An Accessible Behavior Tree Framework for Implementing and Designing Game AI

Master's Thesis

Anders H. Nissen
(anissen@cs.au.dk)

Supervised by Ole Caprani

September 2010

Department of Computer Science
Faculty of Science
University of Aarhus
Denmark

Abstract

An important part of modern computer game development is giving players the impression that non-player characters are behaving rationally. This is the responsibility of the game AI programmer and the game designer. The game AI programmer implements the platform which enable non-player characters to act within a game. The game designer designs the behavior of the non-player characters to fit the gameplay to increase the overall entertainment value of the game.

There are many practical issues related to these tasks. The game AI programmer must determine which technique has the properties needed for being able to achieve the desired behavior, implement it and make it available to the game designer through a suitable tool. The game designer must then use the tool to design and test the behavior. However, most game AI techniques are unsatisfactory and the tools for game designers are crude and troublesome to use.

We propose to use the novel behavior tree game AI technique to create a framework for improving the workflow of both the game AI programmer and the game designer as well as the interplay between them. The framework will provide the game AI programmer with an interface for conveniently exposing behavior from the game world in a modular and data-driven fashion. The process of designing behavior is made easier by a behavior designing tool emphasizing accessibility and an intuitive workflow. The tool is an integration of the game under development into a specialized behavior tree editor via the framework. Combining these components improves the usability and testability of the overall system.

Using the Unity game development tool and the Ext JS JavaScript library we have implemented a web-based tool that presents a novel approach for implementing and designing behavior for non-player character in games. We argue that this tool would be useful in the development of many different types of games.

Resumé

En vigtig del af udviklingen af et moderne spil er at give spilleren indtryk af at figurerne i spillet opfører sig rationelt. Denne opgave tilfalder spillets AI-programmør og spildesigneren. Spil-AI-programmøren implementerer platformen der tillader figurerne at handle i spillet. Spildesigneren designer figurernes opførsel så den passer til gameplayet og øger spillets underholdningsværdi.

Der er mange praktiske problemer vedrørende disse opgaver. Spil-AI-programmøren skal bestemme hvilken teknik der har de egenskaber der er nødvendige for at kunne opnå den ønskede opførsel, implementere den og gøre den tilgængelig for spildesigneren igennem et passende værktøj. Spildesigneren skal så bruge værktøjet til at designe og teste opførslen. De fleste teknikker for AI i spil er dog utilstrækkelige og værktøjerne som spildesignerne bruger er primitive og besværlige at anvende.

Vi foreslår at anvende den nye behavior tree spil-AI teknik til at opbygge et skelet der kan forbedre arbejdsforløbet for både spil-AI-programmøren og spildesigneren og samspillet imellem dem. Skelettet vil stille et interface til rådighed for spil-AI-programmøren der gør det bekvemt at blotlægge opførsel fra spilverdenen på en modulær og data-dreven måde. Arbejdsforløbet med at designe opførsel er gjort lettere og mere intuitivt af et lettilgængeligt værktøj til at designe opførsel. Værktøjet er en integrering af spillet under udvikling ind i et specialiseret behavior tree redigeringsværktøj igennem et bekvemt skelet. Kombinationen af disse komponenter øger anvendeligheden og testbarheden af det samlede system.

Ved at anvende spiludviklingsværktøjet Unity og JavaScript-biblioteket Ext JS har vi implementeret et web-baseret værktøj der præsenterer en ny tilgang til at implementere og designe opførsel for figurer i spil. Vi argumenterer for at dette værktøj ville være nyttigt i udviklingen af mange forskellige typer spil.

This page is intentionally left blank.[1]

[1] http://www.this-page-intentionally-left-blank.org

Contents

Chapter 1

Introduction

In this section we will outline the aims of the thesis. We will start by motivating the reasons for our choice of problem domain. We will then describe the concrete problems we will investigate and the methods we have chosen for doing so.

1.1 Motivation

The very first computer games were released in the 1950s and 1960s, [64]. Since then, there has been a tremendous increase in the hardware capabilities of gaming systems and overall improvements in the software technologies used to create games. These advancements have lead to increasingly realistic games. They have also brought with them an enormous incline in the complexity of developing a computer game:

> *With the ever-increasing processing and graphical capabilities of computers and console products, along with an increase in user expectations, game design moved beyond the scope of a single developer to produce a marketable game in a reasonable time,* [61]

Today, the game industry is big business. Even bigger than the movie industry, [61]. Now most commercial games are the result of years of development by large development teams, [70]. In the larger development teams each developer has a specific role in the development. Developing a computer game requires producers, writers, programmers, artists, designers, testers and many others.

Most development teams have developers dedicated to creating a platform for **artificial intelligence** (abbreviated **AI**) in the game. Such a platform is often called the **game AI** of the game and the developers responsible for implementing and maintaining it are called **game AI programmers**. The game AI platform is concerned with the behavior of **non-player characters** (abbreviated **NPCs**) in the game. Non-player characters are all the characters

artificial intelligence

AI

game AI

game AI programmers

non-player characters

NPC

in the game that are controlled by the computer rather than other players. Figure 1.1 shows several NPCs in a combat situation in the game **Half-Life 2: Episode Two** from 2007 by Valve.

Half-Life 2: Episode Two

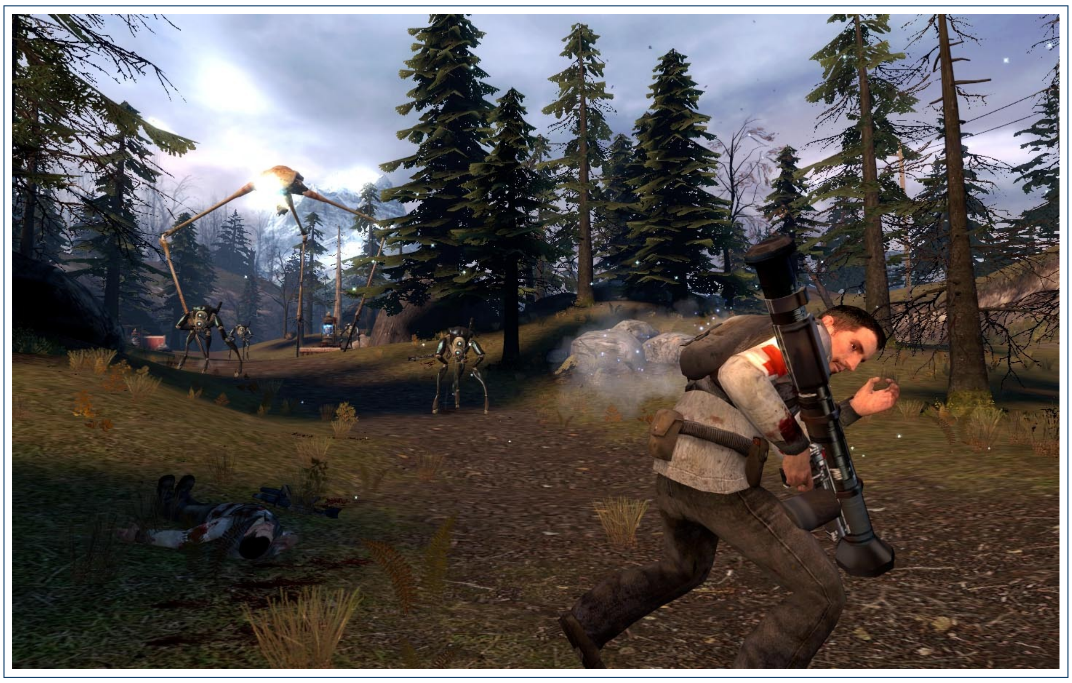


FIGURE 1.1: *Ally and enemy NPCs engaged in combat in* Half-Life 2: Episode Two, *[52]*

The game AI platform is used as a tool for making the NPCs act as if they were rational beings. However, the game AI platform only handles the technical parts of the NPC behavior, such as *how* to do something, e.g. how to move. The NPCs still need to be told *what* to do, *when* to do it and precisely *how* it should be done. For instance, the NPC might know how to move, but needs to be told when to move, where to go and whether to walk or run. Choosing what an NPC should do next is called **decision making** and is the responsibility of the **game AI designer**, or simply game designer. The goal for the game designer is to create NPCs that help to increase the entertainment value of the game.

decision making

game AI designer

There are many techniques for creating the game AI platform. The different techniques all have strengths and weaknesses, but very few of them are well suited for aiding the game designer in creating the behavior of the NPCs. Often custom tools are built by the development teams specifically to aid the game designers in creating the desired behavior, [62]. Many tools are coupled so tightly with the game for which they are used, that the tools must be almost completely rebuilt for the development of the next game. Additionally, many tools have a high learning curve, are troublesome to use and lower the productivity of the game designer, [62, sec. "Use"]. For game designers to be the most productive and creative, they need intuitive tools that make it easy to design, modify and test behavior of NPCs.

Games are increasingly judged by the quality of their game AI and NPC behavior on par with the graphics, audio and story:

> *The era of eye-catching graphics is approaching its peak, opening the path to AI as one of the major ingredients for a successfully commercial game,* [72, p. 41]

In the game industry only the top 5% of the products make a profit, [61]. Having easily accessible tools for creating and testing the behavior of NPCs might give the extra edge that turns a game into a commercial success.

1.2 Problem description

It is often a troublesome process for the game designer to create the desired behavior of NPCs. The two main causes lie with the game AI platform and the tools for creating behavior.

Often, a large part of the problem is the game AI platform provided by the game AI programmer. The game AI platform may be insufficient for designing the desired behavior or it may be inconvenient for the game designers to use. The process of designing behavior is an interplay between the game AI programmer and the game designer. In general, it is hard for the game AI programmers and game designers to agree on a common interface between the game AI platform and the tools for creating behavior that satisfy the needs of both parties, [42, p. 11].

As an example, game designers will often require a game AI platform that allows them to create sequential behavior while at the same time being able to react quickly to new events in the game. Likewise, game designers would like NPCs to be **autonomous**, i.e. to be able to act on their own without being told what to do, while still allowing the game designers to tailor every aspect of the behavior when needed.

autonomous

It can be hard for the game AI programmer to find a game AI technique that makes it possible to satisfy these requirements. Often, game AI programmers rely on the concept of **scripting**, **finite state machines** or **hierarchical finite state machines** to create the game AI platform. Neither of these solutions are particularly intuitive nor suited for the requirements of both the game AI programmers and the game designers, [23].

scripting

finite state machines

hierarchical finite state machines

The tools the game designers use to design the behavior are often crude, cumbersome, very user-unfriendly and with little or no built-in help, [62, sec. "Use"]. Also, they are often coupled very tightly with the game for which they are used. This means that most of the tools need to be rebuilt for each game, thereby not warranting big investments. Game designers are often stuck with having to script all the behavior, a time consuming and sometimes difficult approach. Alternatively, the tools that enable the game designers to design behavior may not allow them to easily test the behavior thus lowering productivity.

In recent years, a novel approach for handling behavior, called **behavior trees**, has emerged. It has gained widespread popularity and recognition in the game industry, and has

behavior trees

Halo 2

Spore

been used as the game AI platform in critically acclaimed games such as **Halo 2** from 2004 by Bungie Studios and **Spore** from 2008 by Maxis, [37], [35]. The behavior tree technique combines the strengths of several popular techniques, most notably scripting, finite state machines and hierarchical finite state machines, [42, p. 334]. Unlike many other techniques, the concepts behind the behavior tree technique are very simple and intuitive. As such, this technique may be very well suited for both game AI programmers as well as game designers, [23].

We would like to investigate if the novel behavior tree technique may be used to create better and more intuitive tools for game designers to aid in creating the desired behavior of NPCs. Giving game designers better suited tools will increase the quality of the behavior of the NPCs, thereby increasing the overall quality of the games.

1.3 Method

We will try to make it easier for game designers to design the desired behavior of NPCs. For that purpose, we have chosen to investigate some of the concrete issues outlined in the previous section:

- First, we will make a survey of the most common and popular techniques for game AI and try to distill the properties that are well suited for games. We have chosen to investigate a specific technique called behavior trees, and will compare this techniques to similar techniques. From this, we will determine if the behavior trees technique is suitable for game AI programmers.

- Second, we will investigate how to create a more suitable tool for game designers. The tool should allow game designers to easily and intuitively create, modify and test the behavior of NPCs. For these purposes we propose to integrate the game under development into this behavior design tool, combining the two systems and allowing the game designer to directly test and interact with the game through the tool.

- Third, we want to investigate the practical implications of such a system on the workflow of the game AI programmer and the game designer. For this purpose, we will create a game and a behavior design tool for designing and testing the behavior of the NPCs. We will use the **Unity** game development tool to create the game system, the implementation of the behavior tree game AI technique and the interfacing framework. For creating the tool for the game designers, we will create a rich web-based behavior editor using the **Ext JS** JavaScript library.

  Unity

  Ext JS

- Finally, we want to make the process of implementing and designing the desired behavior more convenient for both the game AI programmers and the game designers. For this purpose, we will design a framework to act as a general interface connecting the game and the behavior design tool.

  In addition to the framework, we will make the tool general enough to be used for different games without having to modify these components.

1.4 Typographical conventions

Certain typographical conventions are used throughout the thesis:

- The occurrence of a new keyword is highlighted in bold, e.g. **Keyword**. It is also listed in the page margin and in the index.

Keyword

- The first occurrence of a reference to a game is followed by the name of the development studio and the year of its release in square brackets, e.g. Pong [Atari Inc., 1972]
- Code is set in a typewriter font, e.g. `true`
- Class names are set in a dark blue typewriter font, e.g. `Agent`
- Nodes in behavior trees are set in a dark red typewriter font, e.g. `Sequence`
- Web addresses are marked in a violet typewriter font, e.g. `www.google.com`
- Spoken or exclaimed statements are set in quotes and italic style, e.g. "*Hello world*"
- References to pages, figures, tables and listings are marked in blue, e.g. section 2.7.2
- Citations are marked in red and enclosed in square brackets along with a page, section or chapter reference whenever possible, e.g. [50, p. 123] or [50, sec. 1.2]
- Quotations are set in an italic font and shown in a floating figure between oversized quotation marks. Any changes to the original quote is enclosed in brackets and in a regular font. The quote is ended with a reference to the source of the quote. E.g.

“ *A quote that is relevant* [in the context of game AI] *and insightful, [50, p. 123]* ”

1.5 Related material

In this thesis we, among other things, discuss the design and implementation of the "Behavior Tree Workbench" system. This system can be found at the web page at the following address:

- `https://sites.google.com/site/BehaviorTreeEditor`

The system is web-based and runs in all internet browsers on the Windows and Mac platforms. The page also contains the source code and the resources for the system, an user guide and a concrete game example.

The same material can also be found on the enclosed CD. Here, the Behavior Tree Workbench system is run by activating the link in the main directory or opening "code/editor/bte.html" in an internet browser.

Chapter 2

Game development and game AI

In this chapter we will look at the history of game development to see how games and the development process has changed. We will then discuss what constitutes a game agent and game environment and make definitions of both. Following this discussion, we will specify the different types of agents and which environments they are suited for.

In order to get a firm understanding of the use of artificial intelligence in games is we will relate it to classical academic artificial intelligence and illustrate using concrete game examples. We will elaborate on the concept of game AI, how it is viewed by different interested parties, how it is executed and what types of overall game AI approaches are used. Following up on these approaches, we discuss many of the most common and popular techniques for implementing, handling and designing game AI.

2.1 The process of developing a game

Pong

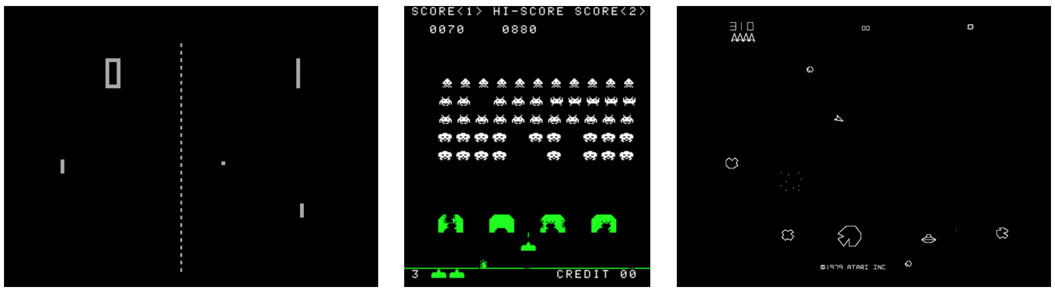
Space Invaders

Asteroids

The first computer games as we know them appeared in the 1970's. Games such as **Pong** [Atari Inc., 1972], **Space Invaders** [Taito Corporation, 1978] and **Asteroids** [Atari Inc., 1979] (figure 2.1 on the next page) gained widespread success on arcade machines, [64]. These games helped to bring video games into the mainstream.

In the 1970's and 1980's the process of developing a game was very different than today. Many games were created by a single enthusiastic amateur programmer in a matter of months. In fact, the development of a complete game could take as little as six weeks, [61, sec. "Overview"]:

> *In the early days of the industry, it was more common for a single person to manage all of the roles needed to create a video game. As platforms have become more complex and powerful in the type of material they can present, larger teams have been needed to generate all of the art, programming, cinematography, and more,*



(a) Pong, 1972 (b) Space Invaders, 1978 (c) Asteroids, 1979

FIGURE 2.1: *Early but highly popular games that ran on arcade machines, [64]*

[70]

Today, the game industry has become so big that in 2007 it exceeded the revenue of the movie industry, [61, sec. "History"]. As the game productions have grown in size, so have the development teams behind them. The members of the teams have also become increasingly specialized. Previously, many programmers also created the art, story, gameplay and levels of the games. Now there are numerous specialized positions for handling these different aspects of the development. These feature artists, story writers, producers, game designers, level designers, audio designers, tool programmers, engine programmers, quality assurance personal, testers and many more. All of these different people must work together in a shared effort to produce a modern computer game.

2.1.1 Tools for game development

As a consequence of the increased specialization, the tools used in the development process have become more suited to specific purposes. For instance, level and game designers often use a customized level editor and the artists, 3D modelers and animators use powerful image editors and 3D modeling tools:

> *Early in the history of the video game industry, game programming tools were non-existent. This wasn't a hindrance for the types of games that could be created at the time, however. While today a game like Pac-Man would most likely have levels generated with a level editor, in the industry's infancy, such levels were hard coded into the game's source code, [62, sec. "History"]*

Some tools are quite general-purpose and can be purchased off-the-shelf and used for most games. Such tools include image, sound and 3D editors. Other tools are much more intertwined with the game under development. These tools are typically developed in-house by tool programmers. The tools are developed by programmers and not by the end users, e.g. the game designers. As a result the tools will often contain what the programmers *think* the end users need, which might be quite different from what they actually need. In the best case the tools merely lack some features that could be desired, [25]:

> *Since facility is often the primary goal for tools, they may be very user-unfriendly, with little or no built-in help, [62, sec. "Use"]*

As a result, the non-programmers (e.g. game designers) having to use the tools might find them cumbersome and slowing the content creation pipeline. Having powerful yet easily accessible tools is vital when developing a game:

> *Stable and reliable toolchains are a hot topic in game development, as they ensure that the artists and designers can create the content in an easy way, while allowing the content to be inserted into the game without manual help, [42, p. 33]*

Tools for designing behavior for non-player characters in games is no different, [66]. How the NPCs should behave depend on the specific game. Suppose, for instance, that an evil-minded monster is approaching a NPC in a game. A sensible course of action for that character would be to flee. This assumes that the character is *aware* of the approaching monster. That is, that the game designer is able to detect this occurrence within the tool and make the character act accordingly. This again requires specific knowledge about the character. If the character has no legs or other means of moving himself, fleeing from the monster is an impossible task. Thus, the behavior of characters are often coupled so tightly with the game that specialized tools need to be used.

2.1.2 Behavior in games

To be able to create interesting games, we need to be able to design the behavior of more than non-player characters. An apple tree wouldn't be considered a NPC[1], for instance, but it might still have some behavior associated with it. It may drop apples when they are ripe or if the trunk is kicked. So, we need to broaden our definition of what we need to design behavior for. In doing so, we turn to a well-known concept from computer science and robotics, namely that of an "agent". More precisely, we need to model behavior for a "game agent". A game agent is an agent in the setting of a game. A game agent models a single entity in a game and have all the visible properties of that entity. This could be eyes, arms,

[1]That is, under the assumption that the game do not allow players to assume the role of various fruit trees

legs, ability to move around etc.. It could also be apples in our example with the apple tree. Additionally, game agents also have all the internal properties of the entities. In intelligent entities this could be things such as a rational mind and a working memory. Although game agents can be very different from game to game, some elements are persistent – the elements that define a game agent.

In the following sections we will factor out the properties that agents share across all contexts. In particular, we will derive a definition of a game agent that is independent of game types and genres.

2.2 Definition of a game agent

agent

game agent

In this section we will ask the question "What is an **agent** in a computer game?". We will use this discussion to create a definition of what constitutes a **game agent**.

Pac-Man

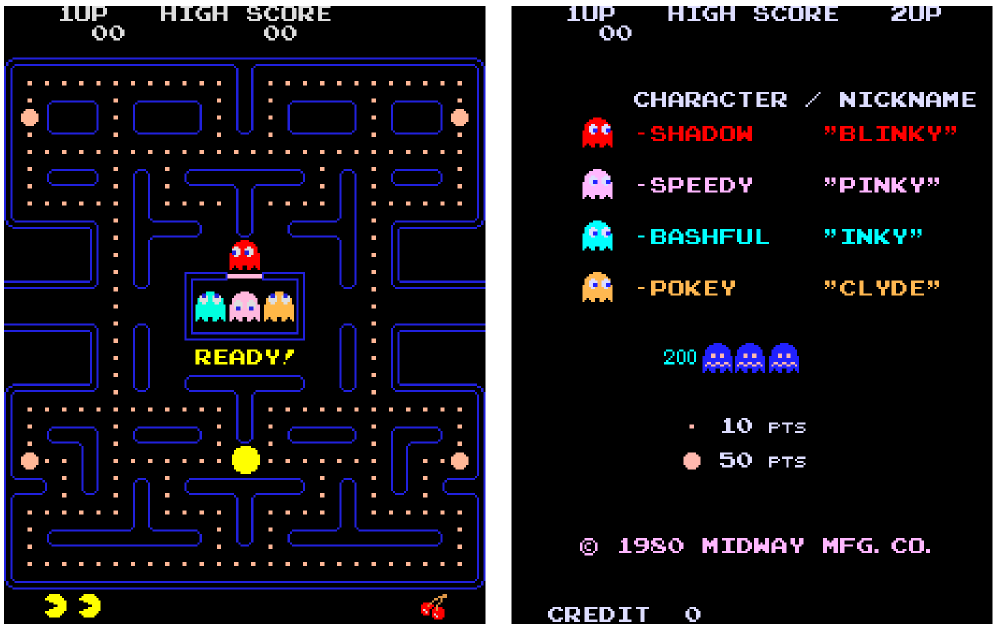
In order to motivate this discussion let us examine a concrete example: the agents in **Pac-Man** [Namco, 1980]. Pac-Man (figure 2.2 on the facing page) is a very well-known arcade game, still immensely popular to this day, [67]. The object of the game is to navigate the players avatar — a yellow character named "Pac-Man" — around in a maze while eating all the dots and avoiding the ghosts[2] chasing him. The score of the player is increased for each dot eaten and each level passed[3] and getting the highest score is the ultimate goal of the game.

Which components of Pac-Man can be classified as "agents"? The four ghosts "Blinky", "Pinky", "Inky" and "Clyde" (figure 2.2(b)) are obvious candidates. The Pac-Man character might also be classified as an agent even though Pac-Man is being controlled by the player. What about the dots and the walls? They do not seem to have the properties required for being an agent. Where, then, is the distinction? Primarily, the dots and walls are inanimate. Had the dots been trying to flee once the player came near, the situation would have been different. The dots would then be ascribed some instinctive properties of living things, e.g. self preservation.

Considering the dots as being alive leads to the player empathizing with them, i.e. putting himself in their place and sharing their emotions. In the case of the fleeing dots, the emotion would be fear of being eaten. The obvious reaction would be to try to flee. However, what if the dots were to react differently, e.g. by only fleeing slightly and then returning to their original position regardless of the risk of being eaten? That reaction does not make sense w.r.t. self preservation and a player would not be able to put himself in their place and would thus loose empathy for them.

[2]Although originally called "monsters", the enemies in Pac-Man became known as "ghosts" after the game was ported to the Atari 2600. The reason was a bug that caused the enemies to constantly flicker, thus appearing ghost-like, [42, p. 7]

[3]The right half of the 256th level of the game is rendered incorrectly due to an overflow bug, making the level almost impossible to complete. However, this feat was apparently accomplished in 1982 by an 8-year old boy, who supposedly received a letter of congratulations from U.S. President Ronald Reagan [67]



(a) Game screen (b) North American title screen

Figure 2.2: *Screens from the classic arcade game* Pac-Man, *[67]*

From this discussion we can draw the following conclusions; agents are animated and react to their surrounding in order to fulfill needs (e.g. stay alive) and archive goals (e.g. catch Pac-Man). Additionally, the agents need to behave as expected for the player to consider them realistic, i.e. maintain **behavioral integrity**, [37].

behavioral integrity

2.2.1 Agent environment

An important aspect of agents is the ability to interact with the **environment** (section 2.3 on page 15). The environment is the surroundings that the agent operates within. If the ghosts had no way of observing their environment or moving around to chase the player, they might as well have been dead, inanimate objects. An agent needs to be able to get relevant input from, as well as manipulating, its surroundings. The ghosts know the layout of the maze, are aware of the location of the player and can maneuver through the maze to try to catch him.

environment

Pac-Man has a way of observing the environment through the eyes of the player and can move in the maze by reacting to input from the player. Our definition of an agent does not differ between how an agent is controlled. In the literature the notion and definition of an agent vary. One widely recognized definition is by Russell and Norvig, [50]. In their definition, an agent is "anything perceiving its environment through **sensors** and acting upon that environment through **actuators**", [50, p. 32]. This is the overall definition we will use.

sensors

actuators

The interaction between the environment and the agent is illustrated in figure 2.3 on the following page. The environment and the agent are here represented as abstract entities that

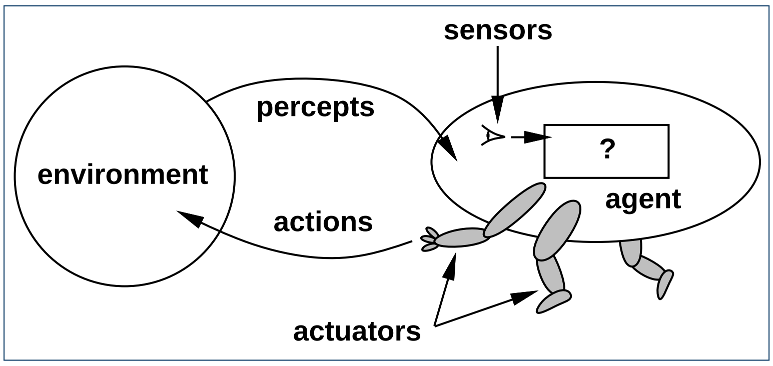


FIGURE 2.3: *Agent and environment interaction, [50, p. 32]*

interact through percepts and actions. The sensors and actuators are components of the agent.

The sensors provide mechanisms by which some aspects of the environment can be observed, [50, p. 32]. Sensors for ghosts could be wall-detectors and player location sensors. Sensors for Pac-Man could be movement input sensors. The sensory input is **stimuli** from the environment. The stimuli is analyzed and interpreted as a perception input or **percept** (figure 2.4 on the next page[4]). The percepts are the agent's abstract representation of that particular stimuli. Returning to our example, the percept from the ghosts' wall-detector could be "walls to the north and east" and the percept from Pac-Man's movement input sensor could be "the 'up'-key is pressed".

stimuli

percept

The actuators provide a way for the agent to interact with the environment, [50, p. 32]. Actuators could be motor devices to enable movement in the maze and a mouth for Pac-Man to be able to eat the dots. The actuators are responsible for changing the state of the environment. Whether such a change actually occurs is not necessarily observable by the agent. Agents control their actuators by issuing **commands**, or more abstractly, by using **actions** (figure 2.4 on the facing page[5]). An actuator acts according to a command which causes a change in the environment. This change is called the **response**. For instance, an agent might desire to move forward and sends this command to its motor actuators. The motors will operate in order to comply with the request. The response will probably be that the agent is moving. However, the agent could be stuck, in which case the response will either be the observation that the agent *tries* to move, but cannot, or that it simply does nothing. An agent that knows beforehand which responses its actions will cause is called **omniscient**. Although omniscient agents may be possible in some virtual or closed environments (such as Chess), omniscience is impossible in reality as unpredictable events may occur, [50, p. 36].

commands

actions

response

omniscient

The **percept sequence** is the complete history of everything the agent has perceived.

percept sequence

[4]In our terminology, the sensors do not receive percepts from the environment, but rather receives stimuli that is interpreted as percepts

[5]In our terminology, the actuators do not act upon actions, but rather acts upon commands that is carried out as actions

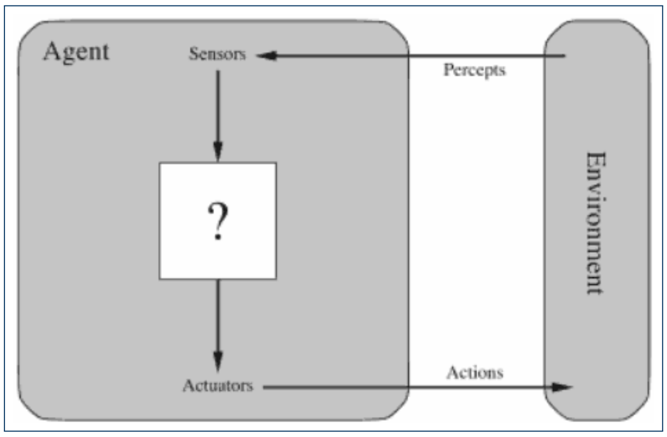


FIGURE 2.4: *Agent interacting with its environment with sensors and actuators, [50, p. 35]*

From this, the **agent function** can be specified as an abstract mapping from percept sequences to actions, i.e. $f : \mathcal{P}^* \rightarrow \mathcal{A}$, where $\mathcal{P} \in$ Percepts, $\mathcal{A} \in$ Actions. A concrete implementation of an agent function is called an **agent program**, [50, p. 32-33, 44].

agent function

agent program

2.2.2 Agent behavior

The **behavior** of an agent can be specified as the response from the actions of the agent, as perceived from the environment. This behavior is what an observer will use to construct a mental model of the agents knowledge of the environment and its mapping from percepts to actions. This model is called the **external state** of the agent. There might be a discrepancy between the perceived and actual state of the agent, i.e. its external and **internal state**. For instance, a ghost might appear to be fleeing if it moves away from Pac-Man while in fact it has found another path that requires it to make a small detour. The illusion of intelligent behavior is quickly broken if an observer cannot relate the actions of the agent to what the observer believes is the internal state of the agent.

behavior

external state

internal state

The desired behavior of an agent is defined by the **intentionality** of the designer rather than of the agent itself, [40, p. 175]. The agent only does what it is designed to do, but this may not be what the designer actually *intended* it to do. In other words, we want the agent to do what is rational (i.e. what we intend it to do) in any given context. We call such an agent a **rational agent**, [50, sec. 2.2]. However, for defining a rational agent, we need to be able to specify what rational behavior actually constitutes for an agent. Rational behavior is the behavior by which the agent maximizes its **performance measure**, [50, p. 35]. The performance measure of an agent is a criterion for measuring the success of an agents behavior, i.e. the degree to which the behavior of the agent matches its intended purpose. In the definition of Russell and Norvig, a rational agent is an agent that "should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has", [50, p. 36].

intentionality

rational agent

performance measure

autonomous

An important aspect of agents being able to act rationally is to be able to overcome initially unknown problems. Agents able to act on their own are called **autonomous**, [50, p. 37-38]. For an agent to maximize its performance measure without autonomy, it would need exhaustive knowledge of the environment or at least how to handle every possible scenario that could occur in the environment. Suppose, for instance, that a certain route taken by the ghost agents in Pac-Man is hardcoded. If the layout of the maze was to be changed to invalidate that route, the agents would be incapable of finding an alternative route to their destination. By using autonomous agents, they could determine a route to the destination and, failing to use that route, determine an alternative route based on the new knowledge of the layout of the maze.

2.2.3 Agent control

robot control

The concept of **robot control** refers to the way in which the percepts and actions (i.e. the agent function) for an agent (or robot) are coordinated[6]. In practice, there are four different schemes of control, [40, p. 191]. They are defined as follows.

Reactive control

- **Reactive control** (or "don't think, react") uses a collection of concurrent stimuli-response pairs (i.e. percept-action pairs, page 12) to be able to react instantly to changes in the environment.

Deliberate control

- **Deliberate control** (or "think hard, then act") uses sensory information and internal knowledge to create a plan of action (section 2.7.4)

Hybrid control

- **Hybrid control** (or "think and act independently, in parallel") combines reactive and deliberate control schemes to be able to react to stimuli as well as planing ahead.

Behavior-based control

- **Behavior-based control** (or "think the way you act") are similar to the hybrid control scheme but uses a network of behaviors rather than a single planner

None of these approaches are better than others – which control scheme to use depends entirely on the environment (section 2.3) that the agent operates within. All of the approaches have strengths and weaknesses which will also be addressed in section 2.4.

sequential control

When creating artificial behavior by coordinating lower-level actions, we often distinguish between two different approaches; **sequential control** or reactive control, [40, p. 190]. Sequential control is used to accomplish some (complex) task by performing a sequence of predetermined actions in order and may be considered a special case of deliberate control.

agent architecture

softbots

An **agent architecture** is a computing device with sensors and actuators. It can be either a physical embodiment or a purely virtual construct as in the case of software robots or **softbots**. For instance, a game agent (section 2.1.2) is a softbot. An agent can be specified by its architecture and agent program, [50, p. 44].

We must acknowledge the fact that an agent cannot stand on its own. An agent is always part of a setting. This setting defines the world the agent operates in. Having a clear

[6]These concepts are drawn from the field of robotics. Pioneers within the field such as Fred G. Martin, Maja J. Matarić and Rodney Brooks have all been concerned with approaches to robot control, [40, ch. 5]

definition of the world is essential for being able to define an agent that operates as desired. In the next section we will create the definition a task environment.

2.3 Definition of a game task environment

In the preceding section we arrived at a definition of an agent in computer games. However, an agent always operates within a **task environment**. A task environment can be thought of as "the problem" to which the agents are "the solution". In this section we will take a closer look at the properties of the task environment and how it affects the design of agents. In the preceding section we discussed agents in the context of Pac-Man. Let us continue this discussion by looking at the task environment in Pac-Man.

task environment

The environment can be thought of as the world excluding the agents. In other word, the surroundings of the agents. In the world of Pac-Man, the entirety of the maze is visible to both the player and the ghosts. The ghosts have complete information about the layout of the maze and the whereabouts of the player. The maze itself does not change over time. However, actions of the agents, such as moving around and eating dots, change the state of the environment. The result of the agents actions depend on the state of the environment. For instance, it is not safe to move Pac-Man to the left if there is a nearby ghost in that direction. The environment changes in a deterministic way and is, in addition to Pac-Man, only influenced by the ghosts. Although the behavior of the ghosts may seem rather random, they are in fact strictly deterministic, [45, ch. 2]. A different set of properties would apply for a version of Pac-Man that features a physics based environment (figure 2.5). In this version the environment does not remain static but can be modified by the user. Hence, the agents in this version of the game would have to be designed to consider a modifiable environment.

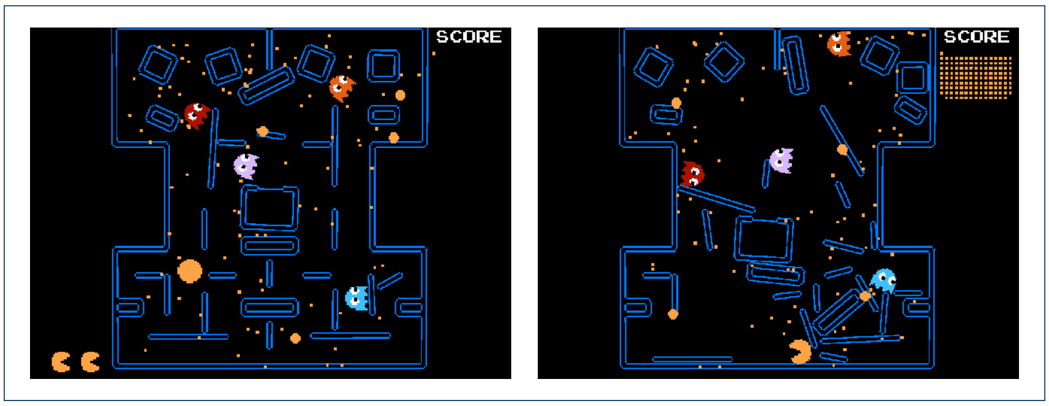


Figure 2.5: *Screens from a non-commercial version of* Pac-Man *where the maze can be manipulated using physics. The user can apply force to move the walls of the maze thus radically changing the layout of the maze, [6]*

2.3.1 Properties of task environments

The description of the environment in Pac-Man does not tell us much unless we are aware of what the various properties entail. The task environment has a number properties. One property is the agents' observability in the environment. If the state of the entire environment is available at any given time, we say that the task environment is "fully observable" as is the case in Pac-Man. Had the ghosts had a line of sight[7] then the task environment would have been only "partially observable" and the ghosts would have had to search the maze to find Pac-Man. Important properties concerning task environments are the following, [50, p. 40-44].

Fully observable

partially observable

- **Fully observable** vs. **partially observable**

  If the complete state of the environment is available to the agent through its sensors at any given time then the environment is fully observable. Otherwise it is partially observable.

Deterministic

stochastic

strategic

- **Deterministic** vs. **stochastic** (nondeterministic) vs. **strategic**

  If the state of the environment is completely determined by the current state and the actions performed by the agent, then the environment is deterministic. If the environment has unpredictable events that change its state, it is stochastic. If the environment is partially observable, a deterministic environment may seem stochastic because only limited information is available. The properties are from the point-of-view of the agent. If the environment is deterministic except for the actions of other agents, then it is strategic.

Episodic

sequential

- **Episodic** vs. **sequential** (nonepisodic)

  The environment is sequential if the order in which actions are performed can change the outcome of those actions. If each action is atomic with the same outcome regardless of ordering, it is episodic. Computer games are almost always sequential and the gameplay evolves by building on top of previously performed actions by the player.

Static

dynamic

semidynamic

- **Static** vs. **dynamic** vs. **semidynamic**

  If the environment can change while the agent is determining its next action then the environment is dynamic, otherwise it is static. A turn-based game like Chess is clearly static because nobody but the active player can change the state of the environment. Some games – like Pac-Man – are harder to classify. From the viewpoint of the player the game may seem dynamic. Whether the task environment is static or dynamic from the viewpoint of the agents depend on the concrete implementation.

  If the environment is static but the agent's performance score changes over time the environment is called semidynamic. This might be the case in games in which time is

[7] "Line of sight" is a notion of visibility where an agent can only observe it's environment within a given radius. Obstacles, such as walls, may cut off the line of sight, thus disallowing the agent to see behind obstacles. See `http://wikipedia.org/wiki/Line_of_sight_(gaming)`.

a factor. If the ghosts in Pac-Man had a measure of success based on how quickly the could catch Pac-Man, then this would be an example of a semidynamic environment.

- **Discrete** vs. **continuous**

  Discrete

  continuous

  If the state of the environment, passage of time and percepts and actions of the agent can be though of as discrete or can be discretized without losing data, the environment is discrete. Otherwise it is continuous.

- **Single agent** vs. **multiagent**

  Single agent

  multiagent

  If the environment have other entities which are best described as trying to maximize a performance measure (section 2.2) that depend on other agents then the environment is multiagent, otherwise it is single agent. If an agent maximizes its performance measure by minimizing the performance measure of other agents, then the environment is a **competitive multiagent** environment. If it instead tries to maximize the performance measure of other agents, then the environment is a **cooperative multiagent** environment.

  competitive multiagent

  cooperative multiagent

  Pac-Man is a clear example of a competitive multiagent system where the other agents (the ghosts) maximize their performance measure by catching Pac-Man, thus minimizing his performance measure.

The Pac-Man task environment can now be defined much more formally. From the point-of-view of the agents (i.e. Pac-Man and the ghosts) the environment is *fully observable*, *strategic*, *sequential*, *static*, *discrete* and *competitive multiagent*. Hence, the agents in this environment must be designed with these properties taken into consideration. For instance, if we were to implement a secondary, computer-controlled, Pac-Man, we had to consider that the environment may change. Specifically, the dots disappear as they are eaten by the player-controlled Pac-Man. Thus the new agent frequently needs to update it's view of the world to keep it consistent.

2.3.2 Specifying task environments

As mentioned in section 2.1, we also need to know which sensors and actuators are available to the agent. If an agent has no means of observing the world or moving about it cannot perform the desired behavior. A task environment can be specified with a **PEAS** (Performance measure, Environment, Actuators, Sensors) description, [50, p. 38]. In figure 2.6 on the following page the PEAS description for the agents in Pac-Man is shown. The performance measure of the ghosts is to catch Pac-Man such as to minimize his performance measure. On the other hand, the performance measure of Pac-Man (i.e. the player) is to get the highest score possible, achieved by eat dots, completing levels and avoiding ghosts to survive.

PEAS

The task environment must always be carefully specified prior to designing an agent to operate in that environment. If the task environment specification is incomplete, the designer will lack information required for designing the agent to operate correctly in that environment. In other words, it is not possible to formulate a suitable "solution" (the agent) if

Agent	**Performance Measure**	**Environment**	**Actuators**	**Sensors**
Ghost	Offer entertaining gameplay	Maze	Move in maze	Maze overview, player location
Pac-Man	Get high score	Maze	Move in maze, eat dot	Maze overview, location of ghosts and remaining dots

FIGURE 2.6: PEAS *description for the agents in* Pac-Man

the "problem" (the environment) is not well-defined. Different types of agents are needed for handling different types of environments. The next section will specify some of the primary types of agents.

2.4 Types of agents

Agents come in great variety. Depending on the task environment and the purpose of the agent different types of agents are applicable. For instance, if the environment is fully observable (section 2.3.1) then that agent do not need to maintain a model of the world – it can simply read the state when required. Some agents are simple in terms of their agent function. The resulting behavior need not be simple, however, and can appear complex to an observer. This is the case with the Braitenberg vehicles (figure 2.7 on the next page) whose agent function is simply a direct mapping between sensory input strength and motor speed. Other agents may have a much more involved agent function that might include goals and the ability to plan a way to achieve those goals.

It is convenient to be able to distinguish between different types, or classes, of agents. Agents can be classified based on how they select their next action(s). Here we have four basic types of agents from which all agents derive, [50, p. 46]. These are listed below, in order of increasing complexity.

Simple reflex agent

- **Simple reflex agent** is the simplest type of agent. It selects actions based solely on the current percept. This agent only works as intended if the desired action can be selected on the basis of the current percept. Therefore this type of agent is not suited for partially observable environments, for instance. It is comparable to the reactive control scheme (section 2.2.3).

Model-based reflex agent

- **Model-based reflex agent** is a type of agent that maintains an internal state to reflect the state of partially observable environments. In a partially observable environment not all state is available to the agent at any given time. To handle this problem, the agent keeps track of its observations in the world (its percept sequence) to create a knowledge base to act as a **model** of the world. This works as a memory bank for

model

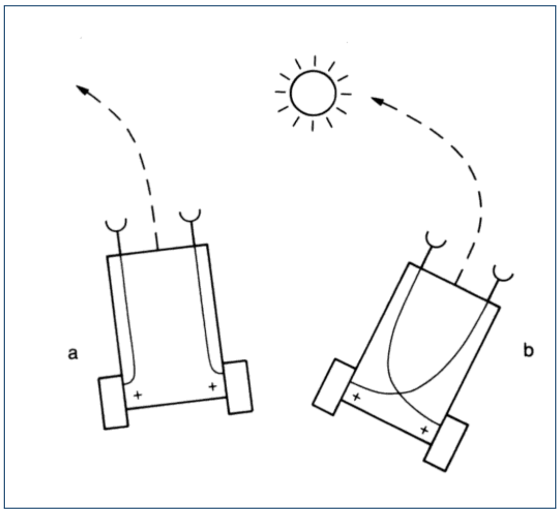


Figure 2.7: *Examples of Braitenberg vehicles with two sensors both connected to a motor. Vehicle* a *orients away from the source and vehicle* b *toward it. If the sensors were light sensors the source would be a light emitter, [10, p. 9]*

the agent, allowing the agent to recall the state of parts of the world it can no longer observe. The world may have changed since the "memories" were created, but it is at least able to make a qualified guess at the state.

- **Goal-based agent** is a type of agent that knows how the world evolves and how its action affect the world. It can then use search and planning to determine a sequence of actions that will change the state of the environment into the desired one, i.e. the goal state. Additionally, this knowledge about how the world evolves enables the agent to make much more precise approximations of the unobservable state of the world. It is comparable to the deliberate control scheme (section 2.2.3).

