How to build a robust and reusable AI Sensory System

Bernd Sommeregger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2013

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 28, 2013

Bernd Sommeregger

Contents

Kurzfassung

Bei der Entwicklung eines Stealth Games wird im Gegensatz zu vielen anderen Genres großer Wert darauf gelegt wie ein Agent die Spielwelt wahrnimmt. Je nach Komplexitätsgrad kommt ein Sensory System zum Einsatz, diese Arbeit stellt verschiedene Implementierungsansätze vor und beschreibt die Entwicklung eines Sensory Systems für die Unity3D Engine. Die Implementierung verwendet Sensoren und Signale um auf der einen Seite die menschlichen Sinne abstrakt nachzubilden und auf der anderen Seite um Objekte von Interesse für die Agenten zu markieren. Des Weiteren wird ein Regional Sense Manager verwendet der diese Sensoren und Signale an einer zentralen Stelle verwaltet. Es wird überprüft ob Sensoren Signale wahrnehmen können und die Ergebnisse dieser Überprüfung bilden die Basis für eine spätere Implementierung der Reaktion eines Agenten.

Für das Sensory System ist eine Testumgebung erstellt worden mit der die Implementierung getestet wurde und die veranschaulicht wie das System bei der Entwicklung eines Spiels eingesetzt werden kann.

Abstract

During the development of a stealth game a high value is set on the way a agent perceives the game world, contrary to other genres. Depending on the complexity a sensory system is utilized, this thesis presents different approaches to implement such a system and describes the development of a sensory system for the Unity3D engine. The implementation uses sensors and signals to create an abstract emulation of the human senses on the one hand and to flag objects of interest to agents on the other hand. Additionally a regional sense manager is used to administer the sensors and signals at a central location. It is being checked if sensors can perceive signals and the results of those checks are the basis for a future implementation of the agents reaction.

A test environment has been created for the sensory system in order to test the implementation and to illustrate how the system can be utilized during the development of a game.

Chapter 1

Introduction

1.1 Motivation

The advancements in video game productions in the past years where mainly focused on improving the visual representation, whereas one area that offers the opportunity to actually realize gameplay improvements and innovations has been neglected by many productions for a long time: artificial intelligence. Well implemented AI allows for deeper immersion and the creation of more novel gameplay mechanics and there has been one genre that always focused on elaborate AI behaviour: stealth games. These rely on the ability of agents to perceive their environment and react properly based on that perception. But the development of a sensory system requires some detailed knowledge about AI and costs time and many developers like indie studios, students or hobbyists don't have many resources to spend. They need tools that reduce their workload and help them to speed up the development.

1.2 Objectives

This thesis engages in the development of a sensory system for the Unity3D engine with the intended goal of building a tool that aids small developers in the creation of stealth games. This genre was chosen because it depends on the utilization of a sophisticated sensory system to create the gameplay.

The tool should be simple to use and set up, without the need to invest a lot of time to become acquainted with it. The intent is to offer the possibility to easily implement AI without requiring a deep understanding of the field and additionally having the chance to concentrate resources on another aspect: the implementation of the actual gameplay.

The result should be showcased in a small example implementation to demonstrate the abilities of the system, point out the possible application range and act as an introduction on how to utilize it.

1.3 Thesis structure

The thesis starts with a basic introduction of AI in games as well as stealth games and the specific requirements this genre demands of the AI. After that the possibilities regarding the options to interface with the world are quickly examined and explained, additionally sensory systems and their perception process is clarified.

Chapter 3 gives a review of some of the existing systems that are used to realize AI perception and sensory systems in games as well as some principles that should be taken into account when developing such a system.

After this the implemented sensory system for the Unity engine is described in detail in Chapter 4. Starting with the requirements the system has to fulfil followed by a overview of all components and the description of each. The explanations are followed by a detailed illustration of the interactions between those components and a description of the user interface. The chapter closes with an implementation proposal for an alternative sensory system implementation.

The last chapter gives a conclusion of the thesis and a prospect how the described implementation could be expanded and improved.

Chapter 2

Artificial Intelligence in Games

2.1 Introduction

This thesis deals with Artificial Intelligence (AI) in video games, but not in the sense of traditional AI and therefore the differences between those fields must be explained first. One of the definitions for AI was made by John McCarthy, describing it as "the science and engineering of making intelligent machines". AI research is divided into serveral subfields, but one of the long term goals is the so called stong AI. This describes an attempt to match or even exceed human intelligence.

