animation.

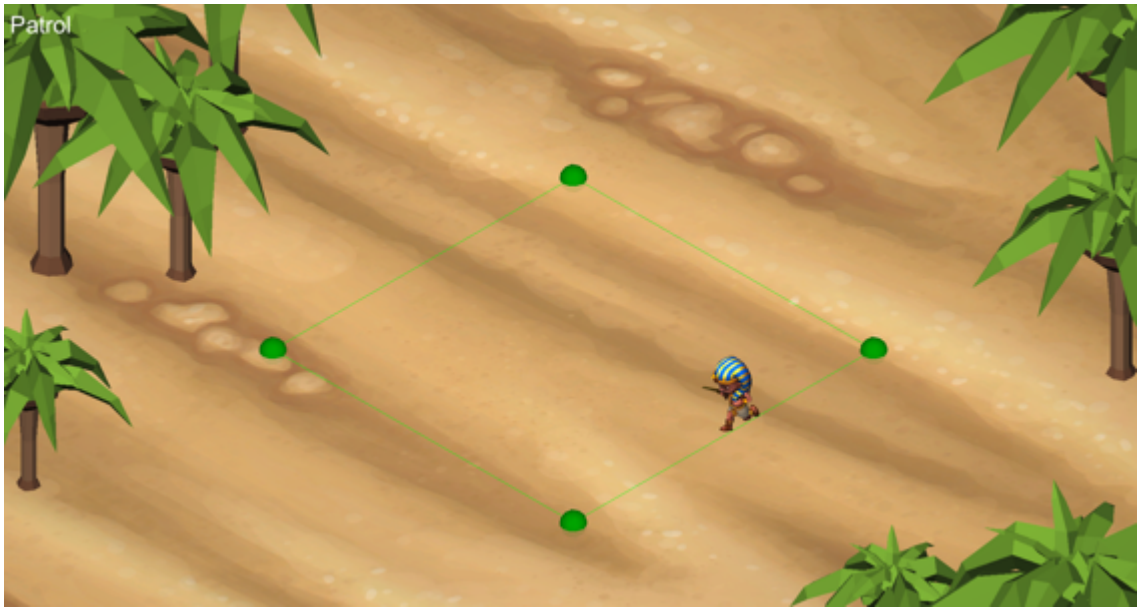


Figure 4.8 – A demo screenshot of the final Patrolling logic of our guard AI

This is really nice: we have successfully coded the first state script of our FSM and we've given our unit a base logic to have it patrol around on its own – and we could even dynamically move the waypoints and the path would adapt because we've used the Transform as references!

But now, we need to integrate this state with the other ones and start to implement some transitions to expand our unit's behaviour...

Adding the enemies, checking for nearby targets

The next step in the implementation of our guard's AI is to add enemies in our scene, and have the unit check for those invaders during its patrol. If any enemy is in its field of vision, the guard will transition to its Chasing state and take care of the problem.

From that point on, we'll consider that our scene contains our guard and its our waypoints, plus three Roman soldiers that are spread around the patrol path and ready to fight:

Figure 4.9 – A screenshot of our scene with some threatening Roman invaders around the guard unit!

Each of these Roman invaders in an instance of the "Enemy" prefab which has attached on it, among other things, the EnemyFSM script and a BoxCollider component:

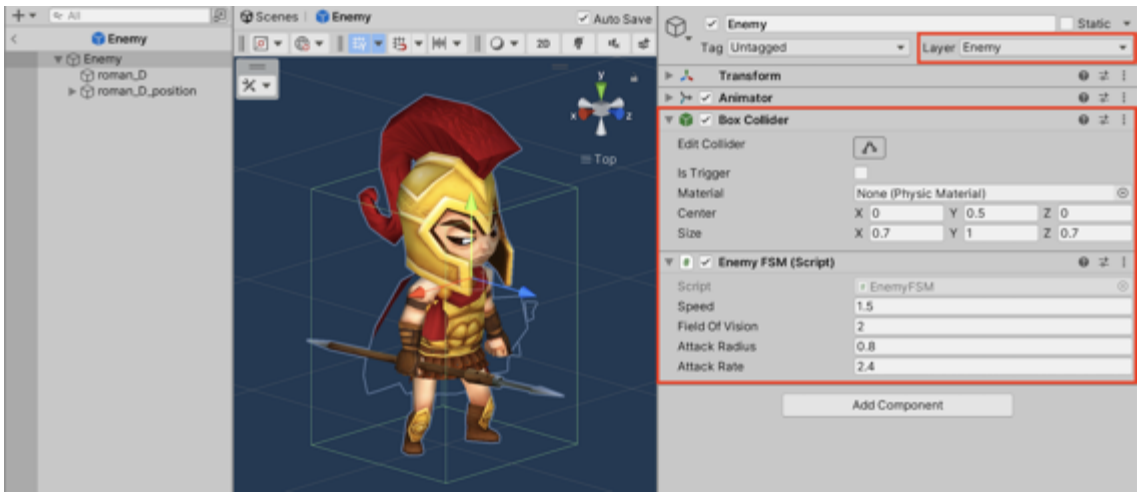


Figure 4.10 – The "Enemy" prefab with a BoxCollider component and the EnemyFSM script attached on it, and some predefined values for the AI logic

Moreover, I've made sure to set this prefab to be on a user-defined "Enemy" Unity Physics layer (see in the top-right corner of *Figure* – which will make it easy to optimise and filter my search for enemy targets in the Patrolling logic of my guard AI.

So, now, we want to update our Patrolling class to have it regularly look for possible targets in the FOV radius of the unit and, if there is indeed an enemy close enough, set this invader as the target and switch over to the Chasing state.

To implement this, we'll do the following:

1. To begin with, we'll once again use our reference to the parent state machine instance to get the field of vision of our guard unit; this is easy to do, we just need to store the value in the constructor of our state object:

Patrolling.cs

```
1 using FSM;
```

```
2  using UnityEngine;
3
4  public class Patrolling : BaseState {
5      // ...
6      private float _fov;
7
8      public Patrolling(UnitFSM fsm, Transform[] waypoints)
9          : base("Patrolling", fsm) {
10             // ...
11             _fov = fsm.fieldOfVision;
12     }
13
14     public override void Enter(params object[] transitionArgs) { ... }
15     public override void Update() { ... }
16 }
```

---

2. Then, we'll override our LateUpdate() entry point to run our search logic. The idea behind this check is to use Unity's built-in Physics.OverlapSphere() method to create a "search bubble" around our unit, and list all the colliders that are inside this bubble.

To only consider the enemies in the scene and optimise the search, we'll also need to pass in the mask of the Physics

layer to use – in our case, "Enemy", which is layer n°7 in my project:

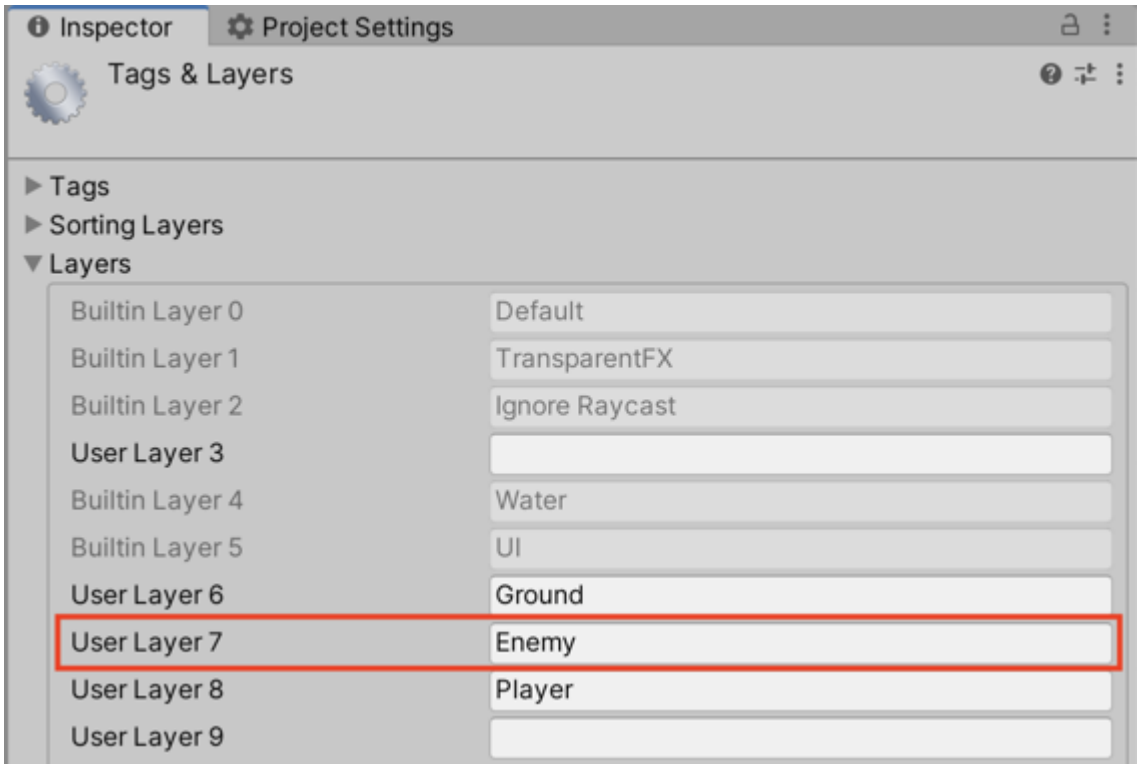


Figure 4.11 – List of Physics layers in the project, with the "Enemy" layer at index 7

In Unity, layer masks are defined as integers; they are **bitmasks** where you have 0 digits for all the layers you're ignoring, and 1 digits for all the layers you're taking into account (with the digits being 0-indexed). Typically, here, my "Enemy" layer has index 7, so I want my "enemy layer mask" to have its eight digit equal to one, and the rest equal to zero (see *Figure*

If you're somewhat used to binary numbers, you know that this is and it is equivalent to the number 128 in decimal representation. In other words, the Unity bitmask that corresponds to the layer is

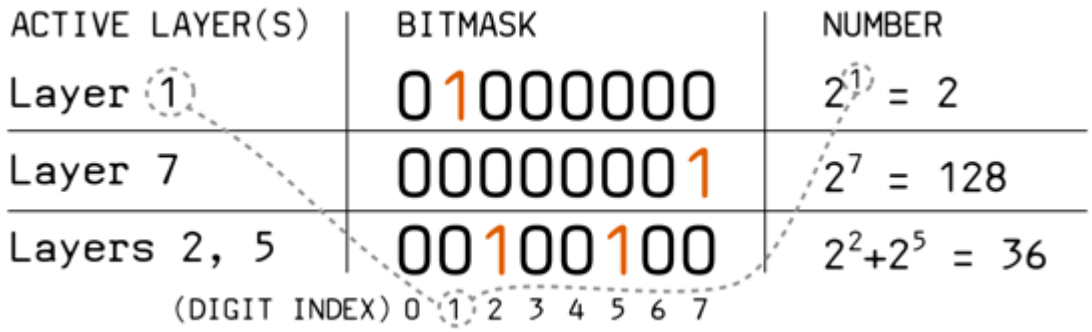


Figure 4.12 – Overview of the layer bitmask system: depending on the layers you choose to enable, you activate or deactivate specific digits in the binary representation of the layer mask, which ultimately corresponds to a specific integer value

However, it might not be very intuitive when we read the code that something like 128 means "activate the layer n°7 in my search filter". So a better way (in my opinion) to declare Unity layer masks is to use **bitshifting** and simply pass in the index of the layer to consider, like this:

Patrolling.cs

```
1  using FSM;
2  using UnityEngine;
3
4  public class Patrolling : BaseState {
5      // ...
6      private const int _enemyLayerMask = 1 << 7;
7
8      public Patrolling(UnitFSM fsm, Transform[] waypoints)
9          : base("Patrolling", fsm) { ... }
10
11     public override void Enter(params object[] transitionArgs) { ... }
12     public override void Update() { ... }
13 }
```

(Note that we can make this _enemyLayerMask variable constant since it's just a specific integer value.)

And now, we just need to call the Physics.OverlapSphere() function with our unit's position as the centre, its FOV as the radius and the _enemyLayerMask as the filter, and we'll get a list of all the colliders currently inside this search bubble:

Patrolling.cs

```
using FSM;
using UnityEngine;

public class Patrolling : BaseState {
    // ...

    public Patrolling(UnitFSM fsm, Transform[] waypoints)
        : base("Patrolling", fsm) { ... }

    public override void Enter(params object[] transitionArgs) { ... }
    public override void Update() { ... }

    public override void LateUpdate() {
        base.LateUpdate();
        Collider[] targets = Physics.OverlapSphere(
            _transform.position, _fov, _enemyLayerMask);
    }
}
```

3. Finally, we just need to check if there is at least one target in this list and, in that case, transition to the Chasing state. To do this, remember that we prepared a ChangeState() method in our StateMachine base class, which can take in a

reference to the new state instance, plus some optional parameters.

Here, we'll arbitrarily say that the first target in our list is cherry-picked as the new enemy target for our guard unit, and we'll pass its Transform as an input parameter to our ChangeState() function:

Patrolling.cs

```
1  using FSM;
2  using UnityEngine;
3
4  public class Patrolling : BaseState {
5    // ...
6
7    public Patrolling(UnitFSM fsm, Transform[] waypoints)
8        : base("Patrolling", fsm) { ... }
9
10   public override void Enter(params object[] transitionArgs) { ... }
11   public override void Update() { ... }
12
13   public override void LateUpdate() {
14       base.LateUpdate();
15
```

```
16        Collider[] targets = Physics.OverlapSphere(
17            _transform.position, _fov, _enemyLayerMask);
18        if (targets.Length > 0)
19            _fsm.ChangeState(_fsm.GetState("chasing"), targets[0].transform);
20    }
21 }
```

Thanks to the auto-boxing of our variables to the object “lazy” type, we can directly pass the Transform variable to the function, and it will get conveyed throughout the different calls down to the Enter() entry point of our new state, which here is the Chasing state.

With these few modifications, we’ve thus added the first transition in our FSM, which takes us from the Patrolling state to the Chasing state. Now, it’s time to actually code the right Chasing logic inside the Chasing class we prepared earlier...

Chasing the target

The good news is that the Patrolling script we just finished was actually the most complex one and it introduced us to almost all the code details we’re going to need for our FSM; so the rest of the journey should be a more relaxing ride.

Typically, the Chasing state is very straight-forward – it's basically a mix-and-match of things we've seen before:

In the constructor, we'll use our reference to the parent state machine to get the unit's its Animator component and its move speed and attack radius properties.

Then, in the Enter() entry point override, we'll use our transitionArgs input parameter to retrieve and store the reference to the target we found in the Patrolling state. We'll also switch to the "Run" animation.

And in the Update() entry point, we'll move towards this target until we're close enough to attack – then, when the distance to the target gets below our guard's attack radius property, we'll switch to the Attacking state and pass on the target reference as we did previously.

This whole logic can be written up as follows:

Chasing.cs

```
1 using UnityEngine;
2 using FSM;
3
```

```
4  public class Chasing : BaseState {
5      private Animator _animator;
6      private Transform _transform;
7      private float _speed;
8      private float _attackRadius;
9      private Transform _target;
10
11     public Chasing(UnitFSM fsm) : base("Chasing", fsm) {
12         _animator = fsm.animator;
13         _transform = fsm.transform;
14         _speed = fsm.speed;
15         _attackRadius = fsm.attackRadius;
16     }
17
18     public override void Enter(params object[] transitionArgs) {
19         if (transitionArgs == null) return;
20         _target = (Transform)transitionArgs[0];
21         _animator.SetBool("Running", true);
22     }
23
24     public override void Update() {
25         base.Update();
26         float d = Vector3.Distance(_transform.position, _target.position);
27         if (d < _attackRadius) {
28             _fsm.ChangeState(_fsm.GetState("attacking"), _target);
29         } else {
30             _transform.position = Vector3.MoveTowards(
```

```
31                _transform.position, _target.position,
32                _speed * Time.deltaTime);
33            _transform.LookAt(_target.position);
34        }
35    }
36 }
```

The only new noticeable thing is the unboxing step in the Enter() override (see *line 20* of the snippet above), which is basically about making an explicit conversion of our object variable into its real type,

To properly handle the reverse transitions, we should also add a bit of logic at the end of the Update() that checks if the distance to the target gets over the field of vision radius – in which case we need to go back to the Patrolling state. To avoid the unit "flickering" between the two states, we're not going to use the exact FOV radius but a slightly higher value:

Chasing.cs

```
1 using UnityEngine;
2 using FSM;
```

```
3
4  public class Chasing : BaseState {
5      // ...
6      private float _fov;
7
8      public Chasing(UnitFSM fsm) : base("Chasing", fsm) {
9          // ...
10         _fov = fsm.fieldOfVision;
11     }
12
13     public override void Enter(params object[] transitionArgs) { ... }
14
15     public override void Update() {
16         base.Update();
17         float d = Vector3.Distance(_transform.position, _target.position);
18         if (d < _attackRadius) {
19             _fsm.ChangeState(_fsm.GetState("attacking"), _target);
20         } else if (d > _fov + 0.5f) {
21             _fsm.ChangeState(_fsm.GetState("patrolling"));
22         } else {
23             _transform.position = Vector3.MoveTowards(
24                 _transform.position, _target.position,
25                 _speed * Time.deltaTime);
26             _transform.LookAt(_target.position);
27         }
28     }
29 }
```

If we re-run the game at this point, we see that as long as the enemy unit is out of the field of vision radius, our guard stays on patrol but then, as soon as the Roman invader gets in range, our new Chasing state logic kicks in and makes the guard run towards the enemy unit:

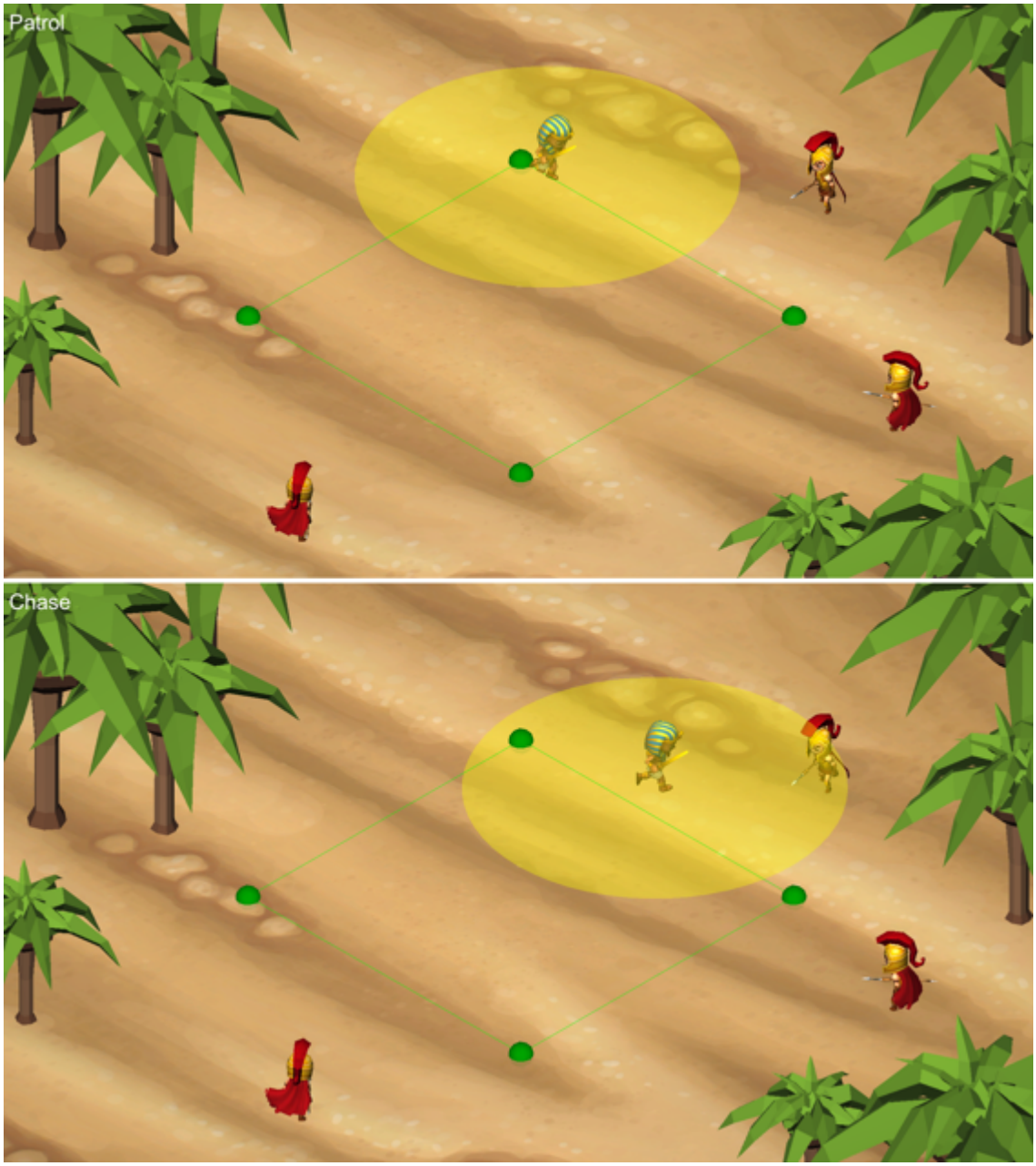


Figure 4.13 – Screenshots of our scene with the guard in the Patrolling or in the Chasing state, depending on whether there is an enemy in its field of vision or not

You can also try to move the enemy by hand in the scene as the guard is chasing it – if the enemy gets out of the guard's field of vision, the guard should return back to its patrol logic!

To finalise this guard FSM, the final state we need to take care of now is the Attacking state.

Setting up the attack logic

Again, compared to the Patrolling state, this Attacking logic is fairly simple and quick to code. Here, we're going to re-use the time counter technique we saw above for the waypoint wait logic except, this time, it will be a way to regularly throw a new sword swing at the enemy.

All in all, the class looks like this:

Attacking.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public class Attacking : BaseState {
5      private Animator _animator;
```

```
 6    private Transform _transform;
 7    private float _attackRadius;
 8    private float _attackRate;
 9    private float _attackCounter = 0f;
10     private Transform _target;
11
12     public Attacking(UnitFSM fsm) : base("Attacking", fsm) {
13         _animator = fsm.animator;
14         _transform = fsm.transform;
15         _attackRadius = fsm.attackRadius;
16         _attackRate = fsm.attackRate;
17     }
18
19     public override void Enter(params object[] transitionArgs) {
20         if (transitionArgs == null) return;
21         _target = (Transform)transitionArgs[0];
22         _animator.SetBool("Running", false);
23         _attackCounter = _attackRate; // hit immediately
24     }
25
26     public override void Update() {
27         base.Update();
28
29         _attackCounter += Time.deltaTime;
30         if (_attackCounter >= _attackRate) {
31             _animator.SetTrigger("Attack");
32             _attackCounter = 0f;
```

```
33          }
34
35          float d = Vector3.Distance(_transform.position, _target.position);
36          if (d > _attackRadius) {
37              _fsm.ChangeState(_fsm.GetState("chasing"), _target);
38          }
39      }
40  }
```

There's nothing very surprising here: we're getting back the reference to our target in the Enter() override, we're initialising our time counter and then, in the Update() entry point, we loop our attack counter again and again, unless the target gets too far away and we need to switch back to the Chasing state.

(By the way, because the "Attack" animation is a one-shot and not a looping animation like the "Idle" or the "Run", we're not using a boolean animator property anymore but rather a trigger property, which is why we have to call the _animator.SetTrigger() method.)

To ensure that we don't try to run a remaining "Attack" animation when we exit this state, we can override the Exit() entry point in our Attacking class and use the _animator.ResetTrigger() method to cancel this animation:

Attacking.cs

```
1  using UnityEngine;
2  using FSM;
```

```
3
4  public class Attacking : BaseState {
5      // ...
6
7      public Attacking(UnitFSM fsm) : base("Attacking", fsm) { ... }
8
9      public override void Enter(params object[] transitionArgs) { ... }
10
11     public override void Exit() {
12         base.Exit();
13         _animator.ResetTrigger("Attack");
14     }
15
16     public override void Update() { ... }
17 }
```

If we debug both the guard's FOV and attack radius, we can now test that the unit stays in its Chasing state for as long as the enemy is further than the attack radius, and then when the Roman gets closer our player guard enters its Attacking state and starts to swing its sword:

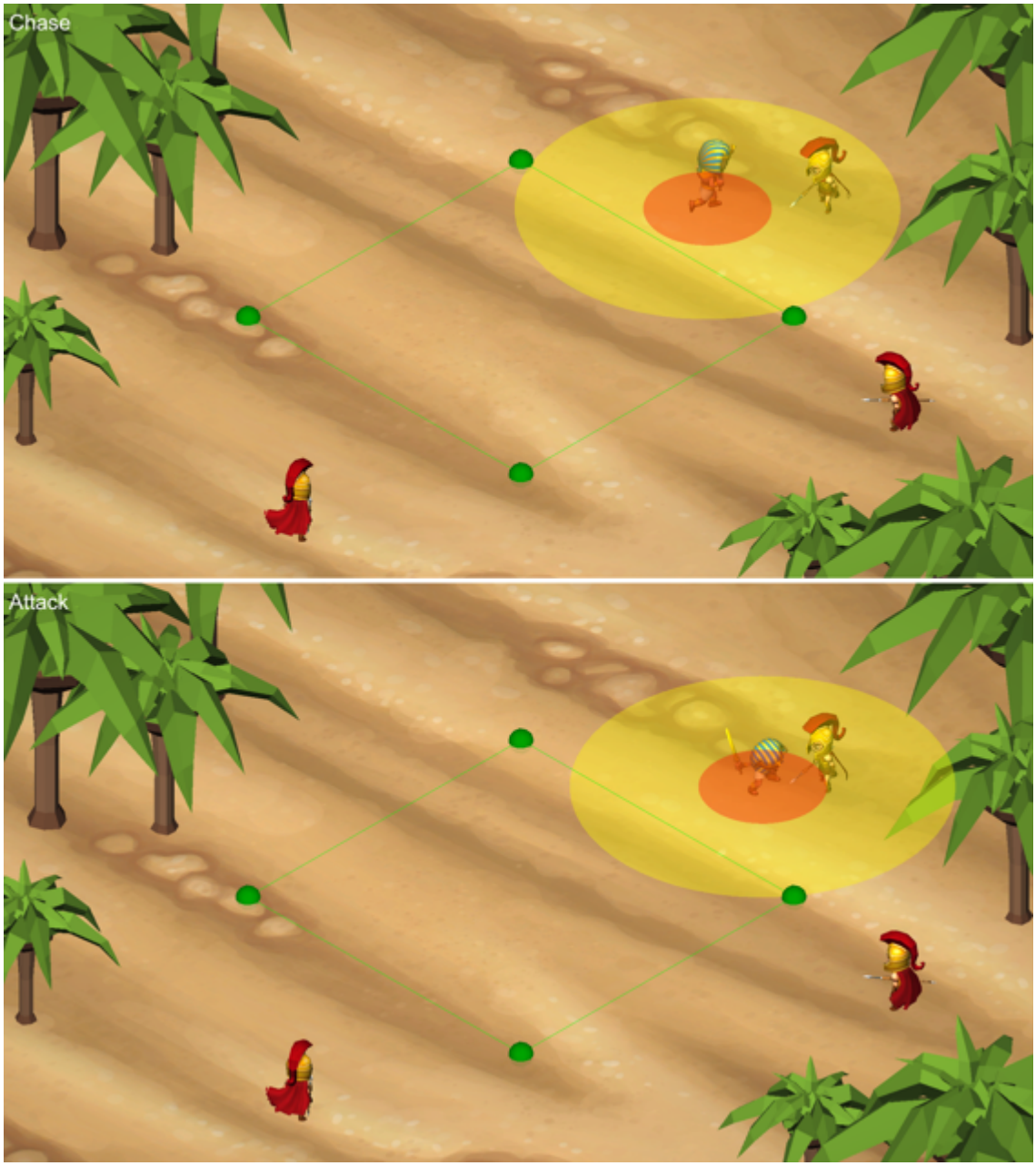


Figure 4.14 – Screenshots of our scene with the guard in the Chasing or in the Attacking state, depending on whether there is an enemy in its attack radius or not

In this section, we've thus developed our guard's AI step-by-step, and we now have a good idea of how to implement a finite state machine. We've seen that this structure, although it is quite simple to understand, can already create pretty interesting behaviours with various contexts, an adaptive logic that takes into account the environment around the entity, and even some fallbacks if we properly code up our reverse transitions.

To wrap up this chapter and really finish our little scene, let's now talk a bit about the Roman invader's AI logic.

Developing the enemy's AI

In this final part, we're going to see how much of our current FSM can be directly re-used for the enemies, and what needs to be tweaked to better fit this scenario. We'll then prepare some healthpoints and death triggers so that the fights actually end, and so that our guard can eliminate the threats...

Modularity, re-usability & decoupling

As we've briefly mentioned in the previous chapter, a nice property of finite state machines is that, sometimes, your states can be abstract enough to be re-used in different contexts.

For example, here, our Chasing and Attacking logic run a logic that is not specific to our player guard, or our enemy. This means that we can directly re-integrate instances of these classes in either FSMs without having to adapt the code in some twisted way.

This idea of creating re-usable logic blocks is called and it's a crucial thing in software development, because it makes your code centralised, less redundant and overall less of a pain to write. Among other things, this design principle avoids inconsistencies, since you always call the same piece of code to do the same thing – you don't run the risk of having two versions of your Chasing logic, with one that is old and obsolete.

Modularity is not always possible, and it often requires a high level of **decoupling** in code – meaning having code chunks that are as self-contained as possible, with no strong dependencies to other elements. If your code stands on its on two feet and doesn't require any other component or script to function properly, then it is *decoupled* from the rest.

Again, decoupling is an ideal for programmers, but it's hard to have in practice.

And finite state machines, although they may allow for a bit of modularity, usually don't really show a lot of decoupling.

If you think about it, our various state scripts all require a direct reference to their parent state machine to work. And, in particular, you need this FSM instance to have a list of all the possible states readily accessible in order to code your state transitions. Even some of the data can be coupled between states, like the current target that is only defined in our Chasing and Attacking logic if the Patrolling state properly passed it on.

This makes finite state machines interesting but also risky tools: as your behaviour starts to grow and you add more states, you'll also need more transitions and more coupling between those various state scripts.

So, even if, in this precise use case, we are lucky enough to have some sharable logic between the GuardFSM and the keep in mind that it's not always the case and that state machines can get out of hands...

Creating the Idle state

But for now, let's not complain about our cool situation, and take advantage of these shared behaviours and data properties to quickly design the enemy's AI.

Because, as we've said before, we know that the only new state we need to add to finish the Roman invaders' behaviour is the Idle state.

And guess what? This state is super basic!

In short, it's just the "check for targets" part of the Patrol logic. The only difference is that we need to look for objects on the "Player" layer, instead of the "Enemy" one. And, in the Enter() override of our state, we'll tell the unit to switch back to its "Idle" animation, if need be.

So the code of this Idle class looks as follows:

Idle.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public class Idle : BaseState {
5      private Animator _animator;
6      private Transform _transform;
7      private float _fov;
```

```
 8     private const int _playerLayerMask = 1 << 8;
 9
10     public Idle(UnitFSM fsm) : base("Idle", fsm) {
11         _animator = fsm.animator;
12         _transform = fsm.transform;
13         _fov = fsm.fieldOfVision;
14     }
15
16     public override void Enter(params object[] transitionArgs) {
17         _animator.SetBool("Running", false);
18     }
19
20     public override void LateUpdate() {
21         base.LateUpdate();
22
23         Collider[] targets = Physics.OverlapSphere(
24             _transform.position, _fov, _playerLayerMask);
25         if (targets.Length > 0) {
26             _fsm.ChangeState(_fsm.GetState("chasing"), targets[0].transform);
27         }
28     }
29 }
```

Thanks to these few lines, we now have a basic fallback state for the enemies that restores the right animation and then

checks for nearby targets to optionally transition to another part of the state machine.

Fixing the Chasing reverse transitions

Another little fix we need to do to truly share the Chasing logic between the GuardFSM and the is to ensure that we use the right unique codes for our state instances.

At the moment, you'll recall that we have:

three "patrolling", "chasing" and "attacking" states in the

three "idle", "chasing" and "attacking" states in the EnemyFSM _states Dictionary

We've named them like this because it felt natural to give a state the name of its class as a code; but nothing forces us to do so. And, in our case, it actually causes a little issue.

Indeed, back in our Chasing class, we said that, in the if the distance to the target gets over the field of vision, we should switch back to the state with the code "patrolling" (see *line*

Chasing.cs

```
1  public override void Update() {
2      base.Update();
3
4      float d = Vector3.Distance(_transform.position, _target.position);
5      if (d < _attackRadius) {
6          _fsm.ChangeState(_fsm.GetState("attacking"), _target);
7      } else if (d > _fov + 0.5f) {
8          _fsm.ChangeState(_fsm.GetState("patrolling"));
9      } else {
10         _transform.position = Vector3.MoveTowards(
11             _transform.position, _target.position,
12             _speed * Time.deltaTime);
13         _transform.LookAt(_target.position);
14     }
15 }
```

This works great for our Egyptian guard, but what about the enemies, whose FSM only contains an “idle” state as a fallback?

The solution here is to take some liberties and rename our “patrolling” state “idle” in the then, in the Chasing logic, we’ll

modify this reverse transition and tell it to go back to the state with the code “idle” in the

---

GuardFSM.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using FSM;
4
5  public class GuardFSM : UnitFSM {
6      private Transform[] _waypoints;
7
8      protected override void Awake() {
9          base.Awake();
10         _states = new Dictionary<string, BaseState>() {
11             { "idle", new Patrolling(this, waypoints) },
12             { "chasing", new Chasing(this) },
13             { "attacking", new Attacking(this) },
14         };
15     }
16
17     protected override BaseState GetInitialState() => _states["idle"];
18 }
```

Chasing.cs

```
1  public override void Update() {
2    base.Update();
3
4    float d = Vector3.Distance(_transform.position, _target.position);
5    if (d < _attackRadius) {
6      _fsm.ChangeState(_fsm.GetState("attacking"), _target);
7    } else if (d > _fov + 0.5f) {
8      _fsm.ChangeState(_fsm.GetState("idle"));
9    } else {
10     _transform.position = Vector3.MoveTowards(
11       _transform.position, _target.position,
12       _speed * Time.deltaTime);
13     _transform.LookAt(_target.position);
14   }
15 }
```

This way, if we're in the the "idle" state will actually be an instance of the Patrolling class; but if we're in the it will be an instance of the Idle class we just prepared. And both will

properly serve as fallback for the finite state machine of the entity!

Setting up healthpoints and death mechanics

To end this chapter, let's work on one last feature for our units' finite state machines – a health pool and a death mechanism.

Creating the healthbars

First of, we're going to edit the "Enemy" and the "Guard" prefabs we've been using until now and add a Quad primitive inside each of them to make a healthbar. This rectangular mesh is very simple but, thanks to the right material, we can make it look pretty neat, as you can see on *Figure*

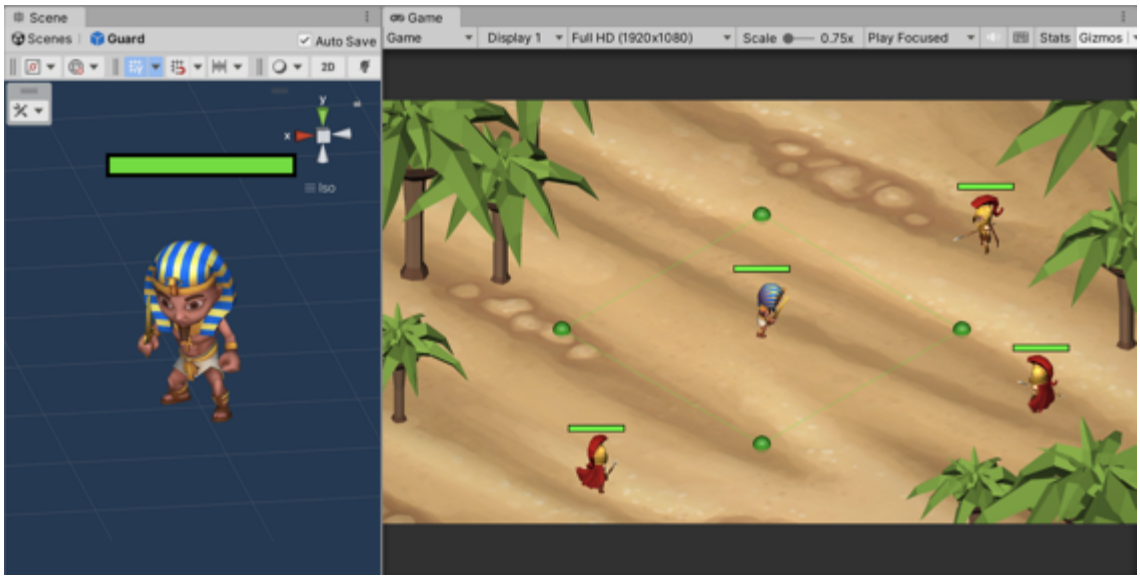


Figure 4.15 – An overview of the new healthbars above our units

Basically, here, I've given my healthbar a material with a custom **shader** I made, "Healthbar", that has three main properties:

Its colour and fill amount are auto-computed based on the current value of its normalised health parameter.

It has a dynamic border with a tweakable colour and thickness.

It forces the mesh it is applied on to always face the camera thanks to a neat technique called

---

WANNA LEARN MORE ABOUT THIS SHADER?

> Note that the project is setup to work with one of Unity's most recent rendering pipeline, the **Universal Render Pipeline** (URP). This shader relies on the Shader Graph which is a tool that only works with this new URP pipeline.

> However, since shader creation is not the point of this book, I won't go into more details about this shader here – don't hesitate to have a look at the Github repo (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) to check out this asset for yourself :)

Now, that our objects are ready, time to get to scripting!

Updating our units health

Because we want both our guard and the enemies to have these health and death systems, we're going to implement most of them in our UnitFSM base class.

And, first of all, we're going to add some health-related variables to the class and prepare a TakeDamage() method that updates the healthbar visuals according to these variables. So let's create:

a public maxHealth variable to make it easy to set the initial and max health points of our various units in the **inspector** panel

a private _currentHealth variable to keep track of the current amount of health points of the unit – this is the variable that we'll want to modify when the unit takes a hit

a reference to the Renderer component of our “Healthbar” child object in the unit’s hierarchy – this will allow us to update the object’s visual (by changing the value of the health parameter of its shader instance)

We’ll setup these three variables in our UnitFSM along with the rest:

UnitFSM.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public abstract class UnitFSM : StateMachine {
5      public float speed = 2f;
6      public float fieldOfVision = 3f;
7      public float attackRadius = 1f;
8      public float attackRate = 2.4f; // in seconds
9
10     [HideInInspector] public Animator animator;
11
12     public Renderer healthbarRenderer;
13     public int maxHealth = 10;
14     private int _currentHealth;
```

```
15
16      protected virtual void Awake() {
17          animator = GetComponent<Animator>();
18          _currentHealth = maxHealth;
19      }
20  }
```

(Of course, don't forget to actually drag the reference for the healthbarRenderer in the unit's hierarchy in the **inspector** panel!)

Now, we need to take care of implementing our TakeDamage() function. We're going to proceed in three steps:

1. First, we'll setup our method to return a boolean to tell the caller whether this last hit was fatal and killed the unit. So we'll say that, if after reducing the _currentHealth variable of the unit, we've reached 0 healthpoints, then the unit is dead and we return else, we return

---

UnitFSM.cs

```
1  using UnityEngine;
```

```
2  using FSM;
3
4  public abstract class UnitFSM : StateMachine {
5      // ...
6
7      protected virtual void Awake() { ... }
8
9      public bool TakeDamage() {
10         _currentHealth--;
11
12         if (_currentHealth <= 0) {
13             return true;
14         }
15
16         return false;
17     }
18 }
```

---

2. If we reach 0 healthpoints, we'll also make the FSM transition to its "die" state – this will be an instance of a new state class, that we'll code in just a second:

UnitFSM.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public abstract class UnitFSM : StateMachine {
5      // ...
6
7      protected virtual void Awake() { ... }
8
9      public bool TakeDamage() {
10         _currentHealth--;
11
12         if (_currentHealth <= 0) {
13             ChangeState(GetState("dying"));
14             return true;
15         }
16
17         return false;
18     }
19 }
```

3. Finally, we have to actually update the visuals and modify our healthbar to reflect the new health ratio of the unit. For this, we'll rely on a fairly advanced Unity tool, the which is,

simply put, a way to modify the parameters of a specific material instance without impacting the other copies in the scene.

SLIGHTLY OFF-TOPIC

> The MaterialPropertyBlock concept that I'll discuss in the following paragraphs is not directly linked to AI programming. If you already know about MaterialPropertyBlocks, or if you're not interested in diving into this right now, feel free to skip to the code snippets – you won't miss anything that is AI dev-related :)

Using MaterialPropertyBlocks is a cool technique to avoid a common issue with materials and prefabs which is that, by default, all instances of a prefab share a reference to the same material asset, and therefore share the exact same material properties, too. In other words, if I were to simply change the value of the health parameter of my guard healthbar's material directly, then all the other objects with this material in the scene (such as the enemies' healthbars) would update as well.

MaterialPropertyBlocks let us update just one specific material instance via the Renderer component that uses it. And because a single MaterialPropertyBlock variable can be used multiple

times to update several instances one after the other, a common pattern is to declare it as a

So we'll use a read-only C# getter that either initialises or returns our MaterialPropertyBlock variable, and then in the TakeDamage() function, we'll use this variable to update the health property of the material on the healthbar object that is in our unit's hierarchy:

UnitFSM.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public abstract class UnitFSM : StateMachine {
5      //...
6      private static MaterialPropertyBlock _mpb;
7      private static MaterialPropertyBlock _Mpb {
8          get {
9              if (_mpb == null) _mpb = new MaterialPropertyBlock();
10             return _mpb;
11         }
12     }
13
14     protected virtual void Awake() { ... }
```

```
15
16    public bool TakeDamage() {
17        _currentHealth--;
18
19        healthbarRenderer.GetPropertyBlock(_Mpb);
20        _Mpb.SetFloat("_Health", _currentHealth / (float) maxHealth);
21        healthbarRenderer.SetPropertyBlock(_Mpb);
22
23        if (_currentHealth <= 0) {
24            ChangeState(GetState("dying"));
25            return true;
26        }
27        return false;
28    }
29 }
```

Now, let's create this new Dying class we want to use when our unit's healthpoints reach 0.

Coding up the Dying state logic

To handle these health and death mechanics, we're going to slightly modify the FSM of our units and give them a new Dying state *4.16* shows the updated state machine diagram for the guard AI).

This Dying state will be final, meaning that there is no way to transition back from this state to other states in the state machine, and it will be accessible from any other state in the FSM.

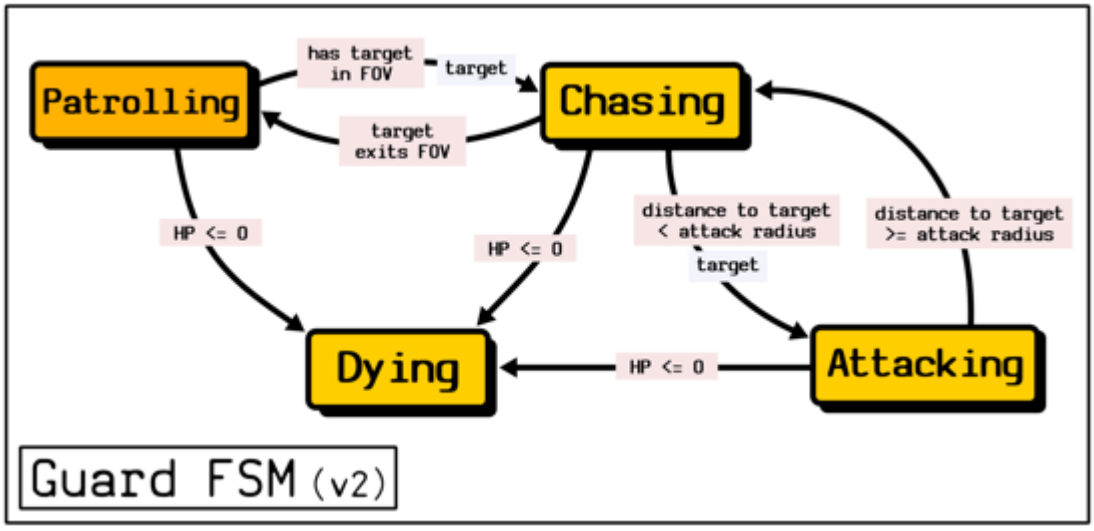


Figure 4.16 – Updated diagram of the finite state machine for our guard AI with the new final Dying state

The Dying class that implements this new state will be very short and pretty similar to the other state classes we coded before. Its logic will be two-fold:

When we enter this state, we'll trigger the "Die" animation on our 3D model.

We'll also use a reference to the BoxCollider component on the unit to disable this component – this will ensure that this target becomes "invisible" to the target search process we have in our Patrolling and Idle classes.

Here is the entire code of the Dying C# state class:

Dying.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public class Dying : BaseState {
5      private Animator _animator;
6      private BoxCollider _collider;
7
8      public Dying(UnitFSM fsm) : base("Dying", fsm) {
9          _animator = fsm.animator;
10         _collider = fsm.GetComponent<BoxCollider>();
11     }
12
13     public override void Enter(params object[] transitionArgs) {
14         _animator.SetTrigger("Die");
15         _collider.enabled = false;
16     }
```

```
17 }
```

And here are the GuardFSM and EnemyFSM updated scripts with this new class instantiated as a “dying” state in the _states

---

GuardFSM.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using FSM;
4
5  public class GuardFSM : UnitFSM {
6      private Transform[] _waypoints;
7
8      protected override void Awake() {
9          base.Awake();
10         _states = new Dictionary<string, BaseState>() {
11             { "idle", new Patrolling(this, waypoints) },
12             { "chasing", new Chasing(this) },
13             { "attacking", new Attacking(this) },
14             { "dying", new Dying(this) },
15         };
```

```
16     }
17
18     protected override BaseState GetInitialState() => _states["idle"];
19 }
```

EnemyFSM.cs

```
1  using System.Collections.Generic;
2  using FSM;
3
4  public class EnemyFSM : UnitFSM {
5      protected override void Awake() {
6          base.Awake();
7          _states = new Dictionary<string, BaseState>() {
8              { "idle", new Idle(this) },
9              { "chasing", new Chasing(this) },
10             { "attacking", new Attacking(this) },
11             { "dying", new Dying(this) },
12         };
13     }
14
15     protected override BaseState GetInitialState() => _states["idle"];
16 }
```

We're almost done – the last thing to do is update our Attacking logic so that it actually calls the TakeDamage() function and triggers this whole new chain of events!

Modifying the Attacking logic

To have our Attacking class integrate this new hit logic and check for a possible death of the current target, we just have to add a couple of lines to our Update() method.

Basically, whenever the attack counter is reset and we start a new sword slash, we'll call the TakeDamage() function of our target's UnitFSM class (or, more precisely, its instance of a class, i.e. GuardFSM or and we'll check to see if this hit killed the target. If it did, then we'll tell our unit to transition back to its "idle" state (which, as we've seen before, is the Patrolling state for the guard, and the Idle state for the invaders).

(Note that here we'll use the C# **polymorphism** feature to ask for a UnitFSM component on our target and yet retrieve one of its derived classes like GuardFSM or

Attacking.cs

```
1  using UnityEngine;
2  using FSM;
3
4  public class Attacking : BaseState {
5      // ...
6
7      public Attacking(UnitFSM fsm) : base("Attacking", fsm) { ... }
8      public override void Enter(params object[] transitionArgs) { ... }
9      public override void Exit() { ... }
10
11     public override void Update() {
12         base.Update();
13
14         _attackCounter += Time.deltaTime;
15         if (_attackCounter >= _attackRate) {
16             _animator.SetTrigger("Attack");
17             _attackCounter = 0f;
18             bool targetIsDead = _target.GetComponent<UnitFSM>().TakeDamage();
19             if (targetIsDead) _fsm.ChangeState(_fsm.GetState("idle"));
20         }
21
22         float d = Vector3.Distance(_transform.position, _target.position);
23         if (d > _attackRadius) {
24             _fsm.ChangeState(_fsm.GetState("chasing"), _target);
```

```
        }
    }
}
```

Our two AIs are now ready, and if we start the game, we can see the scene play before our eyes step-by-step:

Figure 4.17 – Our guard is now able to damage these menacing invaders and make the land secure again, before returning to his routine patrol!

Our guard initially starts its patrol and begins to walk along the path towards the next waypoint in the list. Then, when an enemy enters its field of vision, it goes after him to eliminate the threat. During the fight, both units damage the other and the healthbars update to reflect these changes – until, eventually, our guard wins and kills the invader. His mission accomplished, our player unit then returns to its fallback Patrolling state, ready to repeat his performance...

YOUR TURN!

> Depending on the animations you use, you may want to add some delay between the moment you start the animation and the moment the enemy takes a hit, to make the whole thing more realistic. For example, here, I'm dealing some damage to the enemy right when the animation starts, even though the sword doesn't hit the other model before about a second.

> So – can you improve this script to integrate some delay in the animation and better match the move visuals? :)

> Don't hesitate to share your ideas and improvements via email (at mina.pecheux@gmail.com), or even on the Github (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) as an issue, so that other creators can react too!

Summary

This was a pretty long chapter! We've talked about a lot of things, and we've gradually implemented a complete finite state machine-based AI (well, two, actually) that is capable of a significant amount of behaviours.

We started by preparing some abstract C# classes, the BaseState and the to make ourselves a basic FSM toolbox.

Then, we used this toolbox extensively to design and build the AI of a little guard and its menacing enemies. From sending data using the params keyword and the boxing/unboxing process, to checking for nearby targets with Unity's Physics built-in methods, or communicating with the Animator component of our 3D models, we've explored a wide variety of game tools.

We also briefly discussed the notions of modularity and decoupling, and we saw how one finite state machine could sometimes be adapted and/or extended slightly to model the behaviour of another entity.

However, as we said in *Chapter* this re-usability is not always possible with the classic FSM model, and we're sometimes limited by the rigid structure of state machines.

That's why in the next chapter, we'll see how some of those limitations can be mitigated by enhancing the finite automaton architecture into more advanced structures like the concurrent FSMs, hierarchical FSMs, stochastic FSMs and pushdown automata.

5 - Upgrading your finite state machines

In the last couple of chapters, we've studied the FSM architecture extensively.

When we first introduced finite state machines, back in *Chapter 3: What are* we spent some time analysing the strengths and the shortcomings of FSMs. And, among other things, we said that, because they rely on a very rigid structure, those machines sometimes fall short or bring quite the complexity.

To overcome this problem, there are several possible solutions to upgrade our FSM models and give them more power. So, in this final chapter on state machines, we'll explore some of these "expansions", namely: concurrent FSMs, hierarchical FSMs, stochastic FSMs and pushdown automata.

WAIT, THERE's no code here?

> The following sections are about presenting these extended versions of the state machine architecture, and I want to focus more on what the gist is and why they can be interesting than on the exact implementation details of those variations in Unity/C#.

> But, with FSMs being so popular, you should be able to find numerous examples of how to code those up on the net :)

Concurrent finite state machines

Alright – we know that state machines can impose too much constraints on developers and create unnecessary frustration, in particular when you have states that are similar but not exactly the same.

In this first section, we're going to discuss one technique for somewhat dodging this issue and, most importantly, avoiding the exploding growth of finite state machines by using not just one, but multiple finite state machines cleverly linked together...

Let's take an example!

To figure out why and how this technique can help, we're going to consider a simple use case where a basic FSM would be too limited – in truth, it will be the continuation of the example we mentioned in *Chapter* where we had a fighter that could be either empty-handed or hold a gun.

To recap, we'll assume the situation is the following:

We're working on a basic 2D Metroidvania side-scroller.

We have a hero that is controlled by the player and therefore doesn't need any AI.

But we also have humanoid enemies with similar capabilities who, most notably, can do the usual actions you'd expect in this type of game from a human-like creature: running left and right, jumping, crouching and attacking.

To make the game more challenging, those entities will also be able to equip new weapons to improve their combat statistics. We'll ignore the "spot and collect" sequence here and consider that the unit already has one or more weapons in its backpack.

The game contains four types of weapons: none (meaning the unit is empty-handed), knives, light guns and heavy machine guns.

The attack logic depends on the type of weapon that the unit currently has equipped, which can be: melee fighting if the unit has no weapon or a knife, or ranged fighting if the unit has a gun (either light or heavy).

The move speed will also depend on the type of weapon that the unit currently has equipped: it will be normal if the unit has no weapon, a knife or a light gun, and slightly lowered if the unit has a heavy gun.

The unit will not be able to jump or crouch if it carries a heavy machine gun.

From a gamer's point of view, having enemies with this kind of behaviour doesn't sound too crazy – actually, today, we're sort of expecting *at least* this level of "intelligence" from our opponents. However, on the technical side, the combination of all those actions, reactions, equipments and objectives is no piece of cake to implement!

Suppose we come at this naively and try to design a simple finite state machine to model this behaviour. The most logical train of thoughts would be to consider the main actions of our AI and make an automaton with five states (see *Figure* Idle, Running, Jumping, Crouching and Attacking.

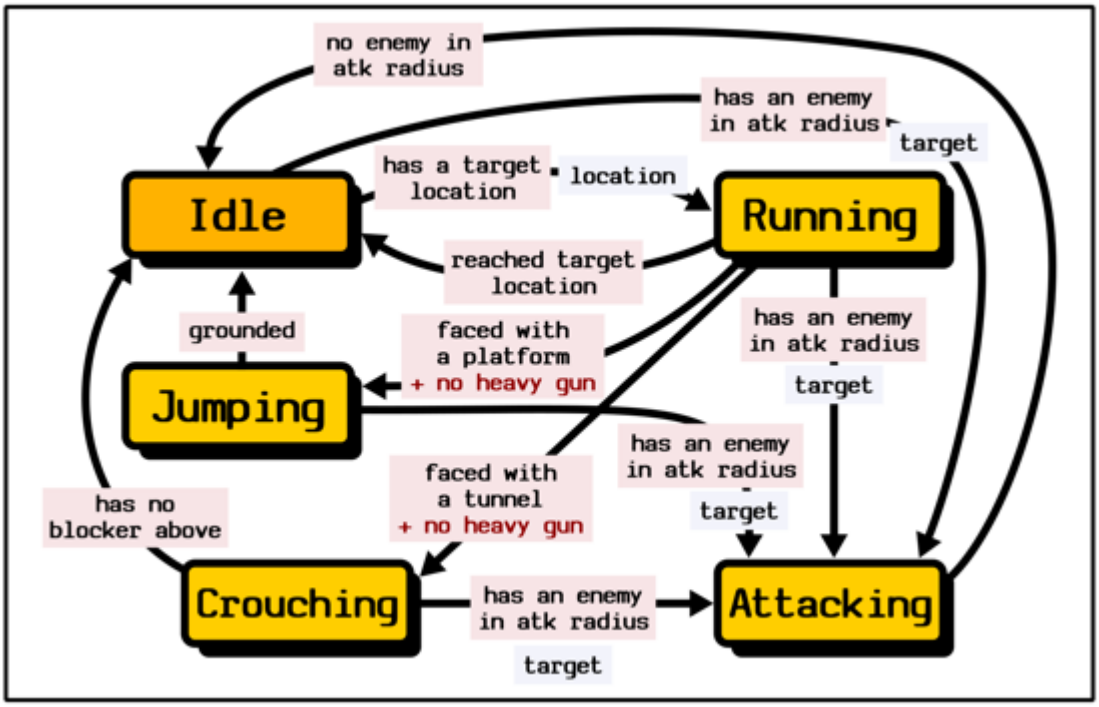


Figure 5.1 – A naive V1 of the diagram of our entity's FSM-based AI

Deciding on the exact transition conditions could require a bit of time, but this diagram looks like a good first representation of the core logic of the AI, right?

The problem is that, of course, we don't really see all the details of our logic just by looking at *Figure* For example, this diagram doesn't show the differences between the melee and the ranged fighting logics in the Attacking state; or the different move speeds in the Running state.

In other words, our diagram is already a bit of a pain to read, and yet it's incomplete.

To truly design our finite state machine with all these particularities, we would need to distinguish in the diagram between the different variations of Attacking and Running states. So we'd get this updated system:

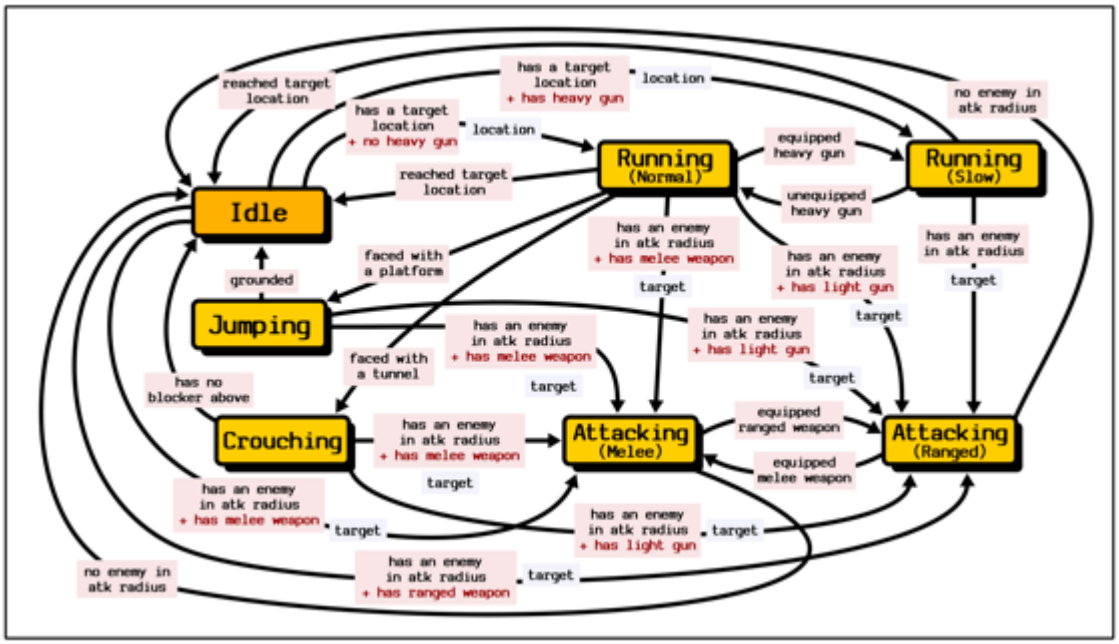


Figure 5.2 – A more accurate V2 of the diagram of our entity's FSM-based AI with the distinction between the Attacking and Running state variations

The diagram has grown significantly, and it's becoming hard to properly think of all the possible transitions. (And yet, I've

simplified things by defaulting back to the Idle state whenever I could, typically when exiting the various Attacking states...)

