Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica

Universitat de Barcelona

DEVELOPMENT OF A STEALTH GAME WITH MULTI-AGENT ARTIFICIAL INTELLIGENCE

Albert Espín Román

Director: Francesc Xavier Dantí Espinasa

Done in: Departament de Matemàtiques i Informàtica

Academic year 2017-2018 (2nd semester)

Abstract

Artificial intelligence is used in video games to create believable characters that challenge the player and enrich the playing experience. The stealth genre is one of the types of games where it is fundamental to have non-player characters with the ability to perceive the environment, detect some of the player character's actions and react to them with astute strategies.

The goal of this project was to develop a stealth game with enemy characters that would cooperate to defeat the player. In the final game, the enemies patrol the game's world trying to locate the player character, and they investigate any potential signs of player actions in the environment, such as objects in motion or bodies of unconscious characters on the floor. The findings of an enemy are immediately communicated to the rest of agents, and the player character is chased and attacked by the enemies after being spotted. While some of the agents pursue the player character and use close combat moves, others reduce speed to shoot the character in the head with precision from distant positions.

The game was developed with the Unreal Engine 4 game engine. The reasoning and decision-making of enemies was modelled using behaviour trees. The implemented system makes agents capable of perceiving visual and auditory stimuli and communicating information to other agents, who compare it with their own knowledge to take independent decisions. The whole set of enemies form an artificial intelligence-based multi-agent system and cooperate to achieve the shared goal of defeating the player in the shortest time possible.

The ability of the numerous enemies to spot the player character and detect sounds penalizes noisy and reckless players, who are rapidly defeated. Unlike them, those players who struggle to be silent and elusive have a reasonable chance to win the game, approaching enemies from behind, knocking them out stealthily, and hiding the bodies to avoid leaving traces that other agents would try to use to find the player character. Stones and other objects can be thrown to make noises and distract the enemies. Nevertheless, the agents' ability to locate the player character should not be underestimated: no hidden location is permanently safe.

Resumen

La inteligencia artificial se aplica en los videojuegos para crear personajes creíbles que desafían las habilidades del jugador y enriquecen la experiencia jugable. En algunos casos, como en los juegos de siglo, es fundamental disponer de personajes controlados por el juego capaces de percibir el entorno, detectar algunas de las acciones del jugador y reaccionar con astucia ante ellas.

El objetivo de este proyecto ha sido el desarrollo de un videojuego de sigilo con personajes enemigos capaces de cooperar para derrotar al jugador. Los enemigos patrullan por el mundo del juego intentando localizar al personaje del jugador, investigando cualquier indicio de actividades sospechosas en el entorno, como objetos en movimiento o cuerpos de personajes inconscientes tendidos sobre el suelo. Cada enemigo es capaz de comunicar sus hallazgos al resto de agentes de forma inmediata; entre todos persiguen y atacan al personaje del jugador cuando algún agente lo avista. Algunos enemigos se centran en perseguir al personaje del jugador y atacarlo con técnicas de combate cuerpo a cuerpo, mientras que otros reducen la velocidad para poder dispararle en la cabeza con precisión desde la distancia.

El juego ha sido desarrollado con el motor de juego Unreal Engine 4. Las capacidades de razonar y tomar decisiones de los enemigos se han modelado usando árboles de comportamiento. El sistema implementado permite a los enemigos percibir estímulos visuales y auditivos y comunicar la información obtenida de tales percepciones al resto de agentes, que la comparan con sus propios conocimientos para tomar decisiones de forma independiente. El conjunto de enemigos forma un sistema multiagente basado en inteligencia artificial cuyos integrantes cooperan para lograr el objetivo compartido de derrotar al jugador en el menor tiempo posible.

La habilidad del numeroso grupo de enemigos para avistar al personaje del jugador y detectar sonidos penaliza a los jugadores más ruidosos y temerarios, que son rápidamente abatidos. Por el contrario, aquellos jugadores que se esfuerzan por ser silenciosos y escurridizos tienen posibilidades razonables de ganar el juego, aproximándose a los enemigos por la espalda para abatirlos sigilosamente y escondiendo sus cuerpos, para evitar dejar rastros que otros agentes podrían usar para encontrar al personaje del jugador. Se pueden lanzar piedras y otros objetos para hacer ruido e intentar distraer a los enemigos. No obstante, no se debería infravalorar la capacidad de los agentes para localizar al personaje del jugador: ningún escondite es permanentemente seguro en el juego.

Resum

La intel·ligència artificial s'aplica als videojocs per crear personatges creïbles que desafien les habilitats del jugador i enriqueixen la seva experiència. Alguns jocs, com ara els del gènere del sigil, utilitzen i controlen personatges capaços de percebre l'entorn, detectar algunes de les accions del jugador i estudiar-les per reaccionar de forma astuta.

L'objectiu d'aquest projecte ha estat el desenvolupament d'un videojoc de sigil amb personatges enemics capaços de cooperar per derrotar el jugador. Els enemics patrullen pel món del joc intentant localitzar el personatge del jugador, investigant qualsevol indici d'activitats sospitoses a l'entorn, com ara objectes en moviment o cossos de personatges inconscients a terra. Cada enemic és capaç de comunicar les seves troballes a la resta d'agents amb immediatesa; entre tots persegueixen i ataquen el personatge del jugador quan algun dels agents el detecta. Alguns enemics se centren en perseguir el personatge del jugador i atacar-lo amb tècniques de combat cos a cos, mentre que d'altres redueixen la velocitat per poder disparar-li al cap amb precisió des d'una certa distància.

El joc ha estat desenvolupat amb el motor de joc Unreal Engine 4. Les capacitats de raonar i prendre decisions dels enemics han estat modelades fent servir arbres de comportament. El sistema implementat permet als enemics percebre estímuls visuals i auditius i comunicar la informació obtinguda d'aquestes percepcions als altres agents, que la comparen amb els seus propis coneixements per prendre decisions de forma independent. El conjunt d'enemics forma un sistema multiagent basat en intel·ligència artificial, els integrants del qual cooperen per assolir l'objectiu comú de derrotar el jugador en el menor temps possible.

L'habilitat del nombrós grup d'enemics per identificar visualment el personatge del jugador i detectar els seus sons penalitza els jugadors més sorollosos i temeraris, que són ràpidament derrotats. En canvi, aquells jugadors que procuren ser silenciosos i prudents tenen possibilitats de guanyar el joc, apropant-se als enemics per l'esquena per neutralitzar-los amb sigil, amagant posteriorment els seus cossos per evitar donar pistes als altres agents, que podrien fer-les servir per trobar el personatge del jugador. Es poden llençar objectes com ara pedres per fer soroll i intentar distreure els enemics. No obstant això, no s'hauria d'infravalorar la capacitat dels agents per localitzar el personatge del jugador: cap amagatall no és permanentment segur en aquest joc.

Table of Contents

1. Introduction

1.1. Introduction to artificial intelligence in stealth games

Artificial intelligence (AI) can be defined as the discipline that studies the design and implementation of intelligent agents. Agents are entities that take actions in an environment they perceive. Intelligent agents are those agents that have goals and choose their actions in such way that the likelihood of accomplishing their objectives is maximized, to the extent permitted by their knowledge and perceptual limitations[1]. Some intelligent agents are called natural agents because they can be found in nature, such as humans and some other animal species, whilst those agents that are created by humans are artificial agents. The artificial agents that reside in a material body that they can control, at least in a partial way, are identified as physical agents, while software agents are those that exist in the context of a program, e.g. a video game, and cannot directly modify the physical world. Hereinafter, the use of the agent term and its plural form, agents, will refer to software-based artificial intelligence agents unless otherwise stated (e.g. using an adjective before the term).

The presence of artificial intelligence in video games is commonplace, but while many games use algorithms to power game logic, not all the techniques fit in the category of artificial intelligence. A notable case for discussion is that of non-player characters (NPCs), which are characters controlled by the game, not a human player. Some NPCs share the same objectives as the player and therefore they are considered allies. Other characters have objectives that are not directly opposing to those of the player, or they even have no objectives at all, so they are called neutrals. A third category of NPCs have goals that are antagonistic to the player objectives, so they can be thought of as enemies. Some NPCs do not fit in the definition of intelligent agents given before, since they either do not have objectives or do not take decisions that maximize the chance of fulfilling their goals. For example, this is the case of characters that wander around a location without an objective. Unlike them, some NPCs have goals and use standard artificial intelligence techniques, such as shortest-path search (to find the optimal way to solve a problem according to some evaluation criteria) and behaviour trees (systems used to analyse the environment and decide to do certain tasks in an organized yet flexible way, in the attempt to reach their goals), so they are considered intelligent agents. Hereinafter, intelligent NPCs will be referred to as NPCs.

There are some contexts in which different intelligent agents that exist in a shared environment can interact with each other, forming a multi-agent system. Agents in a multi-agent system may compete to reach opposing objectives (in which case they can opt for direct confrontation or negotiation, for example) or cooperate to achieve similar or identical goals, eventually forming alliances or coalitions. In video games, it is common to see multi-agent systems formed by enemy NPCs that coordinate to defeat the player (e.g. attacking the player with weapons). They rely on individual perceptions to obtain information from the environment and use communication to share it with other agents. This is a crucial step in the process of building

[1] The definitions of artificial intelligence and intelligent agents used in this document are based on: POOLE, David; MACKWORTH, Alan; GOEBEL, Randy. *Computational Intelligence: A Logical Approach*, Chapter 1, p. 1. New York: Oxford University Press, Jan. 1998.

common knowledge that helps agents decide what to do, and refines their overall understanding of the environment, and particularly the comprehension of the player character's actions or activities. Since the agents' goal is to defeat the player with some optimality criteria (e.g. in the shortest time possible), it is common to see them organize, divide work and take different roles to try to succeed with maximum likelihood. However, computation resources are limited, and calculus takes place in real time, so members of multi-agent systems in games do not always have the time to ensure that they take the best decision possible in a given situation. Instead, they usually choose approximations of the optimal solution that can be processed faster and therefore have a lower impact in the game performance and the frame rate perceived by the player.

One of the video game genres that makes a noteworthy usage of multi-agent systems to model the behaviour of intelligent enemy characters is the stealth genre. Most of the time in this type of games, the player's safest approach to win is to strive to be silent and remain undetected by the enemies. A player is usually considered to be detected when enemies know the player character's location at that time; if the information about the location stops being updated, the enemies lose track of the player, that again becomes undetected. In stealth games, the hostile NPCs are usually able to defeat the player in a much faster and easier way than in games where a direct confrontation with enemies is a more common or even expected player approach, which is the case of action games, for example. For this very reason, a stealthy behaviour is crucial for the player to become victorious. Nevertheless, players that opt for a reckless approach may have a chance to win in some stealth games, but they are likely to be less rewarded than a player that struggles to remain undetected.

Since the enemy detection of the player is a fundamental aspect of the gameplay of a stealth game, it is common to see two completely different enemy states, with different patterns of behaviour, depending on whether the player has been detected by agents or not. In the undetected-player state, enemy NPCs do not have any relevant information about the player, so they do tasks that are not in direct relation with the player's current actions, such as roaming or watching over an area. In other words, in this state the agents' actions are not a consequence of the player's actions, nor does the player influence the agents' decisions in real time. In the detected-player case, the enemies have some information about the player's present status, e.g. the current location of the character, so they can use that information to decide how to actively try to defeat the player. It is common to have a third type of situation that can be considered the conceptual midpoint between the other two: the suspicion state. In this case, the enemy agents are aware of clues or signs about the player's activities, because of a change in the environment they have perceived. They adopt an enquiring attitude to try to detect the player character. Example situations for this state include contexts where the agents have just lost track of the player but still have information about the location where the player character was recently spotted, so they can try to predict the current location. It is not difficult to distinguish this example situation from one of the active state, in which the enemies know the exact, current location of the player, so they can focus on configuring and executing a defeat plan.

To define the way how NPCs choose the tasks to do in each situation they face, it is common to use behaviour trees. A behaviour tree is a mathematical model designed to control the decision flow of an intelligent agent. The system is composed of nodes that are vertically connected: a node that appears below another one to which it is connected is a child node of the one above, that is its parent node. Conventionally, behaviour trees are read vertically from the top node

(also called root and represented in diagrams with the "Ø" sign) to bottom levels (nodes with no children are called leaves) and from left to right for nodes with the same parent.

Behaviour tree nodes are containers of agent logic and they are executed following the previously mentioned order in each update of the game (this can happen multiple times per second), although it is common to see implementations of behaviour trees that store the last-seen node to start executing from there in the next update, which is an important performance improvement, especially for large trees. This is required because some nodes take more than one update to complete execution. For this reason, nodes have a status that can be either running, success or failure. In the first case, the execution has not finished, so the tree keeps waiting, while in the other cases the work is complete, and the parent nodes are notified of the child's result. If the child node is a leaf node, its work is an agent task (e.g. physically moving to a target location), so the result is whether the task was successfully completed or not. Nodes that are not leaves can be either composite nodes or decorator nodes. Composite nodes execute their children one by one and may decide to terminate before all of them have run using certain logic, and considering the children result (success or failure) to produce their own output. One of the most common composite nodes is the sequence node (visually represented in diagrams with an arrow: "→"), that executes all its children until one produces a failure result (in which case the sequence node fails as well) or all the children produce success (then the sequence node notifies a success response to its parent as well). Another noteworthy composite node is the selector node (commonly represented by the question mark sign: "?"), which produces a successful response as soon as one of its children is successful, and only ends with a failure outcome if all the children fail. With respect to decorator nodes, they can only have one child and they often use a function that takes the child as a parameter to produce the response[2]. A notable example of a decorator node is the inverter node, that returns success if its child has a failure output, and vice versa.

Figure 1.1 shows a behaviour tree that depicts the three-state planning model that was mentioned when explaining how enemy NPCs behave in many stealth games, in a simplified way. As it can be seen, the root node has a selector node as its child, which can lead to the three conceptual states or situations that are commonplace in stealth games: trying to defeat a detected player, investigating signs of presumed player actions to try to detect the player, and doing tasks that are not in direct relation with the player in that moment, since there is no information about the player's actions. First, there is a sequence that leads to a child node whose task is to check whether the player is currently detected by the enemy. This node would produce a success response if the enemy had enough information about the player to have a chance to defeat them[3], in which case the sequence would continue and execute the defeating attempt task. If the previous task produced a failure result, meaning that the player would remain undetected, the sequence would stop before attempting to defeat the player, since the

[2] In less usual cases, decorators check external conditions and, depending on their result, they decide whether to run their child node or produce a fixed result. Some decorators set a maximum execution time before aborting their child's execution.