Goal-based agent

- **Utility-based agent** is an extension of the goal-based agent. A goal-based agent is either happy or unhappy with the state of the world. Utility-based agents use a **utility function** to map a state (or sequence of states) onto a real number which describes the associated degree of happiness. This also allows for graceful handling of multiple or conflicting goals. The agent tries to maximize the amount of happiness wrt. the utility function.

Utility-based agent

utility function

Because action selection is the responsibility of the agent function, another view on the types of agents is based on the components of the agent function (figure 2.8).

The agent function of the simple reflex agent consists simply of a function, f, mapping between the input (percepts) and output (actions) (figure 2.8(a)). It is a **stateless** type of agent. The model-, goal- and utility-based agents are **stateful** types of agents. Their agent function use the internal state of the agent to create the mapping function (figure 2.8(b)). Finally, there is the learning (or "adaptive") agent. This type of agent has an agent function that can modify the variables of which it itself depends upon. This allows it to "learn" or

stateless

stateful

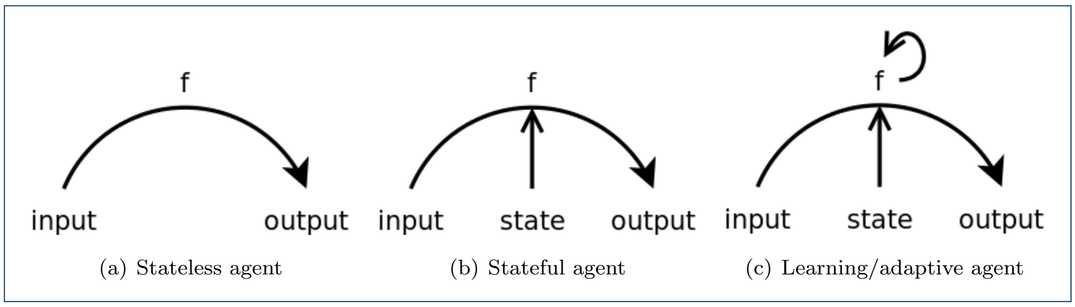


FIGURE 2.8: *Three different types of agents based on their agent function*

"adapt" based on the result of its percepts and actions in the world. This agent function is illustrated in figure 2.8(c) and will be more closely examined in the next section.

2.4.1 Learning agents

learning agent

adaptive agent

All the basic types of agents can be extending to a **learning agent** (also called an **adaptive agent**). A learning agent is a type of agent that is able to change its agent function based on the perceived effects of performing actions in the environment. In effect, this can lead to an agent that is able to learn how to adapt to its task environment. They can thus learn to operate in initially unknown environments and become more competent than their initial knowledge alone would have allowed.

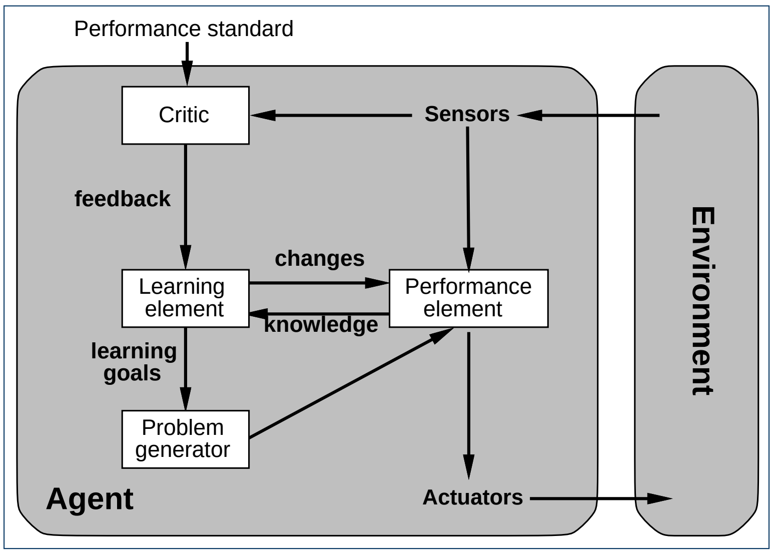


FIGURE 2.9: *The elements of a learning agent, [50, p. 53]*

performance element

A learning agent can be divided into four conceptual elements, as shown in figure 2.9. In the terminology of learning agents, the **performance element** was what was previously thought of as the entire agent, [50, p. 52]. That is, it is the previously static model for taking in percepts and decides on actions. We need some means of manipulating the performance element based on the experiences of the agent. First, however, we need a measure of success to be able to determine when the agent did something desirable and when it did something undesirable. The element responsible for this is called the **critic**. The critic should be thought of as an outside observer that is able to determine the results of the agents actions in the environment. The critic needs to compare the observed results with the **performance standard** (a fixed indicator on success) to determine how well the agent is doing. The critic gives feedback to the **learning element** on how the agent is doing, and the learning element determines how to change the performance element in response. Finally, a learning agent needs a **problem generator** in order to suggest actions that might lead to new and informative experiences, [50, p. 52-53]. Without it, an agent may be stuck in a local maximum of its potential, while the problem generator can help the agent achieve its global maximum potential.

critic

performance standard

learning element

problem generator

For concretizing the concepts, let us consider a simple example of a learning agent. Suppose, for instance, that we need to create an agent to act as a small animal in a desert. We might model this as a simple reflex agent, a model-based, goal-based or utility-based agent (section 2.4). The choice of model is secondary and depends on the animal we would like the agent to represent. Either way, we want to extend the model to a learning agent for handling unforeseen changes in the environment such as drought, decease or the appearance of a new predator. Our original agent model is the performance element of the corresponding learning agent. The performance standard may state that eating, drinking, resting and staying alive are good things that should receive a **reward** while starving, being thirsty and in pain are bad things that should receive a **penalty**, [50, p. 54]. Over time, the critic would provide feedback to the learning agent based on the performance standard. For example, if the agent does not drink enough, it will get penalties and the learning element will change the performance element to make the agent more inclined to drink. Finally, the problem generator could force the agent to explore a larger area, thus allowing it to find lakes large enough to persist throughout a drought.

reward

penalty

In the next section we will discuss how the classical, academic, artificial intelligence differs from the artificial intelligence used in games.

2.5 Classical AI and game AI

Although the term "artificial intelligence"[8] was not coined until 1956 by John McCarthy it is a field that has been studied – in one form or another – long before that, [50, p. 17]. The historical background for classical, or academic, AI is vast. It spans several millennia and research in a multitude of different fields of study, including philosophy, economics, biology, mathematics, linguistics, computer science and more, [50, p. 5-16]. Each field has contributed to the understanding of the field of AI. The field has been through patches of tremendous growth as well as grave depressions.

[8]Russell and Norvig suggests that "computational rationality" might have been better suited, [50, p. 17]

McCarthy defined artificial intelligence as ”the science and engineering of making intelligent machines“, [56]. Since the dawn of AI, it has built on the idea of duplicating human features like creativity, self-improvement, and use of language, [50, p. 18]. Computers are much more adept than humans at handling operations such as arithmetic, searching and sorting, while much worse at things such as face and speech recognition and natural languages, things which come naturally for humans, [42, p. 4]. AI has been applied to many of such difficult problems in the hopes of gaining a better understanding of the nature of thought and the mechanics of the human brain, [42, p. 4]. Take the problem in figure 2.10 for instance. This could be a typical question in an IQ-test and something most people would be expected to solve quite easily. Programming a computer to solve this problem is significantly harder. In 1968, Tom Evans successfully created a program, called Analogy, that used artificial intelligence for solving problems like the one in figure 2.10, [50, p. 19].

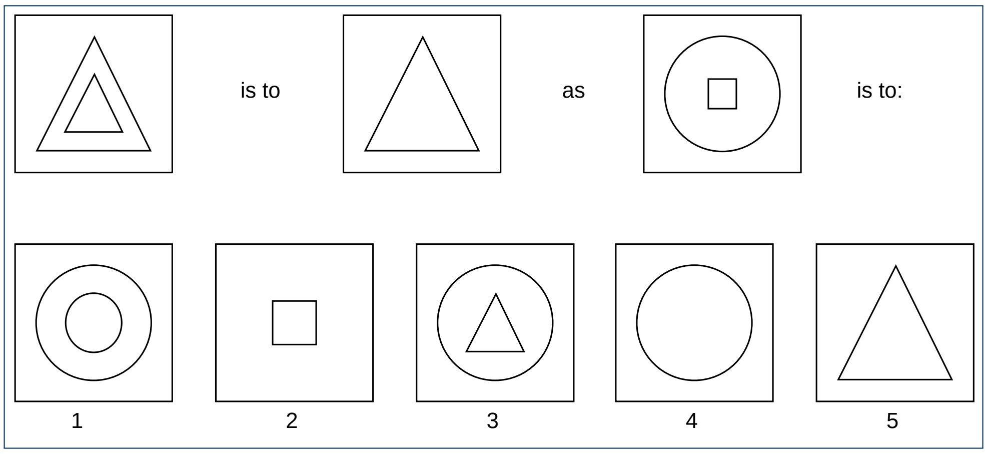


FIGURE 2.10: *Example of a problem solved using AI, [50, p. 20]*

In the artificial intelligence used in games, the focus is also on ”the science and engineering of making intelligent machines“, though in this context the ”machines“ are virtual agents. Game developers are also attempting to create agents who duplicate human features like autonomy, creativity and self-improvement. However, very different means are used for trying to achieve these features in the classical AI and in game AI. Game AI typically draws upon existing methods from the field of artificial intelligence, but do so in order to produce the illusion of intelligence in the behavior of the agents, [59]. That is, the main goal for game AI is not to create intelligent agents, but to create agents which are perceived as intelligent by the player:

> “*As long as the player has the illusion that a computer-controlled character is doing something intelligent, it doesn't matter what AI (if any) was actually implemented to achieve this illusion, [39, p. 41]*”

Artificial intelligence has been an integrated part of computer games for as long as they have existed. Some of the very first computer games developed were Chess (in 1951) and Checkers (in 1952) and they were the first applications of AI in games, [28], [59]. Their goal of the player controlled by the AI was to solve the problem of choosing the best possible move in a given situation, as was the goal of the opponent. These applications of AI in games are actually more in the field of AI than in game AI. The primary reason hereof is that the AI tries to win the game. The ultimate AI here, would be one that was able to win the game each time, if possible.

The ultimate goal for a game AI is to provide entertaining gameplay. In a game of Chess, in most cases it would not be considered entertaining gameplay to lose against a superior player AI. However, most people will find it entertaining to be challenged by an AI of equal skill and winning by outsmarting it. In other words, as opposed to many instances of the classical AI, the goal in game AI is not to find optimal solutions to a given set of problems.

In the next section we will look at the concrete responsibilities and applications of game AI.

2.6 Artificial intelligence in games

Since the dawn of computer and video games, they have included game artificial intelligence in one form or another. The ghosts agents of Pac-Man are among the earliest examples of game AI that used character-based behaviors to produce the illusion of intelligence, [59, sec. "History"], [42, p. 19]. Although many concepts and methods in game AI are drawn from the fields of AI and robotics, the approaches used to achieve the illusion of intelligence often lies well outside these fields. In game AI the means for creating this illusion is secondary. The primary concern is the illusion itself:

> “*Computer games are an application in which the perception of intentionality is often more important than intentionality itself. Players often attribute more intelligence to non-player characters than is actually warranted. [38, p. 46]*”

2.6.1 Responsibilities of game AI

In games, the AI is usually responsible for guiding the behavior of the game agents. Behavior is a broad term, however. In particular, the game AI in most modern games handles three overall problem domains; movement, **decision making** and strategic reasoning, [42, sec. 1.2]. Movement is responsible for pathfinding in game levels (section 2.7.4) and providing tools for believable locomotion, or **steering**, to move characters. Decision making is the process of deciding on the next course of action, either as a reaction to stimuli or for achieving longer-term goals. Strategic reason is used for creating elaborate plans for succeeding in complex scenarios.

decision making

steering

Our focus on game AI is in particular on the decision making process. It is entirely possible to have games in which neither movement nor strategic reasoning is required. The game might have reactive agents, that is, simple reflex agents (section 2.4), who are invisible, stationary or move in a trivial fashion (e.g. ghosts moving through walls). Decision making is required for any agent that needs to decide on an action to perform given a percept sequences. In other words, decision making is the process of creating a suitable agent function (section 2.2.1).

The decision making process is especially important in character-based AI in which the agents have a physical manifestation in the game. Strategic reasoning is applicable when game scenarios are so complex or highly sequential that tactical planning is needed for achieving the goals.

2.6.2 Views on game AI

For game AI we have three interested parties; the game AI programmer, the game designer and the player. Each of these roles has a different view on the game AI.

Initially in the process of developing the game AI, the interested party is the game AI programmer. His concern lies with the infrastructure and the technical details of the AI. His measure for a good game AI is one that provides a simple, effective and modular interface for communicating with the game world and the editor tool used to create the behavior.

Secondly, we have the game designer whose responsibility is to create an interesting, believable and intelligent behavior on top of the technical platform. The designers measure for a good game AI is one that is highly autonomous, while providing full directability whenever necessary for adding story-driven or custom behavior to the game. Further, this behavior should be easy and intuitive to create with the game AI editor.

Finally, we have the end user – the player – whose measure of success is neither the technical underpinnings or the level of autonomy but simply that the game AI adds entertainment value to the game.

2.6.3 Processing game AI

frames

frame rate

Games are typically processed in **frames**, where each frame represent a discreet (often static) slice of time, Δt. Many games have a **frame rate** at about 30 frames per second[9] (FPS), giving each frame a time slice of $\Delta t = \frac{1}{30}$ seconds $\approx$ 30 milliseconds. During a frame, the processing loop need to poll for input, recalculate physics, game AI, geometry, lighting, etc. and render the updated visual output to the screen.

Most game AI do not need to be updated every 30th milliseconds, however, so many games choose only to process the game AI every n'th frame. Another often used approach

[9]However, the frame rate of games vary greatly. Some games (typically console games) use a fixed rate between 30 and 60 FPS. Other games use a varying frame rate that may reach several hundred FPS on sufficient hardware. Frame rates below 30 are generally frowned upon as it often gives the appearance of lag, i.e. unsmooth transitions in the game, [58]

for reducing the processing requirements of the game AI is to calculate the AI using a **level-of-detail** (LOD) technique, [46, p. 7], [32, p. 294]. This can be used to scale down the detail of the AI of agents which are not currently interesting to the player. That is, the AI can be scaled down "if the player won't notice the difference", [32, p. 294]. Using this approach, agents which are far away or completely out of sight may use only the simplest form of AI and animation, if any.

level-of-detail

Figure 2.11 presents one possible architecture for processing one frame of game AI. First, for the AI to be computable, it needs information (i.e. percepts) from the world such that it can decide on actions. As mentioned in the previous section, game AI has three major areas of responsibility: strategy, decision making and movement. These need to be handled in AI processing, as illustrated in the figure.

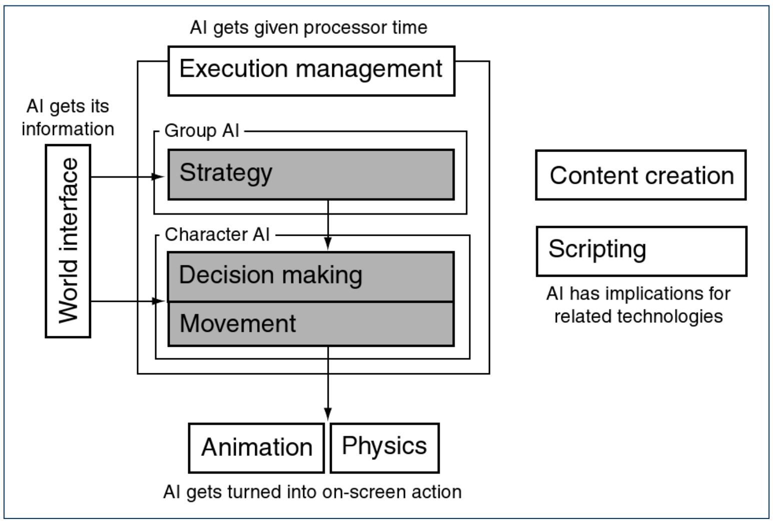


Figure 2.11: *Model of how game AI may be structured in a game engine, [42, p. 9]*

The game AI needs to plan a strategy for squad-based agents (if any), to have each agent decide what to do next and to move those agents around if needed. At each step of this processing, data from outside the AI module and the world representation may be needed. This is often scripted behavior (section 2.7.5) or other types of resources from content creation tools. Having the majority of content available as external resources is known as **data-driven** development and is a requirement for modern game development [42, p. 361, 366-367]. Data-driven development eases the workflow between the game AI programmer, who implement the AI processing, and the game designer, who create the custom behavior that the game AI relies on. Following the processing of the game AI, the actions chosen and executed need to be reflected as behavior in the game. This will often (as shown in figure 2.11) involve the animation or physics systems.

data-driven

In section 2.7 and chapter 4 we will examine game AI from the point-of-view of the game AI programmer and the game designer. The next section will highlight typical examples of character-based AI in games and look at the techniques they have used. The following section will focus on a different type of game AI called tactical AI. These two sections will provide an insight into the diversity of game AI.

2.6.4 Character-based game AI

character-based AI

Most game genres use the concept of a **character-based AI** agent, meaning that the agents are represented as (some form of) characters within the game. That is, the agents actually have a physical manifestation in the game as opposed to purely virtual agents, as we will see in the next section. Having a manifestation of the agents in the game means that they can be directly observed and special care is needed for their behavior to be perceived as intelligent.

first-person shooters

role-playing games

Half-Life

Character-based agents appear in most action games as well as many first and third person games. Examples include genres such as **first-person shooters** (FPS), **role-playing games** (RPG), racing, platform, sports and fighting games. A good example of a game utilizing character-based AI is **Half-Life** [Valve, 1998]. Half-Life is a popular FPS action game, [52]. It was highly acclaimed for the realistic behavior of the ally and enemy agents but especially that of enemy squads, [30, p. 212], [38, p. 46], [72, p. 42].

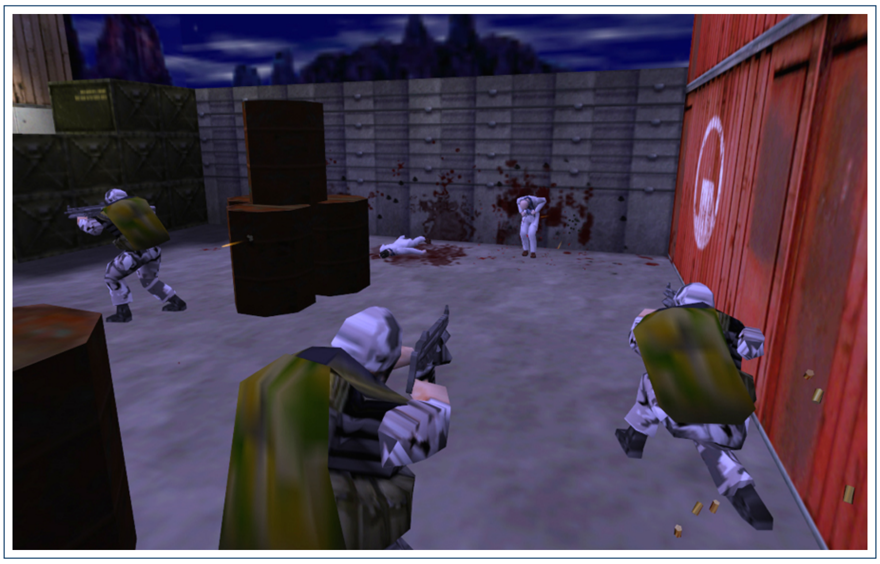


FIGURE 2.12: *Marines using squad-based behavior in* Half-Life, *[44]*

The agents' sensory system allows them to hear and see nearby agents. They have an emotional relation (e.g. fear, dislike or hate) to each type of agent to help determining how to act when confronted with a certain type of agent.

The squads in Half-Life consist of marines with military weapons (figure 2.12) who communicate using walkie-talkies. In combat they utilize a "*Kung-Fu*"-style tactic in which only a few squad members are actively attacking at the same time. Squad members not attacking can be covering, reloading etc.. The enemies engaging in this behavior were perceived as being intelligent as they sought cover when they needed to reload and were replaced by another who had been covering. This effect were amplified by the marines explicitly proclaiming their intentions. When an enemy needed to reload he would yell "*Reloading*" and in confirmation hereof the enemy replacing him would yell "*Cover me*", signifying that he would assume an attacking position and needed cover fire, [38, p. 46].

The actual implementation of this behavior (section 2.7.2 on page 40) is much simpler than the perceived behavior would suggest. This is a good example of **emergent behavior** (2.7.1), where a number of simple behaviors interact in a manner that appears more complex to an observer.

emergent behavior

Figure 2.13: *In-game screenshot of* Halo 2, *[63]*

Halo 2

Another highly popular FPS game is **Halo 2** [Bungie Studios, 2004] (figure 2.13). The enemies in Halo 2 are modeled using a model-based reflex agent type (section 2.4) for achieving a better illusion of human-like intelligence. The task environment of Halo 2 has been made partially observable (section 2.3) for the agents and they continuously maintain a model of the world, [37]. This allows players to outsmart the enemies by e.g. leading them to believe that the player is hiding behind a rock when in fact he had crept away to attack the enemy from behind. The end result was a more realistic appearance of the enemy as stated in the following quote from a Halo 2 AI developer.

> “*Having this state information* [...] *distinct from the actual world-state and gated by the actor's perception filter (an actor* [i.e. agent] *should not be able to see through walls, for example) allows the two representations to occasionally diverge - thus the actor can believe things that are not true, and we now enter the realm of AI that can be tricked, confused, surprised, disappointed, etc.* [...] *Giving AI an internal mental image of its world results in interesting and more "realistic" behavior, [37]* ”

Having agents keep their own model of the world allows them to correlate it with e.g. their perception sequence (section 2.2). This could be used to answer questions such as "have I already searched for enemy X at location Y?". Although the knowledge model in Halo 2 is rudimentary, the game AI developers suggest that adding spatial, structural and time relations to the model could remove many limitations, [37].

The Sims

The Sims [Maxis, 2000] is a life-simulation game series that has achieved a tremendous amount of success[10]. The gameplay of the The Sims revolve around structuring the daily life of virtual characters, the so-called "Sims". The characters have needs and desires of their own, such as the need for food, sleep and personal hygiene. The agents in The Sims are comparable to utility-based agents (section 2.4) in that they attempt to fulfill their needs and, if satisfied, attempt to satisfy their desires.

FIGURE 2.14: *Collage of different agent behaviors in* The Sims 3, *[26]*

The game has no winning condition and has been described as more of a toy than a game, [69]. There is very strong emphasis on the agent-based game AI and that the Sims behave in a life-like manner. It is in effect a demonstration of advanced goal driven game AI. If left alone, the Sims will act on their own, taking care of basic needs such as eating when hungry and socializing when lonely. It requires the intervention of the user only to ensure that the Sim acts upon things that are not directly related to its needs such as cleaning, going to work or arranging a party.

smart terrain

smart objects

The Sims use a novel approach to handling complexity in a game with a vast number of different objects that the characters can interact with. The technique is called **smart terrain** (also known as **smart objects**) and works by "embedding intelligence" into inanimate objects. This often takes the form of directions on how to interact with that particular object as well as what it provides and what it needs. Each object broadcasts ("advertises") which services it provides to the vicinity. Thus a refrigerator might advertise the message "*I contain food*". The food inside will advertise that it provides hunger satisfaction. Should

[10]The series has sold more that 125 million copies worldwide thus making it the best selling game franchise of all time, [8]

the agent choose to use the food, it will instruct him that "*I need cooking*" and the oven advertises that "*I cook food*", [71], [47], [15].

The smart terrain approach has several advantages. First, it allows the complexity to be distributed across the environment rather than having the agent itself handle each type of possible interaction. This helped to greatly reduce the complexity of the development of the numerous expansion packs to The Sims series. The reason is that new objects could be added painlessly, without the agents knowing about them, because the objects themselves instructed the agents on what they provide and the protocols for interaction. Adding a new object, e.g. a freezer, would thus only be a matter of having it advertise "*I contain food*" and agents would be able to interact with it. Secondly, it allows the agents to interact with the environment in a more realistic manner. A soldier might take advantage of a nearby crate advertising "*I offer cover*", [71].

2.6.5 Tactical game AI

As opposed to the physical manifestations of character-based agents, we have a type of virtual agents often categorized as strategic or **tactical AI**. In this type of game AI, the focus is on analyzing game scenarios and executing strategies that achieve goals or subgoals, [30, p. 219].

tactical AI

There are two different views on tactical AI. One view is that it is used to provide guidance for groups or squads of agents who have their own decision making and behavior, [42, p. 10]. An example hereof is **group dynamics**, e.g. **Boids** (section 2.7.1), in which a group of agents can be moved in formation with each agent handling its individual movement and collision detection. Another view is that tactical AI is used for computing a sequence of operations that achieve a goal, and that the agents serve merely as pawns for achieving that goal, [30, p. 219-220]. An example hereof is Chess. The pieces may be thought of as agents, but it is the (invisible) AI player who plans the sequence of operations that results in a winning state.

group dynamics

Boids

Both views take a different approach to game AI than the character-based AI, although the first approach may be thought of as an operational layer on top of character-based AI. The latter is better suited for solving problems such as managing the cooperation of many different entities with different properties, strengths and weaknesses such as soldiers on a battlefield. For this reason, tactical AI is often used in **real-time strategy** (RTS) games, [30, p. 244].

real-time strategy

An example (figure 2.15 on the following page) of a popular RTS game relying on tactical AI is **Age of Empires II: The Age of Kings** [Ensemble Studios, 1999], [30, p. 245]. This game uses a two-layered AI system. The lower layer handles the AI of the individual units which the player controls. The behavior of each unit agent is defined by a simple rule system (section 2.7.3) or state machine (section 2.7.2), [30, p. 245]. The higher layer represents the tactical reasoning where the agents are used as the means for overcoming the enemies. One common technique is called **command hierarchy** and consists typically of three levels of decisions; overall strategy, squad tactics and individual combat, [46, p. 4].

Age of Empires II: The Age of Kings

command hierarchy

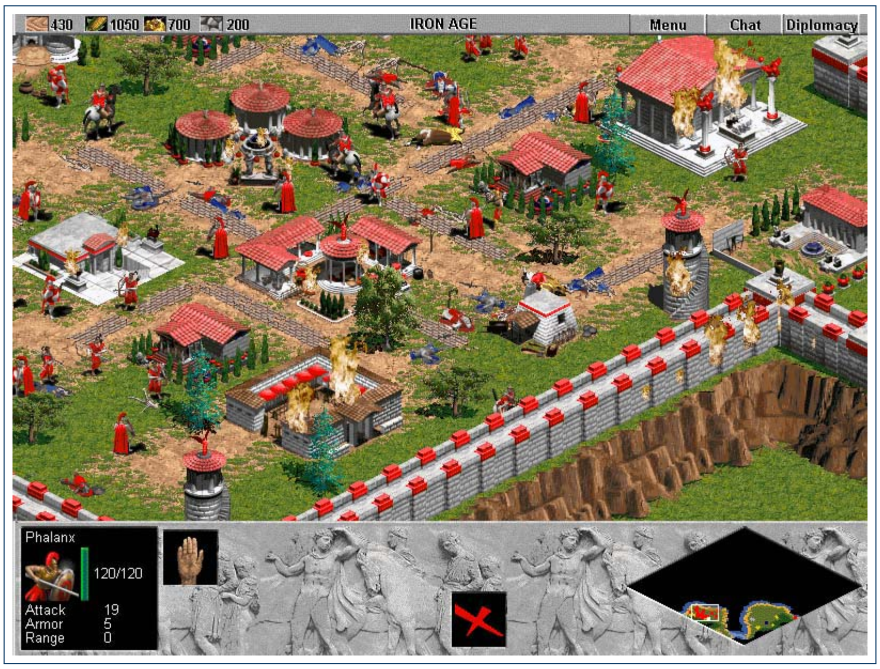


FIGURE 2.15: *Screenshot from* Age of Empires II: The Age of Kings, *[55]*

> *The AI in the* [Age of Empires] *game relies on tactics and strategies to win, instead of "cheating" by giving bonus resources to itself, or tweaking its units to be stronger than normal, [55]*

The next section goes into detail with some of the most common and also some newer, promising, techniques used for implementing game AI, [46], [47]. For each of the techniques, we will discuss its applications in game AI as well as its strengths and weaknesses.

2.7 Techniques for game AI

In this section we will examine many of the common and popular techniques for implementing and handling game AI. The techniques we will discuss are those primarily associated with character-based AI (section 2.6.4), although some of the approaches are also used for other types of game AI, e.g. tactical AI (section 2.6.5).

We discuss and exemplify each technique in the context of games, either with fictional examples or with concrete examples from commercial games. We also discuss the strengths and weaknesses of each technique and compare them with other related techniques.

We begin by looking at emergent behavior and how this can be used for e.g. simulating realistic flocking behavior. Secondly, we present a thorough introduction to finite state machines, their inner workings and their (ubiquitous) applications in game AI. Finite state machines were among the very first techniques for creating character-based game AI and has since provided the foundation for many other techniques. Following this discussion, we will examine rule-based systems, a technique that has some very different qualities and applications than finite-state machines. They are often used to create prioritized global behavior rather than focusing on the sequentiality of character-based AI. We then turn to the domain of problem solving, in which we look at techniques for state-space searching, pathfinding and planning. The concept of scripting and the usage of scripts in both game AI and game development is then considered. Finally, we will introduce and discuss the behavior tree technique for implementing and designing game AI. It is a relatively novel approach that combines the strengths of many other techniques to provide a general solution to many of the problems faced in creating character-based AI. We will go into much greater detail about this technique in chapter 3.

2.7.1 Emergent behavior

Emergent behavior is a popular technique in which several simpler behaviors interact to form complex behavior, [46, p. 5]. It is often used in game AI and game animation to simulate the appearance of intelligent agents with fluid movements. The use of emergent behavior can be a great tool for the development of game AI as it allows each behavior to be simple and modular, but the resulting behavior to be complex and interesting. Emergent behavior originated in the field of robotics, in which it is a well-known concept. The following quote is by artificial intelligence and robotics pioneer Rodney A. Brooks:

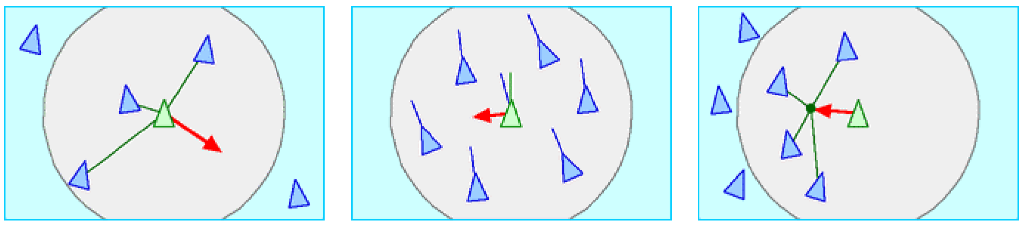
> “*Complex (and useful) behavior need not necessarily be a product of an extremely complex control system. Rather, complex behavior may simply be a reflection of a complex environment. It may be an observer that ascribes complexity to an organism — not necessarily the designer, [11, p. 3]*”

The ability to achieve the illusion of complex behavior from a number of simple behaviors is very desirable. A concrete example of emergent behavior is found in Craig W. Reynolds' influential **Boids** algorithm, [49]. The algorithm is used to simulate the flocking behavior of flocks of birds, swarms of insects, schools of fish, etc., [46, p. 5]. The algorithm uses three simple rules for steering behavior in order to create realistic movement and behavior of the group as a whole. The rules are shown in figure 2.16 on the next page.

Boids

The three rules create the basic behavior of each Boid, that is, each member of the flock. The code needed for combining and applying the rules to create the resulting flocking behavior is easily implementable. The pseudo code for computing the Boids' velocities and positions in $\mathbb{R}^n$ is shown in algorithm 1 on the following page.

The code creates a new velocity and position for each Boid, b, based on that Boids current velocity and position along with the combined result of all rules. Each rule takes as input a Boid and returns a computed velocity vector. In the Boids algorithm, the rules are `separation`, `alignment` and `cohesion`. Other rules may include handling wind, making the



(a) Separation: Avoid crowding (b) Alignment: Align heading (c) Cohesion: Group together

FIGURE 2.16: *The three steering rules for flock behavior in the boids algorithm, [48]*

Algorithm 1 Pseudo code for combining and applying the rules of the Boids algorithm

```
1: for each Boid $b$ do
2:     $v \leftarrow NullVector$                          ▷ NullVector = vector(0, ..., 0)
3:     for $r := 0$ to $size[rules]$ do
4:         $v \leftarrow v + modifier_r \cdot rule_r(b)$
5:     end for
6:     $b.velocity \leftarrow b.velocity + v$
7:     $b.position \leftarrow b.position + b.velocity \cdot \Delta t$      ▷ $\Delta t$ = time interval
8: end for
```

Boids inclined to move towards, or away from, a particular position, limiting the velocity and placing bounds on the positions of the Boids.

Emergent behavior is a very powerful tool for combining behaviors into seemingly more complex and more intelligent behavior. However, it is difficult to use emergent behavior to simulate sequential behavior (section 2.2.3). The rules of emergent behavior is usually formulated as a continuous function depending on some parameters (such as in the Boids algorithm) and is much better suited for reactive control. Standard agent behavior typically depend on a number of discrete states, of which only a few are active at any time. Behavior such as that of the ghosts in Pac-Man depend on an internal state, in this case “Chase” or “Frightened” (section 2.7.2 on page 35). Emergent behaviors are not suitable for modeling such behavior. Historically the approach of choice for handling discrete state behavior is to use finite state machines. In the next section we will look more closely at this highly popular technique, its uses and its advantages and disadvantages.

2.7.2 Finite state machines

state machine

A **state machine** is a well-defined mathematical abstraction. It has been used as a way of formalizing computer programs, regular expressions and many other discreet processes. Apart from its strictly mathematical applications, it has also proved a central cornerstone technique in game AI programming, [46, p. 10], [38, p. 47]. In the terminology of game AI, any type of state machine is usually called a **finite state machine** (FSM), [42, p. 310].

finite state machine

Finite-state machines (FSMs) are without a doubt the most commonly used technology in game AI programming today, [32, p. 283]

The FSMs used in game AI programming have a relaxed definition compared to the strict mathematical formalism. Let us first look at the formal definition. A FSM (also known as **finite automaton**, abbreviated FA) is an quintuple $(Q, \Sigma, q_0, A, \delta)$, [41, p. 95], where

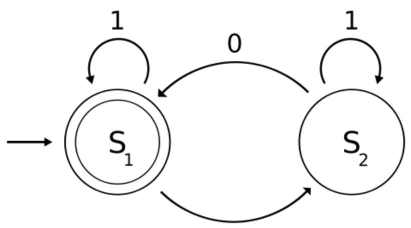
finite automaton

- Q is a finite, non-empty, set of states
- Σ is a finite alphabet of input symbols
- $q_0 \in Q$ is the initial state
- $A \subseteq Q$ is the set of accepting states
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function

As an example finite state machine, let us take the FSM accepting the language of strings with an even number of 0's, i.e. the language specified by the regular expression $(1^*01^*01^*)^*$. Let the FSM $M = (Q, \Sigma, q_0, A, \delta)$ where $Q = \{S_1, S_2\}, \Sigma = \{0, 1\}, A = \{S_1\}$, and δ is given by table 2.17(b). The **transition table** specifies the transition between states based on the input symbols, i.e. 0 or 1. M can be represented using a **state transition diagram** as shown in figure 2.17(a).

transition table

state transition diagram



(a) State transition diagram, [57]

q	0	1
S_1	S_2	S_1
S_2	S_1	S_2

(b) State transition table

Figure 2.17: *Diagram representation for a FSM accepting the language of strings with an even number of 0's*

FSMs in games

In general terms, a FSM defines a number of states and a notion of when it is possible to transition from a given state to a given state. What makes it applicable to game AI is the clear distinction between states and transitions. In game AI, each possible internal state of an agent can be represented as a state in the FSM. The transitions between the states then symbolizes the condition(s) causing the agent to change its internal state, and thereby also its behavior. It can be thought of as a decision-action model in which the transitions are decisions causing a state change and the states represent the actions the agent perform in a given state:

> *This decision-action model is straight-forward enough to appeal to non-programmers, such as level designers, on the game development team,* [32, p. 284]

The FSM is not used for evaluating the input symbols but primarily as a way of directing the transition between the states of an agent. Therefore, there is no accepting states per se. A simple example FSM is shown in figure 2.18. In this FSM we have three states as represented by the rounded rectangles. They corresponds directly to the internal state of the agent. The directed edges symbolize the transitions between the connected states, with the edge labels describing the condition required for the transition to be possible. There can be several outgoing transitions from a state, as e.g. can be seen in the "Fire Weapon" state in the diagram. It is not apparent which transition will be utilized if both are applicable. In such cases, the concrete implementation will provide a prioritization of transitions. Listing 2.1 below show an example implementation of the FSM in question.

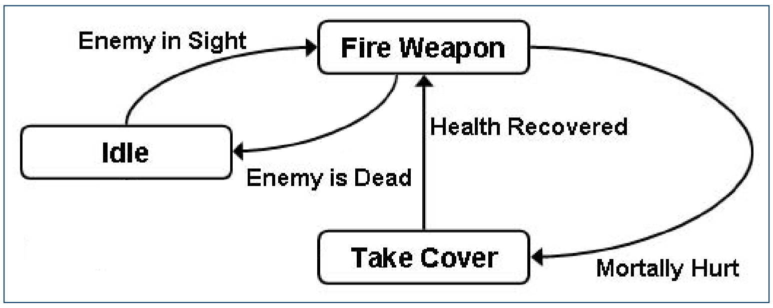


FIGURE 2.18: *A typical game FSM diagram of an enemy agent, corrected version of erroneous diagram from* [72, p. 44]

```
1 STATE(Idle)        IF(EnemyInSight)     THEN(FireWeapon)
2 STATE(FireWeapon)  IF(MortallyHurt)     THEN(TakeCover)
3 STATE(TakeCover)   IF(HealthRecovered)  THEN(FireWeapon)
4 STATE(FireWeapon)  IF(EnemyIsDead)      THEN(Idle)
```

LISTING 2.1: *Example implementation of the example FSM,* [72, p. 44]

`STATE`, `IF` and `THEN` are macros that are expanded at compile time. They would typically be expanded into a control structure of either `switch` or `if-then-else` statements (listing 2.2 on the next page). The priority of execution thus becomes top-down. This explains which transition is chosen if multiple are applicable. In this case of the previous discussion, "Take Cover" (line 2) would be chosen over "Idle" (line 4) as the transition state from "Fire Weapon".

Listing 2.2 shows the typical code structure for handling a single state of a FSM.

```
case [NAME OF STATE]:
  [DEFAULT ACTIONS]
  [CONDITION EVALUATION]
  if ([TRANSITION]) {
    state = [DESTINATION STATE];
    break;
  }
  (...)
```

Listing 2.2: *Typical structure of a switch case for a state in a FSM, [30, p. 159]*

In this case, the entire FSM is represented by a `switch` control structure switching on the `state` variable and each `case` represents a state. In each frame of game AI execution, the currently active state (`[NAME OF STATE]`, e.g. `Idle`) is executed. Initially, the behavior associated with the state (`[DEFAULT ACTIONS]`, e.g. `Patrol()`) is performed. Next we need to determine which state to transition to, if any. Any computations needed for evaluating the transition conditions are performed (`[CONDITION EVALUATION]`, e.g. `QuerySensorySystem()`). Each transition is then handled in turn (`[TRANSITION]`, e.g. `EnemyInSight()`) and if it evaluates to `true`, its corresponding destination state (`[DESTINATION STATE]`, e.g. `FireWeapon`) is set as the new active state (`state`). Listing 2.3 shows how a concrete implementation of the example state might appear.

```
case Idle:
  Patrol();
  QuerySensorySystem();
  if (EnemyInSight()) {
    state = FireWeapon;
    break;
  }
  (...)
```

Listing 2.3: *Example implementation of a state-handling switch case*

Using FSMs for programming game AI has numerous advantages. They have been claimed to be quick and simple to code, easy to debug, have little computational overhead, are intuitive and flexible, [13, ch. 2]. These might be some of the reasons why FSMs have been used in game AI programming since the dawn of video games. In fact, even the ghosts of Pac-Man used FSMs to control their behavior. Let us examine the simple FSM from Pac-Man to get an intuition for the usage of FSMs in game AI programming along with its advantages and disadvantages.

The ghosts of Pac-Man have two internal states; "Chase" and "Frightened" (also called "Evade"), [13, ch. 2], [42, p. 7]. In the "Chase" state, the ghosts try to hunt down Pac-Man, while they in the "Frightened" state attempt to flee from him. The reason for wanting to flee is that Pac-Man can consume a so-called "power pill", enabling him to eat the ghosts. The state transition table representing the FSM of the internal state of the ghosts is shown in table 2.19 on the next page, and the resulting FSM state diagram is illustrated in figure 2.20 on the following page.

Current state	**Condition**	**State transition**
Chase	Pac-Man eats power pill	Frightened
Frightened	Power pill effect wears off	Chase

FIGURE 2.19: *State transition table for the FSM controlling the ghosts in Pac-Man*

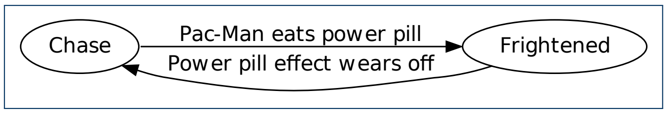


FIGURE 2.20: *State transition diagram of the behavior of the ghosts in Pac-Man*

entry actions

input actions

Moore machines

Mealy machines

The FSM in 2.20 is rather self-explanatory, with simplicity being one of the key advantages of using FSMs. Typical game FSMs use **entry actions**[11] for switching the internal state of the agent and **input actions**[12] for executing the behavior associated with current state. The FSMs used in games are thus **Moore machines** where actions reside in states as opposed to **Mealy machines** where actions are performed on transitions, [32, p. 286].

Adding states and transitions

The FSM used to specify the behavior of the ghosts in Pac-Man is very simple. Let us try to extend the FSM and see how it is affected.

As it turns out, further investigations into the inner workings of the AI in Pac-Man reveal that the ghosts have an additional state called "Scatter", [45, sec. "Modus Operandi"]. When the ghosts are in this state, they move toward their respective corner of the maze. Meanwhile they completely disregard Pac-Man, thus giving the player a brief break from being chased. The scatter state is entered at predefined intervals during each level. However, it may only be entered four times per level or life. The revised state transition table for the ghosts is shown in table 2.21 and the corresponding graph is illustrated in 2.22 on the facing page.

Current state	**Condition**	**State transition**
Chase	Pac-Man eats power pill	Frightened
Chase	Scatter interval reached and scatter counter < 4	Scatter
Scatter	Pac-Man eats power pill	Frightened
Scatter	Scatter timer expired	Chase
Frightened	Power pill effect wears off	Chase

FIGURE 2.21: *Extended state transition table allowing the ghosts in Pac-Man to "scatter"*

[11] Actions performed upon entering a new state

[12] Actions performed continuously in the active states

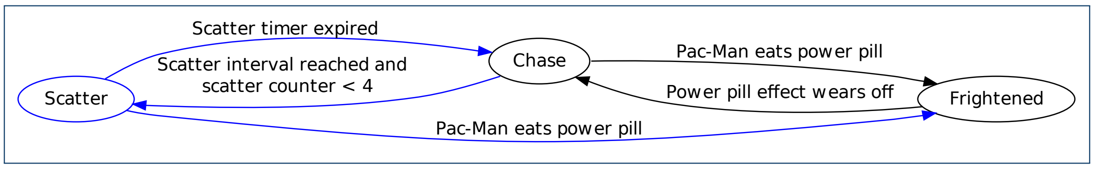


Figure 2.22: *The FSM of the ghosts in Pac-Man, extended with the "Scatter" state (blue)*

As can be seen from the state diagram in figure 2.22, a new state may have in- and outgoing edges to/from all other states. That is, a new state may add up to $2n$ (i.e. $O(n)$) new edges to the graph, where n is the number of states in the graph. Our FSM is rather simple, but this example tells us that if the states are largely interconnected we need to add and maintain many transitions for each added state. In such cases, it may be worth considering whether another approach would be more suitable instead, e.g. a rule-based system (section 2.7.3).

One drawback to the simplicity of FSMs is that it can be difficult to create sequential behavior (section 2.2.3). On the other hand, FSMs are well suited at handling reactive control schemes[13]. In addition to the number of edges increasing greatly in some situations, in some cases multiple states needs to be added to the FSM in order to add a single state that depends on its context. The number of states in the FSM may thus also increase greater than linearly with the number unique states required. Such situations occur when we have states that depend on their context in the graph, i.e. the path in the graph. The next section will show that context dependent states occur quite frequently in game FSMs and that they pose a big practical problem.

Problems with complexity

Suppose we add a new context dependent state to the FSM. Because the state depends upon its context, we need to consider the sequence of states before it.

Returning to the Pac-Man example, suppose we need to add a new state "Exclaim" to the ghosts. The state would cause them to make a textual or auditory statement on the basis of their state. That could be "*I'm coming to get you!*" if the ghost is in the chase state and "*Flee!*" in the frightened state. We leave out the "Scatter" state for simplicity. To be able to achieve the desired behavior, we would need two additional states in the state diagram to be able to determine whether the transition originated from the chase or the frightened state. The resulting diagram is shown in figure 2.23 on the next page.