Something that is very visible on *Figure 5.2* and that clearly complicates things a lot is that the rigid FSM architecture prevents us from decoupling the state logic from the decision logic – meaning that any variation of a state has to handle the transitions to other states, even if this part of the logic is an exact copy from one state to another (for example, in *Figure* we see that there are two arrows going out of the Attacking states that both check for the "no enemy in attack radius" condition and bring the machine back in its Idle state).

Now, the bad news is that we're not done.

Indeed, we also need to take into account that in a real game, we don't just have abstract data and script variables; we have components and visuals to keep in sync with the current state of the entity, too. Typically, even though the type of weapon the unit carries doesn't impact the Jumping and Crouching states in terms of behaviour logic (they only impact the possible transitions in the machine), they do change the animations the unit will use, or perhaps the VFX to instantiate.

Thus, to truly *truly* design the FSM will this extra layer of component-level specificities, we would need to further split the

states into more variations to differentiate between each weapon type... and we'd end up with a real mess:

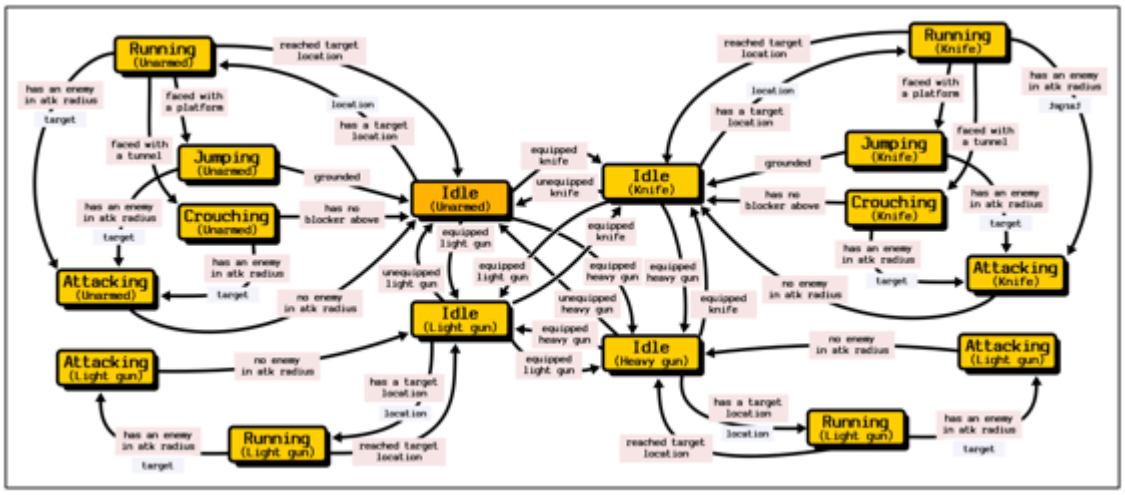


Figure 5.3 – The real V3 of the diagram of our entity's FSM-based AI that distinguishes between all "action-weapon" combos. Yes, it's a pain to read, and pretty much unusable.

Figure 5.3 isn't very reassuring. It contains so much code redundancy it hurts the eye, and it would be an absolute hell to implement and maintain as-is.

So, at this point, it's looks clear that our basic state machine is absolutely incapable of handling this AI. Just imagine we decide to add one state, and suddenly we would need to pop up a dozen variations to deal with all the possible "action-weapon" pairings!

Here, we experience the uncontrollable growth that we discussed in our introductory chapter to FSMs – which is an unfortunate consequence of the formidable power of mathematics.

Understanding the devilish power of combinatorics

The reason why we're faced with this exploding growth of our FSM's size is because this model is, by its very nature, not very good at representing combinations efficiently.

More precisely, you can easily represent states where your entity is running and has a gun, or is running and does not have a gun... but you'll need to code up (and worse, maintain!) both independently. So, in our case, we'd have to implement a state in our machine for each pairing of action and weapon type.

You see how this lacks **scalability** – if we decide to add one action or one weapon type to our gameplay, it will instantly create a whole bunch of states in our FSM! The problem here is that because our states must represent an "action-weapon" combo, their number will increase rapidly even if we add just one item in one category.

If you're into maths (and, in this case, a sub-field of it called combinatorics), then you'll now that if we have **X** possible

actions and **Y** weapon types, we'll need **X** • **Y** states to get all the combos. And if we ever decide to consider a third factor for our combos, for example the current armour the unit is wearing, among **Z** possible types, then the number of states will suddenly jump to **X** • **Y** •

What's worse is that a lot of those **X** • **Y** • **Z** states will have very similar implementations – it will mostly be about mix-and-matching the right logic chunks depending on the value of each factor in the combo!

Basic FSMs are therefore not the best architecture for modelling such a behaviour... but what if we could boost their power by smartly linking multiple automata together?

The concurrent FSMs trick

We know that the issue we're facing in this case comes from this strong entanglement between the action and the weapon the unit is carrying. The solution would thus be to try and separate those two concerns to handle each separately, and avoid the exponential growth if we decide to add items in one of the two categories.

Suppose we create not one, but two FSMs for our unit: the Action FSM and the Weapon FSM. *Figure 5.4* shows the

diagrams of each, where you see that we have decoupled the entity's actions from its currently equipped weapon "state".

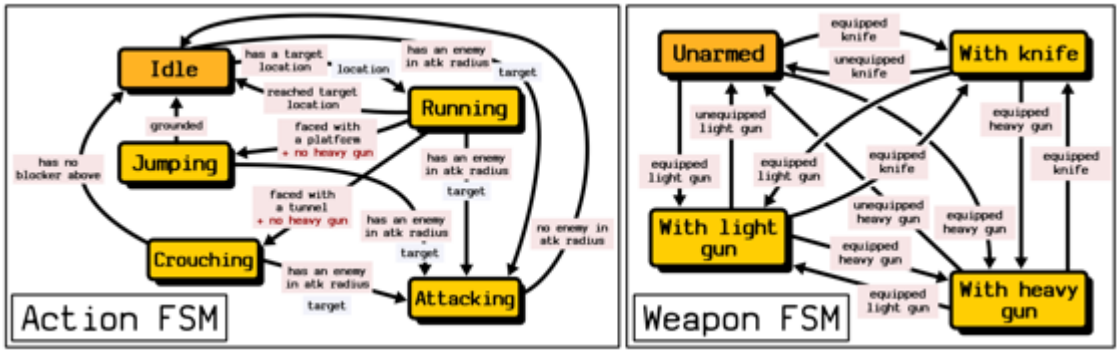


Figure 5.4 – Diagrams of the two concurrent FSMs (the Action FSM and the Weapon FSM)

Even if it's slightly less intuitive, those two **concurrent finite state machines** (or could help us implement the behaviour for our unit by handling the global action on one side, and the weapon-specific logic on the other. For example, we could have the Enter() method of the different states in our Weapon FSM properly change the animations and other visual properties of the entity, so that we don't have to repeat this logic in multiple places like before. And then, we could say that if the Action FSM is in the Attacking state, it delegates the exact logic to execute to the Weapon FSM.

Of course, our two state machines aren't totally unrelated – think of how the weapon can prevent some actions like Jumping and Crouching, and how it impacts the move speed

in the Running state. So we would need to have global data stored at the entity-level, alongside the reference to each FSM, which can be used by both FSMs in their state logics to properly adapt the unit's behaviour. But still, by using CFSMs, we could at least break the combos, and avoid the scalability issue.

CONSUMING INPUTS

> As we discussed in *Chapter 3*, a core idea with finite state machines is that you're checking for specific inputs to trigger your transitions from one state to another. When you have concurrent state machines, this can become more tricky because inputs may only make sense for one automaton, but still try and trigger something in the other. And if your code isn't perfectly designed, some of the reactions of the second machine might counteract or disturb the reactions of the first.

> To avoid this, it can be interesting to **"consume" the inputs** when you react to them – or in other words, consider that once the first machine has responded to this trigger, it should be discarded or ignored by any other machine in the system of CFSMs for this frame.

Concurrent finite state machines are therefore an interesting technique for sidestepping the growth issues of FSMs, and handling more complex logics that can, to a certain degree, be decoupled enough to live in separate automata.

But there is another common issue with simple state machines, which is that of similar states that *do* need to stay in the same system. So let's now move on to the second trick in the FSM AI developer's bag: the hierarchical FSM.

Hierarchical finite state machines

We've seen that concurrent FSMs are an interesting way of reducing complexity when we have "combo states" that mix together relatively disconnected concepts, such as an action and a piece of equipment.

Now, what if, instead, we had actions that looked similar and shared some logic, but couldn't simply be exported to a second state machine like this? Like if our entity behaviour required some sort of multi-level brain, with global reactions that may occur in several states?

In that case, another possible technique is to take advantage of the hierarchical finite state machine architecture.

Aggregating logic?

Let's keep going with the example of the 2D Metroidvania game and the multi-weapon humanoid entity we discussed in the previous section. Using concurrent FSMs, we managed to separate the weapon-related stuff and have two automata with better-defined roles.

But suppose that, now, we decide to add new actions to this unit: walking and sneaking. Those actions work as follows:

If the unit enters an area with mud, then it is slowed down and can't run anymore – it can only walk. The Walking and Running states are very similar, apart from the move speed and the animations.

Rather than attacking, the unit can now also assess if the enemy is way stronger and, if so, try to hide by entering its Sneaking state. This logic is again similar to one of the Walking or Running states, but with a different move speed and other animations.

To integrate these new behaviours in our entity's FSM, it doesn't really make sense to create a new FSM, this time. Clearly, these walking and sneaking actions should remain in the Action FSM.

So we could of course simply create two states, Walking and Sneaking, with C# classes that contain a slightly modified snippet based on the Running state logic:

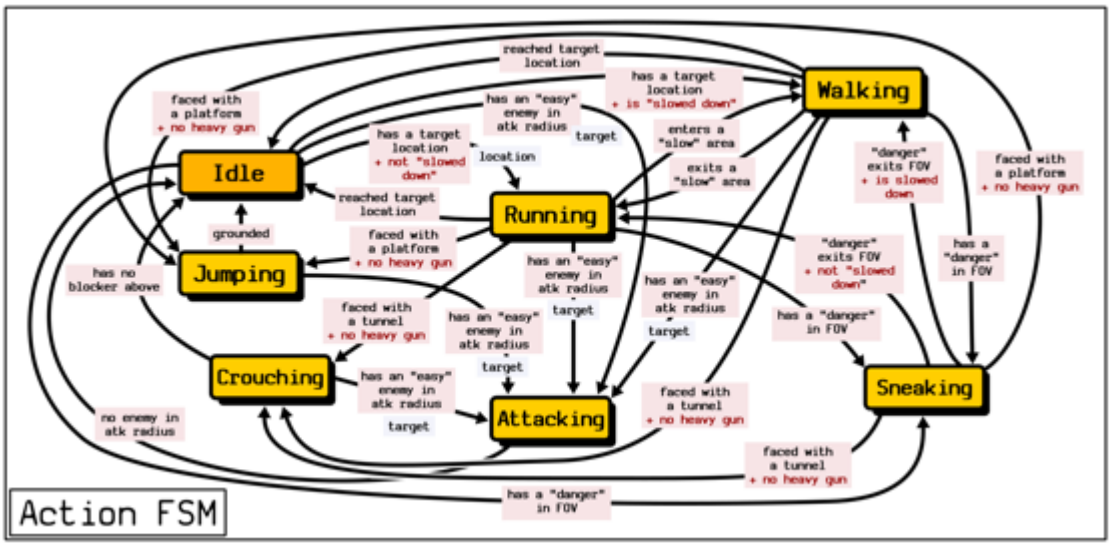


Figure 5.5 – Updated diagram of the Action FSM with the new Walking and Sneaking states

However, we've already said repeatedly that having codes that are almost-the-same-but-not-quite is a bad idea, because it creates redundancy and often inconsistencies. Plus, with finite automata, adding one state also means adding a lot of transitions – so again, not a good idea.

To make our FSM more maintainable, a better solution would be to aggregate the **shared logic** of our Walking, Running and

Sneaking states somewhere, to have it centralised in just a single location in our codebase. Then each state would just specialise the logic in its own way, to set the right speed and animations.

These notions of shared and specialised behaviours probably rings a bell: this is the core principle behind the **inheritance** paradigm we introduced in *Chapter 3* when we discussed our FSM C# toolbox implementation! And indeed, here, we can re-use the notion of inheritance to help us solve our issue. Because what are these Walking, Running and Sneaking states, if not derived versions of a global Moving state?

Creating a hierarchy of states

So let's say that, rather than writing two new C# classes with redundant code for the Running and Sneaking states, we try to introduce a hierarchy of state classes to abstract and share some parts of the common behaviour.

What we could do is create a Moving class that handles the movement logic of our unit, plus the decision logic to transition to other states in our machine, and then have the Walking and Sneaking C# classes inherit from this Moving class. These derived classes would simply override the move speed and define the animations to use, but we wouldn't need to repeat the logic anymore:

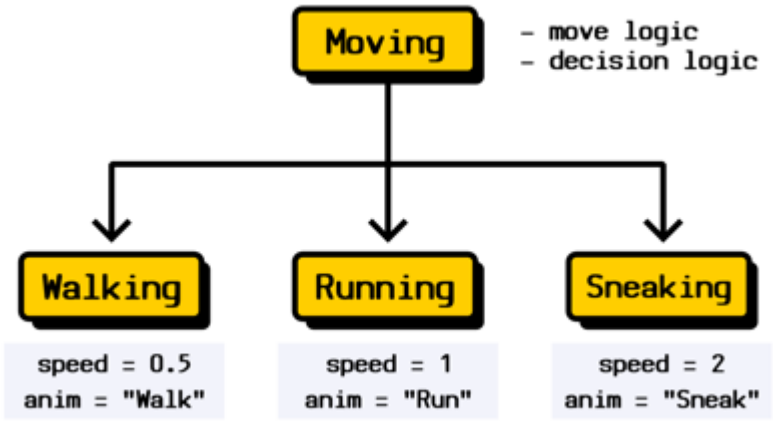


Figure 5.6 – States hierarchy for our unit's movement with an abstract Moving state, to aggregate the Walking, Running and Sneaking logics

The advantage of bundling together the logic of these states is that this really simplifies our entity's state machine, since we come back to the Action FSM we had in *Figure* even if it now also contains the two new walking and sneaking actions of our unit:

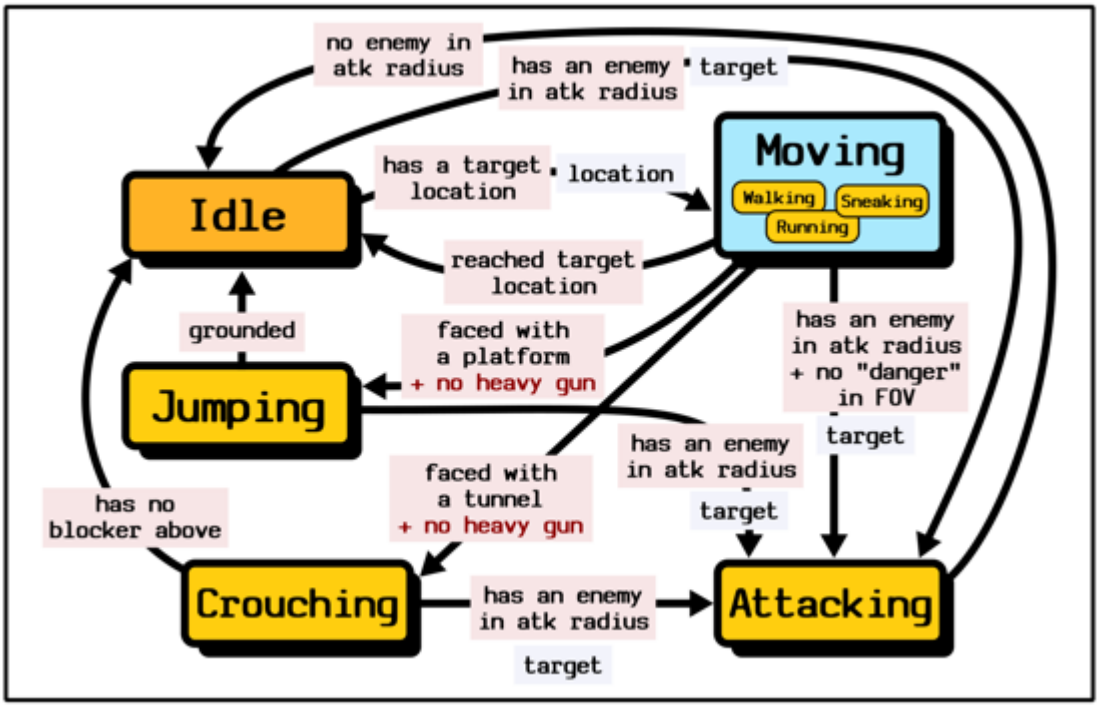


Figure 5.7 – Updated diagram of the Action FSM with the Running, Walking and Sneaking states aggregated as an abstract Moving state that centralises the logic

(In *Figure* the blue rectangle represents an aggregated high-level state.)

This upgraded FSM architecture with multi-level states actually has a name: the **hierarchical finite state machine** (or It is an adaptation of the finite automaton where, in addition to the usual states, you also have **superstates** (like Moving here) and **substates** (like Running, Walking or Sneaking here).

Note that in this example, I'm showing a sort of **hybrid** between the usual "flat" FSM and the HFSM. Generally speaking, in a HFSM, we have multiple superstates that each contain two or more substates, and everything works with this multi-tier structure. Plus, we often enforce that:

Superstates can only transition to other superstates.

Substates can only transition to the substates in the same group (i.e. the substates that are children of the same superstate).

So the hybrid FSM shown in *Figure 5.7* doesn't quite fit the mould for a HFSM... but it kinda works for our example, and it highlights something important with AI design, which is that architectures aren't set in stone. Having a knowledge of the most common techniques in the field is great because it gives you a starting point and, in lots of situations, it directly suggests a framework for coding up your solution; but you should also feel free to go beyond those well-known structures and adapt them to your own use case if need be. (By the way, we'll discuss this in more details in the very last chapter of this book, *Chapter 12: Expanding your*

To sum up, even though it doesn't solve all use cases, the HFSM can be a cool way of centralising and sharing a chunk

of logic between multiple states while keeping everything neatly organised in the same state machine.

Just like concurrent FSMs, hierarchical FSMs are therefore a solution to the rigidity and high coupling that finite state machines inherently produce between their states. They're basically little "extras" you can try if you feel like the behaviour you want to model is too large to be modelled with traditional states and transitions.

AN ALTERNATIVE IMPLEMENTATION

> Here, I've talked about how to implement hierarchical finite state machines with C# inheritance and derived classes, but this is not the only way to use this architecture in our code. In his amazing book Game Programming Patterns, Bob Nystrom mentions an alternative technique to create hierarchical FSMs using a stack of states. If you want to learn more, you can check out the dedicated chapter in Nystrom's book for free on his website over here: https://gameprogrammingpatterns.com/state.html.

Parallel to these scalability issues, there is another big challenge when designing AI for video games, which is the creation of a somewhat realistic "intelligence" mimicking. As we've said in *Chapter 1: AI in* this is hard because we, as humans, are really used to seeing and interacting with

creatures deemed “intelligent” – so we instantly see the flaws and discrepancies in a parody.

And yet, over the years, AI programmers have found subtle ways of improving the illusion – such as, for example, adding some randomness.

Non-deterministic finite state machines

When we look at the living things around us, it often seems like there is a lot of randomness in their actions – typically, they don’t perfectly re-enact the same behaviour each time they are faced with the same problem.

That’s why, as we quickly explored in *Chapter 2: Designing a single-script robot* reproducing this randomness in our game AIs can be a nice trick to creating more lifelike entities.

Deterministic VS stochastic

In the world of programming, we often distinguish between deterministic and non-deterministic, or stochastic, processes. Basically:

When a process is completely fixed, and doesn’t contain any random parts, we say that it is In other words, deterministic

processes always run exactly the same and result in the exact same outcome when given the same inputs.

On the other hand, processes that use randomness are said to be With stochastic processes, you may get different results even though you use the same inputs.

Of course, the great thing is that this is not just a "one-or-the-other" kind of thing: there is actually a spectrum between fully deterministic and fully stochastic, and that's where, as an AI programmer, you can find a balance between natural-looking and controlled-enough behaviours.

And finite state machines are an architecture that can totally benefit from this "deterministic VS non-deterministic" concept.

Adding randomness to a FSM

Up until now, we've only worked on fully deterministic FSMs: we had states with a clear and unwavering logic that always executed the same given the context. In particular, the decision logic (i.e. the potential transitioning to other states) was based on fixed conditions. But an interesting evolution of this basic model is to introduce some randomness to give the impression that the entity is less robotic.

Let's continue our example from before of the Metroidvania unit AI. We'll consider the most recent version of our FSM diagram depicted in *Figure 5.7* with the new walking and sneaking actions.

We've said before that, in our gameplay, the Walking state is not really transitioned to by choice – the entity switches to this state if it enters a specific type of area. On the other hand, the Sneaking state is more of a conscious decision: the unit enters this state if there is a dangerous enemy in its field of vision.

However, to make its behaviour slightly more realistic, we could make the transitions to the Attacking and Sneaking state a bit more stochastic so that, sometimes, the entity feels courageous and faces the danger, and sometimes it rather tries to hide and disappear. It wouldn't be perfect, obviously, as this total randomness would denote of a pretty strange emotional compass, but at least if the player is around, she won't see the AI exhibit the exact same behaviour every time.

In truth, creating non-deterministic finite state machines can be difficult for two reasons:

From a design standpoint, it requires that you identify the behaviours that could rationally be interchanged and that you establish the right transition chances for each. Usually, this is

done by giving the transitions weights that indicate how probable it is that the unit chooses this one instead of the other, when the input that triggers either one is activated.

The weights you define for your transitions may represent a sort of overall **personality** for your AI (e.g. is it more brave and hot-headed? or discrete and sneaky?), but this means you have to be consistent throughout the automaton if you create multiple groups of stochastic transitions.

From an implementation standpoint, it can quickly lead to "flickering" reactions if you're not careful. Basically, if two states contain some contradictory decision logic (typically, in our example, the Attacking and Sneaking state that impose different reactions when close to a dangerous enemy), then choosing one at random and then letting its logic execute could also cause the state machine to transition back to the other alternative! And thus you'd get some endless loop of the unit picking one option, and then changing its mind, and then going back to the first decision, and then switching again...

To avoid this kind of annoying and totally unreasonable oscillations, you could either use different thresholds in your conditions to smooth out the transitions and better handle these undecided ranges of values; or you could try and separate the decision phase from the actual execution, so that

once the entity has chosen a state, it sticks with it for a given amount of time.

This last point actually brings us to an interesting thing about lifelikeness in AI. 'Cause sure, adding stochasticity in our behaviours may fake intelligence a bit better for our game entities. But there is still an issue with all the architectures we've presented so far, with all those FSM models, even the more advanced ones: those machines don't have any memory. Those automata live in the present in the purest sense, since they cannot technically retain information about their past states. But again – what if they could?

To wrap up this chapter and finish this overview of extended finite state machines, let's talk about one last architecture, called the pushdown automata.

Pushdown automata

Up to this point, the finite state machine we've discussed all had one thing in common: they could only be in one active state at a time, and this specific state was referenced in a variable. Which means that we could only switch to a new state by replacing the old one, and jump around the machine without any memory of our past contexts.

However, there is an interesting extension of state machines called the **pushdown automaton** that, thanks to a clever restructuring of the data, can alleviate this issue...

Understanding why memory is important

Before focusing on *how* pushdown automata can solve our memory issues, let's first see *why* it's valuable. And to do that, we'll keep going with the same example as in the rest of the chapter: our multi-weapon Metroidvania minion!

So – at this point, we've learnt a couple of technique for decoupling unrelated state factors into different state machines (with CFSMs), and sharing logic between resembling states thanks to inheritance (with HFSMs). But there is one issue we haven't really tackled, or even discussed, and that's to which extent our state machine is able to restore a past state.

If we come back to the base version of our Action FSM, shown on the left of *Figure* we see that there is a huge simplification of the AI logic in this diagram, since:

multiple states can transition to the Attacking state (actually, all but the Attacking state)

however, the Attacking state can only return to the Idle state

If we imagine that our state machine works instantaneously and transitions to new states as fast as it wants, then that simplification is fine – we can just use this Idle state as a fallback, or even a sort of "hub point", and then re-distribute the flow to the right state by following up with the right transition in our automaton.

But this is not how it happens in real life – or in game life.

In a program, every instruction takes time. Not much, but still. When you chain all those instructions, you get an incompressible minimal time period for running the logic of any state. Thus transitioning to the Idle state as a default, and only *then* to the right state based on current context of the entity is always slightly longer than going directly to the right state from the Attacking state. This is all the more true since our states would need to run some updates on the entity's component in their Enter() and Exit() entry points, which further slows down the execution and may cause visual glitches (such as an animation starting and stopping after a single frame).

To fix this issue, the answer would be to not make a detour via a default state, but instead directly go back to the state we were before entering and executing our current Attacking state. For example, we would love to have the following scenario:

The entity is in its Running state.

An enemy enters its attack radius, so the entity transitions to the Attacking state.

The enemy flees and gets out of the attack radius, so the entity transitions back to its Running state and continues its route to its destination point.

If we had that, it would significantly improve the realism of our AI since it would look like it has both **short-term and long-term** and that it has memorised an overarching plan that may be disturbed by localised events but always stays in the background.

Sadly, if we stick with a basic FSM, this is impossible, because the machine doesn't have any memory of where it was before its current Attacking state. The only solution would be to duplicate our Attacking state into four completely identical versions, one for each previous states, just to be able to link those variations to their right counterpart in the machine. We'd fall back into code redundancy and scalability issues, and even the readability of our state machine would seriously drop.

A better idea would be to not discard the previous state and replace it with the new one but, rather, keep a list of the past contexts to be able to restore the previous state. And this is where pushdown automata come in.

The secret of stacks

In a nutshell, with pushdown automata, instead of storing a single reference to an active state, we keep a **stack** of them.

If you're not too familiar with this type of data structure, you can picture it like a pile of plates. With a stack, you can do three things:

you can "push" an item on the stack, which puts it at the top, above the previous items

you can "pull" the topmost item off the stack, which liberates the top slot and brings back the previously pushed item as the new top of the pile

finally, you can also "peek" at the top of the stack: this doesn't change its content but it gives you a reference to the topmost item

Ok – that's great, but how does this help in our case?

Well, the big advantage of this technique is that, in a pushdown automaton, rather than the transitions *replacing* the current state, they simply add it to or remove it from a pile of current states, but *the other ones stay* The "real" active state that is executed is the topmost item of the stack, but the states underneath that were pushed previously aren't discarded. It's only when you pull the top state that you actually discard it from the stack and reveal the one below as the new active state.

With this new architecture, transitioning to a State A therefore means pushing your instance of the matching C# class on the states stack, and transitioning from State A means pulling this instance from the stack:

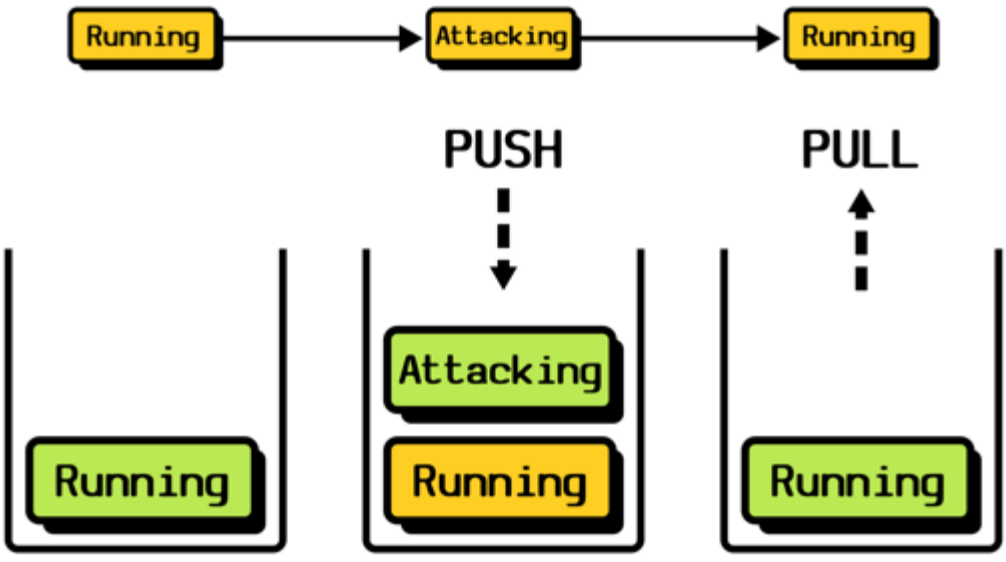


Figure 5.8 – Simplified representation of the state transition process in a pushdown automaton (with the "real" active state in green, and the matching transition link at the top)

This is great for us, because it means we can remember the previous contexts we went through and, if need be, restore those contexts when we exit our active state. For example, this would allow our Attacking state to transition back to Idle, Running, Jumping or Crouching without any detour, by pulling the Attacking C# instance from the stack and directly restoring the previous state.

To conclude, pushdown automata are an interesting technique for giving our finite state machines a **memory** of the past contexts, and better inform them on the best transition to take. This in turn helps simulate more long-term goals for the AI, which creates a more realistic behaviour.

Summary

In this chapter, we finished our exploration of the finite state machine model by exploring some possible improvements on the base FSM architecture.

We first talked about concurrent state machines (CFSMs), and we saw how by linking together multiple FSMs, we can sometimes avoid the exploding growth problem... if the states logic can reasonably be decoupled enough to live in different automata!

Then, we studied hierarchical state machines (HFSMs), and the idea of using C# inheritance to make superstates and substates and create a multi-level FSM. We saw how this pattern can help reduce code redundancy, in particular for the decision logic of the states.

We then briefly mentioned non-deterministic, or stochastic, state machines, and why introducing randomness in our decision logic can make a more lifelike AI.

Finally, we discussed the perks of giving our FSM-based AI some memory of its past contexts by analysing the pushdown automaton model and its stack of states.

So, throughout *Chapters 3, 4 and* we have examined the strengths and weaknesses of finite state machines, how we can implement them in a Unity/C# project, and how we may extend them to deal with more advanced behaviour modelling.

However, there are still many cases where the FSM architecture is too rigid, and you need more modularity in your AI design.

Which is why in the next part of this book, we'll move on to studying another technique for making game AI, that is one of the go-to solutions nowadays: the behaviour trees.

PART 3

BEHAVIOUR TREES

6 - Understanding behaviour trees

In the last chapters, we have studied a common tool for implementing AI in video games: the finite state machine. We saw both the theory behind it and some applications, and we now have a good understanding of when this technique can be interesting, when it lacks flexibility and what tricks we can try to upgrade this model and boost it a little.

To continue our journey, we're now going to shift gears and introduce another essential technique from the AI game dev's toolbox – the behaviour trees.

In this chapter, we'll discover what these trees are and how they draw inspiration from the data-driven philosophy to offer a loose coupling and modular AI design, before eventually taking a step back to outline their major advantages and limitations.

Getting started with behaviour trees

Although they are also used to model entity behaviour, **behaviour trees** (or are quite different from state machines.

Both tools implement task switching logics, but they rely on different data structures to do so.

The idea with behaviour trees is, rather than having decoupled states and transitions between them as we saw for FSMs in *Chapters 3 to* to work on a graph of nodes that describe all the possible actions of the entity and the flow to go from one to the other. Thanks to this fairly modular setup, BTs theoretically provide us with an easily maintainable, scalable and reusable AI-logic.

But this still doesn't explain what behaviour trees are, so let's start from the beginning and detail what these beasts are composed of!

Tree, root & leaves

Ok so, first things first: behaviour trees are trees. So far, so good.

Trees are themselves graphs that have only one node at the very top, the and then a hierarchy of child nodes beneath. In other words, each node can have zero or more children, and those children can themselves have children, and so on (see *Figure*

The nodes at the very bottom that have no children are called In our case, for a behaviour tree, those leaves are going to contain the actual checks and tasks performed by our AI. The rest of the tree, on the other hand, will be the decision logic structure that's going to "switch" from one behaviour branch to another.

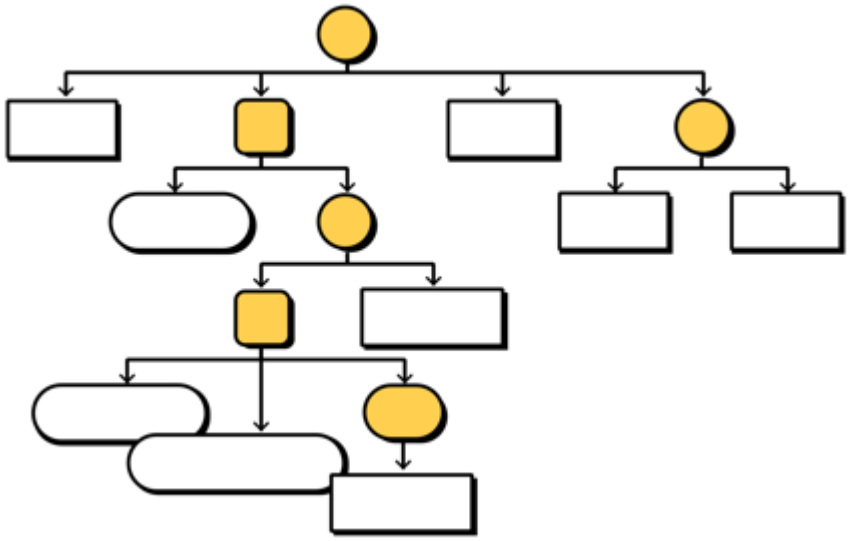


Figure 6.1 – Schematic representation of a behaviour tree

Contrary to the finite state machines we talked about in the previous chapters, behaviour trees are therefore composed of multiple atomic units (the nodes) with a fairly limited logic and a specific role, that are then combined together to form a more complex logic. This time, we don't have explicit

transitions and states: instead, we are going to dynamically determine what our entity "thinks" and "does" by glueing together little bricks, as to form an AI state on-the-fly that reacts to the context the entity is in at this precise moment.

To put it another way: we'll still have entity states that execute actions like toggling a component or setting a variable somewhere, but this state logic will be decoupled from the global **decision logic** of the AI's brain.

Alright so – the inner nodes control the flow of the AI, and the leaves ultimately implement its actions. This all sounds very nice, but: how do we really tell the code that this node is a task node, and this one is an inner node that should impact the flow of the logic?

Well, this is done thanks to the node's evaluation logic...

Evaluating the node

When you code up a behaviour tree, you'll want to give each node its own Evaluate() method to determine its logic. This method may sometimes call the Evaluate() logic of its children (depending on the type of node) to propagate the evaluation

further down the tree, and it should always update and return the **state** of the node, among the three possible ones:

Running: The node is still executing its logic.

Success: The node has successfully finished running the logic.

Failure: There was an error somewhere and the node couldn't complete its execution properly, or it was checking for something and the conditional resulted in a false boolean.

To run our entity's behaviour and essentially have it "live its life", we will thus run the Evaluate() function of its root in an infinite loop (using Unity's built-in Update() entry point) to perform what is called a of the tree, and the root will in turn propagate this evaluation call to all the relevant sub-branches.

IMPORTANT NOTE ABOUT PERFORMANCE

> The "tick" principle I'm presenting here is the easiest way to understand the propagation of the logic flow through the entire structure, and its ability to react to changes in the environment.

> However, in a real video game, having each behaviour tree on each AI in the scene tick every frame would of course require an immense computing power, and demand a lot of resources. That's why, usually, **optimised** BT implementation rely on events and signals to update their logic flow – this way, you can keep the flow fixed a certain way most of the frames, and just re-check the conditions and possible new branching when need be.

> In this book, I'm not going to take the optimisation route, because I'm aiming for readability and clarity first and foremost. So I won't discuss how to pinpoint the change times, and setup this kind of efficient structures – we'll stick with the "tick every frame" method. But still, bare in mind that this performance is a big deal in the context of a real-time interactive media like a video game, so if you're serious about AI, you'll definitely need to look into these optimisations one day… :)

Something crucial to keep in mind is that the tree evaluation depends completely on its structure because, in addition to the inner nodes you pick and that determine the logic flow, the order of the nodes and sub-branches of the tree also defines the action's Basically, in a classical BT implementation, the child nodes are always tried in the same order, from "left to right" (i.e., in code, from first to last in the list). And because these nodes may return a failure, or a running state, they can potentially interrupt the evaluation of this sub-branch of the tree. Thus, when building your tree, you have to carefully

choose the order of the child nodes and sub-branches in order to properly “rank” and prioritise your actions.

But so, this Evaluate() logic is what truly differentiates nodes. Because, of course, although all node evaluation methods define and return the node’s state, the code that’s run inside the function will be unique in each node type – this is what actually specialises the node and allows us to implement the brains of our entity.

Coming back to the distinction we discussed before about inner nodes and leaves, we can classify these evaluation logics into two grand “families”: the “flow-control” nodes, and the “action” nodes.

Flow-control nodes

Flow-control nodes are the backbone of your behaviour tree: they specify how you'll go from one branch to another, and how you should evaluate the nodes inside of a branch. Their impact therefore varies, depending on where they are in the tree, but each flow-control node has a fixed logic and can be re-used from one behaviour tree to another as needed.

We usually distinguish between two types of flow-control nodes:

Composites have one or more children and process them in a pre-determined or random order. Their current state depends on the states of their children (and they are "running" while they are still processing the children).

Decorators have exactly one child node. They are an intermediary layer that may transform the resulting state returned by the child, repeat or delay the processing of the child, etc..

There are obviously many possible composites and decorators; and you could invent pretty crazy ones yourself! But it's quite common to have at least the few following flow-control nodes in your behaviour tree toolbox:

The Sequence is a composite that returns a "success" only if all its children succeeded. It is comparable to a logical AND, and makes it easy to chain a series of action or have an action run only if a particular check passes.

The Selector is a composite that returns a "success" as soon as at least one of its children has succeeded. It is comparable to a logical OR, and typically allows us to implement a choice between two sub-branches.

The Inverter is a decorator that flips around the result of its child, so a success will become a failure and vice-versa – it is

comparable to a logical NOT. This node is pretty useful to re-use checks: rather than defining two nodes for checking that "my enemy is closer than X metres" and "my enemy is further than X metres", you may only define the first node and then apply an inverter to directly get the second behaviour.

The Parallel is a composite that runs all of its children at the same time and doesn't interrupt its execution depending on their return. You can choose various success policies for this node (for example: either one of the children succeeded, or you need all of them to succeeded, or you automatically succeed once they've all finished running...). A parallel node is interesting when you want several sub-branches of similar priorities, and not exclusive to each other – it derives from the concurrency idea we explored in *Chapter 5* with the concurrent finite state machine architecture.

The Timer is a decorator that repeats the processing of its child every X seconds, which is super when you have a looping action, or you want to delay some action for your entity.

In the rest of this book, I will represent these nodes as follows:

Figure 6.2 – Schematic representation of the five common flow-nodes (from left to right: the Selector, the Sequence, the Parallel, the Inverter and the Timer)

USING A RANDOM ORDER?

> In some cases, you might want to add a bit of stochasticity to your AI by having some or all composite nodes pick the evaluation order of their child nodes randomly. For example, a Selector could totally run one of its children first one time, and then last later on, to simulate a different priority for these actions.

> However, this implies that, as a developer, you lose a lot of control on what's happening, and you have to deal with a lot more of edge cases since some nodes may not have run before to prepare the right context for the tree. So you should probably not venture into the stochastic world for your first behaviour tree!

> Moreover, because a behaviour tree is run continuously at each tick, randomness isn't trivial to integrate. If we take the naive approach and put a random ordering on the children at each

> execution, then we'll face the problem of the oscillating AI we mentioned in *Chapter 5: Upgrading our finite state machines* – our entity will keep changing its mind and switching from one action to the other. So, in order to implement stochastic flows in a behaviour tree, you would need to separate the decision phase from the actual execution phase, which may require quite a lot of work.

Action nodes

Action nodes aren't as clear-cut as flow-control nodes, because they are highly dependent on the behaviour you are trying to implement. Basically, these leaves in the tree will be the final "immediate state" the flow guides us to for the current frame.

They can do very diverse things but, here, we'll consider that we have either checks or tasks:

A check node will simply perform a boolean check on something and return "success" or "failure" based on the result of this conditional.

A task node will make the entity perform some actual action (like moving or attacking).

In the rest of this book, I will represent check and task nodes as follows:

Figure 6.3 – Schematic representation of a check node (left) and a task node (right)

For example, to follow a given target, we could setup a sub-branch in our behaviour tree with a Sequence composite at the top, and then a check node first on the left to check for a target, and a task node on the right to actually follow it if it exists. This way, if the check node fails, then the Sequence stops immediately and we ignore the follow task node; but if the check passes, the Sequence goes on to evaluate its next child and runs the move action.

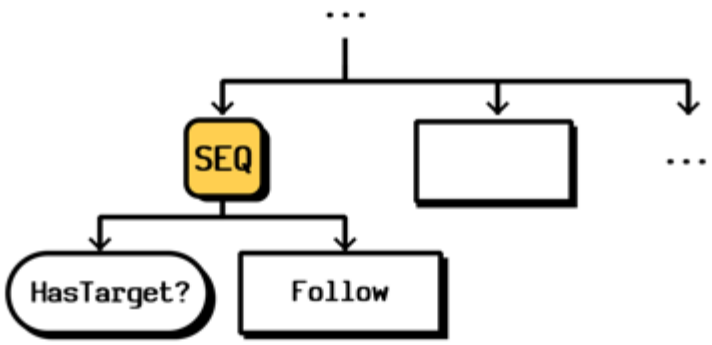


Figure 6.4 – Basic behaviour tree sub-branch structure to implement a "follow the current target" logic

At this point, however, you might be wondering: how exactly could our follow task node know about the target we identified in the check node before? Clearly, although *we* are aware that this is supposed to model just one big artificial brain, the scripts have no way of knowing each other's contents, right?

Or... do they?

Leveraging the power of data

A key idea to understand when working with behaviour trees is that they take advantage of a famous design philosophy called the **data-driven**

What is data-driven design?

In a nutshell, the idea behind this concept is to implement a fairly fixed logic, but then let the input data further specialise the code behaviour and output context-specific results. The exact process is therefore "driven" by the current data, and the same script can act quite differently from one execution to the other, if the inputs have changed.

An extreme level of data-driven programming could be an opcode interpreter, for example. This type of program is

basically able to convert specific tokens into tasks (for example "10" means: "add the two next numbers"), but apart from that it doesn't contain any real instruction – except "start reading the data". So the program simply parses the tokens in the input one by one and converts this data into real actions.

CURIOUS ABOUT THIS EXAMPLE?

> Although it's a bit out of scope here, if you want to learn more about what opcode interpreters are and how they work, I'd recommend you check the 2019 Advent of Code challenges that build upon this idea and have you gradually code up an intcode program: https://adventofcode.com/2019.

What's interesting with this philosophy is that because no process is "hardcoded", you can very easily switch out your logic for another. There are of course some elementary actions you'll need to setup in advance, but then you should be able to assemble these bricks into more complex systems on-the-fly, without having to recompile anything.

Using a data-driven philosophy for behaviour trees

With behaviour trees, we can benefit from this flexibility for our nodes: ideally, we'd like for the evaluation logic of our nodes to be quite generalist, and then depending on where the node is in the tree and what the context around the entity is currently, have it adapt properly and respond to this specific situation.

An important part of behaviour tree implementation is therefore the creation of a **data context** for our nodes where they can store and retrieve data across the entire tree, or a sub-tree (for more "localised" data). This data context can be used to pre-compute values in one node and pass it on to another, to check for some condition in the current context of the entity and potentially interrupt the logic flow, to keep some global but temporary info on the unit, etc.

Now, of course, we could say that the data is always global to the entire tree and have just one big source that all the nodes access and update. This would make it easy and quick to play around with our data source... but it also goes against the scalability and atomicity principles that underlies behaviour trees.

This is why, usually, you should rather make the data **local** to a node, so that:

ancestors of this node (i.e. nodes that are higher in the graph, closer to the root than this node) don't have access to the data

descendants of this node (i.e. children but also grand-children, grand-grand-children and so on) have access to the data

In other words, your node can go and check their parent's or grand-parent's data to retrieve a higher-level piece of info if need be; but, at least, you ensure that each subtree stays self-contained and only defines the data it's interested in. This obviously does not exclude the root of the tree from having global data shared throughout the entire structure if it makes sense for your entity's AI, but it doesn't impose any design and therefore maintains a high decoupling in our behaviour trees by default.

Usually, the data context is a sort of **mapping** between a field name and a field value (of various type), just like common C# variables. So, for example, you could have some data storage with a target field that contains a reference to the Transform of an object in the scene, a health field that contains the current amount of health of the entity, or the distance to the closest headquarters building on the map (see *Table*

Table 6.1 – An example of a possible data context with three fields of various types

The point is to have one node set and store these values, and other nodes in the graph access the data and retrieve a specific value to help them run their logic. Of course, the data may also be split between multiple nodes – typically, here, we could have the health field be stored in the tree's root to make it accessible from anywhere in the graph, while the target and distance_to_hq fields are only declared and usable in the sub-branch of the tree dedicated to the entity's movement on the map. Just remember that larger scopes always means more coupling and less atomicity!

In the rest of this book, I'll show the data fields stored on a node in the graph as follows:

Figure 6.5 – Schematic representation of a task node which stores three local data fields field2 and

(Be careful, though: *Figure 6.5* shows what node the data is stored in, but not where these data fields may be retrieved and used elsewhere in the tree...)

I won't go into too much technical details here because we'll explore this further in *Chapter 7: Creating a behaviour tree* and we'll see how to use it on a particular use case in *Chapter 8: Implementing a RTS collector* But keep in mind that, when working with behaviour trees, you want your nodes to have access to a more or less global data source to ensure communication with the rest of the graph while maintaining a high level of decoupling.

GOING EVEN FURTHER WITH DATA-DRIVEN DESIGN

> In this section, I've discussed how behaviour trees draw inspiration from the data-driven philosophy and make data a crucial building block; but we could actually go further and do "true" data-driven design by defining even our nodes logic with data.

> The idea would be to not code up our nodes in C# classes but instead load and interpret data from external files that represent the node's behaviour. I won't dive into this alternative implementation in this book – but if you're curious, the Java Behaviour Trees package (https://sourceforge.net/projects/jbt/) uses a hybrid of data-driven and code-driven implementation.

> In any case, if you want to give it a try and make a data-driven behaviour tree toolbox, feel free to share your creation via email (at mina.pecheux@gmail.com), or even on the Github of this book (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) as an issue, so that other creators can react too!

Why use behaviour trees?

Now that we have an idea of what behaviour trees are, let's talk a bit about why and when it's interesting to use them – and, in particular, how they compare to the finite state machines we studied in *Chapters 3 to*

Admiring the strengths

When we worked on FSMs, we saw that, even if they are quite straight-forward to design and implement, they can quickly lack adaptiveness. As soon as your number of states starts to grow,

handling all the transitions becomes a real head-ache. Plus, despite your best efforts, you'll probably start to couple more and more code chunks together, because state machines can't completely separate the states from each other and still require some global data to be shared between the various scripts.

Although upgrades like concurrent FSMs, hierarchical FSMs or pushdown automata partially mitigate these issues, there is still clearly a limitation to what you can achieve with state machines because of these strong dependencies.

On the other hand, these concepts of **decoupling** and **autonomy** are at the heart of behaviour trees.

Indeed, one of the greatest thing about behaviour tree is that, as you can see on *Figure* trees are inherently **modular** Nodes don't need to know about each other to work: they just have to have access to the global data storage of the tree, and then it's up to the links between the nodes to properly redistribute the flow and run the AI logic. So, apart from the root node or perhaps their parent, that provide them with the right data context, they live in a fairly isolated context.

This modularity also has an amazingly cool consequence: when working with behaviour trees, once the initial base objects have been prepared and you have prepared the core logic blocks common to most of your routines, it is possible to build new

behaviours and entity AIs without having to go and bother a developer... thus allowing designers to, well, design! BTs are therefore extremely valuable in terms of team organisation and project workflow because, contrary to finite state machines that require coders to specify and maintain the logic of each state in scripts, behaviour trees can be assembled from easy-to-use building blocks in a **no-code** fashion.

SEEING MODULARITY IN PRACtICE?

If you start to dive into the world of behaviour trees, and especially as a Unity game developer, you are bound to bump into Opsive's famous BT assets, which are available on the Unity asset store over here: https://assetstore.unity.com/publishers/2308.

Figure 6.6 – Screenshot of the page of Opsive's *Behavior Designer* package in the Unity asset store

Basically, this collection of behaviour tree utilities and pre-made structures allows you to quickly patch together logic chunks to create your own AI in a flash. And if you browse their products, you'll notice that they have indeed "atomised" their modules in several smaller chunks, by taking advantage of the high modularity of behaviour trees.

Because the decision logic isn't chopped down into multiple state classes, like with FSMs, but instead **centralised** at the "brain level", we don't have to worry about synchronisation and exponential growth anymore.

This makes behaviour trees extremely powerful and inherently too, since they're loosely coupled structures that one can grow at will without instantly creating an exponential number of inter-connections. After all, if you think about it, there is no difference between a single child node, and a child node that is actually at the top of a large sub-branch – both are still just children linked to their parent. So we can actually switch one part of the tree for another quite easily and therefore build our logic

This ability to do **iterative development** is a real perk of the BT model, and it's one of the reasons that have made behaviour trees **one of the go-to AI architectures** in modern video games (lots of AAA or smaller titles actually rely on them, for example

Just Cause 3

,

Halo 2

and

Halo 3

,

Spore

...). This widespread usage is also a consequence of another nice advantage of the architecture, which is that is easily to follow and **debug** while it's running: you can, at any point in time, precisely track down the flow of the logic in the tree.

Spotting the weaknesses

With all that being said, behaviour trees aren't a silver-bullet to all AI programming problems. There are of course cases where there might better-suited techniques in your toolbox.

Typically, behaviour trees can be **hard to design** and implement at first. If you aren't too familiar with this system yet, you might struggle to properly reduce your logic to a basic-enough brick, and try to make all-in-one nodes that resemble more an FSM state logic. This will lead to annoying entanglement issues, that shouldn't happen when using behaviour trees.

These complexity issues may result from another common problem with behaviour trees, which is the improper collusion of the decision process and the execution process. Most implementations (and ours as well, actually) don't clearly separated the "making decisions" part from "acting according to those decisions" part, since we have both flow-control nodes

and action nodes. Those BTs therefore somewhat fail at following a usual programming paradigm known as **"separation of** (But again, we can alleviate this to some extent by creating low-level-enough bricks.)

Because of how modular and atomised they are, BTs also require you to prepare quite a lot of objects before anything can actually be run. Those objects will be small and self-contained, and hopefully they shouldn't be too long to code, but you will have a **start-up phase** to get through before your entity interacts with the world and reacts to it. (In comparison, FSMs are usually quicker to setup since, technically, as soon as you've written down the entry hook of one state, you've already coded a part of your AI.)

And, if your different entities happen to function in a similar way, behaviour trees can become a thorn in your side. By blindly focusing on this AI architecture, you might miss the robustness and inheritance of a well-thought hierarchical FSM...

PICKING HFSMS OVER BEHAVIOUR TREES?

> If you want to learn more about how hierarchical FSMs can best behaviour trees for very complex and specific AIs, and in particular how mixing these with a data-driven philosophy can really crank your models up a notch, I highly recommend you watch this talk from the 2019 GDC conference by Adam

> Noonchester who discusses how his team designed and implemented the enemy AI for the Marvel's Spider-Man game: https://www.youtube.com/watch?v=LxWq65CZBU8.

A quick sum-up

To sum up, always keep in mind that there is no perfect one-size-fits-all solution, and that even if behaviour trees have become a standard in the video game industry, they are fallible.

We could summarise the benefits and shortcomings of behaviour trees as follows:

A node (and its children) has access to a data context that allows you to easily share info between each node in a branch, no matter how far it is in the tree.

This data sharing does not require any strong coupling. The data pool is in read-and-write mode for everyone, and it acts as a global source of truth, but nodes don't really need to be aware of each other to work.

You can iterate and gradually extend the behaviour during your development phase. You can start with a simple tree at first,

and then add more and more branches, one by one, to define additional checks or tasks and simulate more complex behaviours. So behaviour trees are scalable and can be built incrementally.

Setting up the behaviours' relative priorities is straight-forward (except maybe if you use randomly-ordered composites): you read the children from left to right and whatever is on the far left will get evaluated first.

You can re-use nodes in other parts of the tree very easily to re-integrate a particular piece of logic somewhere in the flow, or even assemble together pre-made BT node libraries, thanks to a high modularity. This also means that you can use or prepare base libraries for your most common behaviours, and then let the designers on your team compose building blocks into brand new behaviours autonomously, without needing to ask developers to update the scripts of the AI, like with a FSM.

Behaviour trees inherently handle sequences, fallbacks and interruptions. With BTs, you can easily define a series of actions to take one after the other if one or all succeeded so far; you can designate a specific action or check as the fallback if no other matches the current state of the entity; and you can have any task or check be interrupted by an action

with a higher priority for the current frame, thanks to the continuous tick process.

BTs are also extremely designer-friendly for maintenance and debug: you get a streamlined logic from the root to the leaves that can be followed and studied to understand how the AI "thinks", and why it reacts the way it does at this precise moment.

On the other hand, behaviour trees can be hard and long to setup. You need to have a clear vision of your AI logic to properly chop it down in autonomous and elementary bricks, otherwise you run the risk of blending together the decision and the execution logic, and you'll end up with big scripts that are similar to FSM state classes.

Moreover, you may need to prepare several nodes before you can combine them into a usable tree – so beware the initial start-up phase!

Summary

In this chapter, we've talked about the basics of behaviour trees.

We've detailed the fundamentals of this graph structure, from the root to the leaves, and we saw how each node type has a specific evaluation logic that determines its unique role in the tree. We also discussed the difference between flow-control nodes and action nodes, and we listed some common flow-control nodes that can be used in any behaviour tree.

We then focused on the data-driven philosophy, which is a cornerstone of BTs and supports their modular design.

Finally, we briefly went through the main strengths and weaknesses of this AI technique, in particular compared to the finite state machines.

In the next chapter, we will elaborate on these principles and discover how to create our own behaviour tree toolbox by making a Unity C# package that implements abstract base structures, as well as most of the common flow-control composite nodes we mentioned here.

7 - Creating a behaviour tree toolbox

In *Chapter 6: Understanding behaviour* we introduced a new AI programming technique – the behaviour trees. We saw how this graph-based tool is a nice way of creating modular logic blocks that can then be assembled into higher-level scalable structures to implement expert and reactive behaviours for our entities.

But so far, it's been all about theory, and we don't yet have a clear vision of what a BT might look like in C# code. That's why in this chapter, we're going to roll up our sleeves and create our very own behaviour tree toolbox.

Our objective here will be to prepare a simple BehaviorTree C# package that contains generalist BT tools, to then have this little library ready for when we want to actually code the AI of a unit using behaviour trees, and speed up the start-up phase.

FOR THE HANDS-ON LEARNERS: WANNA READ AHEAD?

> The BehaviorTree C# package we'll prepare in this chapter will be at the heart of the AI we will implement in the next chapter, *Chapter*

> *8: Implementing a RTS collector AI*. So if you're more of a hands-on learner and you'd like to see how these abstract classes and generic tools are actually applied in practice, don't hesitate to check out this other chapter in parallel!

Defining the base behaviour tree objects

To kick things off, let's focus on the core elements that make up a behaviour tree: the nodes, and the edges between them.

A peek at our C# library objects

We know that, overall, all nodes basically function the same: they have a parent, zero or more children, a state, an evaluation logic and an access to the tree's global data storage. This means that we can prepare a C# Node class that implements a base skeleton and then let developers extend it depending on their specific needs.

Also, similar to the FSM base tools we worked on in *Chapter 4: Making a simple guard* our Node class shouldn't be used directly by the users of our BehaviorTree library. What we want is for them to implement their own Evaluate() behaviour and adapt the node's logic – so, once again, we'll rely on an abstract C# class here.

For the edges, it's a little bit different. Since we already said that we would store the parent and children in the nodes, we don't really need to create any object for these connections – the only thing we need is a reference to the behaviour tree itself somewhere so that we can define the nodes we want to put in it. So our BehaviorTree package will also contain a BTree base class that represents the tree as a whole and handles the tick process to get a continuous evaluation of our AI logic.

THE RETURN OF C# NAMESPACES

> Again drawing inspiration from our previous finite state machine-related base tools (the ones we setup in *Chapter 4*), we'll make sure to wrap our various behaviour tree toolbox objects in a BehaviorTree C# namespace to properly isolate and package this code as an easy-to-import library.

Implementing our abstract Node class

Alright – the first thing we'll work on is the base element of our tree: the node.

We'll write this class in a Node.cs file and, at the moment, we can simply create the class wrapped in our BehaviorTree namespace, like this:

Node.cs

```
1 namespace BehaviorTree {
2     public abstract class Node {}
3 }
```

Setting up global parameters

Let's start easy and take care of our node's states. We know that we have three possible values for this state: "running", "success" or "failure". In C#, a neat tool for representing a fixed set of values is the so we'll add a new NodeState enum to our namespace:

Node.cs

```
1 namespace BehaviorTree {
```

```
2     public enum NodeState { RUNNING, SUCCESS, FAILURE }
3
4     public abstract class Node {}
5 }
```

And then, all we have to do is create a _state field in our Node class that uses this NodeState type, and we'll be able to store one of these three values as the current state of a node instance.