Unlike traditional AI, AI in video games wants to create the illusion of intelligence. It is centered on gameplay and on how the AI appears to the player. This leads to one of the most important differences, the fact that cheating is allowed and workarounds are not depreciated but sometimes they are even the better solution to solve a problem in time and on budget. Furthermore game AI is not aimed at creating the best possible actions, the ambition is to create a challenge for the player. Because of that it needs to account for fairness and therefore it should not exceed human skill, because the player still needs a chance to beat the AI.

But the fact that it is often admissible to cheat, doesn‘t mean that it can be considered an answer to every upcoming problem. Because it often can cause unrealistic behavior, for example it would be possible for the AI to always know the position of the player. However it would be not desirable, because it would be very obvious and the player would be aware of the cheating.

Game AI always puts the focus on creating a good gameplay experience and always favours the solution that looks better to the player, even if it is the simpler one.

"Knowing when to be complex and when to stay simple is the

> most difficult element of the game AI programmers art. The best AI programmers are those who can use a very simple technique to give the illusion of complexity" [2, section 2.1].

2.2 Stealth Games

In this genre the player needs to evade enemies and remain undetected. The gameplay is about hiding and trying to avoid alerting enemies. In general there are numerous ways for the player to make it through a level. The core gameplay elements include hiding behind objects and in shadows, disabling light sources, observing enemies and memorizing their pathways. There are options to distract the AI as well, like making a subtle sound to direct the attention to some point, while sneaking past the guard.

If a non-player character (NPC) detects the player, the game isn‘t over but the player can try to escape, hide and wait until the enemies stop searching. Most games leave the decision to engange the enemies or not, to the player thus enabling him to choose between stealth tactics and confrontation, or combining the two.

The gameplay varies, depending on what player actions the AI is able to perceive and the reaction. For this type of gameplay to work, the AI must be ignorant to some parts of the world. For example, although it isn‘t realistic that a guard can‘t see a player in the shadow directly in front of him, this fact allows the player to hide in the shadow. In addition the guard can‘t know the position of the player all the time, by just looking it up in the code. This kind of games need a more sophisticated perception system for them to work. The AI needs to be able to see and hear the world, while still bearing in mind that this must occur somewhat abstract, because the goal is to create fun gameplay.

2.3 AI in Stealth Games

In many games AI agents make decisions that are rather simple, like remaining idle or attacking. Similar to that there are a lot of behaviours that don't need to be based on actual perception of the world. For example in most Real-time strategy (RTS) games the agents don't need to be able to hear the footsteps of enemy soldiers to be able to perceive and attack them. It would be sufficient for the AI to utilize a simple attack radius, and assault enemies who enter it. This could be expanded by using a view cone, which would enable the agent to see the world, whereas that is just a constricted attack radius. In contrast, agents in stealth games need to perceive their surroundings in a much more comprehensive fashion.

Why is that? Let‘s start with a simple example: If a guard sees the player, there is need for quite a large range of distinctions to be made by