[3] It should be noted that this document refers to the player as *they* (using what is known as singular they), because the gender of the player, who is an undetermined person, is unknown. This practice is extended to derivate forms, such as *their*, *them* or *themselves*. Other sections of this document mention the player character and the enemy characters of the developed game, who have an anthropomorphic appearance and human-like behaviour. They are not merely objects, so the *it* pronoun is not appropriate for singular characters, but they are inorganic beings and it is not clear whether they have a gender, so each individual character is also referred to as *they*.

information about the player would be insufficient to do that, and it would notify the selector that it had failed. The defeat-player task would keep running until the player was defeated or until the enemy lost track of the player character, in which case the sequence would also fail. Only in those cases in which the first sequence failed would the selector node execute the second one. The second sequence has another question-based task to determine whether there are signs in the environment that may help to detect the player. The investigation task is executed by the sequence only in those cases in which the answer to the previous question is yes, i.e. it produces a success output. The investigation task finishes with success when the player is detected, and with failure if the presumed evidence of the player's actions becomes obsolete. If this second sequence ended with success, meaning that the player would have been detected, the execution would return success to the selector; the selector node would also produce the same output, since one of its children would have been successful: this would lead to the behaviour tree to restart, beginning execution from the root, and eventually executing the defeating task since the player detection condition under the first sequence would be true. If the second sequence produced a failure output, the last task in the behaviour tree would be executed. This task can be very different depending on the game, but it is usual to see the agent behave as an entity that ignores relevant information about the player. This does not mean that the agent cannot be trying to detect the player in this task. Not only is this a valid possibility but even a common approach: it should not be forgotten that the agents' main goal is to defeat the player, so it is reasonable to see them adapt their plan to achieve this in the way they consider more constructive in every different moment. What makes this task different from the one that involved investigating signs of player activity is the initial lack of specific, recent information that is presumed to be related to the player and that might help to detect them. This task ends with success as soon as the enemy detects the player, or when the agent acquires information that might be relevant for detection. Both scenarios lead to restart the tree and eventually execute the defeating task or the investigation-of-player-signs task, respectively. In the meantime, the tree's leftmost task keeps executing. It can be assumed that in the case of multi-agent systems, communication between agents can take place at any moment of the tree's execution and influence the result of executing tasks, e.g. an enemy might be told where the player character is while executing the leftmost task of the tree, which would lead the agent to consider the player as detected and restart the tree to eventually execute the investigation task to go to the mentioned location and restart the tree again after spotting the character to execute the defeat task to attack them.

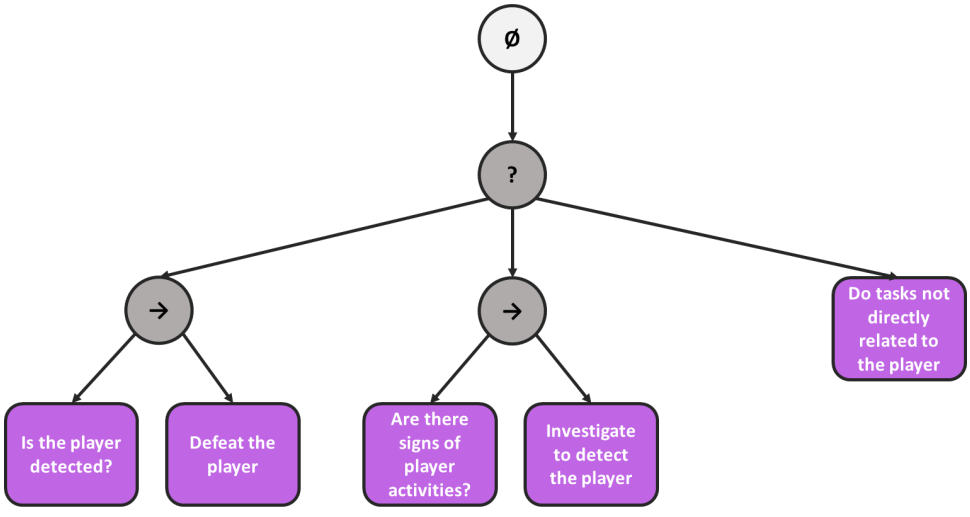


Figure 1.1: Simplified behaviour tree for an enemy NPC in a stealth game.

1.2. Objectives and motivation

The goal of this project was to develop a stealth game with enemy characters that were able to act in ways that a human would consider intelligent, with the objective of defeating the player. The game would be eventually called *Stealth Temple*. The player would have to enter hostile areas in a temple to recover orbs, valuable objects that would lead the player to victory after placing all of them in a strategical location, the temple's central monolith. To make the player opt for a stealthy approach in the game (i.e. being silent and careful in every move, instead of behaving recklessly so that enemies would easily spot the player character), the enemies would be numerous and much more powerful than the unaccompanied player, in such way that it would be very difficult to win the game without struggling to be undetected. The efficacy of the enemy actions to stop the player would be achieved by modelling their decisions and action planning using behaviour trees. The agents' behaviour trees would consider the individual characters as part of a multi-agent system, so that every individual enemy would be able to communicate and cooperate with other agents. The behaviour tree would also model the capacity of the agents to perceive the environment (with sight and hearing) and the ability to modify it with their actions, being able to move and attack the player, since defeating them would be the agents' highest priority. For the sake of simplicity, the game's target platform was Windows PC.

The motivation of the developer of this project was to learn how to use behaviour trees, a common and flexible technique that can improve the process of implementing non-player characters with realistic behaviour. This project would also allow the developer to gain practical experience in the fields of game development and game-oriented artificial intelligence, and this document might be helpful for readers that plan to develop similar projects and those who are willing to learn about behaviour trees, the implementation of reasoning for non-player characters and game development in general.

1.3. Technology, methodology and planning

Which technology would be used in the project was one of the first questions that required an answer. It was a priority to use a game engine with a built-in behaviour tree system. This would not automatically create the behaviour trees that the developer would need for a specific context (which is the programmer's task), but it would offer an architecture that would eliminate the need to implement core, generic functionality of behaviour trees from scratch. The system would allow to focus on the conceptual, high-level aspects of developing efficient and effective behaviour trees rather than those related to implementing them from a low-level starting point. In other words, the developer would still need to define and program the tasks for a specific behaviour tree, and selecting when to use different types of decorator and composite nodes would be a manual decision too, but some functionality common in all behaviour trees would be handled by the engine (e.g. the tree's traversal logic or the ability to start executing in the previous update's last running node).

Unreal Engine 4 is an extensively-used game engine with a built-in behaviour tree system that would fit the project's needs, and one with which the project's developer had some previous

experience, so it was chosen for the development of the planned stealth game. The game would be programmed using a combination of C++ code and Blueprints, the Unreal Engine's visual scripting system. C++ would allow to define base classes of objects and entities that would represent the characters, the game rules and other core gameplay structures, with the help of the Unreal Engine's C++ source code: a very comprehensive set of modules, data structures, functions and classes, that Unreal-based projects take as a starting point and extend instead of using just raw C++. The game's code would also extend the Unreal Engine's C++ template for games with a third-person camera perspective to obtain a basic implementation of camera rotation and input-driven movement for the player character, among other features, that would be adapted to the needs of the project. Blueprints would allow to extend the base C++ classes with more specific functionality, with the fast prototyping advantages of a visual scripting system. The engine's behaviour trees' editor is integrated with the Blueprints system, so the agents' tasks for the tree nodes would be also created using Blueprints. A very simple example of a Blueprint (a visual script made with the Blueprints system) can be seen in Figure 1.2, showing a task for the behaviour tree that changes the walk speed of an agent.

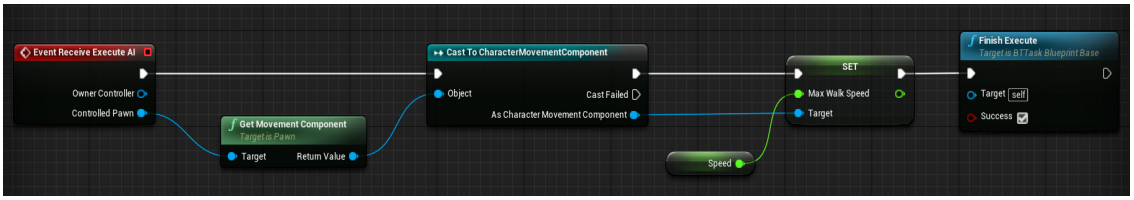


Figure 1.2: Simple task Blueprint used to change an agent's walk speed in the behaviour tree.

Additional software would be used to design some of the game's visual assets: Sculptris, 3ds Max, Fuse and Photoshop. The developer had already obtained these programs along with the required licenses in the past, i.e. no software would be acquired for the sole purpose of making this project. Some visual and audio assets would be obtained from free content samples found in Unreal Engine's digital marketplace, and no paid content would be used. In any case, there would be no monetary investments to develop the game. This was planned to be a research project on game-oriented artificial intelligence made by a single person; eventually distributing the game as a commercial product was an unlikely scenario. Hence, no budget estimation was made.

As already stated, one of the project's focus was the implementation of intelligent agents that would be a fundamental aspect of the gameplay of the stealth game; this was considered the most difficult task as well. Because of the importance of the enemy intelligence and the will to minimize the risks derived of its expected complexity, it seemed reasonable to divide the project development in modules closely linked to enemy capabilities, such as visual perception, auditory perception, communication and advanced reasoning. Every new module would be integrated into the system to coexist and coordinate with the previously-developed ones, and it would include not only enemy-related work (although this would be the main task for most modules), but also the implementation of gameplay functionality linked to the player and the game rules, among other aspects, such as the user interface. A final module would be added to the planning to refine the gameplay and conveniently integrate it with the developed enemy agents, so that the whole set of developed features would be perceived as a unity, as a stealth game.

To demonstrate that the scheduled goals for each module were achieved, 9 videos were made during the development process. Most of them include dynamic texts that explain or describe

noteworthy features as they are shown. Those videos can be found in the Appendix of this document, and Table 1.1 shows the association of the developed modules with the videos that demonstrate how the implemented features work. The table also shows how much weeks were initially expected to be required to develop each module's tasks as well as the actual time that it took to do so. As it can be seen, all the tasks were implemented, and all of them were complete in less or equal time than expected at first. The shortened time to develop some modules, namely the lasts ones, was possible because the work done in previous modules had set the stage for the coming features, i.e. the first ones were developed with the forthcoming ones in mind to facilitate integration. A noteworthy case is that of the enemies' behaviour tree, which was built for the first half of modules and required little modification afterwards. Only some nodes' tasks were affected, and they had already been structured in a modular way to be easy to modify and extend. For example, once the enemies' visual detection of the player character had been implemented in the first module, it was easy to adapt it to make the agents capable of detecting other world objects with sight, such as player-thrown stones. Moreover, some of the last module's tasks required very little work, e.g. the user interface was in an advanced stage of development since the first modules, when a health bar and buttons related to winning and losing were added to the screen, so only some adjustments were required by the end of the development process. This allowed to do an extra task that was not initially planned, that consisted in the production of a cinematic introductory video for the game, to be played before the main menu screen to narrate the game story to the player in an entertaining manner.

Development module's index	Development module's implementation goals	Planned time (weeks)	Actual time (weeks)	Demonstration videos
1	• Initial configuration of the project. • Enemies' visual detection of the player character and chasing.	1	1	Video 5.1
2	• Enemies' multi-agent communication. • Enemies' close combat logic.	1	1	Video 5.2
3	• Enemies' shooting logic. • Player character's ability to knock enemies out. • Win-lose logic.	3	3	Video 5.3 Video 5.4
4	• Enemies' auditory detection of the player character. • Sound system. • Player character's ability to crouch (to be able to move without making noise).	2	2	Video 5.5
5	• Player character's context-specific abilities (grab, drag and drop bodies of unconscious agents, and grab, drop and throw objects, such as stones). • Enemies' advanced investigation and reasoning of signs of player activities (bodies of agents on the ground and thrown objects).	2	2	Video 5.6
6	• Multiple objectives for enemies and different agent roles (patrolling and guarding a location). • Complex cooperation between enemies.	2	1	Video 5.7
7	• Menus and user interface. • Final level for the game. • Adaptation of the enemies to the final level. • Final player objectives and game rules. • Final appearance for the game elements. • Cinematic introductory video (not initially planned, added later). • Game packaging.	4	2	Video 5.8 Video 5.9

Table 1.1: Table that outlines the main tasks of each planned module, along with the planned and actual development time in weeks, as well as links to demonstration videos.

1.4. Introduction to the developed game

This section is aimed to present the developed game, i.e. the final product, before explaining the implementation of the most important features in the Development chapter. This section's global vision of the product may help the reader better understand the development process in the mentioned chapter.

The name of the game is *Stealth Temple*. As its name suggests, the game emphasizes the use of stealth and takes place in a temple, which is divided in four areas. Each area is associated with one of the four elements established by the Greek philosopher Empedocles: air, earth, water and fire. These zones are shown in Figure 1.3. The temple zones are connected by a central monument, the *Monolith of Truth*. The story of the game is narrated in the cinematic introductory video (Video 5.8) that plays when the game is started. It explains that the temple, which was once a peaceful sanctuary for meditation, was assaulted by a group of criminals (*The Rolling Beards*) that fought the temple's guard robots and reprogrammed them to become killing machines instead of protectors of peace. Lost in a state of radical violence, the manipulated robots eventually turned against the criminals and threw them out of the temple. The guards would attack anyone they saw, so nobody would be able to visit to the temple until the possessed robots' destructive acts were ceased. The player takes the role of the temple's guide, who had hidden during the siege and became trapped in the building. With no professional fighting skills nor external help, this character is the only one who can restore the temple's peace. To do so, the player needs to guide the character to the four zones of the temple, find sacred orbs and bring them back to the *Monolith of Truth*, where they should be placed. If the four orbs were gathered together in the monolith, the temple's state would be reset, and the guards would recover their consciousness as well as their vocation for peace.

Figure 1.3: Stealth Temple's zones, linked with Empedocles' four elements: air (top left image), earth (top right image), water (bottom left image) and fire (bottom right image).