Because we only want Node objects to be able to modify this value, we'll make it protected – but we'll also prepare a public **read-only getter** so that other objects in our codebase can check the current state of a node:

Node.cs

```
1  namespace BehaviorTree {
2     public enum NodeState { RUNNING, SUCCESS, FAILURE }
3
4     public abstract class Node {
5        protected NodeState _state;
6        public NodeState State => _state;
```

```
7     }
8 }
```

Another cool high-level property for our nodes could be a unique identifier, to easily track down (and potentially debug or even display) our nodes. For this, we can stick with the common programming pattern of using a static variable to auto-increment a global ID counter, and auto-assign the new ID to our Node instance when it is first created:

Node.cs

```
1  namespace BehaviorTree {
2     public enum NodeState { RUNNING, SUCCESS, FAILURE }
3
4     public abstract class Node {
5        protected NodeState _state;
6        public NodeState State => _state;
7
8        public static uint LAST_ID = 0;
9        private uint _id;
10        public uint Id => _id;
11
```

```
12        public Node() { _id = LAST_ID++; }
13    }
14 }
```

Like before, I've made sure to keep this data as **encapsulated** as possible by making my _id variable private, and by only exposing a getter on it, since only our Node class actually needs to update this value in our program. However, we do need to make the LAST_ID variable public, because we'll want to access it from our BTree class to re-initialise the counter when we create a new tree instance.

So this is pretty cool – with these few additional lines, we have put in place an auto-incremented ID system that will allow us to uniquely identify each of our node instances.

But of course, for now, these nodes can't really do anything! We don't have any relationships, so this node is in completely isolation and can't integrate into a tree's logic flow, and we don't have an Evaluate() method to run our specific logic.

Executing the node's logic

The logic evaluation part is actually quite quick to setup. Since this Node class is abstract and doesn't implement any real

behaviour commands or actions, it just has to define the Evaluate() function as an **abstract** to force derived classes to implement it their own way.

So let's update our code as follows:

Node.cs

```
1  namespace BehaviorTree {
2      public enum NodeState { RUNNING, SUCCESS, FAILURE }
3
4      public abstract class Node {
5          protected NodeState _state;
6          public NodeState State => _state;
7
8          public static uint LAST_ID = 0;
9          private uint _id;
10         public uint Id => _id;
11
12         public Node() { _id = LAST_ID++; }
13
14         public abstract NodeState Evaluate();
15     }
16 }
```

And that's it! From that point on, any class that derives from Node (i.e. any custom node type defined by a developer using our BT toolbox) will necessarily have to implement an overridden Evaluate() method that runs logic specific to this node instance, and that sets and returns the instance's new state.

Storing the node's relationships

The next step is to define the edges between our nodes. We've said it before: we will do it by creating references to parent and child nodes directly in our Node class.

In short, we'll simply create a _parent variable of Node type to keep the reference to the node above us in the graph, and a _children variable which is a list of Node instances to also remember all the ones below us. Don't forget that we want to maintain some prioritisation between our nodes (as discussed in *Chapter* so it's better to use an ordered C# list than a dictionary or a hash map for example, because it will allow us to sort our child nodes and keep these priorities.

As with our previous properties, we'll try to limit as much as possible the reach of our variables by using private members,

and then some public getters:

Node.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public enum NodeState { RUNNING, SUCCESS, FAILURE }
5
6    public abstract class Node {
7      protected NodeState _state;
8      public NodeState State => _state;
9
10     public static uint LAST_ID = 0;
11     private uint _id;
12     public uint Id => _id;
13
14     private Node _parent;
15     public Node Parent => _parent;
16
17     private List<Node> _children;
18     public List<Node> Children => _children;
19     public bool HasChildren => _children.Count > 0;
20
21     public Node() {
```

```
22            _id = LAST_ID++;
23            _parent = null;
24            _children = new List<Node>();
25        }
26
27        public abstract NodeState Evaluate();
28    }
29 }
```

(Also, in order to use the C# generic List container type, we have to make sure that we import the System.Collections.Generic package at the top of our script.)

At the moment, our _parent and _children properties are initialised with some null or empty data in the node instance's constructor. To actually update and set these properties as we build our tree, let's add some methods to our Node class to set all of its child nodes, or to attach or detach just one child node at a time:

Node.cs

```
1 using System.Collections.Generic;
```

```
2
3  namespace BehaviorTree {
4    public enum NodeState { RUNNING, SUCCESS, FAILURE }
5
6    public abstract class Node {
7      // ...
8
9      public Node() { … }
10
11     public abstract NodeState Evaluate();
12
13     public void SetChildren(List<Node> children) {
14       foreach (Node c in children) Attach(c);
15     }
16
17     public void Attach(Node child) {
18       _children.Add(child);
19       child._parent = this;
20     }
21     public void Detach(Node child) {
22       _children.Remove(child);
23       child._parent = null;
24     }
25   }
26 }
```

We can of course also use this new SetChildren() function to create a second constructor for our class that takes in a list of child nodes. This way, we'll be able to directly pass in children for a node upon creation if we want:

Node.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public enum NodeState { RUNNING, SUCCESS, FAILURE }
5
6    public abstract class Node {
7      // ...
8
9      public Node() { … }
10     public Node(List<Node> children) : this() {
11       SetChildren(children);
12     }
13
14     public abstract NodeState Evaluate();
15
16     public void SetChildren(List<Node> children) { … }
17     public void Attach(Node child) { … }
```

```
18        public void Detach(Node child) { … }
19    }
20 }
```

At this point, we have all we need to traverse our tree either from the root to the leaves, or from the leaves to the root. Our relationships system works both ways and it will make it easy to move around in our tree to get or set data in various interesting places, and have multiple nodes combine their logic into one chunk of behaviour.

REDUCING THE MEMORY FOOTPRINT

> It is worth noting that we could technically store our nodes' relationships just in one direction – typically, if for some reason you're limited in terms of memory usage, you should probably just store the children of your nodes, and then recompute back parents in some other way. Here, we're taking the easy road and taking advantage of the still fairly low memory requirements of the double-way system to make our lives easier.

> But, as is often the case in tech, it's about doing the right trade-off between computing power and memory usage!

One last improvement we can make, however, is to also keep a reference to the tree's root node itself. This can be very useful if you want to keep your tree's global data source in this specific node, to get a quick access. Of course, doing this slightly hinders modularity, since your different subtrees all depend on this root node to function properly and they aren't as autonomous as they could be... but in lots of cases it's also way easier to deal with during development and it doesn't matter too much. (Most notably, because you can do a first dev iteration with your data stored in the root like this, and then refactor your tree to properly "localise" the data back down in the right subtree, and therefore restore the modularity.)

So to wrap up our node's relationships implementation, we're going to add another reference to a Node instance called and update our constructor and our various Node methods to properly set this reference.

Specifically, we're going to assume that, by default, the node uses itself as root (as if any new node was at the top of its own behaviour tree); and we're going to allow our nodes to re-assign their _root field either automatically when we add them as child nodes, or manually if we force some node to become the root for a pack of Node instances.

This all boils down to the following C# code:

Node.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public enum NodeState { RUNNING, SUCCESS, FAILURE }
5
6    public abstract class Node {
7      // ...
8      private Node _root;
9      public Node Root => _root;
10
11     public Node() {
12       _id = LAST_ID++;
13       _parent = null;
14       _children = new List<Node>();
15       _root = this;
16     }
17     public Node(List<Node> children) : this() { SetChildren(children); }
18
19     public abstract NodeState Evaluate();
20
21     public void SetChildren(List<Node> children, bool setRoot = false) {
```

```
22            foreach (Node c in children) Attach(c);
23            if (setRoot) SetRoot(this);
24        }
25        public void Attach(Node child) {
26            _children.Add(child);
27            child._parent = this;
28            child._root = _root;
29        }
30        public void Detach(Node child) {
31            _children.Remove(child);
32            child._parent = null;
33            child._root = null;
34        }
35
36        public void SetRoot(Node root) {
37            _root = root;
38            foreach (Node child in _children) child.SetRoot(root);
39        }
40    }
41 }
```

Our nodes now have a clear context to work in with edges to the nodes above and below them, and even a quick reference to the tree's root if need be.

The last thing we need to do is give our nodes some data context so that they can store and retrieve various values, and share information across the tree.

Taking care of the data

We've already seen in the previous chapter that we want our data to be scoped to retain the scalability and atomicity of our behaviour trees. So, we know that we want the data to be declared at the node level (in order to restrain it to a sub-branch of the tree, when possible), which means that we *do* need to handle it in our Node class. We also said that the data fields could be of various types (such as a Unity an int or a

From a technical point of view, the question is therefore: how can we store any kind of data in our nodes, and get or set them easily from various access points?

A nice solution here is to rely on a C# and on C#'s object "lazy" type.

C# Dictionaries (available in the System.Collections.Generic built-in C# package) are a handy way of mapping a key to a value with unique pairing and instantaneous access. In our case, it makes it easy to create a sort of custom variable

system where we define some strings for our different data fields, and we then store the matching value of each field.

The object type, as we've already discussed in *Chapter 3: What are* is the base type for any C# value and can therefore be used to store data in an agnostic way. Typically, here, it's a neat trick to packaging together values of different types in the same

Combining Dictionaries with the object type will thus allow us to keep a mapping of all the relevant data fields for our behaviour tree, no matter the type of value they use, in a centralised and easy-to-use hashed structure.

IMPORTANT NOTE: PERFORMANCE GOTCHAS

> Although they are extremely useful from a developer's perspective to abstract away type differences and mix together multiple value types, it is essential to remember that boxing and unboxing (i.e. the conversion to and from the object type) are computationally expensive processes. Ideally, you should therefore avoid relying on the object type and the boxing/unboxing conversion trick in the critical path of your project, because it can severely reduce the performance of your app.

> Here, I'm focusing more on clarity and readability than on performance, so I will be using the object type for my behaviour tree's data storage despite these performance issues – but in a real project, you might want to optimise this by separating the value stores by type and avoiding these conversions from and to the object type.

With this mind, let's setup our node's data context by defining a new _data field in our class, which is a C# Dictionary that maps a string to an object value:

Node.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public enum NodeState { RUNNING, SUCCESS, FAILURE }
5
6    public abstract class Node {
7      // ...
8      private Dictionary<string, object> _data =
9        new Dictionary<string, object>();
10
11     public Node() { ... }
```

```
12        public Node(List<Node> children) : this() { ... }
13        public abstract NodeState Evaluate();
14        public void SetChildren(
15              List<Node> children, bool setRoot = false) { ... }
16        public void Attach(Node child) { ... }
17        public void Detach(Node child) { ... }
18        public void SetRoot(Node root) { ... }
19    }
20 }
```

Note that this time, because our _data property is of Dictionary type, we can't easily expose a read-only getter. Indeed, returning a reference to our _data field actually means passing on this mutable container, which means that anyone could potentially update its contents. That's why here I'm only defining the private property, and we're going to use public methods to read or write custom data fields in this

More precisely, we're going to define three functions, SetData() and that allow us to retrieve, modify or erase a data field in the _data Dictionary. However, the GetData() and ClearData() methods will be slightly more evolved than just a get/erase, and they'll actually work **recursively** up to the root if need be to find the requested key.

This way, we'll be able to call GetData() or ClearData() on our current node without worrying about its exact position in the tree and, if the data field we ask for is not present in its own context, the node will automatically pass the request on to its ancestors to try and retrieve the value. Note that, obviously, keeping the data close or directly referencing the node with the data is cheaper, so we should do that if possible – but this recursive logic will let us handle more "black-box" cases and provide a more flexible architecture for our behaviour trees.

Here is the code for these three new methods of our Node class:

Node.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public enum NodeState { RUNNING, SUCCESS, FAILURE }
5
6    public abstract class Node {
7      // ...
8
9      public object GetData(string key) {
10       object val;
```

```
11            if (_data.TryGetValue(key, out val)) return val;
12            if (_parent != null) val = _parent.GetData(key);
13            return val;
14        }
15
16        public void SetData(string key, object value) {
17            _data[key] = value;
18        }
19
20        public bool ClearData(string key) {
21            bool cleared = false;
22            if (_data.ContainsKey(key)) {
23                _data.Remove(key);
24                return true;
25            }
26            if (_parent != null) cleared = _parent.ClearData(key);
27            return cleared;
28        }
29    }
30 }
```

Our Node class is now ready, and any developer can import our BehaviorTree package and extend this class to create their own node type. We'll actually see in the next section how to

create new nodes to implement common flow-control nodes such as a Sequence or a Selector.

But before diving into these examples, we're first going to implement the second core object in our behaviour tree toolbox: the BTree class.

Creating the BTree class

The other key element in our BehaviorTree package will be the BTree class.

WHY "BTREE" AND NOT "TREE"?

> Because Unity's library already contains a Tree object, defining our own Tree class and then importing it alongside the usual using UnityEngine; statement could quickly turn into a mess for developers. Basically, they'd need to always prefix the Tree object with the package name to be sure they are accessing the right type, which can be a bit of a pain.

> To facilitate the usage of our BehaviorTree package, it's better to name the class BTree, and avoid any possible confusion in the code!

Again, this will be an abstract C# class that developers must inherit from to define their own custom logic; except, this time, the BTree class will represent the behaviour tree as a whole. In particular, this is where we'll expose a hook to let the coders actually define the structure of their tree, meaning the nodes and edges they want for their AI.

Truth be told, this class is super simple compared to the Node we just made. Here, we only need four things:

We have to inherit from Unity's built-in MonoBehaviour C# class to have access to the usual **lifecycle** such as Awake() and

We must keep a reference to the tree's root node, and that's it ('cause this node will be able to interact with its children, and therefore the rest of the tree recursively, on its own – so we don't need to store any other reference).

We can use our Update() function to run the root's Evaluate() function and trigger the tree's tick process each frame.

Finally, we need to define an abstract function to specify the tree's nodes and edges (the logic of which will be provided by the developer in their own class), that is called in the Awake() to initialise all the required data for this behaviour tree.

The entire BTree class therefore looks like this:

BTree.cs

```
1  using UnityEngine;
2
3  namespace BehaviorTree {
4    public abstract class BTree : MonoBehaviour {
```

```
5       protected Node _root = null;
6
7       protected virtual void Awake() {
8           Node.LAST_ID = 0;
9           _root = SetupTree();
10      }
11
12      private void Update() {
13          if (_root != null)
14              _root.Evaluate();
15      }
16
17      public Node Root => _root;
18      protected abstract Node SetupTree();
19  }
20 }
```

We have finished implementing the two core objects of our BehaviorTree C# package: we or any other AI developer could now import it and create custom node types based on the abstract Node class, before including these nodes into an instance of a class thanks to a specific SetupTree() function.

(Again, if you're curious about how all of this works in practice, feel free to jump ahead to *Chapter 8* and check out how we can apply this BT toolbox to create a RTS collector AI.)

Before we end this chapter, though, let's take a bit of time to setup a few commonly used flow-control nodes and provide our package users with some ready-made tools.

Preparing some flow-control nodes

To finalise our BehaviorTree package, we are going to see how to use our abstract Node class to program several commonly used flow-control nodes.

As we said in *Chapter* these nodes just re-orient the logic flow inside the tree during the tick process, and they always work the same. Contrary to action nodes, they don't depend on the current AI we are programming – rather, they are agnostic nodes with a fixed logic, and it's just their position in the tree that changes their impact on the entity's behaviour. So we can prepare them in advance in our behaviour tree toolbox, and then the AI developers that use our package will be able to pick and place these inner nodes at will.

So, time to expand our BehaviorTree package with some handy utilities!

Setting up composite nodes

To begin with, we're going to see how to setup a few classical composite nodes. Implementing these nodes will show us the steps to creating a custom node type based on our Node class, and it will also populate our BehaviorTree with useful flow bricks to combine sub-branches.

In the following subsections, we'll study the Sequence, the Selector and the Parallel.

The Sequence node

Because this is the first time we get down to the implementation of a real node, we're going to go through the process step-by-step and detail the full train of thoughts.

1. First of all, our Sequence node class will of course inherit from the Node class. We can therefore setup a new Sequence.cs file in our library initialised with the following content:

Sequence.cs

```
1 namespace BehaviorTree {
2     public class Sequence : Node {}
3 }
```

2. Then, like its base class, the Sequence class will define both a default empty constructor and another one that accepts a list of child nodes. Those constructors will directly call the one from the parent class with the C# base keyword:

Sequence.cs

```
1 using System.Collections.Generic;
2
3 namespace BehaviorTree {
4     public class Sequence : Node {
5         public Sequence() : base() { }
6         public Sequence(List<Node> children) : base(children) { }
7     }
8 }
```

3. Finally, we'll simply override its Evaluate() method to implement the right logic.

In our case, we saw in the last chapter that the Sequence is like an "AND" operator for the tree: its goal is to evaluate each child node "from left to right" until either one fails (which interrupts the evaluation process) or we reach the end of the child nodes list.

The final aggregated state of the Sequence node usually depends on whether there are still some children in the "running" state (in which case we'll say that the parent Sequence is "running" too), but you could also consider that your flow nodes can only be in the "success" or "failure" state. This is a design implementation detail that might vary from one lib to the other.

Anyway, here, our evaluation logic for the Sequence class will be as follows:

Sequence.cs

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

```
3  namespace BehaviorTree {
4      public class Sequence : Node {
5          public Sequence() : base() { }
6          public Sequence(List<Node> children) : base(children) { }
7
8          public override NodeState Evaluate() {
9              bool anyChildIsRunning = false;
10             foreach (Node node in Children) {
11                 switch (node.Evaluate()) {
12                     case NodeState.FAILURE:
13                         _state = NodeState.FAILURE;
14                         return _state;
15                     case NodeState.SUCCESS:
16                         continue;
17                     case NodeState.RUNNING:
18                         anyChildIsRunning = true;
19                         continue;
20                     default:
21                         _state = NodeState.SUCCESS;
22                         return _state;
23                 }
24             }
25             _state = anyChildIsRunning ?
26                 NodeState.RUNNING : NodeState.SUCCESS;
27             return _state;
28         }
29     }
```

This code snippet is quite straight-forward: we simply override the Evaluate() function defined in our base Node class, and then we iterate through each child node and evaluate it. Then, depending on the state of each child node, we either keep going or stop the evaluation process, and compute our own local node state.

And here we are! We've successfully implemented our very first custom node based on our abstract Node class, and we now have an easy-to-use "AND" inner node that can be injected into any behaviour tree in the future to compose and evaluate a series of child nodes.

The Selector node

Similarly, we can also make a Selector composite node to have an "OR" logic in a BT. The code is almost the same as for the Sequence, it's just that we have to interrupt the node's evaluation when we get a "success" or "running" state from a child, instead of stopping for a "failure".

So here's the code of our Selector class:

Selector.cs

```
using System.Collections.Generic;

namespace BehaviorTree {
    public class Selector : Node {
        public Selector() : base() { }
        public Selector(List<Node> children) : base(children) { }

        public override NodeState Evaluate() {
            foreach (Node node in Children) {
                switch (node.Evaluate()) {
                    case NodeState.FAILURE:
                        continue;
                    case NodeState.SUCCESS:
                        _state = NodeState.SUCCESS;
                        return _state;
                    case NodeState.RUNNING:
                        _state = NodeState.RUNNING;
                        return _state;
                    default:
                    continue;
                }
            }
            _state = NodeState.FAILURE;
```

```
24            return _state;
25        }
26    }
27 }
```

The Parallel node

A third common composite node we talked about in *Chapter 6* was the Parallel node, which can run all of its child nodes "at the same time". We're not actually going to implement parallelism here, we're just going to say that a Parallel node doesn't have any evaluation interruption condition.

The only tricky part will be too choose a rational process for determining the node's final state depending on the children's result – here my policy will be that:

If all child nodes failed, the Parallel returns a "failure" too.

If any child node is still "running", the Parallel returns a "running" state too.

Else, the Parallel returns a "success".

This whole logic can be coded like this:

Selector.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public class Parallel : Node {
5      public Parallel() : base() { }
6      public Parallel(List<Node> children) : base(children) { }
7
8      public override NodeState Evaluate() {
9        bool anyChildIsRunning = false;
10       int nFailedChildren = 0;
11       foreach (Node node in Children) {
12         switch (node.Evaluate()) {
13           case NodeState.FAILURE:
14             nFailedChildren++;
15             continue;
16           case NodeState.SUCCESS:
17             continue;
18           case NodeState.RUNNING:
19             anyChildIsRunning = true;
20             continue;
21           default:
```

```
22                  _state = NodeState.SUCCESS;
23                  return _state;
24              }
25          }
26          if (nFailedChildren == Children.Count)
27              _state = NodeState.FAILURE;
28          else
29              _state = anyChildIsRunning ?
30                  NodeState.RUNNING : NodeState.SUCCESS;
31          return _state;
32      }
33  }
34 }
```

Coding decorator nodes

Now that we have some common composite nodes to combine and evaluate our child nodes with “AND”, “OR” or parallel logics, let’s shift our focus slightly and talk about decorators.

As we’ve said before, the main difference between composite and decorator nodes is that while composites can have one or more child nodes, decorators have only one (and exactly one)

child node. They usually act as an intermediary layer to post-process or delay the logic in the sub-branch.

This means that, in the Evaluate() method of a decorator node, we'll need to check if the node has at least one child, and then we'll just execute the logic of the first in the list, no matter how many children the node has. Let's see how this applies for two common decorators: the Inverter and the Timer.

The Inverter node

The Inverter node is used to switch the result of its sub-branch and return the reversed state (like a "NOT" logic gate). It transforms a failed state into a success, and vice-versa. To do this, we can use the following code:

Inverter.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public class Inverter : Node {
5      public Inverter() : base() { }
6      public Inverter(List<Node> children) : base(children) { }
```

```
 7
 8        public override NodeState Evaluate() {
 9            if (!HasChildren) return NodeState.FAILURE;
10             switch (Children[0].Evaluate()) {
11                 case NodeState.FAILURE:
12                     _state = NodeState.SUCCESS;
13                     return _state;
14                 case NodeState.SUCCESS:
15                     _state = NodeState.FAILURE;
16                     return _state;
17                 case NodeState.RUNNING:
18                     _state = NodeState.RUNNING;
19                     return _state;
20                 default:
21                     _state = NodeState.FAILURE;
22                     return _state;
23             }
24         }
25     }
26 }
```

The Timer node

Finally, the Timer is a common decorator that lets us wait for a given amount of time before evaluating a sub-part of our

tree, and/or repeat this chunk of logic multiple times – depending on how the Timer is implemented exactly. Here, we'll go for the second version and make a Timer node that loops endlessly. So our goal will be to re-evaluate the child node of our decorator over and over again (and we'll wait for the given delay between each repetition).

The Timer is a nice example of a node that requires some metadata to work, since we have to give it the delay to wait for before (re-)evaluating its child. In other words, contrary to our previous node types, our Timer class will need to receive some extra data in its constructor – more precisely, the _delay variable that represents the wait between each evaluation of the child node:

Timer.cs

```
1  using System.Collections.Generic;
2
3  namespace BehaviorTree {
4    public class Timer : Node {
5      private float _delay;
6
7      public Timer(float delay) : base() {
8        _delay = delay;
```

```
9         }
10        public Timer(float delay, List<Node> children) : base(children) {
11            _delay = delay;
12        }
13    }
14 }
```

Then, to actually run the node's timer and have it count down the right time delay, we'll use Unity's Time.deltaTime built-in to update a _time variable in the Evaluate() function. This variable will be initialised with the length of our cycle and then reduce to zero as time goes by. When it reaches this threshold, we'll evaluate the child node, return a success and reset our _time variable for the next cycle.

Timer.cs

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  namespace BehaviorTree {
5      public class Timer : Node {
6          private float _delay;
7          private float _time;
```

```
 8
 9      public Timer(float delay) : base() {
10          _delay = delay;
11          _time = _delay;
12      }
13      public Timer(float delay, List<Node> children : base(children) {
14          _delay = delay;
15          _time = _delay;
16      }
17
18      public override NodeState Evaluate() {
19          if (!HasChildren) return NodeState.FAILURE;
20          if (_time <= 0) {
21              _time = _delay;
22              _state = Children[0].Evaluate();
23              _state = NodeState.SUCCESS;
24          } else {
25              _time -= Time.deltaTime;
26              _state = NodeState.RUNNING;
27          }
28          return _state;
29      }
30  }
31 }
```

Don't forget that to use we have to import the UnityEngine package.

Another cool feature we can add to this Timer class is a way to warn other systems that a cycle has just finished. For this, we can use **a native C#** or in other words a method with no parameters and no return value, to define a (optional) **callback function** in our node triggered at the end of the countdown:

Timer.cs

```
1  using UnityEngine;
2  using System.Collections.Generic;
3
4  namespace BehaviorTree {
5      public class Timer : Node {
6          private float _delay;
7          private float _time;
8          public System.Action onTickEnded;
9
10         public Timer(float delay, System.Action onTickEnded=null) : base() {
11             _delay = delay;
12             _time = _delay;
13             this.onTickEnded = onTickEnded;
14         }
```

```
15        public Timer(float delay, List<Node> children,
16            System.Action onTickEnded = null) : base(children) {
17            _delay = delay;
18            _time = _delay;
19            this.onTickEnded = onTickEnded;
20        }
21
22        public override NodeState Evaluate() {
23            if (!HasChildren) return NodeState.FAILURE;
24            if (_time <= 0) {
25                _time = _delay;
26                _state = Children[0].Evaluate();
27                if (onTickEnded != null) onTickEnded();
28                _state = NodeState.SUCCESS;
29            } else {
30                _time -= Time.deltaTime;
31                _state = NodeState.RUNNING;
32            }
33            return _state;
34        }
35    }
36 }
```

This way, if we pass in a function to our Timer instance constructor, we'll be able to run some logic from our AI code

when the node finishes a cycle and evaluates its child.

This Timer node, along with the four other flow-control nodes we prepared before and our abstract Node and BTree classes, thus provide developers with some interesting start-up objects to begin making behaviour trees, all neatly organised and packaged in a BehaviorTree namespace.

Summary

In this chapter, we've worked on a little BehaviorTree C# package that implements the core objects of a behaviour tree, and some common flow-control nodes.

We first designed and setup the two main abstract classes an AI programmer needs to create behaviour trees, namely the Node and the

We then saw a few examples of how to create a custom node type based on the Node class by preparing five common composite and decorator nodes to provide our package's users with some ready-made utilities.

At this point, we have all we need to really implement a behaviour tree-based AI entity logic – so, in the next chapter, we'll apply all the theory we've discussed to a specific use

case and see how to use our BehaviorTree library to code our own RTS collector AI!

8 - Implementing a RTS collector AI

In the previous chapters, we discovered the basics of behaviour trees and we implemented a simple C# package to create abstract nodes and trees. Now, it is time to see how they can be used in practice, and how we can take advantage of this toolbox to design and create our own AI logic!

So, to better understand how this behaviour model works, we are going to study a very common AI use case: making the brains of a collector unit AI for a real-time strategy game (RTS).

We will build our logic from scratch gradually, starting with just a basic search-and-walk behaviour so that our units can find resources close to them and move towards them, and then add more and more features as we progress through the chapter.

An overview of the use case

Our goal here will be to have little trucks that spot and navigate to nearby resources, chop down trees or mine rocks

until their storage is full, then go to the nearest storage centre and deliver the goods, before returning to the closest resource location, and doing another round.

Figure 8.1 – Screenshot of the final scene with three collectors moving around to gather resources, and then bring back them back to the nearest storage centre

These collectors will be of two types, depending on the type of resource they gather: blue trucks will get wood from trees, and orange trucks will get minerals from ores.

Also, the resources will deplete after a while — for example, once they are "dead", trees will get another sprite and will be ignored by the collectors. So these collectors will work as long as there are resources of the right type on the map; or else they will simply return to the nearest storage centre and enter the building.

To implement all of this, we will rely on the BehaviorTree C# package we coded in *Chapter 7: Creating a behaviour tree* This will be a good opportunity to see how to use this base library to actually implement a real-life use case, and how the Node and BTree classes we prepared back then can be integrated in a new Unity/C# project.

WANNA CHECK OUT THE CODE AND ASSETS?

> For this RTS collector AI, I'll also use a simple 2D A* pathfinding system to move my units around, which I coded by re-adapting *pixelfac*'s great work available over here on Github: https://github.com/pixelfac/2D-Astar-Pathfinding-in-Unity.

> To get nice visuals, I've downloaded some really nice sprite assets from the famous *Kenney* library (https://kenney.nl/), too. Note that I did add some sprites of my own to have some extra sprites for when the trucks are going up or down, because the initial package only contained a horizontal version.

> As usual, don't forget that you can check the full code and assets for this chapter and its dependencies in the Github repository of this book, over here: https://github.com/MinaPecheux/Ebook-Unity-AIProgramming!

Here, my RTS prototype is a 2D top-down game that relies a lot on tilemaps. This Unity tool makes it easy to create, edit and traverse 2D levels based on tile assets. So the idea is that you have lots of tiny reference objects that you can go and place on a grid to quickly fill your game space:

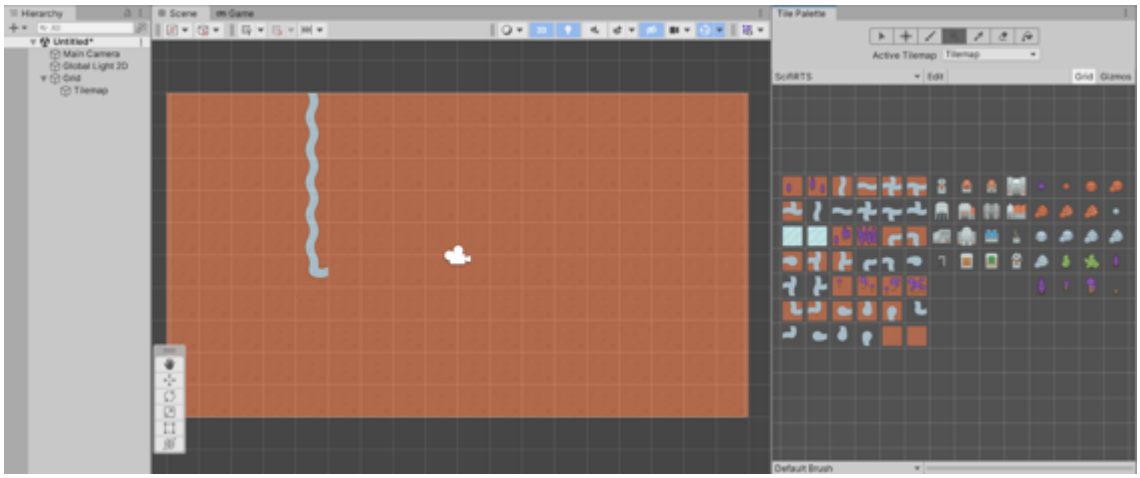


Figure 8.2 – Overview of the Unity editor with the Tile Palette panel open (on the right), and a work-in-progress tilemap level in the scene view (in the middle)

By overlapping multiple tilemaps, you can easily distinguish between different layers and/or contexts for all of those objects. In my case, I'll have three levels of maps:

Ground and obstacles: These tilemaps will be for the unit movement. Basically, the ground map has holes where there are obstacles, and our A* pathfinder will be able to use this data to determine which squares are walkable and which ones are impassable, and devise a clever route from point A to point B on that terrain.

Resources: Then, I'll have one tilemap per resource — so in my case, one for the trees and one for the ores. Each tilemap will be named Resource:resource> so that my collector AI can auto-retrieve the right resource map to work on based on the resource it is supposed to gather.

Buildings: Finally, I'll have a similar setup for the buildings — I'll have tilemaps named Buildings:resource> containing the tiles for the storage locations matching the resource (plus an additional one with some extra buildings in the middle that are just for style and aren't linked to the collector AI system).

Thanks to these references, we will be able to make a global collector AI and, simply by choosing the resource to gather, have the unit auto-pick the right resource spots and storage building tilemaps. So I'll make a new C# script, declare an enum with the possible resource types and and finally expose a variable of this type to make it easy to choose the type of resource this unit will collect:

CollectorAI.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3
4  public class CollectorAI : MonoBehaviour {
5      public enum Resource { Wood, Minerals }
6
7      [SerializeField] private Resource _resource;
8  }
```

This shows up a nice dropdown in the inspector to setup the resource type of the collector:

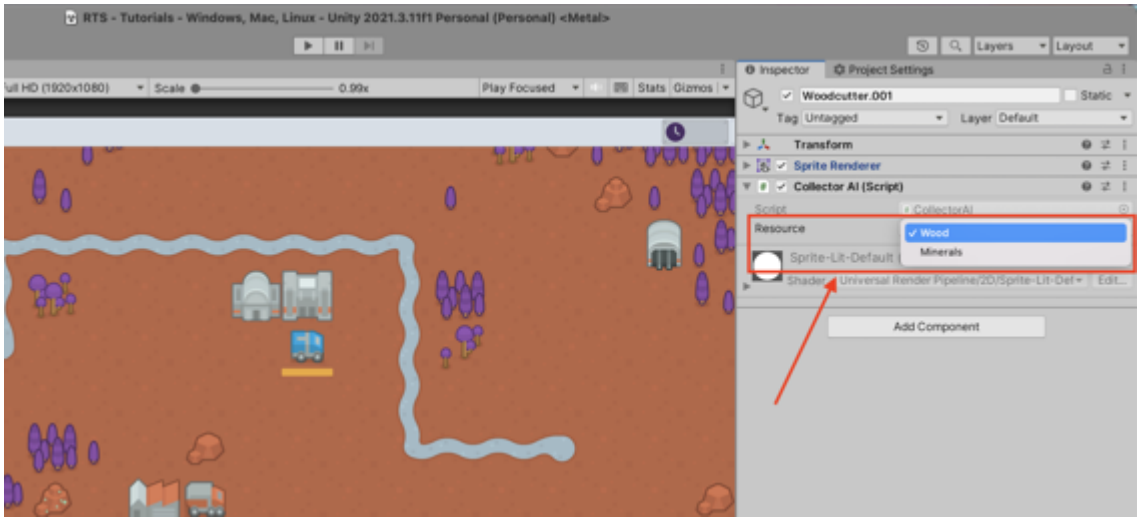


Figure 8.3 – Easy-to-use dropdown input in the inspector panel for our collector custom type, thanks to the Resource **enum**

And with all of that ready, let's get to actually giving some life to our little woodcutters and miners!

Finding a target

The first thing we are going to work on is the base finding mechanic, so that our collectors are able to spot the nearest resource or storage building. The idea here will be to look at either the resource map or the buildings map matching the resource this collector has to gather, and then loop around the current position of the unit in concentric circles to find a target.

TAKING THE OBSTACLES INTO ACCOUNT

> Here, we're not going to consider the obstacles when computing our paths, but we could technically improve our algorithm by assigning a higher weight to the non-walkable areas to artificially increase the distance.

Implementing the base search logic

To begin with, let's define a new TaskFindClosestTarget node for our collector AI behaviour tree that inherits from the Node class in our BehaviorTree package, and keeps references to the transform of the unit and the tilemap to search:

TaskFindClosestTarget.cs

```
1  using UnityEngine;
2  using UnityEngine.Tilemaps;
3
4  using BehaviorTree;
5
6  public class TaskFindClosestTarget : Node {
7    private Transform _transform;
8    private Tilemap _searchTilemap;
9
10    public TaskFindClosestTarget(Transform transform, Tilemap searchTilemap){
12      _transform = transform;
13      _searchTilemap = searchTilemap;
14    }
15 }
```

(Note that to use the Tilemap type, we need to import the UnityEngine.Tilemaps package.)

At the start, we will also get the max size of our tilemap (i.e. the length of its longest side between the width and the height) so that we know how large our search can be before leaving the map completely, and the anchor the tilemap uses has so that we can properly position our target tile location in world space coordinates:

TaskFindClosestTarget.cs

```
1  public class TaskFindClosestTarget : Node {
2    private Transform _transform;
3    private Tilemap _searchTilemap;
4
5    private int _maxGridSize;
6    private Vector3 _gridTileAnchor;
7
8    public TaskFindClosestTarget(Transform transform, Tilemap searchTilemap){
9      _transform = transform;
10     _searchTilemap = searchTilemap;
11     BoundsInt tilemapBounds = _searchTilemap.cellBounds;
12     _maxGridSize = Mathf.Max(tilemapBounds.size.x, tilemapBounds.size.y);
13     _gridTileAnchor = _searchTilemap.tileAnchor;
```

```
14    }
15 }
```

Then, in the Evaluate() function of our node, we will floor our current unity position to the closest integer tile position, and create our search circles by incrementing a search radius variable, and then considering the square around our position with that

---

TaskFindClosestTarget.cs

```
1  public class TaskFindClosestTarget : Node {
2     // ...
3
4     public override NodeState Evaluate() {
5        Vector3Int p = Vector3Int.FloorToInt(_transform.position);
6        for (int radius = 0; radius < _maxGridSize; radius++) {
7           for (int x = p.x - radius; x <= p.x + radius; x++) {
8              for (int y = p.y - radius; y <= p.y + radius; y++) {
9                 // is it a valid target?
10             }
11          }
12       }
```

```
13     }
14 }
```

If during one if the iterations, we get a world position that can be matched to a tile in our search tilemap, then it means we have found a valid target. We will then mark this target position in our tree's root node data and store the position in integer tile space coordinates too, say that our node succeeded and finally cut short the rest of the search:

TaskFindClosestTarget.cs

```
1  public class TaskFindClosestTarget : Node {
2    // ...
3
4    public override NodeState Evaluate() {
5      Vector3Int p = Vector3Int.FloorToInt(_transform.position);
6      for (int radius = 0; radius < _maxGridSize; radius++) {
7        for (int x = p.x - radius; x <= p.x + radius; x++) {
8          for (int y = p.y - radius; y <= p.y + radius; y++) {
9            Vector3 worldPos =
10             new Vector3(x, y, 0) + _gridTileAnchor;
11           Vector3Int cellPos =
```

```
                _searchTilemap.WorldToCell(worldPos);
            if (_searchTilemap.HasTile(cellPos)) {
                Root.SetData("target", worldPos);
                Root.SetData("target_cell", cellPos);
                _state = NodeState.SUCCESS;
                return _state;
            }
        }
    }
}
```

Else, if we didn't find any target during our search, we will return a failure:

TaskFindClosestTarget.cs

```
public class TaskFindClosestTarget : Node {
  // ...

  public override NodeState Evaluate() {
    Vector3Int p = Vector3Int.FloorToInt(_transform.position);
```

```
        for (int radius = 0; radius < _maxGridSize; radius++) { ... }

        _state = NodeState.FAILURE;
        return _state;
    }
}
```

Checking for the type of target

One other thing we can do is to also define a boolean flag in our node and in our final data to remember whether this task searched for a resource, or a building:

TaskFindClosestTarget.cs

```
public class TaskFindClosestTarget : Node {
    // ...
    private bool _searchingForResource;

    public TaskFindClosestTarget(
        Transform transform, Tilemap searchTilemap,
        bool searchingForResource) {
        // ...
```

```
 9        _searchingForResource = searchingForResource;
10    }
11
12    public override NodeState Evaluate() {
13        Vector3Int p = Vector3Int.FloorToInt(_transform.position);
14        for (int radius = 0; radius < _maxGridSize; radius++) {
15            for (int x = p.x - radius; x <= p.x + radius; x++) {
16                for (int y = p.y - radius; y <= p.y + radius; y++) {
17                    Vector3 worldPos = new Vector3(x, y, 0) + _gridTileAnchor;
18                    Vector3Int cellPos = _searchTilemap.WorldToCell(worldPos);
19                    if (_searchTilemap.HasTile(cellPos)) {
20                        // ...
21                        Root.SetData("target_is_resource", _searchingForResource);
22                    }
23                }
24            }
25        }
26        // ...
27    }
28 }
```

This will be useful later on, when we actually reach the target, to know if we should switch to “collect” or “deliver” mode.

While we're at it, we'll also quickly make a check node to look up this root data and verify if the collector currently has a target or not, which is naturally called

---

CheckHasTarget.cs

```
1  using BehaviorTree;
2
3  public class CheckHasTarget : Node {
4      public override NodeState Evaluate() {
5          _state = Root.GetData("target") == null
6              ? NodeState.FAILURE : NodeState.SUCCESS;
7          return _state;
8      }
9  }
```

Ok – now that we have our target, let's create a second task node that takes care of moving our unit from its current position to this target location!

Walking to the target

Our TaskFindClosestTarget node allows us to spot either resource locations or storage buildings, and store the position of the closest one in the root of our behaviour tree. So, the next step is to use this data to actually have the unit move to this position.

Coding up the walk task node

To do this, let's create another node called TaskWalk and pass it our unit transform, an instance of the A* pathfinder class (from the AStar namespace) and a move speed:

TaskWalk.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using BehaviorTree;
4  using AStar;
5
6  public class TaskWalk : Node {
7      private Transform _transform;
8      private Pathfinder2D _pathfinder;
9      private float _speed;
10
11     public TaskWalk(Transform transform, Pathfinder2D pathfinder, float speed) {
```

```
12          _transform = transform;
13          _pathfinder = pathfinder;
14          _speed = speed;
15      }
16  }
```

Because the ground map and its walkable tiles don't change while the game runs, we can actually prepare our pathfinder at the beginning in our and then pass this pathfinder object to our node — we just have to give it the ground tilemap and it will directly compute a grid of walkable and non-walkable cells from it:

CollectorAI.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using UnityEngine.Tilemaps;
4
5  public class CollectorAI : MonoBehaviour {
6      public enum Resource { ... }
7      [SerializeField] private Resource _resource;
8      [SerializeField] private Tilemap _groundTilemap;
```

```
 9
10     private void Start() {
11         AStar.Pathfinder2D pathfinder =
12             new AStar.Pathfinder2D(_groundTilemap);
13     }
14 }
```

In the TaskWalk class, we will also prepare a private variable to hold the list of Vector3 positions the A* pathfinder found to get to the target location, so that we can iterate through it and walk the path once it's been computed:

TaskWalk.cs

```
 1 public class TaskWalk : Node {
 2     private Transform _transform;
 3     private Pathfinder2D _pathfinder;
 4     private float _speed;
 5     private List<Vector3> _pathToTarget;
 6
 7     public TaskWalk(Transform transform, Pathfinder2D pathfinder, float speed) {
 8         _transform = transform;
 9         _pathfinder = pathfinder;
```

```
10         _speed = speed;
11     }
12
13 }
```

Then, in the Evaluate() function, we'll have two cases:

If we don't yet have a path, then we'll get back the target world position from the tree root data, use the pathfinder to compute a route to this location and store the result in the _pathToTarget variable. But this can be a null path if the pathfinder did not find any solution, so we should check for this bad case and, if that happens, set our node state to a failure.

Also, just to avoid useless computations, we can ensure that we are not actually already at the target point! For this, we'll simply add a constant float for the reach threshold distance and, if the distance between the current unit position and the target position is lower than this threshold, return a success directly and bypass the whole logic.

TaskWalk.cs

```
public class TaskWalk : Node {
    // ...
    private const float _REACH_THRESHOLD = 0.01f;

    // ...

    public override NodeState Evaluate() {
        _state = NodeState.RUNNING;

        // if no path yet, compute it
        if (_pathToTarget == null) {
            Vector3 targetPos = (Vector3)GetData("target");
            float d = Vector3.Distance(_transform.position, targetPos);
            if (d < _REACH_THRESHOLD) {
                _state = NodeState.SUCCESS;
            }
            else {
                _pathToTarget = _pathfinder.FindPath(_transform.position, targetPos);
                if (_pathToTarget == null)
                    _state = NodeState.FAILURE;
            }
        }

        return _state;
    }
}
```

The second possibility is that we have already computed a path, and we need to walk through all the tiles in this route to reach the target. The idea here is simply to take the first position in the list (because remember it is sorted from start to target point), and check if the unit is close to this position by re-using our reach threshold distance.

If we have arrived at the tile, then we need to pop this position from the list so that, next time, we look at the next node. And if there is no next node because we've popped everything out, then it means we've reached the target and our node should return a success.

If we haven't arrived yet, then we will simply move our transform towards the next tile position based on our given speed.

TaskWalk.cs

```
1  public class TaskWalk : Node {
2      // ...
3
4      public override NodeState Evaluate() {
5          _state = NodeState.RUNNING;
```

```
6
7       // if no path yet, compute it
8       // ...
9
10      // else walk through the path
11      else if (_pathToTarget.Count > 0) {
12          Vector3 t = _pathToTarget[0];
13
14          float d = Vector3.Distance(_transform.position, t);
15          if (d < _REACH_THRESHOLD) { // if reached tile
16              _transform.position = t;
17              _pathToTarget.RemoveAt(0);
18
19              // special case: reached the end
20              if (_pathToTarget.Count == 0) {
21                  _state = NodeState.SUCCESS;
22                  _pathToTarget = null;
23              }
24          }
25          else { // else move towards the tile
26              _transform.position = Vector3.MoveTowards(
27                  _transform.position, t, _speed * Time.deltaTime);
28          }
29      }
30
31      return _state;
32  }
```

```
33 }
```

Integrating our nodes into a behaviour tree

Ok, we now have all the nodes we need for finding a target and walking to it. So let's go back to our CollectorAI script and do a very basic first version of our behaviour tree. We'll make our script inherit from the BTree class in the BehaviorTree package, define a new speed variable for the walk task and move the current initialisation logic inside the SetupTree() method:

CollectorAI.cs

```
1  using System.Collections.Generic;
2  using UnityEngine;
3  using BehaviorTree;
4
5  public class CollectorAI : BTree {
6      public enum Resource { ... }
7      [SerializeField] private Resource _resource;
8      [SerializeField] private Tilemap _groundTilemap;
9      [SerializeField] private float _speed = 3;
```

```
10
11      private void Start() {}
12
13      protected override Node SetupTree() {
14          AStar.Pathfinder2D pathfinder =
15              new AStar.Pathfinder2D(_groundTilemap);
16
17          Node root = null;
18          return root;
19      }
```

```
20 }
```

Then, we'll check that our ground tilemap is setup correctly, and we'll do a quick search through the available tilemaps to find the ones with names matching the resource this collector should gather (for example Resource:Wood and Buildings:Wood for a woodcutter). If any tilemap is missing, we'll debug a warning and interrupt the init here because the unit won't be able to run its behaviour properly anyways!

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // check for ground tilemap reference + prepare pathfinder
3      if (_groundTilemap == null) {
4          Debug.LogWarning($"No ground tilemap for '{name}'");
5          return null;
6      }
7
8      AStar.Pathfinder2D pathfinder =
9          new AStar.Pathfinder2D(_groundTilemap);
10
```

```
11      // get resource/storage maps
12      Tilemap resourceTilemap = null, storageTilemap = null;
13      Tilemap[] tilemaps = FindObjectsOfType<Tilemap>();
14      foreach (Tilemap t in tilemaps) {
15          if (t.name == $"Resource:{_resource}") {
16              resourceTilemap = t;
17          } else if (t.name == $"Buildings:{_resource}") {
18              storageTilemap = t;
19          }
20      }
21
22      if (resourceTilemap == null) {
23          Debug.LogWarning($"Cannot find resource for '{_resource}'");
24          return null;
25      }
26      if (storageTilemap == null) {
27          Debug.LogWarning($"Cannot find buildings for '{_resource}'");
28          return null;
29      }
30
31      Node root = null;
32      return root;
33  }
```

But if everything is properly set up, then we'll be able to create our tree, which currently looks as follows:

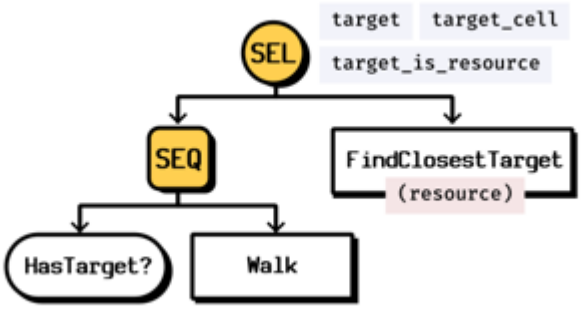


Figure 8.4 – V1 of the behaviour tree for our collector AI which searches for the closest resource target and walks to it if it has found one

This tree translates to the following C# code:

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3
4      Node root = new Selector();
5      root.SetChildren(new List<Node>() {
6          new Sequence(new List<Node>() {
```

```
            new CheckHasTarget(),
            new TaskWalk(transform, pathfinder, _speed)
        }),
        new TaskFindClosestTarget(transform, resourceTilemap, true)
    }, forceRoot: true);

    return root;
}
```

(Don't forget that because of the system of root we put in place for our behaviour tree in the abstract Node class, we have to pass in the forceRoot parameter to the SetChildren() function call to properly reference our root variable as the root, and not the intermediary parents.)

For now, the tree is super simple: it simply checks to see if there is currently a target and, in that case, walks towards it as determined by the A* pathfinder; else, it goes to the other branch of the Selector and tries to find a target using the resource map — so, in other words, it looks for the closest resource spot (which, here, is a tree, because I've said that my unit should gather wood).

Back in the editor, we also have to remember to set a speed for the unit in the inspector, and to drag in the ground

tilemap reference:

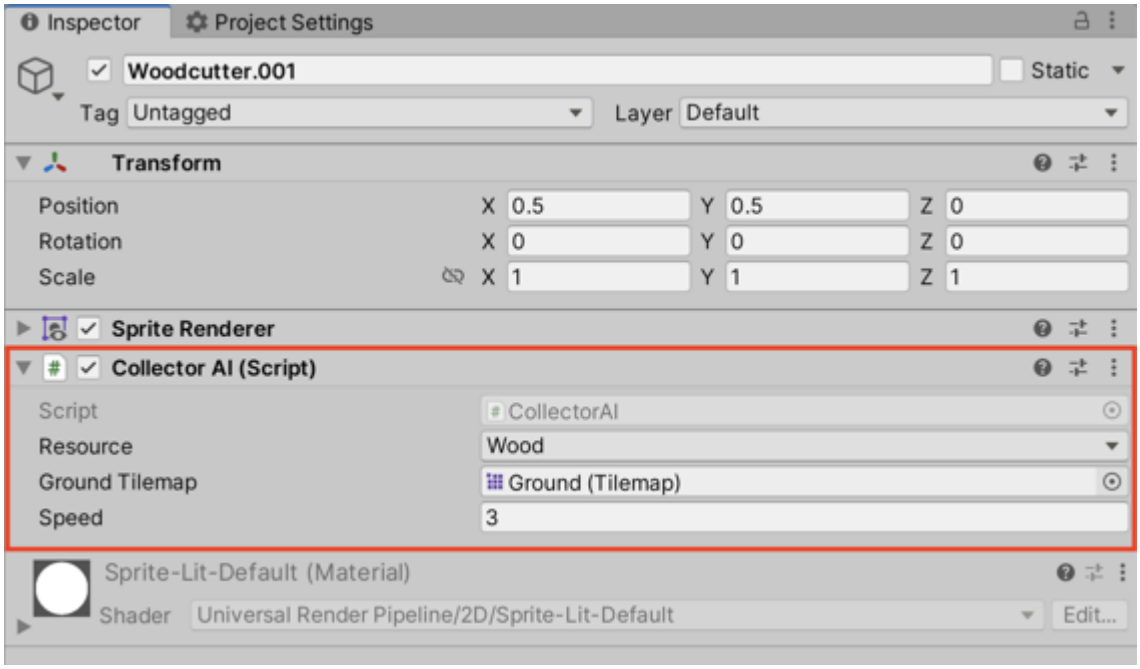


Figure 8.5 – Updated Collector AI **component in the inspector panel for a woodcutter unit, with a** _speed **value and a** _groundTilemap **reference**

And now, if you run this, you'll see that the unit does avoid obstacles like the river thanks to the clever path computing, and eventually ends up on the closest tree tile as the crow flies (see *Figure*

The only issue is that you'll notice the truck doesn't turn in the direction it's going — the sprite is only sliding sideways to the next waypoint. What we need is to implement a way of reacting to the tile switch so that, whenever the unit targets a

new position (even an intermediary one), the sprite adjusts by flipping on the X or the Y axis, and optionally changes from a horizontal to a vertical version.

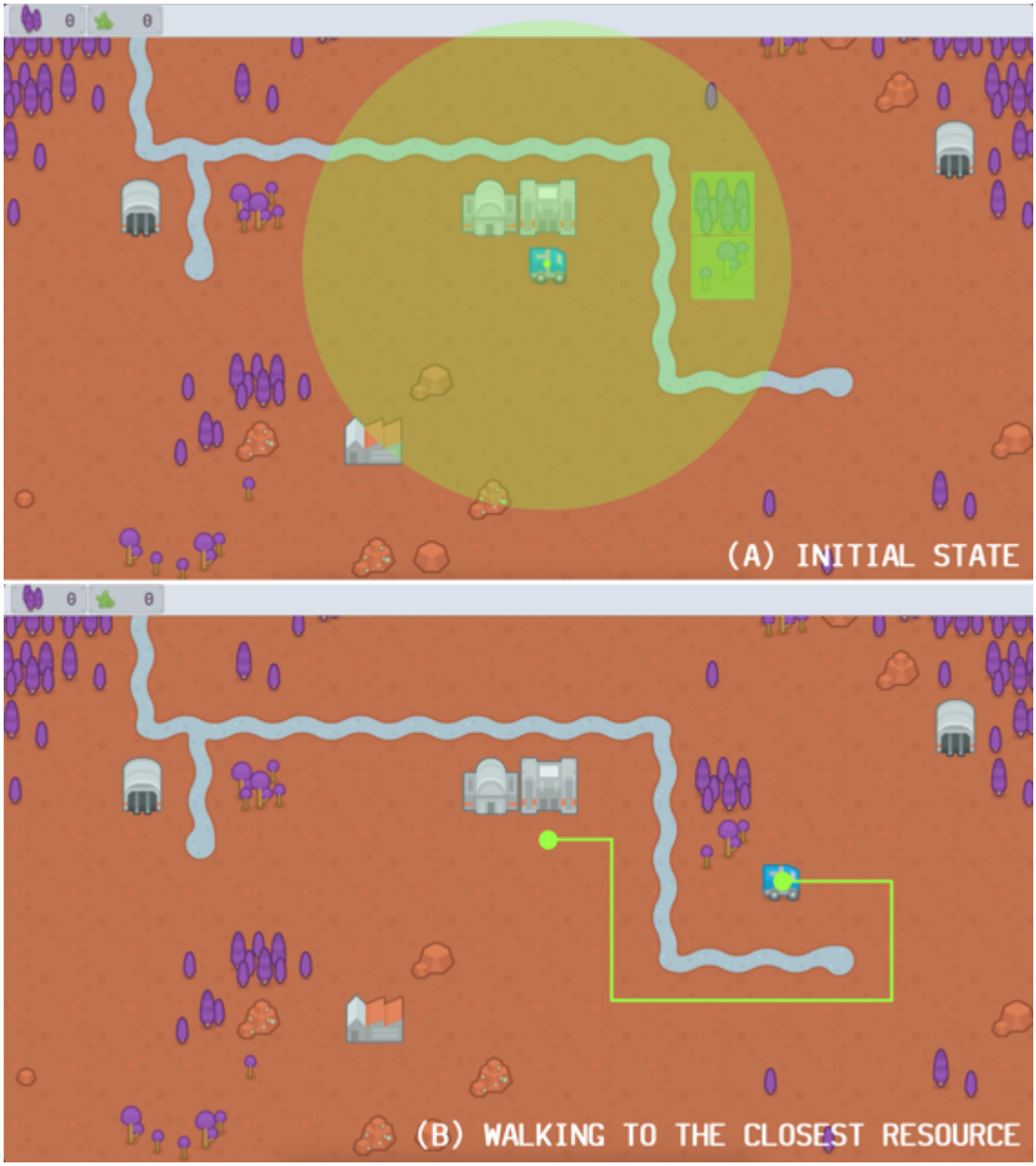


Figure 8.6 – V1 of the collector AI: the unit finds its closest resource in a given search radius, and then walks to it cell by cell on the grid using an A* 2D path

A nice solution to this problem is to go back to our TaskWalk node and, here, define an action called _onReachTile that takes in a Vector2 parameter. We will pass it in via the constructor and then call it when we aim for a new tile — so either when we first compute the path, or when we reach a tile and we have another one left in the path:

TaskWalk.cs

```
1  public class TaskWalk : Node {
2    // ...
3    private System.Action<Vector2> _onReachTile;
4
5    public TaskWalk(
6      Transform transform, Pathfinder2D pathfinder,
7      float speed, System.Action<Vector2> onReachTile) {
8      // ...
9      _onReachTile = onReachTile;
10   }
11
12   public override NodeState Evaluate() {
```

```
13        _state = NodeState.RUNNING;
14        // if no path yet, compute it
15        if (_pathToTarget == null) {
16            Vector3 targetPos = (Vector3)GetData("target");
17            float d = Vector3.Distance(_transform.position, targetPos);
18            if (d < _REACH_THRESHOLD) { ... }
19            else {
20                _pathToTarget = _pathfinder.FindPath(_transform.position, targetPos);
21                if (_pathToTarget == null) _state = NodeState.FAILURE;
22                 else _onReachTile(_GetNextDir());
23            }
24        }
25        // else walk through the path
26        else if (_pathToTarget.Count > 0) {
27            Vector3 t = _pathToTarget[0];
28            float d = Vector3.Distance(_transform.position, t);
29            if (d < _REACH_THRESHOLD) { // if reached tile
30                // ...
31                // special case: reached the end
32                if (_pathToTarget.Count == 0) { ... }
33                else _onReachTile(_GetNextDir());
34            }
35            // else move towards the tile
36            else { ... }
37        }
38        return _state;
39    }
```

```
40
41     private Vector2 _GetNextDir() {
42         Vector3 nextPoint = _pathToTarget[0];
43         return (nextPoint - _transform.position).normalized;
44     }
45 }
```

In both those cases, we compute the next direction by normalising the difference between the upcoming position and our current unit position.

Now, in the CollectorAI script, we can a define our callback for our TaskWalk instance — we'll look at the X and Y component of the direction, determine if the unit is going right, left, up or down, and flip or set the sprite accordingly.

CollectorAI.cs

```
1  public class CollectorAI : Btree {
2      // ...
3      [SerializeField] private Sprite _horizontalSprite;
4      [SerializeField] private Sprite _verticalSprite;
5      [SerializeField] private SpriteRenderer _spriteRenderer;
```