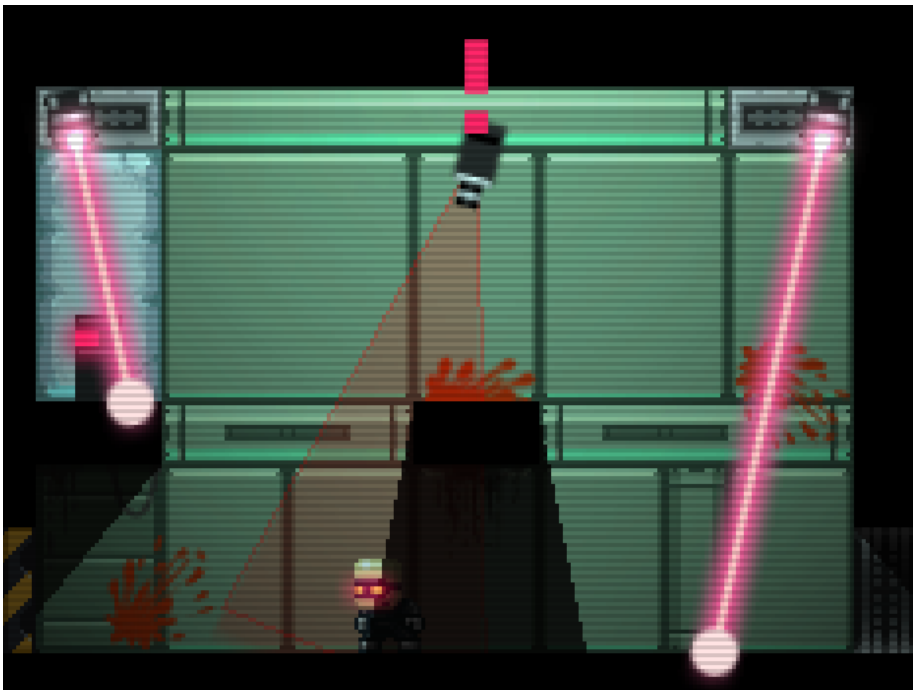

Figure 2.1: Screenshot from the game "Stealth Bastard" [5].

the agent. For instance, the guard only sees a fraction of the player for a short time, before the player disappears behind an obstacle. The AI cannot immediately draw the conclusion that that was the player and start chasing him. It must take several factors into account, like the distance to the player or the angle at which the player is entering the field of vision. Because if the agent see's something that is far away, he shouldn't be able to accurately determine what that object is, he should only get suspicious and maybe start moving closer to figure out what it is exactly. And the same thing goes for sounds. This is commonly expressed by different alert states, those describe roughly the current condition of an agent, like idle, suspicious and alerted. Depending on what the AI perceives the agent switches to a distinct state. The transitions between them must be clearly represented to the player, this is done amongst others by displaying a question mark if the agent is suspicious or an exclamation mark when he is alerted.

One large part of the gameplay is built around the perception of the AI. If an agent detects just a little fraction of the player and immediately starts alerting his allies and begin chasing him, that would be very frustrating. To a certain degree there has to be a margin for error to allow for fun and exciting situations. But that isn't enough, the agents need to be able to distinguish different circumstances when detecting the player. Is he in a restricted area or not? If not, is he visibly carrying a weapon? Is he carrying

a body? When the AI is already alerted and is engaging the player, can the agent shoot at him or is he using an ally as a human shield? Furthermore if the player made changes in the level, like disabling a light source to remain undetected or break a window to gain access to a certain area, the agent has to recognize and react accordingly. As you can see, depending on the details of the gameplay, there are numerous things for the AI to consider.

2.4 World Interfacing

The AI needs to be able to react reasonable to events that happen in the game world. To do that, agents don't only need the ability to detect the player, but they also have to know what is happening to their allies and to a certain extent, what parts of the level are changed by the player. So there is need for some kind of infrastructure to obtain information and provide it to the agent who needs it. If we assume to build a reusable general purpose AI architecture, a specialized AI for a single game might not require a separate system dealing with the world interface. The question is what's the best way to get information?

2.4.1 Polling

One option is called polling and refers to actively sampling the desired information. Every element of the game world that can change and might be interesting to the AI is polled and checked, for example if the player is colliding with the agent. The advantage of this technique is, that it's fast and easy to implement. But as the number of things the AI needs to check grows, a big disadvantage arises. The number of checks grows, but the number of positive results decreases, leading to the case that the AI spends most of its time doing checks, which return negative results. Additionally with the number of checks growing, the initially fast polling speed can decrease, due to the mere mass of samples. In certain situations this approach is ideal, like a single alarm in a level that alerts all guards.

2.4.2 Events

An agent is notified by a central system when something of interest happened by sending an event to that agent, in contrast to looking for that information by himself. Another difference is the possibility to reduce the amount of checks, like a the single alarm that alerts all the guards. With polling every agent needs to check the alarm by himself (and produces a large amount of negative results), whereas a central system does one check and notifies all agents by sending an event. But sending an event adds quite a lot of processing, especially if compared with the simple action of polling an information. Both options have their benefits, polling is fast but doesn't

scale well and events do scale well but are more complex and over the top for small and simple game mechanics.

2.5 Sensory Systems

A sensory system is a part of the nervous system that transduces information received with senses to actually perceive the world. The human sensory system can sense different sensory modalities and it consists of five subsystems:

- vision,
- hearing,
- touch,
- taste,
- smell.

There is no need to go into much detail, what an AI sensory system does is quite simple: It is trying to emulate the human senses. To be more specific, it does that in very simplified manner, it is by far no elaborate simulation.