As already stated, the cinematic introductory video is played when the game starts, but it can be skipped by pressing any key. The player is then brought to the main menu of the game (shown in Figure 1.4), which is composed of different buttons that can be clicked. Firstly, one can see the *Play* button, which leads to the playable world of the game. Secondly, there is the *Controls* button, which lets the player see which game actions can be done with different types of mouse and keyboard input, as well as their gamepad equivalence, since the game is compatible with these devices, as shown in Figure 1.5. Thirdly, one can see the *Settings* button which shows different adjustable options of the game, as one can see in Figure 1.6. The player can change the game resolution by choosing one of the listed ones. The sensitiveness of the camera and the music volume can be modified with sliders or reset to their default values. The last option allows to delete the saved game progress, which requires to click a confirmation button to prevent accidental deletions. Fourthly, one can click the *About* button to see the name of the developer and some information about the game engine, Unreal Engine 4. Lastly, the *Quit* button can be used to leave the game.

Figure 1.4: Stealth Temple's main menu.

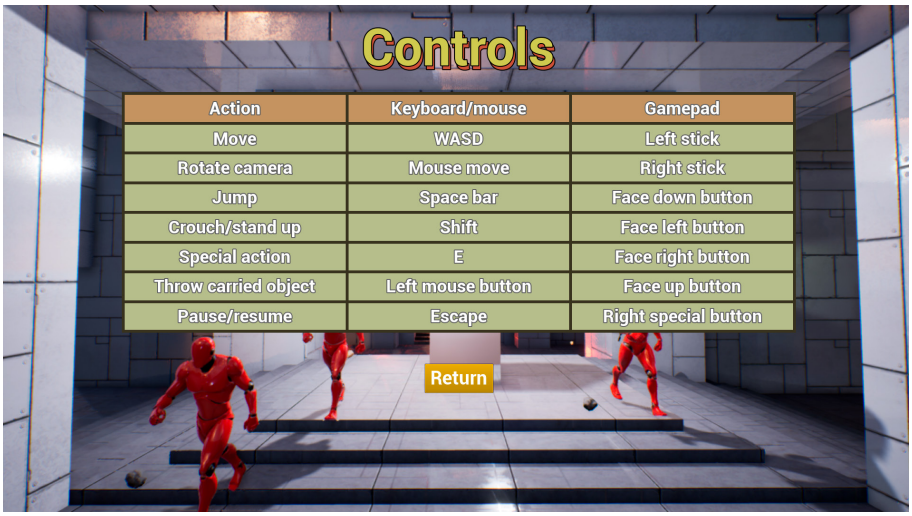


Figure 1.5: Stealth Temple's controls.

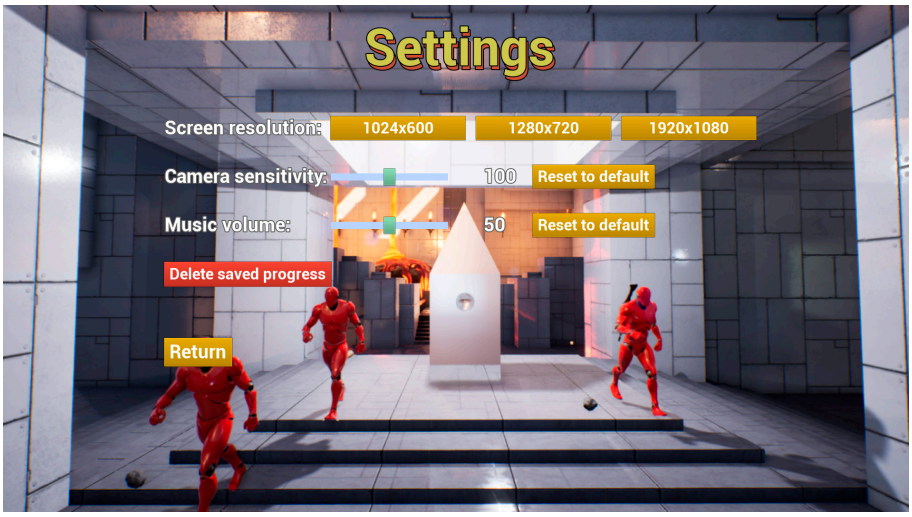


Figure 1.6: Stealth Temple's settings.

After pressing the main menu's *Play* button, the game world is loaded, and the player can start controlling the main character, using a third-person perspective camera that can be rotated to see what happens in the surroundings. The action starts in front of the *Monolith of Truth*, which initially has no orbs. The player can freely decide to go to any of the four zones of the temple to seek the valuable objects and bring them back to the monolith, which is necessary to win the game. Should the player be defeated, the failure menu will appear (as seen in Figure 1.7), which allows to choose between restarting the game or going back to the main menu. If the player chose the first option, the game would restart in front of the *Monolith of Truth*, but any progress would be saved, i.e. the already-placed orbs would remain in the monument. The game can also be paused at any moment, showing the pause menu, which would be undistinguishable from the failure menu if it did not offer the additional option to resume the paused game.

Figure 1.7: Failure menu shown when the player is defeated; in this case, the player character has been neutralized in front of the Monolith of Truth (an orb can be placed in the spherical space that can be seen in the monolith).

The player character can crouch (as shown in Figure 1.8) to be able to move without making noise, at the expense of taking more time to reach locations than if running. The player character

can also jump. When the enemy guards see the player character or hear footsteps or jump noises, they communicate this information to all other agents, and proceed to attempt to defeat the player. Some enemies have no weapon, so they attack the player character with their fists in close combat. Unlike them, other agents carry assault rifles, which they use to shoot the player character in the head when they think they have a reasonable chance of impact. To improve accuracy, some enemies considerably reduce their speed when shooting to be able to aim with higher precision, with the weapon at the height of their eyes, instead of having the arms lowered down to shoot while jogging. Meanwhile, and to avoid losing track of the player, other agents decide to focus on chasing the character. Because of the lethal combination of enemy roles and the high number of agents (shown in Figure 1.9), it is generally more reasonable for the player to try to be silent and remain undetected while trying to find the orbs and bring them back to the monolith.

Figure 1.8: Player character crouched close to the fire orb.

Figure 1.9: Player character running while besieged by enemies (notice how three of the orbs have already been placed in the Monolith of Truth).

Besides running, jumping and crouching, the player character has other abilities that can help the player find the orbs and place them in the *Monolith of Truth*. The player character can

neutralize enemies with a knockout move by stealthily approaching them without being detected, i.e. from behind (as shown in Figure 1.10). This silent move makes the victims collapse onto the ground. An unconscious agent's body can be dragged by the player character. It is useful to move bodies to hidden locations (as shown in Figure 1.11), since enemies use to investigate the surroundings of bodies as soon as they spot them, which may lead to player detection. Enemies also inquire into sounds other than footsteps, such as the impact sound of objects being thrown, namely stones. The player can grab stones and throw them when desired. Should agents hear the noise, they will investigate thinking that they may locate the player character nearby. Hence, throwing stones can be used as a distraction to make enemies go to certain places and stop patrolling near orbs, for example. As shown in Figure 1.12 and Figure 1.13, stones can also be used to make enemies become victims of traps. Nevertheless, stones should be only thrown while hidden, since agents that see a stone moving in the air will try to find its origin, which may lead to reveal the position of the player character. Moreover, throwing stones is not a guarantee that all agents close to an orb will leave the scene to go to the noise epicentre: some agents guard specific areas of the temple instead of patrolling the totality of the zones, which makes the game more challenging. Orbs can be grabbed in the same way as stones, and even thrown (although they are not as suitable as stones for that purpose). Unlike stones, though, orbs can be placed in the *Monolith of Truth*, as shown in Figure 1.14.

Figure 1.10: Player character in the brink of knocking an enemy out.

Figure 1.11: Player character hiding an enemy's body.

Figure 1.12: Player character in the verge of throwing a stone under a big hammer that falls when someone walks underneath.

Figure 1.13: Two enemies were investigating the zone where a stone was thrown because they had heard the impact sound; one of them has been knocked out by a fallen hammer.

Figure 1.14: Player character in the edge of placing the fire orb in the Monolith of Truth.

The guards' hostilities cease when the player places all the orbs in the *Monolith of Truth*, moment in which the player becomes victorious, as seen in Figure 1.15. After that, the temple can be safely visited once again, since the agents stop being aggressive and recover their role of temple protectors, as shown in Figure 1.16. If the player wants to repeat an orb's mission after placing the object in the monolith, all it is required to do is to restart the game, which can be done by simply pausing the game and clicking the restart option: the old orb will be still in the monolith, but a copy of it will be in its original location in one of the temple's zones. The player may also consider deleting the progress of the game in the main menu's settings to start a whole new game.

Figure 1.15: Player character after placing all the orbs in the Monolith of Truth; the player has won the game.

Figure 1.16: After saving the temple, the player can take as much time as desired to visit the temple's areas; the guards have become pacific.

2. Development

2.1. Implementation of the core gameplay structure

2.1.1. Definition of the gameplay framework

The game is composed of multiple entities that are created with the game world. Each of these game elements, which has a specific purpose, can interact with other entities to make the game function and give a certain audio-visual presentation to the player. Some of these entities have a purely aesthetic role, meaning that they could be replaced by others without changing the core game flow, but only modifying the player's perception of some elements. That is not the case of the gameplay framework, which is defined by the essential entities that are necessary for the game to be played as it is, so they are what define a game's genre and gameplay pillars. The implemented gameplay framework extends the base gameplay classes defined in the Unreal Engine, that game developers can take as a starting point and adapt to their needs. The classes of entities that make up *Stealth Temple*'s gameplay framework are explained in the next paragraphs. Despite of this fact, it should be noted that some names were changed to be easier to understand. For example, what is called *Throwable Object* in this section was named *Throwable Actor* in the actual game structure, to follow Unreal Engine's convention to call actors those entities with physical presence in some location of the world.

One of the main classes is the game mode, since it defines the rules of the game and controls the game state. In *Stealth Temple*, the game mode sets that the player character must place all the orbs in the *Monolith of Truth* to win the game (in which case the game mode sets the game state to won), while the game is lost (and the game state is set to lost) if the character's health reaches zero. In *Stealth Temple*, the game mode class is called *Stealth Game Mode*, and it is activated when the player choses to play in the main menu, which is defined in the *Main Menu* class.

Another essential class in the game is the character class, since it defines the functionality that is common for all game characters, such as the possibility to run or to be damaged (implementing a *Health Interface*). However, none of the characters that can be seen in the game is just a character, but an enemy character or the player character, which are two additional classes that extend the generic character template (the *Base Character* abstract class). The player character class, called *Player Character*, defines the actions that can only be made by the player character, such as grabbing and throwing stones, attacking with a knockout move or grabbing and carrying bodies of unconscious enemies. Another feature that is only available for the player character is the camera movement, which requires the character to have a camera component attached. The class also configures how the player's input affects the character, i.e. how the character should move when input associated with movement is pressed. Even though the character is controlled by the player, a player controller class (called *Stealth Player Controller*) is used as a proxy of the player for some operations, such as creating the Heads Up Display (HUD) that allows the player to see how much health is left and how many enemies have been knocked out. The HUD class, which is also told to display pause and failure menus when required (using an *In-Game Menu* class), is called *Stealth HUD*.

The *Enemy Character* class is responsible for defining the actions that only enemies can do, which include close combat attacks and the possibility to carry and use weapons, whose attack behaviour is defined in the *Weapon* abstract class and in the subclass that is used by enemies, *Assault Rifle*. The *Enemy Controller* class can be seen as the brain of an enemy character, since it runs the behaviour tree that models how enemies plan and execute their actions (*Enemy Behaviour Tree*). The behaviour tree uses a specific set of task and service classes, but they are explained in section 2.2.2. Thanks to the use of the behaviour tree, enemies can perceive the environment with sight and hearing, and they can communicate with other agents. While each enemy character is an individual entity executing its own copy of the behaviour tree, there is an entity that stores part of the information that enemies have shared, which is the defined by the *Enemy Shared Knowledge* class.

The *Throwable Object* abstract class defines the common physics-based behaviour for objects that can be grabbed, dropped and thrown by a character. These objects make noises when they impact onto other world elements, that the player and enemy characters can hear. The *Stone* is the most common type of throwable object, but the *Orb* is different in that it can be placed in the *Monolith of Truth*, defined by the *Monolith* class, that notifies the game mode of the placement of the orb so that the progress is saved to disk (using an instance of the *Stealth Save Game* class), and the player wins the game if all orbs are placed in the monolith.

Figure 2.1 depicts the class diagram of *Stealth Temple*'s gameplay framework classes, so it does not include all the game's classes, but the most important ones, which have already been explained. The diagram shows a simplified version of the flow of the game (supressing or altering some relationships between entities), and some of the most relevant attributes and methods of the classes. The names of some classes have been changed to facilitate understanding. For simplicity, the diagram does not show the Unreal API (Application Programming Interface) classes from which each *Stealth Temple*'s class extends, but it should be noted that all of them have a parent class in the API that makes basic functionality possible, e.g. the *Stealth Game Mode*'s parent class is the *Game Mode Base* class from the Unreal API, and the game's *Base Character* class extends the API's *Character* class. Only the most relevant methods are shown, ignoring most of those that set or retrieve a single attribute, and their names are adapted to give a clearer idea of the game's flow. The visibility of some class members and the names of some variable types have also been changed and simplified, e.g. integer values are represented as "int" in the diagram while they are "int32" in the engine, a type of integer that is always made up of 32 bits (it is not influenced by the specific hardware where the game is executed). The class diagram does not include the components of the classes (such as the player character's camera component or the meshes that the define the appearance of some game entities), nor the user interface widgets used by the HUD and menu classes, since they are not as relevant as the classes of the gameplay framework.

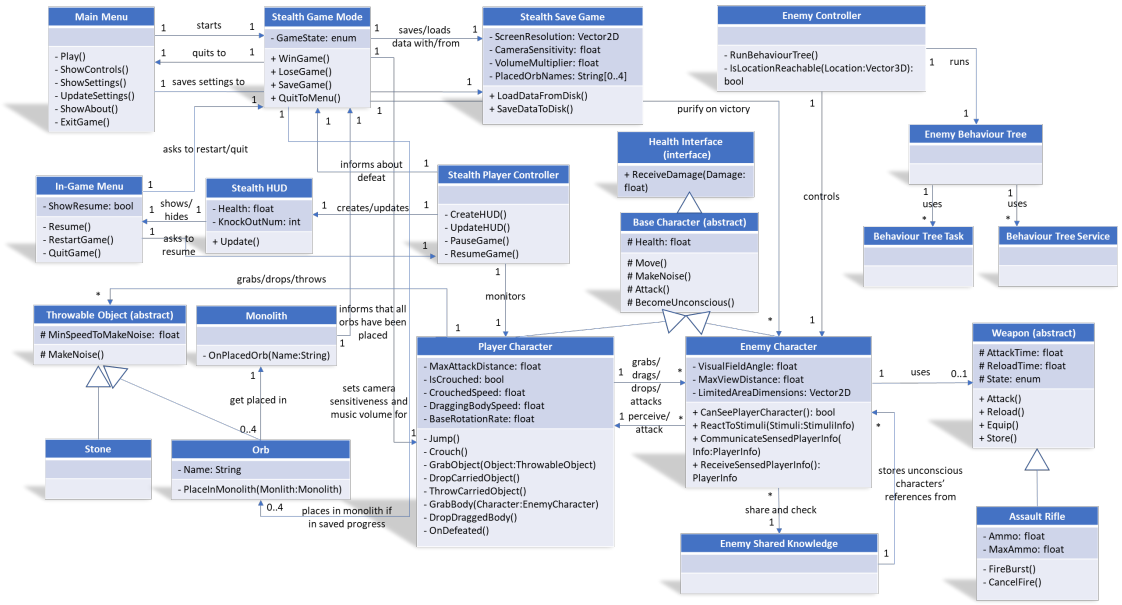


Figure 2.1: Stealth Temple's simplified class diagram.