For each new state we want to add, we need to add as many as $O(n)$ additional states in the FSM, where n is the number of states, to be able to represent the different contexts. As previously noted, each of these states also add $O(n)$ edges. Therefore, for each added node,

[13]Unless the FSM is very sequential, in which case the states need to be highly interconnected. A solution for avoiding this problem could be to use parallel FSMs (section 2.7.2)

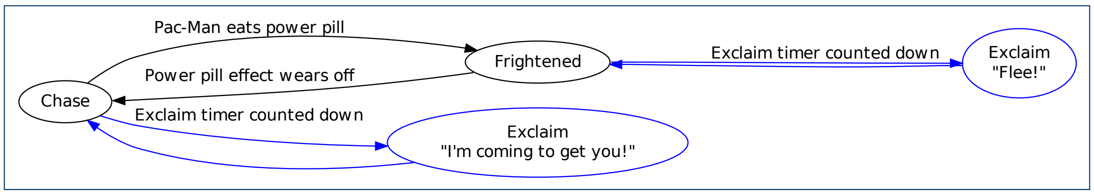

Pac-Man eats power pill
Frightened
Exclaim timer counted down
Exclaim "Flee!"
Power pill effect wears off
Chase
Exclaim timer counted down
Exclaim "I'm coming to get you!"

FIGURE 2.23: *Multiple states (blue) needed in order to insert a context dependent state*

we get an asymptotic doubling in the size of the graph in both nodes and edges. This is sometimes referred to as the "n^2 problem", [35, sec. "Behavior Tree AI"]:

> *Sometimes, especially when modeling complex behaviors, a classic FSM will begin to grow quickly, becoming cluttered and unmanageable. Even worse, we will sometimes need to add some additional behavior and will discover how our FSM's size almost doubles in each step. [30, p. 161]*

In practice, of course, each state will not depend on all other states, but this analysis clearly illustrates the scalability problems of FSMs, [42, p. 318]. Figure 2.24 on the facing page illustrates the problem for a concrete game agent. In 2.24(a) we have the basic locomotive FSM for an agent. The FSM enables the agent to patrol along waypoints (see 4.2 on page 85) and – if spotting the player – to chase the player until it is within shooting distance. We now want to add states to enable the agent to shoot and reload. Figure 2.24(b) shows the FSM after adding the additional states.

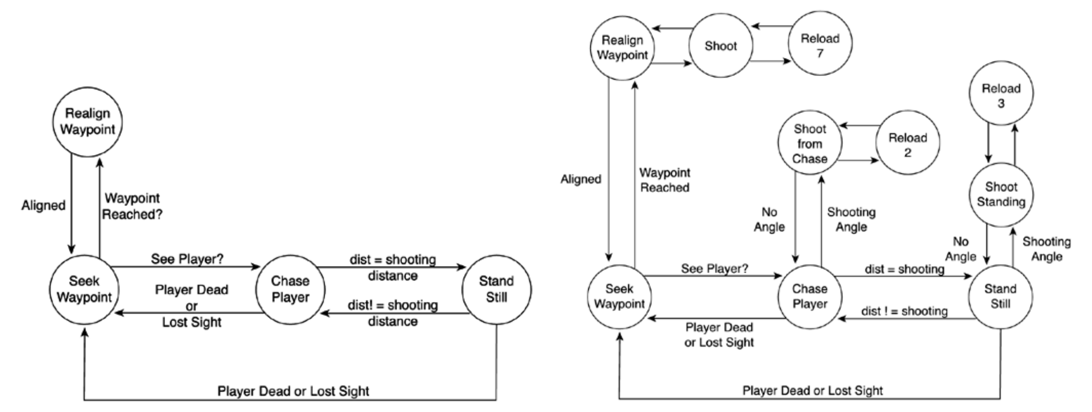
The FSM has clearly grown much more complicated as the shooting behavior needs to take into account the different contexts from which it is invoked. In the next section we will look at extending the concept of a FSM for overcoming the outlined limitations.

Extensions

These complexity problems are difficult to handle because we attempt to obey the definition of the FSM[14]. The means to overcoming the limitations of the FSM lie in further relaxing its definition and extending it with functionality useful to modeling game AI.

> *We argue that this* [FSMs being criticized as being unwieldy] *is not a consequence of the finite-state machine approach per se, but rather is a result of the ad hoc AI development model utilized by*

[14]In fact, the definition we have for the "game FSM" is already highly relaxed compared to its mathematical counterpart



(a) FSM for patrolling and chasing

(b) FSM for patrolling and chasing extended with states for shooting and reloading

Figure 2.24: *Increasing complexity when adding new states to a FSM. In (b) we add shooting and reloading functionality to (a), [30, p. 162, 163]*

most game companies. [38, p. 47] ”

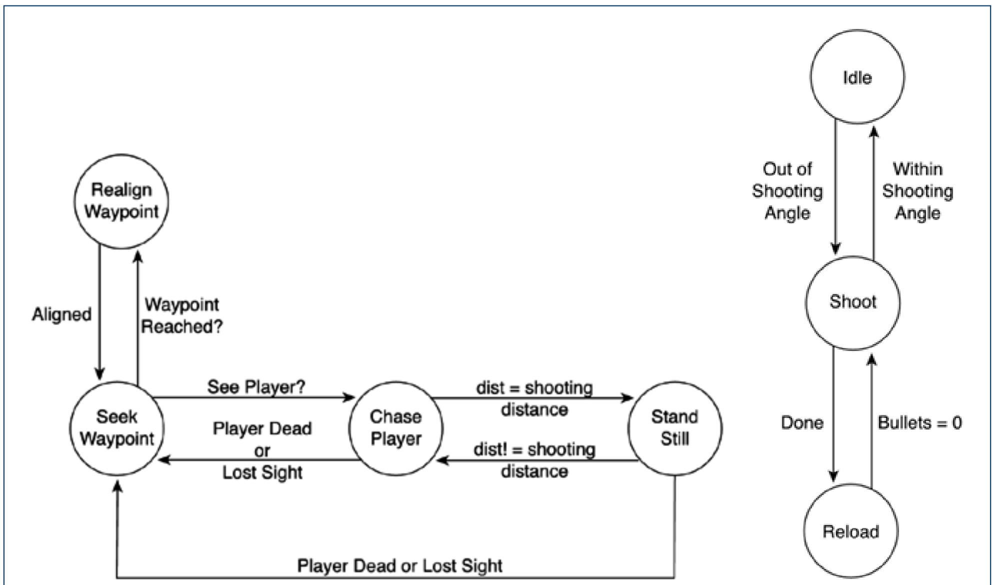
When faced with problems that ordinary game FSMs had problems with handling, developers extended them with new layers of functionality, often dubbed extensions. The increase in expressive power of FSMs from extensions was often specific and restricted to certain domains. Others, however, solved common problems occurring in game AI programming and has been widely used in the game industry. In the next sections we will look at some of the most commonly used extensions.

parallel FSM

Parallel FSMs — One of the most popular extensions to FSMs is called **parallel FSM**, [30, p. 161]. The reason for its popularity is that it provides a work-around for the complexity problem. It allows for complex behavior to be expressed using simpler FSMs by running several FSMs in parallel, thereby avoiding the need for duplicating states.

The concept of having multiple FSMs for the same agent may seem inappropriate, as if trying to simulate the agent by giving it multiple brains. However, it is only a matter of viewing the agent a bit more abstractly. Consider, for instance, an old-fashioned airplane fighter with a pilot, a gunner and a bomber. If you think of the airplane as an agent, the three operators within will each have their specific behavior which, in concert, forms the behavior of the airplane. The pilot handles the navigation, the gunner handles the weapon orientation and firing and the bomber handles the dropping of bombs. Because of the (relative) independence of these behaviors, they can be formed to work autonomously, i.e. independent of the others, and run in parallel to model the entirety of the airplane behavior.

Figure 2.25 on the following page illustrate the concept with two FSMs able to run in parallel: a locomotion handling FSM (figure 2.25(a)) and a weapon handling FSM (figure



(a) FSM for patrolling and chasing (b) FSM for controlling a weapon

FIGURE 2.25: *Expressing more complex behavior by using two FSMs in parallel, thereby avoiding the complexity problem, [30, p. 162, 163]*

2.25(b)). Together they form the behavior of figure 2.24(b) on the previous page while avoiding the added complexity.

The downside to this approach is that the parallel FSMs must be independent of each other, something that is hard to achieve in practice. Take our airplane example, for instance. If the different layers of the FSM were completely separate, the gunner will be unable to inform the pilot of targets that escape his range or angle, and the bomber would be unable to bomb his target if the pilot decides to fly the airplane upside-down. Thus, some information sharing between the layers will often be required. Because the layers operate in parallel, this can result in **write conflicts** or **deadlocks**. The next approach expands on this issue by enabling information sharing between individual agents.

write conflicts

deadlocks

Synchonized FSMs — In a cooperative multiagent system (section 2.3.1 on page 17) the agents work together in order to maximize their performance measure, i.e. achieve a common goal. To create the AI for this, we need some form of information sharing between the agents. Specifically, an agent need to have some means of synchronizing its behavior with the other agents. An extension for handling this is called a **synchronized FSM**. The information sharing is typically handled via either **message passing** or a shared memory architecture, e.g. a **dynamic blackboard**, [47, p. 16].

synchronized FSM

message passing

dynamic blackboard

Message passing requires knowledge of a subset of cooperating agents, to which each new piece of shared information is broadcast. Information sharing via message passing can slow performance in agent-dense systems and is not guaranteed to be lossless if agents do not pass messages to all other cooperating agents. This may be desirable at times, however, such as when the environment is not fully observable (section 2.3.1 on page 16). It could, for instance, be used to only inform agents who are within earshot.

A dynamic blackboard is a shared bulletin board on which each cooperating agent, called **knowledge sources**, can post and read information. The blackboard approach is prone to help agents "cheat" by supplying information they could not have obtained by following the laws governing the game environment. If agents put the position of spotted enemies on the blackboard, it would thus enable the agents to be aware of enemies they could not see yet. This is, in fact, a desirable trait in some scenarious. The squad-based enemies of Half-Life use (or more accurately; they appear to be using) walkie-talkies to alert squad-members when they spot the player, thereby sending a strong signal of intelligence to the player who is able to overhear their communication. In fact the perceived intelligence is an illusion that neatly covers the underlying technology; a post on the squads blackboard and play-back of an appropriate audio-file, [30, p. 166].

knowledge sources

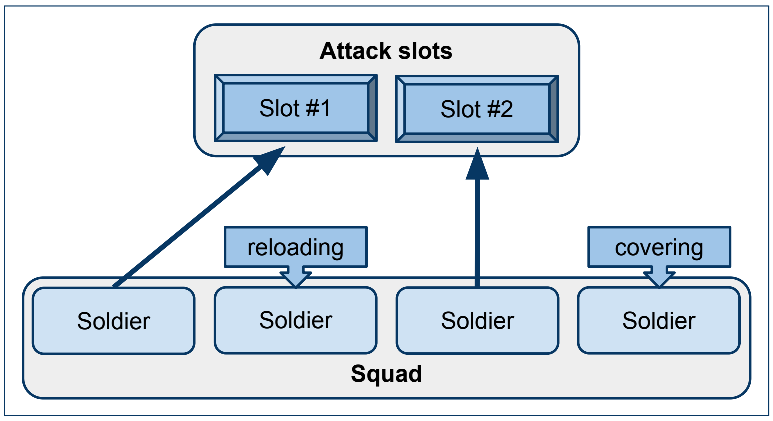


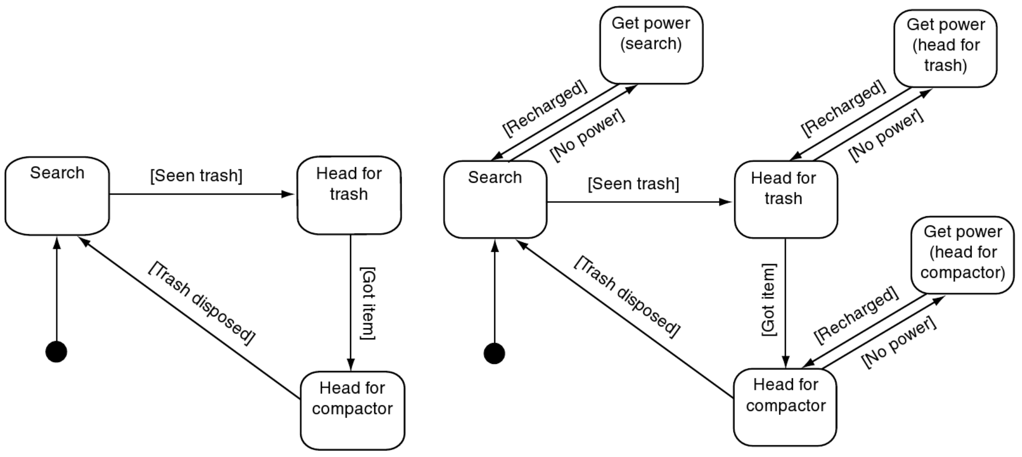
FIGURE 2.26: *Selecting soldiers from the squad to actively engage the enemy. This squad attack behavior is nick-named the "Kung-Fu" tactic due to the limited number of active combatants.*

This blackboard technique is used to control the attack behavior of squads in Half-Life, as also mentioned in section 2.6.4 on page 26, as well. To avoid overwhelming the player, only up to two squad members are actively attacking the player at any time with the rest of the squad engaged in other combat-related activities such as reloading or covering. In the implementation, each squad has two slots available for attacking. A squad member can only attack if there is a free attack slot and if he chooses to attack, he fills one of the available slots. Figure 2.26 illustrates this technique. The explicit communication between the squad members is conveyed simply from playing an audio clip when filling or leaving an attack slot (e.g. "*Moving in!*" and "*Cover me!*", respectively), [39].

Heirarchical FSMs — Traditional finite state machines do not scale very well. Also, using regular FSMs it is hard to model so-called **alert mechanisms** without having to duplicate states, [42, p. 318]. Alert mechanisms are cases where the current behavior temporarily

alert mechanisms

needs to be abandoned for a more important behavior and is then returned to once the more important behavior has finished. An example hereof could be a patrolling behavior (section 4.2) that is interrupted by the agent spotting an enemy. We would then like the agent to engage the enemy and return to patrolling afterwards. Figure 2.27 illustrates the concept of an alert mechanism on a service robot for cleaning. Figure 2.27(a) shows a regular FSM for the behavior of the robot, with the black dot representing the initial "state". In figure 2.27(b) the same behavior has been extended with an alert mechanism for when the robot is low on power. Notice that this new state must be added for each existing state in order to allow execution to continue from where it left off.



(a) FSM for a cleaning robot (b) FSM for a cleaning robot with an alert mechanism

FIGURE 2.27: *Extending a FSM with an alert mechanism will result in duplicating states to be able to return to the previous state, [42, p. 319]*

hierarchical finite state machines

super-states

generalized transitions

A technique known as **hierarchical finite state machines** (HFSMs) has been a popular solution for these problems, [37], [17]. HFSMs make it easier to reuse game logic and to interrupt and resume running behavior. The design of HFSMs differs from that of FSMs in two ways: states may be grouped together to form so-called **super-states** and these super-states can have **generalized transitions** to other super-states, [17]. These super-states can be FSMs on their own and is the means for creating a hierarchical structure:

> *The primary difficulty is coping with courses of action – such as having a conversation or a cup of tea – that consist eventually of thousands or millions of primitive steps for a real agent. It is only by imposing hierarchical structure on behavior that we humans cope at all [50, p. 970]*

Figure 2.28 on the next page illustrates the concept of grouping together states into higher level super-states. The main advantage over regular FSMs is that we avoid cluttering the design with duplicate states for handling context dependent states. As an additional benefit, the grouping allows the designers to think about (super-)states on a higher level of abstraction and is better suited for focusing on the behavior of the states, rather than the details how the FSMs are wired internally. As a result the design of the system becomes more intuitive and easier to reason about[15].

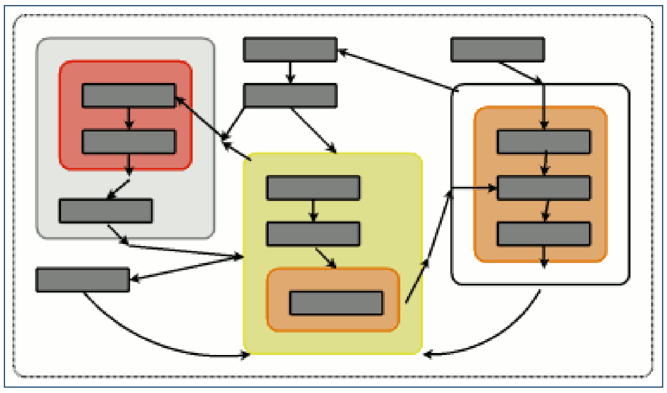


Figure 2.28: *Figurative illustration of the concept of grouping states into super-states, [17]*

Let us return to the example of the cleaning service robot. Its behavior from figure 2.27(b) on the facing page is shown in figure 2.29 using a HFSM instead of a FSM. We now get several states grouped in the super-state "Clean up", describing the higher level combined behavior of the internal states. Also, the alert mechanism for when the robot is low on power and needs to be recharged is now integrated into the design in a much more intuitive manner. The simplicity of the design is not mirrored by the implementation, however, as the hierarchical structure needs to maintain the currently active state in each super-state.

The "H*" in the "Clean up" super-state in figure 2.29 refers to the "history state" which is the state that needs to be re-entered when returning to the super-state after executing (a number of) other super-states, [42, p. 320]. The implementation of a HFSM depend on a nested list of "active" states representing the recursive invocation of states throughout the hierarchy. The complexity of the implementation of HFSM largely comes from not only having regular transitions within the states of super-states and generalized transitions between super-states, but also explicit transitions between the inner states of a super-state *directly* to another state outside the super-state. For instance, the "Search" state may be designed to transition directly to the "Get power" state if nothing is found during the search. Pseudo code for the implementation can be found in [42, sec. 325-330][16].

[15]HFSMs are actually derived from statecharts, a formalism authored by David Harel in 1987 to help in visualizing complex systems, [17]

[16]Be warned: The pseudo code for the HFSM forms one of the longest and most convoluted algorithms of the 800-page book

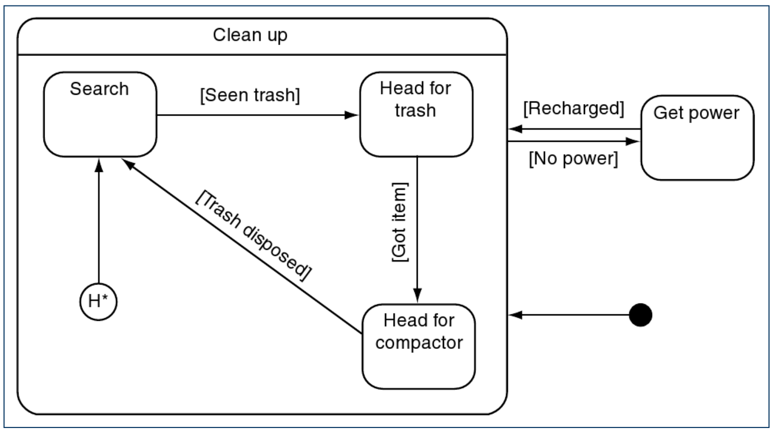


FIGURE 2.29: *The behavior of a cleaning robot with an alert mechanism designed with a HFSM, [42, p. 320]*

Other extensions — There is a long list of extensions to finite state machines. Many extensions are specialized, focusing on optimizing performance, memory or enabling certain types of behavior. Our focus is on general tools useful for modeling game AI, and a few note-worthy extensions from this category are summarized below.

Stack-based finite state machines

push-down automata

- **Stack-based finite state machines**
  A stack-based FSM, also known as a **push-down automata**, is a FSM that stores a recorded history of visited states. This history can then by used to e.g. resume interrupted behavior. [46, p. 10]

Polymorphic finite state machines

- **Polymorphic finite state machines**
  The concept behind polymorphic FSMs is that we might be able to use *almost* the same FSM for similar types of agents, e.g. a rifleman and a grenadier, [32, p. 297]. Polymorphic FSMs can be instantiated into slightly different FSMs by having certain states that are instantiated depending on e.g. the character type of the agent.

Non-deterministic finite state machines

- **Non-deterministic finite state machines**
  Non-deterministic FSMs use (pseudo) randomness to avoid predictability and increase replayability, [30, p. 168]. If a state has several outgoing transitions, each transition may have an weight associated with it that determines the possibility of choosing that transition. For instance, a FSM for a patrolling behavior may include a 25% chance of stopping for a smoke whenever a waypoint is reached. Although this randomness makes the agents less predictable, it also makes it harder to design the desired behavior.

2.7.3 Rule-based systems

localized

sequential

Finite state machines are a suitable tool for modeling many types of game AI. The types of behavior best suited for being represented by a FSM are those which are **localized** (each state has a small number of transitions) and **sequential** (the sequence of states is important). However, if we need to model behavior that depends on global conditions and disregards sequentiality, we clearly need another tool. That tool is often chosen to be a **rule-based system** (RBS) also known as a **production system** or **expert system**, [51], [47, p. 22].

rule-based system

production system

expert system

A RBS consists of a prioritized list of rules called **condition-action** pairs (or **reaction rules**), [72, p. 43]. For each rule, there is a set of conditions on the left-hand side (LHS). On the right hand side (RHS), there is an action. If the conditions on the LHS evaluates to `true`, then the corresponding RHS action is performed. The rules are evaluated from the top down, and terminates whenever an action has been executed. The following list is an example RBS illustrating the form of rules.

condition-action

reaction rules

- Condition$_{1.1}$ $\rightarrow$ Action$_1$
- Condition$_{2.1}$ && Condition$_{2.2}$ $\rightarrow$ Action$_2$
- Condition$_{3.1}$ && $\neg$ Condition$_{3.2}$ $\rightarrow$ Action$_3$
- (...)

A rule-based system for an example enemy AI might look like the following.

- Mortally Hurt $\rightarrow$ Seek Cover
- Enemy Spotted && Have Weapon $\rightarrow$ Attack Enemy
- Enemy Spotted $\rightarrow$ Flee
- (...)
- `true` $\rightarrow$ Idle

The game AI will prioritize staying alive, thus the "Mortally Hurt" condition is the first to evaluate. If it evaluates to `false` the system will instead evaluate whether an enemy is spotted *and* a weapon is yielded. If both evaluate to `true`, it will proceed to attack the spotted enemy. Because of the prioritization, the condition of the next rule actually reads ($\neg$ Mortally Hurt && Enemy Spotted && $\neg$ Have Weapon), thus causing a fleeing behavior if evaluated to `true`. Finally, at the end of the list we have the default action triggering if no other condition is met.

In games, RBSs are used to model the behavior of enemy AI in most real-time strategy games, [30, p. 173], [51]. In such games, players need to manage the tactical aspects of base and unit construction as well as the defense and offense. An example is Age of Empires II: The Age of Kings (see 2.6.5 on page 29), which uses a rule based system for the behavior of the enemy AI. The developers chose to use a **symbolic rule system** for specifying the rules. A symbolic rule system is a scripting language that is specific for the domain of rule-based systems. A concrete example hereof is shown in the listing below.

symbolic rule system

```
(defrule
      (resource-found wood)
      (building-type-count-total lumber-camp < 5)
      (dropsite-min-distance wood > 5)
      (can-build lumber-camp)
=>
      (build lumber-camp)
)
```

LISTING 2.4: *Implementation of a lumber camp building rule in the scripting language used in Age of Empires II: The Age of Kings, [30, p. 173]*

Listing 2.4 specifies a rule for building lumber camps. The conditions are that (1) wood is found, (2) fewer than five lumber camps are owned, (3) the distance to the wood is greater than five units from an existing lumber camp and (4) we are able to build a lumber camp. If all four conditions are met, the AI will proceed with the corresponding action, that is; to build a lumber camp.

user-defined facts

RBSs can be extended in a variety of ways to increase its expressive potential. If the RBS is needed to keep state, it can be extended with **user-defined facts**. User-defined facts allows for setting and checking facts, i.e. flags that can be used to keep state. This could be implemented in the action `(set-fact KEY VALUE)` (e.g. `(set-fact building-lumber-camp 1)`) and the condition `(check-fact KEY RELATION VALUE)` (e.g. `(check-fact building-lumber-camp equals 1)`), [30, p. 179].

knowledge base

inference engine

semantic reasoner

working memory

Another extension to RBSs is to base rules on a **knowledge base** and an **inference engine** (or **semantic reasoner**). A knowledge base is a blackboard of information the agent has been provided with or has gained from experience. It can hold global knowledge or temporary **working memory**. In games, the knowledge often relates objects in the environment to some statically defined properties. That is, it defines the relationship between objects and the properties. For instance, an agent representing a mouse may have a (global or temporary) knowledge base stating that `CHEESE is EATABLE; CATs are ANIMALs; CATs are DANGEROUS`. The agent can then use its inference engine to derive answers from the knowledge base to reason about other things in the environment. For instance, knowing that Garfield is a cat, the mouse can make the following reasoning: `GARFIELD is a CAT; CATs are DANGEROUS` → `GARFIELD is DANGEROUS`[17].

The disadvantages of RBSs is that the behavior must be formulated entirely from `if-then` statements, i.e. condition-action pairs. Additionally, they are hard to maintain for large problem areas where many rules are required.

“ *The difficulty in using RBS lies with the high learning curve and poor tool support. [72, p. 44]* ”

decision tree

RBSs can be implemented using a **decision tree**, [30, p. 172]. Figure 2.30 on the facing page shows a decision tree modeling the behavior for how to engage an enemy in combat.

In the next section we will look at game AI through the more general issue of solving problems by searching to find solutions to a series of subproblems, thereby solving the original problem.

[17]Note that `CATs are DANGEROUS; CATs are ANIMALs` → `ANIMALs are DANGEROUS` is invalid, as should be determined by the inference engine. Also, Garfield is undoubtedly more dangerous to his owner, Jon Arbuckle, than he is to mice.

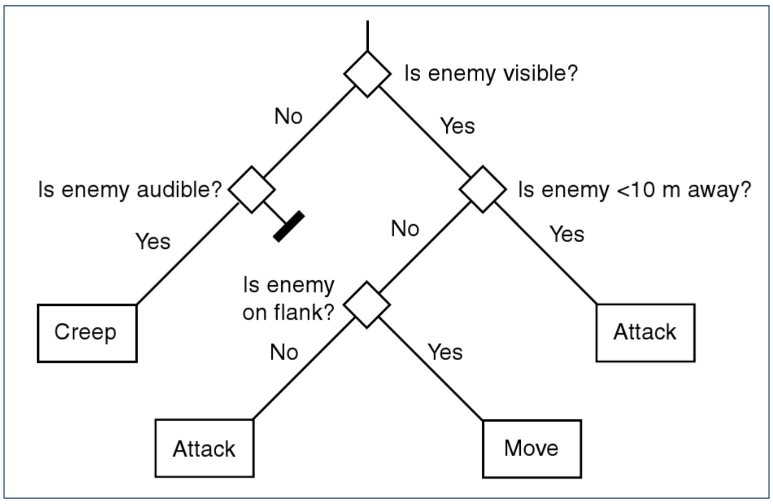


Figure 2.30: *Example of a decision tree for modeling a rule-based system, [42, p. 296]*

2.7.4 Problem solving

Finite state machines (section 2.7.2) are good at modeling sequential behavior and rule-based systems (section 2.7.3) are good at modeling global prioritized rules, but what about the AI needed to play a game of Tic-Tac-Toe? Clearly, it is infeasible to use a RBS or a FSM given the amount of rules[18] or states[19] required to represent each possible situation, i.e. the **state-space**, [60]. Instead, for these kinds of problems, we can turn to the **state-space search** paradigm, [50, p. 94].

state-space

state-space search

State-space search

In state-space search we have an initial configuration from which there is a number of possible states we can transition to. From each of these states we will again have a number of possible states we can transition to and so forth. We can thus represent the possible states as a tree in which the nodes are configurations (or states) and the edges are the actions that change the parent configuration into the child configuration. Such a tree is called a **game tree** and figure 2.31 on the next page shows a game tree for Tic-Tac-Toe.

game tree

The leafs of the tree are termination configurations, some of which may be **goal states**. Goal states are those configurations in which the sought after state is achieved. In agent terms, it is analogous to a maximization of the performance measure, i.e. the agent "wins" the game.

goal states

If the size of the state-space is relatively small, we may be able to search the entire tree using a brute force approach, i.e. a blind or **uninformed search**. This can be achieved by

uninformed search

[18]The upper bound on the size of the state space is $3^9 = 19,683$. Removing illegal positions yield a size of $5,478$, i.e. the number of rules needed to fully represent the state space in a RBS

[19]The upper bound for the size of the game tree is $9! = 362,880$. Removing illegal positions yield a size of $255,168$, i.e. the number of states required to fully represent the game tree in a FSM

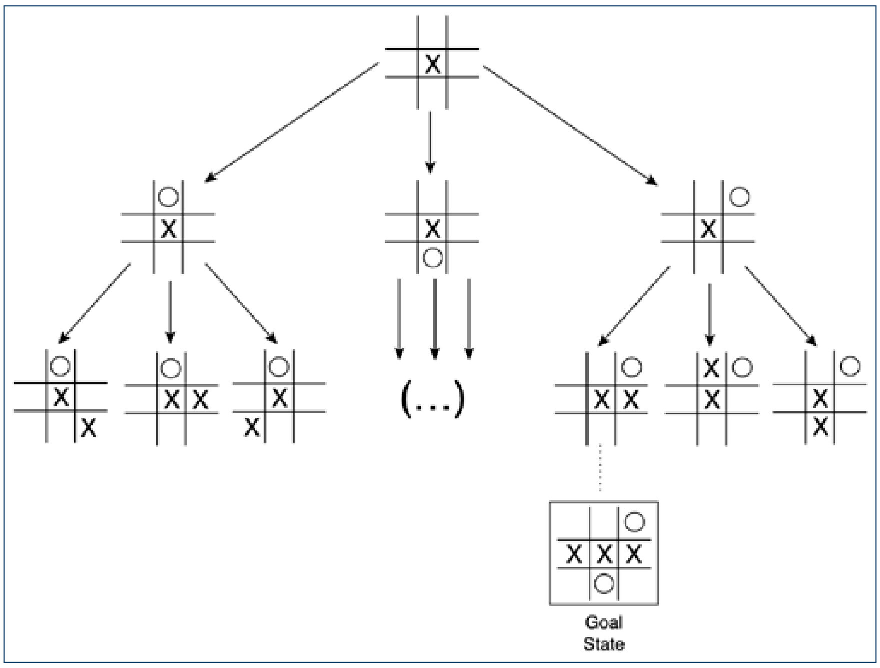


Figure 2.31: *Subset of the game tree of a game of Tic-Tac-Toe, [30, p. 181]*

depth-first search

a **depth-first search** (DFS). We may terminate once we have found a goal state, or we may proceed to search the entire state-space in the hope of finding a better[20] goal. Once we have found an acceptable goal state, we know the sequence of actions (i.e. edges) that change the initial configuration into a goal state. It may not be possible to perform the entire sequence of actions as the opposing player may (with good probability) counteract and disrupt the plans. In that case, we simply recompute the best course of action whenever necessary.

heuristic

exhaustive search

If the size of the state-space is too large for doing uninformed searches[21], we may instead use a **heuristic** based algorithm, [50, p. 73]. In such algorithms we use some auxiliary information in order to disregard portions of the state-space. Because we do not make an **exhaustive search** in the state-space, we are not guaranteed to find the optimal solution in all cases. However, most heuristic algorithms have proven upper and lower bounds on their approximation, so we can ensure the solutions to be within an acceptable fraction of the optimal.

Pathfinding

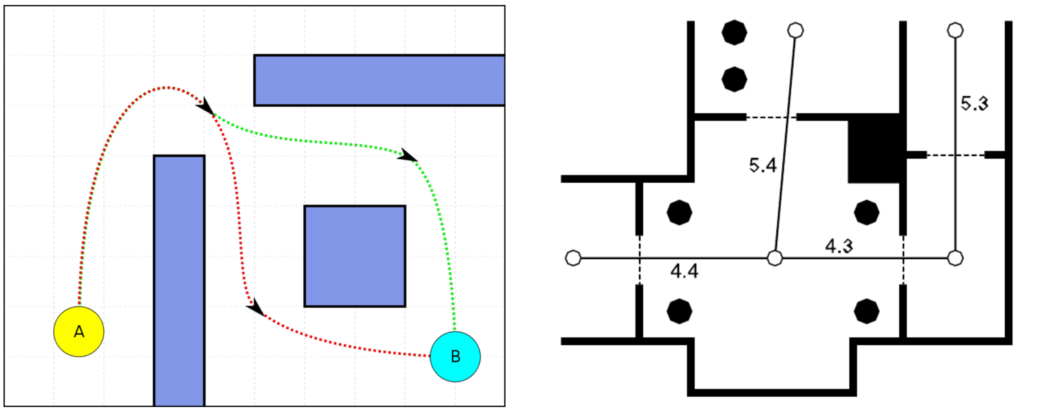
pathfinding

Another highly popular application of search algorithms is in **pathfinding**, [30, p. 221]. Pathfinding is the process of finding a “best“ path from point A to point B (figure 2.32(a)).

[20]Although the goal may be better in itself (e.g. one might deem it better to win Connect-Four with five rather than four connected colors), it may also have a shorter path from the initial configuration, more reachable goal states in its subtree, etc..

[21]That is, using a brute-force approach will consume more time than we are willing to spend

A best path may be the shortest, the easiest, the safest etc. depending on the context.



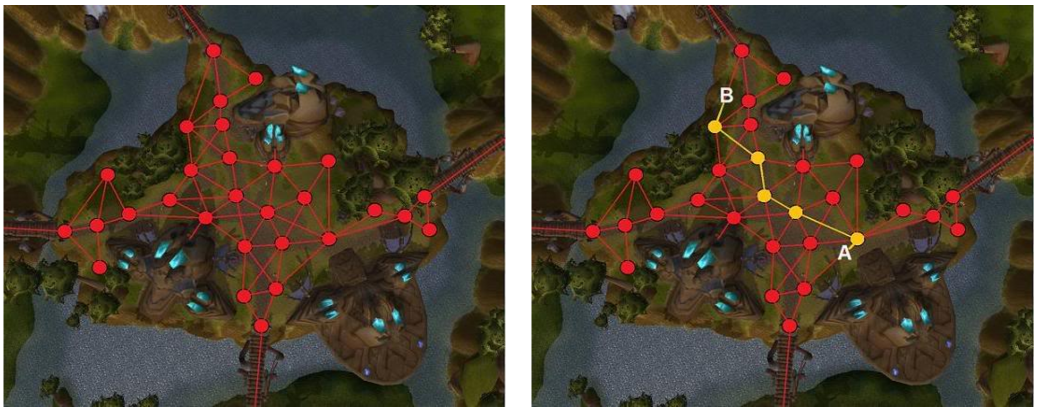
(a) Computing paths from point A to point B in a virtual environment, [68]

(b) A weighted graph overlaid on a game level, [42, p. 201]

FIGURE 2.32: *Pathfinding in game level environments and their geometry*

weighted graph

Often, pathfinding is performed by overlaying the game world with a **weighted graph**, $G = (V, E)$, with vertices V depicting accessible areas of interest (figure 2.32(b)). There is a weight matrix associated with the edges E depicting the cost associated with moving from one node to a connected node. The best path is found by minimizing the cost of moving from one node to another via a number of connected nodes.



(a) A weighted graph, or waypoint mesh, on a game terrain to guide pathfinding

(b) A found path from vertex A to vertex B via connected vertices in the graph

FIGURE 2.33: *Using a graph as an auxiliary tool for computing paths between locations in a game world, [53]*

Dijkstra's algorithm

single source shortest path

One popular approach in games is to use **Dijkstra's algorithm**, a so-called **single source shortest path** algorithm for finding optimal paths in such a graph, [30, p. 224].

It computes the shortest path from a single source, $s \in V$, to all other vertices in V. If we know the endpoint, B, a non-global algorithm will be much better suited for computing a path. The algorithm of choice for most games is the **A*** algorithm, [30, p. 228], [46, p. 3], [50, p. 97]. It is a heuristic-based algorithm that determines a path from A to B while only searching the local state-space for each vertex. Figure 2.33 on the previous page shows an example of the best computed path from point A to point B in the waypoint mesh.

A*

Planning

In **planning** we have a goal (as in state-space searching) and a **plan library** of subgoals that works as means for satisfying the goal, [47, p. 21]. As such, it can be thought of as a graph where the nodes are actions, the edges are the conditions that needs to be met in order to proceed with an action and the leafs are the goals.

planning

plan library

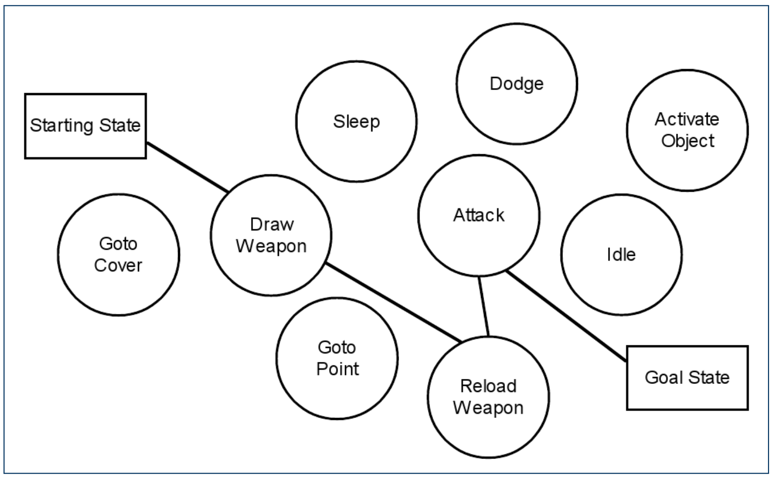


FIGURE 2.34: *Finding a sequence of actions that constitutes a plan for satisfying the goal in the GOAP planning architecture, [43, p. 3]*

We need to search the state-space to find a best path through the graph (figure 2.34). A path through the graph reaching a goal consists of a sequence of actions, called a **plan**. One approach for searching through the state-space is called **Goal-Oriented Action Planning** (GOAP) and is designed specifically for real-time control of autonomous character behavior in games, [43]. An example plan for a agent using GOAP is illustrated in figure 2.35 on the next page. Through searching the GOAP system has found a series of actions that, if successfully executed, will result in the agent achieving its goals of killing the enemy and recovering lost health.

plan

Goal-Oriented Action Planning

Another approach is based on the idea of **hierarchical decomposition**. In this approach, each node in the graph is a composite action which may consist of a number of subactions. The subactions are again composed by other actions until we reach **primitive actions**. The approach is called **hierarchical task network planning** (HTN planning).

hierarchical decomposition

primitive actions

hierarchical task network planning

In HTN planning, we have an initial plan to reach a high level goal, [50, p. 422]. Actions from the plan library are then instantiated and combined to satisfy the goal of the plan.

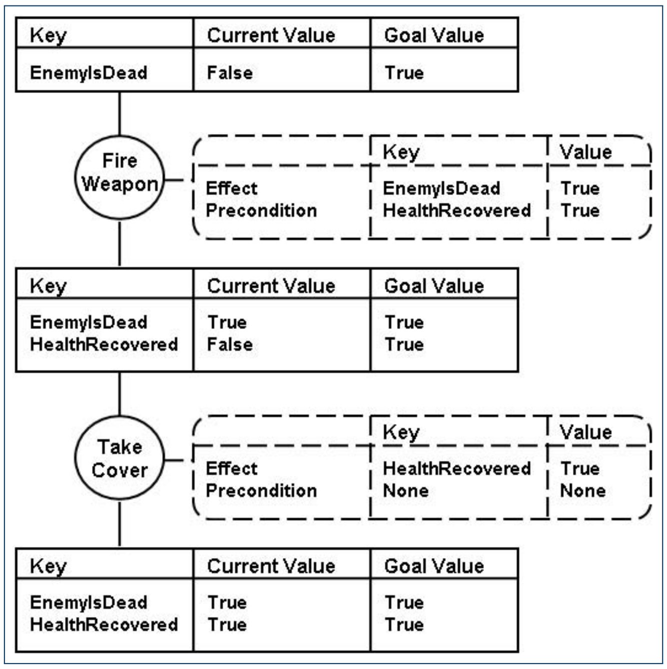


Figure 2.35: *Plan for an agent using GOAP. The goal of the agent is to kill the enemy and recover health, [72, p. 45]*

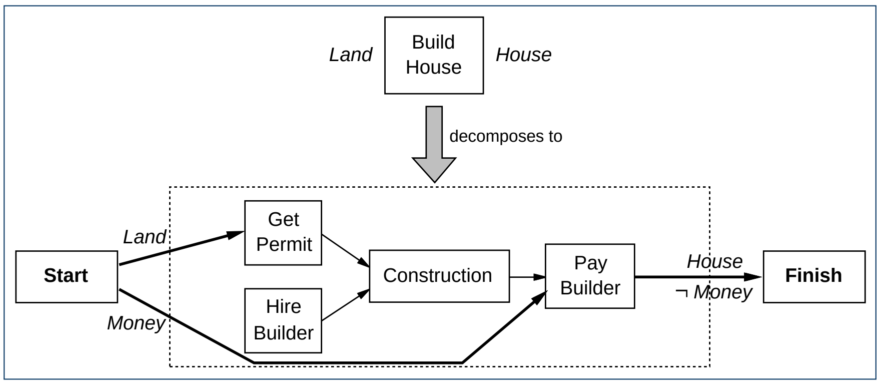


Figure 2.36: *Hierarchical task network planning, [50, p. 424]*

Each (high level) action has a number of **external preconditions** and **external effects**. External preconditions are the conditions required for the action to be viable. External effects are the high level postconditions of performing the action. In figure 2.36 we have the high level plan "Build House". This plan has the external precondition "Land" and the external effect "House", i.e. given a land property the plan will result in a house on the property. A decomposition of the "Build House" plan into a possible action sequence is shown in the buttom of figure 2.36. In the decomposition we also see the **secondary effect** $\neg Money$, that is not used directly to achieve the goal but is consumed within the action decomposition.

external preconditions

external effects

secondary effect

An implementation of a HTN planner used in games is the **Simple Hierarchical Or-**

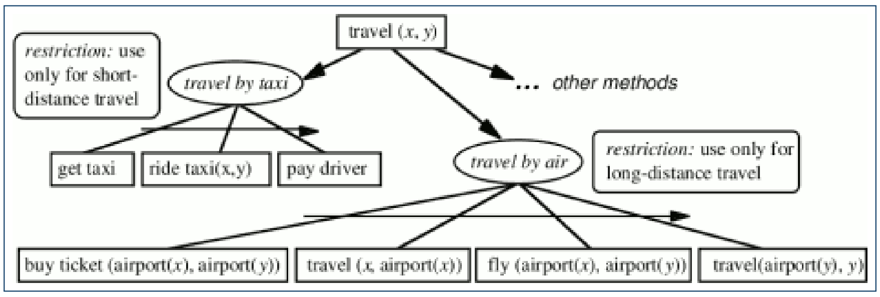

travel (x, y)
restriction: use only for short-distance travel
travel by taxi
... other methods
get taxi
ride taxi(x,y)
pay driver
travel by air
restriction: use only for long-distance travel
buy ticket (airport(x), airport(y))
travel (x, airport(x))
fly (airport(x), airport(y))
travel(airport(y), y)

FIGURE 2.37: *Using SHOP to construct a travel plan in a HTN, [24]*

Simple Hierarchical Ordered Planner

dered Planner (SHOP), [24]. Figure 2.37 shows an example illustration of how the SHOP system may be used to create a traveling plan from X to Y. The SHOP system addresses issues of efficiency and flexibility of common HTN planners by making it easier to insert custom, domain specific, logic to guide the search, [24].

The concept of planning is very powerful and has found usage in many applications besides game AI. It is goal-directed by construction and great for autonomy (section 2.2.2). However, in most cases it is also very difficult to implement, also for games, [24]. The main reason is that most games are complex systems in which it can be hard to determine the outcome of a series of actions[22]. Planning has been criticized for being disconnected from the real world, [23, p. 20].

In the next section we will look at the concept of scripting for aiding in data-driven development and in easing the content creation process.

2.7.5 Scripting

Scripting

scripting languages

In section 2.6.3 we emphasized the advantages of data-driven development. **Scripting** is the concept of loading some or all of the (game AI) logic from external resources, [46, p. 9]. Scripting relies on custom file formats, and sometimes even **scripting languages**, for expressing logic in a domain specific manner, [32, p. 288]. Game AI related resources are the prime candidates for a scripted approach, although other parts of games (e.g. rules for gameplay or game dialog) also can take advantage of scripting.