While coding a game, it wouldn't be difficult to make a character omniscient, all the relevant data can be accessed easily. For example, every guard could know where the player is at every given time, by simply looking up his position. In fact many games work this way and there are plenty types of games, for which this approach works well enough. But in a stealth game this approach doesn't do the job, the AI can't be allowed to know the position of the player all the time. Because this would understandably undermine the basic gameplay mechanic, which is essentially hide and seek. How the agent perceives the world and gains knowledge about it, requires a different approach. The AI shouldn't gain knowledge by directly accessing information, it should be discovered by perception.

> "A game sensory system must be designed in a way that is subservient to the game design and efficient in implementation. The senses need only be as sophisticated as is needed to be entertaining and robust. The result of their work must be perceivable and understandable by the player" [7].

Most games today are using Finite State Machines to model the behaviour of the AI, a simple example would be the following states for a guard in a stealth game: idle, suspicious, alerted. A sensory system can make even such a small state machine much more interesting and appealing to the player. It could be taken into account, how far the player is away, how much of the player did the AI see? Or how fast did the player move and did the agent hear the correspondingly increasing volume of the footsteps? There are many more ways to improve just the one transition between the idle and

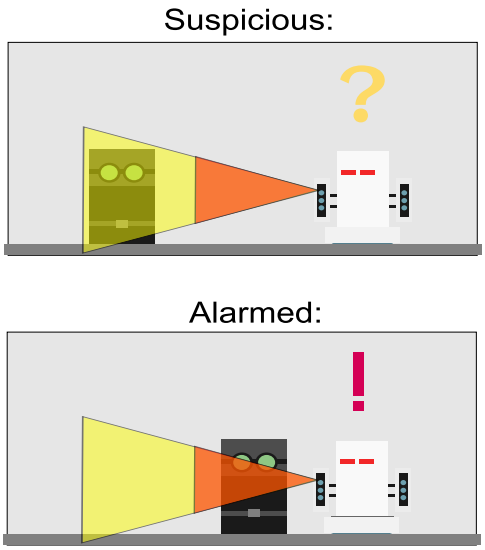

Figure 2.2: Example for a view cone with two awareness levels.

suspicious state, not mentioning the interactions with other agents and level objects (e.g. unconscious guards, broken light sources, light and shadow).

2.5.1 Perception

For an agent to be able to detect objects in the level, it essentially comes down to scanning for intersections. Although there are a few things to consider building a system that is as elaborate as needed and efficient enough to scale well.

Vision

Vision is implemented by creating a sight cone, the basics are the ability to detect game objects that enter it and the inability to see around corners and solid objects. Depending on the gameplay it can assume very different shapes. Starting with a simple single shape, splitting it up into multiple areas to account for several levels of awareness (as shown in figure 2.2) or dividing it into a number of single cones, each responsible for a another influence on sight perception (figure 2.4).

But even the case of having two levels of awareness can entail quite a lot of problems, like how to dimension those two to simulate the field of vision fairly decent (direct vision vs. peripheral vision). Not to mention if additional different gameplay mechanics need to be attended to, like if the player can hide in the shadows, if he can use a camouflage system to hide or if he can use disguises. Granted, some of these things don't affect the composition of the cone, but they need to be considered in the AI code

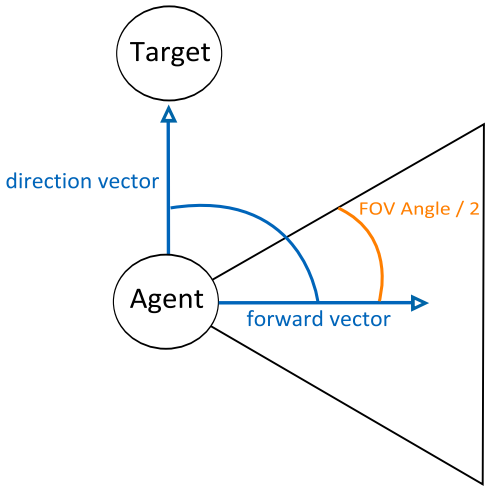

Figure 2.3: Calculation to determine if a target is inside the field of view of an agent.

where the agent decides if and what he actually can perceive.

A common solution to implement a view cone just needs two vectors and an angle, while the angle describes the field of view (FOV) size, there is a forward vector of the agent and a direction vector from the agent to the target. To determine if the target is spotted, the angle between the forward vector and the direction vector is compared to the FOV/2. If the angle between forward vector and direction vector turns out to be smaller, then the target is inside the view cone (illustrated in figure 2.3).