2.1.2. Implementation of the characters' abilities and interactions with the world

To be able to move in the world and interact with other game elements, characters use a set of components, which are the parts that make characters a functional whole. A capsule component is the invisible root element of the character, and it defines the base collision with the world. Its diameter and height values match the desired character dimensions. A mesh component attached to the root capsule gives characters a physical appearance and the ability to play different animations. A special type of Blueprint, an Animation Blueprint, is responsible for toggling animations when the characters' circumstances change (e.g. to switch from an idle animation to a jumping one when the character starts jumping). Figure 2.2 shows the state machine of the Animation Blueprint used by all characters, although some behaviour is specific for the player character (e.g. the knockout animation) and other features are enemy-exclusive (such as weapon carrying, which is handled inside the idle or running state). Many conditions are tested to determine the exact animation that is played on different bones of the character's mesh in each state of the state machine. For example, in the idle or running state, the legs are influenced by the current speed of the character: an idle animation should be played when speed is zero, while a running animation is the right choice when speed is high; the two animations are blended with different weights depending on speed. If the character has a weapon, it is checked whether the character is doing a specific action with it (i.e. aiming, reloading, storing the weapon or equipping it) to determine which animation should be played on the arms.

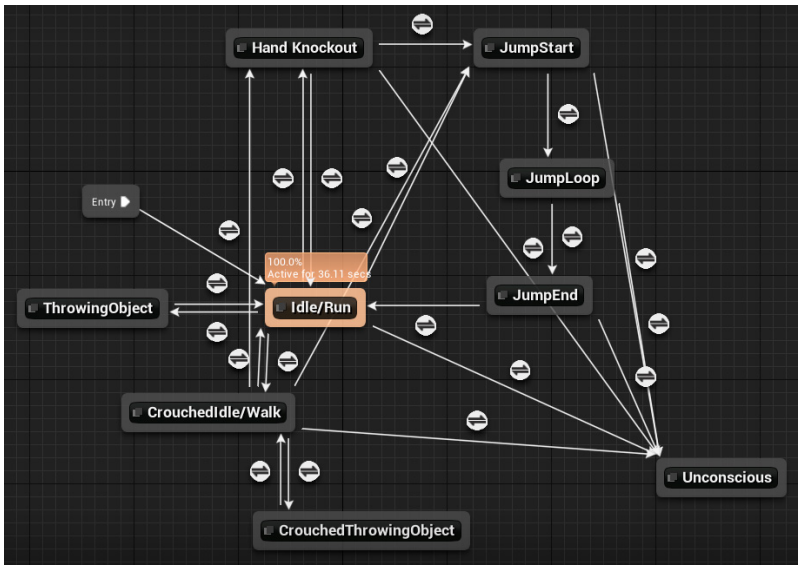


Figure 2.2: State machine of the Animation Blueprint used by characters.

Characters also have a movement component that allows them to move. Unreal Engine has a type of movement component thought for anthropomorphic characters that allows to easily configure how characters should move. Among the parameters that can be set, one can find the maximum speed or the rotation rate of the character, for example. Other values include the maximum height that single steps can have to allow the character to go upstairs without the need of jumping. The ability to jump is precisely another of the built-in features of the characters' movement component, and parameters such as the jump speed can be set. Crouching is supported as well. Despite of the built-in characters' capabilities, it is the programmer's task to handle the animation transitions to be coherent with the change in actions, checking conditions in the Animation Blueprint (e.g. the transition to the jump animation in the state machine should only take place if the character is physically jumping). The player character's physical movement is determined by the player's input, e.g. the character physically crouches when the crouch key or button is pressed (if the character is not already crouched or in a situation where crouching is not possible). The enemies are guided by their controllers and behaviour trees.

In every update of the game, the player character checks if a special action is available for the player: if that happens, this is communicated to the HUD so that a message is shown in the screen, to let the user know that a context-specific action can be done. Special actions include the possibility to grab, drop or throw a throwable object (e.g. a stone), or to place an orb in the *Monolith of Truth*. Knocking out a close enemy is also one of the possible actions, as well as grabbing or dropping the body of an unconscious enemy. For simplicity, only one of these actions is possible at a time, or two if carrying a throwable object, since it can be either thrown or dropped onto the floor. What all these actions have in common is the use of invisible volumes to determine whether the action is possible and how it should be put into practice. These volumes are collision components whose geometry is a simple primitive, such as a sphere or a capsule. They are created in the locations where they are required to test collision with game elements. The volumes do not block other entities or affect movement, they just check overlapping conditions. Some of the volumes are immediately destroyed after the check, while others are kept attached to a specific entity (e.g. a throwable object) and events are configured

so that specific behaviour can be triggered when the volume collides with a certain entity, such as a character.

In the case of throwable objects, they have a spherical volume attached to their main mesh for their whole existence. This volume tests collision only when the object is not being carried by a character. If a character gets close to the object, the volume (which is larger than the object's mesh) triggers an overlap event. Since the player character is the only one that grabs objects in the game, it is required for the overlapping character to be the player character to continue with the test. If that is the case and the character is not carrying another object or dragging a body, the volume's object is added to the character's list of grabbable objects. In the next frame's update, the HUD shows that it is possible to grab an object with a key or button (as shown in Figure 2.3). The action of grabbing the object takes place if the player presses the grab key or button before getting away from the object. If the player character exits the volume, an event is triggered to remove the object from the character's list of grabbable objects. Once the player character is carrying a throwable object, it is possible to throw it or drop it at any time with different input options (as shown in Figure 2.4). Dropping is useful to quietly leave the object in the ground without making noise, while throwing uses the player camera's forward vector to determine the projectile's direction, before throwing it at high speed. A crosshairs icon is shown on the screen to help the player aim.

Figure 2.3: Player character next to a stone that can be grabbed.

Figure 2.4: Player character after grabbing a stone.

When an enemy becomes unconscious, a big capsule-shaped volume around the body is enabled to trigger an event when the player character intersects with it. If the character is not carrying anything, the body is added to a list of draggable characters (which is the case in Figure 2.6), from which it is removed if the player character moves away from the volume. The character can start dragging an unconscious enemy that is in the list, which decreases the player character speed and rotation rate, and activates a continuous collision check between the body and the world, to simulate physical forces when hitting something. The dragged character's mesh is set to play an animation (made in 3ds Max, a 3D modelling package, as shown in Figure 2.5) and it is attached to the player character in such way that it seems that the legs of the dragged character are pulled or pushed by the hands of the player character, as seen in Figure 2.7. A body can be dropped when desired, and this happens automatically if the player character falls from a platform or gets damage. When an unconscious agent's body is not grabbed by the player character, it simulates physics on its bones to act as a ragdoll.

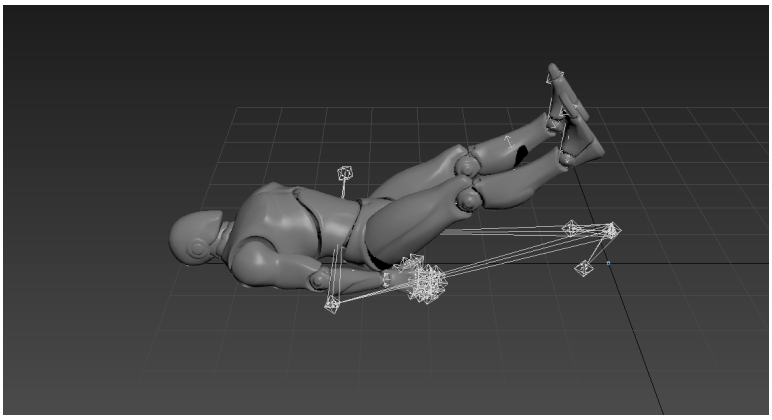


Figure 2.5: Animation played by an unconscious enemy character when dragged by the player character, as seen in 3ds Max.

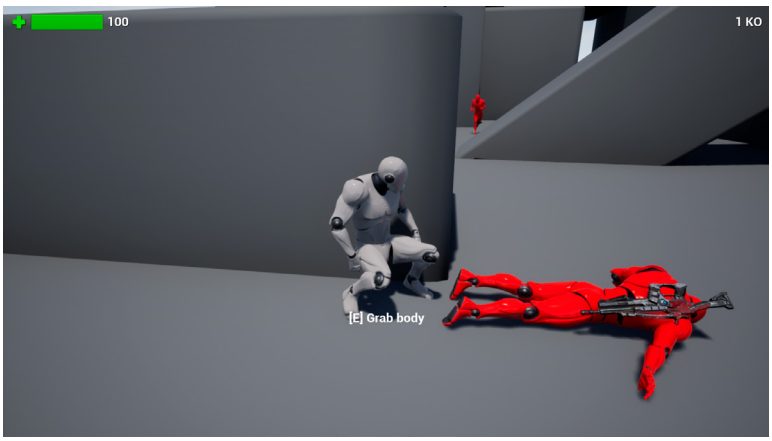


Figure 2.6: Player character next to an unconscious enemy, whose body can be grabbed.

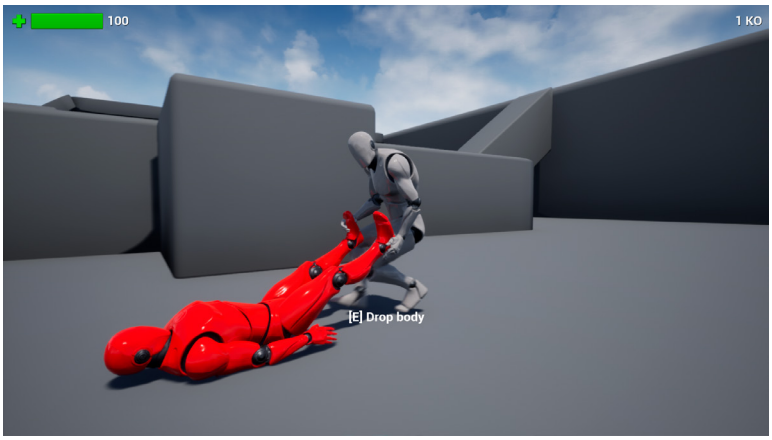


Figure 2.7: Player character dragging the body of an unconscious enemy.

To be able to notify the player of the fact that an enemy can be attacked, the player character checks whether a close, conscious enemy would be successfully hit (and knocked out) if the player attacked them in that moment. To do this, the forward vectors of the player character and the potential victim (an enemy character in a close location) are compared using a dot product, and only if the angle between them is small enough can the attack be possible, since the player character must attack from behind, unseen by the enemy. But since the angle can be small in the case in which the enemy character is behind the player character, this scenario needs to be discarded by ensuring that the dot product between two vectors is positive: those vectors are the player character's forward vector and the vector that goes from the location of player character to the enemy's location. If this is true, the attack can take place if the player presses the key or button shown in the screen (as seen in Figure 2.8). When the two characters are too far from each other or the mentioned conditions are not met, the action is not available.

Figure 2.8: Player character behind an enemy that can be attacked.

When the player character attacks using a knockout move, a spherical collision volume is temporarily attached to the character's hand to check for collision with an enemy. Only if the invisible sphere intersects with an enemy character is the hit processed, making the enemy receive a critical amount of damage (using the health interface they implement) that make them lose consciousness and fall onto the ground, with their bones being set to simulate physics to achieve ragdoll behaviour (as shown in Figure 2.9). The same collision test on a character's hand

is used when an enemy character is attacking the player character with close combat, i.e. punching. The enemy character's hands use volumes to try to detect a hit with the player character for the full duration of the close combat attack. If the impact takes place, the player character receives damage and is pushed in the punching direction (as seen in Figure 2.10). The implementation of the health interface of characters allows to play sounds and show particle systems (i.e. visual effects) when damage is received.

Figure 2.9: Player character using a right-hand move to knock an enemy out.

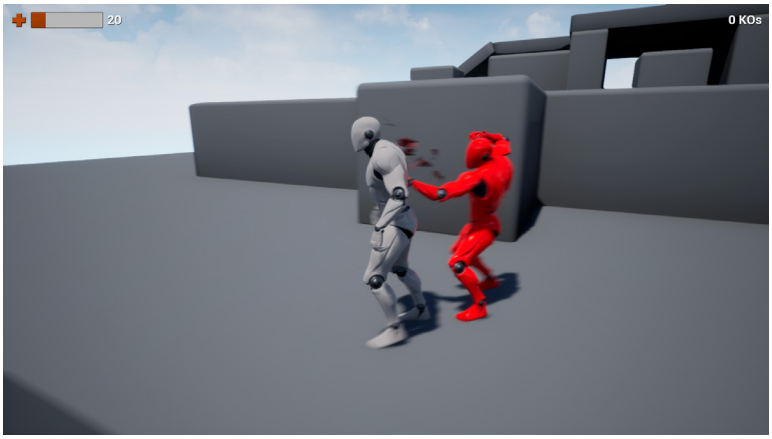


Figure 2.10: Player character hit by an enemy's close combat attack.

To make shooting possible, when a character fires a weapon (e.g. an enemy's assault rifle) an invisible line, also known as a trace line, checks for collision starting in the weapon cannon and ending far away (as far as the weapon's projectiles are set to be able to reach), in the direction of the weapon's forward vector (i.e. where the shooting character is aiming). If some world object intersects with the line, it is checked which type of entity it is to determine what should happen. If the hit element implements the health interface the entity receives damage, which is the case of the player character (as shown in Figure 2.11). In all cases, sounds and effects are used in both the weapon's cannon and the hit location.