> “*Scripting is a very popular method* [...] *It streamlines the creation of the AI and logic content by placing that responsibility in the hands of people different from those actually coding the engine. It also allows for more creative AI design. Often, a dedicated AI programmer using a scripting language will come up with new ideas, and because the language will allow him to test them immediately, he will create more compelling and believable characters.*

[22]As mentioned in section 2.2.1 on page 12, omniscient agents are impossible in reality and very hard to achieve for anything other than simple, closed and well-defined systems

It all boils down to selecting the right feature set for the scripting language so that it combines a powerful set of primitive calls with the ease of use your content team will demand, [30, p. 250]

Scripting can be used on many different levels, from using no scripting at all to creating the entire game using scripts. Most games, however, use a medium level of scripting where the core gameplay and game AI have been hardcoded and scripts are used for handling special gameplay (e.g. in-game mini-games or story telling) and for handling custom agent behavior. In this common ”custom behavior“ level usage of scripting, very specialized behavior is often known as scripted sequences, and is sometimes frowned upon for not being good game AI but rather a form of virtual puppeteering:

Without the luxury of waiting for [game AI] *progress, the commercial entertainment industry has had to rely on fully scripted interactions with human players to support immersive interactions of any complexity besides combat. This disparity between the need for AI in entertainment environments and the state of AI research will continue to grow as Interactive Entertainment forms push in new directions. [34, p. 1]*

As an example of a scripting language, let us examine Lua. Lua is ”a powerful, fast, lightweight, embeddable scripting language“ that has been widely used in the video game industry, [3]. Below is an example of Lua code.

```lua
health = 100

function take_damage(x)
  print("Ouch!")
  health = health - x
  if (health <= 0) then
    dead()
  elseif (health < 30) then
    find_cover()
  end
end

function dead()
  print("You are dead")
  FUNC_DEAD() -- call a registered function
end

function find_cover()
  -- code for finding cover to regain health
end
```

Invoking the take_damage-function three times with 40 as argument yields the following output:

```
Ouch!
Ouch!
Ouch!
You are dead
```

One reason for the popularity of Lua is that it provides a very simple API for communicating between the native and the scripted code. The FUNC_DEAD-function is a native function, for instance. The following C code shows how simple the native function is registered as a function available in the Lua scripts.

```c
static int playerIsDead(LuaState* state) {
  // code for function
  return 0;
}

void registerNativeFunctions() {
  LuaStateOwner state; // Create state
  // Register a native C/C++ function for access via Lua
  state->GetGlobals().Register("FUNC_DEAD", playerIsDead);
}
```

Scripts ease the development process because they are domain specific higher level languages. Furthermore, they do not require the game to be recompiled – or in some cases even restarted – to be able to see the in-game results of changes in the scripted code. They are also very flexible which can be both an advantage and a disadvantage. The advantage is that there are no limitations to what they can compute. The disadvantage is that there is no overall structure. There is nothing to guard the game AI designers from writing sloppy, unstructured and undocumented code. Additionally, the game designers need to be very aware of the inner workings of the native code base as the scripts need to cooperative with the **game engine**. They need to know which functions are exposed and how they work. If the scripting system is developed in-house it may be better suited for specific purposes, but it will often be error-prone and expensive to maintain[23]:

game engine

> *For everything you might expect to benefit from a scripting language* [...] *there are just as many pitfalls [32, p. 289]*

To aid the game designers in the scripting process, scripting tools or editors are often utilized. In many cases, these tools are created by the in-house tool programmers as they

[23]It used to be common for each game studio to develop their own scripting language. However, increasing requirement for the quality and efficiency of scripting engines as well as advances in off-the-shelf scripting languages has as made it much less desirable for game studios to create their own language. Instead, many studios use free or commercial scripting languages such as Lua, [30, p. 251]

are often very tightly coupled with the game engine. The tools ease the interfacing with the game engine, provide syntax highlighting in the scripts and sometimes even provide facilities for profiling and/or debugging. Common for all scripting tools is that the game designer has to write code for expressing the desired behavior. While this approach is magnitudes better than having game designers write behavior directly in native code, it still is not intuitive for many game designers.

In the next section we will take an introductory look at a new and popular technique for creating character-based AI. We will discuss the technique more fully in chapter 3.

2.7.6 Behavior trees

behavior tree

The **behavior tree** (BT) technique is a novel approach to game AI that merges the core qualities of several popular technologies. Plainly put, BTs are a compromise between FSMs, HTN planners and scripting. Although the BT technique is firmly rooted in FSMs and HFSMs, it does not have a single historical precedence, [36, sec. "Overview“]:

> *They* [behavior trees] *are a synthesis of a number of techniques that have been around in AI for a while: Hierarchical State Machines, Scheduling, Planning, and Action Execution. Their strength comes from their ability to interleave these concerns in a way that is easy to understand and easy for non-programmers to create, [42, p. 334]*

It has emerged as a way of easing the modeling of purposeful action selection for agent-based behavior, [35, sec. "Behavior Tree AI“]. Although it is a relatively new technology, it has gained widespread popularity and recognition in the game industry. It has been used in popular and critically acclaimed games such as Halo 2, Spore and Grand Theft Auto: San Andreas, [37], [35], [23]:

> *Behavior trees on their own have been a big win for game AI, and developers will still be exploring their potential for a few years, [42, p. 371]*

So what is a behavior tree actually? In simplified terms, a behavior tree may be regarded as a HFSM where the transitions are implicitly handled by special nodes that impose an ordering and flow of control among its children, [42, p. 334]. To make the definition more tangible, let us examine a simple example.

Figure 2.38 on the next page shows a behavior tree with four nodes named `Behave`, `Idle`, `Suspicious` and `Attacking`. `Behave` is the root of the tree and guides the control flow between its three children. In this example, the children are all composite nodes, that is, they are the roots of subtrees with a number of child nodes. Each of the three nodes constitute

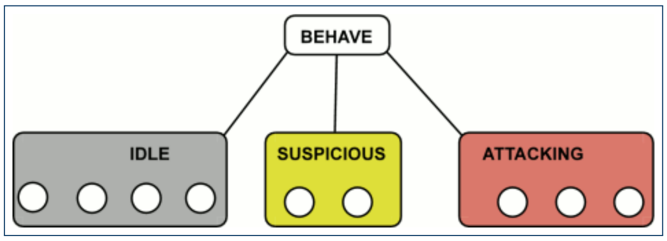


FIGURE 2.38: *Simple high-level example of a behavior tree, [19]*

a certain behavior. Here, we might imagine the agent being an enemy soldier who either performs an idle behavior (e.g. smokes a cigarette), a suspicious behavior (e.g. attempts to track down the player) or an attacking behavior.

The actual implementation of the three behaviors is hidden in the composition of the subtrees, allowing the designer to focus on the appropriate level of abstraction. Selecting which behavior to perform is the responsibility of the parent node. This selection will often depend on observations in the world. For instance, if the agent is in the `Idle` behavior and spots an enemy, it would switch to the `Attacking` behavior. Loosing the enemy from sight, would then switch to the `Suspicious` behavior.

One thing that sets behavior trees significantly apart from standard HFSMs is that transitions between nodes do not need to be explicitly defined. Handling of transitions are built directly into the hierarchical structure of behavior trees. Action selection is performed by searching through the tree. The search traverses the tree and the nodes guide the direction of the search and thus help to select relevant actions. The search is intuitive as only a few distinct types of search-guiding nodes are needed and each has a specific purpose.

In chapter 3 we will go into much further detail with the function of behavior trees and the specific types of nodes most commonly associated with it. Additionally, we will discuss whether behavior trees are well suited for designing behavior.

2.7.7 In conclusion

In the preceding sections we have discussed a broad selection of common and popular techniques for game AI. We have looked at finite state machines (section 2.7.2) as a popular and very powerful choice for decision making and action selection. We have also seen the areas in which FSMs are limited and need extensions to increase their expressive power in the domain of game AI.

> “*When applied to AI for building purposeful goal-directed behaviors, traditional FSMs just don't scale very well. This is the main reason developers in the games industry have been primarily moving towards systems like behavior trees instead, [31]*”

As a possible solution for overcoming the limitations of FSMs, we discussed the advantages of using a hierarchical approach in both hierarchical FSMs (section 2.7.2), hierarchical task networks (section 2.7.4), and behavior trees (section 2.7.6), [17], [19].

We have introduced emergent behavior (section 2.7.1), rule-based systems (section 2.7.3) and different forms of planning (section 2.7.4) as very diverse approaches for game AI. Although these techniques have valid applications in many games, they are typically used for genre-specific game AI. Emergent behavior is great for simulating fluid movement, animations and flocking behavior. Rule-based systems are good at handling non-sequential behavior in games with a discreet set of rules such as many tactical and turn-based games. Planning has been used for action games and provides modular, goal-directed and autonomous behavior, but can be difficult to use in practice.

Finally, we have seen the applicability of scripting languages (section 2.7.5) for easing the development process.

We have chosen to use behavior trees for a number of reasons. First, it combines some of the best qualities of the most popular approaches to decision making such as FSMs, HFSMs, planners and scripting. It is a novel technique that has gotten much focus and gained a lot of popularity, but has yet to become fully standardized in the game industry. Behavior trees also propose an intuitive and convenient way of thinking about behavior in terms of modular, self-contained, building blocks that can be combined to form more complex behavior. This view could make it easier for game designers to design the desired behavior.

In the next chapter we will more thoroughly explain the concepts and inner workings of behavior trees. Following that, in chapter 4, we will describe how we use behavior trees to create an editor to make it easier for game designers to design character-based behavior.

Chapter 3

Behavior trees

In section 2.7.6 we introduced the behavior tree (BT) technique as a synthesis of some of the most popular game AI techniques and noted the rising popularity of this novel technique. In this chapter we will examine more closely the details of behavior trees and why it is better suited for creating character-based AI than many other common techniques. We look at how behavior trees are defined and used in the literature but also create our own interpretation and implementation of a behavior tree along the way.

3.1 Searching in behavior trees

The main responsibility of any character-based game AI (and most other types of game AI) is to handle decision making, i.e. action selection, [42, sec. 1.2]. In section 2.7.6 we mentioned that the action selection in behavior trees is handled by searching through the tree. The internal nodes of a behavior tree specify how the search should be directed and the leaf nodes are encoded with the actions to perform when the search reaches that node. A path from the root to a leaf specifies a course of action and the available paths are searched from “left-to-right”, i.e. by using a **depth-first search**, [42, p. 340].

depth-first search

As mentioned, the behavior resulting from processing a BT is defined by a search through the tree. The search is guided by the nodes in the tree and different types of nodes have different ways of directing the search. There are only few different core types of nodes, each relying on intuitive patterns to make it easier to reason about the behavior at a higher level. Figure 3.1 on the following page shows an example of a search from the root of the tree that attempts to find a suitable behavior for the given situation. The details of searching through the tree will be explained in the next section as we discuss the different types of nodes in behavior trees.

The nodes that make up a behavior tree form a powerful model of search and behavior execution. It can be compared to the basics of a programming language and there are nodes which act like conditionals, statements, compositionals and control structures, [23, p. 39]. Their usage differs from regular programming languages, however, and provide a simpler and

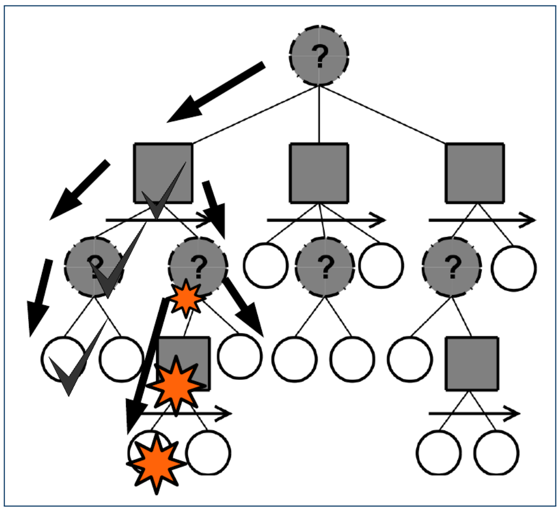


FIGURE 3.1: *Example of a search through a behavior tree, [23, p. 41]*

more intuitive way of reasoning about high-level logic, [23, p. 58]. In the next section we will go into details about the nodes that make up behavior trees and define their functionality.

3.2 Nodes of a behavior tree

Each node in a BT represent a behavior comparable to the actions in hierarchical network task planners (section 2.7.4). Likewise, the nodes of BTs decompose recursively into behaviors of increasing simplicity until the primitive actions of the leaves. Conceptually, a primitive action is an atomic action that cannot be further decomposed. Depending on the level of abstraction, a primitive action may be a very low level action (e.g. taking a single step) or it may be a higher level atomic action (e.g. moving to another location). See section 4.5.3 for more discussion hereof.

As noted in section 2.7.2, imposing a hierarchical structure on primitive actions is essential for being able to understanding and formulate complex behavior. The abstraction inherit in the decomposition of behaviors greatly aid in the process of designing behavior. The high-level behaviors can thus be considered black boxes by designers and their decomposition largely disregarded for the purpose of defining higher-level behavior.

Strategy pattern

The nodes encapsulate behavior and can be replaced or interchanged. They can thus be considered strategies for the behavior tree as per the **Strategy pattern**, [33, p. 127], [30, p. 100]. As such, the types of behavior that behavior trees are able to express depend entirely on the nodes available. A simple behavior tree might have most in common with the types of behavior that a simple reflex agent (section 2.4) can express. If nodes are included that make it possible to store and load variables, the behavior tree could be made to express behavior corresponding to the model-based reflex agent type (section 2.4). Likewise, other techniques may be integrated into behavior trees through nodes as well. Consider the smart terrain technique (section 2.6.4) for instance. Special nodes could be used for listening for broadcast advertisements from the smart objects in the environment. These nodes could then direct the search based on advertisements from nearby objects. We will primarily discuss the basic nodes required for creating the hierarchical structure and function of behavior trees. Some ideas for more exotic nodes are listed in section 5.2.

In the following sections we will outline the core set of nodes available for handling behavior and control flow through the BT.

3.2.1 Tasks

task

A node in a BT is called a **task**, and everything in a BT is made from tasks. A task, as the name suggests, is responsible for performing some work. Tasks express latent computation, that is, they are encoded with a behavior that they may perform upon execution. This is an application of the **Command pattern**, [33, p. 233].

Command pattern

The execution of a task may succeed or fail depending on whether the observed outcome of performing the task matches the expected outcome, [42, p. 334]. For instance, a task that is responsible for moving the agent forward will terminate with success if it actually manages to move the agent forward. The `Task` class defines the overall interface of all nodes in the tree. Pseudo code for the interface of the `Task` class is as shown in listing 3.1. The full source code listing can be found in section A.4.1 on page 133 in the appendix. The interface only has a single function, `execute`, that takes a single argument, `agent`, specifying which agent should execute the task. Having the function take an agent as argument is a way of separating the behavior from the structure it operates on. One instance of a task or behavior tree can thus be used with multiple agents to simulate the behavior of each of them.

```
class Task {
  // Executes the task and returns with SUCCESS, FAILURE or RUNNING.
  function execute(agent)
}
```

LISTING 3.1: *Pseudo code for the interface of tasks*

In addition to terminating the execution in either success or failure, a task can also signal that the execution is on-going and has not yet reached a conclusion. This return value is named "running" and is relevant for tasks which need several frames (section 2.6.3) of processing to finish. For instance, a task that determines the trajectory of a rocket may need several frames to get enough samples to extrapolate the trajectory satisfactory. Likewise, a (very high level) task responsible for finding and eliminating an enemy may require several minutes of frame-by-frame execution before the result hereof can be conclusively determined. The task may then report "running" for hundreds of frames while the control flow fluctuates between the tasks of its decomposition, until the high level task is finally able to terminate its execution. In this case, the result will be a success if the agent managed to find and eliminate the enemy and a failure otherwise (i.e. the agent has died, has given up trying to find the enemy, etc.).

Figure 3.2 on the following page show the flow of control between a parent node and a generic task node, as illustrated by a state transition diagram. The result of the execution

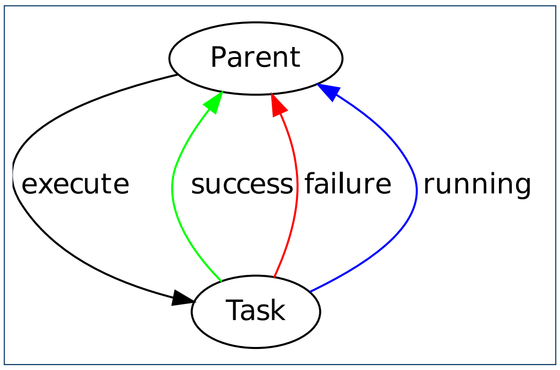


FIGURE 3.2: *State transition diagram illustrating the process of executing a single frame of game AI as seen from the point-of-view of the Parent node*

is represented by the return value and, as we will detail later, is used to build complexity in the tree. In the following sections we will discuss what it means for the execution to succeed or fail, as it depends on the particular type of task.

3.2.2 Conditions

conditions

Some tasks do not change the state of the environment or the agent. Rather, the action it performs in the world is simply a sensory operation. These "read-only" tasks, which only make observations in the world, are called **conditions**. They are used for testing assertions or assumptions of the environment or of the agent itself. The result of executing a condition task is the result of the test. A low-level condition task could e.g. be "Is light intensity $\geq$ 50%?", which would depend on the sensory input from the agents light sensor and the result would either be a success (i.e. `true`) or a failure (i.e. `false`) depending on the result of the test. The result of these tasks are used to filter and guide the control flow of execution through the tree.

A condition task typically depends on data observable by the agent, thereby utilizing one or more of the agents sensors (section 2.2.1). A higher-level condition could be "Is an enemy in sight?". In that condition, the agents vision would be utilized to scan for enemy agents. If enemies were spotted, the result of execution would be a success, otherwise it would be a failure. If the agent were somehow unable to use its vision, the execution would also fail as no enemies could be seen.

The condition task inherits from the `Task` class and implements the interface, that is, the `execute` function.

3.2.3 Actions

actions

The tasks that perform actions in the game environment are called **actions**. They *may* also utilize some sensory facilities, same as conditions, but the primary function of actions is to alter the state of the world or the agent. Examples of actions could be to play an animation on the model of the agent, play an audio-file or use an object in the game world.

The result of the execution of the task depends on whether the desired action is successfully performed. As previously mentioned, the task `Move Forward` succeeds if the agent successfully moves forward and fails otherwise. The task may fail if the agents motor devices are broken, if the agent is stuck, paralyzed or dead etc.. Actions utilize the actuators (section 2.2.1) of the agent for performing actions in the environment.

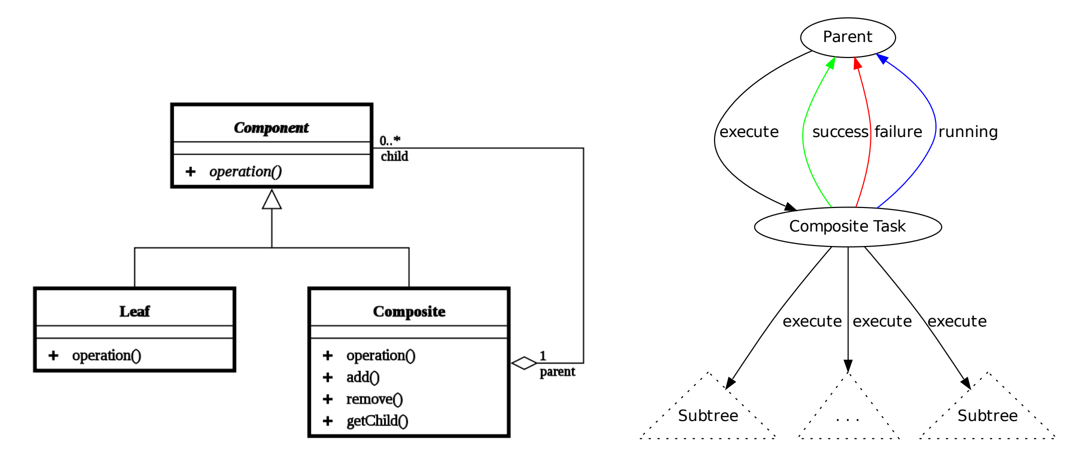
Like the condition task, the action task also inherits from the `Task` class and implements the `execute` function from the `Task` interface.

3.2.4 Composite tasks

To be able to compose tasks into a hierarchical structure, we need a task that can have subtasks. This type of task is called a **composite task**. A composite task is an application of the **Composite pattern** (figure 3.3(a)), [33, p. 163], [30, p. 111]. The Composite pattern enables a tree-structured hierarchy of objects implementing a shared interface. In this case, the interface is that of the `Task` class.

composite task

Composite pattern



(a) UML class diagram of the composite pattern, [33, p. 163]

(b) Execution process of a composite task. Child transitions are disregarded for brevity.

Figure 3.3: *Two different views on a composite: (a) shows the class diagram of the composite pattern and (b) illustrates the composite task in behavior trees*

The composite task implements and extends the interface of the `Task` class. It holds a list of child tasks and include functionality for adding and removing children. Listing 3.2 shows the pseudo code for the interface of the composite task node. The full source code listing can be found in section A.4.1 on page 133 in the appendix.

```
class CompositeTask : Task {
  // List of child tasks
  children : Task list

  // Reference to the currently running task
  activeChild : Task

  // Adds the given child task to the list
  function addChild(task)

  // Removes the child task from the list
  function removeChild(task)

  // Executes the child tasks. Returns SUCCESS, FAILURE or RUNNING.
  function execute(agent)
}
```

LISTING 3.2: *Pseudo code for the interface of the composite task*

A composite task cannot perform any behavior by itself. Its behavior comes from that of its child tasks and their composition. It is the responsibility of the composite task to maintain an ordering among its children and determine the control flow. Figure 3.3(b) shows the (rather abstract) execution of a composite task. The execution depends on the ordering and control flow of the children, as specified by the concrete implementing of the `CompositeTask` interface. The control flow typically depends on the results of the execution of the children. There are many different ways in which the control flow of the children can be handled.

Before we look at concrete implementations of composite tasks, let us first clarify how the actual execution of the behavior tree works. In each frame of game AI processing, a behavior tree (either the same or different ones) is applied to each agent. For brevity, let us focus the discussion on a single agent executing a single behavior tree.

So, we have an agent and need to execute its behavior tree in order to compute and perform the behavior of that agent. The root of a behavior tree will always be a composite task[1]. In the game AI processing step, the agent will invoke the `execute`-function of the root task with itself as argument. If this is the very first time the behavior tree has been executed we will simply traverse it as determined by the nodes. If we have executed the tree before, however, there is most likely a task already actively running somewhere in the tree. The variable `activeChild` refers to this task. It could, for instance, be a repeating patrol behavior. We would like to give the active task more processing time by continuously executing it until it terminates. The composite root task would then execute its active child task, which would again execute its active child task etc.. Consequently, we get a single recursive path down through the tree until we encounter the active task and can continue to execute it. This concept is illustrated in figure 3.4 on the facing page. This search takes $O(\log n)$ time (where n is the number of nodes in the tree) but can be reduced to $O(1)$[2].

The two most important composite tasks are called ”sequences“ and ”selectors“ and will be discussed in the next sections, [42, p. 335].

[1]If it is not, the behavior of the agent can only be a single primitive task

[2]This can be done simply be having a special root node that knows the active task(s) and can execute them directly. It also requires all tasks to have a reference to its parent task.

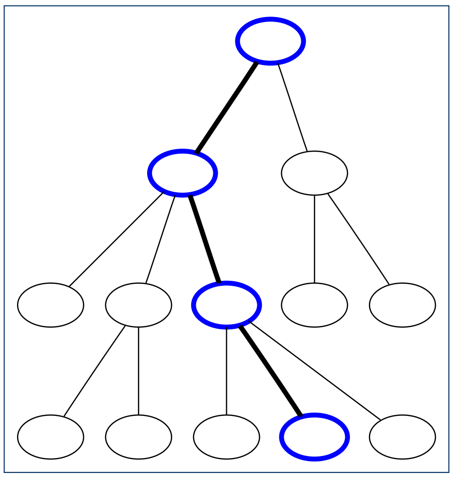


Figure 3.4: *Illustration of the execution of active tasks in the behavior tree. Blue nodes symbolize the currently active nodes.*

3.2.5 Sequences

Sequence

One concrete type of composite task is the **Sequence** task, [42, p. 335][18]. The sequence task is responsible for executing its child tasks in sequence, hence the name. It represents the collective behavior of its child tasks. For instance, consider the sequence that has the children `Acquire Food`, `Prepare Food` and `Eat Food` as shown in figure 3.5.

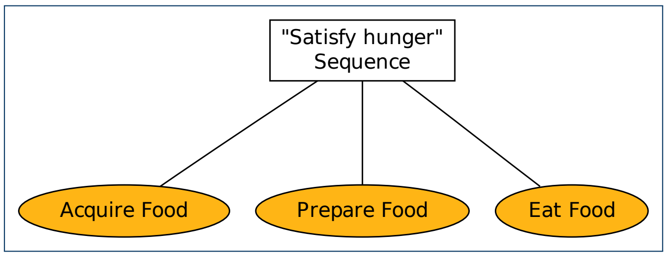


Figure 3.5: *Behavior tree consisting of a sequence task with three child tasks*

The sequence could be considered the behavior of satisfying hunger by (somehow) acquiring, preparing and eating food. Because the tasks in a sequence are sequential, each task depends on the successful execution of all previous task in the sequence. If the `Acquire Food` tasks fails to provide food, then we cannot proceed with the `Prepare Food` task as we have no food to prepare. Likewise, if the `Acquire Food` task succeeds but `Prepare Food` fails (i.e. we have no stove or fire) then we do not want to execute `Eat Food`. In both these cases, we will also have failed the higher level responsibility of the sequence, that is, to satisfy hunger. Therefore, in a sequence task, we will execute the child tasks in order, but fail the entire sequence immediately if a task fails. In other words, sequences deal with the success of a child task by moving on to the next child task, and failures by bailing out and failing the entire sequence. Conceptually, the execution and result of sequences can be thought of as a lazy AND (i.e. && in C and Java) of the children.

Algorithm 2 shows the pseudo code for the `execute` function of the sequence task. It iterates the list of children, executing each child until one does not succeed (i.e. returns "failure" or "running") or it has reached the end of the list. The full source code listing can be found in section A.4.4 on page 138 in the appendix.

Algorithm 2 Sequence (disregarding running tasks for brevity)

```
1: procedure EXECUTE(agent)
2:     for child in children do
3:         $status \leftarrow child.execute(agent)$
4:         if $status \neq$ SUCCESS then
5:             return $status$
6:         end if
7:     end for
8:     return SUCCESS
9: end procedure
```

Figure 3.6 illustrates the inner workings of an example sequence task as a state transition diagram. Here we also disregard the "running" transitions for brevity.

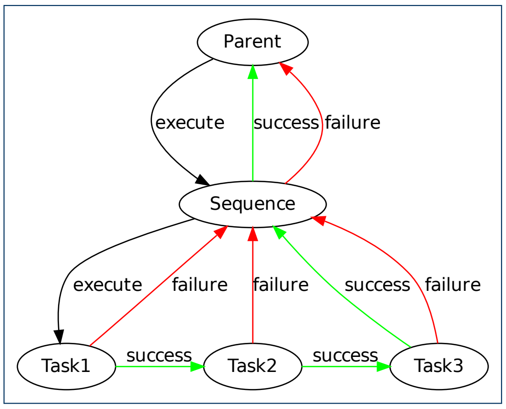
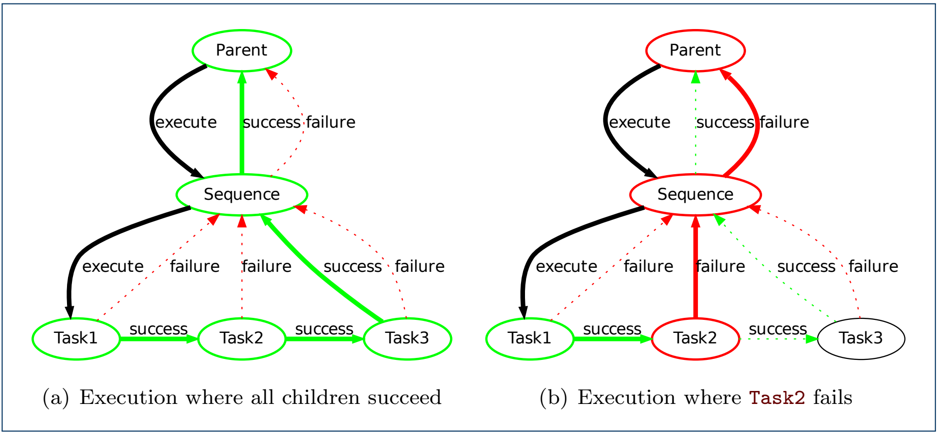


FIGURE 3.6: *State transition diagram of a sequence task with three child tasks*

Figure 3.7 on the next page illustrates the same behavior tree under two example executions. In 3.7(a) we examine the case in which `Task1`, `Task2` and `Task3` succeed. The `Parent` task will execute the `Sequence`, which will then in turn execute `Task1`, `Task2` and `Task3` and then finally return to `Parent` with success. The execution flow thus becomes: `Parent` → `Sequence` → `Task1` → `Task2` → `Task3` → `Sequence` → `Parent`, where → symbolizes the invocation of the `execute()` function of the `Task` interface and → symbolizes the change of execution caused by a successfully completed task execution.

Let us now look at the same behavior tree in a different situation in which `Task2` fails. This is illustrated in figure 3.7(b) on the facing page. Here, `Parent` will execute `Sequence`, which will successfully execute `Task1` and continue to `Task2` which will fail. The entire



(a) Execution where all children succeed (b) Execution where Task2 fails

FIGURE 3.7: *Two different example executions of the child tasks of a sequence task*

Sequence will then fail and will return the control flow to Parent with that result. The execution flow is thus: Parent → Sequence → Task1 → Task2 → Sequence → Parent, where → symbolizes the change of execution caused by a task that fails during execution.

3.2.6 Selectors

The **Selector** task is the complement of the sequence task, [42, p. 335], [16]. The selector task only needs one of its child tasks to execute successfully to succeed and fails only if all its child tasks fail. It thus deals with failures by moving on to the next child task and with success by bailing out with success. The selector task is used for picking the best behavior for the given task. This form of planning that is based on a local set of available behaviors is called **reactive planning**, [42, p. 340]. It is handled by a ordered list of tasks and a fall-back mechanism for when a behavior fails and another needs to be attempted instead. In our previous example, we might have several different ways to acquire food in the Acquire Food task. This scenario is shown in figure 3.8.

Selector

reactive planning

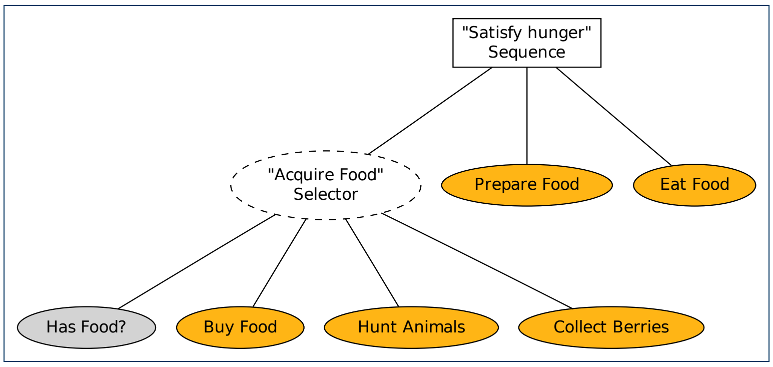


FIGURE 3.8: *Several different approaches to the behavior of acquiring food*

First off, if we already have food available we will just use that, thus solving our problem. If we do not have any food available, we might – as a first approach – simply go to a store and attempt to buy food. If we have no money or the store is closed we need a fall-back solution. One such solution may be to bring down and cook an animal or collect and eat berries. Each task – if it succeeds – fulfills the higher level goal of acquiring food, although the means of doing so is very different. The selector thus enables the agent to choose between different solutions to a single problem. The execution and result of a selector can be thought of as the lazy OR (i.e. || in C and Java) of its child tasks.

Algorithm 3 show the pseudo code for the `execute` function of the selector task. It iterates the list of children, executing each child until one does not fail (i.e. returns "success" or "running") or it has reached the end of the list. The full source code listing can be found in section A.4.5 on page 139 in the appendix.

Algorithm 3 Selector (disregarding running tasks for brevity)

```
1: procedure EXECUTE(agent)
2:     for child in children do
3:         $status \leftarrow child.execute(agent)$
4:         if $status \neq$ FAILURE then
5:             return status
6:         end if
7:     end for
8:     return FAILURE
9: end procedure
```

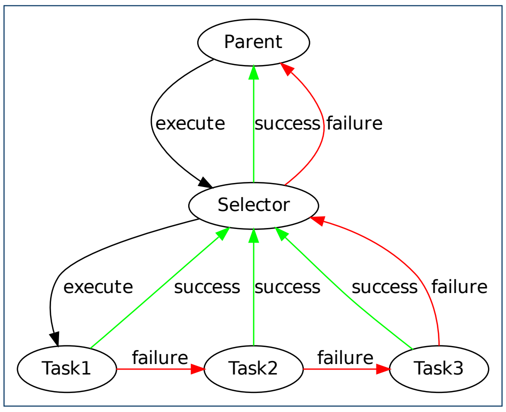
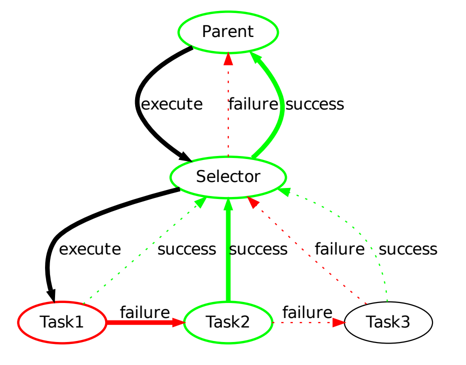


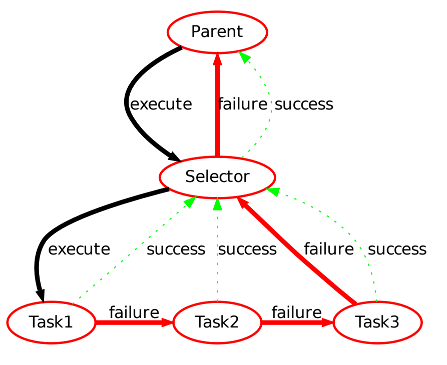
FIGURE 3.9: *State transition diagram of a selector task with three child tasks*

Figure 3.9 illustrates an example selector task, disregarding the "running" transitions for simplicity. Comparing this state transition diagram to the one for the example sequence (figure 3.6) it can be seen that the transition conditions have simply been swapped, effectively inverting the control flow as compared to the sequence task.

Figure 3.10(a) on the next page shows an example execution in which `Task1` fails and `Task2` succeeds. The `Selector` node simply falls back to the next child task (`Task2`) and executes that instead. `Task2` succeeds and the selector has fulfilled its responsibility and can



(a) Execution where `Task1` fails and `Task2` succeed



(b) Execution where all children fail

Figure 3.10: *Two different example executions of the child tasks of a selector task*

thus bail out with success. The execution process is: `Parent` → `Selector` → `Task1` → `Task2` → `Selector` → `Parent`. In figure 3.10(b) we have the situation in which all the child tasks of the selector fail. In that case, the selector has failed to find a solution to the problem and also fails. The execution process is: `Parent` → `Selector` → `Task1` → `Task2` → `Task3` → `Selector` → `Parent`.

3.2.7 Decorators

During the construction of a behavior tree a situation will often arise in which a task needs to be extended with some additional property or functionality. Suppose that we, in our previous example, instead of acquiring food once, we would like to acquire food a varying number of times. We might use this to gather more food when winter is approaching, for instance. How can we create a behavior tree that allows for continuously repeating the `Acquire Food` task? At this point, we really have only one choice for implementing this behavior in the tree and that is to hardcode it directly into the task. This would indeed be a very undesirable solution as we loose the `Acquire Food` primitive and ruin the composite structure.

A more suitable approach would be to decorate the task with a certain property or added functionality without altering the behavior of the original task. A task for doing this is called a **decorator**, [20]. It is an application of the **Decorator pattern**, [33, p. 175]. The pseudo code for the `execution` function of a decorator is shown in algorithm 4.

decorator

Decorator pattern

Algorithm 4 Decorator

```
1: procedure EXECUTE(agent)
2:     Pre-execution code
3:     $status \leftarrow decoratedTask.execute(agent)$
4:     Post-execution code
5:     return $status$
6: end procedure
```

In our example, we could have a decorator task that would loop the execution of the decorated task a number of times. This addition to the example behavior tree is shown in figure 3.11.

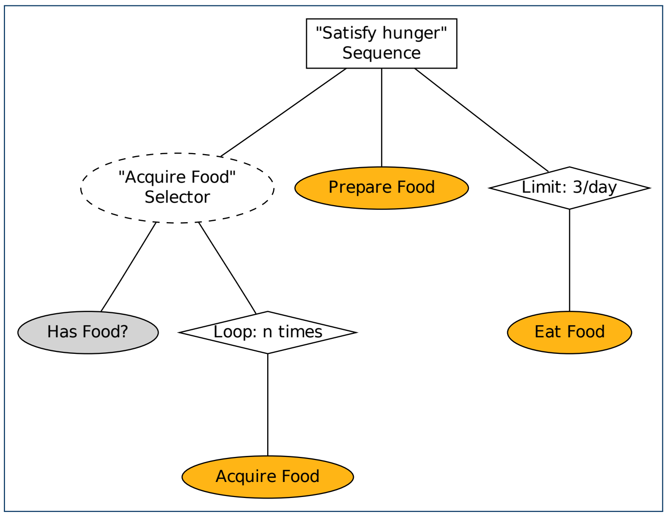


FIGURE 3.11: *The example behavior tree extended with a loop decorator on* `Acquire Food` *and a limitation filter on* `Eat Food`

filter

The most common use of a decorator tasks is as a **filter**, [20]. A filter controls whether the decorated task (or subtree) should be executed or not. In our example, we have a limitation filter on `Eat Food` ensuring that the task is executed no more than three times per day (in the game). The filter effectively prunes the branches of the tree and ensures that the decorated subtree is only searched if the filtering condition is met. Other commonly used filters include timer filters that require a given amount of time between executions and semaphore filters, discussed in the next section. The pseudo code for a filtering decorator is shown in algorithm 5. The full source code listing can be found in section A.4.8 on page 142 in the appendix.

Algorithm 5 Filter decorator

```
procedure EXECUTE(agent)
    if filter condition is met then
        Pre-execution code
        $status \leftarrow decoratedTask.execute(agent)$
        Post-execution code
        return $status$
    end if
    return FAILURE
end procedure
```

3.2.8 Semaphores

Consider a scenario where several agents, who all have the Prepare Food task, share the same stove. This way, only one agent is able to prepare food at a time. This is an instance of a more general problem in computer science, known as a **race condition**, in which we need to avoid that multiple processes or threads use a common resources simultaneously, [54, sec. 6.3.2]. This problem is often handled by using a **mutual exclusion** (mutex) algorithm, [54, p. 496]. In behavior trees, we have a decorator called a **semaphore filter** or **allocator**, [21]. It relies on **semaphores** for handling concurrent resource management, [54, sec. 6.3.3]. In our stove example, we would need a **binary semaphore** (i.e. the semaphore equivalent to a mutex) to handle access to the resource by flagging it as either locked or unlocked. This functionality is shown in figure 3.12, in which the stove resource is locked while the Prepare Food task is executing. For simplicity, we have not handled whether failing to acquire the lock would result in a failure or in simply waiting for the lock to be released.

race condition

mutual exclusion

semaphore filter

allocator

semaphores

binary semaphore

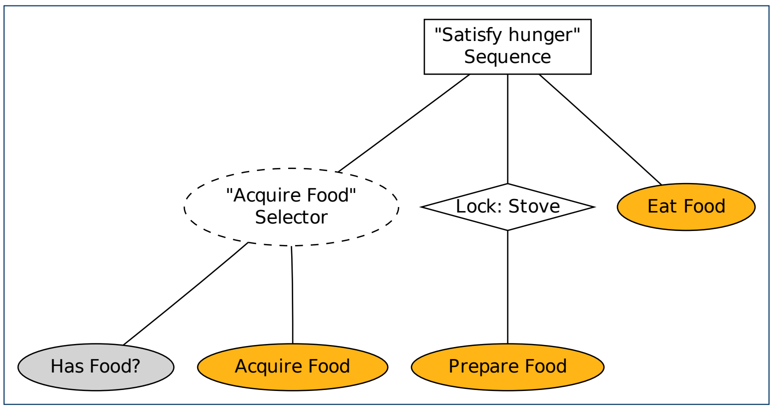


Figure 3.12: *Restricting access to critical or limited resources by using the semaphore decorator*

Suppose the agents had access to a larger stove, which two agents could use simultaneously. We could implement this functionality by a **counting semaphore** allowing two acquisitions of the lock before access to the resource is denied.

counting semaphore

Semaphore filters have many practical applications. For instance, suppose we have an action that simulates that the agent is speaking by playing an audio file and doing lip synchronization. When the agent is speaking the voicebox of the agent should be locked, as should the bones[3] of the mouth. The illusion of realism would be instantly broken if the agent was to make two simultaneous sounds or speaking while inhaling from a cigarette. Thus, such an action would be well suited for decoration by semaphore filters to lock the concerned resources.

Another usage of semaphore filters is for managing group behavior. A semaphore deco-

[3]Here, "bones" represent fictional bones used for handling the animation of game models in techniques such as skeletal animation

rator could be used to limit the number of squad members able to engage the enemy at the same time. This (critically acclaimed) type of squad behavior is detailed in section 2.7.2.

Pseudo code for a semaphore decorator can be found in [42, p. 349-351]. A concrete implementation of a semaphore decorator requires extending the task interface to signal the beginning and end of execution. This is needed to be able to lock (i.e. `acquire()`) and unlock (i.e. `release()`) the resource to signal when it is in use, [42, p. 351]. We provide an extended task interface in listing 3.3 on page 74.

3.2.9 Parallels

Concurrency is an important feature in behavior trees as well as in other AI technologies (section 2.7.2). Consider, for instance, that we have a number of agents who share a stove to prepare food, as in the example in the previous section. How would we go about creating behavior for the agents to wait (doing idle behavior) until the stove is available? One solution is to have condition to check the availability of the stove, and if it is unavailable then to proceed with some idle behavior. However, suppose the idle behavior is a prolonged behavior that takes a long time to finish. It could be smoking a cigarette or watching a television show. With the tasks currently at our disposal, we wouldn't be able to check the availability of the stove until we are finished with the idle behavior. In other word, we have no means of prematurely terminating the idling behaving for proceeding with more important tasks. This type of problem can be handled by the **parallel** task, [14].

parallel

The parallel task is a composite task, running all its child tasks in parallel, that is, they are conceptually executed at the same time. Although the are conceptually executed concurrently, they may in fact be executed one-by-one in a **cooperative scheduling** fashion, [42, p. 351-352]. This way, each child task is still executed in every frame, but there is no overhead (and likewise; no gain) associated with using multiple concurrent threads. Different **policies** can be used for determining the result of the parallel, e.g. to succeed if one child task succeeds, fail if one fails, succeed if all succeed, fail if more than half fail, etc..

cooperative scheduling

policies

In practice, we have two overall applications of parallel tasks. These are to model independent parallel behaviors and as high level assertions.

Independent parallel behaviors is in effect the concept of running subtrees concurrently. This application is mostly focused on doing multiple behaviors simultaneously. For instance, consider an agent who is talking in walkie-talkie and smoking a cigarette while patrolling an area. These behaviors all need to be maintained concurrently, and a way for doing so it to use a parallel task. Each behavior would then have a subtree of its own and each tree would be executed in each frame.

A high level assertion (or "read-only concurrency") application can be illustrated using this example as well. Suppose the patrolling agent spots an enemy. It would then need to react by stopping everything it was doing (talking in walkie-talkie, smoking a cigarette and patrolling the area) and engage the enemy. In other words, we have high level assertions that invalidate an entire subtree. You can use a parallel to monitor (global) assertions/assumptions at a higher level. In this case, the assertion could be `Enemy spotted?` in which case we would bail out of all concurrent behaviors and engage the enemy.

Another example of high-level assertions is the previously mentioned problem of an agent having to wait for the stove to become available. The behavior is illustrated in figure 3.13.