```
6
7   /// ...
8
9   protected override Node SetupTree() {
10      // ...
11      Node root = new Selector();
12      root.SetChildren(new List<Node>() {
13        new Sequence(new List<Node>() {
14          new CheckHasTarget(),
15          new TaskWalk(transform, pathfinder, _speed, (Vector2 dir) => {
16              if (Mathf.Approximately(dir.x, 1f)) {
17                  _spriteRenderer.sprite = _horizontalSprite;
18                  _spriteRenderer.flipX = false;
19                  _spriteRenderer.flipY = false;
20              } else if (Mathf.Approximately(dir.x, -1f)) {
21                  _spriteRenderer.sprite = _horizontalSprite;
22                  _spriteRenderer.flipX = true;
23                  _spriteRenderer.flipY = false;
24              } else if (Mathf.Approximately(dir.y, 1f)) {
25                  _spriteRenderer.sprite = _verticalSprite;
26                  _spriteRenderer.flipY = true;
27              } else if (Mathf.Approximately(dir.y, -1f)) {
28                  _spriteRenderer.sprite = _verticalSprite;
29                  _spriteRenderer.flipY = false;
30              }
31            })
32        }),
```

```
33            new TaskFindClosestTarget(transform, resourceTilemap, true)
34        }, forceRoot: true);
35
36        return root;
37    }
38 }
```

Again, don't forget to assign the sprites in the inspector :)

If we re-run the game, we see that our unit now moves and rotates properly all the way to the target tree location!

Figure 8.7 – Improvement of the collector AI to have it choose the horizontal or vertical sprite depending on the position of

its next target cell, which fixes the rotation issues

Ok — our collector can now find and move towards a target; time to get the fun part and see how to collect these resources...

Collecting resources

Writing the collect logic

The main node we'll need to code up for the collect behaviour is obviously the In here, we'll want to increment a global integer variable that contains the current amount of resource this unit holds, and tell our resource map that this tile has been consumed one time and should start to deplete.

For this use case, we'll assume that the resource management logic is already handled by another script called that is already added as a component on the resource tilemap objects (see *Figure* This script takes care of assigning values to the different tile types in the map (for example, a big pack of trees is worth more than a few isolated trunks), and it will also handle the evolution of the tilemaps and the depletion of the resources for us.

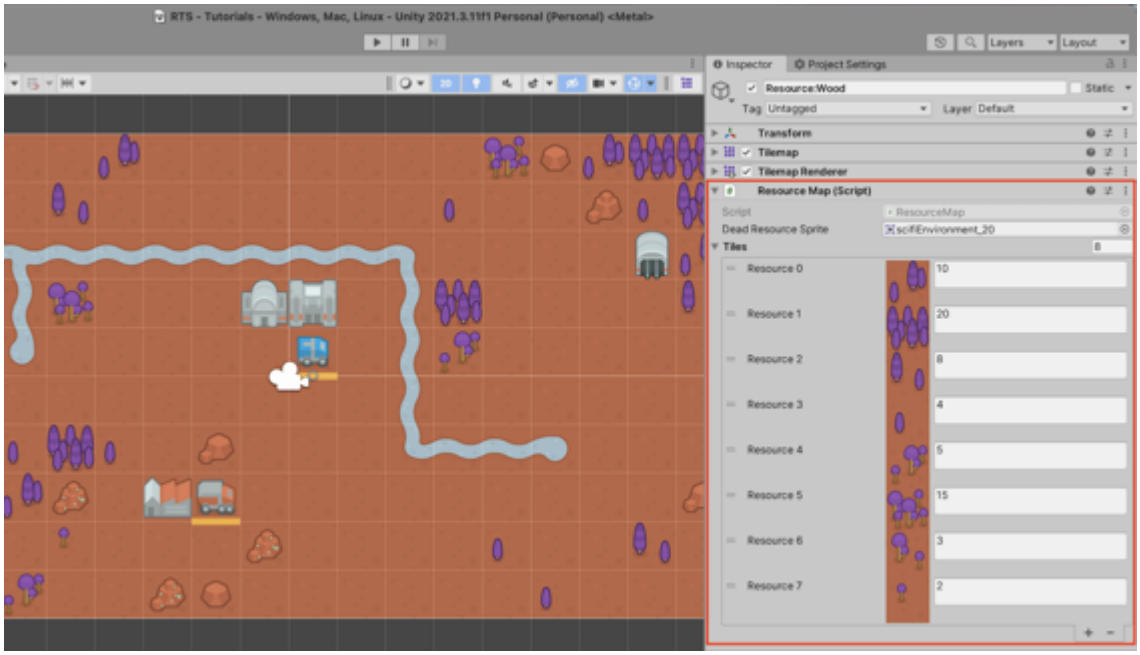


Figure 8.8 – ResourceMap **script instance on the wood resource map – this script sets the resource value of each tile type and can handle resource depletion**

So we'll just get a reference to this script in the CollectorAI init phase:

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3
4      // get resource/storage maps
```

```
5     Tilemap resourceTilemap = null, storageTilemap = null;
6     ResourceMap resourceMap = null;
7     Tilemap[] tilemaps = FindObjectsOfType<Tilemap>();
8     foreach (Tilemap t in tilemaps) {
9         if (t.name == $"Resource:{_resource}") {
10            resourceTilemap = t;
11            resourceMap = t.GetComponent<ResourceMap>();
12        } else if (t.name == $"Buildings:{_resource}") {
13            storageTilemap = t;
14        }
15    }
16
17    // ...
18 }
```

And then pass it on to the TaskCollect node, along with an integer variable that defines the maximum amount of resource this unit can hold before having to return to the storage building:

TaskCollect.cs

```
1 using System.Collections.Generic;
```

```
using UnityEngine;
using BehaviorTree;

public class TaskCollect : Node {
    private ResourceMap _resourceMap;
    private int _maxStorage;

    public TaskCollect(ResourceMap resourceMap, int maxStorage) {
        _resourceMap = resourceMap;
        _maxStorage = maxStorage;
    }
}
```

This max storage value will be defined in the CollectorAI script along with the other high-level unit parameters:

CollectorAI.cs

```
public class CollectorAI : BTree {
    // ...
    [SerializeField] private int _maxStorage = 20;
}
```

Now, in the Evaluate() method of our TaskCollect class, we can access the current_resource_amount data field of our tree root, increment it by an arbitrary amount of _COLLECT_AMOUNT (for example 1), and set back the updated value. We'll also make sure that if we exceed our max storage amount, we clamp the value back to the right range:

TaskCollect.cs

```
1  public class TaskCollect : Node {
2      private ResourceMap _resourceMap;
3      private int _maxStorage;
4      private const int _COLLECT_AMOUNT = 1;
5
6      public TaskCollect(ResourceMap resourceMap, int maxStorage) { … }
7
8      public override NodeState Evaluate() {
9          // update resource amount
10         int curAmount = (int)Root.GetData("current_resource_amount");
11         int newAmount = curAmount + _COLLECT_AMOUNT;
12         if (newAmount > _maxStorage) newAmount = _maxStorage;
13
```

```
14          Root.SetData("current_resource_amount", newAmount);
15
16          _state = NodeState.RUNNING;
17          return _state;
18      }
19  }
```

Except that, for now, we haven't defined this current_resource_amount data field anywhere! So it will be uninitialised and crash if we try to convert it to an integer like this. To solve the issue, we'll go to our CollectorAI and, when we first setup the tree, we'll set a default value of 0 for this field in our root node:

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3      Node root = ...
4
5      root.SetData("current_resource_amount", 0);
6      return root;
7  }
```

Ok, now back in our TaskCollect class, we need to handle the second part of the logic and update the resource map state to mark this tile as down 1 “point”. We can do this easily by retrieving the current target position in cell space coordinates and calling the ConsumeTile() function of our ResourceMap instance with this Vector3Int variable and our arbitrary

TaskCollect.cs

```
1  public override NodeState Evaluate() {
2      // update resource amount
3      int curAmount = (int)Root.GetData("current_resource_amount");
4      int newAmount = curAmount + _COLLECT_AMOUNT;
5      if (newAmount > _maxStorage) newAmount = _maxStorage;
6
7      Root.SetData("current_resource_amount", newAmount);
8
9      // update resource map
10     Vector3Int resourceCellPos = (Vector3Int)Root.GetData("target_cell");
11     _resourceMap.ConsumeTile(resourceCellPos, _COLLECT_AMOUNT);
12
13     _state = NodeState.RUNNING;
```

```
14      return _state;
15  }
```

This will automatically take care of the whole map update, including the possible “death” of the resource if the tile reaches 0 “points”. In that case, the sprite will change and the function will return a true boolean to warn us that this tile died and isn’t valid anymore.

So we’ll have to check for this output and, if the tile is dead, clear our target reference to tell the unit this tile cannot be collected anymore:

TaskCollect.cs

```
1  public class TaskCollect : Node {
2      // ...
3
4      public override NodeState Evaluate() {
5          // ...
6
7          // update resource map
8          Vector3Int resourceCellPos =
```

```
 9            (Vector3Int)Root.GetData("target_cell");
10        bool tileIsDead = _resourceMap.ConsumeTile(
11            resourceCellPos, _COLLECT_AMOUNT);
12        if (tileIsDead) _ClearTarget();
13
14        _state = NodeState.RUNNING;
15        return _state;
16    }
17
18    private void _ClearTarget() {
19        Root.ClearData("target");
20        Root.ClearData("target_cell");
21        Root.ClearData("target_is_resource");
22    }
23 }
```

And because other woodcutters could work on the same tiles at the same time and "kill" it before we do, we should also wrap our resource map update in a try-catch block on a which if triggered will also clear the target reference. This is a quick way to avoid null references and reset the state of our unit if the tile isn't valid but we weren't aware of that yet:

TaskCollect.cs

```
public override NodeState Evaluate() {
    // ...

    // update resource map
    Vector3Int resourceCellPos = (Vector3Int)Root.GetData("target_cell");
    try {
        bool tileIsDead = _resourceMap.ConsumeTile(
            resourceCellPos, _COLLECT_AMOUNT);
        if (tileIsDead) _ClearTarget();
    } catch (KeyNotFoundException) {
        // (target is not valid anymore, but we are not yet aware)
        _ClearTarget();
    }

    _state = NodeState.RUNNING;
    return _state;
}
```

Adding a few more check nodes

Now, to integrate this node in our behaviour tree, we'll need two additional check nodes first: the CheckInTargetRange and These are pretty self-explanatory and easy to implement: we just have to get the data from our tree root node and check for the distance to the current unit position, or the value of our target_is_resource boolean flag.

CheckInTargetRange.cs

```
1  using UnityEngine;
2  using BehaviorTree;
3
4  public class CheckInTargetRange : Node {
5      private const float _REACH_THRESHOLD = 0.01f;
6      private Transform _transform;
7
8      public CheckInTargetRange(Transform transform) {
9          _transform = transform;
```

```
    }

    public override NodeState Evaluate() {
        Vector3 t = (Vector3)Root.GetData("target");
        float d = Vector2.Distance(_transform.position, t);
        _state = d < _REACH_THRESHOLD
            ? NodeState.SUCCESS : NodeState.FAILURE;
        return _state;
    }
}
```

CheckTargetIsResource.cs

```
using BehaviorTree;

public class CheckTargetIsResource : Node {
    public override NodeState Evaluate() {
        bool targetIsResource = (bool)Root.GetData("target_is_resource");
        _state = targetIsResource
            ? NodeState.SUCCESS : NodeState.FAILURE;
        return _state;
    }
}
```

Updating our behaviour tree

We now have everything we need to setup the collect logic in our behaviour tree! So let's go back to our CollectorAI and modify our tree to update it to this:

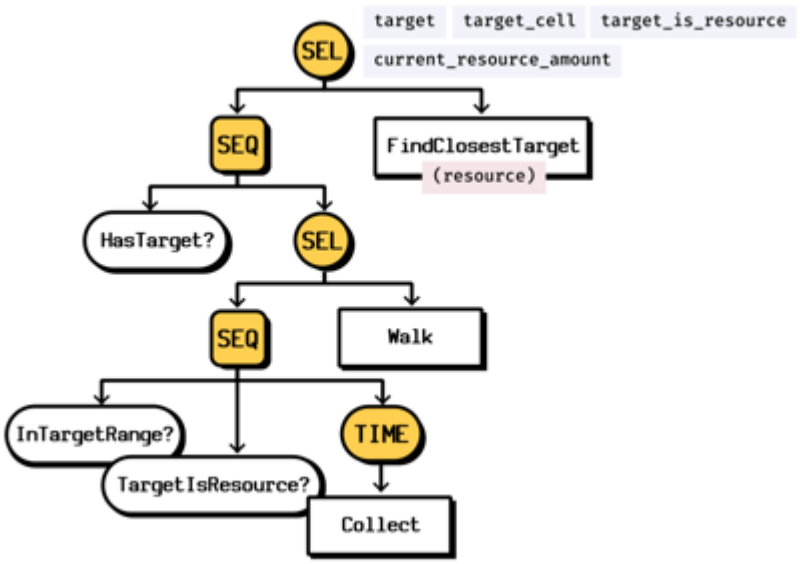


Figure 8.9 – V2 of the behaviour tree for our collector AI which adds the resource collect sub-branch

As you can see, the point is to integrate the sub-branch at the bottom that checks if we are close to the target and if it's a resource, and in that case runs the collect logic. But since we

want the collect task to repeat multiple times, we will decorate our TaskCollect node with a Timer that runs it every so often.

Here is the corresponding C# code:

CollectorAI.cs

```
1  public class CollectorAI : BTree {
2    // ...
3    private static float _COLLECT_RATE = 0.5f;
4
5    // ...
6
7    protected override Node SetupTree() {
8      // ...
9
10       Node root = new Selector();
11       root.SetChildren(new List<Node>() {
12         new Sequence(new List<Node>() {
13           new CheckHasTarget(),
14           new Selector(new List<Node>() {
15             new Sequence(new List<Node>() {
16               new CheckInTargetRange(transform),
17               new CheckTargetIsResource(),
18               new Timer(_COLLECT_RATE, new List<Node>() {
```

```
                    new TaskCollect(resourceMap, _maxStorage),
                })
            }),
            new TaskWalk( transform, pathfinder,
                _speed, (Vector2 dir) => { ... })
        }),
    }),
    new TaskFindClosestTarget(transform, resourceTilemap, true)
}, forceRoot: true);

root.SetData("current_resource_amount", 0);

return root;
}
}
```

DEFINING THE COLLECT RATE

> Here, the collect rate is another arbitrary value we set in our CollectorAI, but we could of course have it depend on the unit type, or its level, or whatnot…

A cool feature of the Timer node we prepared in *Chapter 7* is that we can give it an action to run whenever it finishes its countdown. Here, we'll use that to update the visual of the

unit resource storage bar — this will help us see how the collect progresses and identify when the unit reaches its max storage:

CollectorAI.cs

```
1  public class CollectorAI : BTree {
2      //...
3      [SerializeField] private Transform _resourceFillBar;
4
5      private void Start() {
6          _UpdateResourceFillBar();
7      }
8
9      protected override Node SetupTree() {
10         //...
11
12         Node root = new Selector();
13         root.SetChildren(new List<Node>() {
14             new Sequence(new List<Node>() {
15                 new CheckHasTarget(),
16                 new Selector(new List<Node>() {
17                     new Sequence(new List<Node>() {
18                         new CheckInTargetRange(transform),
19                         new CheckTargetIsResource(),
```

```
20                      new Timer(_COLLECT_RATE, new List<Node>() {
21                       new TaskCollect(resourceMap, _maxResourceStorage),
22                      }, _UpdateResourceFillBar)
23                  }),
24                  new TaskWalk(transform, pathfinder,
25                          _speed, (Vector2 dir) => { ... })
26              }),
27          }),
28          new TaskFindClosestTarget(transform, resourceTilemap, true)
29      }, forceRoot: true);
30      root.SetData("current_resource_amount", 0);
31      return root;
32  }
33
34  private void _UpdateResourceFillBar() {
35      int curAmount = (int) _root.GetData("current_resource_amount");
36      float resourceRatio = curAmount / (float)_maxStorage;
37      _resourceFillBar.localScale = new Vector3(resourceRatio, 1, 1);
38      _resourceFillBar.localPosition =
39          new Vector3(-0.5f + resourceRatio / 2f, 0, 0);
40  }
41 }
```

If we start the game now, the truck goes the closest tree and starts to chop it down tree to gather resources. We see that

its resource bar fills up slowly and, if a tile gets depleted, the collector goes to the next closest one.

Figure 8.10 – V2 of the collector AI: the unit now collects resources and fills up its resource bar

But of course, for now, our unit doesn't really check if it has reached its max storage. So all those trees it is collecting are just vanishing into thin air! That's why we have to handle the next part of our collector AI logic and have it deliver the goods to the nearest storage building.

Delivering the goods

Ok — now that our unit can stockpile resources, we need to tell it to check for its max storage limit and, if it reaches that threshold, switch over to the delivery logic.

Checking the unit's max storage limit

This delivery process will consist in three steps:

1. First, the unit will need to identify the closest storage location matching its resource.

2. Then, it will need to go to it with an A* path.

3. And finally, when it's close enough, it will need to "transfer its resources to the building". What this means for us is that our resource stats in the UI at the top of the screen will increase by the amount of goods delivered, and that the unit current amount will be reset to 0.

The two first part of this logic are already mostly taken care of by our TaskFindClosestTarget and We just need to prepare an additional check node to see if the collector is currently full, the CheckReachedMaxStorage class, which looks like this:

CheckReachedMaxStorage.cs

```
using BehaviorTree;

public class CheckReachedMaxStorage : Node {
    private int _maxStorage;

    public CheckReachedMaxStorage(int maxStorage) {
        _maxStorage = maxStorage;
    }

    public override NodeState Evaluate() {
        int a = (int)Root.GetData("current_resource_amount");
        _state = a == _maxStorage ? NodeState.SUCCESS : NodeState.FAILURE;
        return _state;
    }
}
```

And now, we can easily add a little sequence at the very beginning of our tree so that, if the unit has reached its storage limit and is currently without a target, or with a resource target, it instead finds the closest building on its associated storage tilemap and walks to it:

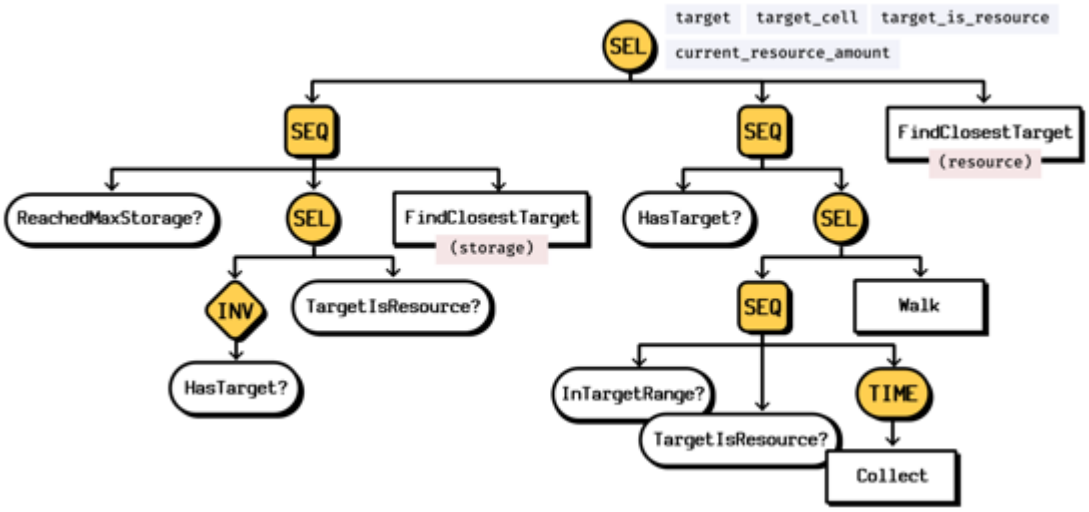


Figure 8.11 – V3 of the behaviour tree for our collector AI which adds the storage limit check and prioritises going back to the storage centre if need be

---

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3
4      Node root = new Selector();
5      root.SetChildren(new List<Node>() {
6          new Sequence(new List<Node>() {
7              new CheckReachedMaxStorage(_maxResourceStorage),
8              new Selector(new List<Node>() {
9                  new Inverter(new List<Node>() { new CheckHasTarget(), }),
```

```
            new CheckTargetIsResource(this),
        }),
        new TaskFindClosestTarget(transform, storageTilemap, false),
    }),
    new Sequence(new List<Node>() {
        // ...("move to target" sequence)
    }),
    new TaskFindClosestTarget(transform, resourceTilemap, true)
}, forceRoot: true);

root.SetData("current_resource_amount", 0);
return root;
}
```

If you run the game at this point, you'll see that the truck indeed stops its collect as soon as it reaches the max storage threshold and then navigates to the storage location on the right:

Figure 8.12 – V3 of the collector AI: when the unit's resource storage is full, it goes back to the nearest storage centre

Setting up the deliver process

Now, to have it actually deliver the goods and free space for further gathering, we're going to create a new TaskDeliver node. This node will be quite simple — we'll just receive the type of resource this collector gathers and then, when we evaluate this node, we'll send a global event telling that we've collected the given amount of resource of this type and reset all of our variables (the current resource amount and the target data):

TaskDeliver.cs

```
1  using BehaviorTree;
2
3  public class TaskDeliver : Node {
4      private string _resourceType;
5
6      public TaskDeliver(string resourceType) {
7          _resourceType = resourceType;
8      }
9
10     public override NodeState Evaluate() {
11         // send global event with amount/type of resource collected
12         int resourceAmount = (int)Root.GetData("current_resource_amount");
13         EventManager.TriggerEvent("ResourceCollected", new object[] {
14             _resourceType, resourceAmount
15         });
16
17         // reset local storage of unit
18         Root.SetData("current_resource_amount", 0);
19
20         // clear target
21         Root.ClearData("target");
22         Root.ClearData("target_cell");
23         Root.ClearData("target_is_resource");
24
```

```
25          _state = NodeState.RUNNING;
26          return _state;
27      }
28  }
```

This event will be conveyed by a global scene event manager I prepared beforehand, and it will be used by the UI manager in the scene to update the resource labels at the top of the screen.

WANNA LEARN MORE ABOUT THIS EVENT MANAGER?

> This event manager is a basic data sharing tool that makes it easy to convey information throughout my scene, from one system to another, without having to define any strong dependencies. It relies on a singleton EventManager instance that stores all the possible events in a C# Dictionary and re-dispatches triggers to the right listeners when an event is invoked.

> If you want to learn more about how this tool works, you can either have a look in the Github repo of this book (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) or check out the Unity tutorial I made on that topic, available over here on YouTube: https://www.youtube.com/watch?v=EvqdcyTgZNg.

Last but not least, let's integrate this new node in our collector's behaviour tree. Basically, we will need to extend our "close to target" sub-branch from before so that:

if the target is a resource we collect it,

or else if it is a storage centre we deliver the goods.

We'll also wrap our TaskDeliver instance in another Timer decorator so that we can have a little delay upon arrival, and use its callback feature to re-update the unit's resource storage bar and show it is now empty.

Our new tree therefore looks like this:

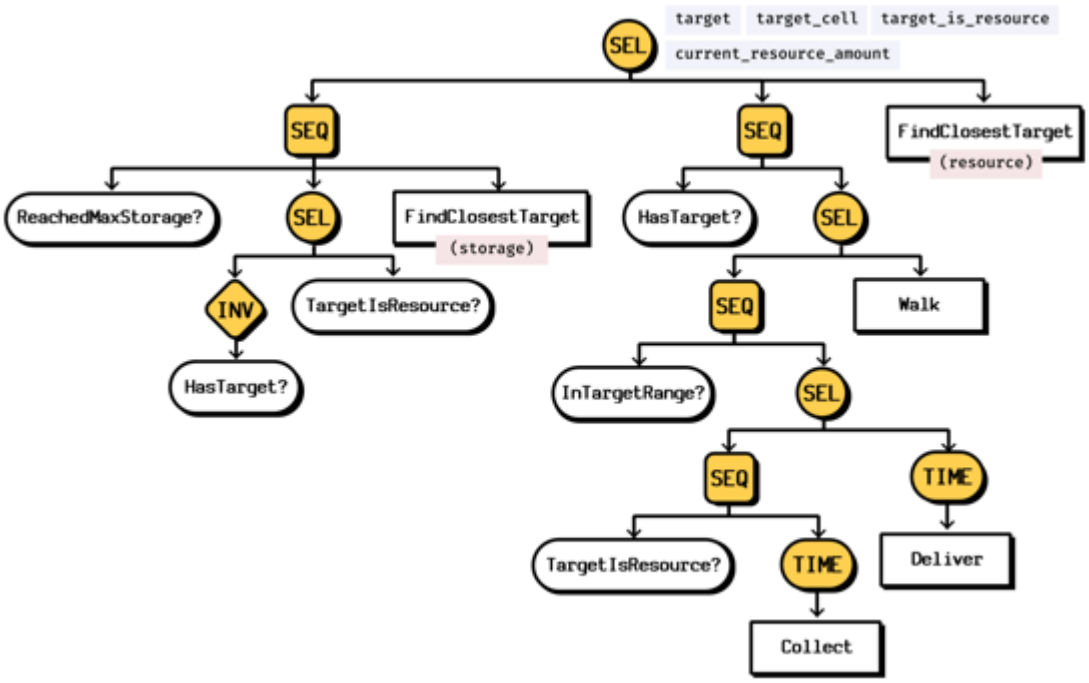


Figure 8.13 – V4 of the behaviour tree for our collector AI which checks for the target type on arrival to collect or deliver the resources

And here is the equivalent in C#:

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3
4      Node root = new Selector();
```

```
5   root.SetChildren(new List<Node>() {
6     new Sequence(new List<Node>() {
7       // ... ("check for max" sequence)
8     }),
9     new Sequence(new List<Node>() {
10      new CheckHasTarget(),
11      new Selector(new List<Node>() {
12        new Sequence(new List<Node>() {
13          new CheckInTargetRange(transform),
14          new Selector(new List<Node>() {
15            new Sequence(new List<Node>() {
16              new CheckTargetIsResource(),
17              new Timer(_COLLECT_RATE, new List<Node>() {
18                new TaskCollect(
19                  resourceMap, _maxResourceStorage),
20              }, _UpdateResourceFillBar)
21            }),
22            new Timer(_DELIVER_DELAY, new List<Node>() {
23              new TaskDeliver(_resource.ToString()),
24            }, _UpdateResourceFillBar)
25          })
26        }),
27        new TaskWalk(transform, pathfinder,
28          _speed, (Vector2 dir) => { ... })
29      }),
30    }),
31    new TaskFindClosestTarget(transform, resourceTilemap, true)
```

```
32      }, forceRoot: true);
33
34      root.SetData("current_resource_amount", 0);
35      return root;
36  }
```

And there we are, our collector AI is nearly done! If we start the game, we see that our woodcutter finds the nearest tree, chops it down and gathers resource, then goes to the next closest resource and continues stockpiling until it has reached its max storage limit. At that point, it switches over to delivery mode and goes to the nearest storage building to drop the resources and reset to an empty storage. This also updates the player resource indicators at the top of the screen.

The core logic of our unit is now finished... but a final improvement we can make is, in case there is no resource left on the map to collect for this unit, have it return to the nearest building.

Returning back home

To end this chapter, let's take care of the idle logic for our collectors and make sure that they go back to the closest

storage building and enter it if they don't have anything left to do.

Finding the closest storage centre

First of all, to have our unit return to a building in case it's idle, we simply have to add one line in our behaviour tree logic. At the very far right, meaning after our TaskFindClosestTarget instance that looks for a valid resource target, we can add TaskFindClosestTarget node that looks for a target in the storage map:

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3
4      Node root = new Selector();
5      root.SetChildren(new List<Node>() {
6          // ...
7          new TaskFindClosestTarget(transform, resourceTilemap, true),
8          new TaskFindClosestTarget(transform, storageTilemap, false)
9      }, forceRoot: true);
10
11     root.SetData("current_resource_amount", 0);
```

```
12      return root;
13  }
```

This way, if every tile on the resource map is depleted and so there is no resource target to be found, the AI will default to this new action.

This is a great example of how well-suited behaviour trees are for handling interruptions and fallbacks, since just by injecting the right node in the right place, we were able to give our unit a brand new action in a flash :)

Disabling the unit's sprite

Now, to actually have the truck "enter the building", we will hide its sprite and resource bar elements if they are still visible. So we'll first make a CheckIsVisible node that takes in the sprite renderer and checks if it's currently enabled:

CheckIsVisible.cs

```
1  using UnityEngine;
2  using BehaviorTree;
```

```
3
4  public class CheckIsVisible : Node {
5      private SpriteRenderer _spriteRenderer;
6
7      public CheckIsVisible(SpriteRenderer spriteRenderer) {
8          _spriteRenderer = spriteRenderer;
9      }
10
11     public override NodeState Evaluate() {
12         _state = _spriteRenderer.enabled ?
13             NodeState.SUCCESS : NodeState.FAILURE;
14         return _state;
15     }
16 }
```

And then we'll prepare another node, that calls a ToggleVisuals() function on the unit's CollectorAI script:

CheckIsVisible.cs

```
1  using BehaviorTree;
2
3  public class TaskEnterBuilding : Node {
```

```
    private CollectorAI _brain;

    public TaskEnterBuilding(CollectorAI brain) {
        _brain = brain;
    }

    public override NodeState Evaluate() {
        _brain.ToggleVisuals(false);
        _state = NodeState.SUCCESS;
        return _state;
    }
}
```

This ToggleVisuals() function simply enables or disables the sprite renderer and the storage bar sub-hierarchy.

CollectorAI.cs

```
public class CollectorAI : BTree {
    // ...

    public void ToggleVisuals(bool on) {
        _spriteRenderer.enabled = on;
```

```
6         _resourceFillBar.parent.gameObject.SetActive(on);
7     }
8 }
```

We will add instances of our CheckIsVisible and TaskEnterBuilding nodes at the same level as our collect and deliver actions, as the last fallback branch:

CollectorAI.cs

```
1  protected override Node SetupTree() {
2      // ...
3
4      Node root = new Selector();
5      root.SetChildren(new List<Node>() {
6          new Sequence(new List<Node>() {
7              // ... ("check for max" sequence)
8          }),
9          new Sequence(new List<Node>() {
10             new CheckHasTarget(),
11             new Selector(new List<Node>() {
12                 new Sequence(new List<Node>() {
13                     new CheckInTargetRange(transform),
```

```
14                  new Selector(new List<Node>() {
15                      new Sequence(new List<Node>() {
16                          new CheckTargetIsResource(),
17                          new Timer(_COLLECT_RATE, new List<Node>() {
18                              new TaskCollect(
19                                  resourceMap, _maxResourceStorage),
20                          }, () => { _UpdateResourceFillBar(); }),
21                      }),
22                      new Sequence(new List<Node>() {
23                          new CheckHasResource(),
24                          new Timer(_DELIVER_DELAY, new List<Node>() {
25                              new TaskDeliver(_resource.ToString()),
26                          }, () => { _UpdateResourceFillBar(); }),
27                      }),
28                      new Sequence(new List<Node>() {
29                          new CheckIsVisible(_spriteRenderer),
30                          new TaskEnterBuilding(this),
31                      }),
32                  })
33              }),
34              new TaskWalk(transform, pathfinder,
35                      _speed, (Vector2 dir) => { ... })
36          }),
37      }),
38      new TaskFindClosestTarget(transform, resourceTilemap, true),
39      new TaskFindClosestTarget(transform, storageTilemap, false)
40  }, forceRoot: true);
```

```
41      root.SetData("current_resource_amount", 0);
42      return root;
43  }
```

So now, if our level gets depleted of almost all its resources and the unit collects the last remaining ones, it will then go back to the nearest storage building and disappear when it reaches it.

And that's it! With those final additions, we've completed the brains of our collector AI and implemented all the expected behaviours for our units. We can run our game and see our little collectors spread about to gather wood and minerals, regularly returning to the closest storage location to deliver the goods and then going back to collect more, until the map is empty.

Final tree & possible improvements

For reference, here is a screenshot of the final collector AI applied to a few trucks, some for wood and some for minerals, that move about to find and store resources:

Figure 8.14 – Screenshot of the final scene with three collectors moving around to gather resources, and then bringing back them back to the nearest storage centre

And here is the final collector AI behaviour tree:

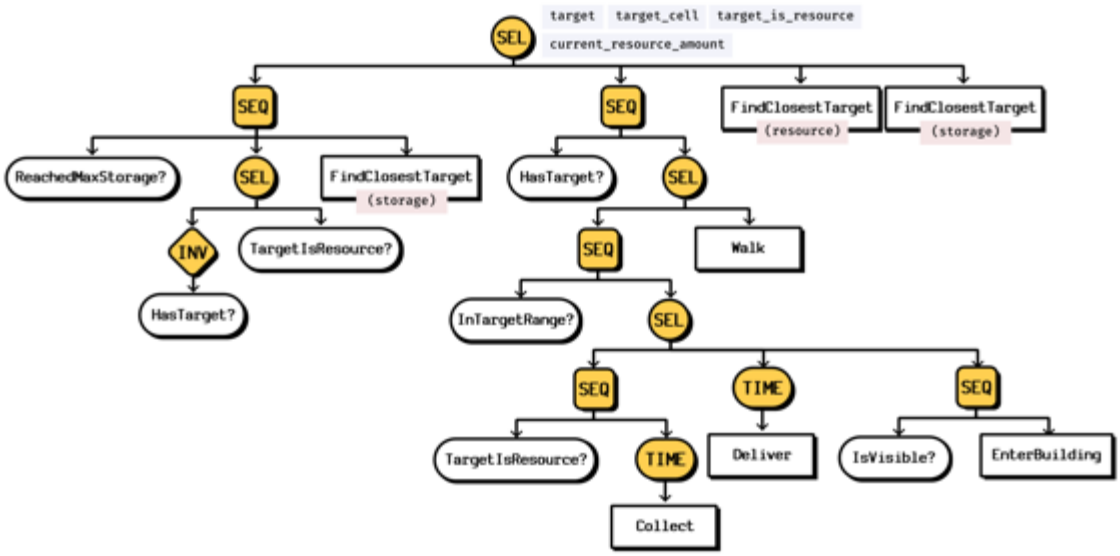


Figure 8.15 – Final V5 of the behaviour tree for our collector AI which integrates the home return mechanic with the sprite disappearance

As a final note, keep in mind that this was just a quick example of an application of behaviour trees to a real game use case. We could obviously think of many improvements to make this system even more comprehensive and robust:

We could compute our shortest paths by taking into account the ground obstacles to avoid very long detours.

We could use enums for the data field names to avoid misspelling errors, because as-is, we could easily make mistakes when writing those strings.

We could factorise a few bits of logic like the target reset that happens both in TaskCollect and or some constants.

We could even add some logic to our collectors to have them re-exit the storage building and switch back to the collect/deliver mechanics if resources re-appear.

YOUR TURN!

> Are you up to the challenge? Do you want to try to code up one of these improvements? Feel free to have a go at it and share your creations via email (at mina.pecheux@gmail.com), or even on the Github of this book (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) as an issue, so that other creators can react too!

Summary

In this chapter, we've implemented a basic RTS collector AI thanks to behaviour trees.

We first talked about Unity's 2D tilemaps and why they can be a nice tool for quickly preparing a sprite-based level, and we saw how C# enums can translate into easy-to-use dropdowns in the **inspector** panel for user-friendly setup.

Then, we discussed how to use the abstract Node and BTree classes from the BehaviorTree package we made in *Chapter 7* to create a basic search-and-walk behaviour for our units, to allow them to find and move to the closest resource on the map.

After that, we improved our AI to have it actually collect the resources when reaching the target tile, and fill up a resource bar until it reaches its max storage capacity.

Once this threshold has been reached, we've made sure that our collectors search for the nearest storage centre and go over there to deliver the goods and reset their resource amount. We also took this opportunity to see how to use a global event system to send notifications across the scene and update the UI display, so that the player knows resources were collected.

Finally, we implemented a "return home" mechanic for our units as a fallback action – so if the map ever runs out of resource, the collector will simply go to its nearest storage centre and "enter the building" to end its work phase.

Throughout this chapter, we therefore took advantage of various strengths of behaviour trees: we saw that they can be implemented very gradually, by slowly extending the logic with new sub-branches, and we were able to re-use the same nodes in different places to quickly add new actions. This is extremely useful when in development, because it allows us to easily prototype, assess and then refine the behaviour of our units in a scoped and well-controlled way.

And yet, despite all their perks, behaviour trees still share a common flaw with finite state machines: they have to be authored and built manually by the devs to really define the decision logic of the AI. Which is fine as long as you have enough time and people to handle this hand-authoring, but also inherently limits the possible complexity of your AIs.

So, in the next part of this book, we'll continue our journey in the world of AI programming and discover two other types of AI architectures that try and overcome this pain point – starting with a focus on the AI planners.

PART 4

PLANNERS &

UTILITY-BASED AI

9 - The reverse-thinking of planners

So far in this book, we've studied three techniques for implementing AI in our games: a single C# script for really basic logic (in *Chapter 2: Designing a single-script robot* the finite state machines for entities with clearly defined states (in *Chapters 3 to* and the behaviour trees for more modular and dynamic behaviours (in *Chapters 6 to*

Those tools can be very powerful when used right, and in lots of cases they are enough to model quite complex AIs. However, they still require us, as developers, to manually code up the entity's actions and reactions, and most notably the decision logic.

Which leads to the question: what if we could avoid this authoring phase? What if we could create AIs that are able to plan and build their own decision logic from a set of actions, without us having to prepare each transition or flow path? Could those AIs have even more realistic reactions, and lifelike behaviours, that perfectly fit any situation?

Realism as the ultimate goal: is it a good idea?

Before talking about this other AI architecture, I would like to pause for a moment and reflect upon this idea of creating realistic AIs.

'Cause, to be honest, up until now, in this book, we haven't talked that much about what "intelligence" we're really trying to make – except briefly in *Chapter 1: AI in* We've taken examples, and we've studied common tools for simulating "intelligent" behaviours, but we haven't really studied what this "intelligence" could be. And yet, it's half of the phrase at the heart of this book, **artificial**

So, now that we're a bit more familiar with game AI and the use cases it can be applied to, I want to take a few moments to get meta, and think about what it means exactly to give brains to a minion, from a design point of view. What is this reality we're trying to simulate, and is it indeed a valid goal in the context of a video game?

Should we try to reproduce reality?

Creating more adaptive and powerful behaviours looks like a logical evolution of the strict architectures games have relied

on for decades; at first sight, it seems offering more believable reactions that match the current context is the ideal goal.

But, now, let's come back to a previous discussion we had in *Chapter 1* on the trade-off between accountability and lifelikeness. Back then, we said that coding game AI was a difficult task because you had to emulate real-life the best you could... while maintaining a minimal level of control on the reactions of your entity. Because remember: you primarily want your players to enter and enjoy your world, and this immersion could be severely endangered if your AIs just moved about frantically with completely unpredictable reactions.

In that sense, making a dynamic and autonomous AI able to decide on its own might be risky. Indeed, if your AI devises its own plan on the spot to react to its current environment, how can you ensure to your players (and your worried team leads) that it will do something "rational" in the game's world? How you can assert that it will stay within certain boundaries and pose a demanding but fair challenge?

Sure, you're still the one who codes the possible actions of the AI, and despite what movies want us to believe, there is no ghost in the shell capable of inventing unscripted behaviour – so the AI will ultimately be limited by what actions you implement. But as we've seen throughout the last chapters, artificial intelligence is as much about the actions the entity can take as it is about the order and priority in which it can take them.

So should we search for a way for our AIs to build sequences of actions like we do, and prioritise them depending on the current situation to emulate all the subtle reactions and richness we find in real-life creatures? Or is it too dangerous because they will become unpredictable and unreliable?

I think there is no right or wrong answer, here – as is often the case in AI programming, it varies from one use case to the other, and you need to find our own balance between realism and control. Some game genres are more interesting when players can anticipate and analyse the computer's moves, while others get way spicier if the opponents react quickly and dynamically to the context to surprise you.

And... *can* we?

With that being said, even if we agree to relinquish some control and rather devote our resources to reproducing reality, there is still the question of how.

Artificial intelligence in games aims at reproducing via scripts a form of smarts found in the living creatures around us. However, as you are probably aware, the notion of "intelligence" isn't a clear-cut one-line definition; there are many types and levels of intelligence, and many ways of studying them.

In a really nice article titled

Creature Smarts: The Art and Architecture of a Virtual Brain

, Burke and al. (a group of researchers from the MIT Media Lab) discussed some of the core principles they used while developing their

C4

agent architecture and, in particular, how they abstracted the brain of a dog – who has a behaviour with a certain level of complexity, still easier to model than that of a human being!

Wanna read the article?

> For the curious, the article by Burke and al. is available for free over here: https://characters.media.mit.edu/Papers/gdc01.pdf.

Although it is just one possible abstraction of the matter at hand, it is interesting to study Burke and al.'s model to perhaps identify some tricks to creating more realistic behaviours for our AIs.

The **cognitive architecture** presented in the article contains a whole lot of mechanisms and systems all connected to each other to reproduce the dog's cognitive abilities with the highest possible fidelity, but we can simplify the gist as follows:

A creature perceives the world around it thanks to its senses. Usually, it should not be able to access the entire world model directly, it is not all-knowing – rather, it has its own mental

representation of the scene. This is what the article's authors call **sensory**

The stimuli the creature receives through its sensory system must then be with regard to its specific cognition and capacity to interpret these external events and give them true meaning. This perception also depends partially on the **beliefs and desires** of the creature – its internal goals drive and transform its mental representation of the world.

In the end, this second system therefore **classifies** the stimuli into various types, so as to make the brain ready to respond appropriately.

In addition to these external inputs, a real-life animal like a dog also has a "self-perception" (also called that makes it self-aware of its current position in the world, or even its current emotional state.

The mental representation the creature has of the scene is also heavily influenced by its **memory** – both the long-term memory, and the immediate working memory. This ability that living creatures have of remembering past events, past stimuli and past reactions – and more importantly past consequences of said actions – is a crucial thing for simulating realistic behaviours, because it directly leads to the notion of **learning and**

Memory also enables **predictions** on the part of the creature, since it allows for object persistence and extrapolations. These predictions might then come true, which would reinforce a connection between an event and a reaction... or, conversely, the world might actually evolve differently from what the creature anticipated, causing

These different systems eventually interact with each other to produce a personal, incomplete and emotionally-tainted assessment of the situation that the creature can use as a base of reflexion for its next move. Those final **actions** can themselves be defined by establishing: when the action should happen, how it should happen, what it should happen to and for how long it should happen.

And of course, some actions might depend on another system in the creature's brain, such as the navigation or locomotion system to perform a movement.

This interconnection of systems is undeniably hard to re-implement on a computer. We have yet to understand all the complexities of living brains and, if our modern neural networks have taught us one thing, it's that having a generalist AI capable of autonomous multi-tasking like us is impossible. We can train an algorithm to become an expert in road sign classification or job offer writing, but we can't have it desire,

make and carry a cup of coffee to its desk to then read its emails.

This is why, in game AI, we usually let go of this expertise and instead aim for a somewhat credible versatility. We try and mimic those senses, and those perceptions, and those internal goals, and have the players believe the entity is able to react to their presence and the current environment in some clever way...

Continuing our study of Burke and al.'s article, we can highlight a few elements that the researchers found key in maintaining the "illusion of life" in AIs:

In the context of a game, an artificial creature's internal turmoils and desires are only interesting if they are actually exteriorise and participate to the entity's behaviour. It is useless to give your unit a superbly complex emotional state if it has no means of expressing it and sharing it in some way with the rest of the world.

This expressivity relies primarily on the actions the entity can take, and the animations we show to support them. As we'll see later on in this chapter, in the *Discovering planning systems* section, animations can actually be paramount to coding a good AI.

Imperfect mental While re-coding a full sensory system and an overlay of perception filters on top might be a bit overkill, giving our entities a biased view of the world is an excellent method for making them more realistic. By limiting the field of view of a guard, or having it ignore some discrete audio cues, you ensure sensory honesty and, even better, you allow the AI to make

Which, like we said back in *Chapter* is a very efficient way of creating more lifelike behaviours since, as you'll recall: to err is human... (or human-like)!

Living creatures are full of surprises. To reproduce this unpredictability, our ideal model should therefore support and even encourage various behaviours. If there are two possible processes to achieve a certain goal, allowing the entity to pick one or the other with some level of randomness will make it less robotic (remember how we added a bit of stochasticity to our basic robot AI in *Chapter 2: Designing a single-script robot AI* for this very reason).

As first theorised by Rodney Brooks in his article

Intelligence Without Reason

---

a complex AI like a video game entity has to solve tasks at multiple levels. Specifically, reaching a goal can often be divided into two phases: first, deciding on *what* to do; second, thinking about *how* to do it.

This idea of a two-levels thought process is referred to as **supersumption** by Burke and al., and it derives from Brooks notion of **subsumption** that basically considers a high-level system focused on a global task (e.g. going to a specific location) which sends signals and orders to a lower-level system dedicated to more immediate and reactive tasks (e.g. avoiding obstacles on the way).

Implementing this kind of multi-level reasoning in a game AI is interesting because it emulates the long-term memory and short working memory systems we discussed before, and it gives the entity a more consistent behaviour thanks to this overarching "mission" at the higher-level.

Learning & An essential skill of living creatures is that, when confronted with a situation, they are able to memorise the sequence of actions and effects to some extent and then re-use this knowledge later on if they are exposed to the same problem. Adding this learning mechanism to an AI would significantly boost its lifelikeness, because we'd feel like it's a real evolutive entity that truly interiorises external events and incorporates them to its mental representation of the world.

From a developer's point of view, we would of course also like to have an **intuitive design process** that "[makes] simple things simple and complex things possible" (Burke and al.,

).

Alright – that's a long wish list, and let's be clear: we won't be able to implement all of this at the same time, and create a perfect AI that is also accountable enough to satisfy our game-context constraints.

Yet with this ideal AI in mind, we can now revisit the models we've studied in the past chapters of this book and try to identify where they fall short, and what we could do to improve them...

(As a side-note: this section is obviously a crude summary of a really great article, so I encourage you to check it out and savour all the details. But I think it is interesting to consider these notions at this point of our journey, even slightly summed up like this; after all, we have looked at several classical game AI examples, and we are now searching for alternative architectures capable of producing more believable behaviours...)

A quick comparison with our previous models

If we consider the basic AI we created back in *Chapter* or the FSMs and BTs we've studied since then in *Chapters 3 to* we see that our architectures were quite simplified compared to

this cognitive model. In particular, we didn't differentiate between senses and perceptions, and we didn't have any interior beliefs to guide the actions. Apart from the pushdown automata we talked about in *Chapter 5: Upgrading your finite state* we didn't even have any memory!

When you think about it, it looks like we didn't have a lot left, actually.

So far, the architectures we've studied had us code up the decision logic of the AI, and decide of its goal ourselves; in short, we took it upon our shoulders to manually setup its sensory system, its perception system and its beliefs system, and all it had to do was run the action we told it to at the right time.

Don't get me wrong: it was still AI, since those actions were taken automatically by the system without us actively pressing a key or typing in an input to trigger them. But it was clearly a scoped form of AI that was fairly pre-determined in its reactions, and that couldn't invent new tricks by itself.

We did re-implement one really cool thing from Burke and al.'s cognitive architecture in our previous use cases, and that's the sensory honesty. Checking for enemies in a specific radius based on the entity's field of vision (like we did in *Chapter 4: Making a simple guard patrol* or keeping an eye on your own

local amount of resources (like in *Chapter 8: Implementing a RTS collector* is a way to reproduce the limited perception of external stimuli, and the proprioception of the dog's brain.

Still, that is clearly not enough to reach the level of intelligence described in this article. Single-scripts become intractable the moment you start to pour in too much logic, state machines lack the flexibility of a real entity, and behaviour trees require a lot of authoring from the developers – which essentially means the size and complexity of your BT-based AIs is going to be related to the size of your dev team. If we wish to build more powerful and adaptable artificial brains, we have to invent new models that dodge these difficulties!

Discovering planning systems

As we've said in *Chapter* artificial intelligence has been around for quite some time, in various forms. From the initial expert systems to our modern deep neural networks, the field has grown in many ways, and it has now entered our everyday lives.

Even focusing solely on AI for games, we can see a huge evolution in the tools and techniques leveraged by developers. From the first hard-scripted enemy waves of the

Space Invaders

shoot em' up to the more recent FSM-powered soldiers in

Half-Life

, or the behaviour tree-based Xenomorph of

Alien: Isolation

, we've seen our AIs become more and more lifelike as time went by.

But you know developers: they like to proudly code what no man has ever coded before! Plus, even after putting FSMs and BTs in the spotlight, and making them the go-to AI architectures for a while, they still felt frustrated with the limited level of realism they managed to reach with those tricks.

So they kept on searching for alternatives, and new architectures. And over the years, they came up with other ideas, to hopefully get more robust, adaptable, reactive and expressive behaviours. And one of those AI architectures was the **AI**

From automated planning to the GOAP architecture

In the mid-2000s, parallel to the

Halo

franchise popularising the behaviour trees in the world of game AI, another other idea started to emerge for modelling the reactions of NPCs in games when the **Goal-Oriented Action**

Planning architecture, or created by Jeff Orkin and his team, made its first appearance in

F.E.A.R.

.

Derived from a pre-existing AI technology known as **automated** or AI planning, this new architecture aimed at offering more adaptable and reactive AIs, while at the same time lightening the load on the AI developers.

So, what is automated planning?

The idea of planning doesn't date back to the 2005

F.E.A.R.

game – conceptualising a logical series of steps to achieve a certain goal given a specific context is an old trick in the book. And applying this method to automated systems such as autonomous robots or unmanned vehicles started back in the 1970s. Automated planning was initially used in robotics and astrophysics for example, to help control the turbines of large power stations, or to help rovers navigate the surface of distant planets.

Simply put, the idea behind automated planning is the following:

We want to implement an algorithm capable of reacting to changing unknown environments by adapting its strategy rapidly. This strategy should be made on the spot by the program, and fit the current situation as best as possible.

This implies we need to describe the current state of the world with some specific language or token set, to properly characterise the context the agent is evolving in. At any given point in time, this description should contain two types of objects:

1. Fact: A small piece of information, scoped and localised (e.g.: “Door A is closed”).

2. State: The collection of all the true facts at a given time.

Using that description of the world, the agent should thus be able to build a **planning action** to reach a specific goal while taking its environment into account. Such a plan includes:

1. Objects: All the elements required for/involved in the action.

2. Preconditions: All the facts that must be true for the action to work.

3. Effects: All the consequences of completing this action on the world state.

By establishing a set of facts and rules about the world, we can therefore give enough information to our entities to have them react in a valid way. And if we also give them a goal to achieve, then they'll have all they need to make and execute a plan, emulating a conscious and motivated reaction to their current situation.

Of course, contrary to the cognitive architecture we discussed in the first section, our AIs aren't real living things, so they don't have beliefs and desires to follow. Thus they won't spontaneously follow internal goals, but rather need to be given **external goals** to then start planning.

In its simplest form, classical AI planning usually also relies on two important assumptions, which is that actions are and that the agent has a **perfect knowledge** of the world. This contrasts quite a lot with the sensory honesty we talked about previously, and it certainly downgrades realism, but it also helps with the predictability and accountability of the system, since we at least have a clear view of what the AI bases its planning on. By removing this complete observability or adding stochasticity in the actions, it is possible to increase lifelikeness... at the cost of an increased difficulty in the planning problem.

In all generality, automated planning also assumes that each agent is oblivious to the other agents around it, if there are any. So everything happens as if the agent was alone in the world, and could only see the ripples of the others' presence by observing the effect(s) of their actions on the world state. This of course has heavy implications on the implementation of an AI planner, since it means the entity has to regularly check if its current plan is still valid or if, in the meantime, some other mysterious external force has transformed the world and made one of the preconditions false (e.g. in layman's terms: "another unit closed Door A"). With this philosophy, any sort of collaboration between the entities is also just faked by having them solve the same task at the same time – but each unit still believes it is the only one.

GOING BEYOND CLASSICAL PLANNING

> If you're interested in learning more about advanced AI planning models, you may want to dive deeper into those common extensions:

> **- Contingent planning**: This assumes the entity perceives the world through a set of sensors (or senses) that may be faulty. Instead of a sequence of perfectly defined actions, the agent then piles up multiple decision trees, that require it to choose between multiple possibilities at each step of the plan execution.

- Conformant planning: Similarly, the agent could have no information on the environment whatsoever and be reduced to guessing based on its beliefs. The plan would then be a sequence of actions, but with way more possibilities since there is less data on the task to perform.

- Preference-based planning: In addition to planning for a specific objective, the agent could also try to satisfy its own personal preferences, for example by looking for some additional rewards along the way.

- Multi-agent planning: To dodge the issue of each planning agent being isolated in the world, some research focuses on how to make decentralised systems with a distributed logic that could support collaborative behaviours.

Now that we are familiar with the fundamentals of AI planning, let's focus on one of its earliest and most famous implementations – the STRIPS model.

The STRIPS model

In 1971, two scientists named Richard Fikes and Nils John Nilsson proposed one of the early famous automated planners: the **Stanford Research Institute Problem Solver, *aka*** Since then, this algorithm and its associated description language have been derived into many other models, and most of the languages used to express automated planning problems today still are direct descendants from STRIPS.

Moreover, STRIPS introduced a core idea for AI planning, which we could describe as Basically, the algorithm doesn't blindly stack random in its plan actions in hope of achieving the given goal; instead, it works **backwards** from the goal and searches for means of accomplishing this objective, piling up actions to take or sub-goals to reach, until it has converged back to the current world state. Then, it simply executes the plan by traversing this stack from top to bottom.

(By the way, if you want a quick reminder on the stack data structure, we've talked about it in *Chapter 5: Upgrading your finite state* in the *Pushdown automata* section.)

Executing the plan and reaching the final goal is thus a matter of pulling out all the items in the plan stack one by one to gradually solve all the intermediary sub-goals, and eventually achieving the original objective.

Despite being quite simple, the STRIPS model encompasses all the fundamental principles of AI planning and it has influenced a lot of the research in the field afterwards – one of the recent offsprings of this architecture being the GOAP model.

Why turn STRIPS into GOAP?

When he worked on

F.E.A.R.

's AI in 2005, Jeff Orkin went for a STRIPS-like system, but with some twists and shouts here and there. There had to be some adjustments because, at its core, automated planning is quite bad at **micro-management** and therefore cannot be applied directly to game AI programming.

To really understand why, and how we need to adapt automated planning in our context, let's take a very basic example. Suppose we have a guard standing in Room 1, and we task it with going next door to Room 2. In our model of the world, we'll describe that:

There is a doorway between those two rooms blocked by a door, Door A.

A unit cannot walk through a closed door, the door has to be open.

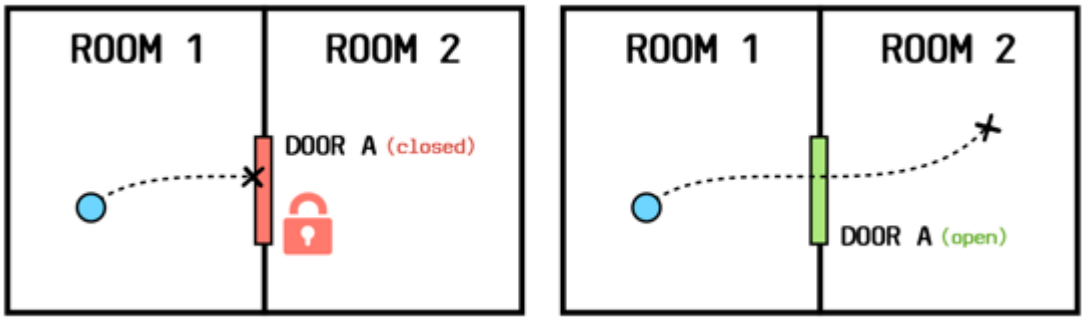


Figure 9.1 – Visualisation of the two rooms and the door in the middle: the unit (in blue) can only walk through the passageway if the door is open

If it is currently in Room 1 and standing near the door, then it is then possible for our character to build a plan to move over to Room 2 that consists in two actions: first, opening the door and changing the state of the world so that now Door 1 is open; second, walking through that passageway to enter Room 2.

Figure 9.2 shows a representation of the situation with the evolution of the world state and the various objects, preconditions and effects of each action in the AI's plan:

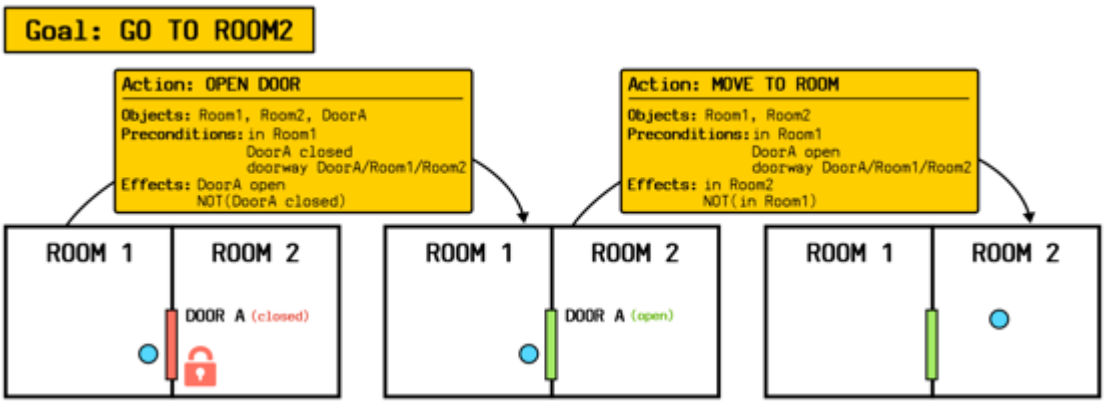


Figure 9.2 – Evolution of the world state as the unit executes its plan to go from Room 1 to Room 2, with detailed intermediary actions

It is important to note that, while the guard is aware of many things here – for example, the existence of a route from Room 1 to Room 2, even though it can't initially see it –, the state we describe here is still relative to this entity, and this entity only, because it is unaware of the other agents. (This is particularly visible with the preconditions and effects of the form: in since this obviously applies just to this specific unit.)

a quick side-note on applying effects

> You'll notice that in *Figure 9.2*, our "door opening" action doesn't simply set the DoorA open fact to true, it also sets the DoorA closed fact to false. This might seem like a duplicate but, in truth, in the STRIPS logic, there is nothing that formally constraints a variable to a single value. So forgetting to "unassign" the DoorA

> closed fact would lead to an inconsistent world state where both DoorA open and DoorA closed are true.

Now, this seems fine, but it does raise a question: how is our guard going to move around and open the door I mean, in terms of game programming, what instructions should we put in our scripts to actually execute this brilliant plan? And what would happen if it is not near the door – how would it know it has to first get near the door to be able to open it?