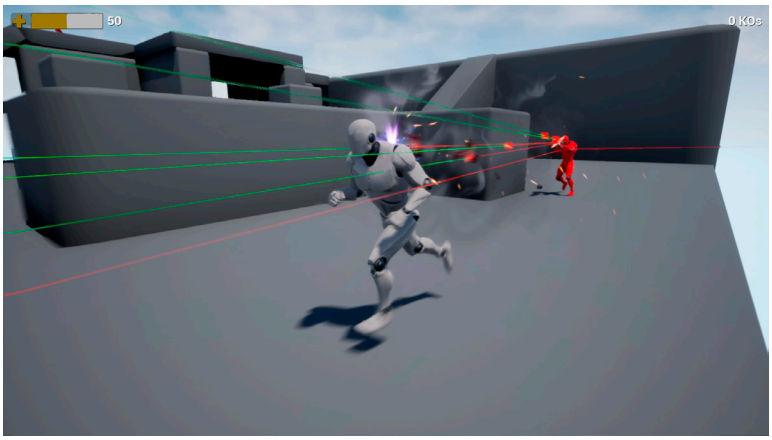


Figure 2.11: Player character hit by a shooting attack (the lines represent the collision detection traces, which are usually invisible).

2.2. Implementation of the artificial intelligence-based multi-agent system

2.2.1. Multi-agent system analysis: communication and cooperation between agents

All the enemy characters in the game form a single multi-agent system. The objective of both individual agents and the system is to defeat the player character as fast as possible. Every enemy can send information to all other agents at any time: it should be thought of as radio or telepathic communication. Enemies examine the information they have received from others and decide whether to use it and how to do so, instead of automatically obeying external orders: agents are independent, but they take in consideration what others say. In *Stealth Temple*, there are different types of information that can be communicated, and they are all related to the player character. In order of importance, as shown in Figure 2.12, they are: the location where the character has just been seen, the location where an unconscious agent's body has been seen in movement (but the player character that may be dragging or pushing the body is not seen, e.g. they are behind a wall), the approximate location from where the sender agent thinks that the character's footsteps have been heard, the estimated location from which a thrown object seen in the air might have been thrown, the location of an agent's unconscious body that had not yet been reported to have been knocked out (so the player might have recently attacked them), and the approximate location where a sound that might have been caused by some player character's action has been heard (e.g. the impact sound of a stone or another object that has been thrown or moved). The higher an information is in the relevance list, the greater the chance to be able to find the player to defeat them, so it is obvious that visually detecting the player is more helpful than the other options. The lower an information is in the list, the higher the probability that the player is not the cause of the event or that the information is obsolete. For example, the sound of an object might have been caused by an agent accidentally moving the object and making it fall from a noteworthy height, without the participation of the player character. A spotted unconscious body that is not in movement might have been knocked out a long time before being seen, so the player character may have gone to somewhere else in the meantime, meaning that the information would be obsolete.

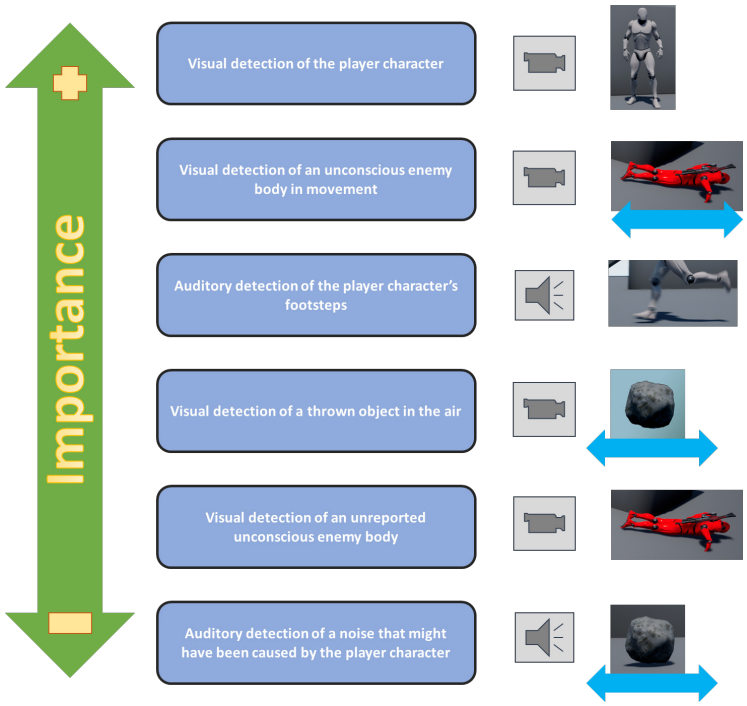


Figure 2.12: Types of information related to the player character that agents communicate to others after perceiving changes in the environment, ordered by importance.

When an enemy sees an unconscious agent, the situation is told to others, so all enemies realize that the spotted agent has been defeated by the player: from then on, they can find that agent in the list of unconscious agents in a shared knowledge structure that all agents can check. This is stored in a commonly accessible entity to avoid creating and maintaining a copy of the same list for every agent. The shared knowledge structure also contains a list with all the agents, to facilitate the process of sending information to others.

Different enemy characters might have different information in a specific moment, e.g. an agent might have visually detected the character while another one may have just heard their footsteps. When their perceptions give them new valuable information, they share it with others. The agents that receive the data only accept it if they think that it is more helpful and accurate than their local information, i.e. it has higher importance in the list of types of information that has already been explained. For example, it would make no sense for an agent that was able to see the player character to focus on locating them in the location where another agent said that a stone impact noise had been heard, since the player character's location would already be known by the agent. Other cases of communication, however, can be very useful for the rest of agents in the system. For example, when an agent sees the player character, all other enemies receive the precise detection location, which help them find and defeat the player. An agent with no information about the player will accept any information that is given to them. The agents discard the information they have when they consider that it is obsolete, which they do after some seconds without being able to use the old data to get new clues about the player's actions.

The enemies' multi-agent system is decentralized: there is no controlling coordination unit. Instead, individual enemies take their own decisions (which they do by comparing their own information with the data sent by other agents), trying to contribute to achieve the system's

goals. The communicative process is simple and concise. There is no complex discussion between groups of agents during communication: an agent sends information about the player and others receive it and decide whether to use it to take decisions and how. They do not need to reply with any kind of message (e.g. an acknowledgement response) to the sender, since the information always reaches the target receivers, who can only give useful information in return when they perceive new stimuli from the environment that might contain information about the player. When that happens, a new communicative act starts, and it ends as soon as all other agents have received the information (there is no need to wait for their decision). It can be concluded that the intelligence of the system relies on the individual reasoning of the enemies, refined with the information received from other agents.

While all agents participate in the explained communicative act (they are either the sender or the receivers), there is another type of situation where only a group of agents interact to try to defeat the player, and it should be noted that not all of them need to be carrying weapons. When multiple agents are close to the player character and they can see the character, some enemies decide to focus on chasing the character to minimize the likelihood of losing track of them, while others (only some shooters) focus on aiming with high precision to maximize the number of shots that hit the player character. The agents in the second group move more slowly, since they must hold the weapon close to their eyes to aim with precision, so they are unable to run. The agents that have no weapon always chase the player character (since they can only attack with close combat), but shooters dynamically decide whether they chase the player character as well (if so, they can shoot as they run, but they will not be able to do so with high accuracy) or whether they opt for aiming with high precision, which they can only do if there are at least a few agents chasing the player character (since they must follow the character to avoid losing track of them. To know this, the enemies can check the near agents' state (aiming with precision or not) without having to ask them and wait for a response. This combination of roles helps to dramatically increase the chance of achieving the system's goal of defeating the player. Figure 2.13 shows an example of this type of cooperation.

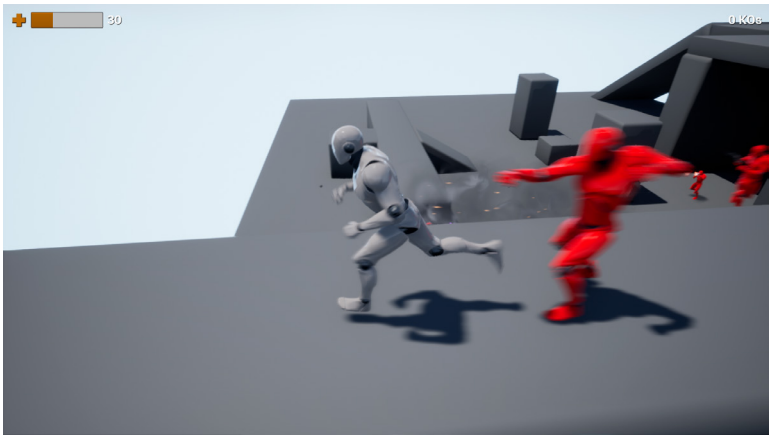


Figure 2.13: Cooperation between agents to defeat the player; an enemy with no weapon is chasing the player character, as well as a running shooter (here reloading the weapon), while a distant shooter aims with precision to hit the player character.

Aside from the described cases of dynamic collaboration between agents, in which they take specific decisions with the influence of others in a certain moment, there are other particularities of the multi-agent system that facilitate defeating the player. There are two

statically-defined types of agents: chasers and guards[4]. Their initial world position and proportion is manually set (but they are autonomous once the game starts), as well as some other details for guards. Chasers patrol the totality of the environment and pursue the player character wherever they go, while the character is detected. Unlike them, guards are prone to stay in a limited area of the environment for a long time, which is useful to protect sensible areas, such as zones with valuable objects, i.e. *Stealth Temple*'s orbs. Guards chase the player character to the edge of the limited area where they stay, but they can continue attacking in the distance (shooting) and tell other agents where the player character is, as seen in Figure 2.14. The guard's area is a square-shaped region of space whose dimensions are set individually for each guard. The combination of chasers and guards can help defeat the player, since chasers tend to form big groups when pursuing the player character, and it is crucial that other agents (guards) keep some distant areas protected just in case the player succeeds in hiding from the big group and decides to go there.

Figure 2.14: Guard protecting a corridor and attacking the player character.

2.2.2. Implementation of the agents' individual reasoning and decision-making

The planning and decision-making of individual enemies is modelled using behaviour trees. In fact, all agents execute a copy of the same tree, but depending on their custom situation and perception of the environment they take different decisions and actions. Section 1.1 introduces behaviour trees and shows a simplified example of a generic tree that would serve as base to model the behaviour of an enemy in many stealth games (Figure 1.1). The behaviour tree of *Stealth Temple*'s enemies is an expansion of that simple tree, keeping the three main situations or states described there: attempting to defeat a detected player, investigating signs of presumed player actions to try to detect the player, and doing tasks that are not directly connected with the player in that moment (since there is no information about the player's actions). However, in the simplified tree, each of the states uses a single task to do all the

[4] In the first chapter of this document (Introduction) the word *guards* was used to refer to all enemy characters, since they had been the protectors of the game's temple before they were reprogrammed and became hostile. In this chapter, however, only some agents are called *guards*: they are the ones that watch over a reduced area of the temple; those enemies that patrol the whole temple are referred to as *chasers*.

possible actions of the situation. While this is possible, dividing the work in multiple task nodes can make the tree much easier to modify when required. Using this modular approach, each of the multiple nodes represents a single action (or a very simple set of actions), not a complex combination of them. Decorator and composite nodes can be used to handle the association of different tasks in the desired way. This methodology has been used to define the final behaviour tree for the enemies.

An adapted version of the enemies' behaviour tree can be seen in Figure 2.15. Its complexity is very similar to that of the actual behaviour tree, shown in Figure 2.16, but it includes some changes that should make the tree easier to understand, while keeping an almost equivalent level of detail[5]. A noteworthy feature of the tree depicted in Figure 2.15 is represented with a green rhomboid associated with the first selector node. It is a service, an Unreal-specific type of node, although it is possible to implement it in other behaviour tree systems. A service is a node that executes some code from time to time, while the subtree under the node keeps running. It is commonly used to do periodical checks and updates of information that can influence the flow of the tree's execution. In the case of *Stealth Temple*, the adapted representation of the tree states that the service updates the agent's information with perceptions, and it may seem that they include vision, audition and the reception of communications. Nevertheless, in the implemented tree, vision is the only sense to be periodically checked by the service to see if some interesting element (i.e. the player character, a thrown object or an unconscious agent's body) can be detected by the agent. This is an optimization, since the other two types of perceptions, those related to hearing and communication, are based in interruptions, i.e. they do not need to be manually checked, since the enemy character is notified by an external entity (the source of the sound in the case of hearing, and the sender of the information in the case of communication) of the availability of new information, that is immediately checked. This can change the internal state of the character and therefore influence the execution of the tree.

The behaviour tree is divided in three different subtrees, depending on whether the enemy can see the player character, whether they can't but have some other information that can be used to try to find the player character (any sound or visual stimulus that may be related to the player and that has an estimated or exact source location), or whether no relevant information about the player is available. This is checked by three different decorator nodes. Decorator nodes are all shown in blue, but they do not all act the same way. The three mentioned ones are represented by a rhomboid shape in Figure 2.15, and they do not execute in a preestablished moment. Instead, they periodically check a conditional expression to ensure that it is true; if not, they immediately abort the execution of their subtree and return false to the decorator's parent node. In this behaviour tree, these three nodes allow to reset the execution flow when something requires changing from one of the three logical states (the one in which the player character is visible, the one in which they are not seen but there is some information that may help to find them, or the one in which no relevant information is available) to another one. This can happen after one of the updates of the service that checks perceptions, or when an

[5] One of the notable differences between the adapted representation of the behaviour tree and the actual tree is a naming convention. In the adapted version the player character is referred to as *player*, but in the real tree the character is called *enemy* (i.e. an enemy of an enemy agent, which in this case is the player character). This was done this way to make the actual implementation of the tree easier to modify if new non-player characters with objetives opposing to those of the existing agents (i.e. potential enemies of the current agents) were ever added. Since this idea may be confusing for readers at first, the adapted representation of the tree, meant to be easier to understand than the implemented one, considers that the only possible enemy of agents is the player character.

interruption related to hearing or communication is received. For example, if an agent has no information about the player, the tree will be executing the idle subtree (e.g. patrolling in some zone), but if the player character is suddenly seen by the agent (i.e. detected by the periodically-executing service node) the decorator that requires to be idle with no information about the player stops being true, so the flow of execution returns to the parent selector, which leads to eventually entering the subtree that has the condition of being able to see the player character, that leads to attack them.