Hearing

Hearing can be approached very similar to vision by creating a spherical area constituting the range a sound can be heard (see figure 2.5). Another possibility would be to assign a sound a certain intensity at its source that is reduced over distance, while each agent has a threshold, below that he can't here the noise[1].

Other Senses

- **Touch**: Touching is obviously needed in most games and can be implemented by using collision detection. Usually the collision system of the engine covers this sense automatically and only the reaction of the agent needs to be implemented. It seems to be quite easy to just make use of the collision detection already in place, but again there can be some additional details to consider. Maybe the player is

[1] As described in "Artificial Intelligence for Games" by I. Millington.

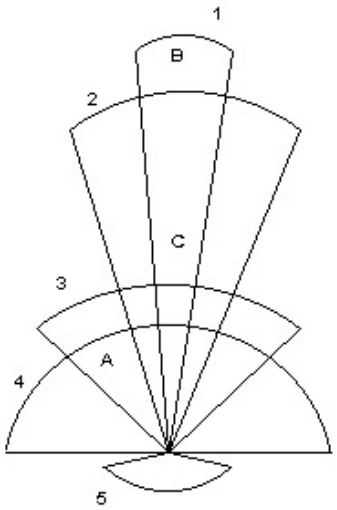

Figure 2.4: View cone from the game "Thief: The Dark Project" [6].

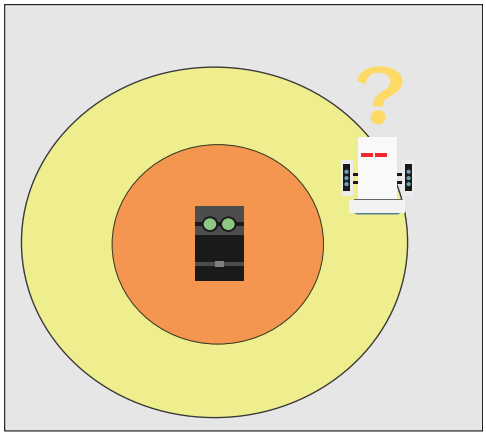

Figure 2.5: Example of a sound sphere with two awareness levels.

offered the possibility to steal items of the guards, and depending on the implementation of this mechanic the touch sense has to be adapted accordingly.

- **Taste**: To my knowledge there is no game that uses this sense as part of a sensory system and so far there are no known ideas how to implement a meaningful mechanic utilizing it.
- **Smell**: This sense is not very widely-used in games, but it offers some possibilities for gameplay. The most rudimental way to build a smell sense would be to just approach it similar to hearing and define a certain influence area (instead of simulating the diffusion realistically),

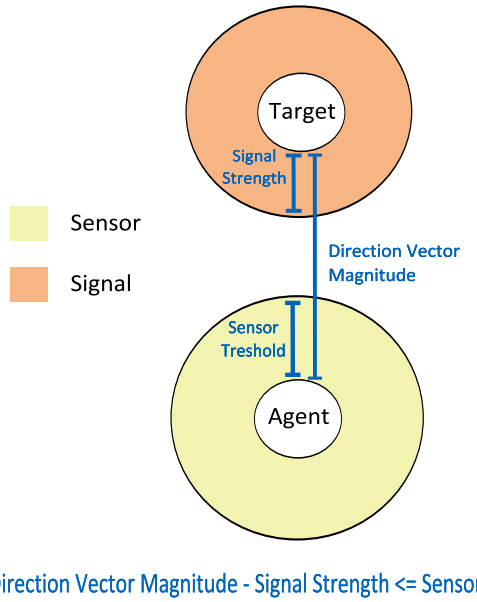

Figure 2.6: The sound perception logic takes the signal strength and sensor threshold into account, as well as the distance to the player.

within a character with an assigned scent can be smelled by an agent. If the diffusion is simulated a bit more elaborated, gas grenades could be created that knockout or kill guards, or the player could leave a scent trail, the AI would be able follow.

- **Fantasy Modalities**: The basic modalities should be created with care, bearing in mind that those senses can be used as building blocks to create a number of fictional senses like magical auras that can detect the player even if he is hidden, spells that modify the basic senses or the introduction of fear, which would cause the AI to react to the actions of the player and alter its behaviour depending on the "fear level". This modalities can be created by altering the behaviour of the basic human senses which have already been implemented first.