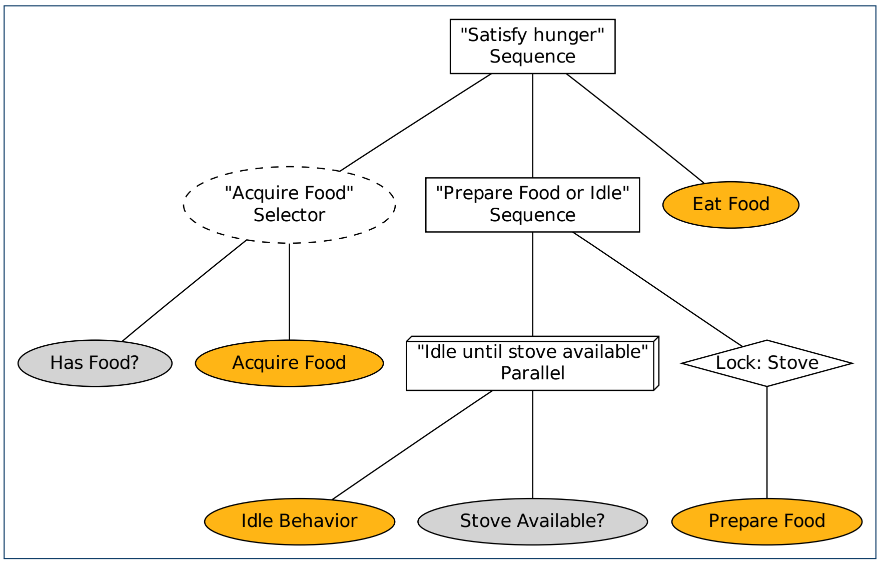


FIGURE 3.13: *Using a parallel task to react to an event in the environment*

We have a never-ending `Idle Behavior` task that runs in parallel with a `Stove Available?` assertion. Because the tasks are run in parallel, the `Stove Available?` assertion is tested in each frame. The assertion will return success the instant the stove is available, thereby returning a success signal to the parent[4] and terminating the `Idle Behavior` task. Because the parent is a sequence task, it will thus proceed to executing its other child task, that is, acquiring the stove resource and proceed with the `Prepare Food` task. When that task terminates successfully (i.e. when the food is prepared), we will proceed to the `Eat Food` task. In the next section we will introduce a new task that allows for a more elegant way of handling such situations.

In the example, we simply assumed that we were able to terminate the parallel behavior and thus also the running `Idle Behavior` task. To be able to bail out of subtrees it is required that we have a graceful way of prematurely terminating the running tasks. For this purpose, we need to extend the interface of tasks to be able to initiate the behavior as well as gracefully terminate it (either normally or prematurely). The extended interface[5] is shown in listing 3.3 on the next page.

[4]Assuming the parallel has a policy that makes it succeed on one succeeding child task and only fail if all child tasks fail

[5]This extended interface could also have been implemented in a new class, `ExtendedTask`, that extended the original `Task` class. However, because we need to be able to invoke these new functions recursively through the tree, the `CompositeTask` class, `Decorator` class etc. would need to be modified to implement `ExtendedTask` instead of `Task`. No class other than `ExtendedTask` would implement `Task`, thus eliminating the need for maintaining it as a separate interface.

```
class Task {
  // Handle the initialization of the task execution
  function start(agent)

  // Handle termination of the task gracefully (normally or prematurely)
  function stop(agent)

  // Execute the task. Returns SUCCESS, FAILURE or RUNNING.
  function execute(agent)
}
```

Listing 3.3: *Extended task interface to enable tasks to perform operations on initialization and termination*

particle system

As an example, take the cigarette smoking behavior. The `start`-function may initialize a **particle system** to create smoke and place a ”smoking“ flag on the agents blackboard (section 2.7.2) for other agents to comment on (e.g. “*Those cigarettes are going to be the death of you*”). The `stop`-function may then involve the agent dropping the cigarette and removing the ”smoking“ flag from its blackboard.

The full source code listing for the parallel task can be found in section A.4.6 on page 140 in the appendix.

3.2.10 Prioritized lists

prioritized lists

activation desire

precondition

The application of **prioritized lists** are very close to that of high level assertions, but uses a more pragmatic approach. It is used for selecting the highest priority *relevant* child task for execution. The concept of priority is derived from the ordering of the children, i.e. the first child is considered the highest-priority task and the last child is considered the lowest priority task. Each child has an **activation desire** for indicating the relevancy of executing that task, [37]. Some systems use a floating point number for determining the relevancy. However, this poses a scalability problem when increasing the number of competing tasks, [37]. Other systems use a binary value for determining relevancy, in which case the test may be considered a **precondition** for the task.

In listing 3.4 we have extended the task interface of listing 3.3 even further. Here, we have added a `precondition`-function that corresponds to a binary relevancy value.

```
class Task {
  // Handle the initialization of the task execution
  function start(agent)

  // Handle termination of the task gracefully (normally or premature)
  function stop(agent)

  // Boolean relevancy value (preconditions for executing the task are met)
  function precondition(agent)

  // Execute the task. Returns SUCCESS, FAILURE or RUNNING.
  function execute(agent)
}
```

Listing 3.4: *Extended task interface that enables the parent to test the relevancy of executing the task*

Using a binary relevancy value, the execution of a prioritized list becomes the same as that of the selector task, with two differences. First, before trying to execute a child task, we need to test if it is relevant for that task to execute. If not, we skip it just like if we had failed an execution of a child task in the selector task. Second, we start by processing the highest priority tasks first in each frame, regardless of a task is already running, [38, p. 48]. This way, if a higher-priority task becomes relevant, we can switch to that task instantaneously and not having to finish the execution of the lower priority task first. This is great for reactive behavior (section 2.2.3), that is, quickly reacting to changes in the environment, but requires that we have a graceful way of prematurely terminating the running task. The pseudo code for the `execute` function of the prioritized list task is shown in algorithm 6. The full source code listing can be found in section A.4.7 on page 141 in the appendix.

Algorithm 6 PrioritizedList

```
1: procedure EXECUTE(agent)
2:     for $child$ in $children$ do
3:         if $child.precondition(agent)$ then
4:             if $child \neq activeChild$ then
5:                 $activeChild.stop(agent)$
6:             end if
7:             $status \leftarrow child.execute(agent)$
8:             if $status =$ RUNNING then
9:                 $activeChild \leftarrow child$
10:                return $status$
11:            else if $status \neq$ FAILURE then
12:                return $status$
13:            end if
14:        end if
15:    end for
16:    return FAILURE
17: end procedure
```

An example of using the prioritized list task is shown in figure 3.14 on the next page. Here we use it to handle the example problem (discussed on page 72) of an agent having to wait for the stove to become available.

We would like for the agent to use the stove as the first priority but failing that, due to it being unavailable, we would like to proceed with an never-ending idle behavior. In each frame, we would then initially test the relevancy of using the stove (i.e. the `precondition` function) and only if this fails, will we proceed with the idling behavior. This will continue

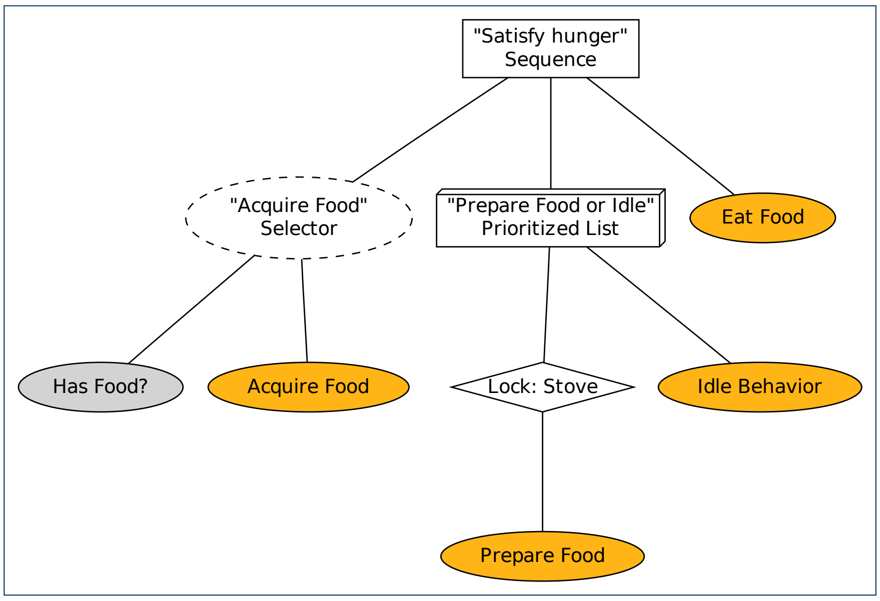


Figure 3.14: *Using a prioritized list to get a concurrent monitoring of the availability of the stove while falling back to performing idle behavior. If the stove becomes available, the idle behavior is prematurely terminated.*

until the stove is available, in which case we would invoke the `stop` function on the already running behavior and instead executing the higher priority task of acquiring the stove resource and then executing the `Prepare Food` task. Only when the `Prepare Food` task successfully terminates will we proceed to the `Eat Food` task.

3.3 Implementing behavior trees

As mentioned in section 3.1, the tasks of behavior trees form a powerful model that is comparable to the basics of a programming language. The tasks provide conditionals and statements in the form of conditions and actions, composition from sequences and `if-then-else` control structures from selectors. Behavior trees are efficient because they have the same control flow as would a well-written script for handling the same behavior, [23, p. 50]. They can even be optimized by using search techniques such as saving and reusing the result of computations of subtrees or using a limitation on the frequency with which behaviors can execute. This effectively prunes the tree as some branches can be simply disregarded, [36]. In concrete implementations of behavior trees, the tasks of the tree would often be references to a look-up table of tasks, [42, p. 369], [23, p. 66]. Such an approach may be considered an application of the **Flyweight pattern**, [33, p. 195], [30, p. 115]. Using a look-up table of tasks the behavior tree is actually not a tree in computer science terminology but rather a **directed acyclic graph** (DAG) as the same task may occupy several different nodes in the graph.

Flyweight pattern

directed acyclic graph

So how do we actually implement game AI as a behavior tree? Starting from scratch, we first need to implement the interface for the `Task` class. We also need to implement the composite `Sequence` and `Selector` classes to be able to combine the tasks in a hierarchical fashion. Which concrete conditions and actions we need depends on the game (that is, the task environment) we design the game AI for. Generally, the tasks represent the observations and actions that the game agents are able to perform in the environment. We will go into further detail about both the technical considerations and the design considerations of implementing behavior trees in section 4.5.

3.4 Designing behavior trees

When designing game AI as behavior trees it is important to embrace patterns in the design. During the design process some common patterns tend to emerge. It is an important design challenge to notice these recurring structures, as they are candidates for being factored out as high-level tasks in the tree. For instance, consider the sequence task in figure 3.5 having the child tasks `Acquire Food`, `Prepare Food` and `Eat Food`. If we need to create a new task to encapsulate the behavior of making dinner for another agent, this might be comprised of the subtasks `Acquire Food`, `Prepare Food` and `Serve Food`. The first two tasks would then occur in both subtrees and would be obvious candidates for composing into a higher level task. In figure 3.15 we illustrate two different subtrees, both of which has a look-up decorator task pointing to the same decorated task. The decorated task, `Acquire and Prepare Food`, is stored in a look-up table and is references through the decorators.

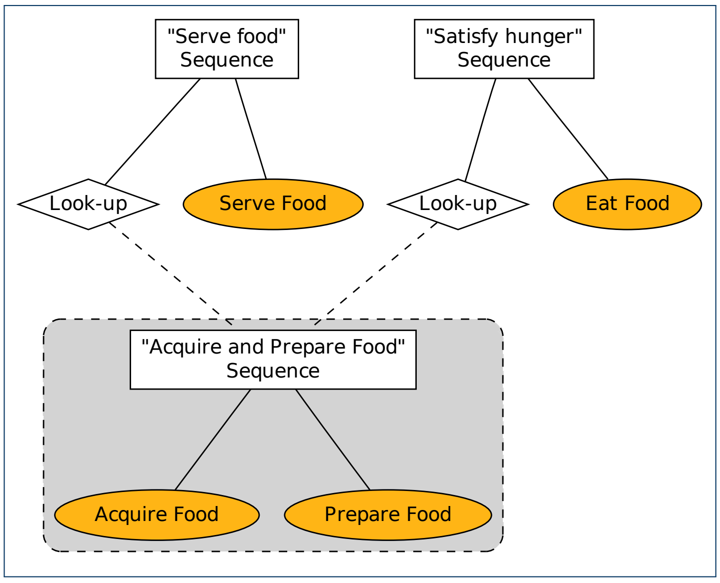


Figure 3.15: *Illustration of the concept of look-up decorators*

The simplicity of this example suggests there is not much to be gained from grouping the two tasks. However, it provides both design advantages as well as technical advantages. The overall design becomes simpler and more intuitive as the tasks become continuously higher-level, thus allowing designers to focus on high-level behavior rather than the details of the lower levels. Further, it helps the designers to mix and match between the available high-level task rather than having to build up behaviors from primitive tasks each time. With regards to the technical aspects, the trees become more maintainable and extendable. If, for instance, we need to add an `Is Poisonous?` condition between the `Acquire Food` and the `Prepare Food` tasks in the behavior tree in figure 3.15, we can simply do so in the higher level `Acquire and Prepare Food` task, and the change will be reflected throughout the tree.

3.5 In conclusion

In section 2.7 we looked, among other things, at the three techniques from which behavior trees have inherited some properties. Behavior trees are a synthesis of HFSMs, planners and scripting (figure 3.16), [42, p. 334].

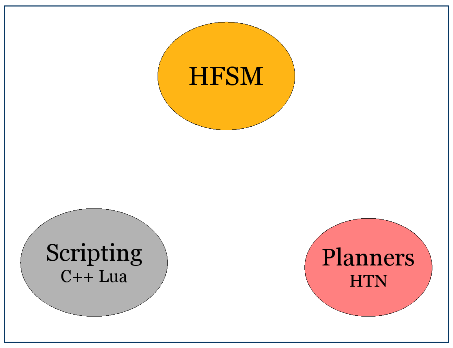


FIGURE 3.16: *The techniques from which behavior trees has been synthesized, [23, p. 14]*

In this chapter we have examined behavior trees more closely. Behavior trees are a compromise between HFSM, planners and scripting. However, when compromises are made, some things that might have been desired are often lost. Let us compare the properties of behavior trees against the techniques from which it was synthesized. The following list provides a short summery of the major advantages or disadvantages of behavior trees and its related techniques. The "+" specifies an advantage, and the "−" a disadvantage.

- **Hierarchical finite state machines**
  - + Great for designers, as the concept behind them is simple and intuitive. It is easy to visualize different states and their connection to other states.
  - + Provides low-level reactive control. It is easy to react to changes in the environment.
  - − Takes a lot of work to implement [42, sec. 5.3.9]. They can be difficult for programmers to implement and for designers to use.
  - − Are hard to make goal-directed.
- **Planners**
  - + Have modular, self-contained, actions.
  - + Are autonomous and goal directed by searching for a path through the available action to a solution.

  – Do not do well in practice, [23, p. 20]. Difficult to integrate into game code.
  – Are not good at handling errors that occur in the plan.

- **Scripting**

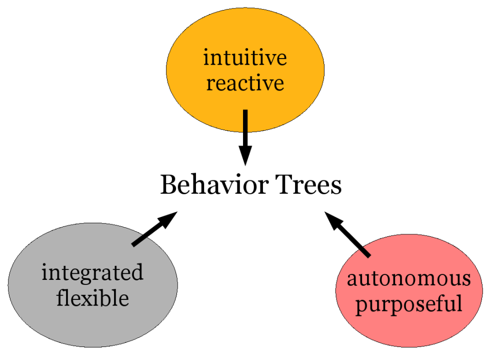
  + Is flexible. Scripting languages are Turing complete and can compute anything.
  + Is well-known. Scripting has widespread usage throughout the game industry.
  + Is data-driven. Helps with the separation of the game engine and game content.
  – Is difficult to introspect. It is not intuitive what the resulting behavior will be.
  – Is not well suited for designers.

- **Behavior trees**

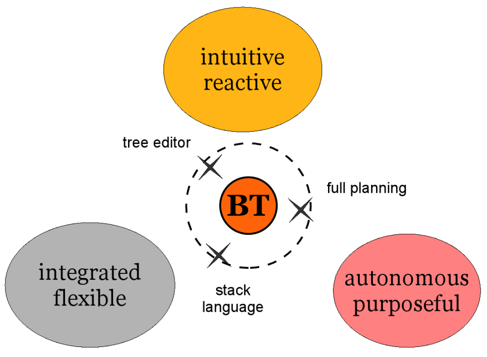
  + Are the best compromise between the HFSMs, planners and scripting, [23, p. 16]. Works well in practice.
  + Good at local replanning, i.e. reactive planning.
  – Are not as good as any of the other approaches individually.

To sum up, HFSMs are *intuitive* and *reactive*, planners are *autonomous* and *purposeful* and scripting is *flexible* and *data-driven*, [23, p. 21-22]. It is exactly these qualities that the behavior tree technique attempts to synthesize (figure 3.17(a)).

Behavior trees are an improvement over HFSM in that the game AI programmer and the game designer do not have to think about the transitions in the FSMs. Behaviors are simply put together using high level building blocks such as sequences and selectors. Behavior trees are an improvement over planners because they are very simple to implement. However, behavior trees can only provide reactive planning, and not full look-ahead planning, meaning that some problems cannot be prevented elegantly. Behavior trees are an improvement over scripts because they make it easier to handle errors. The scripts can be used in tree without the need for checking a lot of preconditions. It can simply use a selector to automatically fall back on other solutions.



(a) Desirable properties of behavior trees



(b) Some extensions for specializing behavior trees

FIGURE 3.17: *Illustration of properties of behavior trees*

Behavior trees provide a good solution for handling many of the practical problems that are encountered in game development. Although it may lack some specialized features, such as full planning, it will often be an acceptable middle ground for both game AI programmers and game designers. Like FSMs (section 2.7.2), behavior trees can also be extended to suit more specialized needs. Figure 3.17(b) on the previous page illustrates some directions in which behavior trees can be extended for serving more specific purposes. Section 5.2 on page 120 list some ideas on how behavior trees may be extended.

In the next chapter we will extend behavior trees in the direction of a more *intuitive* & *reactive/integrated* & *flexible* solution (figure 3.17(b)). We will discuss the development of an editor for making it easier to design the desired behavior using behavior trees.

Chapter 4

Designing the behavior tree workbench

The “Behavior Tree Workbench” is a system designed to make it easier to model character-based game AI. It bridges the gap between the game AI implementation and the actual behavior of the agent in a game.

In other words, it creates the link between the generating mechanism and observed behavior of the game AI. The generating mechanism is the concrete implementation of actions of the agent and the decision making process for choosing which action to perform next. The observed behavior is how the end user, i.e. the player, perceives the behavior in the game. It is a reflection of the underlaying game AI technique, but even more so of the design choices of the game designer. A game may have a state of the art AI technique, but that is immaterial if the game designers or the tools at their disposal are not able to express the desired behavior:

> “*In the end, if an NPC looks, talks and walks like a duck, then it will probably be perceived as a duck, regardless of its actual inner workings,* [38, *p. 47*]”

The purpose of the behavior workbench is first and foremost to ease the behavior modeling process of the game designers. However, we need to acknowledge that without the game AI programmers there would not be anything to design for the game designers. As mentioned in section 2.6.2, the game AI programmers build the platform on top of which the behavior is designed. So, for the workbench to be part of the content creation pipeline, it must not only facilitate the game designers, but also the game AI programmers. That is, it must create the link between the game world and the workbench.

The behavior tree workbench is made up of three components: the game component, the behavior tree framework and the editor component. The game component (figure 4.1 on the next page, left) is the concrete game for which we want to design behavior for agents. The behavior tree framework is in the game component and handles the execution of game AI

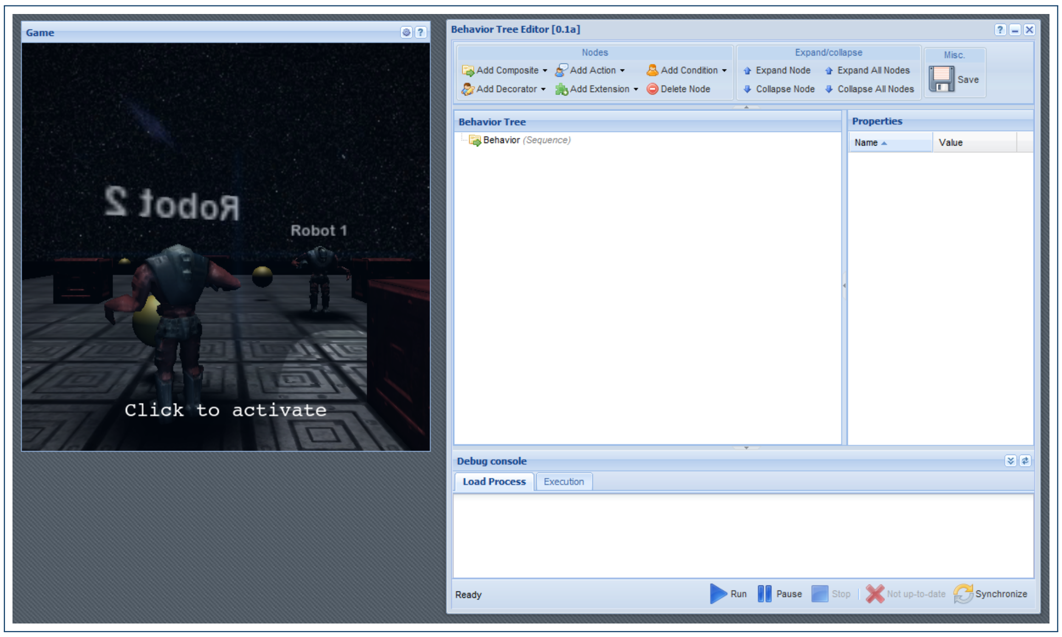


FIGURE 4.1: *The Behavior Tree Workbench. On the left is the game component and on the right is the editor component.*

using behavior trees and facilitates the communication between the game component and the editor component. The editor component (figure 4.1, right) makes it possible to design the actual behavior of agents by combining tasks into a behavior tree.

The workbench needs to facilitate the workflow of both the game AI programmer and the game designer. As such, it needs to meet a lot of requirements from both parties. Most importantly for the game AI programmer, there needs to be a framework for easily interfacing between the editor and the game. For the game designer, it needs to be easy to create, tweak and test behavior.

modding

Finally, in recent years, the concept of **modding** has gained a footing and is considered a selling point for many games, [65]. Modding is slang for “modifying” and, w.r.t. games, refers to the concept of modifying the original games with the purpose of creating a different gaming experience or adding more custom content. Modding is usually done by editing directly in the source code and resources of the games, but some games provide entire suites of tools for making modifications. A secondary goal for the workbench is to function as a modding tool for players to experiment with.

In this chapter we will go into detail with the design and implementation of the behavior tree framework for interfacing with the editor, the editor itself, an example game implementation and the flow of data between components. First, we will take a brief look at the system as a whole, and then go into more detail with the individual components in the following sections.

4.1 Description of the workbench

The behavior tree workbench consists of two interacting components: the game component (which includes the behavior tree framework) and the editor component. The components are illustrated in the figure below. The two components have different responsibilities which will be detailed in the next sections.

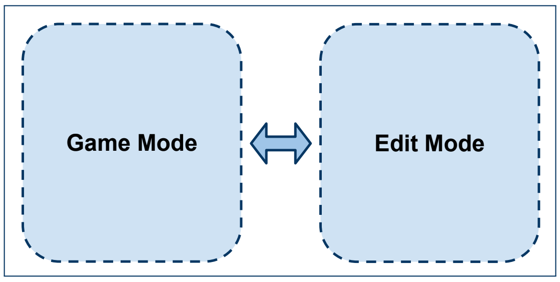


FIGURE 4.2: *Illustration of the conceptual components of the behavior tree workbench*

There are numerous reasons for splitting the workbench into two parts. We want the editor to be data-driven and more general-purpose than many of the current tools for creating game AI. For this purpose, we need a clean separation between the game and the editor. This is, in concept, an application of the **Strategy pattern** [33, p. 315], allowing components to be replaced by others with different functionality as long as the interface remains the same.

Strategy pattern

The editor knows nothing about the game other than the data sent by the game at runtime. This means we can replace the game with an arbitrary other system as long as the same interface is implemented. The game do not know anything about the editor either. The editor can thus also be replace with another component with the same interface. Additionally, this results in a game AI that is data-driven and will be more easily maintainable and extendable.

4.1.1 Web-based workbench

We have chosen to make the workbench web-based. The tools for creating web-based applications are somewhat cumbersome compared to those available for creating standard application. However, having a web-based workbench has a lot of advantages. It is highly accessible as any device with a browser will be able to access and use it. There is also no software distribution, installation or updating required, so the workbench will always be available in its newest version.

The concrete technology we use for the game component is a **Unity**[1] (section 4.3.1) object and the editor component is an **Ext JS**[2] (section 4.4.3) framework (figure 4.3 on the following page).

Unity

Ext JS

[1]http://unity3d.com

[2]http://sencha.com/products/js

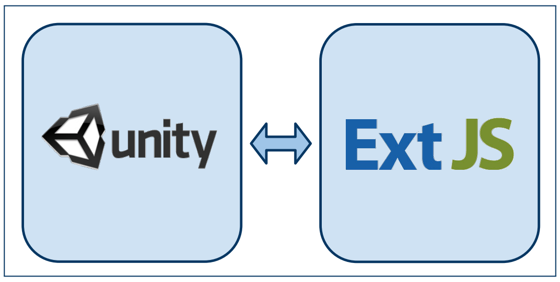


FIGURE 4.3: *The concrete web-based components of the behavior tree workbench*

Unity is at its core a multiplatform all-round game development tool (section 4.3.1). Unity is a highly popular[3] tool for game development enthusiasts as well as professionals. It can be used for all aspects and phases of game development. From importing textures and 3D models, to scripting game logic and game AI, to handling terrain, shaders, lights, physics, audio, networking and much more. Finally, it can be used to deploy the finished game to a variety of different platforms include the PC, Mac and the web. Additional licenses can be bought to deploy to the Wii as well as iPhone and Android cell phones.

Unity Web Player

We have chosen Unity primarily for its ability to publish web-based games using the **Unity Web Player** Plug-in, [4, sec. "Features – Deployment"]. Any game created in Unity can be embedded and played in a web page through a browser[4]. The size of the plug-in is small (about 3 MB) and it can be installed without restarting the browser. The plug-in also enables communication between the Unity web-player object and the web site it is embedded in. This is of great importance for our purpose as it provides a painless mechanism for exchanging data between the game and the editor (section A.2).

graphical user interface

For the editor we have chosen to use JavaScript and Ext JS (section 4.4.3). Ext JS is an open-source framework for creating rich internet applications[5]. It provides high-performance application-like components such as forms, controls and widgets which enable the creation of the **graphical user interface** (GUI) of the editor, [2].

In order to illustrate which features are needed in the behavior tree framework and in the editor, let's start with a example use case. We will focus on a use case that could be found in a real game, while still keeping the example simple to clearly illustrate its points. The use case will be used in our further discussion of the responsibilities of both the behavior tree framework and the editor. We will define the use case in section 4.2.

[3]In June 2010, Unity surpassed the 200,000 developers milestone and the Unity Web Player had surpassed 30 million total installations, [4, sec. "Company - Fast Facts"]

[4]The plug-in works on all modern browsers including Internet Explorer, Firefox, Safari, and most Mozilla-based browsers, [4, sec. "Features – Deployment"]

[5]All major web browsers are supported, including: Internet Explorer 6+, FireFox 1.5+ (PC, Mac), Safari 3+, Chrome 3+ and Opera 9+ (PC, Mac) [2]

4.1.2 Purpose of the workbench

The behavior tree workbench is designed for a specific purpose. That purpose is to enable us to answer whether this approach for implementing and designing game AI is a practicable solution.

The workbench is thus created as a tool for experimenting with the problems that we are investigating. The work in creating and using the workbench will help us in trying out the concepts in practice. The primary experiences we want to extract from this work include:

- Implementing a game environment for testing observed agent behavior
- Implementing a behavior tree technique for handling game AI
- Creating an editor for designing behavior
- Designing a framework that enables a coupling between the game and the editor
- Integrating the game into the editor
- Experimenting with the overall workflow of the process of designing agent behavior

The behavior tree workbench is designed for the purpose of experimentation. That is, it is merely a *prototype*. We do not attempt for the workbench to compete against industrial strength editors such as Unity (section 4.3.1). We have done some considerations as how the workbench might be improved and extending, however. Some of these ideas are listed in section 5.2.

4.2 Game use case

A type of behavior that is used in many games is **patrolling** [30, p. 203]. Patrolling is the behavior of going the rounds of a route with the purpose of guarding or scouting that area (figure 4.4).

patrolling

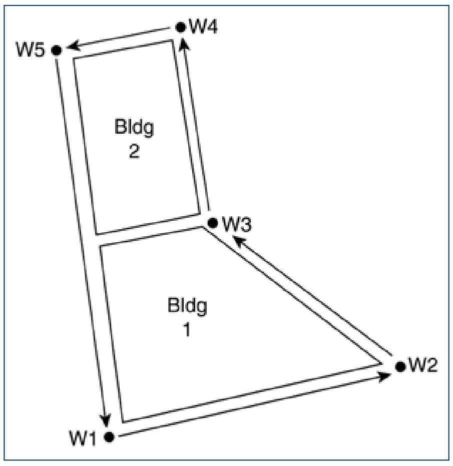


FIGURE 4.4: *Patrolling around two buildings by moving between points $W_1, \dots, W_5$ in sequence [30, p. 204]*

The process of patrolling is typically handled by using **waypoints**, which are virtual

waypoints

beacons added to the game environment, [72, p. 42]. The waypoints can be used to guide the navigation of agents. They are often enriched with information about their immediate surroundings, allowing agents to make more intelligent decisions about their movement. A waypoint might signal the position of a shielded area, for instance, giving the agent a place to seek cover. In the simplest case a waypoint is nothing but a point in space. Patrolling can be defined as the behavior causing the agent to move between the points in space specified by a list of waypoints.

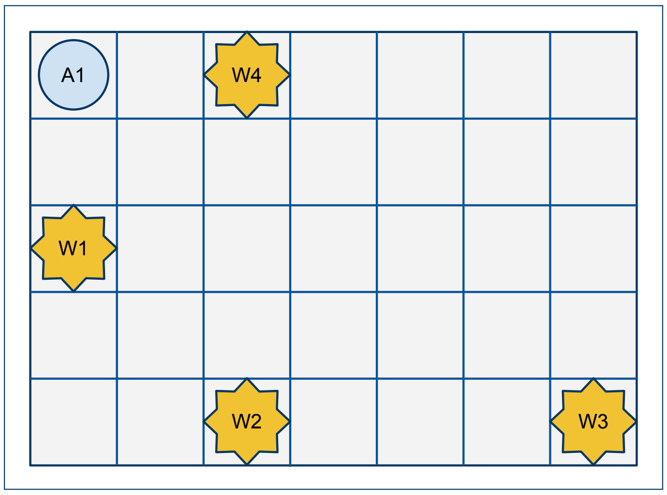


FIGURE 4.5: *The use case environment with an agent and four waypoints*

What is needed to set up a patrolling behavior for an agent in a game environment? To be able to patrol, we need an agent and at least two waypoints (go patrol between). Consider the environment illustrated in figure 4.5. The figure shows a top-down view of a map representing the game objects in the example environment. We have an agent, A_1, and four waypoints, $W_1, \ldots, W_4$. For simplicity, we have divided the environment into rectangles so we obtain a grid and restricted the agent movement to vertical, horizontal and diagonal in the grid.

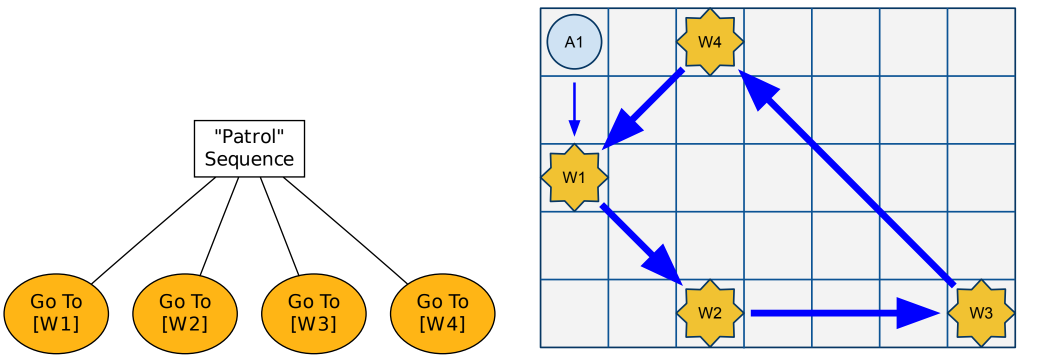
For a more precise definition of the task environment for the agent we need to define the properties hereof (section 2.3). The task environment that we have set up is *fully observable* (the entire state is visible), *deterministic* (only actions of the agent change the state of the environment), *sequential* (the ordering of actions matter), *static* (the state of the environment remains unchanged while the agent decides its next action), *discrete* (the state of the environment changes discretely) and *single agent*.

Now that we have defined the task environment, let us define the properties of the agent. The PEAS description for agent A_1 is laid out in table 4.6 on the next page. The goal of the agent is simply to move between the waypoints in the grid. For doing so, it needs to know its own location in the grid and needs to be able to move to an adjacent grid cell.

We can now construct a behavior tree that implements the desired behavior of the agent. The desired behavior is the patrolling behavior which cyclically visits waypoints $W_1, \ldots, W_4$. A behavior tree implementing this behavior is shown in figure 4.7(a) on the facing page.

Agent	**Performance Measure**	**Environment**	**Actuators**	**Sensors**
A_1	Patrol along way-points	2D grid	Move up, down, left, right, diagonal	Location in grid

Figure 4.6: *PEAS description for agent A_1*



(a) Behavior tree for the patrolling behavior (b) The corresponding behavior in the environment

Figure 4.7: *Two views on an example behavior*

The behavior tree consists of only two types of nodes: a sequence node named `Patrol` and an action node named `Go To`. The action is assumed to somehow move the agent to the specified location, in this case provided by a waypoint. The action tasks have been parameterized with a specific waypoint as argument (in square brackets). If we execute the behavior tree with A_1 as agent argument, it will result in the agent moving to waypoint W_1, then W_2, then W_3 and finally W_4 after which the process will repeat. The movement of A_1 is illustrated in figure 4.7(b) as indicated by the blue arrows. The fat arrows represent the movement of the agent through an execution of the entire `Patrol`-subtree.

The illustrated behavior is what we wanted for our use case. Now we can investigate which features are required in the editor and in the behavior tree framework of the game for creating this behavior. We will start by looking at the game component of the workbench, then moving on to the behavior tree framework and finally combining them with the editor to form the entire workbench.

4.3 Game component

The game component (figure 4.8 on the following page) serves two major purposes. The primary purpose is as the actual implementation of the game. This exposes the resources of the game, i.e. the game models, gameplay, agent behaviors etc. (figure 4.9 on the next page).

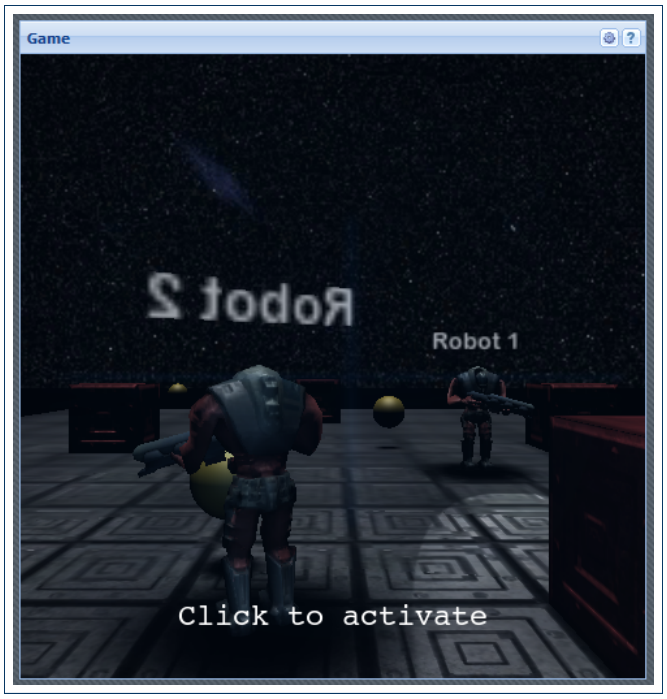


FIGURE 4.8: *The game component in the behavior tree workbench*

It allows the game designer to play the game to test the perceived behavior of the agents. In our example use case (section 4.2), we could run the game to see whether the agent would patrol between the waypoints, as was the desired behavior. That is, it is a tool for debugging the game AI.

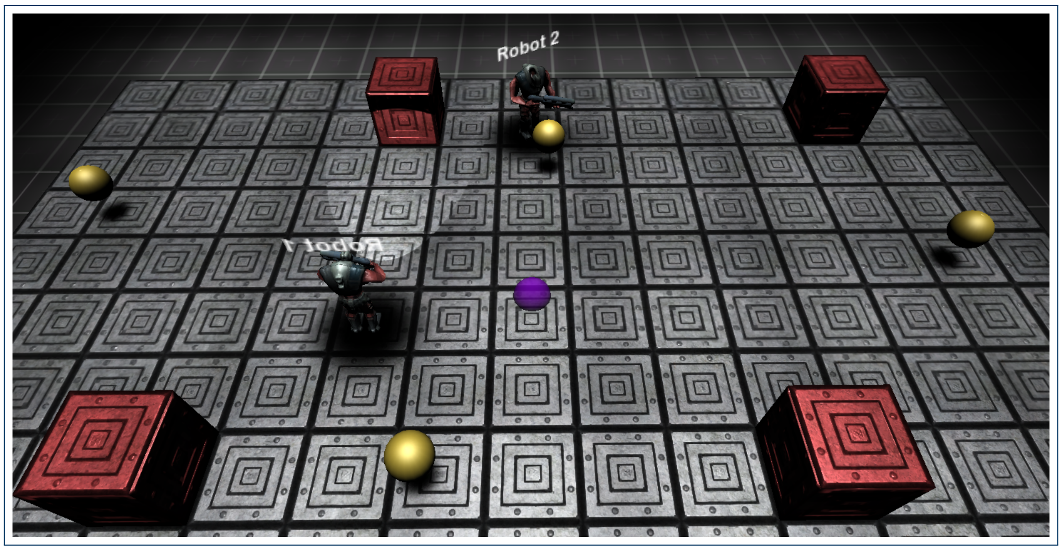
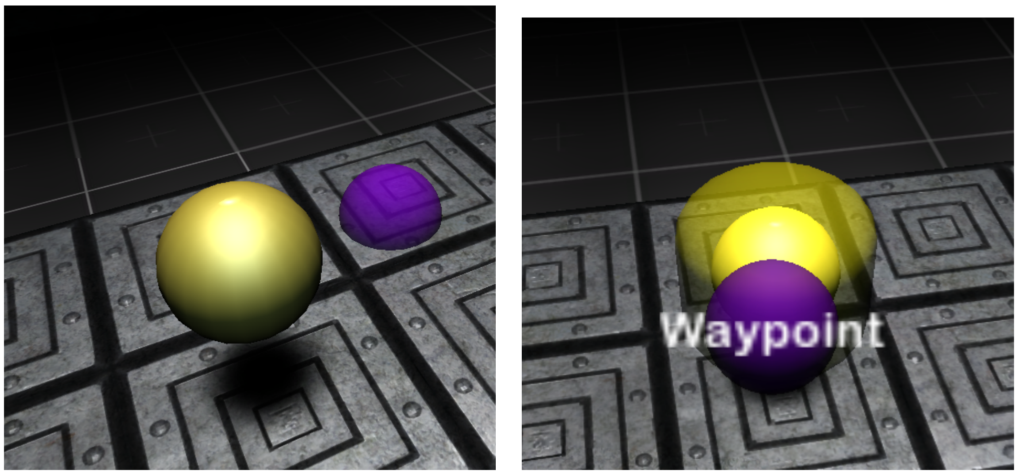


FIGURE 4.9: *An example game. This scene shows waypoints, obstacles and two agents*

The secondary purpose is to help making editing of agent behavior more intuitive and convenient. That is, integrating the game component into the editor. We need to couple the game component with the editor by exposing the game data, representing the resources of the game, to the editor. Without this data the editor would not know which agents, actions etc. are available. However, the game component can also be used as a great tool for easing the editing process because it exposes its resources visually to the game designer. One usage hereof is to select game objects directly from within the game rather than having to rely on a textual representation of objects in the editor.



(a) A waypoint (b) Moving the cursor over the waypoint highlights it for selection

Figure 4.10: *An example of the integration of the game into the editor. In (b) we select a waypoint from within the running game*

For instance, suppose we need to pick the argument for the waypoint parameter of the `Go To` action in the editor. In many tools, we would be presented with a list of waypoints to choose from. However, when designing the behavior, this is not actually what we want. We would like to make the agent go to a specific place, as represented by a waypoint. In other words, the textual description is not very helpful, but it would be to be able to visually pick the desired waypoint from within the actual running game. Figure 4.10(a) shows a waypoint in the game component and figure 4.10(b) shows the waypoint being selected from within the game (the cursor is the purple sphere labeled “Waypoint”).

4.3.1 Game implementation

We have made a concrete implementation of an example game, similar to that of the example use case (section 4.2). The reason for creating a concrete example is to be able to test the workbench. Figure 4.11 on the next page shows the concrete game scene, which is basically a slightly more involved version of the environment from the example use case shown in figure 4.5 on page 86. There is no winning condition or actual gameplay in the game scene;

it is simply a basic scene that makes it possible to test the workbench system.

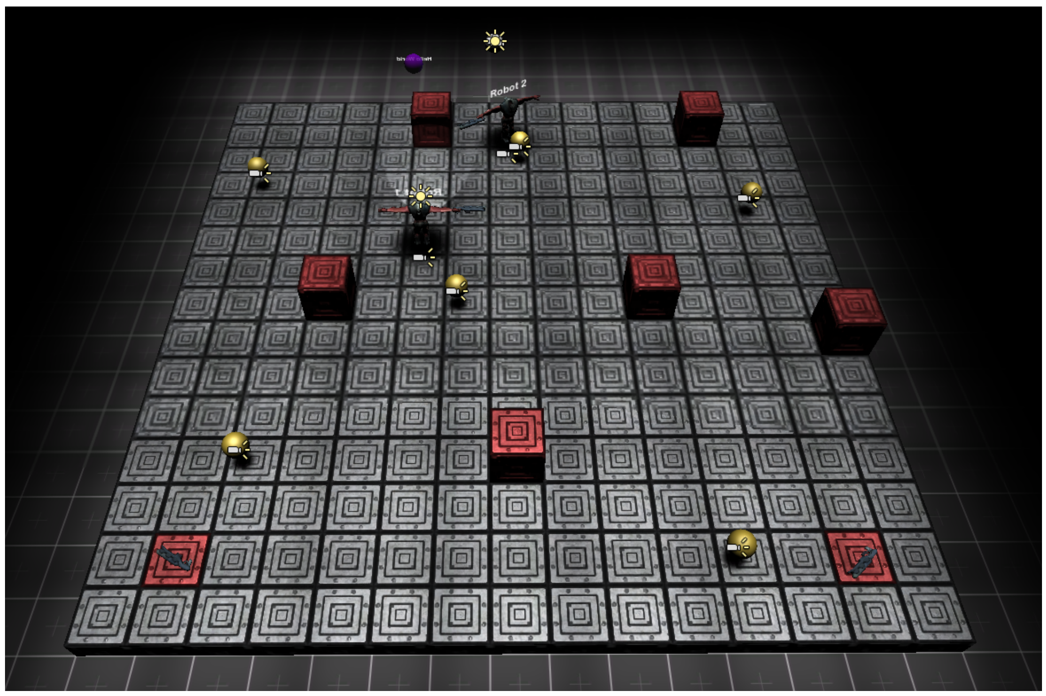


FIGURE 4.11: *The game scene as seen from above. There a two agents (robots), six waypoints (yellow spheres) and six obstacles (red cubes).*

The concrete implementation of the game has been created using Unity (figure 4.12) for several reasons.