Unlike the ones with rhomboids, the decorator nodes represented by rounded rectangles in Figure 2.15 only execute once, but they have different criteria as for when to run. The ones whose name that contains a question mark (e.g. the one that checks if the agent is a guard, not a chaser) perform a conditional test when the tree execution reaches them, and only allow their subtree to run if the condition is true, otherwise they return false to the parent. The decorator nodes that mention time allow their children to execute, but they will be aborted after some time if they have not finished. The rest of nodes represented with blue rectangles always execute their child subtree and apply a function to its result, e.g. the force success decorator allows to return true even if the child task or subtree returns false.

Unreal Engine also includes a custom type of composite node, called simple parallel node (represented by a double arrow in Figure 2.15: "⇒"). This class of composite node executes a main task (the one that in the diagram is connected to the parent node with an arrow that starts in a small purple circle) and a secondary subtree (which can be a single task as well, connected to the parent with an arrow whose origin is a small grey circle) that keeps running until the main task ends. These nodes are useful to parallelize the tasks of an agent, i.e. to do multiple actions at the same time. In the implemented tree, this type of node is used to try to attack the player while chasing them (or while waiting in the closest possible location in the case of guards, whose movement area is limited), and to be able to store the enemy's weapon after patrolling for a while, without having to stop walking. The action of going to a certain location, which is used when patrolling or chasing the player character, uses Unreal Engine's navigation system to determine an approximation of the shortest path to the goal location.

When an agent can see the player, the first sequence under the main selector node is executed. The character starts running and equips a weapon if they have one (otherwise the node returns false, but the sequence is not stopped thanks to the force success decorator). The agent subsequently starts chasing the player, which has a time limit that is reset if the player character moves, so that the pursuit continues while the player character is in the field of view and not defeated (unless the agent is a guard, in which case the chasing action ends as soon as the player character leaves the guard's area and the agent reaches the limits of the zone). At the same time, the agent tries to attack the player, but the exact way of doing so depends on the agent's circumstances. If an enemy has no weapon, attacking will be only considered when the distance to the player character is close enough to be able to hit them with close combat moves. The agents that carry a weapon try to shoot the player character if they estimate that the angle between their aim vector and the vector from the weapon to the location of player character's head (i.e. the perfect aim vector) is small enough to have a high probability of hitting. They also use an approximate trace to ensure that the shot would not be blocked by an obstacle, but they cannot measure this with complete precision (it would not be realistic), so they sometimes fail. Shooters continuously rotate their arms to try to aim at the player character's head, if they can see the character. When the enemies do not have enough ammunition to fire a burst, they reload the weapon first.

When an agent has information about an unseen player character, they set a reasonable time to investigate the zone where the suspicious stimuli originated, before considering the information obsolete and forgetting it. Firstly, the enemy stops attacking if they were doing so. This task always returns true, no matter if no attack was taking place, and it also leads to weapon reloading, if that is possible and needed. Secondly, the enemy starts equipping their weapon if they have one and it is not equipped yet. The task ends after a while using an internal timer, so it is non-blocking. In the meantime, the enemy moves to the suspicious location. Depending on the circumstances, the location is the last place where the player character was spotted, the approximate location where some sound that might be related to the player was heard, or the place where an unreported or moved unconscious enemy body was seen. If the agent detects the character at any time the tree will reset to execute the combat subtree, but otherwise the enemy will try to predict where the player character has gone when reaching the previously mentioned location. For example, when an unconscious body is seen the agent checks the body's orientation to try to guess the direction from where the player character attacked and where they might have gone. When the player character stops being visible, the forward vector of the character in the last-seen location is assumed to be the direction where the character has gone. In other cases, the agent goes to a random location that is reasonably close, to investigate. If no evidence or clues of player actions are discovered after some time, the enemy forgets the obsolete information.

When no relevant information about the player is available, an agent starts reloading the carried weapon if that is possible and necessary, and they subsequently walk around and patrol the environment in random directions, heading to different places from time to time, with the goal of finding the player character or some information that may eventually help to detect them. While chaser agents can go anywhere, guards stay in a limited zone: the patrolling task determines where guards can go, ensuring that they stay within their restricted movement area. The temple has multiple hideouts, i.e. partially enclosed spaces where the player character can hide, but some of them are watched over by guards and all of them are periodically visited by chasers, since they patrol the totality of the temple. For this reason, passively staying in one of the hidden locations is not enough for the player character to remain undetected and survive for long. The agents store their weapon (if they were holding one) after some time, unless they obtain relevant information about the player.

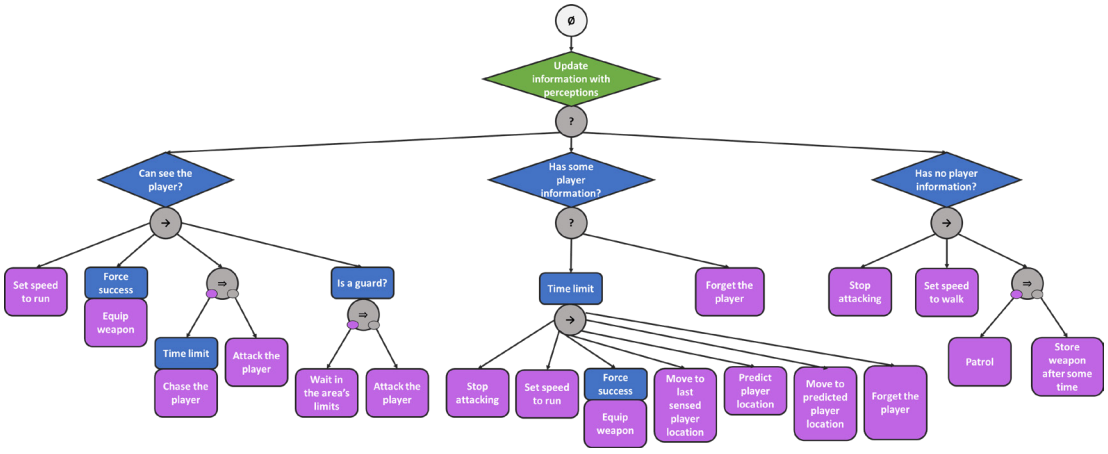


Figure 2.15: Representation of Stealth Temple's behaviour tree for enemy characters, with some changes to make it easier to understand.

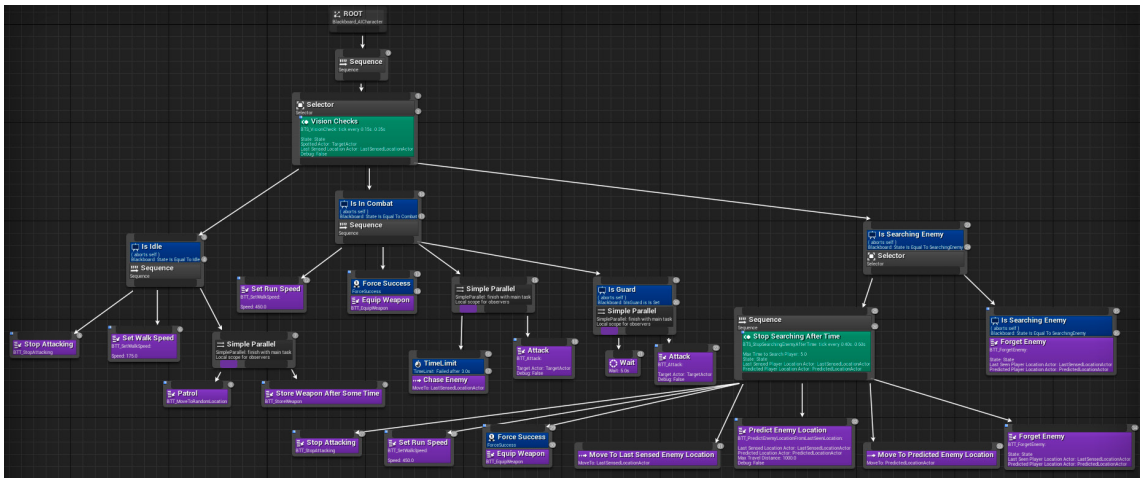


Figure 2.16: Stealth Temple's behaviour tree for enemy characters, as seen inside Unreal Engine's behaviour tree editor.

2.2.3. Implementation of the visual perception

Even though Unreal Engine has a built-in system to handle visual detections of game elements and use events to tell characters when they can see something, it was decided to implement the vision system from scratch to be able to customize it as desired and apply some optimizations to restrict it for the intended use. The behaviour tree of the enemies has a service node that periodically checks whether it is possible to see the player character, an object thrown in the air, or an unreported or moved body of an unconscious agent. If so, the enemy reacts to the stimuli and determines whether valuable information can be obtained from the perceptions, in which case they communicate their findings to other agents. As stated, there are three different types of entities that can give useful information (the player character, thrown objects and unconscious agents' bodies). The process of checking if they are seen is similar for the different mentioned cases, but with some different steps to optimize the procedure. One should think of the ideal visual detection process as the agent asking a neutral, omniscient entity which elements can be seen from the agent's eyes, but in practice it is the actual enemy who checks the information about other entities directly (such as the location), since it is available in the engine. This makes the checking process easier but requires ensuring that the agent does not cheat, i.e. that the enemy does not use any information that was used to determine if something can be seen (e.g. the location of a hidden player character) to take decisions, unless the result of visual detection allows the agent to know that information. For example, if an enemy can see the player character, using their location to take decisions is not cheating, since it is obvious where they are. When checking throwable objects (to see if they have been thrown in the air and they are visible) and agents (to see if they are unconscious and visible), the enemies access the lists of both types of entities found in the enemies' shared knowledge structure. The visual detection process checks first if the player character can be seen. If not, throwable objects and agent bodies are checked afterwards. If a single entity is seen, the search ends and the agent communicates the findings to others and updates the individual decisions.

The first step to determine whether the player character can be seen is to calculate the distance between the enemy and the player character to check that they are not too far away from each

other, otherwise it is impossible for the enemy to see the player character. If the distance is reasonable, the next step is to check that the player character is inside the cone-shaped region that models the vision field of the enemy. To do so, the vector from the enemy character to the player character is obtained. The dot product between this vector and the agent's forward vector (i.e. the centre-of-vision vector) is calculated, and the arccosine trigonometric function is used to determine the angle between the two vectors. If the angle is lesser or equal than half of the field-of-view angle of the agent, the player character is, at least partially, within the enemy's vision cone's region[6]. When that happens, there is only one step left to determine whether the player character is visible, and it is to discover whether some body part of the character can be hit by the vision ray from the enemy's eyes. A set of line traces are created to test for collision from the head of the enemy to different body parts of the player character: the head, elbows, hands, thorax, abdomen, knees and feet. If some blocking entity (e.g. a wall) intersects with a trace, the body part is hidden by some obstacle. If at least one body part is hit by one of the ray traces, the process ends with the player character considered as visible, in a location that the enemy character knows and can share with other agents. The use of multiple traces to body parts, instead of a single check (e.g. to the body centre) allows to visually detect the player character even if only a foot, a hand or any other tested body part is visible. The visual detection process is depicted in Figure 2.17, while Figure 2.18 shows an in-game example.

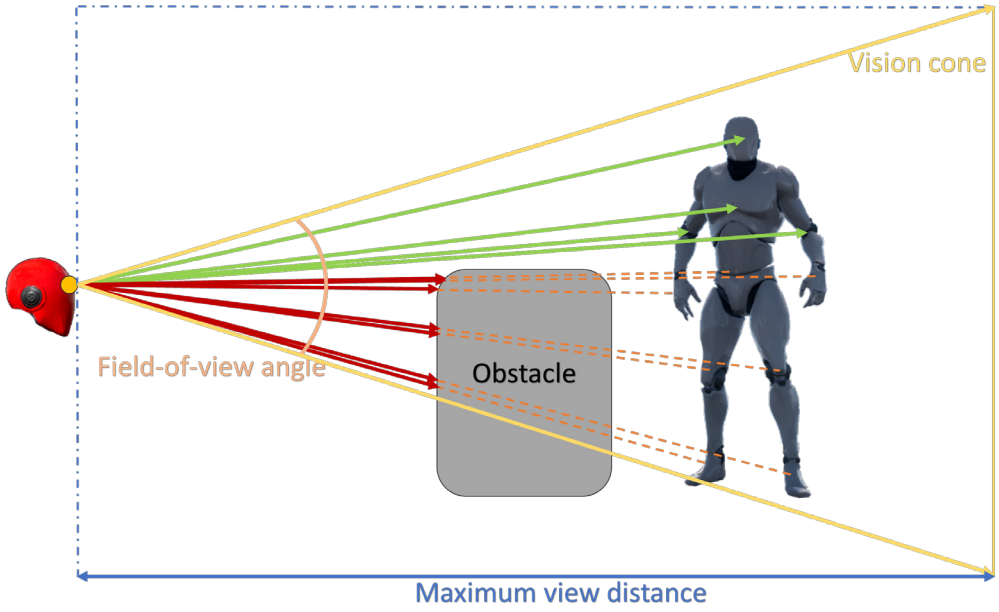


Figure 2.17: Representation of an enemy character's visual detection of the player character, who is within the agent's vision cone (defined by the field-of-view angle and a maximum view distance), and with some body parts hit by vision rays, while others are blocked by an obstacle; this representation is two-dimensional, but the game uses a three-dimensional implementation.

[6] The division by two is required because the calculated angle (the one between the two checked vectors, which is the angle from the agent's centre of vision to the player character) needs to be compared with the angle from the centre of vision to the lateral limit of the field of view, i.e. the limit in one side, either right or left, but since there is another side, it is only half of the whole field-of-view angle.

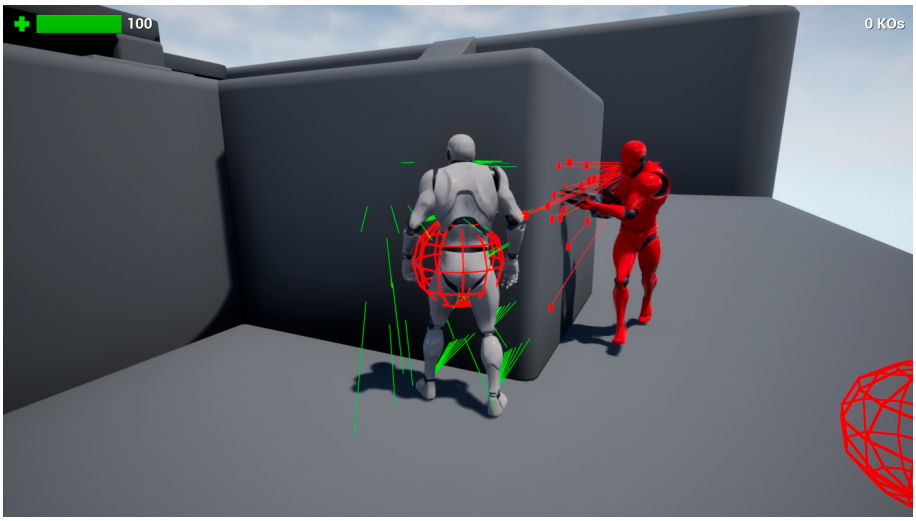


Figure 2.18: An enemy character has discovered the player character after spotting their right elbow (all the other checked body parts were behind an obstacle); the enemy character memorizes the player character's last-seen location (represented with a red sphere in this image, which is not rendered under normal circumstances).