Perception is implemented with sensory systems as an abstract version of real human senses, and with the combination of a few senses it is possible to create a broad range of AI behaviour that rests upon the perception result. There is another aspect that further enhances AI perception: the categorization of the game objects that are perceived. The principle is that different senses react to different modalities and game objects can be subdivided into those that a sensor can and can't perceive. For example if an agent has to see the player, vision is added to him and the player is described with a sight modality. Now the agent can see the player because he has the appropriate sense to perceive the corresponding modality by which the player is

described. If the agent needs to hear the player too he would need the hearing sense and the player would need a sound modality, because with vision only the AI couldn't perceive the different modality. Senses only react to corresponding modalities and this can be utilized to ignore modalities that can't be perceived anyway and speed up the system. It is almost vital to have some kind of categorization for perceivable game objects, because otherwise the sensory system would grow out of hand and couldn't be managed anymore.

Those are the basics of a sensory system, on the one hand different abstractions of the human senses and on the other hand abstractions of the perceivable modalities. This concludes the introductive part, the next section continues with related work and gives and overview of some different options on how to implement sensory systems and perception. While the presented solutions have some things in common, they put an emphasis on different aspects and are therefore suited for different domains.

Chapter 3

Related Work

3.1 General-Purpose Trigger System

This system was described by Jeff Orkin in "AI Game Programming Wisdom" [3], it illustrates a central system that keeps track of events that agents can respond to and aims to decrease the processing effort needed for that response. The outlined benefit is the fact that "triggers[1] can be culled by priority and proximity before delivering them to agents" [3, section 2.2]. This remains as a big benefit until today, because the ability to cull increases the performance tremendously, especially if the size of the game world and the number of agents increases. Without effective culling such a system simply wouldn't scale well.

The basic idea of this centralized system is that a trigger is registered when an event occurs. Then the system iterates over all agents and for each agent it tests if the AI is actually interested in one of the triggers (additionally triggers are culled). The system consists basically of two parts:

- Trigger
  - *TriggerRecordStruc*: Defines an instance of a trigger.
  - *EnumTriggerType*: Describes the type of a trigger.
- Trigger System
  - *Register Trigger*
  - *Remove Trigger*
  - *Update Trigger System:* The update function is the heart of the system, expired triggers are removed, dynamic trigger positions are renewed and most importantly, this function notifies agents about triggers of interest, and additionally the culling is also done here.

[1] A trigger describes a catalyst for an event. It is commonly used in games to start a scripted event, when the player activates an invisible trigger.

3.2 Generic Perception System

This approach, presented by Frank Puig Placeres in "AI Game Programming Wisdom 4" [4] understands perception as something more than the simulation of human senses like vision and hearing. It comprises all possible ways that an agent collects information about the world, especially including abstract data that is set up by level designers like environmental data (e.g. nav meshes or obstacle meshes) or tactical data (e.g. cover spots). The system has two big parts (see figure 3.2) :

- *Perception System:* Updates all data gatherers.
- *Data Gatherer:* Stores perceptual information that agents can sense.

The system distinguishes between two types of information:

- *Static Data:* Prefabricated data about static locations like tactical or environmental data.
- *Dynamic Data:* Data that is created and modified at runtime, for example from dynamic objects or events (this includes amongst other things sight and sound perception).

The different types of information are implemented as subclasses of the data gatherer, this allows to only include those ones that are really needed for a certain game. The system has three different data gatherers:

- *Environmental Data:* A simplified representation of the game world to help the AI navigate through it.
- *Tactical Data:* Often manually placed information about the tactical characteristics of the level that are relevant to the AI, like cover spots or sniper positions.
- *Beacon Data:* A beacon encapsulates an event in the game, for example sight or sound messages. The event specific data gatherer notifies the agent of the beacon, and the agent can see or hear the beacon (this is the equivalent to the signal of the regional sense manager, see section 3.7).

This perception system enables AI agent's to be aware of their surroundings, by receiving information about the environment. But to utilize this data, the agent needs one particular capability: a sensor.