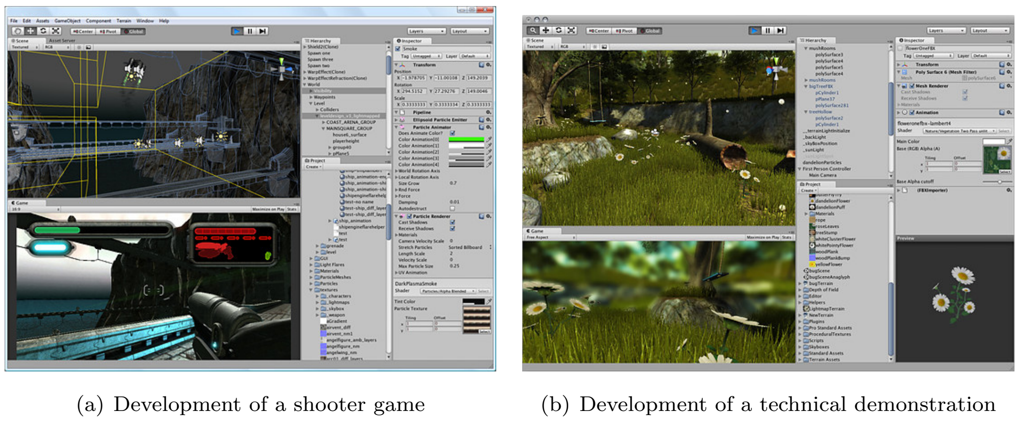
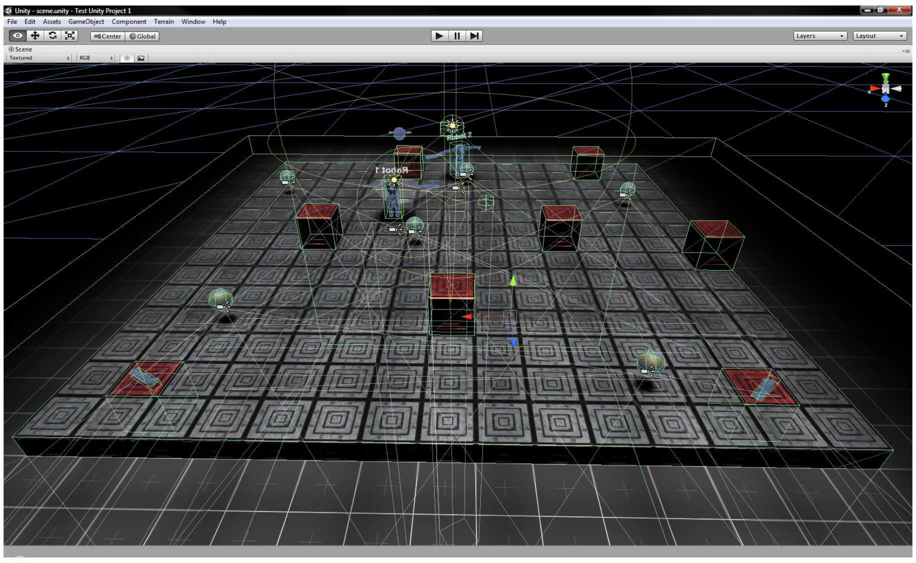

(a) Development of a shooter game (b) Development of a technical demonstration

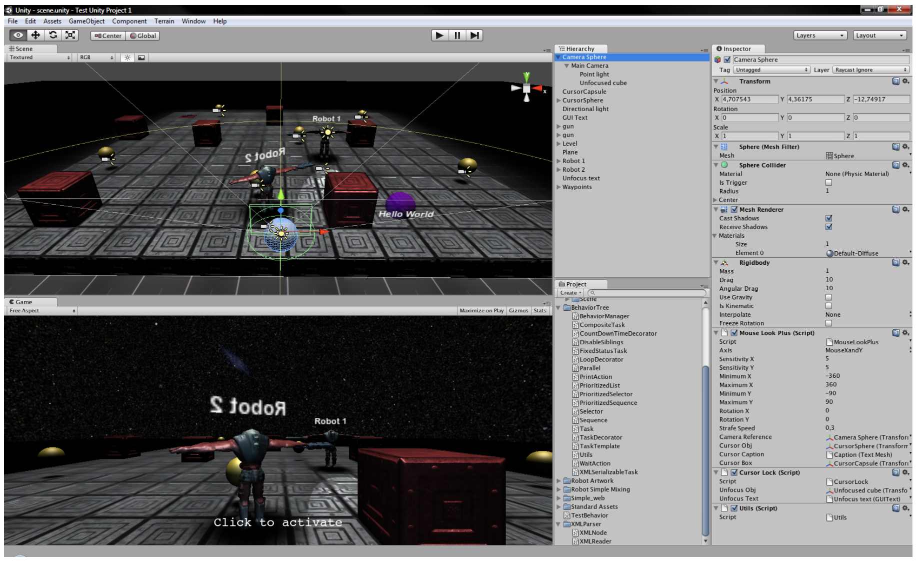
FIGURE 4.12: *Typical screens from the Unity editor environment, [4, sec. "Features – Editor"]*

First, Unity is free, well-documented and easy to use for creating fast prototype games.

Secondly, Unity is able to deploy games directly to the web via a special browser plug-in featuring a simple way of communicating with JavaScript code on the website in which it is embedded (section A.2). Our goal for the game is simply to create a prototype to test the applicability of our proposed approach to the process of designing and editing behavior. As such, the Unity tool fully satisfies our needs. Figure 4.13 shows different views on the game scene, i.e. the concrete game environment, inside the Unity tool.


Unity - scene.unity - Test Unity Project 1
File Edit Assets GameObject Component Terrain Window Help
Center Global
Layers
Layout
Scene
Textured
RGB
Robot 1
Robot 2

(a) The game scene inside Unity


Unity - scene.unity - Test Unity Project 1
File Edit Assets GameObject Component Terrain Window Help
Scene
Game
Hierarchy
Project
Inspector
Robot 1
Robot 2
Hello World
Click to activate

(b) Different perspectives on the game scene

FIGURE 4.13: *Screens from the development of the game scene using Unity*

As Unity is a fully-featured game development tool, it also has built-in support for creating game AI. In Unity all game AI needs to be implemented by scripting (section 2.7.5) using a scripting language. Unitys scripting system is used to handle both game AI but also all other game logic. Some of the key selling points about their scripting system are, [4, sec. "Features – Scripting"]:

- *Visual Properties*, i.e. that variables and references within the scripts can be visually edited directly from the Editor
- *Handling Events*, i.e. that scripts are able to handle both local and global events
- *Cooperative Multi-Threading*, i.e. that threads can "yield" (wait) without affecting other threads
- *Flexible and Easy*, i.e. it is easy to manipulate game objects and change their properties and that game objects can be referenced by e.g. name, hierarchy, tags or proximity

The developers of Unity seem to agree in that a visual indication of properties may help to ease the design process, as we discussed in section 4.3. Although reacting to global events may also be applicable to other game logic, it is certainly true that game AI needs to be able to (re)act on global events. We discussed the need for this in section 3.2.9 and section 3.2.10 in which we mentioned high-level parallel assertions and prioritized lists as ways of reacting to global events in behavior trees. Unity also acknowledges the need for concurrency, to which end we presented a parallel node for adding concurrency to behavior trees (section 3.2.9). Finally, they state that their scripting system is "flexible and easy". Other than claiming that their system makes it easy to manipulate and reference game objects, they do not elaborate on how it is flexible and easy to use.

Let us look at how to implement agent behavior using Unity as the game AI tool. Unity allows developers to script in JavaScript, Boo or C#. Each of them has equal expressive power, i.e. they are all Turing complete, and are equally fast, [4, "Features – Scripting"]. We have chosen to use JavaScript as the primary language for scripting. We do, however, use a few Boo data structures, mostly dictionaries (or "hashes" in Boo terminology). We have choosen not to use Boo as it is a relatively new language (from 2003) and neither the language nor its documentation is sufficiently mature. We have chosen JavaScript over C# because JavaScript is the language of choice for most Unity developers and has a less verbose syntax than C#. The following code listing shows a behavior tree constructed in JavaScript in Unity.

```
var patrollingSequence : Sequence = Sequence();
patrollingSequence.addComponent(new GoToAction(Vector3(10.5, 3.2, 12.0)));
patrollingSequence.addComponent(new GoToAction(Vector3(-1.5, 3.2, 12.0)));
patrollingSequence.addComponent(new GoToAction(Vector3(-3.5, 3.2, -2.0)));
patrollingSequence.addComponent(new GoToAction(Vector3(12.0, 3.2, 2.0)));

agent.behaviorTree = patrollingSequence;
```

The behavior tree that we construct is equivalent to that of figure 4.7(a) from the example use case. It consists of a sequence named `patrollingSequence` having four child tasks specified by `GoToAction` tasks. Each `GoToAction` task takes as argument the destination to which the agent should go upon execution of the task.

The code itself is easily understandable and implementable. However, the code becomes steadily harder to understand as the complexity of behavior trees increase. As the code is written in a scripting language, the behavior tree may to some extend be changed at runtime. Had Unity not relied on scripts, the game would likely had to be shut down, the code recompiled and the game restarted in order to test the change.

The game that is deployed to the web page has several purposes. It is both the actual game that can be played, but its purpose is also to be integrated into the editor and thereby easing the designing and testing processes. Although these different purposes have some overlapping functionality, they are also very different. A player has no need for being able to select agents and waypoints while playing the game, for instance. Likewise, the game designer has no need for a high-score system when testing the behavior. The concrete implementation of the game is a combination of the functionality for satisfying these purposes. The main focus is on the aspects of editing and testing as these are processes we want to investigate. Our game component thus acts primarily as a game environment providing handles for integration into the editor. For more discussion on this subject see section 5.2.3 on page 124.

The Unity Web Player has functionality for communicating between the Unity object and the web page it is embedded in. We use this functionality to exchange data between the game component, i.e. the Unity Web Player, and the editor. Section A.2 in the appendix describes how to implement this communication.

4.4 The editor component

In this section we will discuss the responsibilities of the editor, we will discuss the purpose of the editor, how it is supposed to work and its required features along with considerations about additional features.

> “*While tasks* [in behavior trees] *of all kinds can contain arbitrarily complex code, the most flexibility is provided if each task can be broken into the smallest parts that can usefully be composed. This is especially so because, while powerful just as a programming idiom, behavior trees really shine when coupled with a graphical user interface (GUI) to edit the trees. That way, designers, technical artists and level designers can potentially author complex AI behavior, [42, p. 334]*”

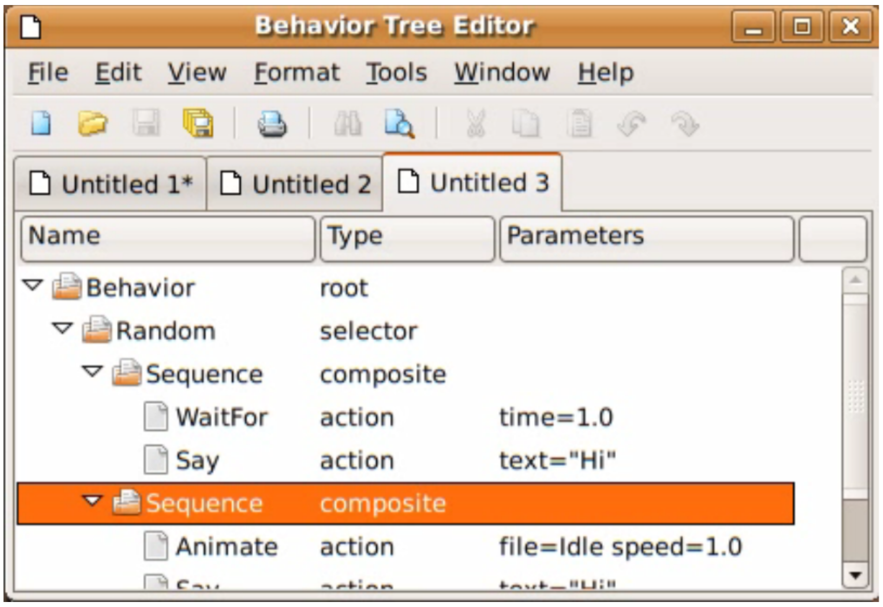
Compared to a scripting-based approach, such as that of Unity (section 4.3.1) for instance, the goal is to make the design process easier, and to help the designers visualize the range of possible behaviors and their results in the game.

4.4.1 Features of the editor

The overall purpose of the overall workbench is to enable the designer to rapidly design, test and tweak the behavior of agents. What functionality do we actually require from the editor component in particular? We discussed some of the technical requirement in section 4.5.2 on page 106, and we obviously need to take care of those. However, there is also a long list of other features which are either requirements or which simply would be convenient to have.

We want to design the editor to be intuitive to use. A big help in achieving this comes from our choice of using a behavior tree technique to model the behavior of the agents. In behavior trees we have very well-defined modular tasks which we can in simple manner utilize and we can react to whether the execution hereof fails or succeeds. The main idea is to build up a tool in which the tasks act as self-contained building blocks (sort of like LEGO bricks) that can be dragged and dropped to combine them into a behavior tree. Once a "building block" has been dropped into place, it should be easy to pick it up and relocate or discard it if testing showed that it was in a wrong place in the tree. Further, the properties of a "building block" should be exposed and easy to modify and test for verification. The collection of building blocks that are available should be those tasks available for the agent in question.

Before we begin discussing the concrete features of the editor, let us examine how others have approached the task of creating a behavior tree editor. The figure below shows one approach to designing a behavior tree editor and section A.3 on page 131 in the appendix shows the graphical user interface of four other behavior tree editors.



Name	Type	Parameters
Behavior	root	
Random	selector	
Sequence	composite	
WaitFor	action	time=1.0
Say	action	text="Hi"
Sequence	composite	
Animate	action	file=Idle speed=1.0
Say	action	text="Hi"

FIGURE 4.14: *User interface of the "Behavior Tree Editor" showcasing how a behavior tree editor might look, [22]*

Figure 4.14 has no actual editor functionality, but is simply an example of how a behavior tree editor *might* look. The editors shown in figure A.5 and A.6 are hobby projects aimed at editing behavior for the “Visual 3D” game engine and Unity, respectively, [5], [1]. The editor in figure A.3 is an editor for a special case of behavior trees the developers call “patterns”. It has been used as an ongoing research project at the University of Alberta, [7]. Figure A.4 shows a behavior tree tool for integrating script-based behavior trees into the game engine. This concrete tool has been used by game developer studio Big Huge Games, [27].

One recurring concept across all but one of the editors is to represent the behavior tree using a hierarchical tree control. Using this control the tree branches downward rather than to the sides and the leaf nodes are rightmost rather than bottommost. This representation of a tree seems to be acceptable. One editor (figure A.6) use a more graphical approach to represent the structure of the tree. All of the editors feature some means of providing arguments to the parameters of the tasks. The editor in figure A.3 uses the arguments of the tasks to form a textural description of the task. In figure A.4 the arguments are defined within the game and are extracted using scripted code. Some of the editors (figure A.3, A.6 and A.4) compile the behavior trees to a format which the game engine can interpret or execute. The editor in A.5 makes it possible to run and stop execution of the behavior directly from the editor.

Let us continue to consider which features our behavior tree editor needs. To begin with, we have the features that make up the core of the editor and are a necessity to be able to test the system. These features include the editor being able to receiving a list of available tasks and their parameters, to create and move tasks, to edit the parameters of tasks and to convert the behavior tree to an intermediate data format and send it to Unity.

Some features are needed as they provide convenient functionality and help in making the editor easy-to-use. One such feature is to be able to explicitly start, pause and stop the execution of the game AI. Suppose that we have tested some behavior of an agent and found that something needed to be changed. After applying the change, we would like to test the agent in exactly the same scenario to see whether the applied change was sufficient. Therefore, it would be nice to be able to stop the execution and then restart it with a new behavior tree. The ability to pause the execution of the game AI would allow the game designer to better analyze complex behavior.

As the editor is part of a system designed to continuously test behavior, it would be convenient to have some way of logging debug data inside the editor. This could be information related to a specific task, it could be warning and error messages such as errors in parsing the behavior tree or unhandled errors within the tasks. Having an integrated debug console would make it easier to communicate this kind of debug information. Further, a filtering mechanism could be overlaid this console, allowing the game designer to choose which types of debug information to display. Finally, the editor would need some functionality to load and save behavior trees to allow the work process to continue across multiple sessions.

4.4.2 Comparing to Unity

In section 4.3.1 we looked at the main features of Unitys scripting system. Likewise, let us look at the key selling points of the editor in Unity. A selection of these are as follows, [4, sec. “Features – Editor”]:

- *Play, Pause, and Step*, i.e. being able to pause and step through the frames of the game to analyze complex behavior in detail
- *Drag ’n Drop Everything*, i.e. assets and objects can be assigned by visually dragging the resources in the editor
- *Prefabs*, i.e. objects can be turned into “prefabs” that can be instantiated. Any changes to the prefab can be propagated to all dependents

The first bullet point is concerned with testing the game. There might be a discrepancy between the implemented generating mechanisms and the observed behavior and testing is needed to be able to find that. This is thus used as a tool for visually debugging (potentially complex) behavior in the editor in Unity. We also listed the ability to play, pause and stop the execution (of the game AI) as a key feature of our editor.

Secondly, the developers of Unity have found it convenient to be able to drag and drop different types of assets to objects and scripts. This method is used instead of having to script everything or to choose from drop-down boxes or similar. We have also chosen to use a visual approach in the editor, e.g. being able to select objects in the game component and to drag and drop the tasks to form the behavior tree.

Finally, Unity relies on the concept of prefabs, or “premade fabrications”, also found in many other game editors and 3D modeling tools. This relates, in concept, to a class hierarchy in which we have classes that inherit functionality and properties from super types. There are often many recurring objects in games to lower the amount of content required and to make different object types easily identifiable. For instance, explosive barrels may be found at many different locations throughout a level in a game. If the texture, the explosion type or any other properties need to be changed for all the explosive barrel objects, this can simply be changed in the super type, i.e. the prefab, and will be reflected in all objects inheriting from it. This concept is comparable to using a look-up table for referring to subtrees. By doing so, we can change the subtree and all references will reflect the change as well. However, this does not enable us to use **subtype polymorphism** for behavior trees as prefabs do. For that another approach would needed (section 5.2).

subtype polymorphism

In the next section we will look at the concrete design of the editor.

4.4.3 Editor interface

The interface for the editor is shown in figure 4.15 on the next page.

The editor consists of a lot of different components, some of which were outlined in section 4.4.1. Some of the major component are collapsible and can be either a hidden or visible mode. The user can toggle between these modes to focus on certain aspects of the editor. Figure 4.16(a) on the facing page shows the editor will all collapsible components shown and figure 4.16(b) shows the editor with all collapsible components hidden.

The layout of the editor consists of four overall components. First and foremost, there is a component for constructing and manipulating a behavior tree (figure 4.17(a)). In figure

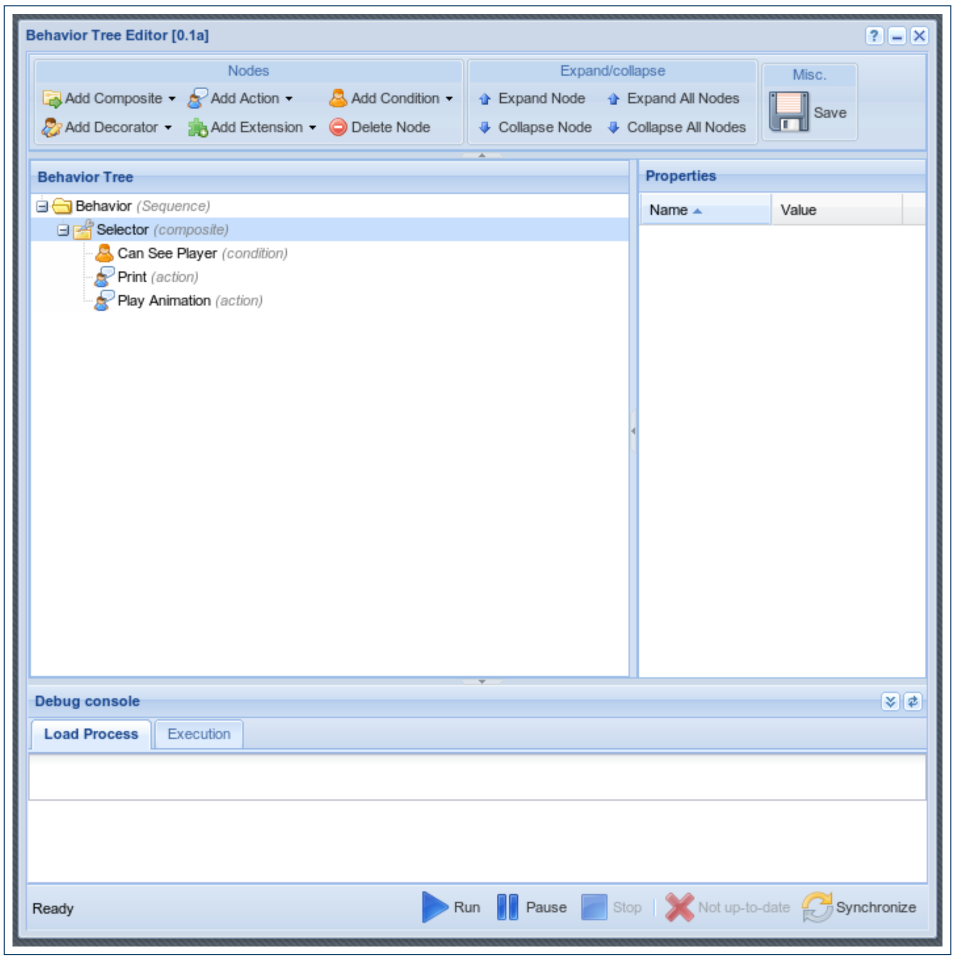
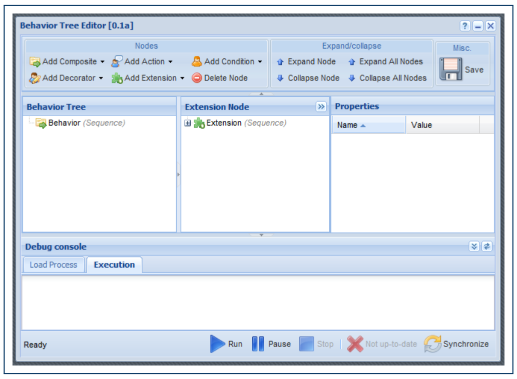
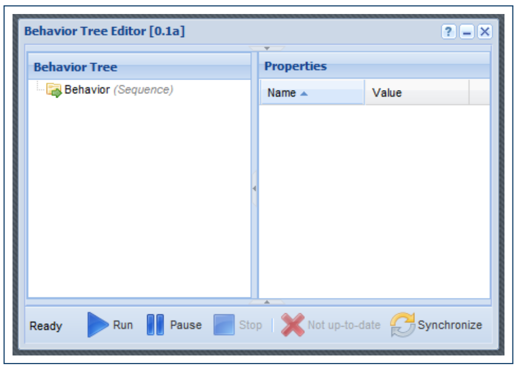


FIGURE 4.15: *Layout of the behavior tree editor*



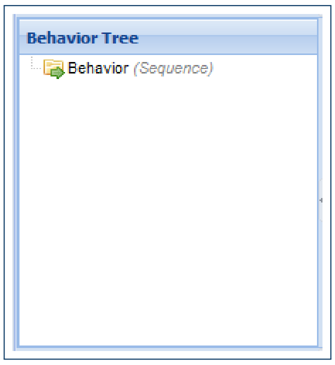
(a) Full layout view



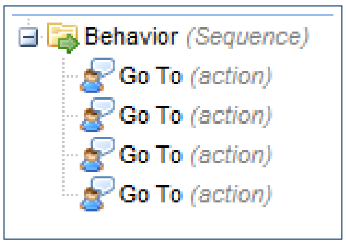
(b) Compact layout view

FIGURE 4.16: *Examples of customizing the layout by collapsing/expanding components*

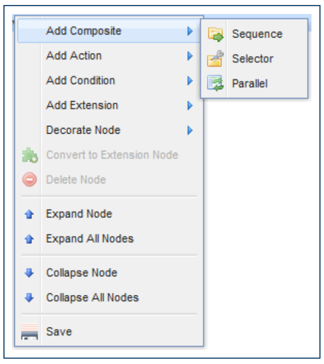
4.17(b) is shown how the behavior tree for the patrolling behavior from figure 4.7(a) looks in the editor. Each node represents a task, and the nodes can be dragged and dropped to



(a) The tree component



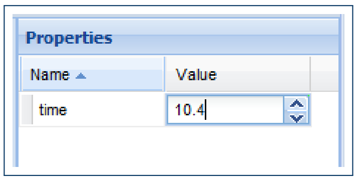
(b) Example node tree



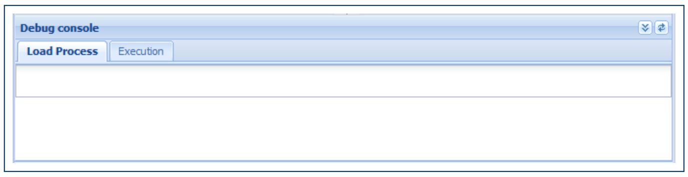
(c) Node context menu

FIGURE 4.17: *Different aspects of the tree editor component*

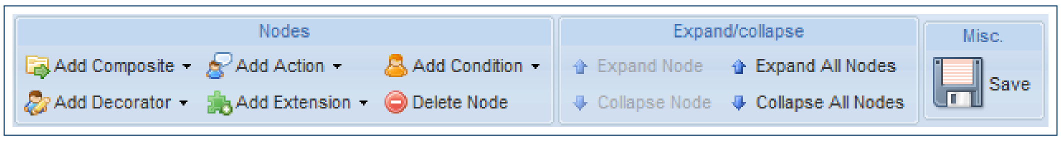
rearrange the tasks. If a node is right-clicked, a context menu for that node will appear. An example hereof is shown in figure 4.17(c). The context menu makes it easy to insert new nodes into the tree, to decorate existing nodes, to delete nodes or simply to collapse or expand (parts of) the hierarchical structure of the tree.



(a) The property grid



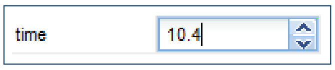
(b) The debug console



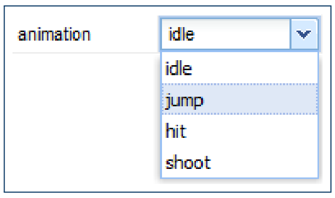
(c) The menu bar

FIGURE 4.18: *Other major components of the behavior tree workbench interface*

(a) Editing text



(b) Editing a decimal number



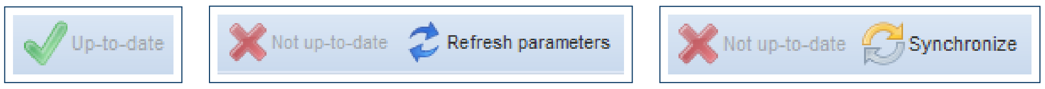
(c) Choosing from a list

FIGURE 4.19: *Different ways to handle editing of parameter arguments depending on the data type*

Another key component is the property grid (figure 4.18(a)). When a node is selected in the tree component, its parameters will be displayed in the property grid. The parameters are listed in name-value pairs. The value is the argument to the parameter. The arguments to the task parameters may have various data types (section 4.5.2 on page 109). The property grid enables the argument to be modified in a way that is suitable for that particular data type. Figure 4.19 on the preceding page shows examples of editing three different types of arguments; a string, a decimal number and a list.

In figure 4.18(b) we have the debug console. This component is used for logging various data from the game component as well as from the behavior. It can be used to log information to aid in debugging, warnings and errors to let the user know what has gone wrong. As of now, the debug console has two different tabs for logging. One is entitled "Load Process" and the other is entitled "Execute". Each represent a feed of debug information relating to its respective phase, i.e. loading a behavior tree into the behavior tree framework and executing the behavior, respectively.

Another overall component is the menu bar (figure 4.18(c) on the facing page). This bar provides easy access to the functions relating to nodes in the behavior tree. All the actions of the menu bar are also accessible through the context menu in the tree component.

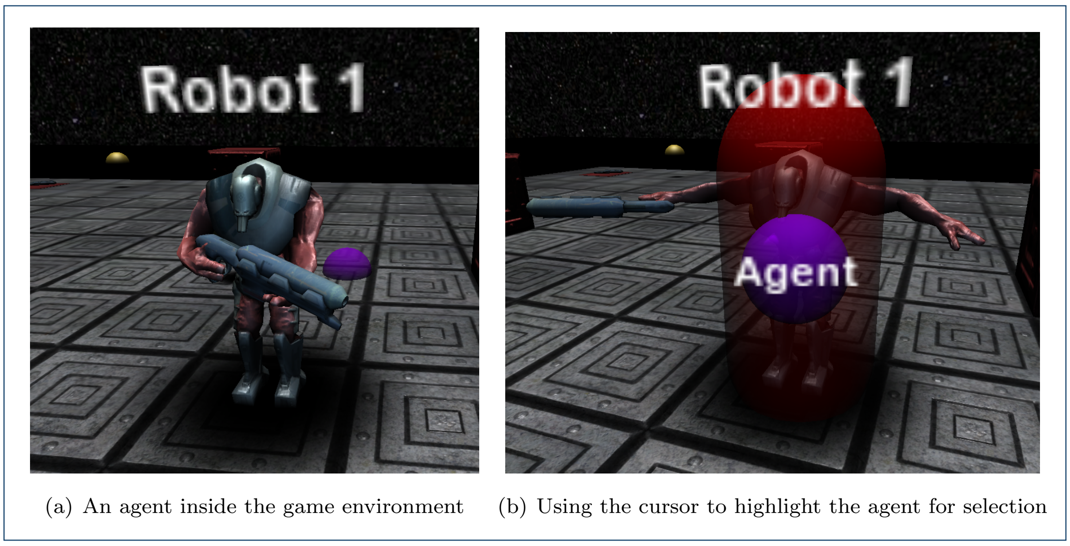


(a) Game behavior tree up-to-date (b) Game behavior tree needs to update its parameters (c) Game behavior tree needs to be fully synchronized

Figure 4.20: *Graphical indications on the state of the behavior tree of the game component compared to that of the editor component*

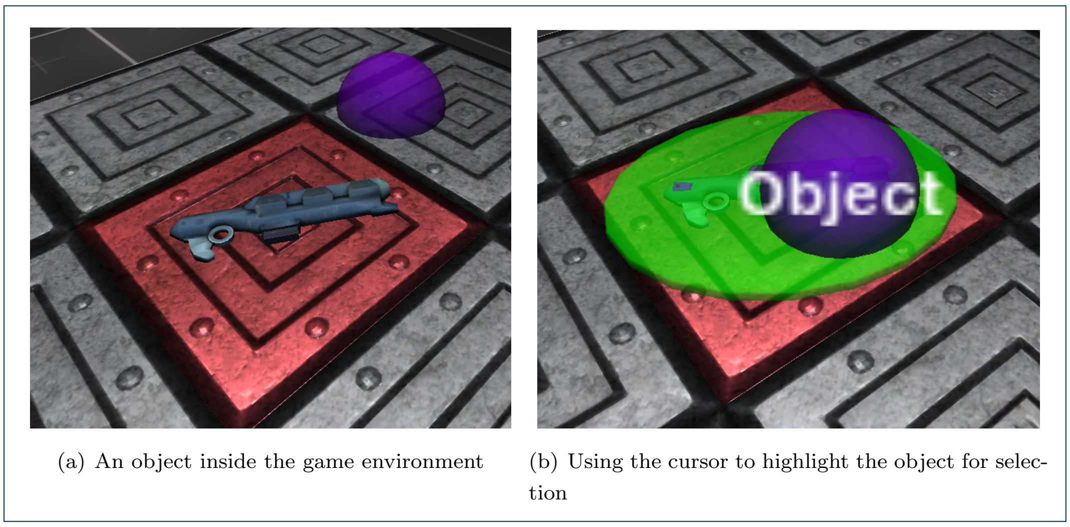
The status bar features a number of buttons. The "Play" button starts the execution of game AI in the game component, and "Pause" pauses the execution. The "Stop" button halts the execution and resets the behavior tree and the scene to their original states. On the far right side of the status bar is an indication of the correspondence between the behavior tree in the editor component and the behavior tree in the game component. The two behavior trees are identical if the behavior tree has not been changed in the editor since it was last synchronized with the game component. In that case, the "Up-to-date" icon is displayed (figure 4.20(a)). If the arguments to the tasks of the behavior tree has been modified, but the structure of the behavior tree is unchanged, then the "Refresh parameters" button is displayed. This button will update all the parameters of the behavior tree in the game component. If the structure of the behavior tree has been changed, the "Synchonize" button is displayed. This button will transfer the behavior tree and construct the corresponding object hierarchy in the game component.

The game component is integrated into the editor component, allowing a visual selection of certain objects through the game component. There is a special control for handling game object arguments. A task, like the `Go To` action, might have a parameter that needs a waypoint object. Clicking the (waypoint) argument value in the property grid will activate the game component and put it into focus. The user can then use the cursor object to



(a) An agent inside the game environment (b) Using the cursor to highlight the agent for selection

FIGURE 4.21: *Taking advantage of the integration of the game component into the editor component to visually select an agent from inside the game*



(a) An object inside the game environment (b) Using the cursor to highlight the object for selection

FIGURE 4.22: *Selecting an object from within the game*

highlight and select a waypoint object (figure 4.10 on page 89). Doing this will return focus to the editor and set the selected object as the value in the property grid. This selection process can work with all types of game objects in Unity. Figure 4.21 and figure 4.22 shows how to use the same approach to select an agent or an object from within the context of the game.

4.5 Behavior tree framework

The game scene component (section 4.3) uses the behavior tree technique (section 2.7.6 and chapter 3) for the game AI. We have implemented a framework, the behavior tree framework, to handle the execution of the behavior tree. It is implemented in Unity using JavaScript and, in addition to handling the core game AI, also features other related functionality. It can be used to load a behavior tree from a file or string, exposing the available tasks (and their parameters) for an agent and updating the parameters of tasks. The framework is self-contained and game independent. It provides an interface for communicating with the concrete game implementation and for defining new tasks. The majority of the framework can be thought of as a black box. The framework can be considered an application of the structural **Facade pattern**, [33, p. 185], [30, p. 100].

Facade pattern

In this section we will examine some of the technical aspects of the framework and how it interfaces with the game (section 4.3) and the editor (section 4.4). In the behavior tree framework we need three overall components; agents, behavior trees and an environment. In the next section we will look at how we represent those components in our framework.

4.5.1 Agent architecture

We need to define some appropriate structure for general-purpose agents. In general, an agent is “anything perceiving its environment through sensors and acting upon that environment through actuators” (section 2.2.1), [50, p. 32]. However, we are able to narrow our definition of an agent to agents whose agent function is determined by a behavior tree. Thus, we may broadly consider the condition tasks (section 3.2.2) to be the sensors of the agent and the action tasks (section 3.2.3) to be the actuators of the agent.

This being said, from a object-oriented viewpoint the agent should aggregate its sensors and actuators directly. Tasks would then have to query sensors[6] and utilize actuators[7] through the agent.

Let us disregard the concrete handling of sensors and actuators and focus on creating behavior for an agent. For this purpose, consider the following code.

```
class Agent extends MonoBehaviour {
  var behaviorTree : Task = null; // Behavior tree

  function Update() {
    if (Utils.isRunning() && behaviorTree) {
      behaviorTree.execute(this);
    }
  }
}
```

[6]For example, getting the visual range by `agent.getSensor("vision").getParameter("range");`

[7]For example, initiating movement by `agent.getActuator("movement").command("move forward");`

Here we have a class representing an agent. It has a single variable, `behaviorTree`, that represents the behavior tree of the agent (we assume that each agent has a single behavior tree). It extends the `MonoBehaviour` class, which is Unitys superclass for all game objects implementing a range of different functionality. One of the functions inherited from `MonoBehaviour` is the `Update` function. It relies on the **Template Method pattern** and is invoked every frame, [33, p. 325]. In each frame, the agent then tests if the game AI system is running and if it has a valid behavior tree (line 5), in which case it will invoke the `execute` function of the behavior tree with itself as argument for the "agent" parameter (line 6). The `behaviorTree` variable (line 2) will typically be instantiated to a `CompositeTask`, thereby creating a tree structure (figure 4.23).

Template Method pattern

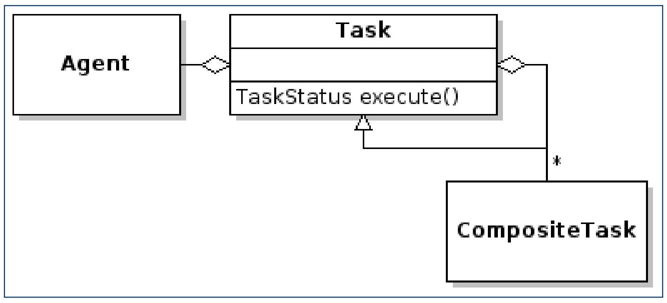


FIGURE 4.23: *Simplified `Agent` class diagram illustrating its relation to its behavior tree*

The `execute` function (line 6) will then invoke the `execute` function recursively through the tree (section 3.2.4). Figure 4.24 shows an example `execution` sequence that could occur in a frame.

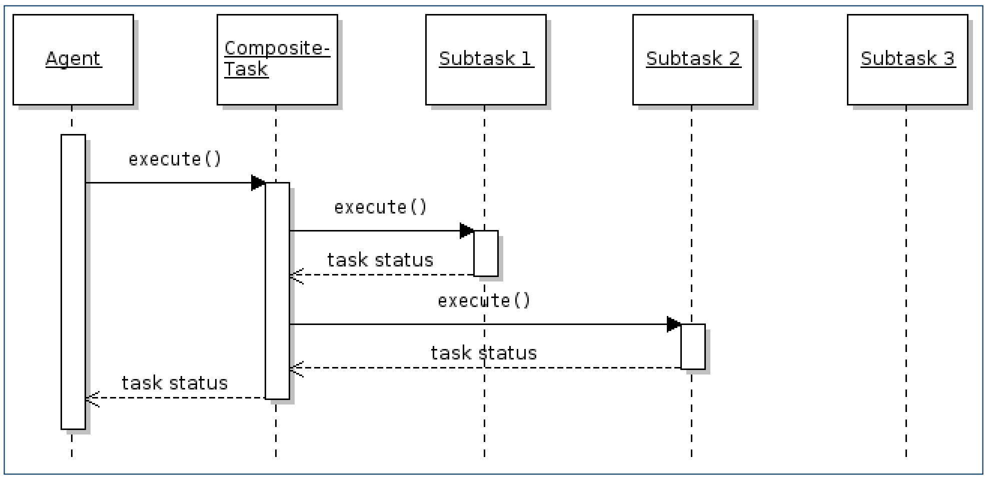


FIGURE 4.24: *Sequence diagram of an example execution of a behavior tree with three child tasks*

In this example, "CompositeTask" might be a selector and "Subtask 1", "Subtask 2" and "Subtask 3" could be three different solution to the problem that the selector tries to solve. The agent invokes `execute` on its behavior tree ("CompositeTask") which then proceeds to invoke execute on its first child task, which finishes its execution with a result that causes "CompositeTask" to proceed to invoke execute on its second child task. Once the second child task finishes it returns to "CompositeTask" at which point "CompositeTask" is satisfied with the returned status and returns to the agent thus terminating the execute cycle for that frame.

The framework includes a number of built-in tasks. These are tasks which do not depend on any specific games, but can be used in all types of games. An example is composite tasks which every behavior tree needs. Many more game independent tasks could be added to the framework, but we have chosen the ones we needed for experimenting with the workbench (section 4.1.2). The concrete tasks (i.e. not super types such as Task, XMLSerializableTask etc.) available in the framework is listed below.

- Composite tasks
  - Selector
  - Sequence
  - PrioritizedList
  - Parallel
- Action tasks
  - PrintAction (prints a string to the debug console)
  - WaitAction (pauses the agent for a number of seconds)
- Decorator tasks
  - LoopDecorator (loops the execution of the decorated task a number of times)
  - CountDownTimeDecorator (requires a given amount of time between executions of the decorated task)

These are tasks which aid in testing behavior and that we require to be able to experiment with the workbench. Each game needs to provide all game specific tasks to be able to model the desired agent behavior. Our concrete game implementation (section 4.3.1) adds a few additional actions tasks and a condition task. These are required to be able to model the desired behavior of the example use case (section 4.2). They are as follows.

- GoToAction (action that makes the agent go to a given location)
- PlayAnimation (action that plays an animation on the agent model)
- CanSeePlayer (condition that succeeds if another agent is spotted)

4.5.2 Serialization architecture

The behavior workbench consists of two overall components: the game component and the editor component. We need some way to share data between these components. In particular they need to be able to exchange all relevant behavior tree data. In this section, we will look at the concept of converting behavior trees (and related information) to/from a data format and how we use this technique to emphasize the generality of our solution.

In figure 4.23 on page 102 we saw a class diagram for the `Agent` and the `Task` classes. This diagram was simplified to emphasize the relationship between the agent and its behavior tree. However, the framework also needs to handle loading of behavior trees, exposure of parameters of tasks and to update those parameters. In order to handle this functionality, we need to extend the definition of a task. In figure 4.25 we show an extended class diagram in which we have included additional classes for handling said functionality.

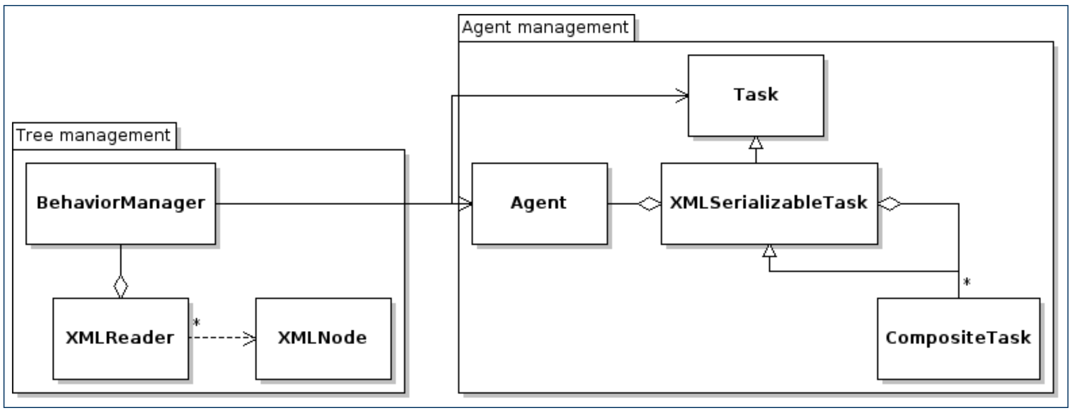


FIGURE 4.25: *Class diagram showing the relationships of the serialization architecture*

In the diagram we have divided the classes into two conceptual groups. The group on the left, named "Tree management", takes care of the higher-level functions related to the behavior trees, while the group on the right, named "Agent management", takes care of the more concrete responsibilities w.r.t. the tasks of the behavior tree.

Let us start by describing the lower level functionality and working our way up. Compared to the `Task` structure in figure 4.23, we have added a new class, `XMLSerializableTask`, which inherits from `Task`. Each `Agent` has a behavior tree of type `XMLSerializableTask` (whose super type is `Task`). That is, `CompositeTask` forms a composite pattern with `XMLSerializableTask` as interface, thus forming a tree of `XMLSerializableTask` objects.

serialization

deserialization

This intrusive change to the agent/behavior tree relation is needed to solve one particular problem: the **serialization** and **deserialization** of behavior trees. That is, the process of converting an object structure to/from persistent data. In this case, we want to be able to serialize (or deflate) the behavior tree into a data format that can be transmitted from the game component (section 4.3) to the editor component (section 4.4) over the web (section 4.1.1). Likewise, we want to be able to deserialize (or inflate) a behavior tree, i.e. construct a `Task` object structure to represent a behavior tree, based on data received from the editor component. As the name "XMLSerializableTask" suggests, we use the XML data format for serializing and deserializing tasks. We will go into further technical detail on page 107. For now, however, we will focus on the overall responsibilities of the classes.

By adding `XMLSerializableTask` as interface for the `CompositeTask` we have enriched all tasks of the behavior tree with the ability to be serialized and deserialized. Thus, the entire tree can be recursively serialized and deserialized. These higher-level operations are the responsibilities of the classes in the "Tree management" group, and the `BehaviorManager` class (section A.4.9 on page 142 in the appendix) provides the interface and much of the functionality hereof. `BehaviorManager` is responsible for creating behavior trees from XML data, creating XML "task templates" from different types of tasks and creating XML data for concrete behavior trees. We have already discussed the need for being able to serialize and deserialize behavior trees and will argue for the additional need for creating XML "task templates" on page 106.

The `BehaviorManager` class has a `XMLReader` object that is responsible for parsing XML data and instantiating the corresponding hierarchy of `XMLNode` objects. The `XMLNode` class is primarily a container class, storing the data corresponding to a behavior tree task or subtree in a convenient structure. It also has some helper functions for extracting the data in the correct formats, e.g. being able to convert the string "2.3" to the floating point number 2.3 or the string "2.0, 0.0, 3.5" to a Unity `Vector3` object describing the vector $(2.0, 0.0, 3.5)$.

In the next section we will examine which information is actually needed for building a behavior tree. We will continue this discussion on page 106 by looking at what data the editor component needs in order to enable the game designer to create the desired behavior trees. On page 107 we will define the concrete data protocols for transferring the required data between the behavior tree framework in the game component to the editor component. We will also look at the technical underpinnings for enabling this communication.

Data needed to construct a behavior tree

In this section we will examine what data is needed to fully construct a behavior tree.

Each agent has one behavior tree associated with it, and therefore we need to know which agent (or agent type[8]) the tree in question is associated with. The agent (or agent type) could be specified by a unique identification key. For instance, the key "soldier" could refer to the soldier agent type and "patrolling-soldier1" could refer to an individual agent. These identifiers could be used to retrieve the concrete behavior tree of that agent or agent type. In our prototype we have not made this mapping as we could test the system by simply using hardcoded agents.

In addition to a unique reference to an agent or agent type, we need a unique identifier for each task as well as the hierarchical structure of their composition. Also, each task may be parametrized with any number of parameters having a range of different **primitive data types** or even **composite data types** (e.g. a list of primitive data types or a Unity `Vector3`).

primitive data types

composite data types

As an example of what data is actually required to construct a behavior tree, let us examine the behavior tree in figure 4.7(a) on page 87 from the example use case. The behavior tree consists of a `Patrol` sequence, and four `Go To` actions, each with a single parameter. An example representation encapsulating all the required data is shown below:

[8]E.g. each agent of the type "soldier" may share the same behavior tree

```
Behavior [agent: A1 (string)]
- Sequence
- - GoTo [waypoint: W1 (string)]
- - GoTo [waypoint: W2 (string)]
- - GoTo [waypoint: W3 (string)]
- - GoTo [waypoint: W4 (string)]
```

Here, the dashes represent the depth of the hierarchy and the first word represents the unique task identification (e.g. `Sequence` and `GoTo`). The square brackets represent the list of parameters. They are listed in key-value pairs with the data format of the value in parenthesis. The `Behavior` task has been added as a default root for specifying the agent parameter without having to add specialized data fields. To sum up, the data we need to construct a behavior tree is the following:

1. Agent: Unique agent ID
2. Task: Unique task ID and the following:
   (a) Hierarchy: Depth of task hierarchy
   (b) List of parameters: List of key-value pairs in which the key is the variable ID and the value is a pair representing the actual parameter value and its data type
   (c) Children: Ordered list of Tasks (2)

On page 107 we will define the data format for containing all the required data.