See, that's the micro-management issue I was talking about before: while it is great for formulating adaptive and dynamic plans, an AI planner like STRIPS lacks precision when it comes to running the sequence of actions in an environment as complex as a game. If we reconsider the supersumption idea we mentioned in the first section, it's like we only get the high-level brain, and no low-level system to truly make the required actions a reality.

To solve this issue, the

F.E.A.R.

---

developers decided to modify the STRIPS structure, which led to the new Goal-Oriented Action Planning model.

Getting into GOAP, finally!

As explained in Jeff Orkin's 2006 GDC conference,

Three States and a Plan: The A.I. of F.E.A.R.

, the primary motivation behind the GOAP architecture was to ease the burden of the single AI developer in the team, who had to give brains to a whole bunch of non-playable characters in an over-the-top intense action-packed game. So, as Orkin puts it: "If the A.I. are really so smart, and they can figure out some things on their own, then we'll be all set!".

CHECK OUT THE ARTICLE!

> If you want to dive into the details of the GOAP architecture and read about it in Orkin's words, his conference was accompanied by a paper version that is available for free on his website, over here: http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf.

This necessity to manage complexity meant that the developers couldn't just use finite state machines (because of their exponential growth of states) or behaviour trees (because of their manually authored logic flow), and they had to find a way to make the system as **"self-constructive"** as possible.

The

AI team therefore turn itself to automated planning, and more precisely the STRIPS model. They knew they needed to change some of it to use it in a video game, because this architecture wasn't able to micro-manage a character and have it move in a real 3D scene; so they decided to make their own version of this behaviour model, and ended up with the **Goal-Oriented Action Planning** system.

The GOAP architecture relies on two core elements:

Action Each unit type in the game is given its own list of possible actions, called its action set. These actions may range from going to a specific location to patrolling an area, attacking the player, taking cover... The full database of actions in

contains 120 actions that designers can then browse and cherry-pick from to determine the action set of a specific type of entity (and thus the possible level of complexity of its plans). The GOAP system therefore accommodates for a wide variety of behaviours that are just various recombinations of a shared pool of possible actions.

Figure 9.3 shows different action sets for the Solider, Assassin and Rat AIs ('cause, yes, even rats make plans in this game!) in the studio's

GDBEdit

tool (the image comes from Orkin's paper):

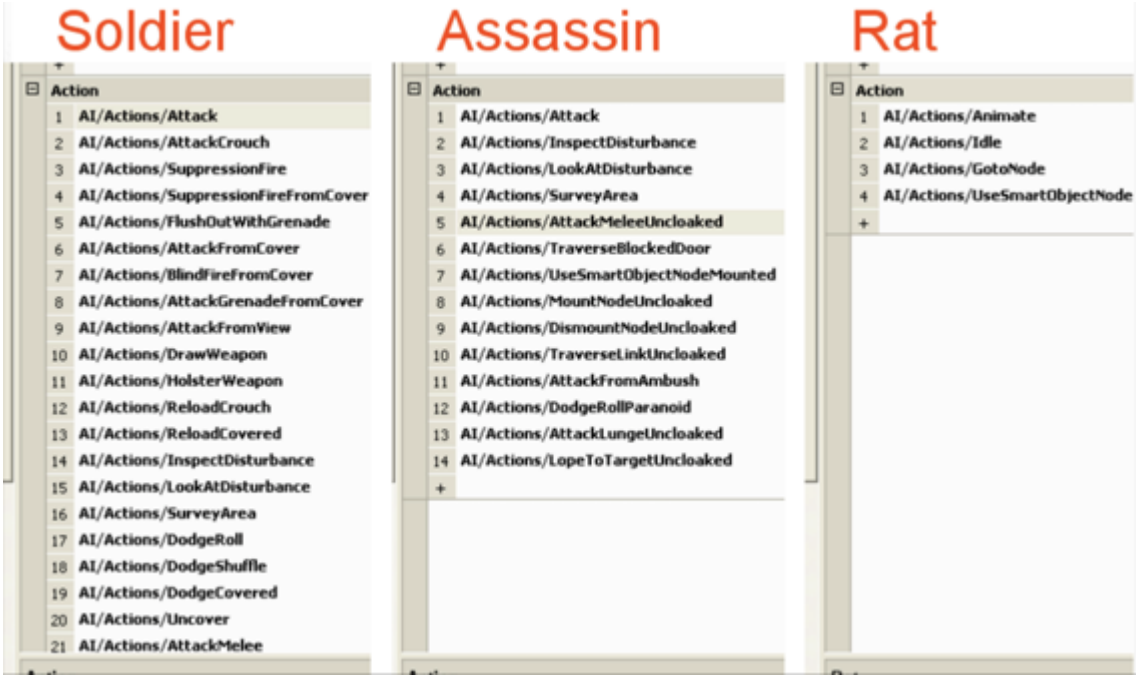


Figure 9.3 – Three different action sets for F.E.A.R. unit types in GDBEdit (image from:

In order to trigger planning and an actual movement or animation on the part of the AI, it also needs one or more goals. Those objectives are the key to bringing the entity to life – if it doesn't have any goal, it won't do anything.

In

F.E.A.R.

, the characters can have multiple goals at the same time, but those goals are **prioritised** depending on the current situation. For example, a soldier may have both a “patrol” objective and a “kill player” objective, but if the player is nowhere nearby, the “kill player” goal will just be downgraded in terms of priority and ignored for the time being.

Once it has a goal and an action set, the AI can then start formulating plans to achieve its goal by slowly transforming the world state.

However, because, as we said before, classical AI planning models create agents in a vacuum, meaning that agents are unaware of one another, executing plans isn’t as easy as just running through the prepared list of actions. To account for the potential external changes in the world state made by others, the planner has to continuously **revalidate** its plan while it’s executing it. The GOAP architecture thus (re)validates plans at three crucial points:

When the plan is first formulated, it is immediately validated, or else replaced with an alternative. To do this, the AI creates a copy of the current game state and executes the proposed plan on it, to check it will indeed lead to the expected new game state.

Then, if the entity is not currently playing a non-interruptible animation, it will constantly ensure that it does not need to replan.

When the AI pulls out an action from its plan sequence and wants to run it, it first rechecks all the preconditions are still valid and, only then, runs through the action.

Any of these three check points might force the entity to re-evaluate its plan and fallback to another series of actions to reach its current goal – or, even, default to some idle inactive behaviour if there is no way to achieve the goal anymore. If the unit does manage to successfully complete its plan uninterrupted, then it eventually pops out the current goal and is given a new one to replan for. The AI is therefore **always planning** for both failures and successes, which gives it an unprecedented level of **lifelikeness and**

Note that this does not mean plans have to extremely complex, or contain a lot of actions. Typically,

's AIs usually execute up to two actions, and then get assigned a new goal, which matches the quick-pacing of the game quite well. In a different game genre such as turn-based strategy, you might want to encourage longer plans but, for a FPS like

here, having the AI reach its goal and replan regularly helps maintain the high level of tension wanted by the designers.

BEWARE THE RATS...

> Of course, this constant re-processing of the possible plans isn't cheap, and F.E.A.R. actually had some performance overhead in a few levels where the number of AIs was too high.

> As studied by Eric Jacopin in his 2014 article Game AI Planning Analytics: The Case of Three First-Person Shooters (https://ojs.aaai.org/index.php/AIIDE/article/download/12728/12576), the game sometimes incurred slowdowns and performance issues because of the numerous AIs in the level all replanning continuously, among which… the rats. Especially because a lot of the planning didn't consider whether the player was actually around to watch the entity – so some of the rats encountered at the beginning of the level would still be executing and reformulating plans ten minutes later while you were on the other side of the level!

From a more technical point of view, it is also interesting to study how the GOAP model differs from the original STRIPS, and how the

---

F.E.A.R.

AI team managed to make it compatible with the context of a video game. The first question being: how did they handle the micro-management issue we discovered before?

In fact, to allow the AI to go through its actions and run the right behaviour as it follows the plan, the GOAP architecture

relies on a very simple finite state machine, composed of only *three* This might surprise you since, as we saw in *Chapters 3 to* FSMs are hardly ever that simple, and certainly cannot describe that complex a brain unless they are an unbearable mess of numerous states and transitions.

And yet, *Figure 9.4* shows the state machine of the AI in F.E.A.R., shared in Orkin's article:

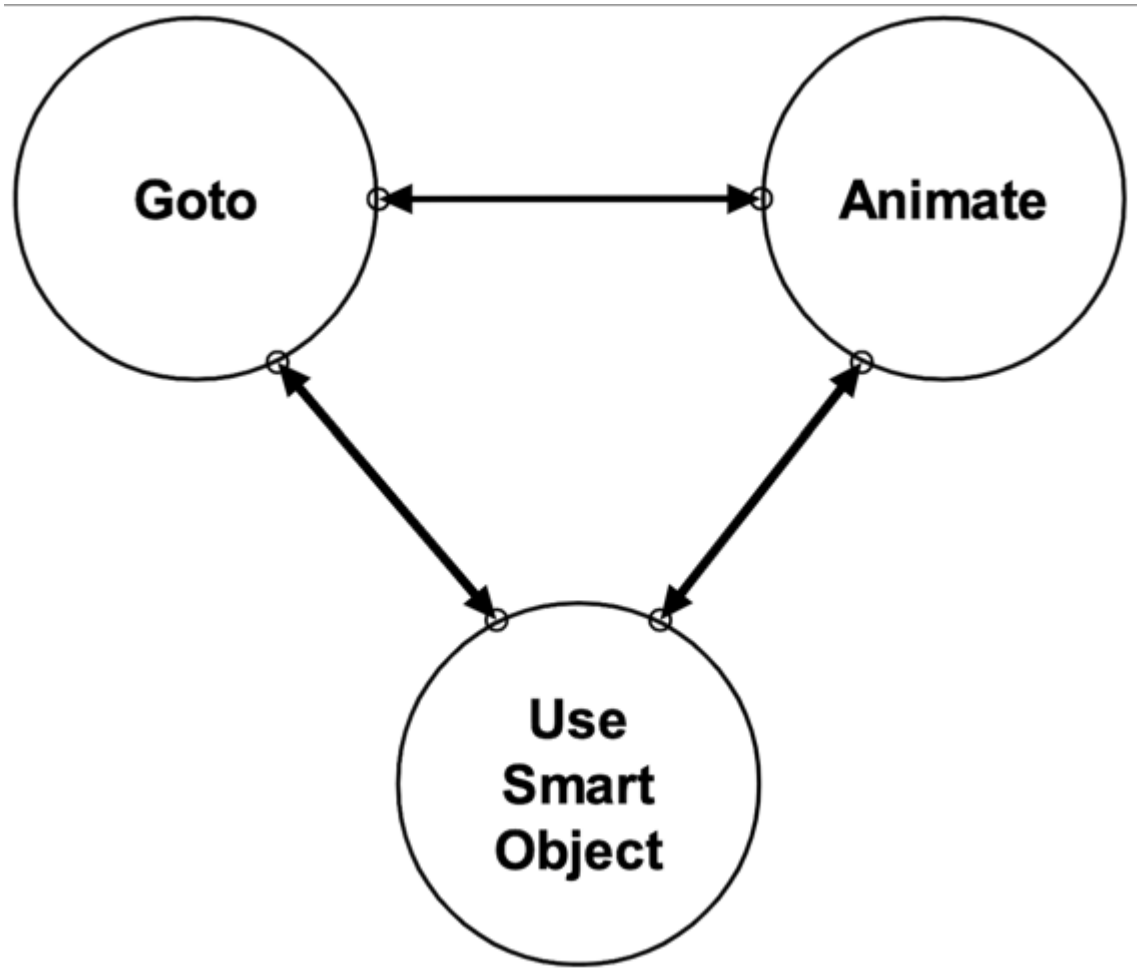


Figure 9.4 – Finite state machine running the micro-logic of all the AIs in F.E.A.R., composed of only three states (image from

The trick here relies on a brilliant observation made by Orkin and his teammates, which is that any AI action in a video game can be actually be defined as an Some animations imply also moving the 3D model (this is the Goto state in *Figure* others imply interacting with an object like a lever, a door or a chair (this is the Use Smart Object state in *Figure* and finally, a third category is just the animations that run from start to finish in-place, such as reloading a weapon (this is the Animate state in *Figure*

But, at their core, all of those are just about playing a clip on a 3D model.

Having a GOAP entity execute its plan in the context of the game thus becomes more a problem of picking the right state among those three, and loading in the right animation on the model to execute the required action. In

F.E.A.R.

, this is done simply by declaring in the action code the related animation, as a direct application of the **data-driven philosophy** we discussed in *Chapter 6: Understanding behaviour*

USING ANIMATIONS AS EVENT SOURCES

> To go even further, in modern game engines, it is also possible to hook **events and signals** to model animations, and thus embed extra logic points inside the timeline. This way, starting an animation might indirectly triggers other changes while it's playing. For example, you could play a gun reload animation and, in the middle, emit an event to update the current ammo count of the AI, and have it say a little dialogue to warn its colleagues it's going to start shooting again.

Besides using this three states-FSM, the team also made four big changes to the old STRIPS architecture:

Actions have specific **costs** which help prioritise plans if the AI happens to find multiple ways of reaching the same goal. To find the less expensive series of actions, the GOAP model uses the famous A* pathfinding search algorithm (which we already peeked at in *Chapter* to compute the shortest path to the goal, given the current state of the world.

'Cause while the A* technique is often applied to navigation problems in a 2D or 3D scene, it is actually far more generic than that and can find the shortest path in any node-based graph. So by building a graph where each node is a state of the world, and the edges are the actions the entity can take to go from one state to the other, you get a weighted graph that

represents all the possible plans to achieving a specific goal. This graph contains all the intermediary evolutions of the world state, and the cost of the actions in this plan are directly used as the edge weights. Thus solving the shortest path problem with an A* instance in this graph is equivalent to finding the plan with the minimal cost.

Having the costs of the actions evolve throughout the game, or tuning those weights based on some particular emotional state on the part of the entity can simulate the system of beliefs and desires we talked about in the *Realism as the ultimate goal: is it a good idea?* section, and it can help with producing various reactions even if the entity is faced with the same situation to avoid monotonous robotic decisions.

The system relies on a global **fixed-size array** to represent the current world state, shared between all entities, with one slot for each variable in the state. These variables may of various types (booleans, enums, numbers...) but using this fixed structure instead of the informal state description of the STRIPS model makes it easier to switch variable values.

A drawback of this decision, though, is that if you have a list of elements where only one can be active at a time, you have to find a way to explicitly designate the active item, since you can't just add or remove it anymore. Typically, if the entity has an inventory with multiple weapons available but just one

currently active, it has to plan according to this weapon and not the other ones.

In addition to the variables stored in the fixed-size array that partially describe the world, GOAP also allows actions to compute some **procedural preconditions** on-the-fly, when need be. This allows to not continuously check for pieces of the world that depend on a computationally expensive process, but rather run these checks on-demand, and apply this additional filtering at the right time.

For example, an AI that can technically flee to safety but has currently elected the attack goal has its primary objective has absolutely no need for computing a navigational path to the safe zone. This expensive operation should therefore be ignored until it is relevant for the decision logic of the AI.

As a sort of mirror, the GOAP model also has **procedural** which make it possible to delay the result of an action on the world state. Thanks to those procedural effects, instead of instantaneously updating the fixed-size array and thus the shared world info, it is possible to wait for an animation to finish, or for the unit to arrive at its destination point, before actually modifying the data.

In the end, the GOAP architecture is therefore quite simple and straight-forward :)

We have actions and goals, and we use a high-level planning system to chain actions in the right order and transform the state of our world until it matches the desired objective – and we can choose which plan is the best among the different possibilities by resolving an A* shortest path problem on a well-defined abstract graph. Then, to truly execute the actions we've elected as the best, we can run different animations on our entity's model that may move it in the world, or have it interact with the environment in some way, or simply proceed in-place.

To ensure the plan stays valid, we need to regularly recheck the current state of the world against the necessary preconditions of our planned actions, or else replan another series of actions to reach our goal. This is particularly important to incorporate the effects of other agents on the world in our incomplete mental representation of the environment.

Alright – now that we know how it works, let's quickly go through the main advantages and the drawbacks of this alternative architecture, and discuss to which extent the GOAP model is a valuable technique for coding robust and complex game AI.

Understanding the strengths and the dangers of the GOAP model

GOAP clearly had a lot of potential from early on, and it's no wonder it has now joined the ranks of common game AI tools, alongside the state machines and the behaviour trees. With the AI of F.E.A.R. still being cited as one of the most impressive to this day, AI planning is now an appreciated technique for modelling the behaviour of non-playable characters.

Three key perks of the GOAP architecture that Orkin highlights in his article are:

Decoupling goals and Because each character type has its own action set, it is possible to create unique behaviours without having to create new building blocks in the project. The way to reach a goal isn't embedded in the goal itself, thus each unit is free to satisfy the goal the best it can – and you don't have to define specific "action-goal" combo for each type of entity in your game.

This decoupling also implies that the information about the entity isn't stored in any particular "state" object (like in FSMs), but is instead accessible from any goal or action. The unit therefore has a local data store that acts as a working memory shared between all actions and goals. This allows for

faster reactions and accurate re-planning, because you don't have to first exit some state to clean up your entity's context, and then re-enter another one to setup the right variables.

Layering Similar to behaviour trees, this planning approach lets developers iterate and build the AI's logic incrementally. By stacking and prioritising new goals in the entity's brain, and then injecting the right actions in its action set, you can essentially boost the character's abilities bit by bit, with a high level of control.

Because the system auto-computes the transitions between each behaviours when aiming for a goal, expanding the unit's logic is just about throwing in new goal and action objects, and the AI takes care of the rest. Plus, it's less risky to add behaviour than in a state machine, for example, where you could always inadvertently break a previously coded state by messing up a transition condition or a clean-up phase.

Introducing **dynamic problem** Finally, because the decision logic is built at runtime by the AI, it can re-adapt dynamically to better fit the new world state after an update. This capacity to replan and re-evaluate the possible reactions makes the AI more robust to sudden paradigm shifts than with our previous models. There might be some cases where the entity can't do anything and has to stay idle, but usually it should be able to

come up with a new plan, be it a long and intricate one, or a short two-actions sequence.

This planning technique is also interesting because it allows the AI to learn and gradually incorporate new information on the world to its own mental representation, and have it influence its future reactions.

For example, if the entity has planned to move in the direction of the player but the player closes the door and blocks the way, then the entity can re-plan a new series of actions where it tries to use its key on the door to unlock it and restore the previous state of the world. But if this is impossible to do because the player has hacked the system to shutdown the opening mechanism, the AI won't just repeatedly retry the same action – instead, it will take this new piece of data in its description of the world and re-plan according to it, which could lead to the entity jumping through the window to force its way in the room.

In a previous paper from 2003 named

Applying Goal-Oriented Action Planning to Games

, Jeff Orkin also described how the GOAP architecture always produces valid plans, and is therefore interesting in terms of correctness, compared to a manually authored logic that may contain errors (of course, that's assuming you properly defined the preconditions in each of your actions...).

AI planning, and more precisely the GOAP model, thus seems to encompass most of the advantages of the FSMs and the BTs. It significantly reduces the workload of AI programmers since they don't need to hand-author the decision logic of the entity anymore, and it lets them focus on more high-level design considerations.

IMPLEMENTING SQUAD BEHAVIOURS

For example, liberated from having to deal with the decision logic and immediate reactions of each AI in the system, F.E.A.R.'s AI team was able to spend more time on the design and scripting of squad behaviours to coordinate entities at a more global level and give them consistent behaviours as part of the group, based on their current proximity. In particular, the author points out how, sometimes, the self-constructive AI planning at the entity-level translated to surprisingly realistic **emergent behaviours** at the group-level, such as an AI flanking the player or apparently coordinating with another to surround their opponent (even though it's truly just each planner looking for the closest cover point, or the best way to get a good line of sight).

The article also discusses the importance of integrating communication in those squad behaviours to externalise and express this coordination in the eyes of the players, typically by having the soldiers in the same squad talk to each other and send various audio cues about their current state of mind. This emulates

> intention and helps the player understand some of the remaining quirks in the AI's behaviour – like an inactive soldier that vocalises the issue by saying: 'I can't find any good cover!', or a variation thereof. (It even allowed the game makers to trick the players into thinking the AI did broader-scoped actions, such as a unit shouting it's calling reinforcements… when there was absolutely no reinforcement mechanics coded in F.E.A.R., and it was just the player assuming the enemies encountered later on were born out of this previous order!)

> Once again, if you want to learn more, don't hesitate to have a look at Orkin's amazing article:
> http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf :)

However, it is crucial to understand that this system is also **highly unpredictable** and thus, in a sense, Although the action set of each unit is limited, it can still combine all of these actions in very diverse ways... and more or less rational plans. Even worse, getting a consistent behaviour can be difficult, and you may end up with entities that regularly act "out of character", and surprise the players in a *bad*

These unexpected bursts of folly might endanger the **willing suspension of disbelief** that allow players to immerse themselves in your world and get impacted by the events on-screen, because they can blatantly expose the ugly truth that

despite all of these smoke and mirrors, this enemy is just an artificial creature living about in a video game.

If you start to increase the number of AIs in your game and create large packs of enemies, you may also be confronted to the **performance issues** mentioned in Eric Jacopin's article. This efficiency issue, mixed with the randomly weird behaviours, means that applying the GOAP architecture to big groups of NPCs is risky, and probably won't go as smoothly as you'd like.

At least, it didn't for the creators of the

Transformers: War for Cybertron

---

game who, after implementing their AI system based on the GOAP architecture, were eventually forced to hardcode some action sequences by hand at certain points of the story to enforce a rational and consistent behaviour in their entities, and properly handle the heavily-populated scenes! Although it worked overall and made the enemies fairly challenging in the game, this intelligence modelling was clearly not ideal, and even the designers felt like they couldn't update and tweak the AIs' logic the way they wanted.

Which is why for the next game in the series,

Transformers: Fall of Cybertron

, the team at High Moon Studios decided to regain a bit of control on their AIs by leveraging an alternative AI planning

model, called HTN.

Levelling up to hierarchical task network planning

A QUICK DISCLAIMER

> The story I narrate here about the Transformers game series comes from Tommy Thompson's amazing video on the AI of this franchise: HTN Planning in Transformers: Fall of Cybertron | AI and Games (https://www.youtube.com/watch?v=kXm467TFTcY). If you're curious about the inner workings of popular game AIs, or if you want some summaries on the well-known AI models used in the industry, then be sure to check out his channel, AI and Games!

After seeing how well the GOAP architecture did in F.E.A.R. in the mid-2000s, the idea of using AI planning in video games started to grow in the community. This tool began to spread, and other studios incorporated this new technique to their toolbox, for example the Bethesda Softworks team with Fallout 3 in 2008, or Eidos Interactive's developers for Deus Ex: Human Revolution in 2011.

But for some titles like

Transformers: War for Cybertron

, the GOAP model fell a bit short. The architecture required too much manual re-corrections to really be viable, it demanded a lot of resources to manage the large packs of units, and the AI designers disliked how unpredictably and inconsistently their entities behaved.

Eager to find a better solution for their next title, the creators thus looked at the other alternatives in the game AI landscape, hoping for something that would still be along the lines of AI planning and enable them to keep this high reactivity and flexibility that GOAP provided. Enter **Hierarchical Task Network** or

A brief history

Although this AI architecture was already discussed in the 1980s in academic research, hierarchical task network planning wasn't really applied to the video game industry until Guerilla Games used it for their 2009 title,

Killzone 2

.

As explained in a 2009 presentation from the Game AI Conference by the team (free over here: the creators at Guerilla Games wanted to make the bots in their FPS even better than the ones they had in

Killzone

, and in particular improve the multiplayer experience. For that, they relied on a multi-level AI system with a global Strategy AI capable of giving grand orders to the AI troops, then a more scoped Squad AI that could retranslate those orders at the local level, and finally an Individual AI that took care of carrying out these local orders.

And, for the Individual AI, they went for a HTN architecture, drawing inspiration from the previous research shared by Kutluhan Erol and al. in an influential 1994 article:

HTN Planning: Complexity and Expressivity

.

WANNA CHECK OUT THE ARTICLE?

> Once again, this article is available for free online over here: https://www.cs.nmsu.edu/~tson/classes/spring04-579/HTN-planning.pdf – don't hesitate to have a look if you want more details :)

Throughout the development, the

Killzone 2

AI team decided to follow one principle: “[build a strategy] with a goal-driven approach by separating ‘what’ to do and ‘how’ to do it” (quote from the 2009 presentation).

Remind you of anything? Yep – that’s the supersumption idea we discussed earlier, and that’s how the team managed to create rational and interesting behaviours for their bots, even with varying bot group sizes and changing objectives!

So, now that we have an idea of where it comes from, let’s see how HTN planning works exactly, and how it helped the developers of

Killzone 2

and those of

Transformers: Fall of Cybertron

boost their AI systems even more...

The basics of HTN

Hierarchical task network planning focuses, as its name implies, on building a network of tasks. In a nutshell, this architecture, derived from the STRIPS and GOAP models, tries to re-inject more control in the design authoring by structuring actions into higher-level behaviours.

The core idea behind HTN is to pack together GOAP-like small individual actions into macro-actions, called to create more resilient and overall logical plans.

This model relies on two types of tasks:

Primitive The traditional base actions, like in a STRIPS or GOAP architecture, that represent a small-sized bit of logic, with preconditions and effects.

Compound An assembly of one or more primitive and/or compound tasks.

Thanks to this **hierarchical and recursive** structure, HTN planning allows designers to build and maintain complex AI behaviours that still possess the flexibility of an AI planner (since compound tasks are stacked automatically in a sequence by the planner to reach the given goal), but also provide more control over the exact reactions (since compound tasks are hand-authored pre-made small action packs that make sense on their own).

For example, let's assume we want to model the behaviour of a guard NPC that can take cover, throw grenades and fire at the player.

With the regular GOAP architecture, we would define these three actions in the character's action set, assign it an "attack player" goal, and ask the AI to formulate a plan using those actions to solve the problem. By checking the preconditions and effects of each action, the guard could eventually end up with something like: jump to cover, and then attack from behind this crate. Or throw a grenade, and then fire a few shots. Or just throw the grenade.

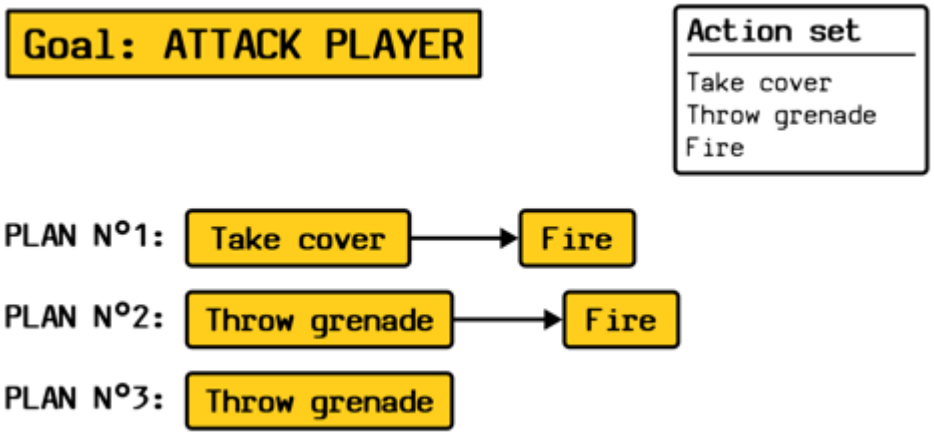


Figure 9.5 – Visualisation of some possible plans for attacking the player, in the GOAP mindset

While it is nice to have this variety for our AI, it also means that it is hard to predict what it's going to do. And if we start to add more actions to the action set, such as yelling to warn the other soldiers around, the possibilities quickly expand. The annoying part is that, in some cases, it could be hard to

choose between two actions that lead to the same effect: how indeed should the AI distinguish between directly firing at the player, or asking its colleagues to do so?

As we've said earlier in this chapter, expressiveness is the key to communicating the personality of the AI to the players – in this context, *how* you do things sort of blends with *why* you do them, because, as players, we don't have any other way of knowing the inner thoughts of these artificial creatures.

The GOAP model leaves the door open to a lot of expressiveness... so much so that AI designers may feel like they're not really designing anything, and they end up just praying for the best.

HTN can be a solution to this unpredictability issue. With hierarchical task network planning, an AI designer can prepare a little self-contained chunk of behaviour as a compound task, "Ambush player", that contains the specific actions the AI should trigger when faced with that situation:

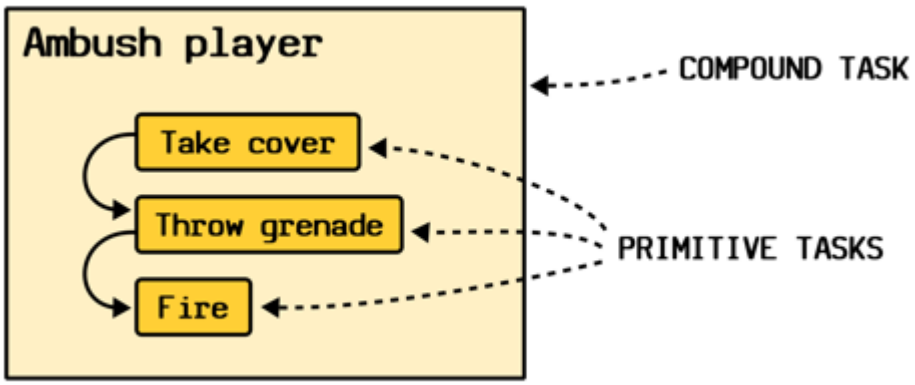


Figure 9.6 – "Ambush player" compound task containing three primitive tasks, in the HTN mindset

This well-defined task can now be integrated into larger behaviours, or re-used at the right time, to allow the entity to have consistent reactions, and to allow the designers to better control the plan execution.

By cleverly spreading variations in the primitive tasks, for example with unique preconditions per-character, weighted effects depending on the character's personality or even some blunt randomness here and there, those structured compound tasks can be made softer and more diverse throughout the scene, while still ensuring a global rationality for the AI. (The variations can even be chosen and elected at the compound task-level, typically to run a different attack action depending on the current distance to the players, or the type of weapon the AI is carrying.)

A QUICK PEEK AT PERFORMANCE

> HTN planning is also interesting in terms of computation because you don't have to check for preconditions and apply effects for each single action anymore; instead you can simply validate the preconditions of the first action in the task, and bundle together the effects of all the actions in the task to apply this global update to the state at the end.

What's great is that, because of this recursive and networky nature, a HTN planning system can be modelled as a where the nodes are the actions in the plan and the execution flows from the left to the right. In other words, HTN-based AI actually creates **behaviour trees** in real-time!

This means that HTN re-integrates some of the perks of BTs, most notably:

Modularity in design: Since a compound task can represent a whole subset of actions corresponding to a specific behaviour, it can be prepared once and then re-used by the designers in various places to build a more complex logic for the AI. Just like with behaviour trees, you can make a library of small self-contained chunks of logic ready for re-assembly, that designers

can then pick from to create advanced behaviours on their own.

Expressiveness and Since HTN also works with a top-down logic flow (from the root, i.e. the current world state, to the leaves, i.e. the final goal), it is possible to debug and understand the AI by examining its planning tree, and it lets designers define behaviours very precisely.

Moreover, because those behaviour trees can evaluate the effects of the actions on the world, they can reflect upon the **future** and better emulate anticipation or surprise.

It is important to note that HTN is not necessarily a better replacement to GOAP in every situation – rather, it is a **valuable** especially for large groups of enemies or behaviours that require more control. (As for efficiency, although Eric Jacopin's 2014 article does indicate a slightly improved performance with HTN over GOAP in his comparison of

---

F.E.A.R.

---

against

Transformers: Fall of Cybertron

, he also warns about the possible impacts of the individual implementations on the benchmark, and how this should be taken with a grain of salt.)

Anyway, since then, the HTN planning system has been used several times, for example in

Max Payne 3

or in

Horizon Zero Dawn

, and it is slowly taking its place in the AI developer's toolbox. Gone are the days where the only way to give brains to your minions was the state machine, or the behaviour tree – 'cause as usual, designers yearn to explore new ideas, more flexible structures, and to find other techniques for balancing out realism and predictability...

How to (not?) implement AI planners

The automated planning technique and its more recent evolutions such as the GOAP architecture or the HTN planning system are therefore an interesting way of creating flexible and reactive AIs in our games.

From a coder's standpoint, however, the issue with planner AIs is that they require quite a lot of elements to get started. Even our example of the guard walking from one room to the other would need a description of the game's world, which means having a language to write this description in, and a list of rules to define the grand laws of this world, and a set of facts that completely specify the current world state.

All of this is highly dependent on your use case, and it cannot be summed up in a dozen pages – there is a reason why there are entire books dedicated to the topic! That's why, here, contrary to the other tools we've introduced so far and the utility-based AI we'll discuss in the next chapters, I am not going to detail the implementation of an AI planner.

Rather, for the GOAP architecture, I'll point you to some of the incredible resources shared by Jeff Orkin on his website, among which:

the

F.E.A.R.

SDK which contains all the source code of the original GOAP implementation: https://www.gamefront.com/games/f-e-a-r/file/f-e-a-r-v1-08-sdk

Luciano Ferraro's

ReGoap

project, which is a generic C# GOAP library compatible with Unity: https://github.com/luxkun/ReGoap

(The

ReGoap

Github repository is really great and it is definitely worth checking out in details if you want to discover more about AI

planning for Unity games!)

And for the HTN planning, I recommend you check out the chapter dedicated to this topic in the famous Game AI Pro book collection that is available for free over here:

More specifically, HTNs are studied in the first volume of the series, as part of the "Architecture" section, in Chapter 12 (written by Troy Humphreys, who worked on the

Transformers: War for Cybertron

and

Transformers: Fall of Cybertron

games):

Summary

In this chapter, we've studied automated planners and their descendants.

We've first taken a bit of time to discuss what we mean by "intelligence" in the phrase "artificial intelligence" by studying a cognitive model proposed by MIT researchers, and to which extent this is truly the ideal objective for game AIs.

Then, we talked about planning systems – from the initial AI planning architectures like STRIPS to the more recent

evolutions such as GOAP or HTN. We saw how this alternative design architecture can free AI developers from a lot of work and create highly reactive behaviours, capable of adapting dynamically to the environment... but also how this is sometimes done at the cost of some accountability and ease-of-design.

Finally, we listed a few resources for actually implementing or using the GOAP or HTN architectures in game projects.

In the next chapter, we will continue exploring those recent AI models and introduce another adaptive and not-too-structured technique, called utility-based AI.

10 - Discovering utility-based AI

In the last chapter, we talked about the AI planners. We saw how this alternative behaviour modelling architecture has now entered the video game AI developer's toolbox, and how it offers new techniques for creating more flexible, dynamic and realistic entities. Specifically, we focused on the GOAP and the HTN models, that have been used in various titles since the mid-2000s to make more interesting opponents for the players and provide better behaviour authoring for the designers.

A key idea we developed throughout this discussion was that, compared to the finite state machines and behaviour trees we'd studied earlier in the book, these planners require almost no hand-written decision logic. Instead, these systems are limited by the actions at their disposal and the necessary conditions for those actions to be valid, but then it's up to them to formulate plans and compose behaviours as they want.

Although this sometimes results in unexpected reactions from the AIs and thus decreases the level of control the designers have on the behaviour of their entities, planners are nonetheless an interesting tool for having the AIs partially self-

construct their brain – which both reduces the workload for AI programmers and makes for more lifelike behaviours.

In the same vein, there is another AI programming technique that has started to grow in the game community these past few years and aims at spontaneously crafting context-adapted behaviours: the utility-based AI.

An overview of utility-based AI

The concept of utility is by no means specific to game AI design – it has been used in many domains ranging from business to classical game theory, economics or even biology.

But in the context of video games, **utility-based AI** (or has notably been brought to the public by Dave Mark and Kevin Dill, two game developers who used this technique in some of their artworks (like the weapon and dialogue selection processes in

---

Red Dead Redemption

), and then shared their knowledge and research via books and GDC conferences.

In particular, their conference from the 2010 GDC AI Summit,

Improving AI Decision Modeling Through Utility Theory

, recalls the base principles of this technique and gives a nice overview of the strengths and common applications of utility-based AI. So let's sum up this discussion to get a bird's-eye view of UBAI :)

WANNA SEE THE CONFERENCE?

> By the way, the video of this conference is available in the GDCVault archives, over here: https://www.gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through.

Getting the gist

The core idea behind utility-based AI is to consider all of the possible actions for the entity at this precise moment in time, and assign each of them a **utility score** to determine the best option. To put it another way, we want to **compare**

Think of it like going to the restaurant and having a large menu with lots of possibilities. Depending on your current state of mind, what you've eaten recently and what your personal preferences and/or food allergies are, some choices will instinctively feel more interesting than others. In your head, you'll give them a higher "score", a higher utility, and consider those your top priority. The rest of the menu still exists, but given the current situation, it will be demoted to the bottom of the list... and you'll simply choose the

(subjectively) best meal in this list, or one among the meals that caught your eye the most.

Similarly, in a game, a unit could have a list of possible actions and give each a utility score to create this ranked list of preferences. Then, all that's left to do is pick the one at the top, or choose a random item among the top ones, and mark it as "the best action for the moment". (Choosing one of the **N** best possible choices can be a good way to make the AI less predictable and robotic – we 'll study that in more details in *Chapter 11: Designing a utility-based wizard*

And yes, if you're still wondering: utility-based AI is at the heart of Maxis's

The Sims

; it actually transpires in the gameplay, since the players are basically asked to guess the utilities of the various actions of their Sim to find a way to maintain its overall happiness!

VALUE VS UTILITY

> It is essential to understand that **utility** is not the same as **value**.

> While value is inherent to the item and remains constant, utility depends highly on "who's asking", i.e. who's computing it, because it is a mix of the item's properties and the evaluator's.

> Coming back to our example of the restaurant from above, although each meal has a price on the menu, it may not correlate with how "good" *we* consider it. Our utility for these items is subjective, and it is influenced by our own history, our own desires, our own specificities. So if you ask someone else, they may come up with a different ranked list, because they have their preferences too, and those will result in different utility scores. (While the value of the items stays the same for everyone…)

This whole process is sometimes summarised as: **the expected** or in other words, figuring out which choice among the ones available to the AI right now will likely give the best utility score, given its current situation.

And this technique of always taking the whole potential **decision space** into account and weighing every choice actually results in a fairly unique type of AI architecture, quite different from the ones we've seen so far.

Comparing to FSMs, BTs and planners

A key thing with UBAI is that, because it always considers the entire set of possibilities and ranks them more or less dynamically, based on the current context and the entity's desires, it creates a **flexible and adaptive** behaviour. The unit's reactions aren't fixed forever, they are built dynamically to fit the context – similar to what an AI planner does.

However, utility-based AI doesn't formulate and execute plans. Instead, it constantly re-checks to see what's the best course of action at this exact time. A nice consequence of this is that, if a cool goal suddenly pops on the way to your current destination, you're not blocked into your current plan. You can temporarily derive from it, accomplish this localised secondary objective, and then come back on track to achieve your main goal from before.

Typically, imagine an AI walking from Room A to Room B to approach the player, but it's wounded and there is a medipack on the way. With an AI planner, if picking up this medipack hasn't been incorporated into the current "Move to Room B" plan, it will simply be ignored. And then the unit will have to **backtrack** to the medipack if it really wants to take it. On the other hand, because utility-based AI does an immediate continuous evaluation of the entity's action set and its context, we could design a more realistic and clever behaviour, such as:

Move towards Room B.

Walk by the medipack.

(The utility of picking up this item skyrockets to encourage this behaviour to interrupt the current one.)

Pick up the medipack.

(Re-evaluate the actions with the initial utilities.)

Go back to the initial "Move towards Room B" behaviour.

In a sense, UBAI is thus even more flexible than AI planners!

Plus, because you evaluate the utility of every action before comparing their scores and electing the best choice, you're less prone to missing out on **edge**

In comparison, consider the finite state machines or the behaviour trees we've studied before in this book. Those techniques adopt another mindset, where you browse through each action one after the other; you check if it's doable, and only if it's not valid you then go and test the next one in line. This means that, with those architectures, you have to handle prioritisation, fallbacks, interruptions, and possibly some rare or

weird edge cases if the logic flow gets crazy and you can't do any of the expected things.

Utility-based AI is therefore pretty dynamic and powerful, and it creates very reactive AIs that constantly consider the whole context and don't stay stuck in pre-made plans or state-based logic.

But wait – how is it possible? How can the entity assign those utility scores and pick the right action continuously in a context as complex as the one of a video game?

From data to behaviours

A fundamental idea when doing utility-based AI is that of the **data** and the **data** of raw pieces of information into actual game concepts. To put it simply, we have to use numbers because that's the only thing computers understand, but the whole point of UBAI is to combine those numbers into rational behaviours for our entity.

Converting data to concepts

For example, we could create a logical association between the amount of health points a unit has, and how willing it is to use its healing spell. Typically, here, we could say that as the

health of the entity decreases, its envy for healing grows; and if the unit ever has a near-death experience, the utility of this action will go through the roof. Thus linking this integer health value to the utility of the entity's healing action seems fairly coherent.

Some concepts may be a bit more complex, and in that case you may not be able to fully describe them with just a single data point. Instead, you can combine multiple variables, called **decision** to determine the utility of your advanced game concept. (E.g. the desire a unit has of attacking will most probably depend on several things such as its current health, the strength of the enemy, whether or not the AI has some big armours and weapons, etc.)

However, to truly establish these links and design relevant relationships, we cannot simply look up absolute numbers in our game. Unfortunately, this raw data doesn't make much sense without a global context – in particular, we have to know what is "normal", "good" and "bad" inside our game to properly assess what the data means and make informed choices.

If we just read in our current game data that the unit currently has 10 healthpoints, without any additional context, we can't really do much. We have no idea if that's the maximal health and the character simply has a small healthbar,

or if it's a critical point where the unit should absolutely find a way to heal! It would be easier to understand what was going on if we were told the unit had 20% of its health. Here, absolute values are hardly ever useful; for utility-based AI, you want to standardise your data and make it relative to specific scales, so that you can spot outliers and rare events, and average or combine your utilities into new interesting variables.

To do so and re-integrate the game context into the raw data, we usually rely on three notions:

Conversion By carefully picking the right mathematical functions, we can process our initial data into more meaningful values, and utility scores. This conversion logic may be more or less complex. It usually relies partially on external constants and global variables that are described by the developers to specify key elements of the game context.

Response curves (or **falloff** curves): These formulas can then be visualised as plots. On these plots, we thus see how the processed value evolves compared with the initial one, and we can get an intuition for the type of relationship between the two.

Finally, in order to work in a more context-fitted scale, and in particular to be able to perform our utility comparisons, we have to make sure everything lies in the same range of values.

That's why, with UBAI, we renormalise a lot of the data, and most notably all the utility scores, to the [0, 1] range.

Thanks to normalisation, it becomes possible to compare any two utility scores, no matter what the underlying data scale is, because the final result will necessarily be in this [0, 1] range.

To define a utility function and find the good mathematical formula for modelling the impact of this variable on the entity behaviour, we usually start by considering a few points:

Is the utility increasing or decreasing when the variable increases? (or in other words, are we looking for an increasing or a decreasing function?)

Is the rate of change steady or variable? Is this rate of change fast or slow?

Is there a particular inflection point or specific threshold that determines a change in the function's behaviour?

Are the utility values real numbers, or do we have an asymptote that approaches a limit?

Those questions can help us better scope our search, because they can help us identify the family of functions we need to

use for our conversion formula (should it be a constant? a linear function? an exponential?...).

So, our goal when designing a utility-based AI architecture is to identify the relevant variables in our game context, transform them if need be with a well-chosen math function, and finally normalise the result to get well-scaled values. Then, the whole trick will be to use, re-use, combine and even re-process those normalised values into new ones to get higher-level concepts, and eventually action utilities for the entity.

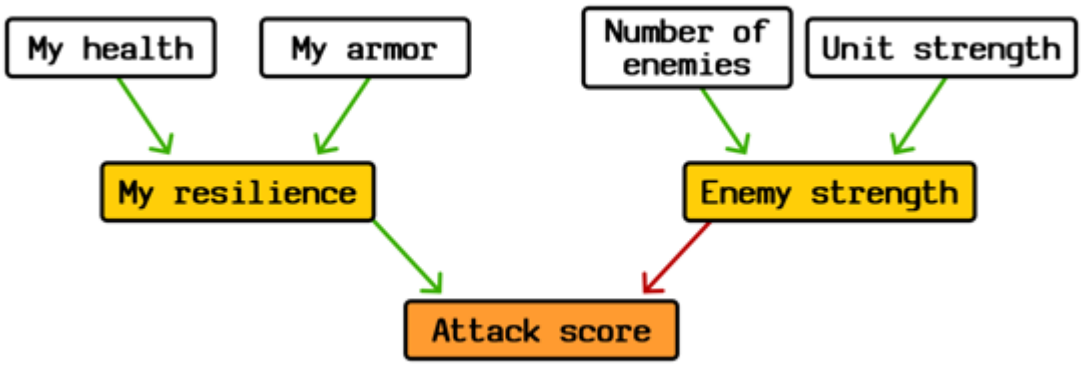


Figure 10.1 – Basic UBAI utility computation tree for determining the utility score of the Attack action for some example AI (green arrows show positive contributions, red arrows show negative contributions)

As you can see on *Figure* UBAI therefore relies on plenty of little blocks stacked up as a tree. Each block maps a number

input to a normalised number output, and can be more or less atomic. For example in this image, the blocks at the top are **individual utilities** computed directly from a variable in the game data My Number of while the blocks on the next rows are **conceptual utilities** Enemy that derive from previous computations.

To assemble blocks into higher-level conceptual utilities like this, we usually rely on simple formulas, like basic additions and multiplications; more precisely, we often define the utility functions of those abstract concepts as a weighted sum of lower-level individual or conceptual utilities – such as:

My resilience = 0.7 * My health + 0.3 * My armor

The weights (here: 0.7 and indicate the impact that this specific individual/conceptual previously computed utility has on our higher-level conceptual utility. In my example, the normalised output value of the My health block is about twice as important as the one of the My armor block; meaning that changes in the My health value will have twice the impact on the My resilience value. Also, a positive weight indicates a positive contribution, while a negative weight indicates a negative contribution (see the green and red arrows in *Figure*

Sometimes, the formula can be even simpler and just merge together the input values in some way – typically, for our

Enemy strength value in *Figure*

Enemy strength = Number of enemies * Unit strength

Of course, since we want to be able to chain utilities in a tree, we need to ensure that all of our utilities are normalised to the [0, 1] range. If your computations involve negative factors or higher weights, you might need to renormalise the formula or re-clamp the values in the end to make sure it stays in the right range.

A modular and scalable technique

This whole system has a few interesting consequences for us:

Each individual decision factor (e.g. My My can use its own model, and this model is self-contained, with no relationships to the other utility functions. And because each pack of blocks can be treated and validated on its own (e.g. we can first check that My resilience is computed properly, before adding the Enemy strength branch and moving on to checking the utility function for My UBAI is highly **modular and**

We can insert new concepts in our AI's brain by giving them utility functions, and we can then mix and match those concepts as we please to create new ones, at our own pace,

once we're sure that the previous ones have been designed properly. Each group of blocks is which allows us to iterate gradually.

(This modularity partly comes from the fact that conceptual utilities often don't depend on hundreds of previous blocks. Rather, you usually just merge two or three lower-level utilities together, so as to keep the whole system fairly modular.)

Because many blocks aren't specific to this entity, but instead inform us about its immediate surroundings, parts of our utility tree can actually be **shared** across multiple AI instances. As an example, if we have some soldiers in the same squad targeting the same group of enemy, then all the info about these enemies can be computed once and shared between all of the squad soldiers.

And because many of the first blocks are quite generalist, they can also be re-used inside a single AI utility tree in various places, and participate in the evaluation of multiple goals at the same time, like a base building block. For example, here, the My health block could probably be part of the Attack score computation as shown on *Figure* but also of the Heal

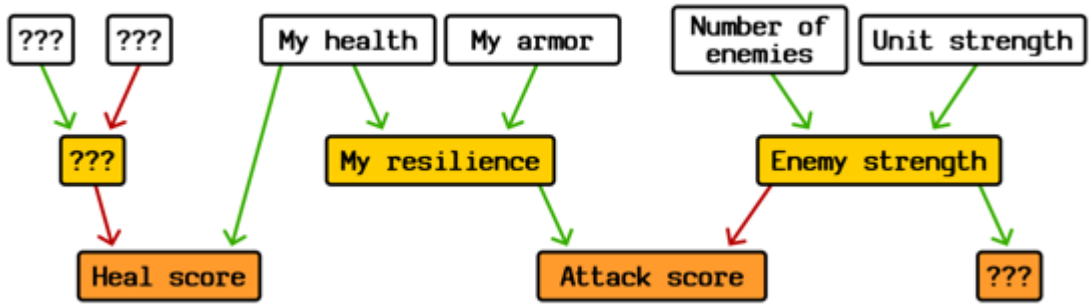


Figure 10.2 – Extended UBAI utility computation tree with re-used utility blocks for some example AI (green arrows show positive contributions, red arrows show negative contributions)

All of this also highlights how, with utility-based AI, the data processing phase (i.e. going down these utility computation trees and filling in the output values of each block) is distinct from the decision processing phase (i.e. comparing the utility scores of the entity's actions and ranking them to find the best option at this moment).

A few extra notes

To improve on and properly implement these principles, there are a few additional ideas we can discuss, such as:

Using **time/distance-dependent decision** Be it to directly define a conversion formula or have some individual utility become dynamic with regard to the time or the distance to an

element, adding time/distance-dependent factors can be a great way to boost the realism of our AIs. In particular, they can help emulate less robotic and predictable reactions by making the AI evolve throughout the game.

To avoid the AI constantly switching between two actions if their utilities happen to be close in the current situation, a solution can be to define some sort of "inertia" in our AI's brain. Basically, the idea is to temporarily increase the utility of the choice it just made so that it sticks with that option for a little while. This artificial overweight can then fade away thanks to a system of decay values that gradually bring it back to its normal value, as a sort of cooldown before potentially choosing something else; or you can even prevent the unit from taking another decision until its current action has finished.

Scalability/performance tricks: Obviously, evaluating and re-updating the entire utility tree of each AI in our scene is quite resource intensive. Even if a single entity can come up with the best possible actions quickly, especially if the utility functions it uses are simple, things become more complicated when you have large packs of AIs in your scene. Just like with AI planners (see *Chapter 9: The reverse-thinking of* constantly computing everything for everyone takes a toll on the machine.

To mitigate this problem, we can decide to only re-run the utility evaluation process every N frames, since it will still look

pretty instantaneous in the eyes of a human player; or we can use triggers to re-update the result of utility functions when their input changes; and of course, we can cache and store data to reduce the number of redundant calculations.

Also, what's nice is that by definition UBAI works quite well with each agent can take care of its own data in its own thread or process without any collisions with the others, and you can therefore run these computations in parallel. (Just be careful with shared data – if you want to centralise some utility values somewhere to share them between multiple entities, then you'll need to designate a particular agent to compute those.)

A ZOOM ON MARGINAL UTILITY

> In their conference, Dave Mark and Kevin Dill mention an interesting extra concept, called **marginal utility**, which basically translates to: "how much do I get out of adding 1 more?". This notion is a way of computing the utility of an item at a given time not only based on the current situation, but also on the **past utility score** of this item. (Yes, once again, we're rediscovering it's interesting to input some **memory** in our artificial brains…!)

> This marginal utility can either increase over time (meaning having more of this item gets more and more useful), or decrease

> over time (meaning having more of this item gets less and less useful).

> Having an increasing marginal value may make sense if the item presents some kind of **emergent** properties, which make it more interesting to have multiple instances than just a single one. It's like with LEGO blocks: if you simply have one brick, you won't go very far; but if you have a whole pack of bricks, you'll be able to build a lot of things!

> Conversely, having a decreasing marginal value can simulate a sort of **disgust** or **boredom** for our AI. For example, if the unit has already eaten a dozen pies to fill its stamina bar, it's pretty likely the thirteenth would be as valuable as the first one. With each new addition, the utility of repeating the action lowers.

UBAI is therefore very straight-forward in theory: it offers a rational remapping of relevant data to real game concepts, it combines those concepts into higher-level ones, and it eventually chains a bunch of computation with highly customisable influence weights to get intuitive utility scores. Moreover, because it doesn't abruptly ignore actions and relies on softer comparison mechanics, utility-based AI allows for finer **granularity** and more **delicate transitions** between behaviours. Specifically, this technique avoids a common

problem with game AI, which is entities behaving based on arbitrary thresholds – which results in pretty artificial reactions.

However, in my experience, in practice, things aren't always as smooth and you might need to tweak your response curves and combination weights quite a lot before you can get the right behaviour. And all this theory is nice, but it's still a bit fuzzy how these numbers can actually translate to behaviours. So, even though we'll examine a more in-depth example in *Chapter* we're going to take some time to discuss a basic example, and go through designing response curves and utility functions step by step.

A quick example

To get a good grasp on these notions of response curves and "data to behaviour mapping", let's take a simple example. We're going to re-study the case of the dragon guarding its treasure that we discussed in *Chapter 3: What are*

A review of the situation

You might remember that, back then, we had modelled the behaviour of our creature using a finite state machine with three states: Guarding, Attacking and Sleeping (see *Figure*

We said that while the dragon was in the Guarding state, it would look out for menacing heroes, and after a while without doing anything, it would go to sleep for a pre-determined amount of time. However, if the hero got too close, then the dragon would switch to the Attacking state and start to regularly breath fire on the mighty adventurer to defend its riches.

This AI architecture was easy enough to design, and it made sense to use finite state machines because the dragon had clear states and transitions.

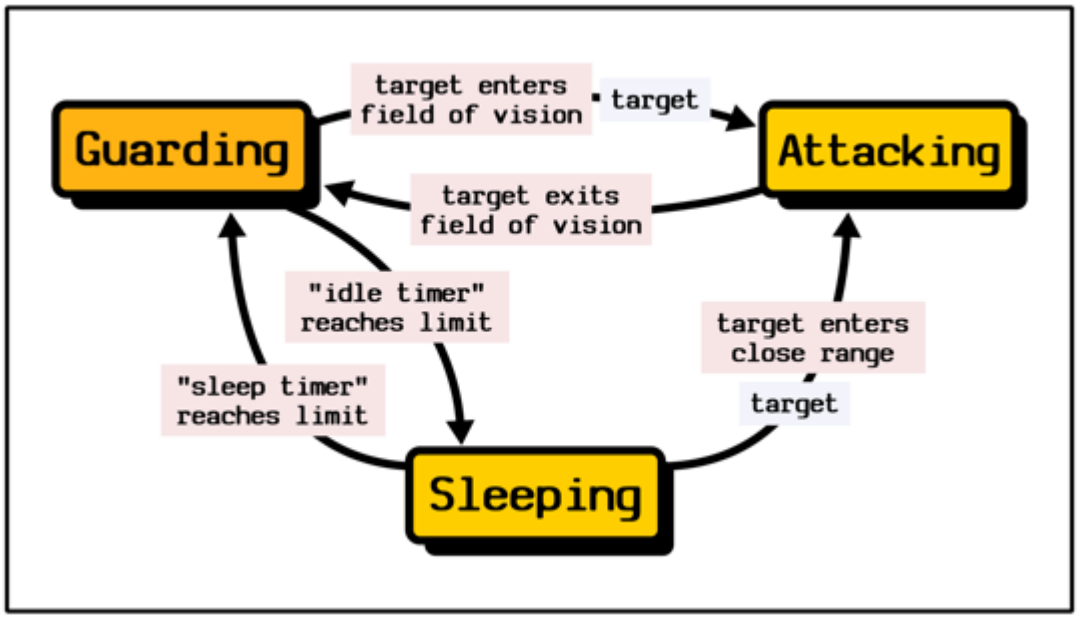


Figure 10.3 – The finite state machine architecture for our dragon AI from *Chapter 3*

But we could actually reconsider the same problem from a UBAI perspective, and see how to formalise our data models to get a similar behaviour. More specifically, we're going to ignore the Sleeping state here, to rather focus on the Guarding-to-Attacking transition, and how this relates to the distance between the hero and the dragon.

An aggressive beast!

Our goal here will be to apply the utility-based principles, and in particular the data processing step we discussed, to determine the utility score of the attack action of our dragon, versus staying idle. This score will depend on the level of aggressivity of our dragon, which is itself based on its distance to the hero.

At the moment, we've said that there was a certain threshold at which the dragon would spot and target the hero – this is very common in video games, AI often have a field of vision that determines a "spot area". However, this system has a huge flaw: it doesn't look natural at all!

Indeed, it means that for our dragon (which you'll recall is static on its pile of gold), there is a sort of magical line in the sand that arbitrarily shuts down its perception above a certain distance. If the hero enters its field of vision and then

takes one step back, the creature will instantly forget it exists. Pretty strange, right?

With this model, the aggressivity of our dragon entity therefore correlates to the distance to the hero in a super basic way: it's either maximal if the hero is below the threshold, or null if the hero is above the threshold. This corresponds to a **binary threshold** response curve, a simple on/off instant falloff.

For example, assuming this field of vision has a radius of 4 metres:

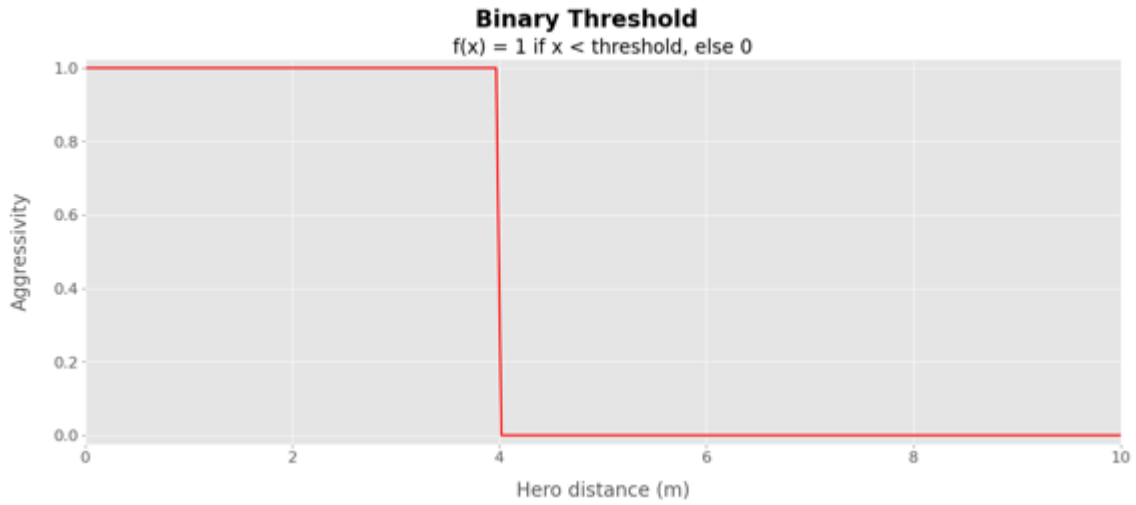


Figure 10.4 – Binary threshold response curve for the aggressivity level normalised value as a function of the distance to the hero

Note that here, as discussed in the previous subsection, I've renormalised the data so that the aggressivity is in the [0, 1] range. Also, in this graph and the upcoming ones, we'll consider that the distance to hero is in metres.