In the case of throwable objects, the first check is very simple: to be able to catch the agent's attention, an object must have a high speed. Otherwise, the object might have been pushed by another agent while walking or running, which is not interesting for the enemy. If the speed is high, it is checked whether the object is within the field-of-view cone in the same way as in the player character's case. If so, a single trace to the object's centre determines whether it is hidden or visible. If the object is seen in movement, a ray trace that starts in the object and ends in the first surface found in the direction of the inverted version of the trajectory's forward vector is used to estimate the location from where the object might have been thrown by the player character, i.e. the approximate place where the player character is thought to be[7]. After this, the most usual decision of the enemy is to turn to the estimated location and share the coordinates with other agents. If the player character throws an object without hiding, this can help agents detect the character if they see a projectile in movement and then turn to the estimated throwing point, as seen in Figure 2.19. Even a hidden player character can be eventually detected, since the enemies will investigate the zone where an object has been thrown.

[7] The agent estimates the source location of the object's movement without treating the object as one with a gravity-influenced movement (i.e. with parabolic trajectory). Instead, a linear movement is assumed. For this reason, the estimated location can be imprecise, but it allows to know that the player character threw the object from somewhere moderately close to that point, and this approach has proved to be effective when put into practice.

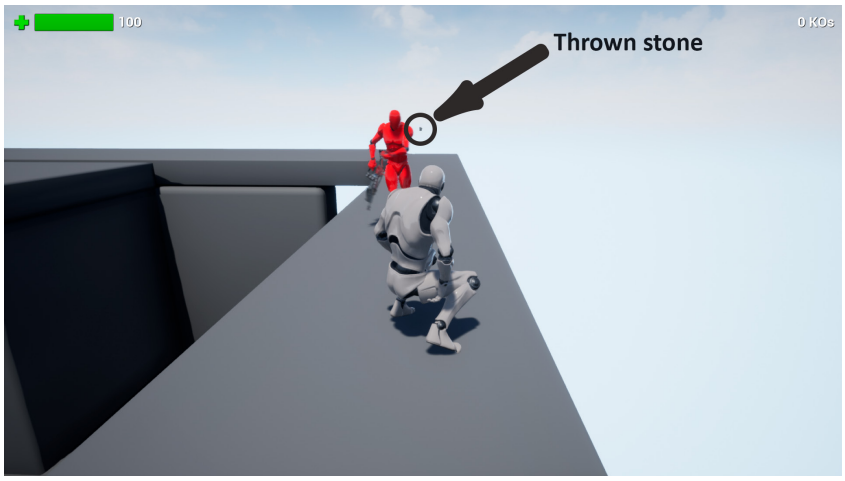


Figure 2.19: Player character detected by an agent that estimated the location from where the player character had thrown a stone and turned to that point, where the character was immediately spotted.

In relation to unconscious bodies of enemies, it should be noted that not all of them give relevant information. The only bodies that are potentially connected with recent player actions are those that are seen unconscious for the first time (i.e. no agent had reported them as knocked out characters, so maybe they had been defeated a short time before), and those bodies that are seen while being dragged (which makes it obvious that the player character is deliberately moving them) or pushed with high speed (which can happen if the player character hits a body while running). For each body, it is checked whether it is being dragged by the player character or has a high speed. In any of the two cases, it is determined that the player character is in a location very close to the body (from where the dragging or pushing action is taking place), as seen in Figure 2.20. Before, it is checked whether the body is inside the vision cone and not hidden by something, with the techniques used for the player character. If no bodies are moved, it is still necessary to check whether at least one of those agents that have been knocked out but are not reported to be unconscious is within the cone and not hidden, in which case the body is determined to be visible. If that happens, the agent that was performing the visual detection operations may decide to investigate the body's zone to try to find the player character, and other agents are told about the findings, adding the enemy whose body was spotted to the list of reported unconscious agents.

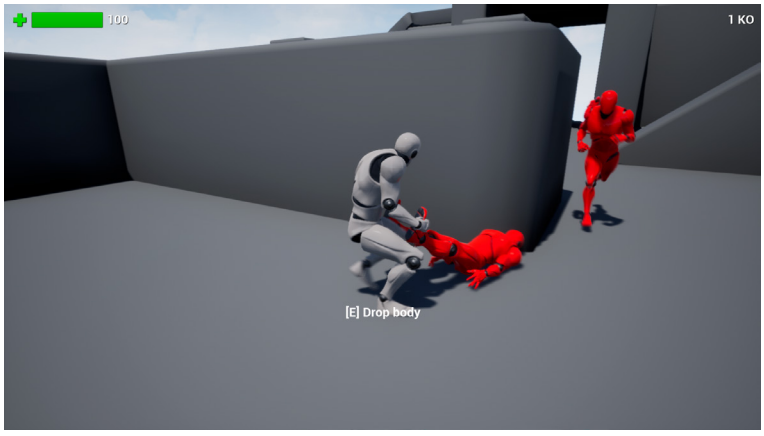


Figure 2.20: An enemy character saw the body of another agent being dragged and deduced that the player character was next to the body.

2.2.4. Implementation of the auditory perception

The player character and throwable objects have been configured to produce sounds that enemy characters can hear and identify if they are close enough and the medium does not attenuate the sounds to the extent of making them inaudible. These sounds are the player character's footsteps and the jump landing noise, as well as throwable objects' impact sounds. A function was defined to set the loudness that these noises should have for enemy characters, since it is not necessarily proportional to the volume with which the human player can hear the sounds: this way it was possible to make the jump landing sound louder for enemies, to penalize the indiscriminate use of jumping. Unreal Engine's built-in perceptions update system was used to notify enemies of sounds with an interruption. This allows to fire an event when a certain type of stimulus (a sound in this case) is perceived, as shown in Figure 2.21. This way the engine performs the complex calculations related to the sound attenuation through walls and other architectural elements, and it will only notify an enemy about a noise if the character is close enough to be able to hear the sound. When an enemy character perceives a noise, they cannot know the sound's exact source location, but they can only estimate it. The estimations of distant source points are more imprecise than the ones of close locations. Sounds that are recognized to be directly caused by the player character (footsteps and jump noises) are given more importance than those made by other entities that may or may not relate to the player character (thrown objects' impact sounds). In any case, enemies immediately share the information (i.e. the estimated source location of the sound and its type) with other agents, using communication.

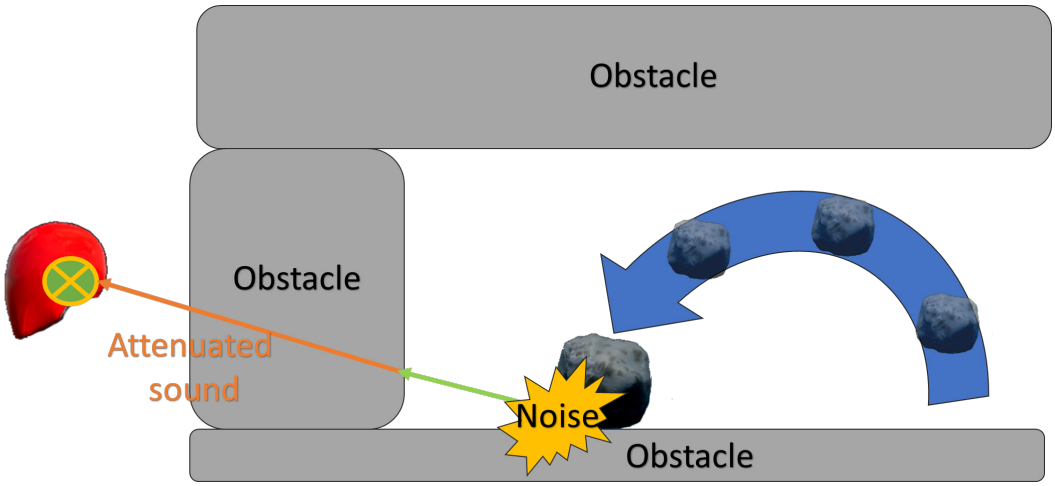


Figure 2.21: Representation of an enemy character's auditory detection of a thrown stone's impact noise, that is close enough to be heard, but attenuated by the medium (there are physical obstacles between the character and the source location of the sound); this representation is two-dimensional, but a three-dimensional implementation is used in the game.

2.3. Implementation of other features to obtain the final game

2.3.1. Definition of the player's objectives, game setting and story

Since the beginning of the development of the game it was clear that it would have enemies with the goal of defeating the player character, who would need to be stealthy to survive. However, the final objectives of the player (i.e. those that would allow them to win the game) were not defined until the last stage of development. Before that moment, the player was able to win when the character touched a golden object. For a long time, the game world was a combination of minimalist platforms, which was an environment that was easy to modify to test different abilities of the characters as they were introduced, and the game had no story. When most of the core gameplay features and the enemies' multi-agent system had been implemented, it was decided to give the game a final setting, a story and player objectives that would potentially make the experience of playing the game more interesting and entertaining.

As explained with more detail in section 1.4, the final version of the game is called *Stealth Temple* and takes place in a temple with four zones, each of them containing an orb that the player needs to find and bring to the centre of the temple to place it in a monument, the *Monolith of Truth*. Orbs are throwable objects, so they can be used in the same way as stones (i.e. they can be grabbed, dropped and thrown). As stated, they can also be placed in the monolith: when the player character is carrying an orb and gets very close to the monolith, a message is shown on the screen to let the player know that the placement action is possible. Each time that an orb is placed in the monolith, the game progress (i.e. the names of the different orbs that are in the monolith) is saved to disk using a special type of Unreal entity, a save-game object. When the temple level is loaded, the save-game object obtains the saved progress from disk and places copies of the orbs whose names had been saved in the monolith, since they had been put there before. The game also loads and applies the saved music volume and camera sensitiveness, which sets how fast the player camera should move in reaction to the associated input.

The orbs and the monolith are some of the few visual elements of the game that were modelled by the developer of *Stealth Temple*. Most of the visual and audio assets were obtained from free samples and content examples that developers who work with Unreal Engine can use in their projects. Some of the exceptions (i.e. the assets made by the *Stealth Temple*'s developer) include the *Rolling Beards* character seen in the introductory video of the game (Video 5.8), some animations for the player character (such as those related to throwable objects and body dragging, although they are a modification of pre-existing animations), some animations for enemy characters (such as the one for an unconscious agent whose body is dragged by the player character), and some materials and textures (such as one with a plus sign used in the health bar). The mesh of the player character and the enemy characters is the default character mesh in Unreal Engine, although the material's diffuse colour parameter was changed to red for the enemies. The temple environment was originally a sanctuary part of an Unreal Engine sample game, and it did not have the four-elements-based decoration seen in *Stealth Temple*. This was achieved by adding assets from other samples to different areas of the sanctuary, e.g. clouds were placed in what would be eventually called the air zone, plants in the earth zone, waterfalls in the water zone, and magmatic rocks in the fire zone. The colour, position and intensity of some lights was also changed to reinforce the perception of having four distinct temple zones, e.g. the red colour was emphasized in the fire element area with the use of very intense red point lights, and the same practice was applied to other zones with different colours. The geometry in the centre of the temple was replaced by the *Monolith of Truth*, which was modelled in 3ds Max (as seen in Figure 2.22). The orbs, whose mesh was designed in the Scupltris 3D sculpting software (as shown in Figure 2.23), were added to the zones of the temple. Some of the assets that were not created by the developer come from external sources other than the

Unreal Engine, namely the public domain sounds played when the characters receive damage, which were obtained from Freesound, a website where free sounds can be downloaded[8].

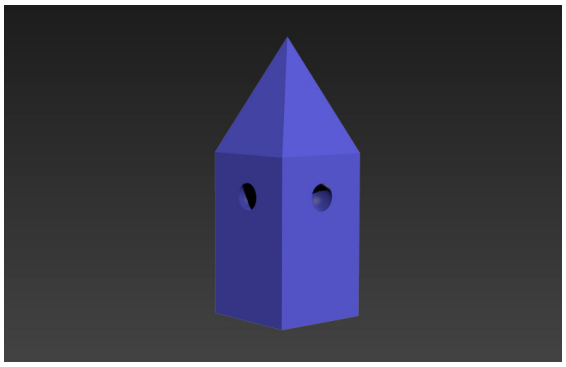


Figure 2.22: The Monolith of Truth as seen in 3ds Max.

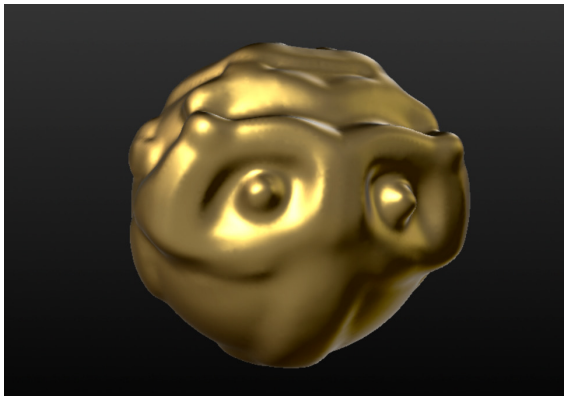


Figure 2.23: An orb, as seen in Sculptris.

The addition of the game story, narrated in section 1.4, gave a background to the game characters and explained the reason why the orbs should be placed in the *Monolith of Truth*: to purify the temple's guardian robots, that had become hostile and perilous because the *Rolling Beards* criminal gang defeated them after attacking the temple, and reprogrammed them to become killing machines. The story of the game is very simple, as well as the cinematic video that explains it (Video 5.8). The video is the result of combining different takes, i.e. video captures of the screen recorded while running the game. To obtain some of the takes, certain aspects of the game were temporarily changed. For example, the player character was replaced by a floating, invisible character to make some scenes more immersive. In other takes, the character was given the appearance of one of the members of the gang that attacked the temple, to explain what they did. The criminal character was designed in a character creator software called Fuse, as shown in Figure 2.24. Fuse includes customizable modules (i.e. different types of body parts and clothes) that can be used to produce characters in a fast way, without modelling them from scratch. The introductory video was set to be the game's start movie; it is automatically played when the game is started, but it can be skipped by pressing any key or button, which leads to the main menu.