Exposing tasks and parameters

In this section we will examine what data is needed by the editor to design the desired behavior tree. In other words, what data needs to be sent from the behavior tree framework (through the game component) to the editor, in order for the editor to know all it needs to know.

First, what does the editor need to know? The purpose of the editor is to design behavior trees which can be constructed by the behavior tree framework. On page 105 we saw what data was needed to construct a behavior tree. Therefore, the editor needs to know *at least* enough to complete the data requirements outlined on page 105. Suppose that you are a game designer who, using the behavior tree workbench, needs to create the behavior tree in 4.7(a) on page 87 from the example use case. You would need for the editor to know that there are two different types of tasks: `Sequence` and `Go To`. Furthermore the editor will need to know that the `Go To` task has a parameter, whose key is “waypoint” and whose value is of type string[9].

The agent may also have tasks at its disposal other than `Sequence` and `Go To`. It may have a `Smoke A Cigarette` or `Idle` task that it is able to perform. Not all tasks are available for all agents however. You may not want an agent guarding a highly flammable gas tank to have access to the `Smoke a Cigarette` task, for instance. Likewise, a guard tower would

[9]If the editor did not know the data type of the parameter it could not verify the correctness hereof.

not have a `Go To` task even though it may be available to another agent in the environment. Therefore, we need to associate a collection of available tasks with each agent (or agent type), and the editor needs to be able to extract this data from the behavior tree framework.

Additionally, the parameters of a task may be a list of possibilities (e.g. the chosen waypoint should be either W_1, W_2, W_3 or W_4) or it may be a composite data type (e.g. a Unity `Vector3`). In the case of the list we need to know the elements the list holds, and in the case of the composite data type we need to define some syntax to encode the structure (page 107). For assisting the game designer, we will require that each parameter has a default value. This also enables the game designer to reset the parameter values.

Finally, we need to know which agents or agent types are present in the environment. To sum up, the editor needs to know which agents are in the environment, which tasks they have available to them and the name and value(s) of their parameters. The key elements that need to be exposed from the environment are the following.

1. List of agents: List of unique agent ID
2. For each agent a collection of available tasks, each task containing the following:
   (a) Task: Unique Task ID
   (b) List of parameters: List of key-value pairs in which the key is the variable ID and the value is a pair representing the actual parameter value, its data type and a default value

In the next section we will define the concrete protocols we have chosen for transferring data from the game component to the editor component and vise versa.

Data passing protocols

In the two previous sections we determined what data needed to be exchanged between the game component and the editor component. In this section we will specify the protocols for communicating that data.

Extensible Markup Language

We have chosen the **Extensible Markup Language**[10] (XML) as the format to serialize to and deserialize from. XML is a textural data format that allows for storing arbitrary data. The primary reasons for choosing is that XML is simple, wide-spread, cross-platform, human readable and has focus on generality and usability over the internet, [9]. In this section we will define the actual data passing protocols and will also look at some examples of serialized behavior trees.

First, let us clarify the actual data flow between the components of the behavior tree workbench (figure 4.26 on the following page). If we assume the point-of-view of the editor, we need to get information from the game environment (page 106), e.g. which agents are available and which tasks they have. This data is queried through the game component. The behavior tree framework is responsible for serializing this data into a format that can be transfered to the editor. We do this by converting the tasks into "task templates" (figure

[10]See `http://www.w3.org/TR/REC-xml` for specification

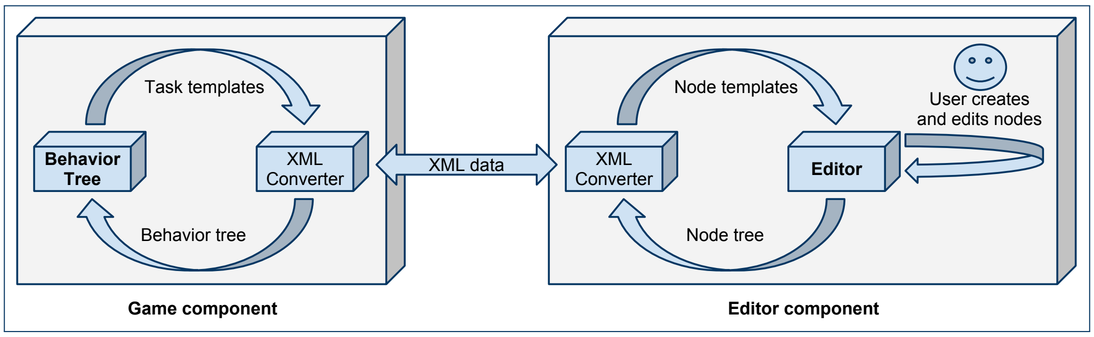


FIGURE 4.26: *Data flow in the behavior tree workbench*

4.26, left) and sending them to the editor component. The editor then deserializes the "task templates" into node templates. The node templates are used to instantiate new nodes which the user (e.g. the game designer) creates and edits to form the behavior tree (figure 4.26, right).

When the user wants to test the behavior, the nodes are serialized into XML data corresponding to the behavior tree the nodes represent. This data is then transfered to the game component which parses the data and constructs the corresponding behavior tree (page 105). The user can then observe, and experiment with, the in-game behavior that was designed in the editor.

The following section defines the data format for passing the data between the Unity Web Player and the editor component.

Behavior tree data — On page 105 we described the data needed to represent a behavior tree in our framework. The basic format for a task is shown below.

```
<[TASK ID] [PARAMETER LIST]>
  <!-- child tasks -->
</[TASK ID]>
```

`[TASK ID]` is the unique task identification key representing a type of task. If the task defined by `[TASK ID]` is parameterized then `[PARAMETER LIST]` is the list of arguments for the parameters of the task. For instance, a `Sequence` node without any child tasks will be represented by `<sequence/>`. The `Go To` node has a single string parameter and would be represented by e.g. `<gotoaction waypoint="W2"/>`. Thus, the XML data needed to represent the behavior tree from figure 4.7(a) on page 87 from the example use case would be the following:

```
<behavior agent="robot">
  <sequence>
    <gotoaction waypoint="W1"/>
    <gotoaction waypoint="W2"/>
    <gotoaction waypoint="W3"/>
    <gotoaction waypoint="W4"/>
  </sequence>
</behavior>
```

Task templates — On page 106 we specified that the editor needs a list of the available agents in the game scene. However, as the agents are visually available through the game component, we do not need to send this list explicitly. Instead we will allow the user to select an agent through the game component. Once an agent has been selected, we send the list of available tasks for that particular agent or agent type[11]. Thus, we need to send a list of all the "task templates" to be able to instantiate the tasks which are available for the given agent. The basic format for a task template is shown below.

```
<task codename="(STRING)" name="(STRING)" type="(STRING)" description="(STRING
    )">
  <parameter name="(STRING)" value-type="string" default="(STRING)" />
  <parameter name="(STRING)" value-type="int" default="(INTEGER)" min="(
      INTEGER)" max="(INTEGER)" />
  <parameter name="(STRING)" value-type="float" default="(FLOAT)" min="(FLOAT)
      " max="(FLOAT)" />
  <parameter name="(STRING)" value-type="list" default="(STRING)" values="(
      STRING LIST)" />
  <!-- more parameter tags... -->
</task>
```

A task is defined by a `codename`, a more descriptive, human readable, `name` a node `type`, a node `description` and optionally a list of parameters and their properties. A `Sequence` node that has no parameters will have a task template that looks like the following.

```
<task codename="sequence" name="Sequence" type="composite" description="
    Executes child tasks in sequence"/>
```

All parameters have a `name` a `value-type` and a `default` value. Different value types (i.e. data types) may have additional properties, as shown in the list of example parameters. For instance, the `Go To` task has a single parameter that is a list of possible choices. It thus needs to have a `values` field to hold the list of available choices. The task template for the `Go To` node will look like the following.

```
<task codename="gotoaction" name="Go To" type="action" description="Makes the
    agent go to the given waypoint">
  <parameter name="waypoint" value-type="list" default="W1" values="W1|W2|W3|
      W4" />
</task>
```

The task templates for agent A_1 in the example use case would consist of the task templates for `Sequence` and `Go To`, but also all other tasks available to it. It might have access to the

[11]The functionality for doing this is implemented in both the game component and the editor component. However, this feature is not important w.r.t. our purpose for the workbench (section 4.1.2) and we have thus simply used a static list of tasks

`Selector` and `Parallel` composite tasks, a `Say` action and a `Can See Agent?` condition. The complete task template data for this example could be the following (leaving out the `description` fields for brevity).

```xml
<tasks agent="robot">
  <task codename="sequence" name="Sequence" type="composite"/>
  <task codename="selector" name="Selector" type="composite"/>
  <task codename="parallel" name="Parallel" type="composite"/>
  <task codename="gotoaction" name="Go To" type="action">
    <parameter name="waypoint" value-type="list" default="W1" values="W1|W2|W3
        |W4" />
  </task>
  <task codename="sayaction" name="Say" type="action">
    <parameter name="text" value-type="string" default="Hello!" />
  </task>
  <task codename="canseeagent" name="Can See Agent" type="condition">
    <parameter name="agent type" value-type="string" default="robot" />
  </task>
</tasks>
```

The editor can load a behavior tree for an agent given the behavior tree and the task templates for that agent. The behavior tree data holds the actual behavior tasks, parameter arguments and hierarchical structure while the task templates provide the additional task information that the editor needs.

The implementation of the serialization/deserialization architecture is detailed in section A.1 on page 125 in the appendix. In it we describe how we have integrated the serialization and deserialization functionality into the behavior tree structure through the `XMLSerializableTask` class. We also introduce the `BehaviorManager` class to provide the overall interface for loading behavior trees from XML data and to register agents and tasks to be able to create task template data for agents.

4.5.3 The binding problem

One issue that needs to be considered when creating game AI is how to “bind” the representation of behavior to the representation of the environment:

> “*In order to actually build AI for a game, we'll need a whole set of additional infrastructure. [...] the AI needs information from the game to make sensible decisions. This is sometimes called “perception” (especially in academic AI): working out what information the character knows. In practice, it is much broader than just simulating what each character can see or hear, but includes all interfaces between the game world and the AI, [42, p. 11]*”

In designing behavior for game AI, it is the game designer who needs information from the game to determine how the agent should act. To use the example mentioned in section 2.1.1, if the goal of an agent is to flee from a monster, then the agent needs at least a way of sensing the monster and an actuator to move itself away from the monster.

Designing the desired behavior is the responsibility of the game designer but some sensors and actuators need to be available for doing so. The implementation of these is the responsibility of the game AI programmer. Thus, there need to be an interplay between the game AI programmer and the game designer to allow the designer to create the desired behavior. The game AI programmers need to define the *vocabulary* of the game which, in turn, the game designers will use to create behavior:

> “*The interface between the script and game engines facilitates the binding between vocabulary* [of actions and conditions available] *and the game world, [32, p. 288]*”

Furthermore, as mentioned in section 3.2, tasks may be defined at various degrees of abstraction. A task may, for instance, consist of applying current to a motor in order to trigger movement, taking a single step, taking multiple steps or navigating to another location altogether. Choosing a low level of abstraction can make it laborious to design the desired behavior and choosing a high level of abstraction can make it impossible to create specialized behavior.

There is a number of unanswered questions with regard to the binding between the game environment and the game AI technique for an agent. Some of these questions are as follows.

- How do agents perceive the world?
- How do agents perform actions in the world?
- Which sensors do the game designers need for obtaining the required information?
- Which actuators do the game designers need for creating the desired observed behavior?
- What level of abstraction should the available tasks have?

The answers to these questions depend entirely on the specific game environment and agent type:

> “*But what is the correct symbolic description of the world around the intelligence system? Surely that description must be task dependent, [12, p. 2]*”

Overall, there is no right or wrong way of answering these questions, it depends on the game and its agents. However, for a specific game, the choices do have a big impact on the whole process of creating game AI. Poor choices may, in fact, make it impossible to create the desired behavior so care must be taken to choose a sound approach for creating the interface between the game environment and the agent.

4.6 Extending the use case

We have already looked at a simple use case in section 4.2 and used that as an example throughout the chapter. In this section we will outline a new use case in which we build on top of the previous use case but increases the complexity of the game environment and the desired behavior. We will then discuss how a game designer would go about designing the required behavior using the behavior tree workbench.

The task environment of the previous use case is *fully observable*, *deterministic*, *sequential*, *static*, *discrete* and *single agent*. We will now modify the task environment of the game to be *partially observable* (not all state of the environment is available) rather than fully observable, *multiagent* (multiple agents whose performance measure depend on other agents) rather than single agent and *strategic* (only the other agents affect the state of the environment) rather than deterministic.

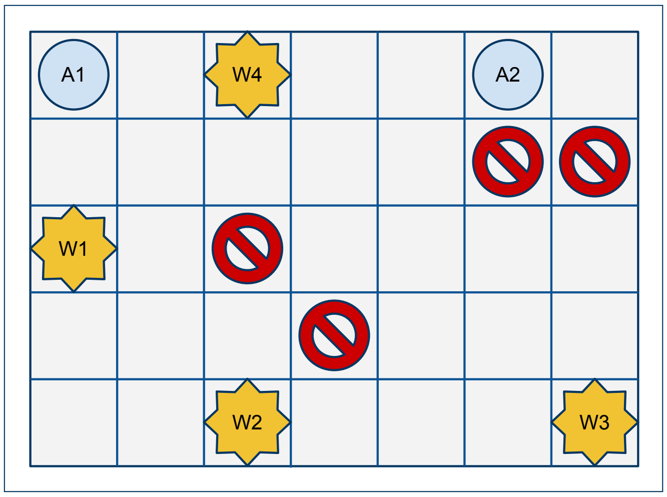


FIGURE 4.27: *The extended game environment with agent A_2 and four (red) obstacles added*

The game environment is illustrated in figure 4.27 as seen from above. Compared to the previous use case, we have added another agent, A_2, to the environment and made a successful interaction between A_1 and A_2 part of their performance measures (i.e. they are cooperative agents). Furthermore, we have given the agents a vision sensor providing a view-cone in the direction the agent is heading. Four obstacles (the red stop signs) have also been added to the environment and these block the agents line-of-sight.

We will continue to focus on the behavior of A_1. It will still be the goal of the agent to patrol along $W_1, \ldots, W_4$. However, now we also want to look out for agent A_2 and – if we spot it – move to it and initiate a short conversation before proceeding with the patrolling behavior. The PEAS description (section 2.3.2) of A_1 and A_2 is shown in table 4.28 on the facing page.

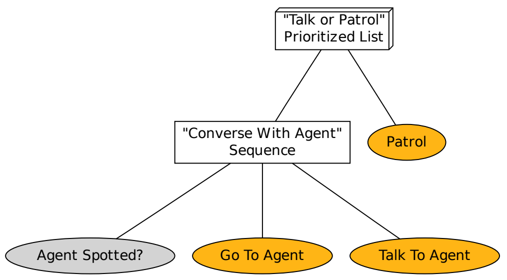
To design this behavior, we could start with the patrolling behavior tree from figure 4.7(a) on page 87. Here we have a `Sequence` encapsulating the patrolling behavior. It has four `Go To` actions as child tasks, each representing the action to moving to the next waypoint.

Agent	Performance Measure	Environment	Actuators	Sensors
A_1	Patrol along waypoints, converse with A_2	2D grid	Move up, down, left, right, diagonal & talk	Location in grid, vision
A_2	Converse with A_1	2D grid	Talk	Vision

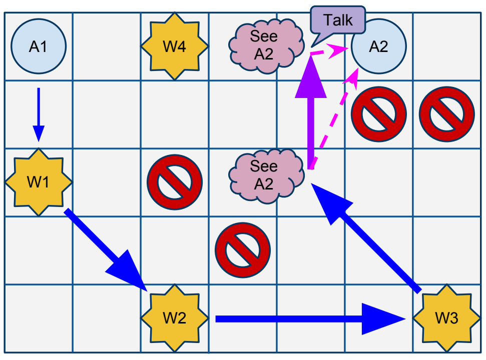
FIGURE 4.28: *PEAS description for agent A_1 and A_2*

To get the desired behavior we need to add a condition task to query the vision sensor and test whether we can see A_2. We will call this condition task `Agent Spotted?`. However, we cannot simply add it to the `Sequence`. Suppose we add `Agent Spotted?` as the first child task and in between each `Go To` task. We will get a lot of redundant tasks, and we will only execute `Agent Spotted?` after we arrive at each waypoint.

What we want is to continuously execute `Agent Spotted?` as we patrol, such that once the condition succeeds we can instantly react on it. For this purpose we will use a `Prioritized List`[12]. We want our highest priority child task to be the subtree that goes and converses with A_2 and the patrolling behavior as the lower priority child task. This way, the conversing behavior can take precedence over the patrolling behavior once the `Agent Spotted?` condition succeeds thus triggering the assertion. The resulting behavior tree is shown in figure 4.29(a). Note that the `Patrol` task refers to the patrolling sequence of figure 4.7(a).



(a) Behavior tree for patrolling and conversing with A_2



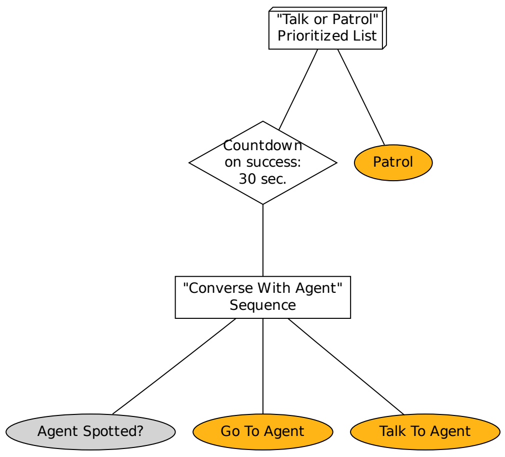
(b) Observed behavior of A_1

FIGURE 4.29: *The generating mechanism (a) and observed behavior (b) of the extended use case*

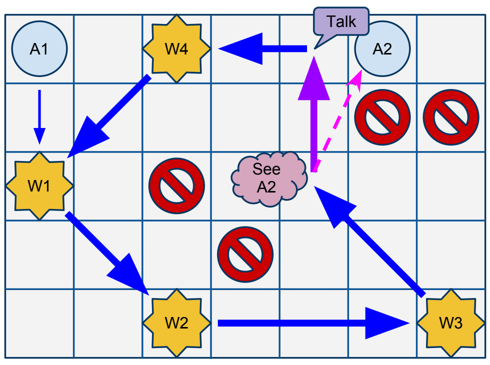
[12]We could also have used a `Parallel` task but a `Prioritized List` is a more pragmatic choice for this purpose (section 3.2.10)

Now let us test to see if the behavior we have designed actually corresponds to the desired observed behavior. Figure 4.29(b) on the previous page illustrates the observed behavior generated by the behavior tree in 4.29(a). Agent A_1 moves to W_1, W_2, W_3 and on the way to W_4 it spots A_2 ("See A_2", mid), moves to the agent and initiates a conversation. This behavior was exactly what we wanted. However, once the conversation has ended (`Talk To Agent` terminates), agent A_1 does not proceed to W_4 but instead initiates a new conversation with A_2. The reason is that the next frame of game AI processing will again execute the `Talk or Patrol` prioritized list which will try to execute its highest-priority child, the `Converse With Agent` sequence and the `Agent Spotted?` assertion will trigger again ("See A_2", top).

We need some way of keeping A_1 from performing the `Converse With Agent` sequence if it has executed it within a short amount of time. However, in this context, we are only interested in avoiding execution of the subtree if it actually succeeded. If `Agent Spotted?` failed, for instance, then we do not need to postpone the next execution of the `Converse With Agent` sequence. We will therefore define a new decorator type named `Countdown on success` which simply test the returned status of its decorated task, and if it succeeded then the decorator will return fail on execution until a given time has passed. This updated behavior tree is shown in figure 4.30(a).



(a) Updated behavior tree to avoid continuously repeating the `Converse With Agent` sequence



(b) The corresponding, observed, behavior

FIGURE 4.30: *Updated behavior tree (a) resulting in the desired observed behavior (b)*

Figure 4.30(b) shows the observed behavior in testing the updated behavior tree. This time, after `Converse With Agent` has terminated, the `Countdown on success` decorator ensures that the `Converse With Agent` subtree cannot be executed until 30 seconds has passed. The desired behavior is thus achieved after a few iterations of design and testing the observed behavior.

4.7 In conclusion

In this chapter we have described the work of designing the behavior tree workbench and each of its individual components. We have created a game environment that allows us to test the observed behavior of agents. On top of this we implemented a framework for handling game AI using a behavior tree technique. We have created an editor for designing and testing behavior trees. The game component has been integrated into the editor through the behavior tree framework. Finally, we have discussed how the workflow of designing agent behavior breaks into incremental iterations of designing and testing. The behavior tree workbench and the concrete example scene can be found as described in section 1.5.

The work with the behavior tree workbench has enabled us to do practical experiments in many different areas. We believe that this approach for creating behavior is suitable for both the game AI programmer and the game designer. The processes of designing and testing behavior becomes much more interlaced, resulting in a workflow that is more intuitive. We have had a few people test our tool and they agree with this observation. Although the tool is simply a prototype the feedback from testers has been positive. The testers ranged from novice and intermediate in game development experience and all found the prototype to have the potential of being a useful tool for designing and testing behavior.

Before attempting to extend the tool to be applicable in commercial game development, an extensive investigation into the specific needs of the game developers would have to be undertaken. Also, more rigorous and structured tests would have to be carried out of verify the usefulness of the tool in different contexts.

Chapter 5

Conclusion

In this thesis we have examined the problem that we described in chapter 1. We listed some specific issues in section 1.3 which we had chosen to investigate based on the description of the problem. In the next section, we will synthesize and conclude on each of those issues. In section 5.2 we present considerations for how to extend and improve the tool prototype that we have created for designing behavior.

5.1 Conclusion

This thesis has been concerned with the problem of game designers lacking good tools for creating behavior of NPCs thereby resulting in a troublesome design process. We have implemented a prototype of a tool for making this process easier, both for the game designers and the game AI programmers.

5.1.1 Survey of game AI techniques

We wanted to investigate which properties the game AI platform required for handling the decision making process in a modern game. To this end we made a survey of the most popular and commonly used techniques in the game industry. The survey showed that many of the techniques still in use today, such as finite state machines and rule-based systems, have much in common with some of the first approaches to handling game AI. Finite state machines, in particular, are still a popular technique even though its first applications in games was more than 30 years ago. It was, for instance, used for the behavior of the ghosts in Pac-Man, a game which was released in 1980.

However, these techniques started to show shortcomings as the game industry matured and games became more complex. The old techniques were extended in numerous ways as work-arounds for overcoming these issues. The extensions focused, among other things, on coping with the increased magnitude and complexity of behaviors, on using parallel behaviors and on combining sequential and reactive behaviors. One of these extensions, named hierarchical finite state machines, allowed the behavior of finite state machines to be structured as a hierarchy of behaviors. This was especially convenient for the game designers, as behavior could be considered at different levels of abstraction. Finite state machines already had the desired property of being *reactive* but this extension also made the technique *intuitive*.

Another technique that has only recently gained acceptance in the game industry is the use of planners. It has traits that are especially desirable in creating rational agents, namely that the technique creates *autonomous* and *purposeful* behavior by construction. Planners have been criticized for being difficult for game AI programmers to integrate into games and hard to game designers to use for creating the desired behavior.

The technique of scripting, on the other hand, is highly *flexible* as it provides full access to the functionality of game AI platform thereby enabling the game designers to model any behavior. Additionally, scripting is a *data-driven* approach, a trait which has become indispensable for game AI. However, scripts are not intuitive and are difficult to introspect. In spite of this, scripting has become a very widespread technique in the game industry.

Our focus has been on handling the decision making process for character-based game AI. For this purpose, we have chosen to use a novel technique called behavior trees. Since its emergence, the behavior tree technique has become increasingly popular in the game industry. One reason for this popularity is that the technique is a synthesis of some of the most popular game AI techniques today, namely hierarchical finite state machines, planners and scripting. Behavior trees compose modular behavior into a hierarchical structure, making it intuitive for both game AI programmers and game designers. It relies on reactive planning and provides autonomy by supporting fall-back solutions. Behavior trees combine the desirable properties of many other popular techniques such as being *intuitive*, *reactive*, *autonomous*, *purposeful* and *flexible*.

Behavior trees are well suited for game AI programmers as they are conceptually simple to grasp, very simple to implement and use modular behaviors.

5.1.2 Design of a tool for game designers

Creating the desired behavior of a game agent is an iterative process. The game AI programmers provide the game AI platform on top of which the game designer will design the behavior using a tool. The tool expresses the functionality of the game AI platform in a way that is more suitable for game designers. Creating the desired agent behavior consists of iterations of design and test phases. The game designer uses the tool to design behavior, then tests the behavior to verify that the perceived behavior in the game is as expected.

We investigated how to design an intuitive tool for aiding the game designer in creating the desired behavior. The design process should be intuitive and it should be possible to visualize the range of possible behaviors. Additionally, it should be easy to introspect the designed behavior. We have chosen to use the hierarchical structure of behavior trees as a central element in the tool. The representation of the behavior is nearly identical for the game designer and game AI programmer, which helps to establish a shared vocabulary for discussing behavior. This is important as the process of designing behavior is an interplay between the game AI programmer and the game designer. Additionally, the hierarchical structure allows the game designer to focus on certain layers of abstraction. The structure can be manipulated directly by simply dragging and dropping behaviors.

Our behavior design tool provides a list of the available behavior, allowing the game designer to pick behaviors to satisfy a given task. Each behavior can be customized by manipulating its parameters. Also, the same behavior may be tweaked to apply in multiple situations. The game designer needs a convenient way to select the arguments to the parameters of behaviors. We have chosen to implement a game system that can be integrated into the behavior design tool. We hereby enable the game designer to visually select arguments such as agents, waypoints and objects directly from the game. This makes the design process more intuitive and is less error-prone and ensures a better correlation between the designed behavior and the perceived behavior. In addition, the integration allows the game designer to test the designed behavior immediately without having to leave the behavior design tool. The overall workflow becomes more fluid, leading to shorter design iterations, more productive game designers and ultimately a better behavior in a shorter amount of time.

5.1.3 Behavior tree framework

We have designed and implemented a framework for bridging the game AI platform and the behavior design tool. The primary purpose of the framework is to facilitate the communication between the two systems. This involves transferring data between the game AI platform and the behavior design tool. The framework makes it easy to convert behavior trees to XML and to create behavior trees from recieved XML data.

The framework also makes it more convenient for game AI programmers to implement behaviors and use the behavior tree technique in conjunction with the design behavior tool to handle the game AI.

5.1.4 Implementing the tool

Our focus has been on implementing the features that will enable us to examine the practicability of our approach to implementing and designing game AI. For our purpose, the creation of the tool serves primarily as means of experimenting with the investigated concepts in practice.

We have implemented a concrete game environment, a behavior design editor and a framework for handling the communication between them. We have chosen to make the tool web-based to make it more accessible and to avoid the need for software distribution, installation and updating.

The game environment and the framework for handling the game AI using the behavior tree technique are created using the Unity game development tool. The framework is implemented using Unity's JavaScript implementation and relies on classical software design patterns for achieving a composite and robust architecture. Implementation of the framework has also focused on making it convenient for the game AI programmer to create new behaviors and expose them to the editor.

We have also built a concrete example game scene using Unity. The scene reflects the need for some of the behaviors often found in games, i.e. patrolling along waypoints. This scene served as a testbed for experimenting with the integration of the game into the editing and testing process. The framework facilitates the integration of the game component into the editor component and can be treated as a black box.

The editor for designing behavior is implemented using a JavaScript library for creating rich web applications called Ext JS. It relies on standard controls such as buttons, toolbars and menus which are familiar to most people. The editor uses a tree control to model the design of behavior trees and supports drag and drop of tree nodes for easy manipulation of the tree.

The purpose of the tool we have designed and implemented is to experiment with the proposed approach to designing behavior. We believe that we have accomplished this goal by showing how to create a system for making a convenient workflow for both the game AI programmer and the game designer. We have had a few people test the tool and gotten positive feedback. Detailed user tests would have to be carried out to more rigorously test the applicability. The tool we have created is a prototype for a specific purpose, and its functionality does not have the breadth or depth that would be required in the development of commercial games. The approach we have investigated seems practicable and we expect that, if given enough time, the tool could be made useful for a wide variety of game development efforts. Some ideas and considerations on how the tool could be improved and extending are listed in the next section.

5.2 Future work

In this thesis we started out examining game development in general and game AI development in particular. Even game AI is an enormously broad field spanning many different disciplines. As we went along, we narrowed the scope further. We finally settled on a number of issues we wanted to shed light on. The prototype was implemented with the features required for enabling us to test our approach to handling these issues. In this section we will list some of the features and extension we would have liked to have investigated further and implemented, had we had more time. Many of the listed features would be required if the prototype was to be extended for use in developing commercial games.

5.2.1 Behavior trees

- **Blackboards**

  In section 2.7.2 we discussed blackboards as a way for handling communication between multiple agents. The same technique may be used to handle data for the behaviors of a single agent. It can be used as a memory model of the agent for model-based agents (section 2.4). That is, it can store information about objects in the world, e.g. where an object was last seen. This can help to form a more realistic AI because the AI's view on the world can diverge from reality. This also alleviates some of the storage problems concerned with behavior trees. Another usage is to integrate blackboards into the behavior tree structure, thereby forming a hierarchy of blackboards, [42, sec. 5.4.5]. This can then be used to contain data for tasks across multiple executions or as a way for tasks to pass data between them.

Let us look at an example in which a hierarchy of blackboards could have been used. In figure 4.30 on page 114 the behavior of agent A_1 is hardcoded to interact with agent A_2. Instead, we can use a blackboard to record which agent is spotted by A_1 in the `Agent Spotted?` condition, and then read this value back in the `Go To Agent` and `Talk To Agent` actions.

To do that, we could include two parameters in the `Agent Spotted?` condition: “Write agent to blackboard” (a boolean value) and “Agent blackboard variable” (a string value). You could give the arguments `true` and `"spottedAgent"`. The `Go To Agent` and `Talk To Agent` actions could then have the parameter “Agent blackboard variable” which would be set to `"spottedAgent"`. The `Agent Spotted?` condition would then write the unique identification key of the spotted agent (if any) into the variable in the blackboard for the other tasks to read. A `Fresh Blackboard` decorator could be used to create a new clean blackboard to ensure that no variables are overwritten or left as garbage [42, p. 364-365].

- **Level-of-detail**

  The idea of using a level-of-detail technique (section 2.6.3) is to simplify computations by making approximations when the player will not notice the difference, [32, p. 294]. There is no built-in support in behavior trees for handling this. One approach to integrate a level-of-detail technique in behavior trees would be through decorator tasks, e.g. a `Distance LOD` decorator. For instance, the `Distance LOD` decorator task could flag the desired level-of-detail (e.g. on a blackboard) based on distance and its decorated task could act accordingly.

- **Memory optimization**

  The memory footprint of behavior could be greatly reduced by using a look-up table of behavior trees (section 3.3 and 3.4) rather than instantiating one or more behavior trees per agent. This is a clear-cut approach for handling the behavior of a group of agents having the same behavior. However, this approach is not as obvious if agents generally have customized behavior. An approach similar to that of polymorphic FSMs (section 2.7.2) may be used to allow custom behavior to be used in conjunction with looked-up behavior.

  Additionally, there are problems concerning the persistent state of tasks, e.g. decorators which depend on some data and parallel processes which need to keep a list of active tasks. One solution is to flag these tasks to always be stored in the tree. Another is to have them store their data at each agent, e.g. on a blackboard (section 2.7.2).

- **Polymorphism**

There are no natural way for a behavior tree to inherit the tasks and arguments of another behavior tree. This is relevant in many situations, e.g. for adding character-to-character variety. Consider a game having two different types of agents: a “Melee soldier” and a “Ranged soldier”. The only difference would be that the “Melee soldier” has a close-range aggressive combat style while the “Ranged soldier” has a long-ranged defensive combat style. It would be convenient to be able to define an overall “Soldier” behavior that both types of agents could inherit from and customize.

One approach could be to inject custom behavior into the “Soldier” tree depending on which agent type instantiated it. This approach is similar to that of polymorphic FSMs (section 2.7.2). However, there is still no obvious elegant way of customizing the parameters of the tasks of the “Soldier” tree. One solution for this is to read all arguments to parameters from a “character” file, instead of embedding them into the tree. This approach can also be used to fall back to the “character” file of the parent (i.e. the super type) if the parameter is not found in the “character” file itself. Thereby a hierarchy of character types is created, [37].

- **Integrate scripting**

  The concept of scripting could be integrated in behavior trees by a special `Scripted Behavior` task. That task could e.g. take as argument a regular script such as a Lua script (section 2.7.5). It could also take as argument intermediate data representing a behavior tree. This data could then be expanded into the tree at runtime. Further, such a task may be triggered by some condition in the game, a concept called **stimulus behavior**, [37]. This makes it easy to enable custom (i.e. scripted) behavior under certain circumstances.

  stimulus behavior

- **Non-deterministic tasks**

  In section 2.7.2 we discussed non-deterministic FSMs. It is also possible to introduce randomness into the composite tasks of behavior trees. This can be done simply by shuffling the list of child tasks before executing any children. This can help to increase replayability by reducing the predictability.

- **Behavior impulses**

  Types of behavior may occur in which it is required to bypass the regular search paradigm and jump to a task somewhere in the tree. An approach for doing this is called a **behavior impulse** and is a task that redirects the execution to another behavior in the tree, [37]. It can e.g. be used for situation-based activation of otherwise lower-priority behaviors.

  behavior impulse

- **Interruptibility and resumability**

  In some situations, when a task is being prematurely terminated as another task gets precedence, we would like to resume the interrupted task once we are done with the other task. For instance, suppose agent A is going to talk to agent B on the other side of the room, but is interrupted by agent C for some reason. Once agent C is done with agent A, we would like for agent A to continue to go to talk to agent B. Likewise, we would like two agents having a conversation to continue with that conversation after they have been interrupted. A solution for this using a queueing system is proposed in [29].

5.2.2 Editor

- **User-created look-up tasks**

  As described in section 3.4, there are many instances in which it would be convenient to represent tasks by using a look-up table rather than instantiating objects. This also allows the game designer to create subtrees that can be used in several different places in a behavior tree or even in different behavior trees. For instance, consider the behavior of figure 4.7(a) on page 87 composed of a `Patrol` sequence with four `Go To` actions as children. This behavior could be made into a single task that could be referenced through a table look-up. It would then be very applicable to have a built-in subtree editor. Even more so to have it working in concert with the regular behavior tree editor. In that way, a look-up subtree might be created by simply dragging the subtree from the regular behavior tree editor to the subtree editor.

  This concept could be extended to allow the game designers to create a library of custom trees. These trees could be referenced by different behavior trees, and changes in the custom trees would be reflected in each behavior tree. It might also be convenient for the nodes representing custom trees to have all the collective parameters of its children, such that entire high-level behaviors could be easily configured for specific requirements.

- **Run-time highlighting of active tasks**

  To aid the game designer additionally in the testing process it could be convenient to emphasize the correlation between the generating mechanism and the observed behavior. One way of doing so would be to highlight the task(s) active in the game component in the editor. This would give the game designer a better intuition about the result of the generating mechanism and allow more easily to isolate faulty behavior. Completed tasks could in addition be flagged with their terminating status such that the game designer is able to follow the execution flow.

  As some tasks terminate instantly or within very few frame, it could be convenient to be able to step through the execution. Alternatively, a way of controlling the execution speed could be added.

- **Breakpoint system**

  For the purpose of debugging it could be convenient to have a way of inserting breakpoints into the behavior. This could be implemented simply as a built-in task those behavior is to halt execution and return to the editor. Ideally it should be possible to inspect the state of the agent, the behaviors and, for instance, the blackboard at the instant the breakpoint triggered. An easy way of handling this, would be to dump the relevant data to the debug console.

- **Manipulating multiple nodes**

  As of now, there is no way of selecting multiple nodes. Having this functionality would help in moving or deleting several nodes at the same time. Also, this may allow for editing the arguments of several nodes simultaneously.

5.2.3 Miscellaneous

- **Data validation**

  Both the game component and the editor component of the behavior tree workbench are designed to be replaceable. It would thus be important to have a robust mechanism for validating the data received by either component. One such mechanism could be to validate against a XML schema. The schema would then be used to verify the correctness of the syntax of the received data. If the validation fails, the data should be rejected.

- **Game modes**

  Our implementation of game component operates as a combination of both a game and an integrated editing and testing tool. In a concrete development scenario, the game would require three explicit modes: play, edit and test.

  The play mode would simply be the game. This mode would have all the things required by the game such as user interface, high score, story, gameplay etc..

  The edit mode would be the mode in which it was possible to interact with the resources of the game, primarily with the purpose of visually selecting objects from within the game as part of the design process. This mode would be a static representation of a particular level in the game and the agents and objects in that level.

  The testing mode would be similar to the play mode, but focusing on debugging behavior. Thus, there would be no need for user interfaces or high scores, but maybe some other mechanisms suitable for testing. This could e.g. be to draw lines symbolizing where the agent is going, having icons to symbolize the current state of the agent etc..

- **Using the editor with other systems**

  As mentioned, both the game component and the editor component have been made to be replaceable. We have not yet experimented with replacing either component. However, it would be possible to replace the game component with a completely different system, for instance a LEGO Mindstorms robot. Depending on the context, it might be convenient to keep a simulated environment for editing the behavior, while the testing could by trying the behavior in the real world. For this to work, the robot would need to be able to receive behavior tree XML data and use this to create behavior. Such a system might find usage in teaching robot systems.

Appendix A

Appendix

A.1 Implementation of the serialization architecture

The implementation of the serialization is mainly located in the XMLSerializableTask class, and the implementation of deserialization in located in the BehaviorManager, XMLReader and XMLNode classes.

The interface for XMLSerializableTask has been extended from Task to include the functionality required to be able to serialize tasks as well as parsing and reading parameters. Disregarding some helper functions, the interface looks as follows:

```
class XMLSerializableTask extends Task {
function XMLSerializableTask(codeName : String, name : String, type : String);

  function getTemplateXML() : String;
  function getParameterXML() : String;

  function registerParameter(paramName, paramValue);

  virtual function readParameters() : void;
  function readParametersFromXML(node : XMLNode) : boolean;
  function resetParameters() : void;
  function getDefaultParameterValue(key : String) : Object;
  function getParameterValue(key : String) : Object;
  function setParameterValue(key : String, value : Object);

  virtual function clone() : XMLSerializableTask;
}
```

The constructor takes as arguments the code name, descriptive name and type of the task. The arguments are used when the task is serialized. The two next functions, `getTemplateXML` and `getParameterXML`, are functions that work together in order to serialize the task. The `getTemplateXML` function creates the `<task/>` tag structure and invokes `getParameterXML` to fill in the `<parameter/>` tag structure. Figure A.1 on the next page illustrates the serialization sequence. This process will create the XML template task data for a single task.

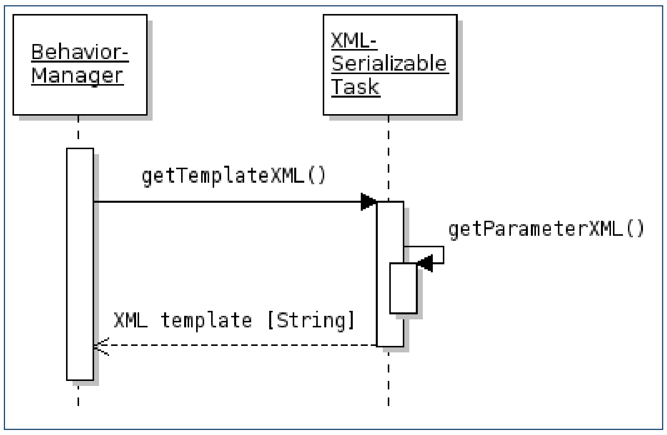


FIGURE A.1: *Sequence diagram of the serialization of a single task*

The `registerParameter` function binds the parameter to the task, along with the default value (the second argument). The function is able to automatically determine the value type based on the default value, and this type is stored as well. Because the parameters are now bound to the task, they will automatically be included when a task template of the task is created.

Template Method pattern

The `readParameters` function is an application of the **Template Method pattern** and will be invoked whenever the task is created, or the parameters have been modified or reset. Reading the parameter values here will then ensure that the parameters are always up to date.

We will not go into further detail about the next functions (`readParametersFromXML`, `resetParameters`, `getDefaultParameterValue`, `getParameterValue` and `setParameterValue`), which are helper functions concerned with the parameter values of tasks.

Prototype pattern

The `clone` function is an application of the **Prototype pattern**, [33, p. 117]. It is a virtual function that instantiates a new object of the current object, i.e. returns a *clone* of the object. It is possible for the caller to clone the object without actually knowing the specific type of object, but simply that it is a subtype of `XMLSerializableTask`. We use an intermediate data format to represent the behavior trees and use a cloning operation to instantiate the different tasks of the tree. This approach to instantiating and reusing behavior trees is also proposed in [42, sec. 5.4.6].

A concrete implementation of `XMLSerializableTask` is in `GoToAction`. `GoToAction` is the implementation of the `Go To` action, and its implementation shows how tasks are initiated and how they read their parameters. An abbreviated version of the `GoToAction` class is shown in the code listing below.

```
class GoToAction extends XMLSerializableTask {
  var destination : Vector3;
  var acceptableDistance : float = 0.1;

  function GoToAction() {
    super("gotoaction", "Go To", "action");

    registerParameter("waypoint", "NW Waypoint");
    registerParameter("acceptableDistance", 0.1);
  }

  virtual function readParameters() {
    destination        = getPosFromString(getParameterValue("waypoint"));
    acceptableDistance = getParameterValue("acceptableDistance");
  }

  virtual function execute(agent : Agent) {
    // ...
  }

  virtual function activate(agent : Agent) {
    agent.animation.CrossFade("run");
  }

  virtual function deactivate(agent : Agent) {
    agent.animation.CrossFade("idle");
  }

  virtual function clone() : XMLSerializableTask {
    return new GoToAction();
  }
}
```

The constructor invokes the constructor of the super class (i.e. `XMLSerializableTask`) and provides as arguments the code name, name and type of the task. It then registers the two parameters, "waypoint" and "acceptableDistance", that the task requires. In the `readParameters` function it parses and reads the parameters of the task and stores them in local field variables.

The `activate` and `deactivate` functions are invoked when the task is initiated and when it is terminated, respectively. Here, we utilize the animation system to fade the animation to the "run" animation in `activate` and fade to the "idle" animation in `deactivate`.

The deserialization of XML data into behavior trees in the responsibility of the `BehaviorManager` class. To deserialize data to a behavior tree, `BehaviorManager` needs to know which types of tasks are available. It needs to know the different types of tasks such that it can instantiate them to create the object hierarchy of the behavior tree. In the following code listing we create an instance of the `BehaviorManager` class and use it to register several different types of tasks.

```
var behaviorManager = new BehaviorManager();

behaviorManager.registerBehavior(new Sequence());
behaviorManager.registerBehavior(new Selector());
behaviorManager.registerBehavior(new Parallel());

behaviorManager.registerBehavior(new GoToAction());
behaviorManager.registerBehavior(new WaitAction());
behaviorManager.registerBehavior(new PrintAction());
behaviorManager.registerBehavior(new PlayAnimation());
```

After this, the `behaviorManager` object knows these tasks and is able to clone them to instantiate new objects using the `clone` function in `XMLSerializableTask`. To instantiate a new task, we just call the `createBehavior` function, which takes a task code name as argument. It will then look-up the task corresponding to the code name and return the object created by that tasks `clone` function.

However, before we know the code name of the tasks of the XML data, we need to read and parse it. For this purpose, we use the `XMLReader` class, which is a custom built XML parser. It has a function called `read` which takes XML data as argument and returns the corresponding hierarchy of `XMLNode` objects holding the generic behavior tree data. This hierarchy of `XMLNode` objects is then traversed, and for each object the corresponding task is instantiated by cloning and the parameters are read using the `readParametersFromXML` function from `XMLSerializableTask`. This traversal is recursive, and the hierarchical structure is maintained throughout the recursion. When the recursion ends, we will have created and combined all the tasks of the behavior tree as well as assigning arguments to all their parameters. Figure A.2 illustrates the entire process.