This kind of response curve is the simplest of them all, and it directly maps to an if/else statement in code. Basically, our arbitrary line creates a very harsh transition between two behaviours (here Guarding or Attacking), with no in-between. This is of course nice for developers because it's easy to code, but it also feels extremely artificial and unrealistic. Especially if the player discovers this threshold, and then oscillates around it, making our AI crazily fluctuate between the two behaviours!

What would be better is if we could smooth out this threshold and get a more continuous reaction. Something so that there is no sudden discontinuity, but rather a progressive adaptation of the dragon's aggressivity level as the hero gets closer.

A natural improvement on our binary threshold could thus be a **linear threshold** which, based on a maximal view distance for the AI, would make the reaction continuous:

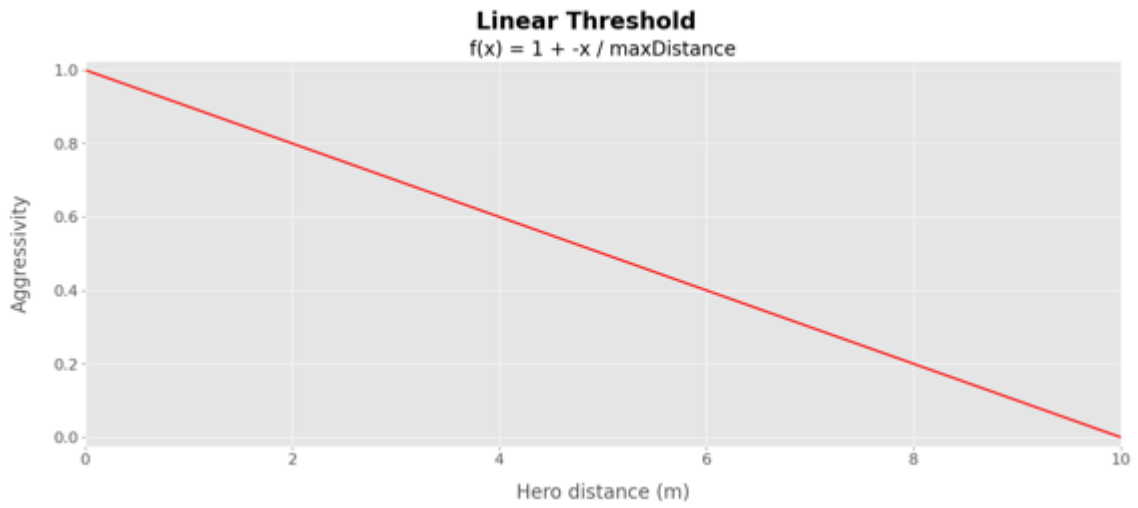


Figure 10.5 – Linear threshold response curve for the aggressivity level normalised value as a function of the distance to the hero

We again get the aggressivity level for our dragon as a value in the [0, 1] range, but this time you see that the **rate of change** is completely different: it slowly crawls up as the distance to the hero decreases, and eventually reaches its maximal value when the hero is literally at the dragon's feet. Of course, there is still some arbitrary choice here, since, for the sake of simplicity, I've decided that the creature cannot see further than 10 metres (the aggressivity level drops to 0 at this distance).

But all in all, this is already way more natural than the binary threshold from before. With this new setup and some clever animations, we could emulate the dragon getting itchy and annoyed as we get closer, and then slowly calming down if we

decide to walk away – rather than having the peaceful creature suddenly bursting with rage if we take a single step.

Now that we have a somewhat reasonable function for our aggressivity level, let's see how this would help us make a utility-based AI.

Designing utilities and picking the best action

Alright – we've chosen a linear mapping that can transform the distance between our dragon and the hero into an aggressivity level. The next step is to actually use it to compute a utility score, and then have a ranking of our possible actions.

For this example, we'll simplify our creature's action set and consider only two possible actions: either stay idle, or attack. And to start off and simplify even more, we'll say that for now, the utility of staying idle, denoted is a constant at

Meaning that, at any given point in time, and no matter the distance between the dragon and the hero, the utility score of the idle action will always be This gives us a first (incomplete) graph comparing the utilities of our actions, with only the

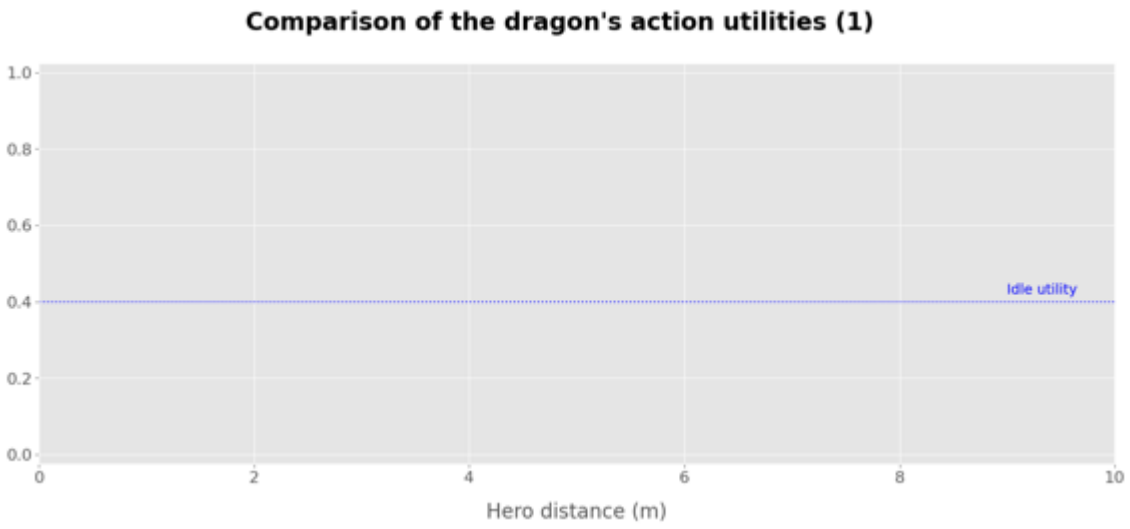


Figure 10.6 – (Incomplete) graph comparing the utilities of the different actions available to our dragon AI

Then, we'll assume that our utility for attacking, denoted only depends on the aggressivity level of the dragon. So we'll directly use our previous value as the utility, and just re-integrate this linear ramp into our action utilities comparison graph:

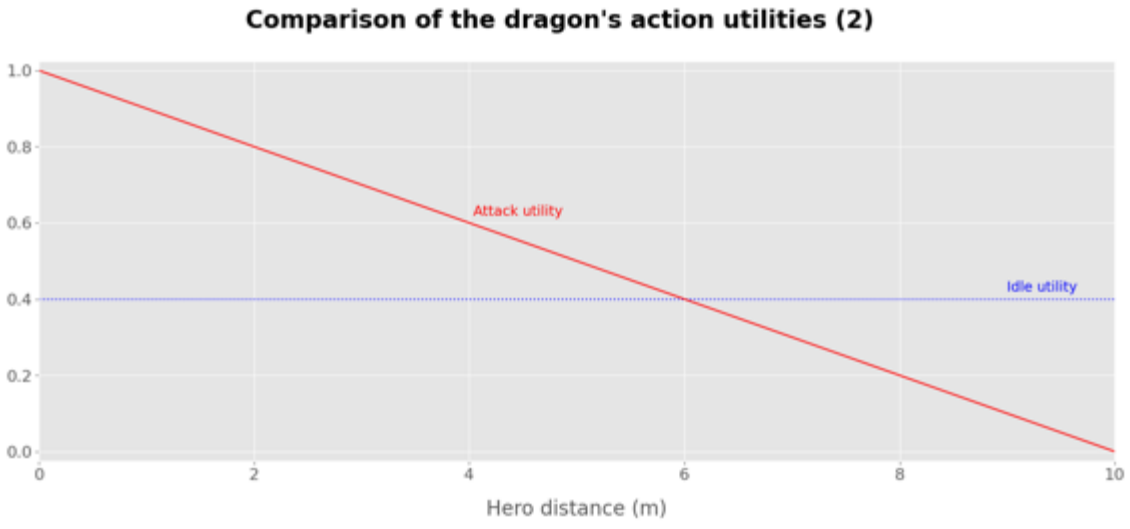


Figure 10.7 – Final graph comparing the utilities of the two actions available to our dragon AI

And that's where UBAI really shines! With a glance at this graph, it's instantly obvious how our AI is going to react, and when it is going to choose the idle action over the attack action. Basically, as soon as the red curve gets below the blue one (here, for a distance above 6 metres), the idle_utility will take precedence and rank staying idle above attacking.

Of course, this sort of re-creates a binary yes/no situation, which as we've said is not very realistic. But the really cool thing is that this idle_utility doesn't have to be constant! We can have it derive from some other piece of data in our context, such as how long the dragon has been inactive:

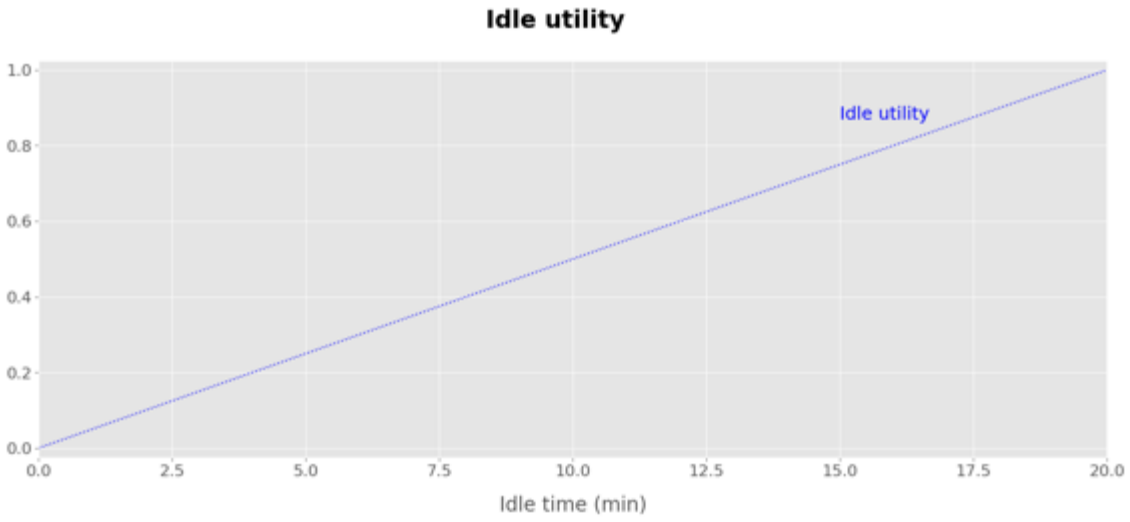


Figure 10.8 – Linear threshold response curve for the normalised idle_utility as a function of the idle time

You see that here, we use another **situational** the "idle time" (in minutes) to determine the idle_utility based, once again, on a linear function.

THE LIMITATIONS OF PLOTTING

> At that point, comparing our utilities side-by-side becomes quite tricky, because they're not using the same axes anymore. The utility values are easy to compare, because they're normalised, but plotting them requires us to go from 2D to 3D, in order to get both our Hero distance and Idle time variables on the same graph – and check for intersection between surfaces, which isn't easy to picture. Even worse, if we add a variable somewhere (either because there is another action which utility is computed based on a third variable, or because we decide that one of our two idle_utility or attack_utility should mix multiple variables), we would need to plot a 4D graph, which is impossible!

> The intuitive graphs we've had so far are therefore mostly usable for individual response curves.

Because we've used a time-related variable for our idle_utility computation, this utility has become a dynamic value that will change throughout the game. This means that the intersection point of our two curves is now dynamic as well, and that it depends on the combination of the distance to the hero and the current idle time.

This very basic example therefore shows us that utility-based AI is interesting from a developer's point of view because you can add your actions gradually, by implementing and mixing more and more utility functions in each iteration, and you can express relatively complex situations by combining components that are straight-forward on their own.

Still, using just binary, constant or linear thresholds will get boring real soon, and it clearly lacks some finesse. So, to continue our case study, let's now discuss how picking the right type of function for our data transformation is key in creating unique and expressive behaviours for our AIs.

Expressivity via maths?

The nice thing with using falloff curves like this is that it allows for a lot of expressivity and quick per-character tweaks. For example, we could easily transform our linear threshold into other curves, to get some variations in our dragon's behaviour (note that this is far from a comprehensive list):

Exponential threshold: Suppose the dragon is a bit old and has bad eyes; or the hero only has a sword and therefore presents no threat when at a distance; or the dragon is a pacifist and only gets to the angry roasting stage if you're *really* menacing.

Then we could switch out our linear aggressivity level response curve for an exponential one:

Figure 10.9 – Exponential threshold response curve for the aggressivity level normalised value as a function of the distance

to the hero

This way, the dragon would stay fairly calm even if we start approaching it, but then when the distance gets really too small it would eventually get angry and start to respond to the threat. This much sharper transition emulates a big emphasis on the last metres in our AI's brain (although it is still more believable than a simple binary threshold).

Also, the interesting thing is that, as you can see, simply changing the falloff of our value can translate to very subtle concepts, either on the part of the creature (e.g. it is a more or less "peaceful" entity) or on the part of the hero (e.g. it uses a particular type of weapon).

Exponential threshold with a different exponent: Even sticking with this idea of the exponential response curve, we could modulate the reactions of our AI by changing the exact parameters of our exponential. Typically, by lowering or increasing a multiplicative factor in the exponent, we could precisely adjust the falloff and thus the reactions of the creature:

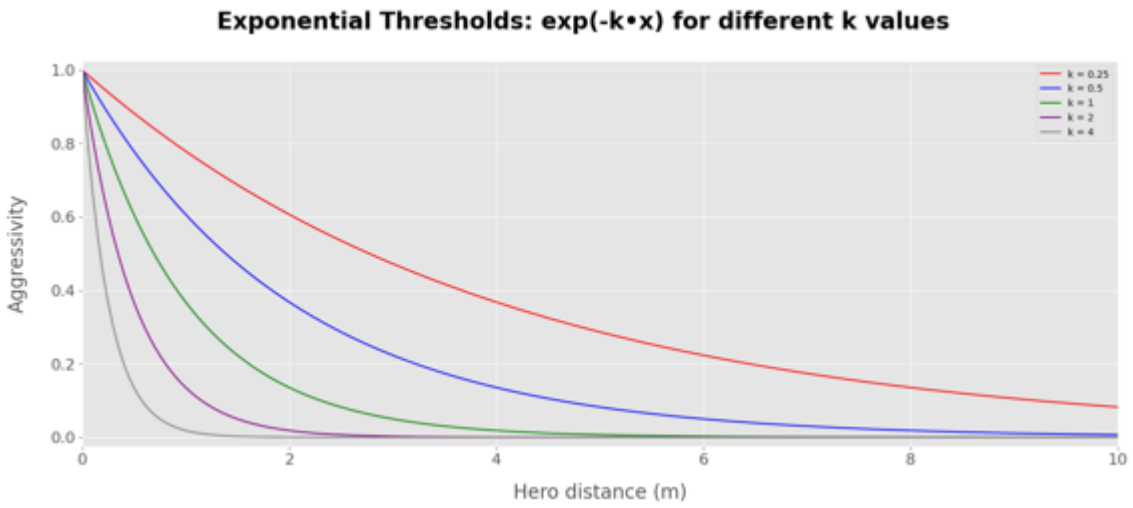


Figure 10.10 – Different exponential threshold response curves for the aggressivity level normalised value as a function of the distance to the hero, depending on the exponent factor

Here, the red curve (with a low exponent) is way smoother than the gray one (with a high exponent). Each exponent thus gives a specific rate of change for our aggressivity level, which would in turn have our attack_utility and idle_utility compare differently, and our creature change its state of mind in unique ways as the hero gets closer.

Logistic threshold: Another common type of response function is the logistic function, which graph is a sigmoid, with its recognisable "S" shape:

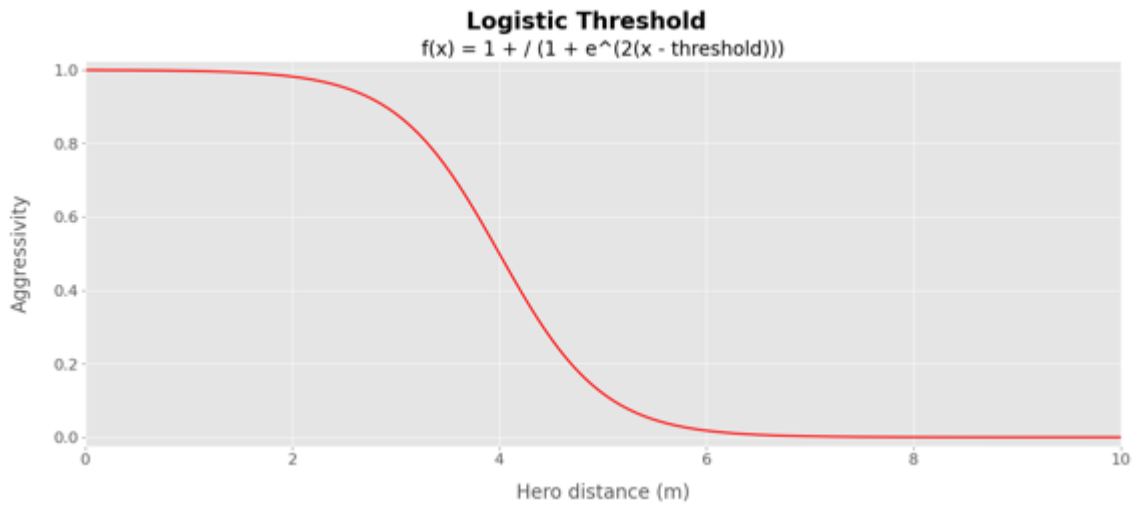


Figure 10.11 – Logistic threshold response curve for the aggressivity level normalised value as a function of the distance to the hero

Using a logistic threshold like this allows us to create smooth transitions, but with some plateaus at the start and the end of our distance range. So, typically, this could simulate this sort of "inertia" we talked about, to have our AI focus on a task for a bit longer.

Piecewise threshold: In some cases, relying on just a single mathematical equation like this is too limiting, and doesn't provide the designers enough control for the intended AI behaviour. When that happens, a possible solution is to build your own custom response curve "by hand", by defining multiple inflection points and, between each those, describing a very specific conversion formula. You end up with a falloff curve chopped down into several pieces that each have their

own rate of change and allow a super high fine-tuning of the AI's reaction:

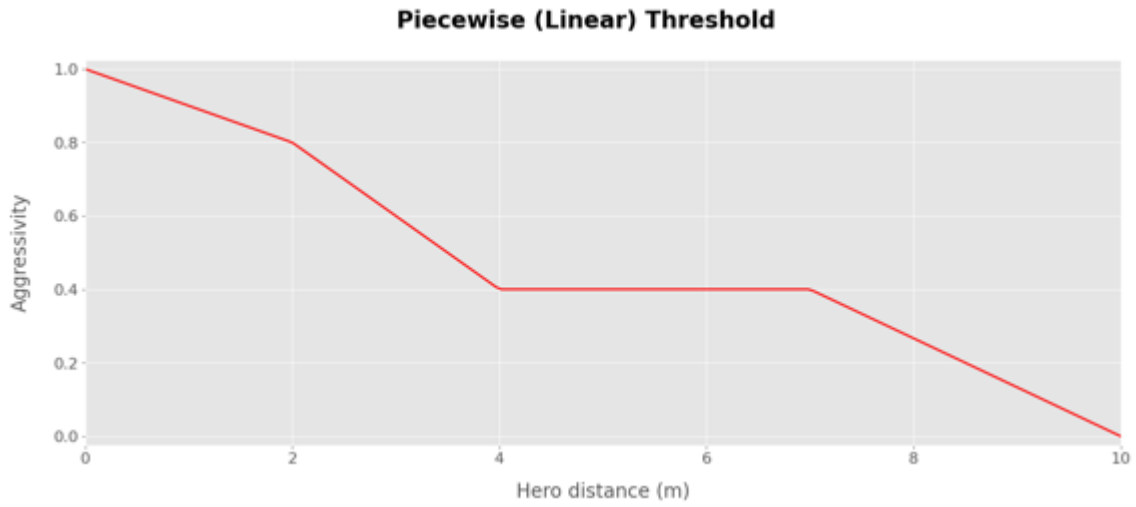


Figure 10.12 – Piecewise linear threshold response curve for the aggressivity level normalised value as a function of the distance to the hero

Usually, we stick with piecewise *linear* response curves (like here, in *Figure* because having multiple inflection points already gives a lot of control, and we don't really need to add the complexity of a big math formula on top of that. But, technically, you could totally have each piece of your curve use a different type of function, or make little sections with exponentials using different exponents for example.

However, it is important to try and keep the response curve continuous, meaning that a small change in the decision factor shouldn't result in a harsh jump in the utility score. So if you

create piecewise falloffs, be sure to have each chunk match its neighbours and avoid vertical discontinuities!

Again, all of this depends on the other curves, how they are translated into actual utility functions and how they are linked to the AI's action set; but this shows us that UBAI is exceptionally detailed and precise, and it gives us a high level of control on our entity's behaviour – in particular, in terms of expressivity.

THE POWER OF INFLUENCE MAPS

> When your utility depends on a position in your world or on a distance-related situational variable, there is another nice way of plotting your context: an **influence map**, or a **heatmap**. In a nutshell, the idea is to chop down your scene in some way (we call that **partitioning space**), optionally inject in the **connectivity** points (such as doors or, on the contrary, walls), and finally use some **colour code** to indicate how high or low a specific utility value is in each little cell of this map.

> For example, imagine we want to make a RTS player AI to completely automate our little resource gathering scene from *Chapter 8: Implementing a RTS collector AI*. Then, among other things, this computer-controlled player would need to be able to place resource storage buildings, ideally in optimal spots. This would mean finding the areas that are close to intense resource clusters… but also not too close to another storage centre of the same type,

because that's just redundant. So we could draw an overlay on our scene from *Chapter 8*, that is the influence map, and that indicates the recommended areas (in green) and the disapproved ones (in red):

Figure 10.13 – Example of an influence map on our RTS demo tilemap for the best (in green) and worst (in red) resource spots for looting wood

This is a fairly intuitive visualisation of our data, and this 2D per-cell data can actually be used as a decision factor for our AIs, for example here to determine the best building spot. (By the way, influence maps are one of the main tools that Will Wright used when developing SimCity, to incorporate some urban dynamics and simulate real-life patterns like urban decay.)

If you want to learn more about this tool, don't hesitate to have a look at:

- this nice article by Alex J. Champandard, The Core Mechanics of Influence Mapping: https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-core-mechanics-of-influence-mapping-r2799/

> - or this other conference by Dave Mark and Kevin Dill at the GDC 2012, Embracing the Dark Art of Mathematical Modeling in AI : https://www.gdcvault.com/play/1015683/Embracing-the-Dark-Art-of

---

At this point in our study, it looks like utility-based AI is an amazing technique that should be in every game. It is flexible, incremental, robust and yet adaptive, quite easy to debug for developers and even offers a lot of control to designers to create more expressive and realistic behaviours.

So, why isn't it used in every game?

Is utility-based AI the perfect solution?

Now that we are clear on what UBAI is, it is time to talk a bit about the advantages and drawbacks of this tool... and see if this technique is really a silver-bullet!

Understanding the strengths of UBAI

First of, let's recap the different strengths of utility-based AI.

We can re-use the same falloff curves for multiple entities, share utility computations between agents, and even share

results between several branches of a single entity AI's utility tree. By picking different weights in our utility combination formulas, we can use the same factor for modelling different entity behaviours, and even reverse the influence of said factor from one unit to the other.

For example, suppose you compute the distance to the closest monster. Then, you could use this utility value for an innocent peasant, for whom this decision factor would have a negative influence, so that this AI would try and run away. But you could also use it for a guard for whom, on the other hand, this decision factor would have a positive influence, and so this AI would go and fight the beast.

Therefore, by using scoped logic bricks and well-designed weight combinations, you get a very modular system. (Note that this does require you to establish some conventions in your code to have consistent design patterns and really coherent re-usable structures – such as "higher is always better".)

Another huge advantage of utility-based AI is that it offers us super precise tools for expressing our entities' internal preferences and desires. It allows us to craft highly specialised response curves that tune the reactions of this specific character both to the characteristics of its surroundings... and to its own personality.

Changing the type of conversion function or some of its factors directly impacts the utility of the entity's actions and, in turn, the behaviour of the AI. Those little coefficients thus translate into a subjective perception of reality and re-create this biased mental representation we discussed in *Chapter* and deemed one of the core elements of an interesting cognitive architecture.

Reactivity & Similar to planners, utility-based AI systems are extremely reactive because they self-construct the decision logic on-the-fly, based on the current context. UBAI is actually even more adaptable since it doesn't formulate and then execute plans from start to finish; rather, it continuously evaluates the best possible action and can therefore quickly react to temporary stimuli to achieve a quick interesting goal in the middle of a longer mission. This is particularly interesting to avoid backtracking.

By its very nature, UBAI also gives birth to a lot of **emergence** in our AI behaviours, just like planners. With only a set of simple functions and weighted combinations, you can see a lot of different reactions occur, because the current context offers many intertwined data points that result in a unique ranking of the action set utilities.

By integrating some marginal utility and/or inertia notions in a UBAI brain, we can also mimic a kind of memory, and have the past choices of the entity influence the utility of the actions currently available. This can make its behaviour more consistent and yet evolutive throughout the game (typically by emulating interest with an increasing marginal utility, or disgust with a decreasing one).

Ease-of-design (to some extent...): A really nice perk of utility-based AI is that designing the entity's logic can often be done by analysing our own behaviour as humans and trying to mimic it – 'cause the idea of evaluating decision factors and finding the most useful action at this precise time is actually exactly what we do as a player. So, by imagining we are the AI, we can use our intuition to identify the relevant data points, craft the right response curves and build our utility tree.

We'll dive into this in more details in the next chapter, when we work on our wizard AI and setup the brains of our character by also studying our own reactions as a human player.

And yet... despite all these perks, utility-based AI also comes with its limitations and gotchas! So before we end this chapter, we're going to browse some of those and put the spotlight on a few important things to remember when working with UBAI.

Watching out for the caveats and shortcomings

Utility-based AI poses a few interesting questions and problems to keep in mind:

To start off, a crucial thing is that, because utility-based AI always checks for all the possible actions and necessarily elects one, it will always have the entity do something, even if the entire action set has low priority right now. (Doesn't matter if it's 0.002 against 0.001: there will always be an action with the highest utility!)

This means that, as opposed to an AI planner for example which only acts if it has a goal, this time if you want to allow your unit to stay idle, you have to properly encode that into your model and have the "do nothing" action be part of the action set, with its own utility function.

Also, the high modularity of UBAI brings us to the same tricky design point as with behaviour trees or AI planners: you have to be able to abstract your entity's behaviour quite thoroughly, in order to create self-contained logic bricks and assemble them into rational higher-level concepts. Even if the "action-reaction" response is often easy to get an intuition for, it doesn't mean coding it is as simple!

In particular, you will need to properly **balance out** all the weights and response curves of your decision factors and your conceptual utilities in the entire AI's utility computation tree to get the right prioritisation of actions on evaluation... and as the action set starts to grow, and the utility tree expands, this means you'll have to conceptualise and aggregate quite a lot of variable. In my opinion, utility-based AI is thus an interesting technique for advanced AI programmers who are used to turning ideas into code, and who are happy enough with maths to meticulously adjust each parameter and design the perfect conversion formula.

Another important note is that, as-is, the UBAI system we've described here can only elect a single action each time – it doesn't allow for multiple **concurrent** So, for example, you couldn't model the behaviour of a character walking and shooting at the same time like this. (It is however possible to decouple the locomotion system from the attack system and have two separate reasoners handle each one, to have both actions happen at the same time.)

Moreover, this entire architecture relies on the assumption that the AI (and, actually, the world) is if the AI chooses to do an action, it is because it thinks that this action will necessarily execute entirely and succeed. That is a very common assumption for game AI, of course, and our other techniques

did rely on it too, but here it means that we are really taking decisions based on an **expected ideal**

Typically, consider the

Final Fantasy

series. In those games, when you throw an attack or cast a spell during a fight, you're not always certain it's going to hit. So, although some attacks and spells might technically have the best utility at this point in the fight, if you know there's a high risk of missing the target then, as a player, you might deprioritise them. In order to implement this kind of thinking in an AI, we could try and factor in some notion of **of** into our utility functions, to also account for failures and perhaps make our ranking a bit more relevant.

HOW TO TAKE THE RATE OF SUCCESS INTO ACCOUNT

> If you're curious as to how we could add these success probabilities in the mix, you should have a look at the chapter dedicated to this topic in the amazing Game AI Pro book series, available for free over here: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf.

> In this chapter, David Graham walks us through various UBAI implementation details, among which tainting our utilities with the

> rate of success of the action.

Finally, as we've experienced in our *A simple example* section earlier, while individual variables and response curves are often easy to read and debug, creating a proper mental representation of the entire utility landscape of your AI is way harder. It is possible to a certain extent, and even without plotting there are things like the weights in conceptual utilities that remain intuitive... but there's a limit to this and, with complex multi-layered utility schemes, you might end up guessing some of your tweaks.

Designing proper utility-based AI is as much an art as it is a science, and it usually requires you to iterate numerous times to properly weigh all the action utilities, sometimes even getting off the beaten track and going on a limb, to eventually create more natural behaviours for the AIs.

UBAI is thus a powerful tool with some important gotchas that you need to remember if you don't want it to get the better of you...

Summary

In this chapter, we've introduced a new technique in our game AI toolbox: the utility-based AI.

We've talked about the base principles of this tool, how it compares to the other architectures we've studied so far in this book, and how it allows us to convert contextualised raw data into actual behaviours for our entity.

Then, we applied these concepts to a simple example by re-designing the AI of the dragon guarding a treasure we'd discussed in *Chapter 3* through the prism of UBAI. We saw how to relate data to utility, and we explored how adjusting the parameters of the conversion functions and plotting the matching response curves can help us tune the reactions of our AI creature with a high level of expressivity.

Finally, we took a step back and discussed the perks and the limitations of this technique.

In the next chapter, we will continue working on utility-based AI and study a more complex use case step-by-step – the creation of a wizard AI for a magic duel game...

11 - Designing a utility-based wizard AI

In *Chapter 10: Discovering utility-based* we discussed the fundamentals of utility-based AI, and we saw how this technique is an interesting evolution of the planner systems we'd studied in *Chapter 9: The reverse-thinking of* especially in terms of reactivity and expressivity.

To follow up on this theory, we are going to study a simple magic duel game where two sorcerers face each other in a deadly fight, all pumped up with magic spells. Our goal will be to develop the AI of our enemy, so that we can battle against a worthy opponent, using a utility-based approach.

Let's magic duel!

To begin with, let's describe the main features of our game and understand what our AI needs to do exactly.

The magic duel game we'll explore throughout this chapter is a turn-based solo game where the (human) player plays against a computer (the AI). Both heroes have healthpoints and mana

points, and the goal is to be the first to bring the health pool of the opponent to zero.

The game will be in first-person view, with the enemy mage standing in front of us; and we'll display both some controls for the player in the bottom-left corner and a visualisation of the AI's choices on the right. Moreover, we'll show the current turn (and the magician who needs to play for this turn) in the top-left corner (see *Figure*

To help them in this fight, the magicians can choose among three spells: the Fireball, the Ice Shard and the Heal. (We'll detail those in just a second, in our *Identifying the AI action set* section.)

Our primary focus here will be the computer's AI – we'll consider that the base game mechanics are already implemented, and that everything is handled in terms of inputs to allow the player to play when it's her turn. So the point is just to design a good enough brain for our enemy to actually cast spells among those three, and provide an interesting magic fight experience in this solo game.

This situation could of course be modelled with all the tools we've seen so far – a state machine, a behaviour tree or even a GOAP/HTN planner... but here, we'll see how to do it by applying a utility-based philosophy.

Figure 11.1 – Let's study a little turn-based magic duel game in first-person view!

Understanding the game context

The game is turn-based: we, the player, always start first; and then once we've executed an action, the enemy plays by executing one action, and then it's back to us, etc.

The game progression is handled by a C# class already placed in the scene, the This script is also in charge of locking or unlocking the UI to clearly identify whose turn it is, and checking whether one of the two sorcerers is down to 0 healthpoints, in which case we will get either a victory or game over panel.

Parallel to this each wizard also has its own script: a PlayerManager (on our end) or an EnemyManager (on the AI's end). The AgentManager is a high-level class that takes care of updating the mage's stats, starting the right animations and instantiating the proper spells VFX to really sell the magic fight. The PlayerManager and EnemyManager child classes specialise some of its methods to adapt them to either a player input-controlled or an AI-controlled scheme.

Finally, in order to define the properties of our three spells (such as the amount of damage or heal, the mana cost, the VFX to show...), we have a Spell class that can be instantiated for each spell type, and that allows us to easily edit the characteristics of every skill to properly balance out the gameplay. The Spell class also contains a static LIB Dictionary with all the available spells referenced by name, which is filled automatically when the game first starts thanks to the This means that, for example, we can get back the data for our Fireball skill in any other part of the codebase by calling:

```
Spell fireball = Spell.LIB["Fireball"];
```

These various classes are not directly related to the AI system of our game, and we won't need to know exactly what's in each of them to craft our enemy's behaviour – so let's just leave it that for now, and assume that we have all these managers taking care of the game flow for us.

WANNA CHECK OUT THE CODE AND ASSETS?

For this magic duel game demo, I've downloaded a very cool 3D mage model by *gadohoa* (https://www.cgtrader.com/free-3d-models/character/fantasy-character/mage-model, under the Royalty Free CG Trader license), as well as some UI elements from *PaperHatLizard*'s Cryo's Mini GUI set (https://paperhatlizard.itch.io/cryos-mini-gui, under CC4.0 license). The VFX come from *Jean Moreno*'s Cartoon FX Remaster Free Unity package (https://assetstore.unity.com/packages/vfx/particles/cartoon-fx-remaster-free-109565).As usual, don't forget that you can check the full code and assets for this chapter and its dependencies in the Github repository of this book, over here: https://github.com/MinaPecheux/Ebook-Unity-AIProgramming.

Identifying the AI action set

Just to be clear on what our AI should be able to do, let's have a quick look at its possible options, its final action set.

'Cause we know that our magicians can cast spells – still, what exactly do those spells do? Well, the Fireball, Ice Shard and Heal skill each have their own mana costs and effects:

The Fireball (2 mana) deals 2 damage to the other sorcerer.

The Ice shard (2 mana) deals 1 damage to the other sorcerer, and freezes him for one turn, which increases the mana cost of its next spell by 1.

The Healing (1 mana) touch restores 3 health to the magician who casts it.

(As the player, when it's your turn to play, you can choose and cast one of these spells by clicking the matching button in the bottom-left corner, or pressing the matching hotkey.)

A spell can only be cast if the sorcerer who's currently playing has at least as many mana points as the spell's mana cost (plus 1 if the sorcerer is frozen).

Moreover, each sorcerer can choose to skip the current turn to restore 2 mana points. This can be an interesting strategic choice if you're not too far behind in terms of damage dealt and/or health lost, and you'd rather accumulate some resources for the upcoming turns. (As the player, you can skip the turn by clicking the **End turn** button in the top-left corner, or pressing the spacebar.)

When we are finished with the implementation, our AI will thus have four possible actions, just like us:

Casting the Fireball spell.

Casting the Ice Shard spell.

Casting the Heal spell.

Skipping the turn to restore mana.

We'll consider that, if no action is picked, then the AI defaults to its idle action and skips the turn.

So, in short, the point will be to define the right utility functions for each spell to have the AI focus more on dealing damage, freezing the opponent, healing itself, or planning ahead and keeping some mana at its disposal.

We will do this gradually by implementing one spell at a time, taking advantage of the high modularity of UBAI. As we study this example, we'll also have a chance to discuss gameplay balancing and AI expressiveness.

Yet before we talk about our magician's AI *per* we first need to explore the tools we'll need for creating a utility-based AI system, at a more meta level...

Setting up our UBAI architecture

After this little introduction to our base game mechanics and context, let's dive into the implementation itself. And before coding up our own AI, let's discuss the toolbox we'll be using to design this AI mage brain.

In truth, because utility-based AI is quite a recent idea in the game dev community, there isn't as many examples of how to program UBAI as there are finite state machine or behaviour tree implementations.

If you look around the Internet, you'll find dozens of FSM projects and BT examples, and you'll see that there are numerous projects (either free or paid in asset stores) that make it easy to integrate those tools in your Unity/C# projects. For utility-based AI, on the other hand, not so much. So far, I personally haven't found a lot of references on how to actually program UBAI (apart from the

book series which is always an amazing resource when working on that topic, and where David Graham shares a demo project similar to ours, but in C++:

All of this means that, this time, we don't have any "conventional tricks" or "common techniques" to follow for writing up a UBAI system. So, for this demo game, I decided to just go for it and make up my own utility-based AI library from scratch, with a modularity-first philosophy.

In this section, we're going to have a quick look at my UBAI C# package, including the tools it relies on, some of my design choices, some of the limitations of my system, and a few possible improvements we could think of to make this library more powerful.

EXPLORING THE CODE

> The point of this section will not be to dive into the code, though – we'll mostly focus on the design ideas, not their implementation. So if you're curious about the exact implementation details, feel free to have a look at the Github of this book and checkout the Dependencies/SOMD/ and Dependencies/UBAI/ folders!

Using Scriptable Objects for modularity

My main goal with this UBAI package was to create easy-to-combine elements that would allow designers to create and

maintain a UBAI logic for their entities without having to code (or at least, as little as possible).

For this, one of my base tool were **Scriptable**

What are Scriptable Objects?

In a nutshell, Scriptable Objects are a special type of data containers in Unity that make it easy to define our own custom data types, and then create, update, share and save data instances in the editor thanks to clean UIs.

Rather than having to code up complex C# classes that only your developers can modify and maintain, with Scriptable Objects, you just have to prepare some global blueprints for your data, and then you can instantiate those blueprints to make actual data sources. Then, you can tweak the data inside those sources in a no-code fashion, just by manipulating asset files in your project. And because the data is stored and serialised in this asset specifically, it essentially acts as a **single source of truth** that can then be accessed from various places in your game to read from or write to this data instance, while keeping the info consistent everywhere.

Those Scriptable Objects can be used in many ways.

A very common usage, for example, would be to define a custom PlayerData type in a C# script, and then instantiate this class as a new Scriptable Object asset in the project to have each field in the blueprint be editable and accessible easily in the editor:

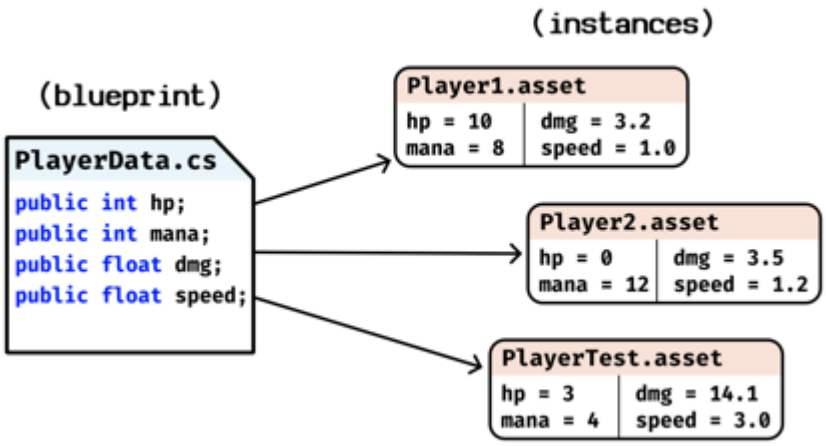


Figure 11.2 – Basic visualisation of the C# blueprint class and matching instances for an example PlayerData Scriptable Object

To create these objects, you'd just need to:

1. Setup a new PlayerData.cs file in your Unity project with the following code inside:

PlayerData.cs

```
1  using UnityEngine;
2
3  [CreateAssetMenu(menuName = "Scriptable Objects/Player Data")]
4  public class PlayerData : ScriptableObject {
5      public int hp;
6      public int mana;
7      public float dmg;
8      public float speed;
9  }
```

This snippet shows us how to declare a Scriptable Object C# blueprint. The class has to inherit from the built-in ScriptableObject C# class (imported from the UnityEngine package), and it has to contain serialisable fields (i.e. either public fields, or private fields with the [SerializeField] attribute). Plus, if you want to be able to instantiate it from the contextual creation menu in the Unity editor, it has to have a [CreateAssetMenu] attribute to specify the path of the associated menu item.

2. Back in the editor, right-click in your **Project** dock, then go to the **create** > **Scriptable Objects** > **Player data** submenu and select this last item to create a new instance (you see that the

path matches the one we defined in our [CreateAssetMenu] attribute):

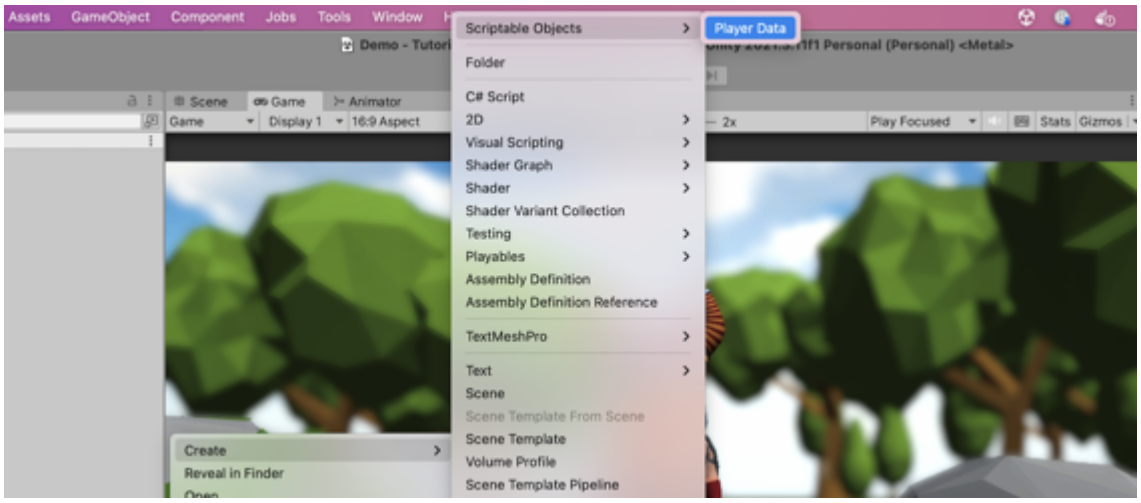


Figure 11.3 – Custom creation menu for instantiating a Scriptable Object-defined data structure

3. Select the newly created asset and then edit its values in the **inspector** panel:

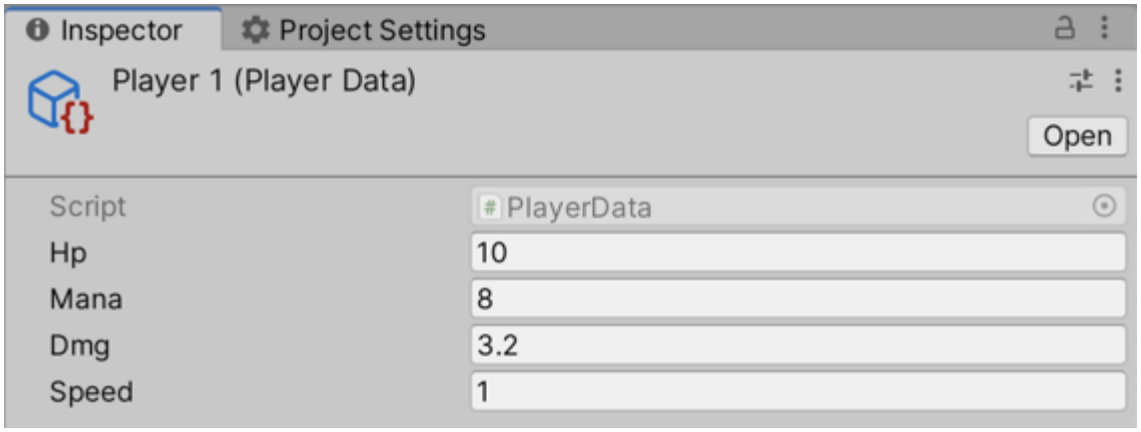


Figure 11.4 – Inspector of a PlayerData instance: it contains the serialised fields we defined in the PlayerData C# blueprint script

Using a Scriptable Object for all the player's data like this is nice because it bundles all of the player-related information in a single place, and it lets us super easily display some of our variables in the UI, update them from some manager script or even manually set the values in the inspector to test a precise scenario.

The only issue with that method is that if you want to change the fields of your PlayerData structure afterwards (say you want to add an integer for the current level of the player, or an extra energy amount), you'll have to go and re-update the PlayerData C# script containing the blueprint structure. It's okay if you're a coder, or if you can ask the ones in your team to do the change, but it does break the flow a little for non-dev designers.

That's why another possibility is, rather than using a single asset for storing all the data in one structure, to instead chop down your data in many small data asset files that each represent just a single field, and altogether represent the same piece of data:

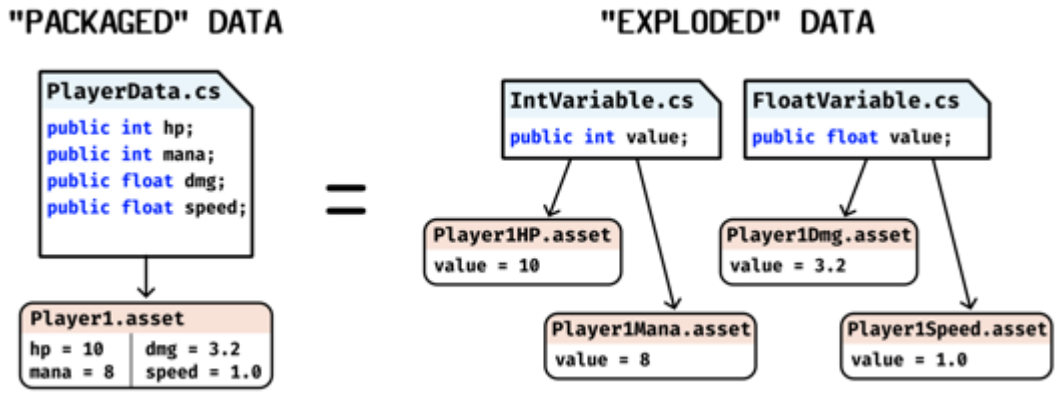


Figure 11.5 – Alternate organisation of the Scriptable Object blueprints and instances using more low-level variable types to explode the data and get re-usable core blueprints

With that new setup, your Scriptable Object C# classes become simple low-level variable types, that you can then instantiate as many times as you need to create the different pieces of information in your game's context.

Of course, it means you need to manage more files in your project... but as a designer, it also empowers you to setup everything on your own, without waiting on any developer, just by adding, removing or updating these little data bricks in the Unity editor!

WANT TO LEARN MORE ABOUT SCRIPTABLE OBJECTS?

> I'm not going to go into too more details about Scriptable Objects here because they're very Unity-specific and they're not directly related to AI programming. However, if you want to learn more and get a refresher on the perks and caveats of Scriptable Objects, you can have a look at the tutorial I made on that topic:

> - As a YouTube video: https://www.youtube.com/watch?v=ZnHxxADBAQ0

> - Or as a blog post on Medium: https://medium.com/codex/discovering-the-power-of-unitys-scriptable-objects-53ae6e0acef4

Creating Scriptable Object-based variable types

The first step, before diving into utility-based AI notions, was thus to use this "mini-data-brick Scriptable Objects" philosophy to create my own little C# library of low-level Scriptable Object-based variable types: the SOMD C# package (for "Scriptable Object-based Modular Design").

Basically, my goal was to prepare C# classes for the **common variable types** IntegerVariable and – and then, I'd just have to instantiate those classes to create all the string, float, integer or boolean game variables I needed.

I decided to create a base type for all C# variable type classes to share some logic between all of those, and most notably put in place an **event-based update system** that allows me to easily warn all the components that depend on a specific piece of data that this asset was just updated, and they should re-check their data source to get the new value.

Using events is a neat way of keeping objects decoupled and self-contained, and yet allow for some communication between the different systems. To put simply, events let us fire a warning from the modified Scriptable Object asset at the time of the change in our codebase, and this warning can then be caught by some interested script somewhere else to react to this change. But if no one is listening to this event, then it will just be forgotten and won't disturb the rest of the systems. We therefore work in a very loosely coupled paradigm where the info *can* be used, but it can also be ignored without any missing reference issues (see *Figure*

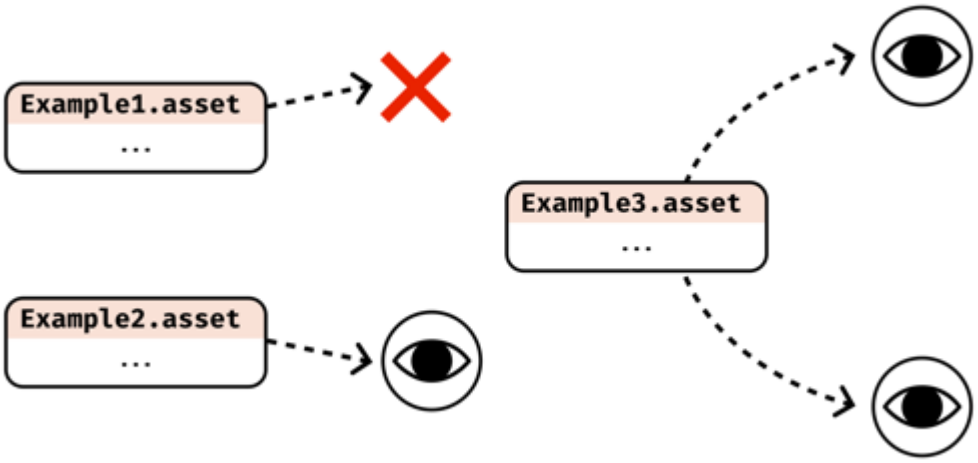


Figure 11.6 – Events may have zero, one or more observers/receivers; they can be invoked no matter what, without any missing references, in a "fire-and-forget" way

The SOMD library also contains a few basic "visualisers". Those scripts can be used on UI components to display the contents of our low-level data bricks in common ways – typically, a formatted label, a conditional text depending on a bool, a filled bar, etc. Part of the UI shown in *Figure 11.1* is actually powered by these scripts. (Note that those components are compatible with Unity's Canvas-based UI system, and most of them require the TextMeshPro library.)

So, to sum up, this package allows us to craft an entire pool of data variables just by instantiating and editing Scriptable Object assets, and then easily visualise the data inside those assets at runtime by dragging in some pre-made C# scripts on our UI elements. And thanks to our event-based update

system, we are sure that whenever we change the value of one asset variable in the inspector or in our scripts, a warning will be propagated to all interested listeners so that they can react to the change appropriately (e.g. to refresh the UI).

Creating different utility evaluation tools

But, of course, this SOMD package was just the beginning; I then focused on UBAI-specific objects and prepared some tools that, hopefully, should cover most of our use cases. The idea was to define more advanced Scriptable Object-based structures to describe our utility computations.

When I designed the UBAI library, I wanted to have it work seamlessly with the SOMD package. In particular, I wanted the UBAI Scriptable Objects to accept both other UBAI objects and SOMD variables as their decision factors, to allow users to build utility tree iteratively with a minimal amount of variable types.

To allow for this mix, I implemented a base Evaluatable type in my SOMD package for all my Scriptable Object C# blueprint classes to inherit from, and that contains an Evaluate() abstract function. This function has to be implemented in the derived classes and it will allow us to get the “value” of the Scriptable Object instance as a which can be one of two things:

If the Scriptable Object is a low-level variable type from the SOMD package, then it's simply its value field converted to a (The operation is undefined for the StringVariable class.)

If the Scriptable Object is a UBAI object, meaning that it is an asset that computes a utility, then the Evaluate() function returns this utility score – the exact calculation depends on the type of UBAI object.

So, thanks to this base Evaluatable type and its Evaluate() method, we can easily reference other SOMD or UBAI Scriptable Objects instances inside one another, and build a utility tree by recursively embedding variables or previously computed utilities as decision factors for a new utility computation.

For clarity's sake, *Figure 11.7* shows an overview of all the Scriptable Object classes we have in the SOMD and UBAI package, plus the base Evaluatable type:

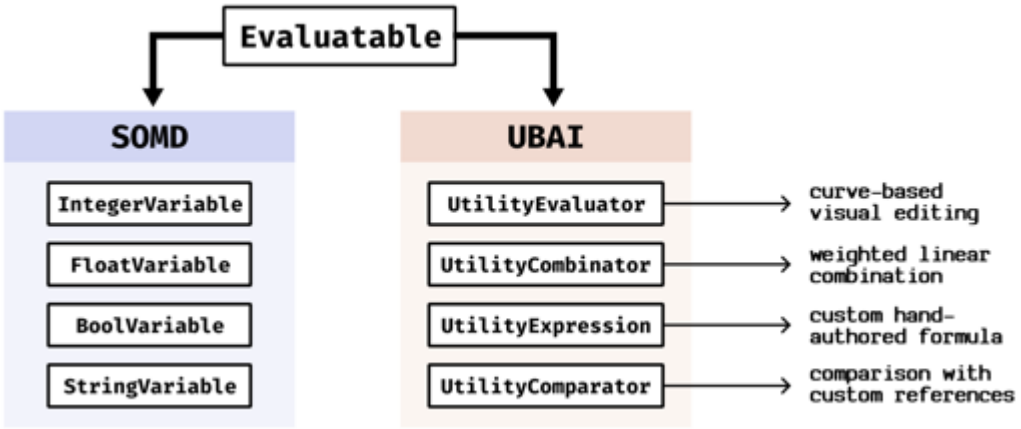


Figure 11.7 – Overview of our SOMD and UBAI Scriptable Object C# blueprint classes

And *Figure 11.8* shows an example of a small-sized utility tree mixing various SOMD and UBAI object instances without distinction:

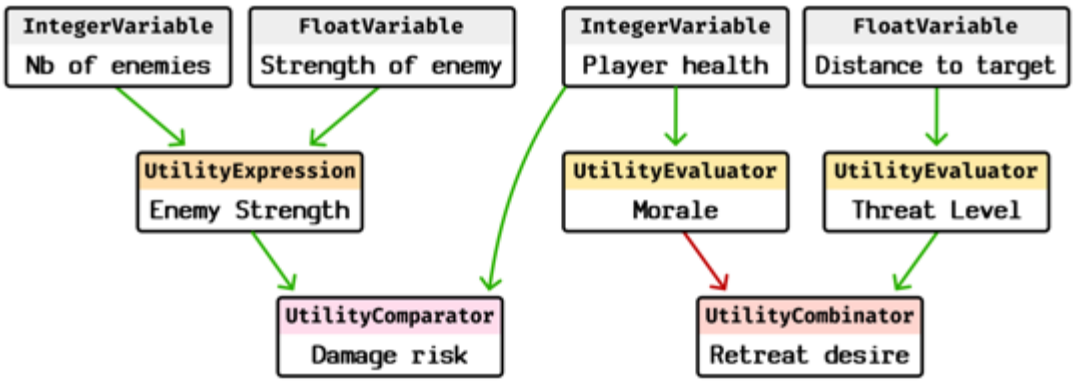


Figure 11.8 – Example utility tree mixing together SOMD and UBAI Scriptable Object instances

You see in these diagrams that the UBAI utility scoring objects can be of four types: UtilityExpression or Although they all compute a utility score based on one or more data points, each of those offers us a unique method for defining the calculation:

First of, the UtilityEvaluator relies on a Unity AnimationCurve to provide us with an easy-to-use visual editor to directly define a response curve. This tool is a great way to define a highly detailed custom rate of change for our utility by setting up specific keyframes, and evaluating the resulting curve at a given input.

For example, we could use this UtilityEvaluator to craft a threat level utility response based on the current distance between the AI and its target:

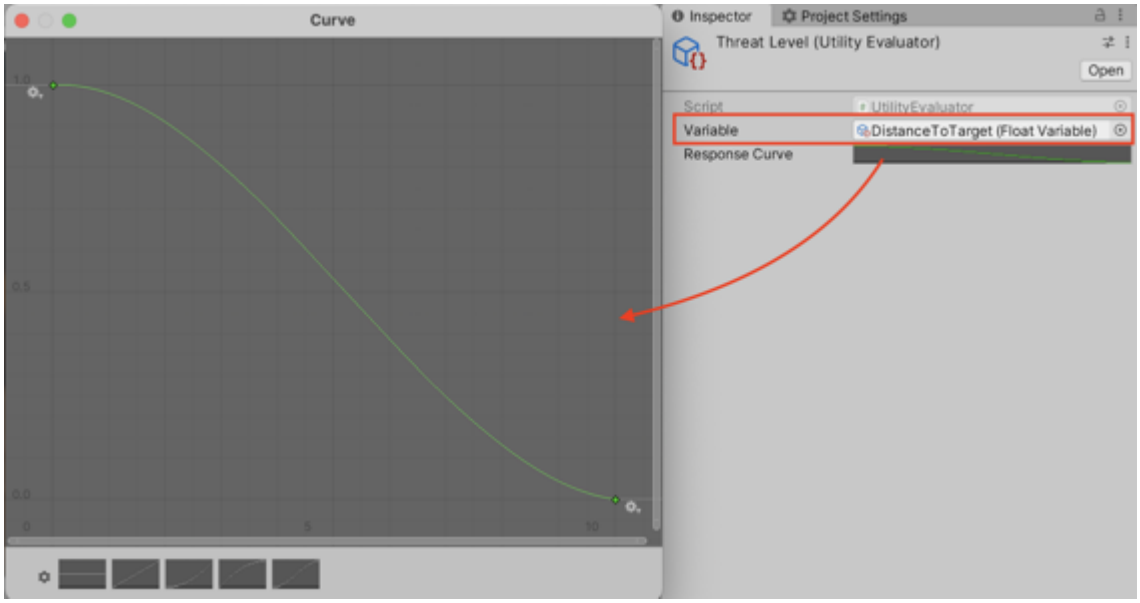


Figure 11.9 – Example of a UtilityEvaluator instance, remapping the DistanceToTarget decision factor to a normalised value via a logistic function

You see in *Figure 11.9* that, to use this UtilityEvaluator tool, we just need to instantiate a new asset, pick the Scriptable Object instance to use as the decision factor for our utility computation, and finally edit the Response Curve field. And the advantage of using a Unity AnimationCurve is that we have this built-in curve editor to quickly add, modify or remove keyframes; we can also move around the handles of those keyframes to change the curvature of the line and adjust it to our liking.