Sensors

Sensors inform the agent about useful information in the world, they get this data by connecting to the data gatherers of the perception system, which provide this information (figure 3.2). Depending on what information the agent actually needs he can connect to one or more data gatherers for different types of data: environmental, tactical and beacon data. This happens when the agent is created, it registers only those sensors it will need. It is

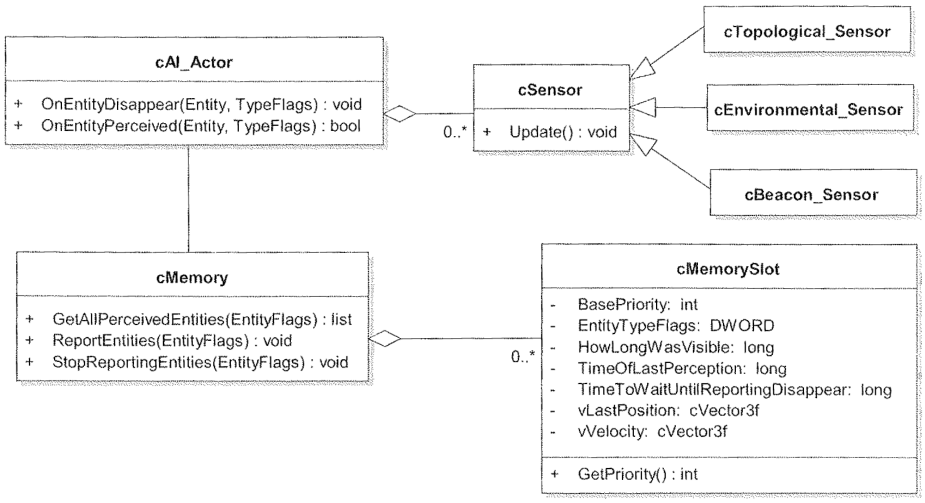

Figure 3.1: Class diagram showing the components of an agent [4, section 3.7].

also possible to define the sensory capabilities of an agent by setting a number of filter properties of the agent's sensor, which vary according to the data type (e.g. a beacon sensor has a attribute to specify the type of beacon it can detect).

If a sensor is updated it checks the connected data gatherers to get new information, after this the data is put into the agent's memory. Although memory doesn't play an integral role in most implementations, it does in this architecture and can be used to build more realistic agent behaviours. It stores all sensed data which is prioritized and can be retrieved by agents, but it is a short time memory system which means the priorities are decreasing until they fall under a threshold and are discarded.

3.3 RAIN indie

It is a freeware AI toolset developed by "Rival Theory" for the Unity3D engine and its the successor to the "RAIN one" toolset which was a purchased product. It is aimed to be a complete AI solution and offers techniques like a visual behaviour tree editor, path finding, or navigation meshes and also part of this toolset is a sensory system. The system consists of two major parts, aspects and sensors, to explain the workflow in short: First there is an aspect added to every game object that should be detected by the AI, and then a sensor is added to every agent that needs to sense the environment.

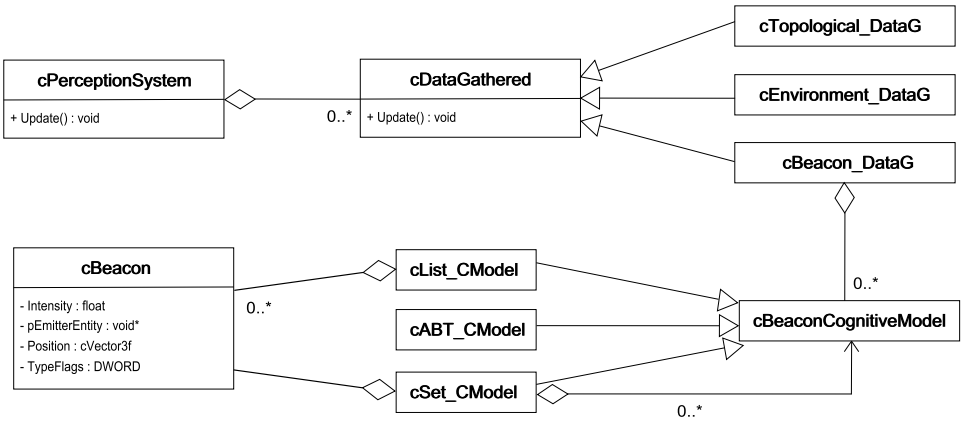

Figure 3.2: Class diagram of the generic perception system [4, section 3.7].

Aspects

Aspects are added to a Unity game object as a component and can be detected by sensors. If an aspect is added to a gameobject there are two separate components added:

- *Entity:* Entities are objects of interest that can be detected by sensors.
- *Decoration:* An entity has one or more decorations describing the associated sensation, the following default sensations exist: visual, tactile, auditory, olfactory, and taste.