[8] Freesound's main page can be found in: https://freesound.org.

Figure 2.24: Criminal that attacks the temple in the introductory video, as seen in Fuse.

2.3.2. Implementation of the in-game user interface

The user interface (UI) that is seen while playing the game is also called Heads Up Display (HUD). The elements that form it were designed using Unreal Motion Graphics (UMG), the engine's tool for creating widgets and other UI elements. The HUD contains a health bar composed of a plus sign image and a progress bar, that are updated when the player character receives damage. The colour of both elements is the same, and it is obtained with a linear interpolation between tones of green (seen when the character's health bar is full) and red (visible when the health is almost zero); a grey colour is used when health is zero. Different values of health in the health bar can be seen in Figure 2.25. The HUD also shows the number of enemies that the player has knocked out, a quantity that is updated when a new agent becomes unconscious.

Figure 2.25: Different values of health make the health bar have distinct colours, ranging from green (maximum health) to red (minimum positive health), and grey when health is zero.

An in-game menu is shown when the player character is defeated, which contains two buttons that allow to restart the game (an action that causes the temple level to reload) or quit the game (which leads to exit the temple level and load the main menu). When the player pauses the game, the in-game menu is shown with an extra option to resume the game, as seen in Figure 2.26.

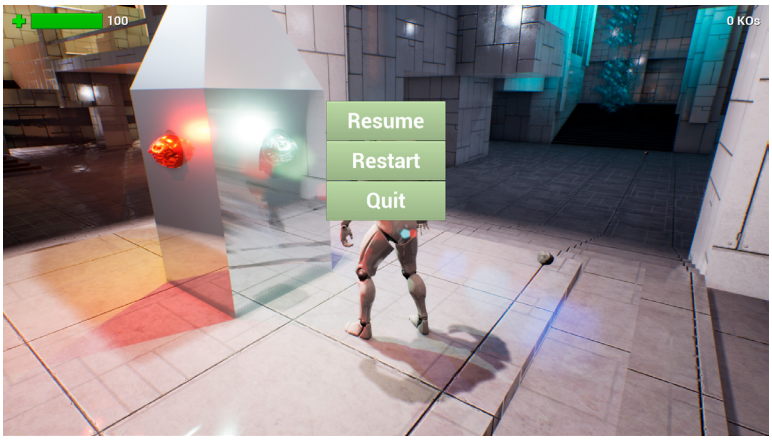


Figure 2.26: Stealth Temple's pause menu.

2.3.3. Implementation of the main menu and game settings

The main menu exists as an independent level that shows a background image and uses UMG-designed widgets with events (e.g. click) that make them interactive. The menu is composed of four sections (main menu buttons, controls, settings and about). Each of them is encapsulated in a container, and only one container (including all children widgets) is visible at a time. By default, the section with the main menu buttons is shown, and a different event is triggered depending on the clicked button (*Play*, *Controls*, *Settings*, *About* or *Quit*). The *Play* button loads the temple level (quitting the menu). The *Controls* button hides the main menu buttons and shows a table with the game controls. The inverse action takes place when the *Return* button of the controls section is clicked. When the *Settings* button is pressed in the main section, the main menu buttons become invisible and the game settings are shown. The options include the screen resolution (adjustable with buttons of different resolutions, that apply their resolution to the game when clicked), the camera sensitiveness (that can be changed with a slider) and the music volume (that also uses a slider). All three settings are saved when altered using a save-game object, and their last set value (or the default one, if it is the first time that the game is run) is obtained from disk when the main menu is loaded. Both the camera sensitiveness and the music volume can be reset to a default value. The settings section also contains a red-coloured button to delete the game progress (the saved names of the orbs placed in the *Monolith of Truth*), but the action is not immediate when clicking the button. Instead, a confirmation button is shown (that can be clicked to effectively delete the data), along with a cancel button to abort the action (which can be clicked to make itself and the confirmation button invisible and disabled). The *Return* button below the settings allows to hide the section and show the main menu buttons. The *About* button hides those buttons to show the about section, composed of text with the name of the developer of the game and some information about the game engine. It contains a *Return* button to hide the section and show the main menu buttons. The *Quit* button allows the player to stop the game. Different images of the main menu are shown in section 1.4: Figure 1.4, Figure 1.5 and Figure 1.6.

3. Conclusions and possible improvements

The objective of this project was to develop a stealth game with an artificial intelligence-based multi-agent system that would model the behaviour of enemies, and this goal was achieved. All the planned development tasks were completed within the expected time range. The motivation of the project's developer was to learn how to design and implement behaviour trees to make non-player characters take decisions and execute actions in consistent and realistic ways, as well as gaining experience in game development and particularly in the implementation of game-oriented artificial intelligence solutions; these objectives were accomplished.

Behaviour trees proved to be very flexible during the development process: it was easy to modify the structure of a tree to incorporate new character abilities and types of decisions. The final game's agents can detect characters and objects with vision and hearing. The perceived stimuli are used to obtain information that can potentially help them achieve their goals by taking decisions and sharing their findings with other agents, who compare the received information with their own knowledge. In other games, however, the reasoning capabilities of non-player characters is much more elaborate than in *Stealth Temple*, and agents participate in complex cooperative strategies. There are many abilities that non-player character use in other games (such as driving vehicles or negotiating with characters that have different goals) that would be interesting to model using behaviour trees. The interaction between the agents in the developed multi-agent system is not very complex (it was not necessary for the enemies' objective of chasing and defeating the player), while other games, such as some strategy games, have multi-agent systems in which agents interact in many creative ways. Some games include non-player characters that instead of trying to defeat the player have the objective of helping them to achieve their goals. It would be interesting to model the way how these characters try to interpret the player intentions (observing the player character's actions) and suggest ways of achieving the shared goals: it would require defining a human-machine communication system.

The gameplay of *Stealth Temple* includes some interesting player abilities, such as throwing objects from hidden locations to distract agents, attacking enemies from behind, and hiding unconscious agents' bodies to avoid detection. Other stealth games, however, allow the player to do more actions, such as disguising their character (e.g. using a defeated enemy's clothes as a costume) to make enemies perceive the player character as a different entity, which can enrich the gameplay. Allowing the *Stealth Temple*'s player character to disguise as a knocked-out enemy for some seconds would be an entertaining feature. It would also be interesting to make the character visually undetectable in darkness. This would require altering the enemies' visual perception to consider the light intensity in the body parts of the player character, checking point lights and directional lights in real-time to measure their light contribution to the character.

This project was not focused on the design of visual or audio assets. However, many games have a more consistent art style because the assets they use were mainly designed by the developers, instead of using assets from free samples. This practice can help to achieve more original and professional-looking results. To produce more immersive cinematic videos, many developers use tools that allow to control the movement of cameras and animations in modular ways, such as Unreal Engine's Sequencer, instead of directly capturing game scenes, which was the approach used to produce *Stealth Temple*'s introductory video sequence.

4. Bibliography

EINHORN, Asher. *Introduction to AI - Part 1: Behavior Tree Basics*. Website: Youtube.com, June 2015.

EINHORN, Asher. *Introduction to AI - Part 2: UE4 Behavior Tree Specifics*. Website: Youtube.com, Jun. 2015.

EINHORN, Asher. *UE4 Tutorial Series - AI*. Website: Youtube.com, Dec. 2015.

EPIC GAMES. *Unreal Engine 4 Documentation*. Website: UnrealEngine.com, frequently updated, last checked in May 2018.

EPIC GAMES. *Unreal Engine API Reference*. Website: UnrealEngine.com, frequently updated, last checked in May 2018.

LEMCOVICH, Alex (*Stealth Docs*). *A History of Stealth Games*. Website: Youtube.com, Sep. 2017.

LEMCOVICH, Alex (*Stealth Docs*). *Stealth Game Analysis*. Website: Youtube.com, Dec. 2017.

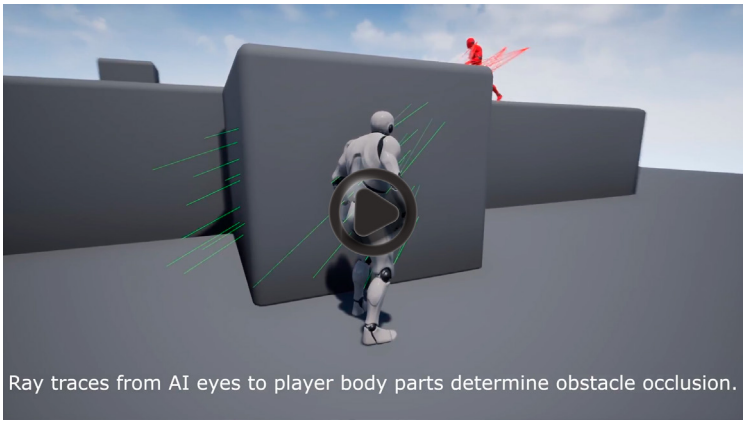
POOLE, David; MACKWORTH, Alan; GOEBEL, Randy. *Computational Intelligence: A Logical Approach*, Chapter 1. New York: Oxford University Press, Jan. 1998.

SIMPSON, Chris. *Behavior trees for AI: How they work*. Website: Gamasutra.com, Jul. 2014.

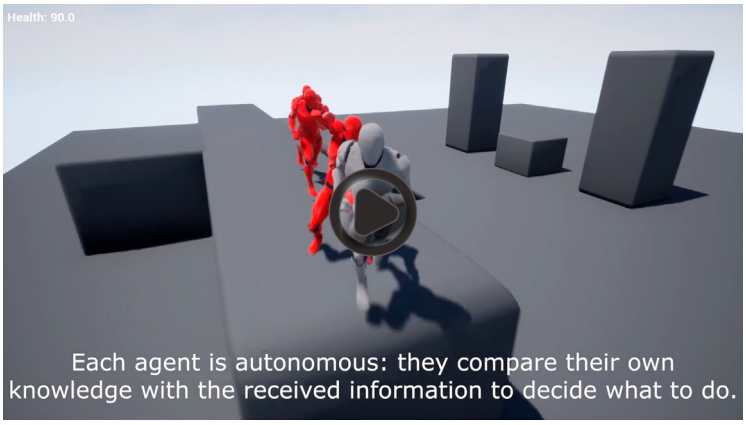
5. Appendix

5.1. Demonstration videos

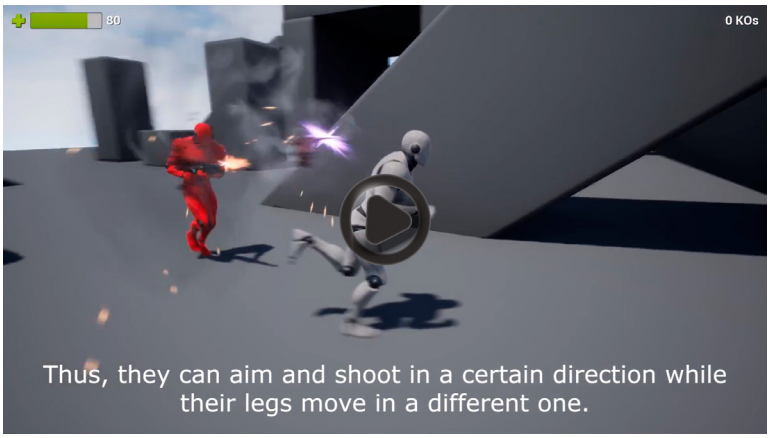
Playlist 5.1: Playlist with the 9 demonstration videos in chronological order.



Video 5.1: Video that shows the enemies' visual detection of the player character, as well as patrolling and chasing capabilities.

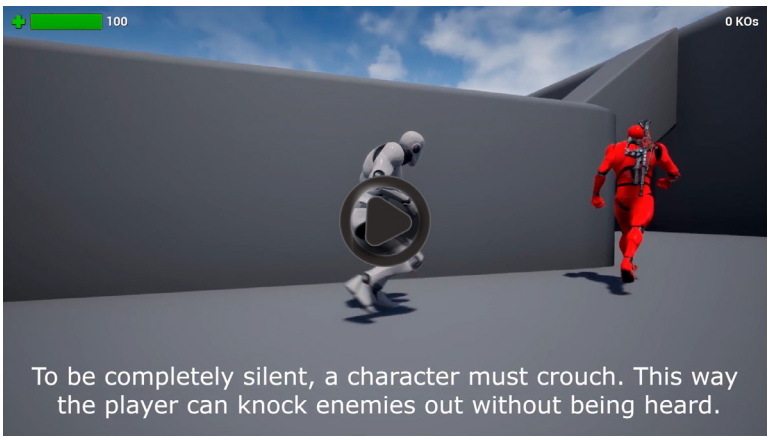


Video 5.2: Video that shows the enemies' initial communication skills and close combat moves.



Video 5.3: Video that shows the enemies' ability to shoot.

Video 5.4: Video that shows the player character's ability to knock enemies out.

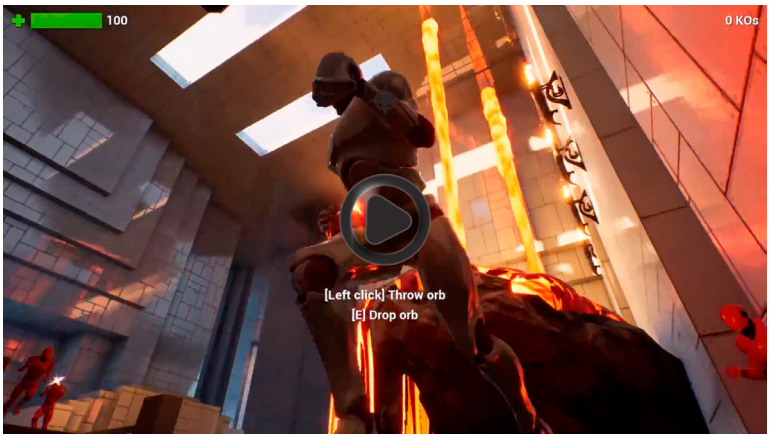


Video 5.5: Video that shows the enemies' auditory detection of the player character.

Video 5.6: Video that shows how the player can throw stones and grab the bodies of unconscious agents, and how the enemies can reason about signs of player activities and investigate them in the environment.

Video 5.7: Video that shows how enemies take different roles and coordinate to defeat the player.

Video 5.8: Stealth Temple's cinematic introductory video.



Video 5.9: Stealth Temple's gameplay video.