Note that the input variable has to be a reference to an Evaluatable Scriptable Object.

A slightly more evolved utility computing asset is the which lets us define a sum of weighted decision factors as a linear combination. Meaning that we'll get something of the form:

combined_utility = * + * + * + ...

Where are the weights of each factor.

The weights can be either positive (the factor has a positive impact on the utility) or negative (the factor has a negative impact on the utility), and the larger the weight the greater the influence of the factor on the utility score. To make sure the result stays within the expected range, we can also turn on the Has Min and/or Has Max toggles, which reveals two number inputs for those extreme values – if either is enabled, the value of the utility will automatically be clamped to obey those min/max rules.

Typically, if we wanted our previous ThreatLevel individual utility to be compound with another Morale individual utility into a conceptual RetreatDesire utility, we could create a basic linear combination with various weights like this:

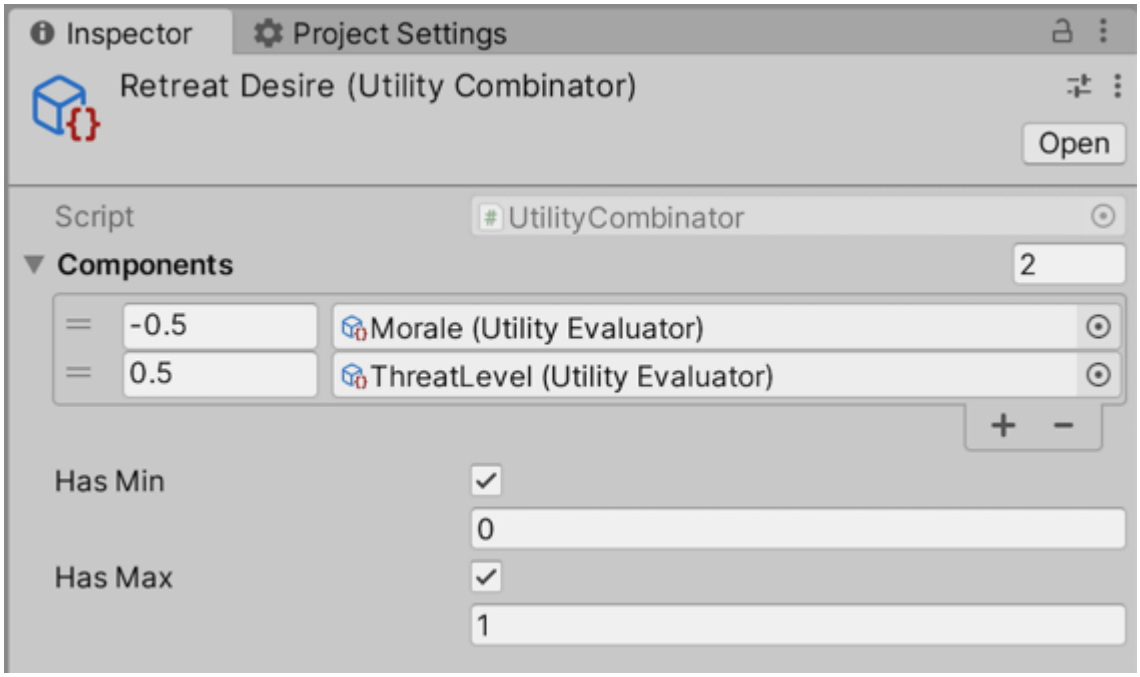


Figure 11.10 – Example of a UtilityCombinator instance, mixing the Morale and ThreatLevel individual utilities into a higher-level conceptual utility

In this example, the result of the evaluations of the Morale and ThreatLevel Scriptable Objects are mixed with a negative and a positive weight respectively, so that the unit's desire to retreat increases when the morale degrades or when the threat level goes up. We also ensure our final result remains in the 0-1 range thanks to the auto-clamping feature.

Like before, the UtilityCombinator expects its decision factor(s) to be reference(s) to Evaluatable Scriptable Object(s).

Even if they are handy and quick to use, the UtilityEvaluator and UtilityCombinator logics can only go so far. At a certain point, they'll be too limited to accommodate for all our fantasies. Most notably they don't allow us to multiply or divide factors together, which could clearly be useful.

That's why I also prepared a super adaptable utility computor: the In short, this tool takes in a list of decision factors as references to Scriptable Objects and a mathematical expression written out as a string, and it inputs the values of the factors into this expression to compute the result.

In order to get as flexible a system as possible, I've based my UtilityExpression on C# This built-in tool makes it possible to define a dynamic set of columns on-the-fly and then insert a row of data in it; and those columns can contain string expressions that will get evaluated and turned into actual results when you add in the data. What's even cooler is that you can use the value inside other columns inside your expression, kind of like local variables – which, for us, means that we can build expressions based on the result of other SOMD or UBAI Scriptable Object instances.

The idea is simply to list the decision factors we're interested in by picking the references to their Scriptable Object instances, associate each object with a local variable name, and

then input the mathematical expression for our utility score using those local names:

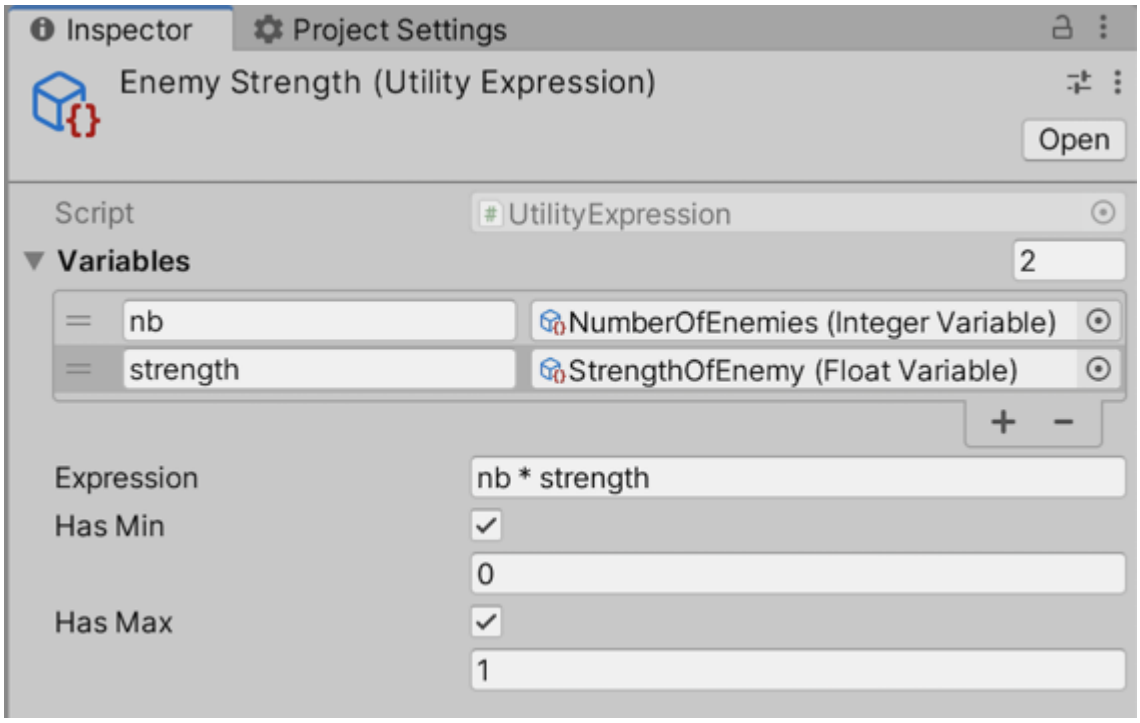


Figure 11.11 – Example of a UtilityExpression instance, which is the product of the NumberOfEnemies and StrengthOfEnemy base variables

Again, we can have the object auto-clamp the result to a specific range if need be. However, it is important to note that, because we're using DataTables under the hood, our expressions can only contain the operators, aggregates and functions defined in this page of the Microsoft C# docs:

Now, contrary to the UtilityEvaluator and UtilityCombinator that only accept Evaluatable instances as inputs, here I've voluntarily widened the scope by allowing any Scriptable Object instance to be picked in the inspector as a variable. It reduces the level of control a little, but it also allows for a pretty cool feature: a direct access to another source of data's field value.

Basically, I've made it so that just by including a dot character inside the local variable's name, we can tell our UtilityExpression to go look inside the given Scriptable Object asset and fetch the value of one of its fields, using this name reference:

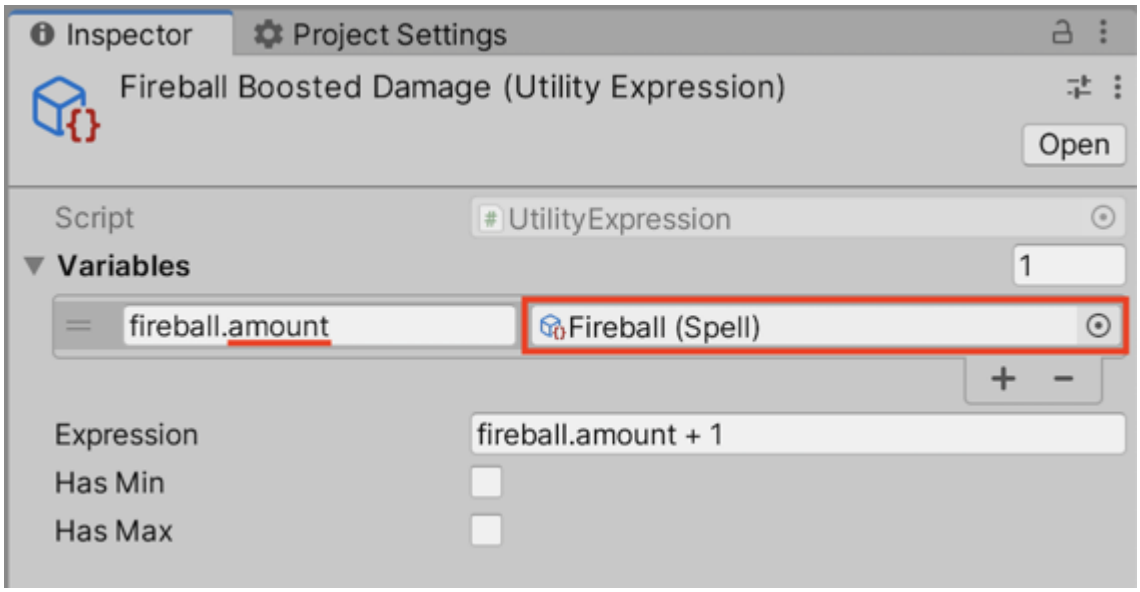


Figure 11.12 – Example of a UtilityExpression instance, based on the amount field of the Fireball Scriptable Object instance

Here, this UtilityExpression instance would take our Fireball Scriptable Object, extract its amount field value, add 1 to it, and then return the result. You see that what comes before the dot doesn't matter, but what comes after has to be the name of the field to fetch inside the target Scriptable Object (in my case:

This tool will make it easy to mix together our utilities using custom formulas and even directly reference other pieces of data stored as Scriptable Object to ensure total consistency and avoid redundancy.

Finally, the UBAI package contains one last type of utility computation asset – the As the name implies, it is meant to compare values or expressions and return a "boolean-like" result – i.e. a float value of either 0 or 1.

Just like the the UtilityComparator relies on a DataTable and it can thus take in a variable amount of inputs and inject them into written out string expressions. Except this time, we have a left expression, a right expression and a comparison operator (see *Figure*

Those three elements define the condition to check for, and when this asset gets evaluated all the expressions are converted to float values thanks to the underlying and compared according to the defined operator (among:

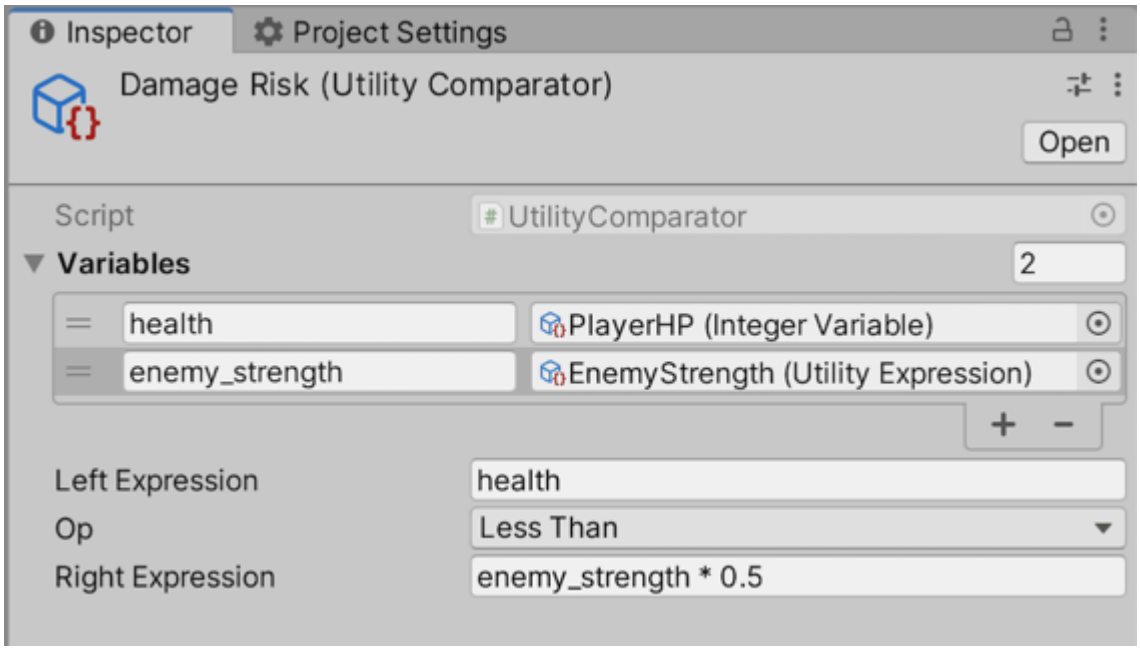


Figure 11.13 – Example of a UtilityComparator instance, which compares the current health amount to the total strength of the enemies

By instantiating those UBAI Scriptable Objects blueprints, as well as some SOMD objects, we'll thus be able to describe our game context as a pool of base variables, and then assemble those variables into utilities, and then again re-assemble those utilities and variables into higher-level utilities, until we eventually define our entity's final action scores.

But then, to actually use those action scores, rank the action set and identify the best option among the ones available, the AI needs to have a utility-based brain...

Preparing the UBAI brain

Last but not least, the UBAI package contains a class to use all of this utility data and run some logic – the UBAIBrain C# class. This script is simply in charge of listing the possible actions of the entity and electing the best one at any given time by evaluating the whole utility tree and finding the best option.

This election process can follow various depending on how you'd like the script to pick the best action. The most common policies are:

The simplest election policy is, once you've ranked all your action utilities, to just choose the one with the highest utility. This ensures that your AI does what is best for it (according to your utility tree design), but it also means that the entity will always react exactly the same when faced with a similar situation. This policy can therefore create fairly predictable behaviours.

Weighted To somewhat mitigate this predictability issue, it is possible to have the AI choose a random action in a clever way. More precisely, we can use the utility scores of our various actions as weights and then do a weighted random pick based on those values. This will make the most useful

actions more probable, but it won't totally prevent the other options from being chosen and therefore allows for a bit more variability.

Except that if you consider the entire action set in your weighted random, every so often the AI will just do something completely irrational with a super low utility – just because it can.

Weighted random in a A possible solution to this last problem is to limit your weighted random selection to only a subset of actions, the ones with the highest utilities. To do this, you can simply take the "top actions, or you can define a threshold for the minimal utility to consider (ideally, this threshold should be dynamic to better fit your values, so something such as "10% below the highest utility in my list").

In my UBAI package, I focused on two efficient policies: the Best and the WeightedTopN policies, which I defined in a C# enum called This parameter (as well the size of the top to use) can be passed to the UBAIBrain instance in its constructor.

Of course, in our case, given that we only have a few possible actions, picking in a top seems a bit overkill. At most, we could have the mage choose randomly between its two best options, to try and avoid the spamming of a single spell. But

as we'll see in the upcoming section, I've actually decided to stick with the Best policy for this example.

The very last object worth peeking at in the UBAI package is a core and yet very simple element: the UBAIAction itself.

Defining our UBAI actions

Compared to all the other classes in the UBAI library, the UBAIAction is extremely short and easy-to-understand:

UBAIActions.cs

```
1  using SOMD;
2
3  namespace UBAI {
4    [System.Serializable]
5    public class UBAIAction {
6      public Evaluatable utility;
7      public string func;
8      public Evaluatable[] vetoes;
9    }
10 }
```

Apart from the package includes and namespace definitions, we see that this script just defines our UBAIAction class as serialisable so that it appears in the Unity **inspector** panel, and declares three fields inside it:

The utility computation asset to use for evaluating the score of this action.

The name of the callback function to run if this action is elected as the best option.

An array of Evaluatable Scriptable Objects that return a "boolean-like" value, and may completely block the AI from considering this action.

The two first properties are straight-forward; the third one, however, deserves a bit more explanation.

As we've said in *Chapter* a tricky thing with utility-based AI is that, by default, there will always be a "best" action. Even when all utility scores are nearly zero, there will always be one that is slightly higher. And yet an action with a utility of 0.001 can't decently be considered useful.

This is why, sometimes, it may be interesting to explicitly prevent an action from being taken into account in the election process. This is usually done with one of two common techniques:

An obvious solution is to just write a veto condition – if you run a check on something and it returns then you simply skip this action in your ranking altogether. This will effectively disregard all options that are "invalid" at that time, and only compute the utility of the ones that could be executed.

Another possibility is to group one or more actions in "buckets", and assign those buckets overall weights. Then, you process your buckets from the highest weight to the lowest weight and, if any action is valid in a bucket, you pick it above the ones in lower-weighed buckets. When it has various "modes" (e.g. resting with a coffee, defending the room, leading an assault...), this will give the AI a big incentive to focus on one of those modes in particular and disregard the other ones.

This bucketing technique, also known as **dual utility** is typically used in

---

The Sims

: if it's so hungry its life is in danger, a Sim won't even think about playing a video game or watching the TV – it is focused solely on solving this hunger problem.

In my UBAI C# package, I've decided to go with the vetoes because it is easier to implement, and it will be enough for use case. (Bucketing shouldn't be too hard to add, but it does require the users of the lib to specify those buckets, and here we'd probably end up with four buckets each containing one action, anyway.)

For any action, it is therefore possible to give it a list of references to SOMD or UBAI Scriptable Object instances that return a "boolean-like" value, which will all be evaluated when the AI has to take a new decision. If the veto's value is 0 (meaning then the process aborts instantly and the action is simply ignored; if the value is 1 (meaning then the process continues. The action will only be considered and scored for utility if all of its veto conditions pass (i.e. return 1). We'll see how this works in more details in the *Checking for forbidden actions and making our AI evolve* section at the end of this chapter.

This UBAIAction class isn't complex, but it will be sufficient for our wizard AI... and thanks to our very modular system of Evaluatable Scriptable Objects, we should be able to craft and tweak our utility tree as need be to design our AI sorcerer's brain.

So, with this overview of the SOMD and UBAI packages in mind, let's keep going and explore how to apply those tools to our own example: the creation of an opponent player AI for our wizarding duel game...

HAVE SOME FEEDBACK ON THE UBAI PACKAGE?

> I hope you'll like the UBAI architecture proposed here, and that it will give you some ideas for further experimentations! As usual, if you have any feedback or remarks, don't hesitate to send them over by email (at mina.pecheux@gmail.com) or even share them on the Github of this book (https://github.com/MinaPecheux/Ebook-Unity-AIProgramming) so that other creators can participate too :)

Implementing our base wizard UBAI

Now that we have a better idea of how these custom SOMD and UBAI C# packages work, it is time to actually apply them to our little magic duel game and build our opponent's AI based on utility. We will do this gradually, by adding more and more spells into its action set.

And to start off, let's take care of creating our wizard's UBAI brain and allowing it to cast a fireball!

Setting up the brain instance

To begin with, we'll consider that the enemy sorcerer is already in the scene with its model and various components such as the Animator or the EnemyManager script, but that these don't tie into any real logic.

Basically, at this point, the Evaluate() function just returns true all the time, which means that the AI is simply skipping every turn:

EnemyManager.cs

```
1 public class EnemyManager : AgentManager {
2
3     public bool Execute() {
4         return true; // skip turn
5     }
6
7     // ...
8 }
```

Our goal is therefore to setup our own class and create an instance of this child class in the EnemyManager to really define and evaluate a utility tree.

Step one is to create a C# script for this custom mage AI, so we'll go to our project files and create a new MageBrain.cs file. Then, inside, we'll remove the startup code, import our UBAI package and make our empty class derive from the UBAIBrain class:

MageBrain.cs

```
1  using UBAI;
2
3  public class MageBrain : UBAIBrain {}
```

In order for our script to compile, we need to define a constructor that at least passes on the right parameters to the parent UBAIBrain constructor (namely, the election policy and the list of UBAIAction instances):

MageBrain.cs

```
1  using UBAI;
2
3  public class MageBrain : UBAIBrain {
4      public MageBrain(UBAIAction[] actions)
5          : base(ElectionPolicy.Best, actions) {}
6  }
```

But, in fact, we also want to keep a reference to the EnemyManager that contains and uses this brain, because we'll want our data to eventually translate into actual animations, VFX and game updates, so we need to have access to this manager script:

MageBrain.cs

```
1  using UBAI;
2
3  public class MageBrain : UBAIBrain {
4      private EnemyManager _agent;
```

```
6     public MageBrain(EnemyManager agent, UBAIAction[] actions)
7         : base(ElectionPolicy.Best, actions) { _agent = agent; }
8 }
```

Now, we want to have our UBAI actions trigger some real effects when they are picked. For this, we need to define specific callback functions in our MageBrain script.

For now, we'll create a _CastFireball() callback. Be careful because it has to be **parameterless** so that it can be called automatically by our parent class Evaluate() function when this action is picked. (Behind the scenes, I'm using Unity's Invoke() built-in which only supports calling methods with no parameters.)

This _CastFireball() will just forward the call to a more generic _CastSpell() function that receives a particular spell type and calls the right methods on the associated

---

MageBrain.cs

```
1 using UBAI;
```

```
2
3  public class MageBrain : UBAIBrain {
4      private EnemyManager _agent;
5
6      public MageBrain(EnemyManager agent, UBAIAction[] actions)
7          : base(ElectionPolicy.Best, actions) { _agent = agent; }
8
9      private void _CastFireball() { _CastSpell(Spell.LIB["Fireball"]); }
10     private void _CastSpell(Spell spell) {
11         _agent.Cast(spell);
12         _agent.HighlightSpell(spell);
13     }
14 }
```

Our wizard is now ready to cast a fireball... except it doesn't actually have it in its action set, and it has no idea what the utility of this action is! The next step is thus to use the different utility evaluators from the UBAI package we discussed before to craft a first version of the sorcerer's utility tree.

Defining the score of our "cast fireball" action

Ok – for this first iteration, we'll keep things simple and say that the utility score of casting a fireball depends on a single decision factor: the attack desire of our entity. This individual

utility will be computed based on the current amount of healthpoints of the player; the idea will be to have the attack desire increase as the player's HP decreases, to simulate the AI's willingness to end the fight by dealing a fatal blow.

For the sake of simplicity, let's assume that we already have our core game variables defined in the project as Scriptable Object assets thanks to low-level variable types of the SOMD package – most notably, we have the stats of each wizard, plus a few global game variables like the current turn or the maximum amount of damage sorcerers can inflict, which will be useful later on when we extend our entity's action set:

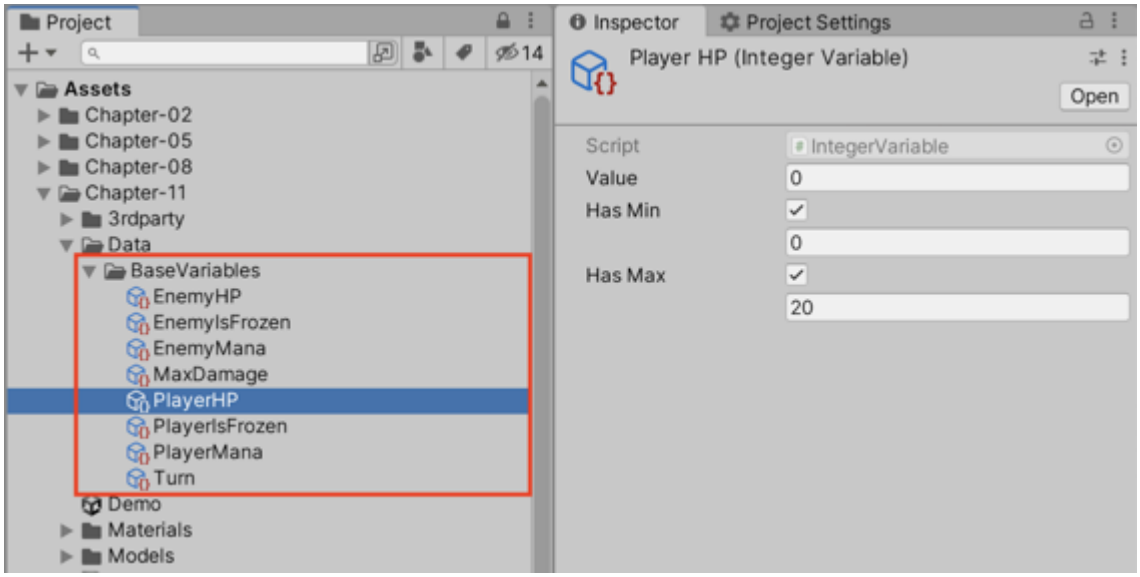


Figure 11.14 – Various asset variables relevant for our game context that are instances of SOMD low-level Scriptable Object-based variable types

Those variables are of different types and they contain some global game data that is updated by the GameManager and the scripts in the scene continuously to stay consistent with the current game state.

For example, the PlayerHP variable that we want to build our attack desire from is an IntegerVariable automatically clamped to the 0-20 range. (By the way, it is initialised to the max value of 20 when the game first starts to properly reset the player's stats for a new duel.)

So we know that we have to create two utility assets to setup the entire utility tree of our wizard AI (see *Figure*

the attack desire, which is a conversion from the PlayerHP value into a normalised individual utility

the score of our "cast fireball" action, which for the moment is directly equal to the attack desire utility

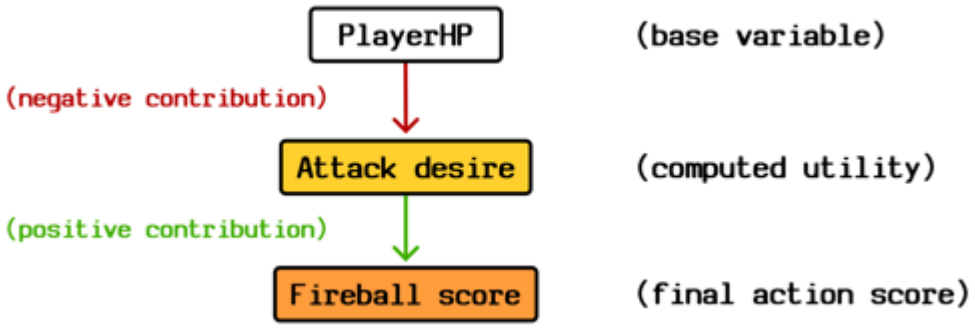


Figure 11.15 – V1 of our AI's utility tree that handles the scoring of the "cast fireball" action

To compute the attack desire of our entity, the easiest solution is to use our UtilityEvaluator tool from the UBAI package: we don't need any advanced formulas, and it will allow us to define a response curve for this conversion formula with a nice and easy-to-use visual editor.

Let's create a new UtilityEvaluator instance in our project and name it AttackDesire – then, we'll link our PlayerHP variable as the input, and click the curve field to open the curve editor and define our response curve:

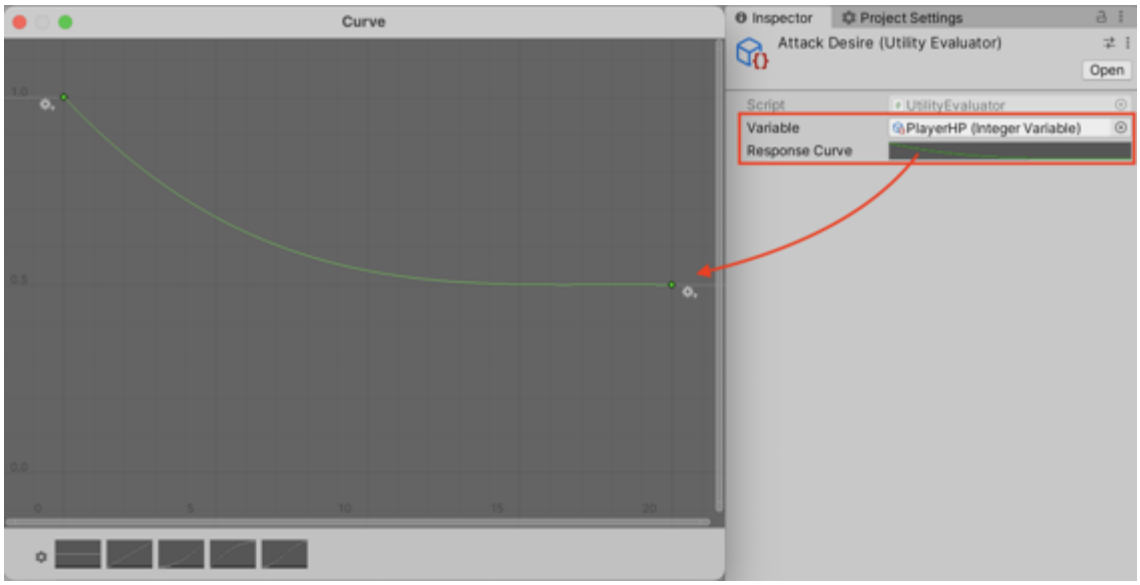


Figure 11.16 – Definition of our AttackDesire utility using the UtilityEvaluator tool

As shown in *Figure* the AttackDesire utility goes from 0.5 when the player's health is maximal to 1 when the player gets closer to zero with a quadratic-like rate of change. Of course, because this is a utility value, is should be limited to the 0-1 range, so make sure that your Y values stay in this range. And also, remember to check that the right point has an X coordinate of 20 (by default, it will be positioned at X = but we want our utility to be normalised based on the max value of the PlayerHP variable, which here is 20).

Now, we have to do the same process for our CastFireballScore utility computation – but, this time, we don't really need to design a precise response curve like this. Rather, we want to have some elbow room for further utility mixing, and possibly a custom formula to really tweak this high-level concept utility to our liking. So I'll instantiate a UtilityExpression from our UBAI package, name it CastFireballScore and, for now, just use my AttackDesire object directly as the result:

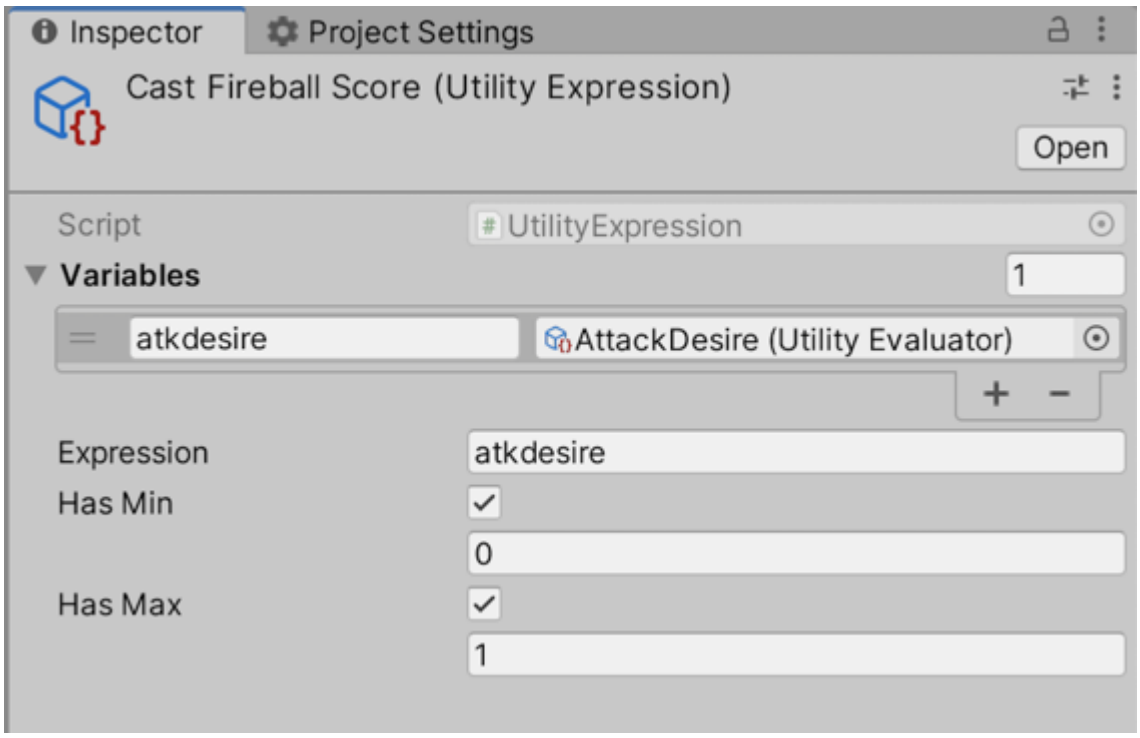


Figure 11.17 – V1 of our CastFireballScore utility using the UtilityExpression tool (just a passthrough for the AttackDesire individual utility)

Again, this value needs to remain in the 0-1 range, so I've made sure to enable my min and max value checks to get an auto-clamping of my result if need be.

This is obviously a bit convoluted for a simple passthrough; however, as we'll make this scoring computation slightly more complex in the rest of the chapter, it's better to be prepared and anticipate by taking our most tweakable utility tool.

Anyway – we've now chained together our three data bricks from *Figure* and we have but one thing left to do: integrate this utility computing asset in our wizard's action set, and link it to our _CastFireball() callback so that we can trigger the right effects if this action is picked.

This is quite easy to do: we just need to select our enemy game object and, in the inspector of its EnemyManager script component, add a new entry in the exposed actions array. We'll set the CastFireballScore as the utility scoring asset and the string "_CastFireball" as the name of the callback function associated with this action:

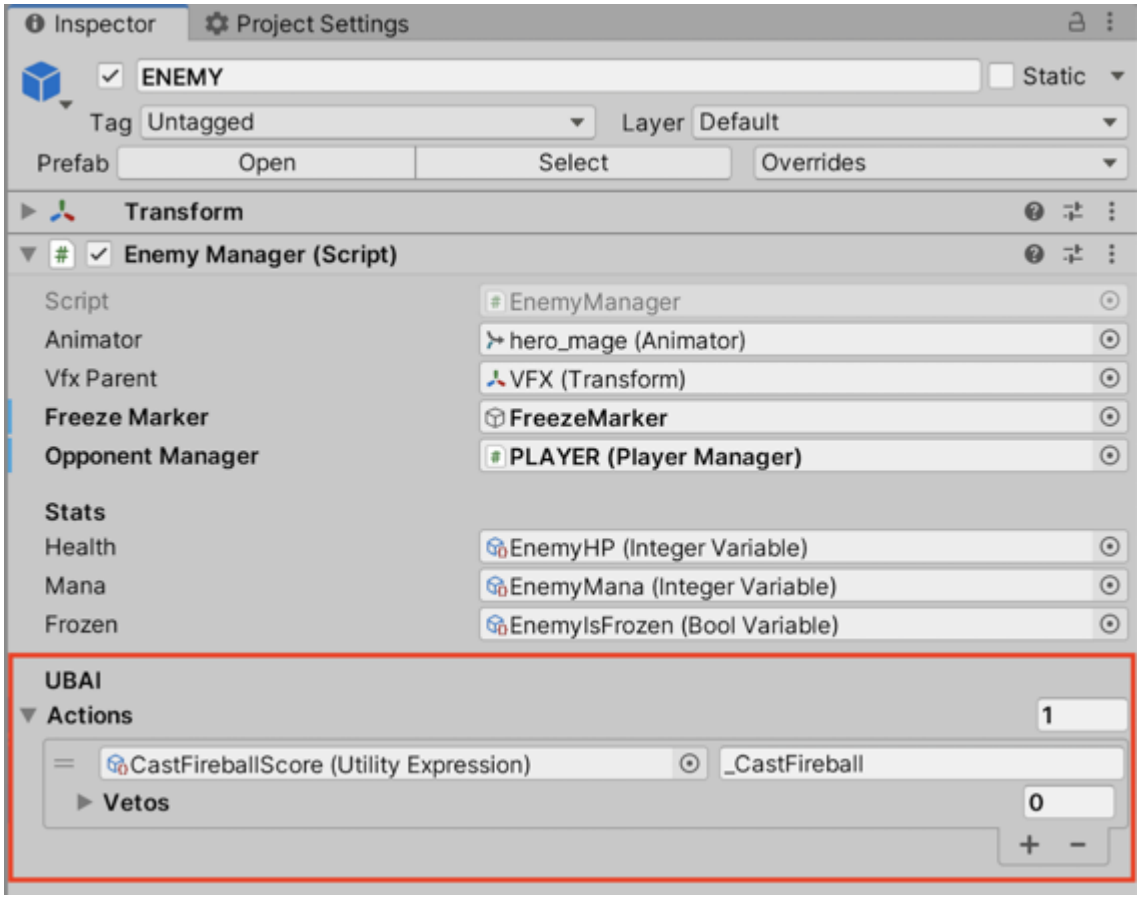


Figure 11.18 – Inspector of the AI's EnemyManager component with the "cast fireball" action integrated, using the CastFireballScore asset as utility computor and the _CastFireball() **method as callback**

(We won't bother with the vetoes mechanic right now – we'll talk about this later on, in the *Checking for forbidden actions and making our AI evolve* part.)

And here we are! Our opponent is now able to evaluate the utility of its "cast fireball" action, elect it as the best action (since it doesn't have any other choice currently...) and run the matching _CastFireball() function to throw a big ball of fire our way!

Figure 11.19 shows a screenshot of the game during the first turn of the AI, with a debug of the AttackDesire and CastFireballScore utility values on the left, the highlight of the chosen spell in the enemy's spell list on the right, and the matching animation and VFX that were started by the callback function:

Figure 11.19 – The enemy wizard is now capable of casting devastating fireballs on us!

You see that, thanks to the SOMD and UBAI C# packages we explored in the previous section, we were able to setup a basic utility-based logic for our sorcerer AI in just a few lines of code, and a dozen Scriptable Object assets. We can define the required individual and conceptual utilities fairly intuitively, and our system seems quite adaptable so far.

What's great, too, is that thanks to the modularity of the UBAI architecture, we can build and tune our enemy's AI step-by-step, starting with just the fireball spell, and ignoring the two others. As opposed to finite state machines where any new state requires a certain refactoring of the existing code, here

we'll be able to gradually inject more and more spells, and extend the wizard's action set with new behaviours at will.

Still, growing the utility tree with other branches and other actions is not always straight-forward from a design point of view, in particular because it requires you to properly balance out the new action's score with the others to get the desired behaviour. So let's see how this works by giving our mage a second spell: the Ice Shard.

Adding a second spell

As we've said in the *Let's magic duel!* section, the Ice Shard spell is also an attack skill – however, it is slightly more tactical than the Fireball since it freezes the opponent for one turn, which increases the cost of all its spells by 1. It's thus a good way of reducing your enemy's mana pool if you feel like it's too high at the moment.

This means that, ideally, the AI should value this spell more if the player's mana amount is high, and less otherwise. And, in terms of balancing, it should actually value this spell more than casting a fireball if the player's mana amount is really high, because the priority in this case is to weaken the enemy and prevent them from throwing too much magic.

We're going to model this need for freezing the opponent as a new curve-based UtilityEvaluator instance called

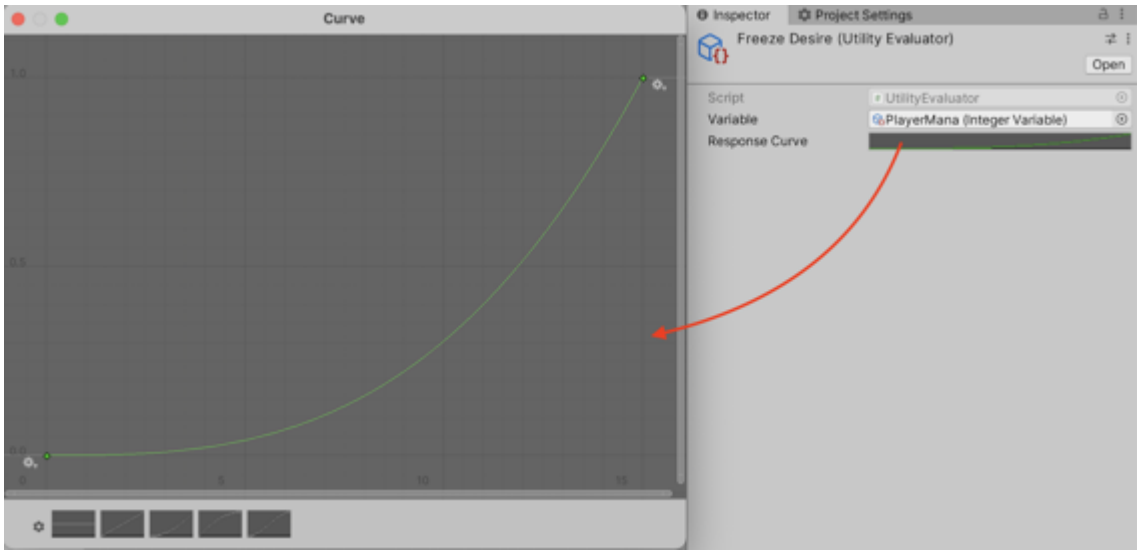


Figure 11.20 – Definition of our FreezeDesire utility using the UtilityEvaluator tool

Then, we'll say that the score of the "cast ice shard" action depends both on this FreezeDesire value, and the AttackDesire utility we computed before; we'll just use a linear combination of the two, so we'll create a new instance of the UtilityCombinator tool from the UBAI package, and then link and weigh both components as follows:

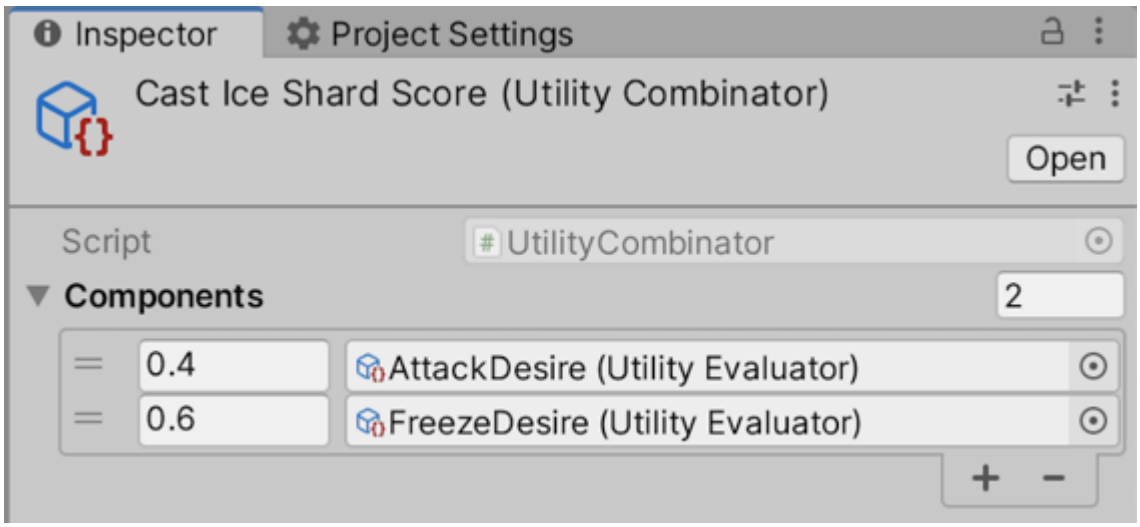


Figure 11.21 – Definition of our CastIceShardScore utility using the UtilityCombinator tool

In other words, this simply means that:

iceshard_utility = 0.4 * attack_desire + 0.6 * freeze_desire

(As usual, don't forget that, as utilities need to be normalised, we need to use weights that maintain our result in the 0-1 range or clamp back the result.)

Alright, we're nearly ready to have our AI cast ice shards – all that remains is to update our MageBrain class and add the callback function for this action,

MageBrain.cs

```
1  using UBAI;
2
3  public class MageBrain : UBAIBrain {
4      private EnemyManager _agent;
5
6      public MageBrain(EnemyManager agent, UBAIAction[] actions)
7          : base(ElectionPolicy.Best, actions) { _agent = agent; }
8
9      private void _CastFireball() { _CastSpell(Spell.LIB["Fireball"]); }
10     private void _CastIceShard() { _CastSpell(Spell.LIB["IceShard"]); }
11     private void _CastSpell(Spell spell) { … }
12 }
```

And finally add this new action to our wizard's action set like we did before with the Fireball, by listing it in our exposed actions array in the EnemyManager component inspector:

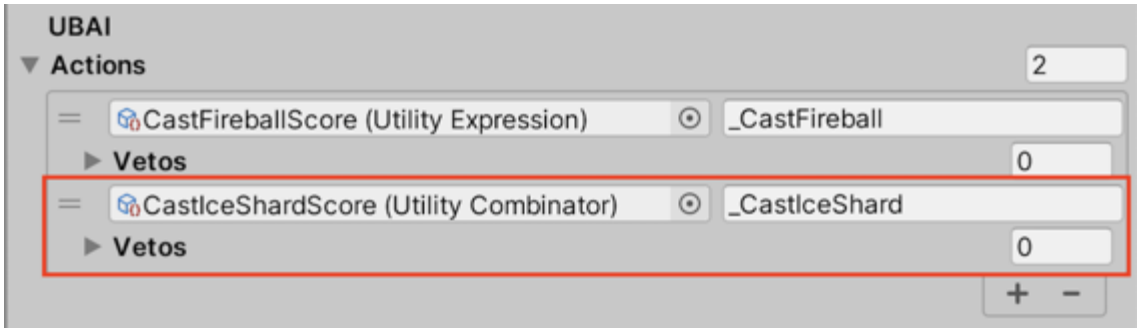


Figure 11.22 – Zoom on the action set in the AI's EnemyManager component inspector with the "cast ice shard" action integrated, using the CastIceShardScore asset as utility computor and the _CastIceShard() **method as callback**

And voilà: our mage AI is now able to evaluate the utility of the Ice Shard spell against the one of the Fireball, and in the early game it will elect this spell as the best option to try and bring our mana pool down as quickly as possible!

Again, *Figure 11.23* shows a screenshot of the game during the first turn of the AI with a debug of the different conceptual utilities and action scores on the left, and the highlight of the picked spell (here, the Ice Shard) on the right:

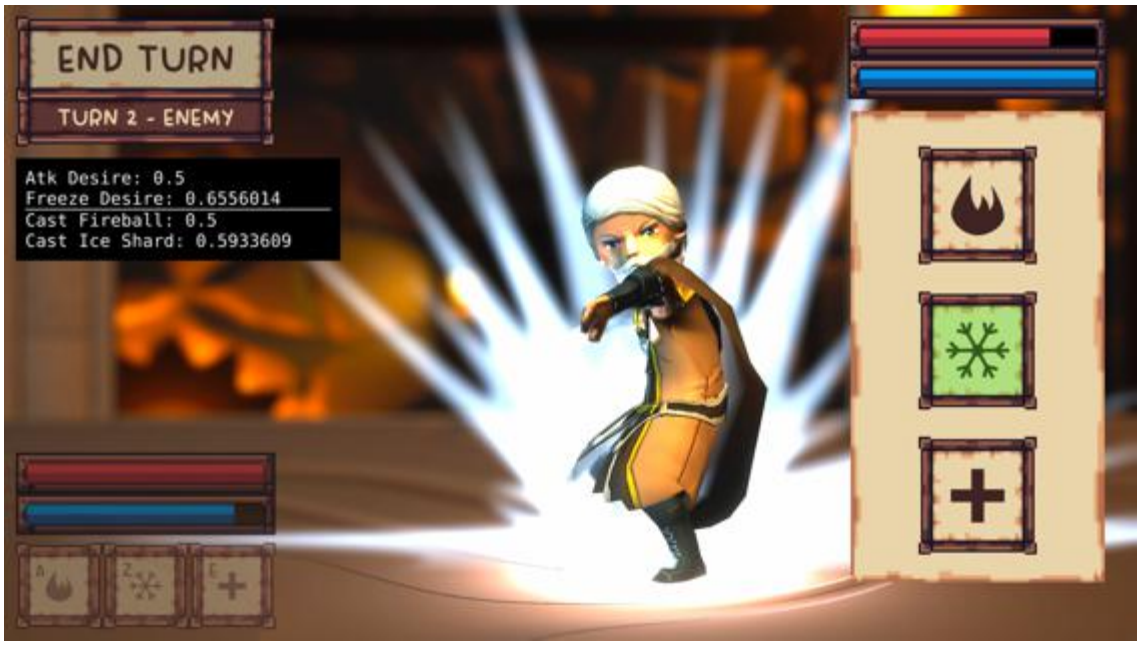


Figure 11.23 – The enemy sorcerer is getting stronger: he's now able to cast ice shards, too...

After a few turns, as it recomputes the different bricks of its utility tree and the player mana's amount reduces, the CastIceShardScore will slowly go down and the CastFireballScore will eventually come back as the highest value, having the AI reprioritise this action.

That's pretty cool: we now have a basic magic duel game against an adversary that can emulate a basic form of intelligence, and choose among multiple possible actions depending on the current game context.

The problem, however, is that our AI still cannot heal – thus unless it has an unfair advantage to begin with, such as a really high amount of healthpoints, it doesn't stand any chance against us (given that *we* have access to the heal spell, and we can basically regain health until the opponent has exhausted its entire mana pool).

So let's finalise our AI's action set by teaching it the third and last skill available, the Heal spell.

Integrating the healing mechanic

The enemy mage is now skilled enough to cast fireballs or ice shards, and assess which spell is the most interesting given the current situation. The final spell it needs to learn is the Heal spell, so that it can regain some health and survive longer in this fight.

So far, designing our utility response curves has been fairly easy – we've just taken one decision factor, or a couple of individual utilities, and applied some conversions and weights to get a viable behaviour. For the healing logic, however, it's going to be slightly more complex.

Mimicking human behaviour

As human players, we usually are pretty good at evaluating when we should temporarily refrain from damaging the opponent, and instead focus on healing to stay alive. We sort of "feel" when the situation is dire and we really have to hit that Heal button.

And as we've mentioned in *Chapter* a very cool thing with UBAI is that it works similar to our train of thoughts: just like a utility-based AI, we naturally tend to spot the relevant variables in our environment, combine them in some ways to estimate usefulness, and finally pick the option that seems the most interesting at this precise moment. This means that,

when confronted with a more complex situation like scoring the utility of our “cast heal” action here, we can try and **reverse-engineer** our own process, and re-implement it in the AI.

But then... how exactly do we determine the utility of healing? What are the relevant variables in the current game context? What do we instinctively compute in our heads to assess whether stepping back and casting a Heal spell is the best course of action?

We know that a crucial decision factor is, of course, the current amount of health we have. So our wizard AI should clearly look up the EnemyHP variable, and use it as an input for a heal desire individual utility.

However, considering just the current health, or even the current health ratio, isn’t enough – when we assess whether we need to heal, we also need to take into account the threat our enemy poses. ‘Cause suppose we’ve lost half of our healthpoints; then, if the enemy can deal such a great amount of damage as to one-shot us, it’s clearly way more important to heal than if the enemy is a trash-mob that can just barely scratch us. In other words, given the same health ratio, we won’t prioritise healing the same depending on the opponent in front of us.

Creating our utility assets

Translating all this as assets in our project is quite easy. We'll simply do the following:

For the heal desire utility, we'll again use a UtilityEvaluator instance, this time with a logistic conversion formula that creates a sigmoid:

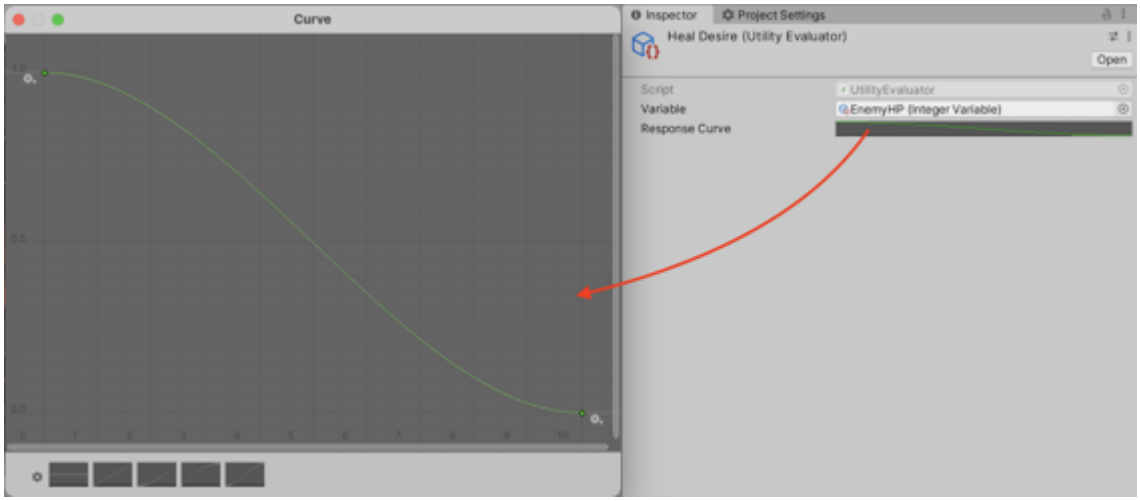


Figure 11.24 – Definition of our HealDesire utility using the UtilityEvaluator tool

This will create a soft threshold around the middle point, which corresponds to the AI having half of its healthpoints (once again, be sure to move the rightmost curve key to the max HP value for the enemy mage, here for example X =

Then, for the threat utility, we'll use a UtilityExpression and compute the ratio of our MaxDamage and EnemyHP values:

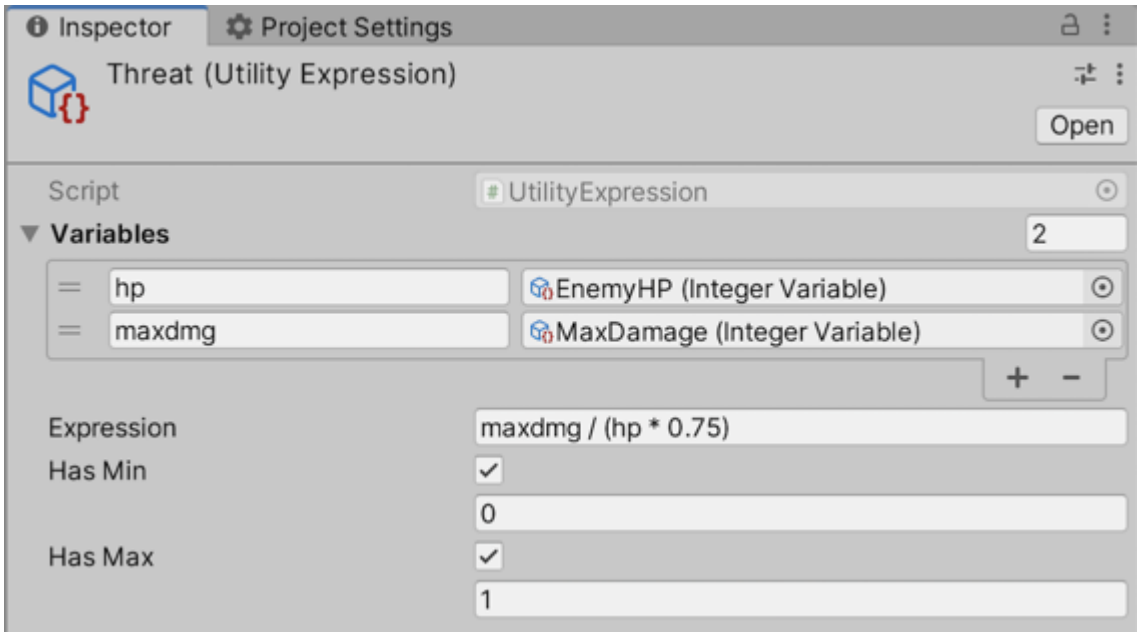


Figure 11.25 – Definition of our Threat utility using the UtilityExpression tool

Note that in this demo, I've arbitrarily lowered the threat value with an extra 0.75 coefficient – this could be changed to modulate the personality of the AI and make the "cast heal" action more or less probable.