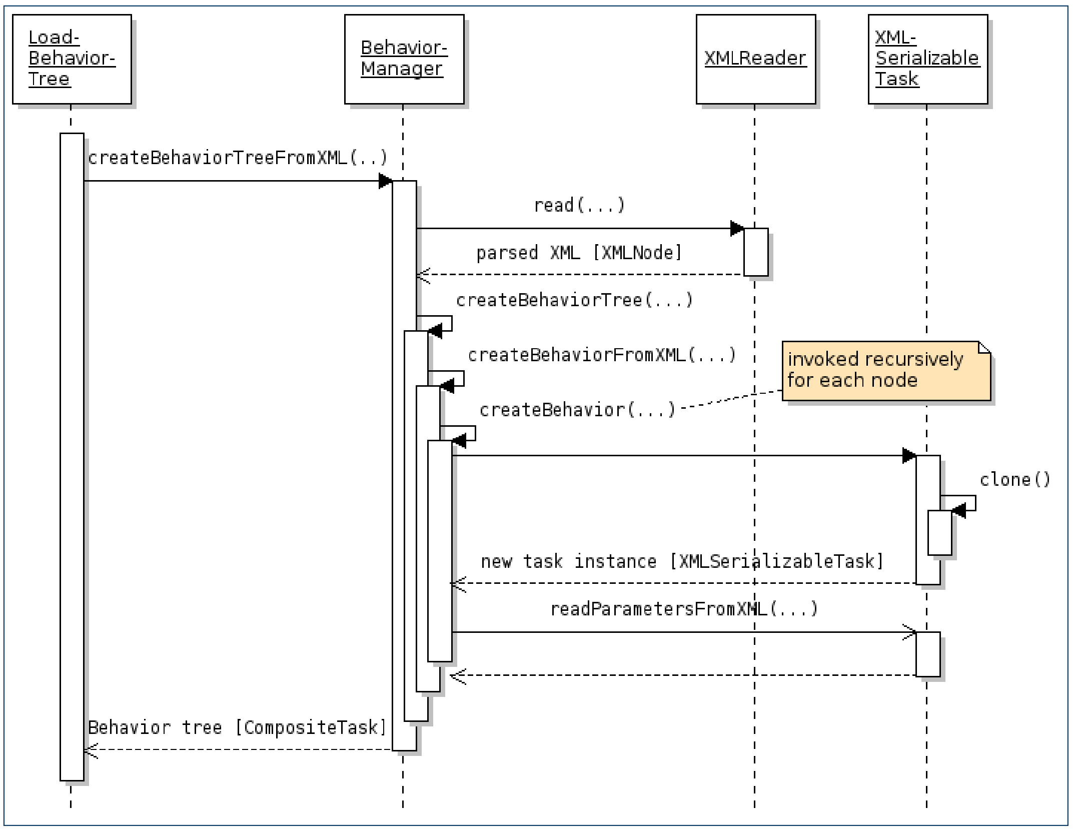

Load-Behavior-Tree
Behavior-Manager
XMLReader
XML-Serializable Task
createBehaviorTreeFromXML(..)
read(...)
parsed XML [XMLNode]
createBehaviorTree(...)
createBehaviorFromXML(...)
invoked recursively for each node
createBehavior(...)
clone()
new task instance [XMLSerializableTask]
readParametersFromXML(...)
Behavior tree [CompositeTask]

FIGURE A.2: *Sequence diagram of the process of deserialization*

The createBehaviorTreeFromXML function has the overall responsibility for deserialization. It invokes the XMLReader to parse the inputted XML, creates the root task and starts the recursive createBehaviorTree function. It traverses the XMLNode structure recursively, for each object instantiating the corresponding XMLSerializableTask object (through the createBehaviorFromXML and createBehavior functions) and inserts them into the behavior tree.

If we know that the parameter arguments of a behavior tree has been modified, but the structure of the tree has not, we do not need to build the object tree from scratch but can simply update the parameters. This could be done exactly as discussed above, only without calling createBehavior to instantiate new objects.

Just like we registered tasks, we can also register agents (or agent types) with the BehaviorManager. This can be used to bind available tasks to that agent. This way, the BehaviorManager can easily collect and send all template task data related to an agent simply by being provided the identification key of that agent (or agent type). An example hereof is shown below.

```
behaviorManager.registerAgentType("robot");
behaviorManager.bindBehaviorToAgent("robot", "gotoaction");
behaviorManager.bindBehaviorToAgent("robot", "waitaction");
behaviorManager.bindBehaviorToAgent("robot", "printaction");
behaviorManager.bindBehaviorToAgent("robot", "playanimation");
```

A.2 Unity web communication

The Unity Web Player has built-in support for communication between the Web Player and the web page it is embedded in. That is, Unity can invoke JavaScript functions on the web page and the web page can invoke functions defined in Unity. In this section we will look at how to implement this communication.

Unity has a function, Application.ExternalCall, for making external calls, i.e. to call a function on the web page on which the Web Player is embedded. It takes as argument the name of the function to call on the web page and a list of additional arguments.

```
Application.ExternalCall(webFunctionName, [arguments]); // Do external call
```

Executing the above code will result in calling the web page function which signature consists of the name defined by webFunctionName and the arguments defined by arguments. If no function matches the signature, no function is called. Suppose we have the following function defined in the web page embedding the Web Player.

```
<script type="text/javascript" language="javascript">
  <!--
  function debugPrint(phase, type, text) {
    // Implementation
  }
  -->
</script>
```

We could then call that web-based JavaScript function by using the following code from within Unity.

```
Application.ExternalCall(debugPrint, "load", 2, "Parsing complete");
```

As the external call matches the function signature of the `debugPrint` function this function will get called. The arguments for the function will be the string “load” as the `phase` variable, the integer 2 as `type` and the string “Parsing complete” as `text`. Suppose we have the following function in a game object called “Loader” inside the Unity object:

```
function loadBehavior(param : String) {
  // Implementation
}
```

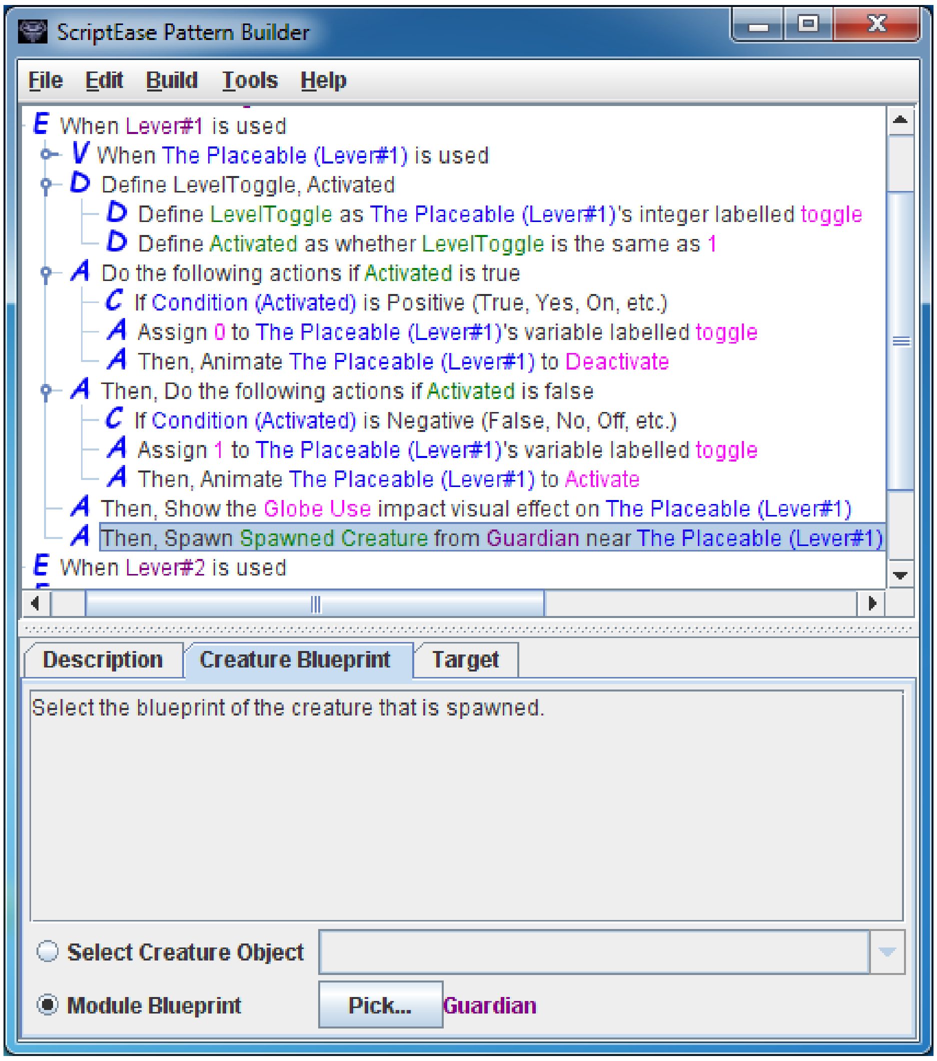
If we want to call the function from the web page we need to call the `SendMessage` function of the Unity Web Player object in the web page. This function takes three arguments: the name of a game object in the Unity scene, the name of a function within that object and a single string to hold arbitrary data.

```
unityObject.SendMessage(unityObjectName, unityFunctionName, data);
```

Specifically, we could use the code below:

```
unityObject.SendMessage("Loader", "loadBehavior", "<behavior></behavior>");
```

A.3 Design of other editor implementations

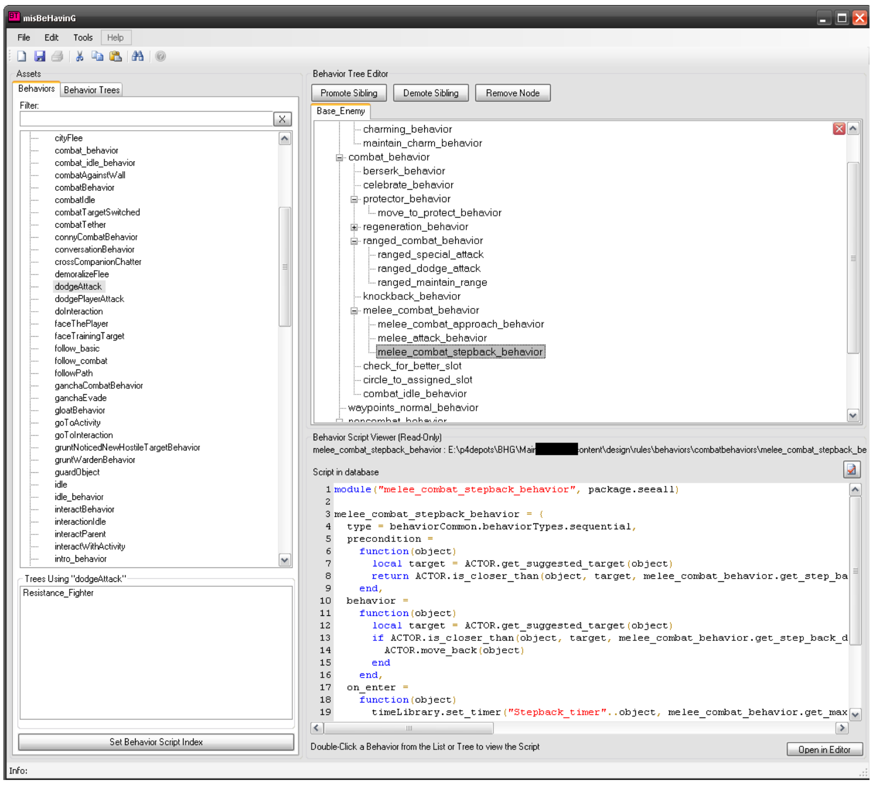


E When Lever#1 is used
V When The Placeable (Lever#1) is used
D Define LevelToggle, Activated
D Define LevelToggle as The Placeable (Lever#1)'s integer labelled toggle
D Define Activated as whether LevelToggle is the same as 1
A Do the following actions if Activated is true
C If Condition (Activated) is Positive (True, Yes, On, etc.)
A Assign 0 to The Placeable (Lever#1)'s variable labelled toggle
A Then, Animate The Placeable (Lever#1) to Deactivate
A Then, Do the following actions if Activated is false
C If Condition (Activated) is Negative (False, No, Off, etc.)
A Assign 1 to The Placeable (Lever#1)'s variable labelled toggle
A Then, Animate The Placeable (Lever#1) to Activate
A Then, Show the Globe Use impact visual effect on The Placeable (Lever#1)
A Then, Spawn Spawned Creature from Guardian near The Placeable (Lever#1)
E When Lever#2 is used

Select the blueprint of the creature that is spawned.

( ) Select Creature Object
[● Module Blueprint] Pick... Guardian

Figure A.3: *User interface of the "ScriptEase Pattern Builder", [7]*



```
cityFlee
combat_behavior
combat_idle_behavior
combatAgainstWall
combatBehavior
combatIdle
combatTargetSwitched
combatTether
connyCombatBehavior
conversationBehavior
crossCompanionChatter
demoralizeFlee
dodgeAttack
dodgePlayerAttack
doInteraction
faceThePlayer
faceTrainingTarget
follow_basic
follow_combat
followPath
ganchaCombatBehavior
ganchaEvade
gloatBehavior
goToActivity
goToInteraction
gruntNoticedNewHostileTargetBehavior
gruntWardenBehavior
guardObject
idle
idle_behavior
interactBehavior
interactionIdle
interactParent
interactWithActivity
intro_behavior
```

Trees Using "dodgeAttack"

Resistance_Fighter

```
      charming_behavior
      maintain_charm_behavior
  combat_behavior
      berserk_behavior
      celebrate_behavior
      protector_behavior
          move_to_protect_behavior
      regeneration_behavior
      ranged_combat_behavior
          ranged_special_attack
          ranged_dodge_attack
          ranged_maintain_range
      knockback_behavior
      melee_combat_behavior
          melee_combat_approach_behavior
          melee_attack_behavior
          melee_combat_stepback_behavior
      check_for_better_slot
      circle_to_assigned_slot
      combat_idle_behavior
  waypoints_normal_behavior
```

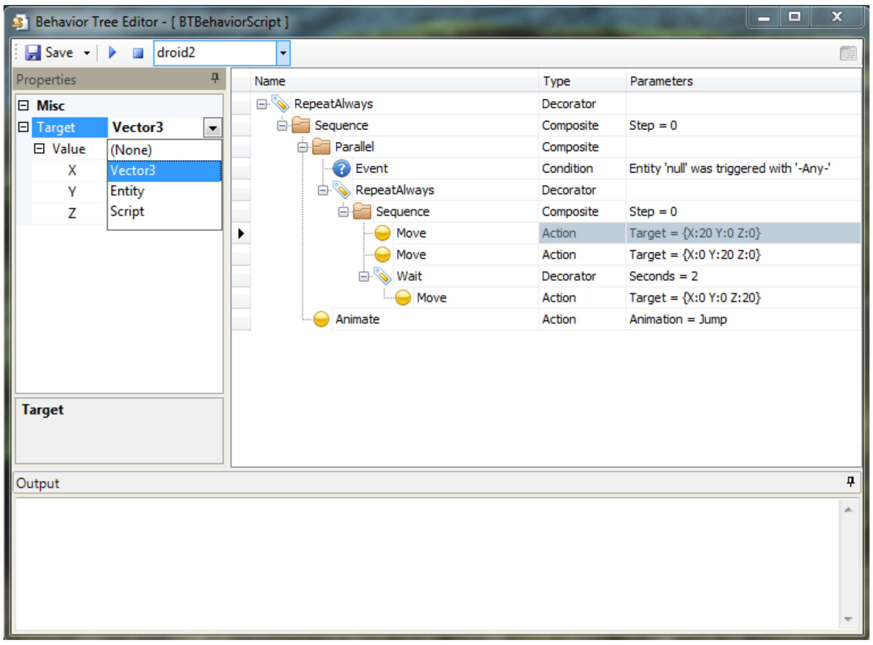
melee_combat_stepback_behavior : E:\p4depots\BHG\Mai[redacted]ontent\design\rules\behaviors\combatbehaviors\melee_combat_stepback_be

```
module("melee_combat_stepback_behavior", package.seeall)

melee_combat_stepback_behavior = {
  type = behaviorCommon.behaviorTypes.sequential,
  precondition =
    function(object)
      local target = ACTOR.get_suggested_target(object)
      return ACTOR.is_closer_than(object, target, melee_combat_behavior.get_step_ba
    end,
  behavior =
    function(object)
      local target = ACTOR.get_suggested_target(object)
      if ACTOR.is_closer_than(object, target, melee_combat_behavior.get_step_back_d
        ACTOR.move_back(object)
      end
    end,
  on_enter =
    function(object)
      timeLibrary.set_timer("Stepback_timer"..object, melee_combat_behavior.get_max
```

Double-Click a Behavior from the List or Tree to view the Script

FIGURE A.4: *User interface of the “misBeHavinG” behavior tree tool, [27, p. 36]*



Name	Type	Parameters
RepeatAlways	Decorator	
Sequence	Composite	Step = 0
Parallel	Composite	
Event	Condition	Entity 'null' was triggered with '-Any-'
RepeatAlways	Decorator	
Sequence	Composite	Step = 0
Move	Action	Target = {X:20 Y:0 Z:0}
Move	Action	Target = {X:0 Y:20 Z:0}
Wait	Decorator	Seconds = 2
Move	Action	Target = {X:0 Y:0 Z:20}
Animate	Action	Animation = Jump

FIGURE A.5: *User interface for the “Behavior Tree Editor” for the “Visual3D” game engine, [5]*

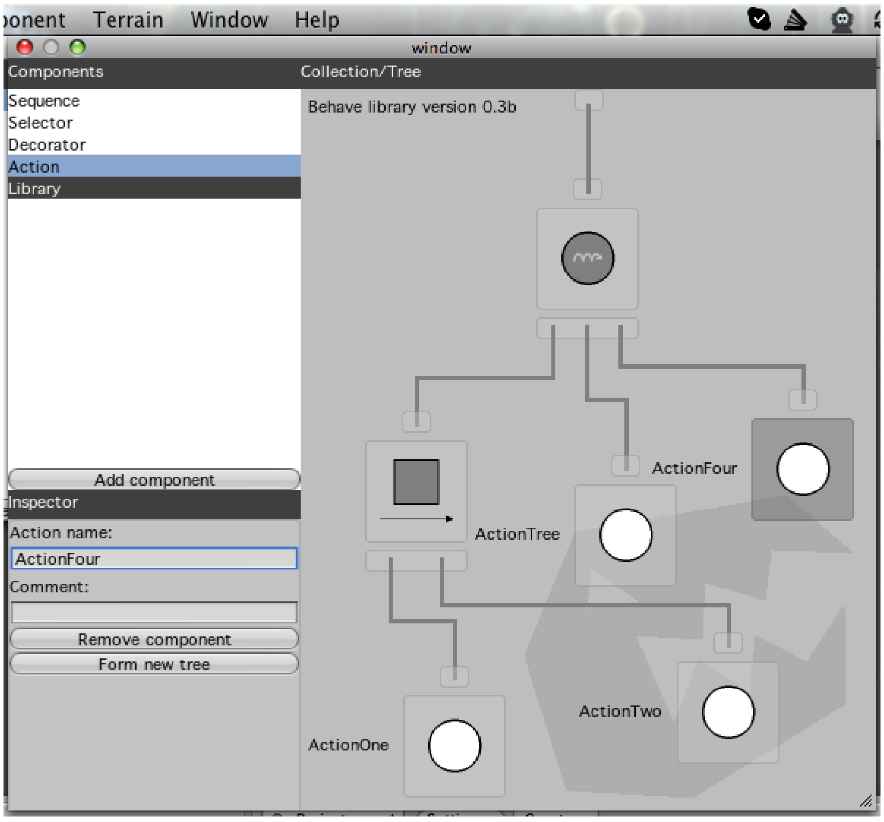


FIGURE A.6: *User interface of the "Behave" behavior tree editor integrated into Unity, [1]*

A.4 Code

This section contains code listings of some of the major classes referenced to in thesis. For the full code and the other resources refers to section 1.5.

A.4.1 Task

```
#pragma strict

enum TaskStatus {SUCCESS, FAILURE, RUNNING, ERROR};

class Task extends ScriptableObject {

  virtual function decide(agent : Agent) : boolean {
    return true;
  }

  virtual function execute(agent : Agent) : TaskStatus {
    return TaskStatus.SUCCESS;
  }

  virtual function activate(agent : Agent) : void {

  }

  virtual function deactivate(agent : Agent) : void {

  }

  virtual function isLeaf() : boolean {
    return true;
  }

  virtual function toString() : String {
    return getPrettyPrint(0);
  }

  virtual protected function getPrettyPrint(indent : int) : String {
    return String("-"[0], indent-1) + "Task";
  }

}
```

Listing A.1: *../code/unity/BehaviorTree/Task.js*

A.4.2 XMLSerializableTask

```
#pragma strict

class XMLSerializableTask extends Task {
  protected var xmlCodeName : String;
  protected var xmlName : String;
  protected var xmlType : String;
  protected var params : Boo.Lang.Hash;

  function XMLSerializableTask(codeName : String, name : String, type : String
      ) {
     xmlCodeName = codeName;
     xmlName     = name;
     xmlType     = type;
     params        = new Boo.Lang.Hash();
  }

  function getTemplateXML() : String {
    return "<task codename=\"" + getCodeName() + "\" name=\"" + getName() + "
        \" type=\"" + getType() + "\">\n" +
          getParameterXML() +
          "</" + getCodeName() + ">";
  }

  function getParameterXML() : String {
    var s : String = "";
  for (p in params) {
    var param : Parameter = p.Value;
    s += "<parameter name=\"" + p.Key + "\" value-type=\"" + param.valueType + "\" default=\"" + param.value + "\" />\n";
  }
  return s;
}

function readParametersFromXML(node : XMLNode) : boolean {
  for (p in params) {
    var param : Parameter = p.Value;
    param.value = readValueAttrFromNode(p.Key, param.valueType, node);
  }
  readParameters();
  return true;
}

function resetParameters() {
  for (p in params) {
    var param : Parameter = p.Value;
    param.value = param.defaultValue;
  }
  readParameters();
}

virtual function readParameters() {
  // place holder
}

function getDefaultParameterValue(key : String) : Object {
  if (params.ContainsKey(key)) {
    return params[key].defaultValue;
  }
  Utils.debugPrint("load", DebugType.ERROR, "getDefaultParameterValue: No parameter with key '" + key +"'");
  return null;
}

function getParameterValue(key : String) : Object {
  if (params.ContainsKey(key)) {
    return params[key].value;
  }
  Utils.debugPrint("load", DebugType.ERROR, "getParameterValue: No parameter with key '" + key +"'");
  return null;
}

function setParameterValue(key : String, value : Object) {
  if (params.ContainsKey(key)) {
    params[key].value = value;
    return;
  }
```

```
    Utils.debugPrint("load", DebugType.ERROR, "setParameterValue: No parameter with key '" + key +"'");
  }

  function registerParameter(paramName, paramValue) {
    params[paramName] = new Parameter(getValueTypeString(paramValue), paramValue);
  }

  private function getValueTypeString(val) : String {
    var typeString : String = "";
    switch (typeof(val)) {
      case System.Int32:       typeString = "integer"; break;
      case System.Single:      typeString = "float"; break;
      case System.String:      typeString = "string"; break;
      case System.Boolean:     typeString = "boolean"; break;
      case UnityEngine.Vector2:    typeString = "vector2"; break;
      case UnityEngine.Vector3:    typeString = "vector3"; break;
      case UnityEngine.Vector4:    typeString = "vector4"; break;
      default:            typeString = "NA"; Utils.debugPrint("BehaviorToXML", DebugType.ERROR, "getValueTypeString unknown type of value: " + typeof(val)); break;
    }
    return typeString;
  }

  private function readValueAttrFromNode(key : String, valueType : String, node : XMLNode) {
    var val;
    switch (valueType) {
      case "int":    val = node.getIntFromAttr(key); break;
      case "float":    val = node.getFloatFromAttr(key); break;
      case "string":   val = node.getStringFromAttr(key); break;
      case "boolean": val = node.getBoolFromAttr(key); break;
      case "vector2": val = node.getVector2FromAttr(key); break;
      case "vector3": val = node.getVector3FromAttr(key); break;
      case "vector4": val = node.getVector4FromAttr(key); break;
      default:     Utils.debugPrint("XMLToBehavior", DebugType.ERROR, "readValueAttrFromNode unknown type of value: " + valueType); break;
    }
    return val;
  }

  virtual function getCodeName() : String { return xmlCodeName; }
  virtual function getName() : String     { return xmlName; }
  virtual function getType() : String     { return xmlType; }

  virtual protected function getPrettyPrint(indent : int) : String {
    return getName();
  }

  virtual function clone() : XMLSerializableTask {
    return null;
  }
}
```

Listing A.2: *../code/unity/BehaviorTree/XMLSerializableTask.js*

A.4.3 CompositeTask

```

class CompositeTask extends XMLSerializableTask {

  public var components : Array;
  protected var runningTask : Task;

  function CompositeTask(codeName : String, name : String, type : String) {
    super(codeName, name, type);
    components = new Array();
    runningTask = null;
  }

  virtual function decide(agent : Agent) : boolean {
    // Accepts if at least one child accepts
    for (var task : Task in components) {
      if (task.decide(agent)) {
        return true;
      }
    }
  }

  virtual function execute(agent : Agent) : TaskStatus {
    return TaskStatus.ERROR; // cannot be run on its own
  }

  virtual function executeChild(task : Task, agent : Agent) {
    // If the executed task is different from the running task ensure that the
        running task is gracefully deactivated
    if (task != runningTask) {
      if (runningTask) runningTask.deactivate(agent);
      task.activate(agent);
    }

    var status : TaskStatus = task.execute(agent);

    // If the task finished executing, deactivate it. Otherwise, flag it as
      the currently running task
    if (status != TaskStatus.RUNNING) {
      task.deactivate(agent);
      runningTask = null;
    } else {
      runningTask = task;
    }

    return status;
  }

  virtual function addComponent(c : Task) {
    components.Add(c);
  }

  virtual function removeComponent(c : Task) {
    components.Remove(c);
  }

  virtual function isLeaf() : boolean {
    return false;
  }

  virtual protected function getPrettyPrint(indent : int) : String {
    indent++;
    var s : String = getName() + "\n";
    for (var child : Task in components) {
      s += String("-"[0], indent) + " " + child.getPrettyPrint(indent) + "\n";
    }
    return s;
  }
}
```

Listing A.3: *../code/unity/BehaviorTree/CompositeTask.js*

A.4.4 Sequence

```

class Sequence extends CompositeTask {

  function Sequence() {
    super("sequence", "Sequence", "composite");
  }

  virtual function decide(agent : Agent) : boolean {
    // We only need to validate the overall sequence as CompositeTask do not
        validate the decide() of running tasks (as per the SPORE article)
    // Accepts if the first child accepts
    for (var task : Task in components) {
      return task.decide(agent);
    }
  }

  virtual function execute(agent : Agent) {
    // Iterate through the children and pick the one that:
    //   1) Is already running
    //   2) Is not running but whose decide() function accepts
    for (var task : Task in components) {
      // If there is a running task, skip children until we reach the running task
      // If there is not a running task, skip children who do not accept
      if ((runningTask && task != runningTask) || !task.decide(agent)) continue; // Skip non-running non-accepting tasks

      // At this point in execution we have either the running task or a non-running but accepting task

      var status : TaskStatus = executeChild(task, agent);

      // If the task either fails or continues to run we are finished. Otherwise we continue to the next child.
      if ((status == TaskStatus.FAILURE) || (status == TaskStatus.RUNNING)) {
        return status;
      }

    }

    return TaskStatus.SUCCESS;
  }

  virtual function clone() : XMLSerializableTask {
    return new Sequence();
  }

}
```

LISTING A.4: *../code/unity/BehaviorTree/Sequence.js*

A.4.5 Selector

```

class Selector extends CompositeTask {

  function Selector() {
    super("selector", "Selector", "composite");
  }

  virtual function execute(agent : Agent) {
    for (var task : Task in components) {
      if (runningTask) {                // If we have a currently running task we...
        if (task != runningTask) continue; //   skip children until we reach the running task
      } else {                          // If we do not have a running task we...
        if (!task.decide(agent)) continue; //   skip children who do not accept
      }

      // At this point in execution we have either the running task or a non-running but accepting task

      var status : TaskStatus = executeChild(task, agent);

      if (status != TaskStatus.FAILURE) return status;
    }

    return TaskStatus.FAILURE;
  }

  virtual function clone() : XMLSerializableTask {
    return new Selector();
  }

}
```

LISTING A.5: *../code/unity/BehaviorTree/Selector.js*

A.4.6 Parallel

```
#pragma strict

class Parallel extends CompositeTask {
  var succeedOnOne : boolean  = false;
  var failOnOne : boolean  = false;

  function Parallel() {
    super("parallel", "Parallel", "composite");
  }

  virtual function execute(agent : Agent) {
    var allSucceeded : boolean = true;
    var allFailed : boolean = true;
    var isRunning : boolean = false;

    for (var task : Task in components) {
      var status : TaskStatus = task.execute(agent);

      if (status == TaskStatus.SUCCESS) {
        if (succeedOnOne) return TaskStatus.SUCCESS;
        allFailed = false;
      } else if (status == TaskStatus.FAILURE) {
        //if (failOnOne || !succeedOnOne) return TaskStatus.FAILURE;
        if (failOnOne) return TaskStatus.FAILURE;
        allSucceeded = false;
      } else if (status == TaskStatus.ERROR) {
        return TaskStatus.ERROR;
      } else if (status == TaskStatus.RUNNING) {
        allFailed = false;
        allSucceeded = false;
        isRunning = true;
      }
    }

    if (isRunning) return TaskStatus.RUNNING;
    if (allSucceeded) return TaskStatus.SUCCESS;
    if (allFailed) return TaskStatus.FAILURE;
    return TaskStatus.ERROR;
  }

  virtual function clone() : XMLSerializableTask {
    return new Parallel();
  }

}
```

LISTING A.6: *../code/unity/BehaviorTree/Parallel.js*

A.4.7 PrioritizedList

```
#pragma strict

class PrioritizedList extends CompositeTask {

  function PrioritizedList() {
    super("prioritizedlist", "Prioritized List", "composite");
  }

  virtual function execute(agent : Agent) : TaskStatus {
    // Iterate through the list and pick the first task that is:
    //   1) a task with higher priority than the active task whose decide()
        accepts
    //   2) the active task (without calling decide())
    for (var task : Task in components) {
      if (task != runningTask && !task.decide(agent)) continue;

      var status : TaskStatus = executeChild(task, agent);
      if (status != TaskStatus.FAILURE) {
        return status;
      }
    }

    return TaskStatus.SUCCESS;
  }

  virtual function clone() : XMLSerializableTask {
    return new PrioritizedList();
  }

}
```

LISTING A.7: *../code/unity/BehaviorTree/PrioritizedList.js*

A.4.8 Decorator

```

class TaskDecorator extends CompositeTask {
  protected var decoratedTask : Task = null;

  function TaskDecorator(codeName : String, name : String, type : String) {
    super(codeName, name, type);
  }

  virtual function addComponent(task : Task) {
    if (decoratedTask) Debug.Log("[TaskDecorator::addComponent] WARNING:
        Overwriting already decorated task!");
    components.Clear();
    components.Add(task);
    decoratedTask = task;
  }

  virtual function execute(agent : Agent) {
    if (!decoratedTask) {
      Debug.Log("[TaskDecorator::execute] ERROR: No decorated task!");
      return TaskStatus.ERROR;
    }
    return decoratedTask.execute(agent);
  }
}
```

LISTING A.8: *../code/unity/BehaviorTree/TaskDecorator.js*

A.4.9 BehaviorManager

```

class BehaviorManager {

  private var behaviors : Boo.Lang.Hash;
  private var agentTypes : Boo.Lang.Hash;

  function BehaviorManager() {
    behaviors = new Boo.Lang.Hash();
    agentTypes = new Boo.Lang.Hash();

    registerBehavior(Sequence());
    registerBehavior(Selector());
    registerBehavior(Parallel());
    registerBehavior(PrioritizedSequence());
    registerBehavior(PrioritizedSelector());

    registerBehavior(GoToAction());
    registerBehavior(WaitAction());
    registerBehavior(PrintAction());
    registerBehavior(PlayAnimation());
```

```
}

function registerAgentType(agentType : String) {
  if (agentTypes.ContainsKey(agentType)) {
    Utils.debugPrint("load", DebugType.WARNING, "'" + agentType +"' agent type already registered");
    return;
  }
  agentTypes[agentType] = new Array();
}

function bindBehaviorToAgent(agentType : String, behavior : String) {
  if (!agentTypes.ContainsKey(agentType)) {
    Utils.debugPrint("load", DebugType.WARNING, "'" + agentType +"' agent type not registered");
    return;
  }
  if (!behaviors.ContainsKey(behavior)) {
    Utils.debugPrint("load", DebugType.WARNING, "'" + behavior +"' behavior not registered");
    return;
  }
  var arr : Array = agentTypes[agentType];
  arr.Add(behavior);
}

function printAgentBehaviors(agentType : String) {
  for (var behavior in agentTypes[agentType]) {
    Debug.Log("Behavior: " + behavior);
  }
}

function registerBehavior(xmlTask : XMLSerializableTask) {
  behaviors[xmlTask.getCodeName()] = xmlTask;
}

function createBehavior(codename : String) : XMLSerializableTask {
  if (behaviors.ContainsKey(codename)) {
    return behaviors[codename].clone();
  }
  Utils.debugPrint("load", DebugType.WARNING, "No behavior with code name '" + codename +"'");
  return null;
}

function createBehaviorFromXML(node : XMLNode) : XMLSerializableTask {
  var newTask : XMLSerializableTask = createBehavior(node.tagName);
  if (newTask) newTask.readParametersFromXML(node);
  return newTask;
}

function createBehaviorTreeFromXML(xmlData : String) : CompositeTask {
    Utils.debugPrint("load", DebugType.INFO, "Reading and parsing XML and
        building the behavior tree...");
    var parser : XMLReader = XMLReader();
    var node : XMLNode = parser.read(xmlData);

    var behavior : CompositeTask = new Sequence();
    createBehaviorTree(node, behavior);

    Utils.debugPrint("load", DebugType.INFO, "Behavior Tree loaded");

    return behavior;
  }

  private function createBehaviorTree(node : XMLNode, treeNode : CompositeTask
      ) : CompositeTask {
    var task : Task = null;
    var newComposite : CompositeTask = treeNode;

    Utils.debugPrint("load", DebugType.INFO, "Creating '" + node.tagName + "'
        behavior...");

    task = createBehaviorFromXML(node);

    // If we created a new task
    if (task) {
      // Add the task to the parent node
      treeNode.addComponent(task);
      // If the task is a composite task, set it as the parent for the
          recursion
      if (!task.isLeaf()) {
        newComposite = task;
      }
    }

    for (var n : XMLNode in node.children) {
      createBehaviorTree(n, newComposite);
    }

    return treeNode;
  }

}
```

Listing A.9: *../code/unity/BehaviorTree/BehaviorManager.js*

Bibliography

[1] AngryAnt: Behave. http://angryant.com. Accessed July 15 2010.

[2] Sencha – Ext JS – Client-side JavaScript Framework. http://www.sencha.com/products/js. Accessed August 18 2010.

[3] The Programming Language Lua. http://www.lua.org. Accessed August 5 2010.

[4] UNITY: Game Development Tool. http://unity3d.com. Accessed August 12 2010.

[5] Behavior Tree Editor for State Machines and Sequences. http://game-engine.visual3d.net/wiki/behavior-tree-editor-state-machines-and-sequences, December 2009. Accessed August 17 2010.

[6] Pac-Man Physics. http://www.indiegames.com/blog/2009/02/freeware_game_pick_pacman_phys.html, February 2009. Accessed May 4 2010.

[7] University of Alberta: ScriptEase. http://webdocs.cs.ualberta.ca/~script, september 2009. Accessed July 15 2010.

[8] EA Press. http://info.ea.com, 2010. Accessed March 25 2010.

[9] XML. http://en.wikipedia.org/w/index.php?title=XML&oldid=367508169, 2010. Accessed 15 June 2010.

[10] V. Braitenberg. Experiments in synthetic psychology, 1984.

[11] R. Brooks. A robust layered control system for a mobile robot. *IEEE journal of robotics and automation*, 2(1):14–23, 1986.

[12] R.A. Brooks. Elephants don't play chess. *Robotics and autonomous systems*, 6(1-2):3–15, 1990.

[13] M. Buckland. *Programming game AI by example*. Wordware, 2005.

[14] Alex J. Champandard. Enabling Concurrency in Your Behavior Heirarchy. http://aigamedev.com/open/articles/parallel, july 2007. Accessed August 7 2010.

[15] Alex J. Champandard. Living with The Sims' AI: 21 Tricks to Adopt for Your Game. http://aigamedev.com/open/highlights/the-sims-ai, October 2007. Accessed March 25 2010.

[16] Alex J. Champandard. The Flexibility of Selectors for Hierarchical Logic. http://aigamedev.com/open/articles/selector, july 2007. Accessed July 7 2010.

[17] Alex J. Champandard. The Gist of Hierarchical FSM. http://aigamedev.com/open/articles/hfsm-gist, september 2007. Accessed July 7 2010.

[18] Alex J. Champandard. The Power of Sequences for Hierarchical Behaviors. http://aigamedev.com/open/articles/sequence, july 2007. Accessed July 7 2010.

[19] Alex J. Champandard. Understanding Behavior Trees. http://aigamedev.com/open/articles/bt-overview, september 2007. Accessed July 7 2010.

[20] Alex J. Champandard. Using Decorators to Improve Behaviors. http://aigamedev.com/open/articles/decorator, july 2007. Accessed July 7 2010.

[21] Alex J. Champandard. Using Resource Allocators to Synchronize Behaviors. http://aigamedev.com/open/articles/allocator, july 2007. Accessed August 7 2010.

[22] Alex J. Champandard. What Does a Behavior Tree Editor Look Like? http://aigamedev.com/open/articles/behavior-tree-editor-example, December 2007. Accessed August 17 2010.

[23] Alex J. Champandard. Behavior Trees for Next-Gen AI. page 96, 2008.

[24] Alex J. Champandard. SHOP: Simple Hierarchical Ordered Planner. http://aigamedev.com/open/reviews/shop-htn, feburary 2008. Accessed July 14 2010.

[25] Alex J. Champandard. Big Mistakes of New AI Engineers. http://aigamedev.com/open/coverage/isla-principle-engineer-mistakes, december 2009. Accessed June 22 2010.

[26] Alex J. Champandard. This Year's Brightest Games: 2009 AiGameDev.com Awards for Game AI. http://aigamedev.com/open/editorial/2009-awards-results, December 2009. Accessed April 14 2010.

[27] Alex J. Champandard, Michael Dawe, and David Hernandez-Cerpa. Building a Better Battle: The Halo 3 AI Objectives System. page 44. Bungie Studios, 2008.

[28] Jack Copeland. A Brief History of Computing. http://www.alanturing.net/turing_archive/pages/Reference%20Articles/BriefHistofComp.html, 2000. Accessed March 15 2010.

[29] M. Cutumisu and D. Szafron. An Architecture for Game Behavior AI: Behavior Multi-Queues. 2009.

[30] D.S.C. Dalmau and D. Sanchez-Crespo. *Core techniques and algorithms in game programming*. New Riders Games, 2003.

[31] Luke Dicken. HCSM: A Framework for Behavior and Scenario Control in Virtual Environments. http://aigamedev.com/open/reviews/hcsm-concurrent-state-machine, December 2009. Accessed July 6 2010.

[32] D. Fu and R. Houlette. The Ultimate Guide to FSMs in Games. *AI Game Programming Wisdom*, 2, 2003.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[34] A. Gordon. Tough Love Between Artificial Intelligence and Interactive Entertainment. 2004.

[35] Chris Hecker. My Liner Notes for Spore. http://chrishecker.com/My_Liner_Notes_for_Spore, may 2009. Accessed July 15 2010.

[36] Chris Hecker. Spore Behavior Tree Docs. http://chrishecker.com/My_Liner_Notes_for_Spore/Spore_Behavior_Tree_Docs, april 2009. Accessed July 15 2010.

[37] Damian Isla. Handling Complexity in the *Halo 2* AI. http://www.gamasutra.com/gdc2005/features/20050311/isla_pfv.htm, March 2005. Accessed May 11 2010.

[38] A. Khoo, G. Dunham, N. Trienens, and S. Sood. Efficient, realistic NPC control systems using behavior-based techniques. 2002.

[39] L. Lidén. Artificial stupidity: The art of intentional mistakes. *AI Game Programming Wisdom*, 2:41–48, 2003.

[40] F.G. Martin. *Robotic Explorations: A Hands-On Introduction to Engineering*. Prentice Hall Upper Saddle River, USA, 2000.

[41] J.C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, Inc. New York, NY, USA, 2002.

[42] I. Millington and J. Funge. *Artificial intelligence for games*. Morgan Kaufmann Pub, 2009.

[43] Jeff Orkin. Applying Goal-Oriented Action Planning to Games. 2003.

[44] Combine Overwiki. Hazardous Environment Combat Unit. http://half-life.wikia.com/wiki/HECU, April 2010. Accessed April 14 2010.

[45] Jamey Pittman. Pac-Man Physics. http://home.comcast.net/~jpittman2/pacman/pacmandossier.html, February 2009. Accessed May 4 2010.

[46] S. Rabin. Common game AI techniques. *AI game programming wisdom*, 2:3–24, 2004.

[47] S. Rabin. Promising game AI techniques. *AI Game Programming Wisdom*, 2:15–27, 2004.

[48] Craig Reynolds. Boids (Flocks, Herds, and Schools: a Distributed Behavioral Model). http://www.red3d.com/cwr/boids, September 2001. Accessed April 14 2010.

[49] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34. ACM, 1987.

[50] S.J. Russell and P. Norvig. *Artificial intelligence: A Modern Approach, Second Edition*. Prentice hall Englewood Cliffs, NJ, 2003.

[51] Joe Sapp. Working Group On Rule-Based Systems. http://www.igda.org/aireport/2003/rbs, november 2009. Accessed July 8 2010.

[52] Valve Software. Half-Life. http://www.valvesoftware.com, 2010. Accessed April 14 2010.

[53] Paul T. Fixing Pathfinding Once and For All. http://www.ai-blog.net/archives/000152.html, july 2006. Accessed July 14 2010.

[54] AS Tanenbaum. *Structured computer organization; 5th*. Upper Saddle River, NJ: Pearson Prentice Hall, Ed, 2006.

[55] Wikipedia. Age of Empires. http://en.wikipedia.org/w/index.php?title=Age_of_Empires&oldid=375279564, 2010. Accessed August 4 2010.

[56] Wikipedia. Artificial intelligence. http://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=376898424, 2010. Accessed August 3 2010.

[57] Wikipedia. Finite-state machine. http://en.wikipedia.org/w/index.php?title=Finite-state_machine&oldid=371375941, 2010. Accessed July 6 2010.

[58] Wikipedia. Frame rate. http://en.wikipedia.org/w/index.php?title=Frame_rate&oldid=376790834, 2010. Accessed August 4 2010.

[59] Wikipedia. Game artificial intelligence. http://en.wikipedia.org/w/index.php?title=Game_artificial_intelligence&oldid=373547314, 2010. Accessed August 1 2010.

[60] Wikipedia. Game complexity. http://en.wikipedia.org/w/index.php?title=Game_complexity&oldid=368644577, 2010. Accessed July 13 2010.

[61] Wikipedia. Game development. http://en.wikipedia.org/w/index.php?title=Game_development&oldid=360227108, 2010. Accessed May 5 2010.

[62] Wikipedia. Game development tool. http://en.wikipedia.org/w/index.php?title=Game_development_tool&oldid=332227161, 2010. Accessed August 20 2010.

[63] Wikipedia. Halo 2. http://en.wikipedia.org/w/index.php?title=Halo_2&oldid=377358386, 2010. Accessed August 8 2010.

[64] Wikipedia. History of video games. http://en.wikipedia.org/w/index.php?title=History_of_video_games&oldid=359709858, 2010. Accessed May 5 2010.

[65] Wikipedia. Mod (computer gaming). http://en.wikipedia.org/w/index.php?title=Mod_(computer_gaming)&oldid=378329459, 2010. Accessed August 12 2010.

[66] Wikipedia. Non-player character. http://en.wikipedia.org/w/index.php?title=Non-player_character&oldid=377432237, 2010. Accessed August 20 2010.

[67] Wikipedia. Pac-Man. http://en.wikipedia.org/w/index.php?title=Pac-Man&oldid=376121540, 2010. Accessed August 6 2010.

[68] Wikipedia. Pathfinding. http://en.wikipedia.org/w/index.php?title=Pathfinding&oldid=361408820, 2010. Accessed July 14 2010.

[69] Wikipedia. The Sims. http://en.wikipedia.org/w/index.php?title=The_Sims&oldid=376280495, 2010. Accessed August 2 2010.

[70] Wikipedia. Video game. http://en.wikipedia.org/w/index.php?title=Video_game&oldid=360236749, 2010. Accessed May 5 2010.

[71] Steven Woodcock. Game AI: The State of the Industry. http://www.gamasutra.com/view/feature/3570/game_ai_the_state_of_the_industry.php, November 2000. Accessed March 25 2010.

[72] B. Yue and P. de Byl. The state of the art in game AI standardisation. In *Proceedings of the 2006 international conference on Game research and development*, page 46. Murdoch University, 2006.

Index