Finally, our CastHealScore utility will be the product of both those utilities, again as a new UtilityExpression instance:

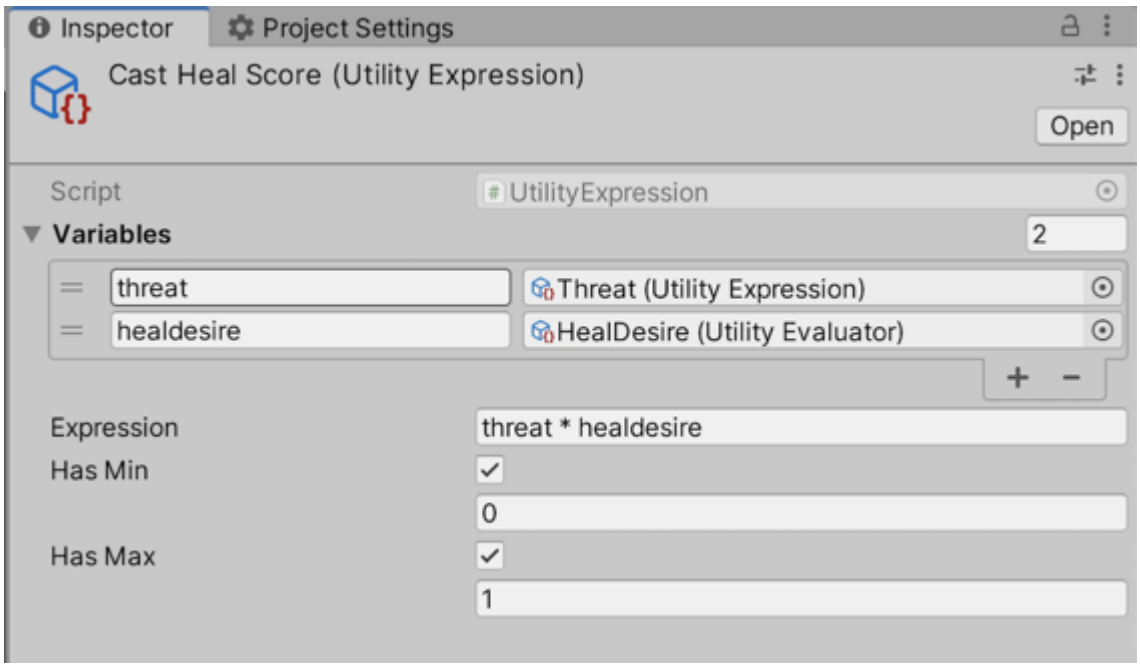


Figure 11.26 – Definition of our CastHealScore utility using the UtilityExpression tool

This is typically where we could impact the behaviour of our entity by adding some weights to boost or hinder the influence of our previous utilities, and thus make the AI more aggressive or cautious.

Adding the action and testing out the result

We now have all the assets we need to insert this new branch in our utility tree and allow our wizard AI to cast the Heal spell.

So let's define the callback function for this action in our MageBrain script:

MageBrain.cs

```
1  using UBAI;
2
3  public class MageBrain : UBAIBrain {
4      private EnemyManager _agent;
5
6      public MageBrain(EnemyManager agent, UBAIAction[] actions)
7          : base(ElectionPolicy.Best, actions) { _agent = agent; }
8
9      private void _CastFireball() { _CastSpell(Spell.LIB["Fireball"]); }
10     private void _CastIceShard() { _CastSpell(Spell.LIB["IceShard"]); }
11     private void _CastHeal()     { _CastSpell(Spell.LIB["Heal"]);     }
12     private void _CastSpell(Spell spell) { … }
13 }
```

And, last but not least, we'll add this action in the exposed array inside the EnemyManager component inspector:

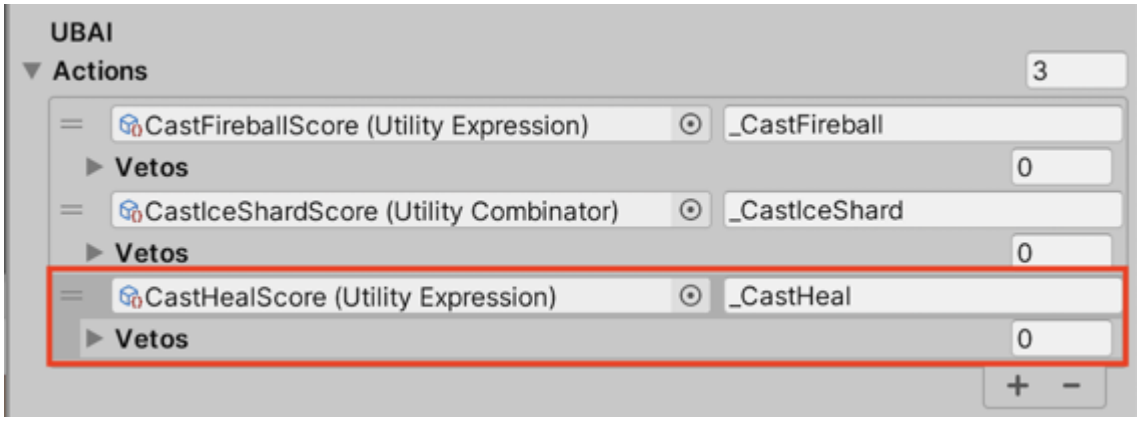


Figure 11.27 – Zoom on the action set in the AI's EnemyManager component inspector with the "cast heal" action integrated, using the CastHealScore asset as utility computor and the _CastHeal() **method as callback**

If we restart a new game, we see that the enemy mage now also checks for the current threat level and his heal desire, and if his life is too low and there is a risk that we could kill him, he picks the "cast heal" action to restore some HP (see the debugs on the left and the highlighted spell on the right):

Figure 11.28 – When low on health and really threatened by our potential damage, the enemy wizard now casts the Heal spell to restore some health!

At this point, our opponent has access to exactly the same arsenal as we do, and it should technically be able to "think" in a similar way thanks to its utility-based AI brain. So we're quite close to having a real magic duel!

There are just two things we should add to wrap up our game's AI: first, some vetoes so that the sorcerer doesn't cast spells when it doesn't have enough mana – because, for now, he can cheat and throw as much magic as he wants; and second, some global evolution of his behaviour throughout the fight, to emulate the usual mood swings a real player has...

Checking for forbidden actions and making our AI evolve

Discarding actions using vetoes

When we described the UBAI package in the *Setting up our UBAI architecture* section, we said that our UBAIAction class contains three fields: the reference to the asset that computes its utility score, the name of its callback function (as a string) and a list of vetoes, to completely disregard this action for the time being.

Those vetoes rely on an Evaluatable asset that returns 0 if the condition to check is false, or 1 otherwise. And, of course, a UBAIAction instance with a list of vetoes defined can only be considered for election if none of its vetoes return 0.

In our case, there is just one blocker that can stop a wizard from casting a spell: the lack of mana. Basically, if the sorcerer currently has less than the spell's mana cost (plus 1 if he's frozen), then he shouldn't be able to cast it.

So to check for forbidden actions and eliminate the spells that are too costly given its current amount of mana, the AI is going to need three additional UBAI Scriptable Objects: the CanCastIceShard and CanCastHeal assets.

Those three objects will be instances of our very last tool in the UBAI package – the In short, we just have to take in the EnemyMana and EnemyIsFrozen variables and use them to compare the mana amount of the AI to the mana cost of each spell. We can access this value directly from the Spell Scriptable Objects, thanks to our dot accessor trick (and of course, we should add 1 to the cost if the AI is currently frozen):

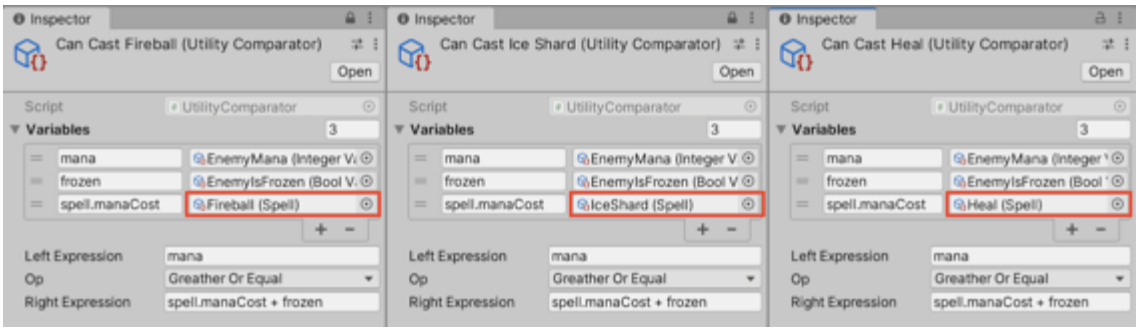


Figure 11.29 – Validators for our three spells that compare the current amount of mana of the AI to the cost of the spell (plus 1 if the AI is currently frozen)

You see in *Figure 11.29* that the three comparators are almost identical: we just need to pick the right spell for the spell.manaCost local variable, and then setup the left and right expressions for the comparison, and the comparison operation.

With those three assets added to the project, we simply need to go back to our EnemyManager component inspector and, in the Vetoes sub-arrays inside each item of the AI's action set, select the right comparator:

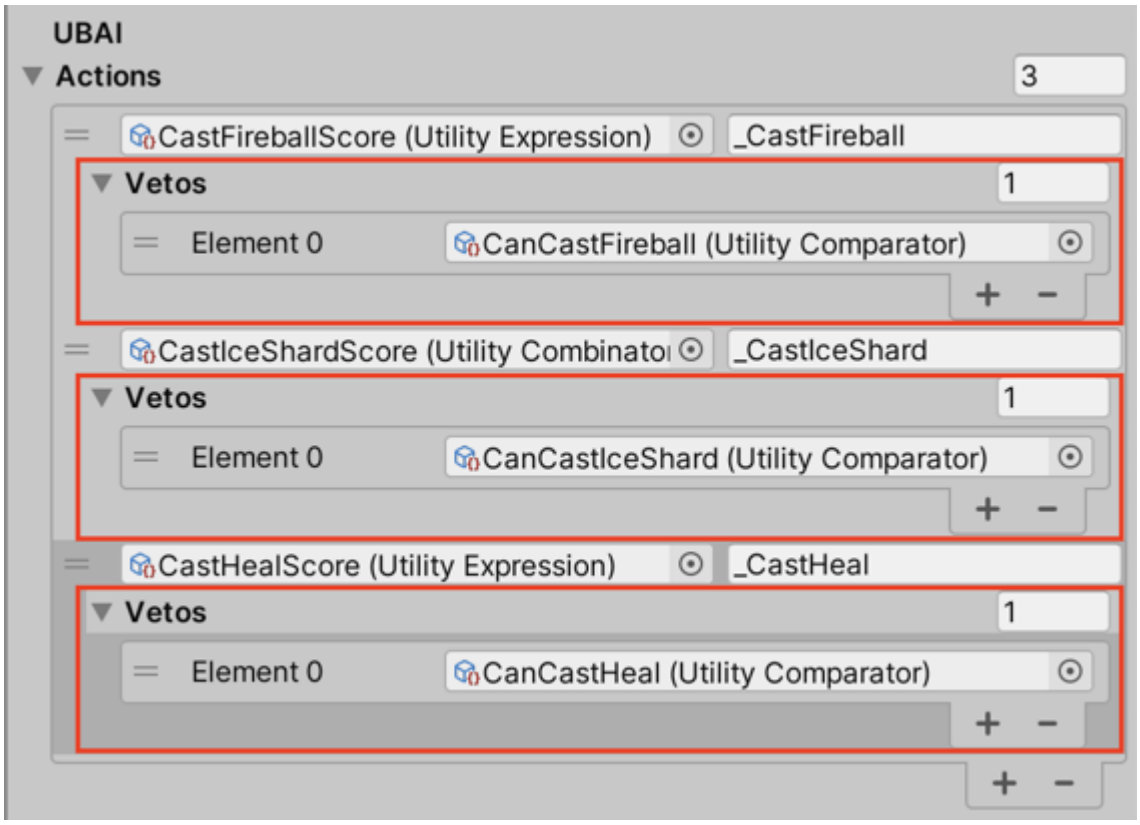


Figure 11.30 – Zoom on the action set in the AI's EnemyManager component inspector with the action veto conditions integrated

Just by setting up those three checks, we've ensured that the AI now properly compares its current amount of mana against the spell's cost before adding it to the list of possible actions,

and that if it can't do anything, it fallbacks to its default idle action.

While we're at it, we could also create another veto condition for the "cast heal" action, so that the wizard only considers the Heal spell if the Threat utility score is above 0.5. The advantage of using this threat level as a veto is that it will prevent our AI from prematurely healing if we are not *really* threatening its survival, just because it doesn't have enough mana to do anything else – instead, it will skip the turn to restore some mana, and hopefully have more options next time.

To implement this, we can make a HighTreat check asset that is a simple UtilityExpression and uses a hardcoded threshold of 0.5:

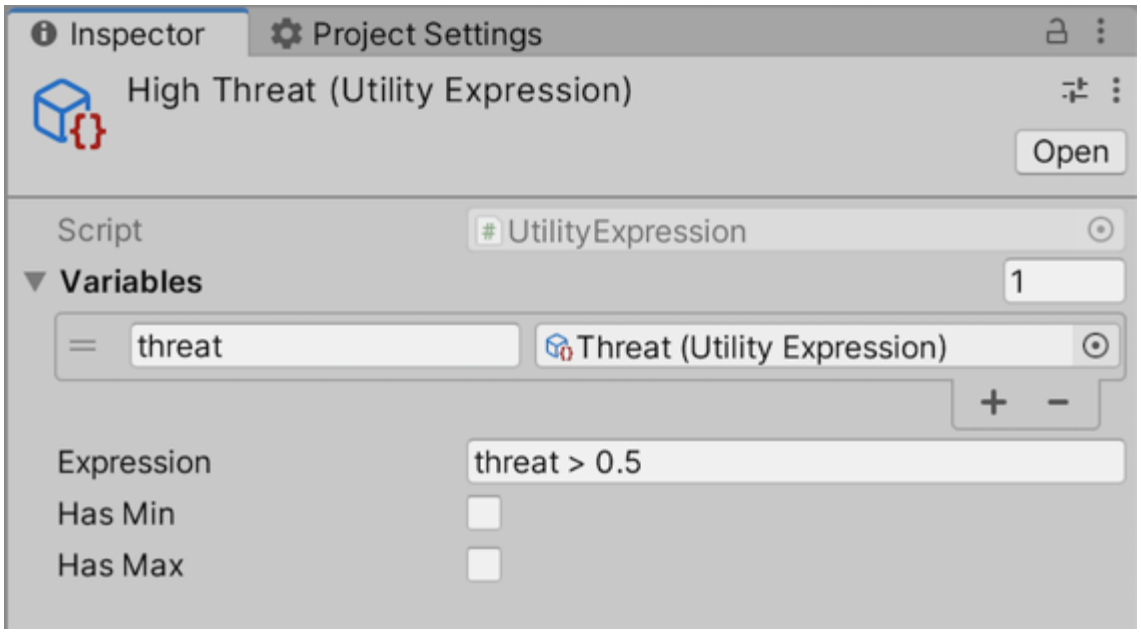


Figure 11.31 – Definition of our HighThreat comparator using the UtilityExpression tool

And then, we'll simply add this asset to the list of vetoes for the "cast heal" action in our EnemyManager component inspector:

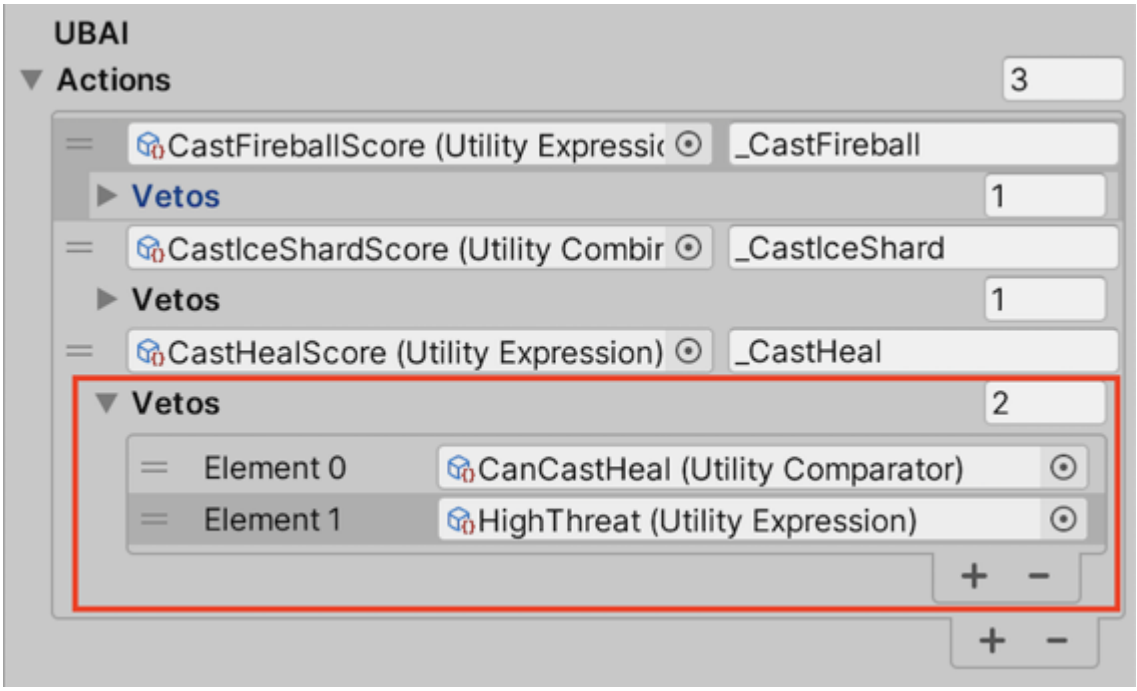


Figure 11.32 – Zoom on the action set in the AI's EnemyManager component inspector with a second veto condition integrated for the "cast heal" action

Thanks to this extra blocker, the mage AI is now able to wait for the next turn if the situation isn't too critical, to

accumulate more resources and anticipate the upcoming turn. We've thus given it a more cautious temper, and a limited but undeniable capacity to plan for the future.

But guess what? It's actually possible to go even further and emulate a real personality for this character by having it evolve and change its preferences as the game progresses – just like a human player who would take more risks as time goes by because she wants to end this fight...

Giving our AI "mood swings"

As a final bonus, let's see how to make our AI even more lifelike by allowing it to evolve through the game. In this final section, we're going to give the wizard the notion of anger so that, as time passes and the fight starts to linger, our enemy starts to get more aggressive and slowly disregards healing in favour of just brute force.

This is obviously not a perfect behaviour, and it is also, admittedly, a crude technique for avoiding endless duels with the AI always healing just at the right time and being a pain for us, but it will be an interesting opportunity to discuss long-term goals in the context of utility-based AI.

Indeed, since UBAI doesn't make any plans, and it doesn't assign the entity a specific state to represent its current

mindset, it sounds like having long-term objectives is impossible when using this type of AI architecture. In particular because the notion of “time” and “past durations” can be a bit fuzzy to the entity, unless we’re carefully maintaining variables at various scales to keep track of all the necessary timespans.

In our case, however, we can take advantage of the fact that our game is turn-based and use this ever-increasing turn count as a proxy for the time elapsed since the game has started. And with this global variable at our disposal, it is then extremely straight-forward to define a curve-based UtilityEvaluator instance called Anger that increases steadily as the number of turns grows, and emulates the AI just losing patience:

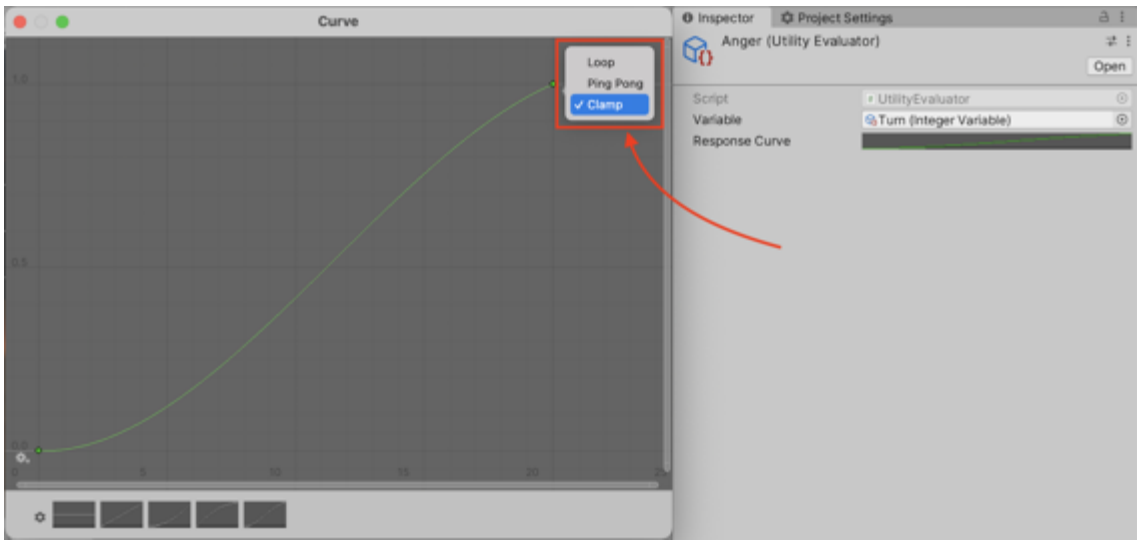


Figure 11.33 – Definition of our Anger utility using the UtilityEvaluator tool, with a focus on the three possible wrap modes

PICKING THE RIGHT WRAP MODE FOR OUR CURVE

When defining an AnimationCurve in Unity, you can setup various keyframes to manually author a custom curve, but there is always a limit to the amount of points you'll be able to define by hand. That's why, in addition to these keyframes, you also have to choose the right WrapMode for your curve, among: Loop, PingPong and Clamp.

This WrapMode option will tell Unity how to evaluate the curve outside of the defined range:

- If it is set to Clamp, then the value will stay at the same Y coordinate when leaving the defined range. So, on the left of the leftmost key, it will keep the value it had at this initial key; and on the right of the rightmost key, it will keep the value it had at this final key.

- If it set to Loop, then the value will loop back to the Y coordinate it had on the other side of the defined range with the

> exact same rate of change as before.

> - If it set to PingPong, then the value will revert back to the Y coordinate it had on the other side of the defined range with the reversed rate of change.

> The curve editor allows you to easily preview the result of the WrapMode you picked as a dimmed out green line on both sides of the defined range. Here, we should stick with the default Clamp option since it makes sense to just prolong the current utility outside the defined range.

We can then inject this new individual utility inside our “cast fireball” action scoring to encourage the wizard to throw more firy spells at us as time goes by:

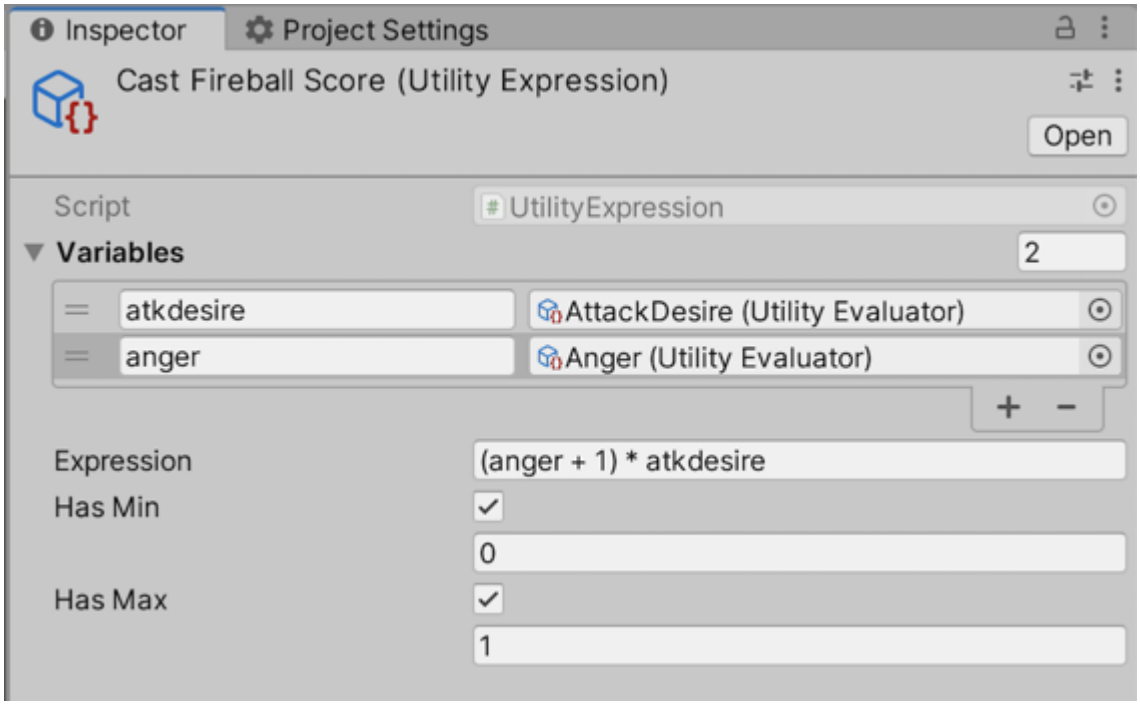


Figure 11.34 – V2 of our CastFireballScore utility with the additional Anger value taken into account

Again, there are probably more advanced ways of modelling this temper evolution, and giving an interesting persona to our enemy – plus, if we really wanted to express this anger, we should probably have slight changes in the model's animations, too, and perhaps a few emotes or visual cues to truly highlight this growing anger.

But, anyway – for now, this is good enough, and it will ensure that the AI never enters a full loop and always re-assesses the situation a tiny-bit differently.

Our wizard AI is now complete, and if you try and battle it you'll see that it holds its end of the bargain: we get an entertaining magic duel that lasts for quite a while, with various spells cast on both sides. The enemy is able to accurately pick the best possible option at each turn and, once it has chosen an action, the corresponding callback function takes care of actually executing it to update the current game state. The AI can even do some clever strategic choices like a temporary turn skip to accumulate more mana, thanks to its evaluation of some conceptual utilities such as the threat level.

Moreover, we managed to make all of this with a really short script and very basic Scriptable Object assets, which make it possible to assemble and refine our AI in a completely no-code fashion.

By the way, just for reference, here is the diagram of the final utility tree of our mage AI:

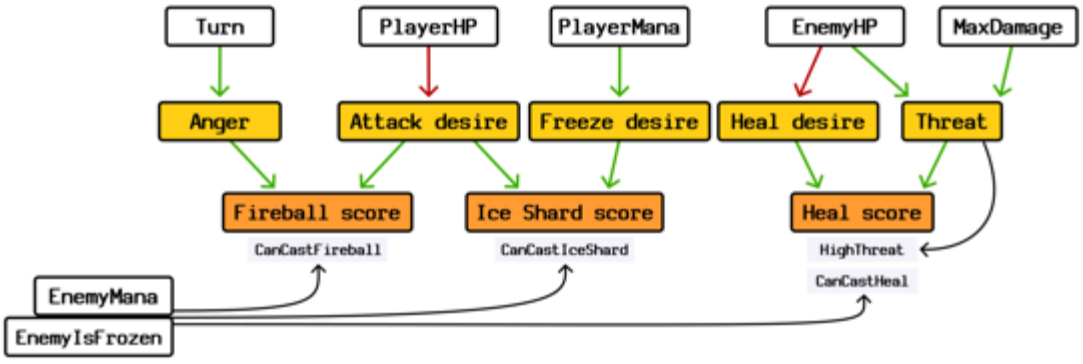


Figure 11.35 – Final version of our AI's utility tree (the grey blocks with incoming arrows at the bottom show the veto conditions and their dependencies for the different actions)

This diagram clearly shows the modularity of the utility-based AI approach: we have several blocks that are used in multiple computations, and just by re-arranging or re-weighing some links we could easily create a completely different behaviour for another character, without necessarily having to create other data bricks.

Studying this example of a turn-based magic duel game has taught us how to design and build a basic UBAI architecture that is expressive, robust, powerful and yet easy to maintain and scale if we ever want to integrate more actions. Of course, as always with utility-based AI, some utility response curves have to be tweaked until the overall behaviour feels right, and crafting the perfect utility tree is a delicate art of balancing. Yet, all in all, we've managed to setup our very own AI enemy player based solely on a clever description of our current game state, and a few well-picked math formulas!

Summary

In this chapter, we've worked through an application of the utility-based AI principles we'd discussed previously, and we've learnt how to make an AI player capable of competing with us.

First, we had a quick overview of the game we were developing our AI for. We saw it was a turn-based magic duel game, with a small set of spells available on each side, and a handful of relevant data points (e.g. the wizards' health and mana amounts, whether they were frozen by an Ice Shard cast...). We also identified the possible actions for our AI, including the default "skip turn" action that can be an interesting strategic choice.

We then looked at the SOMD and UBAI C# packages that I prepared for this demo, and we discussed the grand design ideas behind each of them. We explored how they use Scriptable Objects to ensure a high level of modularity, and we talked about the core UBAI objects at our disposal for the implementation of our mage AI.

After that, we built our opponent's AI behaviour step-by-step, by defining more and more utility assets in our project, and thus teaching it more and more spells. We browsed through all of our UBAI library to create adapted and tweakable curve-based evaluators, weighted linear combinations, math expressions and even comparators, and we gradually setup the utility tree of our AI as an assembly of re-usable and modular data bricks.

We are now nearing the end of this book, and this chapter concludes our study of the most common techniques for developing artificial intelligence and entity behaviours in games.

In the final chapter, we will take a step back, and we will discuss a few ideas for going further and expanding our horizons on game AI.

PART 5

GOING FURTHER

12 - Expanding your horizons

Our journey in the world of game AI programming is slowly coming to an end – we've discovered many techniques for giving our minions some brains and "mimicking intelligence". From finite state machines to utility-based AI, we've explored a lot of tools and discussed a lot of concepts; but this was just a peek at an immense field.

AI programming is a daunting task, a crazy dream of emulating life on a computer... and yet it is, in my opinion, one of the most rewarding things. Seeing one of our creatures suddenly move about and react to the world around it is a really cool feeling as a developer :)

This book is in no way a comprehensive study of all the available AI techniques, and I want to encourage you to explore further. Still, before I leave you to this exploration, here a few final notes and ideas to broaden your perspectives and perhaps guide your creativity...

Exploring some alternate designs

Throughout this book, we've covered the most common AI techniques, the tools that all game AI developers know and love, and that make up the brains of virtually all the enemies and NPCs in today's games.

And yet, as you know, AI designers like to go beyond and constantly invent new architectures – or at least variations and improvements of the current ones. They have this insatiable thirst for ever more powerful AIs, and more realistic behaviours, and more adaptable reactions.

This has led to a lot of experimentations in the game dev community, and some interesting alternative systems that we're going to briefly discuss here.

Blending multiple techniques at once

First of all, a crucial idea is that, while the state machines, behaviour trees, planners and utility-based AI techniques that we've learnt about in the previous chapters are usually used only one at a time for modelling the behaviour of an entity, it is possible to mix them at different levels for a more complex AI structure.

For example, consider the little RTS demo we worked on during *Chapter 8: Implementing a RTS collector* This use case

was about using behaviour trees to control individual units by giving them a basic collector behaviour. But what if it was a real RTS game, and we wanted to be able to play against the computer in solo mode?

Then, we'd need to code up an AI player, just like we did in *Chapter 11: Designing a utility-based wizard* And although it is not the only option, utility-based AI would probably be a good way of handling this high-level AI "player brain".

So, for this use case, we could end up with a two-levels AI structure:

The AI player relies on UBAI to take global decisions and react quickly to the current state of the game.

These decisions are then sent over to each of the units owned by this AI player and update their local behaviour trees to have them react to the order appropriately.

The nice thing with a multi-level/multi-scheme AI like this one is that we can leverage the perks of each tool at each level. Rather than forcing ourselves to use the same type of AI architecture everywhere, we can pick the best tool for each job: we use the flexibility and ease-of-maintenance of utility-based AI

for our player's AI because it might evolve a lot, and it needs very detailed responses to the game data; but we take advantage of the robustness and ease-of-debugging of behaviour trees for our individual entities because we have to accurately understand and predict their reactions.

Similarly, we could blend planners with finite state machines to have a squad leader take a general decision, and then the soldiers in his team execute this order; or we could power some of our units with behaviour trees when they need advanced behaviour, and others with basic C# scripts when they are a bit simpler.

All those AI techniques aren't exclusive to one another: they each have their advantages and drawbacks, and it's always interesting to try and see if another tool couldn't complement your current system to improve its stability or its realism.

Having an AI evolve by unlocking behaviours

Another possibility for adding some flexibility and adaptability to your AIs is to have their action set evolve throughout the game.

In

Alien: Isolation

(Creative Assembly, 2014), the Xenomorph who lurks around doesn't start at full capacity. In order to maintain the stress and drama for the players, the monster actually "learns and improves" as time goes by, simply by **unlocking new sub-branches** in its behaviour tree. You're thus threatened by an enemy in constant improvement, which further increases the tension.

Middle-Earth: Shadow of Mordor

(Monolith Productions, 2014) was also critically acclaimed for its amazing **Nemesis** which populates the story with randomly generated procedural mini-bosses that gradually evolve each time you encounter them. This system creates dynamic secondary stories of loyalty, vengeance and betrayal between you and those unique orcs – and these stories translate to actual gameplay experiences, since in addition to their initial random traits and powers, the enemies that manage to defeat you get promoted and become stronger.

Never forget that memory is an essential part of human intelligence, and that with memory comes learning. So if your game is to show a memorable character, be sure to give it the means to grow and improve, parallel to the player, and consider taking advantage of those simultaneous player/AI progressions.

WAIT, WHAT ABOUT PROCEDURAL GENERATION, THEN?

Middle-Earth: Shadow of Mordor is an excellent example of how mixing randomness with hand-scripted elements can give birth to incredible on-screen characters. Yet some games go further and try to bring the manual authoring down to almost zero thanks to **procedural generation**.

This is typically one of the big principles behind **roguelike** games (Spelunky, The Binding of Isaac…), or games that are based on procedurally generated content like Dwarf Fortress or Rimworld. All those examples are powered by a sort of controlled randomness that takes care of automatically filling the game's world with well-designed level elements, objects and even entities. In other words, the point is not to program and populate scenes but rather to code up world rules that are capable of auto-generating these scenes for us.

This is extremely valuable in terms of **replay value** because it means that you'll be able to create an infinite amount of unique adventures for the players, and it also allows for **bigger game worlds** (as demoed by Minecraft or No Man's Sky). However, it can also severely reduce the level of **control** you have on the end result and requires a good abstraction of your world's rules to work properly.

Pouring “modern” AI in the mix?

In the very first chapter of this book, *Chapter 1: AI in* we spent some time talking about the differences between modern machine learning-based AI and game AI. In particular, we discussed how those neural networks, despite being quite efficient at solving specific tasks, are often too specialised and too uncontrollable to be usable in video games.

But is it really that bad an idea?

Restating the problem

Machine learning is a powerful tool that can abstract away a lot of the complexity and help you design powerful and adaptive tools for various problems.

But because of how neural nets work, with all their internal weights that approximate a mathematical function and just spit back a result for a given input without any other details, they tend to suffer from the **black-box** no matter how accurate your results are, you can’t for the life of you explain where they came from and you therefore lack predictability.

This is mostly why, although machine learning tools have developed quite rapidly in other domains (e.g. image classification, text generation...), they have been frowned upon and ignored by most game AI developers. We still favour more easy-to-use and easy-to-debug tools like the ones we've studied in the previous chapters.

The primary issue with neural networks is that you can't just jump in the code and tweak the entity's behaviour precisely in one place to have the AI react like so in a given situation. It is not an "if you perceive that, then do this" paradigm like a finite state machine or a behaviour tree. Instead, machine learning-based AI is about training a set of weights to approximate the behavioural mathematics of your unit.

What this means is that, basically, you want to train an ensemble of nodes that can be fed a set of input data describing the current game state, and get back the action to perform in return. Which obviously requires you to properly design these inputs, nodes and outputs to allow the network to learn by itself and compute the right kind of info for you.

This might seem like a difficult thing to do – and it is! –, and more importantly a risky endeavour since you're essentially entrusting your algorithm with determining the entire decision logic on its own.

However, when done right, it can also be an interesting approach to creating flexible and well-balanced AIs for your games... like what Mike Robbins did for the platoons of

Supreme Commander 2

!

The example of Supreme Commander 2

The

Supreme Commander

franchise by Gas Powered Games is a series of RTS games set in a futuristic sci-fi world, where you produce troops to defend your main Commander unit and, if possible, destroy the enemy one.

Figure 12.1 – Screenshot of the

Supreme Commander 2

game (image from:

In the second title,

Supreme Commander 2

, the gameplay engineer Mike Robbins decided to rely on neural networks to power the intelligence of the AI's platoons. The point was to craft some mid-level squad brain logics to help these packs of units pick the best possible action given the current situation.

As explained in his 2012 GDC conference Mike Robbins wanted to fight the bad reputation of neural networks for game AI programming and show how, when used the right way, this tool is just as powerful and robust as the other more common ones.

For this game, the developers trained four networks (one for the land units, one for the naval units, one for the fighter jets and one for the bomber planes) because each unit type as a different action set.

Those networks are back-propagating neural networks (see *Figure* that:

Take in inputs about the local game situation. More precisely, the platoon considers a total of 17 data points about itself and the enemies it is currently facing such as the number of units, the total health, the overall DPS and more.

Feed those inputs to a layer of hidden nodes. Those nodes weigh the inputs and "learn" which ones are relevant, what the impact of each variable is, etc.

Feed the values of those intermediary hidden nodes to the output nodes, which choose the best action for the platoon at this precise moment (e.g. attack the closest unit, attack the weakest unit, attack the unit with the highest value...).

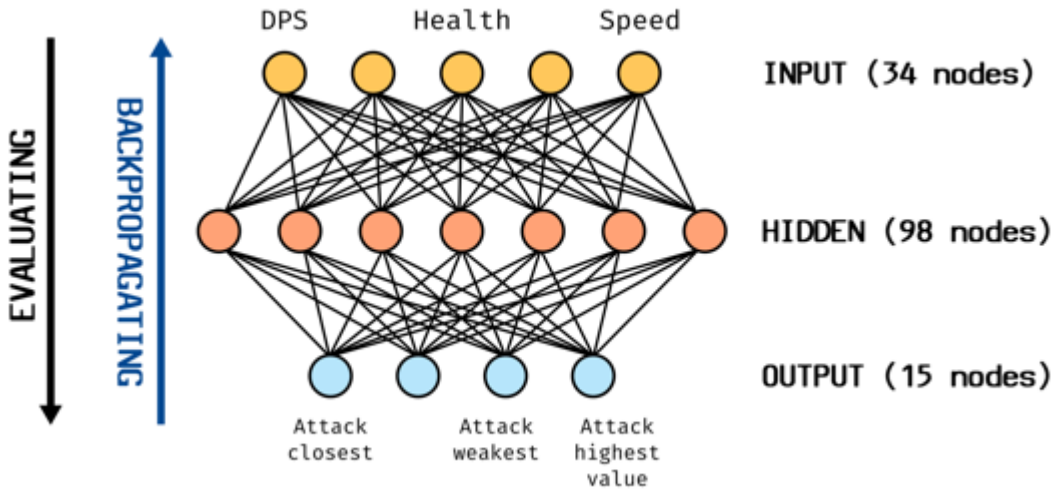


Figure 12.2 – Visualisation of the back-propagating neural network structure used by the AI developers of

Supreme Commander 2

for the platoon behaviour (adapted from Mike Robbins's conference slides)

The neural networks therefore determine a threat value for the platoon and use the set of inputs to ultimately pick the right response.

Training the network is done by making the platoon choose actions at random and then evaluating how well these actions performed. This **fitness evaluation** is done by estimating if "I hurt them more than they hurt me", meaning that the overall stats of the platoon after executing this action are better (or at

least less worse) than the enemies'. The platoons thus learn to assess the overall situation and react immediately to any changes in their 17 input data points, to continuously pick the best possible action, by slightly tweaking the weights in the network to get the best fitness value at all times. (Of course, this implies that if you change your neural network structure or the intended behaviour of your units, you'll need to start the training over to re-tweak your network's weights.)

These neural networks allow the platoons to engage or retreat, back off from a position while continuing firing at close opponents or fleeing for their lives if the situation gets too dangerous. And they allowed the

developers to create an engaging and challenging AI to battle against for the players.

What's interesting with this approach is that, once you've carefully designed your network shape and chosen your input data points, you don't need to explicitly try and craft a formula for comparing DPS and health, or weighing the inputs, or even identifying the most relevant inputs: all of this is done by the network during its training. And because this training is run automatically by feeding in inputs, picking a random action, evaluating the fitness and re-adjusting the weights in the network (that's the "back-propagating" part), you don't even need to sit by the computer and watch it work – you can go

work on other projects while your AI is effectively learning to play your game.

Moreover, because the AI is not tuned to handle a particular situation but takes into account the whole context for tweaking its weights, this neural network approach handles balance changes very well. So if you suddenly change the stats of a specific unit type because you feel it will improve the gameplay experience, you just have to retrain the nets and your platoons will automatically react according to this new balance.

There might be some specific cases that require you to adapt your fitness evaluation formula and have your AIs learn a few rules by heart, but all in all, when used right, neural networks are a valuable tool for auto-tuning your entity's action weights and automating the decision logic design.

For Mike Robbins, "neural networks are like any other AI tool", and their bad reputation among game developers comes from people applying them improperly. Julian Togelius, the game researcher and professor we mentioned back in *Chapter* shares this vision and regrets that game AI developers restrict themselves to using old AI techniques – noting that "at conferences, [he] would try to convince game designers that their company stood to gain from using new AI methods, with the response being that it was not necessary".

Maybe the example of

Supreme Commander 2

is proof that, just because we can't interpret all the inner weights of a neural network, we shouldn't simply discard this tool and rather reconsider the idea of having deep learning auto-generate decision logic for us. Perhaps we should take some time to think about how to better integrate it in our AI toolbox...

A quick note on reinforcement learning

> Among all the different combos people have tried to mix AI and games, there is a subdomain of machine learning worth mentioning, called **reinforcement learning**.

> The idea of this neural network-based method is to train a net without any labelled input/output pairs, and instead give the AI regular **rewards** (resp. **punishments**) to show positive (resp. negative) appreciation for a particular action, or set of actions, in a given environment. Similar to conditioning, the point is to teach the agent what is "good" and "bad" not by explicitly laying out, but by having it figure out what series of decisions led to the best cumulative reward and should therefore be reproduced next time.

> This type of machine learning has become pretty famous within the game community, because it can enable us to train AIs to play fairly complex games such as a side-view Mario platformer, a racing game like Trackmania or even an advanced RTS like StarCraft II. (And, as a somewhat parallel example, it is also funny to note that Peter Molyneux's Black & White game offered a system *à-la* reinforcement learning for training your avatar creature with positive and negative rewards, in order to encourage or discourage behaviours…)

Doing artificial genetics

To wrap up this section, I want to mention another out-of-the-box tool that, from what I've seen, isn't that well-known among game developers, but is however an interesting way of improving the performance of your AI brains: the **genetic**

To put it simply, genetic algorithms are inspired by the **Darwinian evolution** process. They are **evolutionary algorithms** that rely on iterative generation breeding to get from a (mostly random) initial population of individuals to one that counts many individuals "well-fit" for the task at hand. To do so, they re-implement three core concepts in genetics: natural selection, breeding through DNA crossing and random mutations (see *Figure*

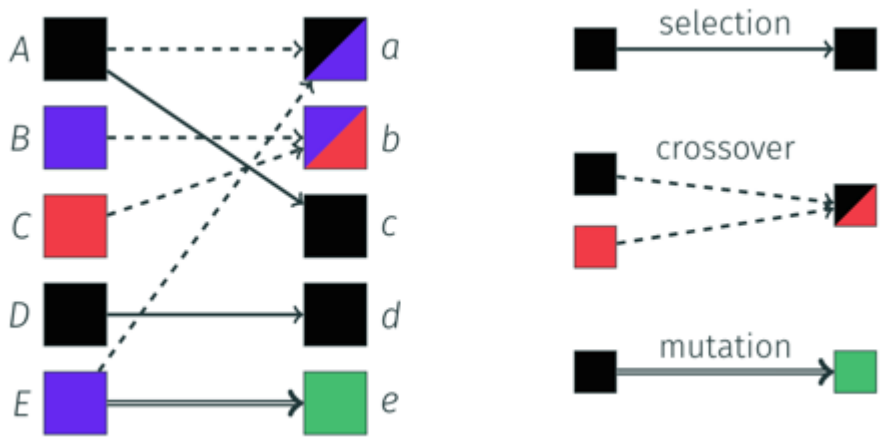


Figure 12.3 – Simplified diagram of three core concepts in genetics: natural selection, crossover and random mutations

In a sense, this is similar to how we tried to mimic natural intelligence and animal cognition by analysing and abstracting the brain's systems into computer-compatible stuff – genetic algorithms also re-enact a real-life phenomenon to give an **"illusion of**

In our case, genetic algorithms can be a neat way of optimising the decision logic of our AI systems, in particular if is based on weights like a planner or utility-based AI. The idea could be to:

1. Consider multiple copies of our architecture with various sets of random weights (that slightly deviate from a mean value, for example) to create an **initial** Each copy is called an

and will evolve throughout the generations until, gradually, we are left only with the "best individuals".

2. Define two functions for the crossover and mutation processes.

The crossover function should take in two "parents" (meaning, for us, two instances of our architecture that each have their own weights) and mix them in some way to create a new individual for the next generation.

The mutation should take in an individual and apply some random small modifications on its weights to transform it into a slightly different version.

3. Define an evaluation function to assess the **fitness** of an individual. This method will compute a score that describes how performant this individual is.

4. Run the evolution process on the individuals of the initial population to create a second generation, and use the evaluation function to keep only a subset of individuals – the ones that get the highest fitness score (in order to simulate natural selection). Then, repeat the process as many times as needed to transform and (hopefully) improve the population.

By the end, the population should contain a few cherry-picked individuals that have performed well and technically contain the most interesting weight values.

Of course, the next big question would be: how do we determine performance in our example?

Indeed, implementing a good fitness function is usually the tricky part with genetic algorithms. If you are too lenient, you run the risk of having too many individuals survive each generation, and you won't really filter anything out; but if you are too strict, your population will shrink too quickly and you'll end up with no results at all.

Evaluating the fitness of an AI architecture is obviously very complex – we would need to establish some automated evaluators for our entities' reactions, and those are clearly too context-dependent to try and sketch here. So I won't pretend to dive into this much further, and I'll leave it to you to examine whether this technique could apply to your own game projects.

With that being said, who knows: perhaps this idea of re-mixing computers and biology in yet another way expanded your horizons on the topic, and fuelled your creativity to contribute to the ever-growing toolbox of the community? :)

CURIOUS ABOUT WEIGHT-BASED AI OPTIMISATION?

> If you want to read more about how to use genetic algorithms and optimisation for weight-based game AI, you can have a look at this 2015 thesis by Dylan Anthony Kordsmeier: https://scholarworks.uark.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1031&context=csceuht. In this paper, the student explores how to use genetic algorithms to have an AI "learn" to play the famous Connect 4 board game.

Extra resources and creators

Actually, I believe it stands to reason that we should end this book talking about the game community. Because, as always when diving into a new topic and gathering information, it's important to look everywhere, and find as many sources as you can. The crux of learning is to discover, understand, test and re-create over and over again, until finally you feel like you've mastered a tool enough to make it your own.

So, to wrap up this journey in the world of game AI, I want to leave you with a list of references that, I hope, can help widen your knowledge of this field, and nurture your creativity.

---

book series Not much surprise there! We've already referenced this incredible ensemble of books several times in previous chapters, and it is clearly one of *the* resources to keep close by when you work on the AI of your games.

In addition to being completely free and easily available online, this series is co-authored by a plethora of expert game AI developers who have been in the industry for a long time and worked on legendary games (

Bioshock: Infinite

,

Zoo Tycoon 2

,

Watch Dogs 2

,

The Sims

or

Thief

, to name just a few).

With these three books (plus the additional Online Edition 2021 chapters), this pool of authors thus shares an in-depth perspective on the state of the art of game AI and presents us with a great deal of theoretical concepts and practical implementations.

Amit's Game Programming Information

web page If you're looking for a nice link hub, you should definitely go and bookmark this page. In here, Amit Patel maintains a list of dozens of game programming-related papers, books and blog posts – and there's a whole section dedicated to game AI...

Tommy Thompson's

AI and Games

YouTube channel In case you are more into videos, I really recommend you check out Tommy Thompson's amazing YouTube channel.

In his various video series, this AI developer and consultant explores the tools behind some of the most famous game AI systems and gives some insider tips on "how AI makes for better games and games make for better AI" (quote from his YouTube bio).

The content is clear and straight to the point, and Tommy Thompson manages to find the right balance between grand definitions, real-life examples and game code analysis, to offer easy-to-follow little nuggets of knowledge.

In truth, I could go on and cite another bunch of articles, and theses, and conferences, and talks, because AI is one of those big topics that almost every game developer takes an interest

in at one point or another, which means a lot of resources have been produced over the years.

But to me, those are good places to start, and then bounce back from as you get more familiar with the field. So, if you wish to continue your exploration of the amazing domain of game AI, don't hesitate to take a look and go through those links...

Playing around with the demo project!

Last but not least, remember that you can also access and swift through the Github repository of this book that contains the entire Unity/C# demo project.

https://github.com/MinaPecheux/Ebook-Unity-AIProgramming

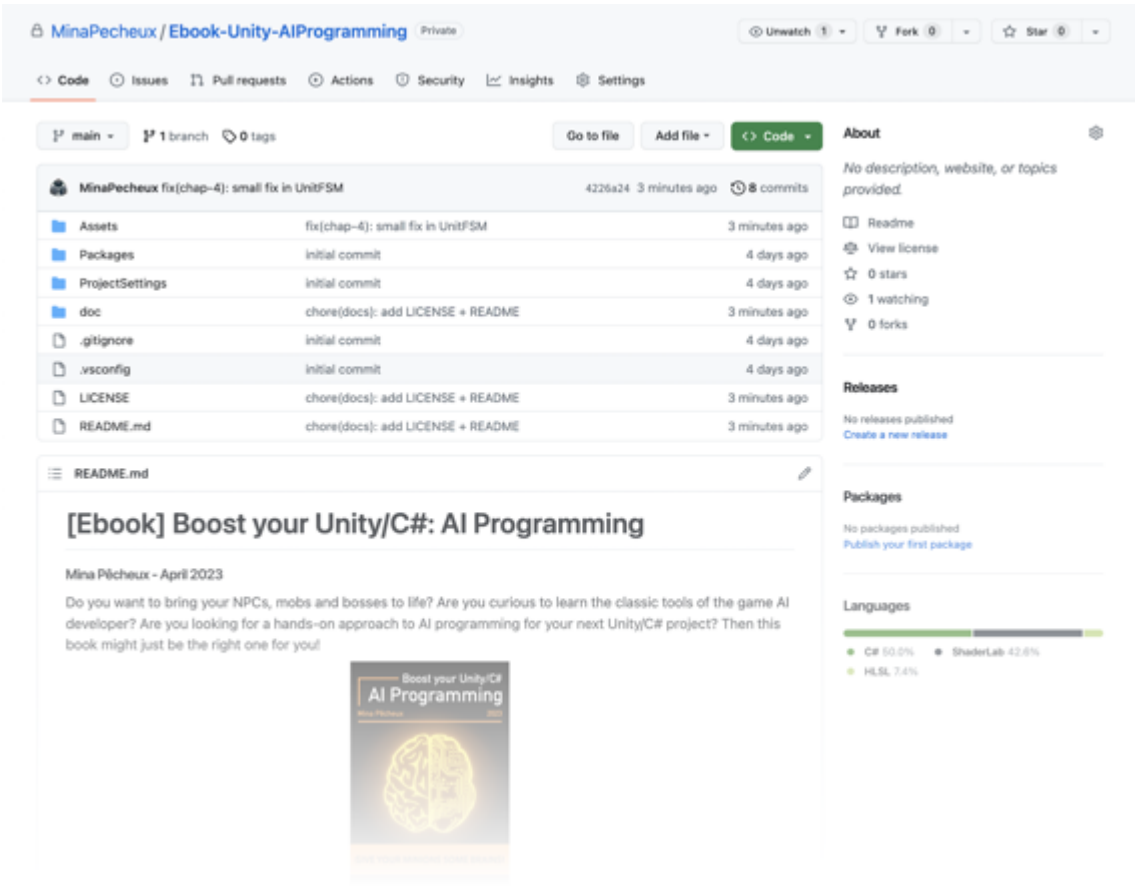


Figure 12.4 – Screenshot of the page of the Github repository for this book

USING A COMPATIBLE UNITY EDITOR VERSION

> This Unity project was made in the **2021.3.11f1 LTS** Unity editor. To avoid any incompatibilities, it is best to open it in the same version on your end – or at least, one that is roughly equivalent, like another 2021 Unity editor.

In this repository, you will find the assets and the code for *Chapters 8* and plus all their dependencies like the BehaviorTree C# package we made in *Chapter 7: Creating a behaviour tree toolbox* or the SOMD and UBAI libraries we discussed in *Chapter*

I wanted to share this along with the book because I strongly believe that practice is the best way to learn and improve, and tweaking pre-existing code is a nice bootstrap. You're therefore more than welcome to download, modify and play around with this project to your liking **for a personal usage** :)

Otherwise, the contents of the Github repository are licensed under the **CC BY-NC 4.0 license**

Figure 12.5 – The CC BY-NC 4.0 license imposes that you retain the name of the original author and you don't use the licensed content for monetary compensation

(The Github repository contains the full license file for more details.)

Summary

Well, here we are!

This final chapter concludes our exploration of common AI tools, and it hopefully provided you with some extra food for thought. From blending together multiple techniques to introducing modern neural networks and machine learning algorithms into your games, there is still much to be tested – this field has a long history, and it is in constant evolution... so perhaps the next generation of game AIs will be of yet a different breed, and you will contribute to shaping this new vision? :)

In any case, I hope you enjoyed this adventure and that you discovered a few interesting techniques to populate your AI developer's toolbox!

Thanks a lot for reading, and happy AI crafting...

13 - References & resources

Books

Playing J. Togelius (2019)

The Master P. Domingos (2015)

Game Programming R. Nystrom (2014): https://gameprogrammingpatterns.com/

Game AI Pro (series), multiple authors (since 2013): https://www.gameaipro.com/

Articles / Conference slides

Amit Patel's AI resources: http://www-cs-students.stanford.edu/~amitp/gameprog.html#ai

AI Architectures: A Culinary Guide (GDMag D. Mark (2012): https://web.archive.org/web/20220513064719/https://intrinsicalgorit

hm.com/IAonAI/2012/11/ai-architectures-a-culinary-guide-gdmag-article/

Behavior Trees for Computer Y. A. Sekhavat (2017): https://www.researchgate.net/publication/312869797_Behavior_Trees_for_Computer_Games

Creature Smarts: The Art and Architecture of a Virtual R. Burke, D. Isla, M. Downie, Y. Ivanov and B. Blumberg (2003): https://characters.media.mit.edu/Papers/gdc01.pdf

Intelligence Without R. A. Brooks (1991): https://people.csail.mit.edu/brooks/papers/BrooksIJCAI91.pdf

Three States and a Plan: The A.I. of J. Orkin (2006): http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf

Applying Goal-Oriented Action Planning to J. Orkin (2003): http://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf

Agent Architecture Considerations for Real-Time Planning in J. Orkin (2004): http://alumni.media.mit.edu/~jorkin/aiide05OrkinJ.pdf

Game AI Planning Analytics: The Case of Three First-Person E. Jacopin (2014): https://ojs.aaai.org/index.php/AIIDE/article/download/12728/12576

Killzone 2 Multiplayer A. Champandard, T. Verweij and R. Straatman (at Game AI Conference, 2009): https://www.guerrilla-games.com/media/News/Files/GAIC09_Killzone2Bots_StraatmanChampandard.pdf

HTN Planning: Complexity and K. Erol, J. Hendler and D. S. Nau (1994): https://www.cs.nmsu.edu/~tson/classes/spring04-579/HTN-planning.pdf

At-a-glance functions for modelling utility-based game A. Aitchison (2013): https://alastaira.wordpress.com/2013/01/25/at-a-glance-functions-for-modelling-utility-based-game-ai/

The Core Mechanics of Influence A. Champandard (2011): https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-core-mechanics-of-influence-mapping-r2799/

Using Genetic Learning in Weight-Based Game D. A. Kordsmeier (2015): https://scholarworks.uark.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1031&context=csceuht

Videos

AI and Games YouTube channel, T. Thompson: https://www.youtube.com/@AIandGames

AI and Game Design | The History of Artificial Intelligence In Video The Game Overanalyser (2020): https://www.youtube.com/watch?v=wh9kpe1Dn8s

Marvel's Spider-Man AI A. Noonchester (at GDC, 2019): https://www.youtube.com/watch?v=LxWq65CZBU8

Improving AI Decision Modeling Through Utility D. Mark and K. Dill (at GDC AI Summit, 2010): https://www.gdcvault.com/play/1012410/Improving-AI-Decision-Modeling-Through

Embracing the Dark Art of Mathematical Modeling in D. Mark and K. Dill (at GDC, 2012): https://www.gdcvault.com/play/1015683/Embracing-the-Dark-Art-of

Off the Beaten Path: Non-Traditional Uses of AI (excerpt), M. Robbins (at GDC, 2012): https://www.gdcvault.com/play/1015667/Off-the-Beaten-Path-Non

Implementing an event system [Unity/C# M. Pêcheux (2021): https://www.youtube.com/watch?v=EvqdcyTgZNg

How to use Unity's Scriptable Objects [Unity/C# M. Pêcheux (2023): https://www.youtube.com/watch?v=ZnHxxADBAQo

Assets

Tilesets in *Chapters 2* and *8* are from the *Kenney* https://kenney.nl/

Free Ancient markinhofaci (on CG Trader, Royalty Free license): https://www.cgtrader.com/free-3d-models/character/fantasy-character/free-ancient-warriors

*2D A** pixelfac (2021): https://github.com/pixelfac/2D-Astar-Pathfinding-in-Unity

Mage gadohoa (on CG Trader, Royalty Free license): https://www.cgtrader.com/free-3d-models/character/fantasy-character/mage-model

Cryo's Mini PaperHatLizard (on itch.io, under CC BY-NC 4.0 license): https://paperhatlizard.itch.io/cryos-mini-gui

Cartoon FX Remaster J. Moreno (on the Unity asset store): https://assetstore.unity.com/packages/vfx/particles/cartoon-fx-remaster-free-109565

Bonus

Opsive's *Behavior Designer* behaviour trees package on the Unity asset store: https://assetstore.unity.com/publishers/2308

About the Author

I am a freelance content creator who's been passionate about game development since an early age. I graduated from the French Polytech School of Engineering in applied mathematics and computer science. After a couple of years as a data scientist and web developer in startups, I turned to freelancing and online instructional content creation to reconnect with what brightens my days: learning new things everyday, sharing with others and creating multi-field projects mixing science, tech and art.

Read more at Mina Pêcheux's