Sensors

Sensors enable the AI to detect aspects and they utilize trigger colliders[2] to detect collisions by looking for intersections with game objects. When creating a sensor the sensation it should be able to detect can be chosen. There are three different types of sensors:

- BoxSensor,
- ExpandingBoxSensor,
- SphereSensor.

Each of the sensors has an option to do an additional line-of-sight raycast to check if the object is not behind an obstacle and can actually be detected. There are some other details like the ability to create custom aspects and sensors or the option to enable and disable a visualization of the raycast, which is quite useful for debugging purposes.

[2]In Unity3D colliders can be used to trigger events, in that case they are ignored by the physics engine.

3.4 Unified Agent Sensing Model

This model was proposed in the article "Designing a Realistic and Unified Agent Sensing Model" in Game Programming Gems 7 [1], and it focuses on how to augment the traditional agent sensing models. Until today the most common way to implement vision is by a view cone, while the popular option to realize hearing is by a radius check. These subjects will be explained in greater detail in a later section, so I have to anticipate a bit at this point and just describe the proposed possibilities to enhance them:

- Vision Model with ellipse.
- Augmenting discrete tests with a certainty percentage.
- Hearing Model with zones and path finding.

Although sensing models are still quite simplistic in most games, they are efficient, easy to implement and serve many gameplay mechanics reasonable well. But the biggest flaw of those methods is the fact that they use discrete tests and these lead to game objects that are either seen completely by the agent or not seen at all. Due to the shape of the view cone, agents can't see game objects right next to them, which can be either solved by using a second cone to be able to provide a broader field of view at close distance. Another solution would be to use an ellipse instead of a cone to model the field of view (see figure 3.3). As mentioned discrete tests lead to some behaviour

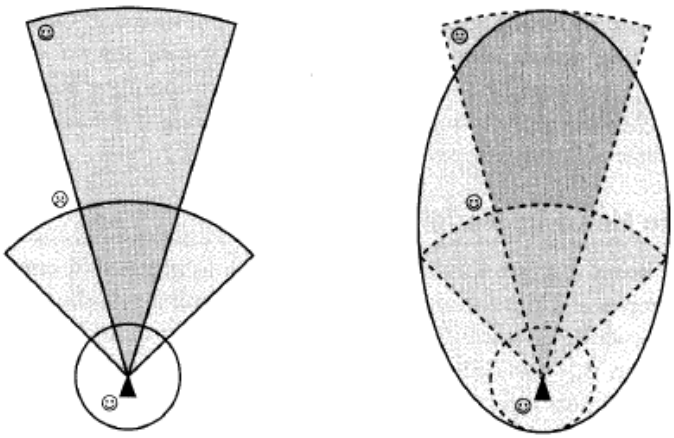

Figure 3.3: The left figure shows a vision system with an additional cone to create a larger area of vision, to better detect objects close by. The right figure illustrates the use of an ellipse to model vision [1, section 3.2].

that is recognized and exploited by players, like an total blind zone beyond the view distance of the sight cone. This can be improved by introducing granularity to sensing models, similar to awareness levels (see page 8), but more elaborate. The view cone is segmented into several areas and each of

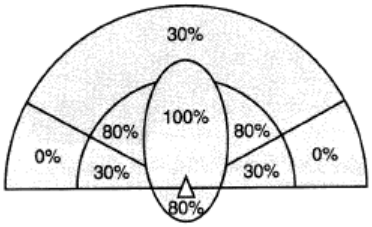

Figure 3.4: Certainty in vision as a collection of discrete tests to model human vision more accurately [1, section 3.2].

those has a certain percentage assigned, which describes the certainty, that an object entering this specific area, is identified. Additionally many other influences on the vision of the agent can be accounted for, like the movement of the object. Peripheral vision for example is very sensitive to movement, this can be incorporated by adding a score for the speed of the object. Looking at stealth games, elements like camouflage or hiding in shadows can be modelled with this approach too, even the profile of the player can be taken into account. When the player is in a dark corner he is more difficult to see for the agent and the identification percentage is decreased, if he is crouching it is decreased even further. The reaction of the agent depends on the percentage of certainty and is implemented by defining thresholds at which the AI executes particular actions. In the same way the hearing model can be improved to work more subtle, by augmenting the simple radius check that is typically done for sounds. The certainty percentage for a sound describes the falloff of the volume, affected by the distance the sound travelled. Walls pose the more difficult problem, because they should affect the certainty percentage as well. There are many different solutions, but each of them quite processor intensive. One option would be to do a line-of-sight check between the source and the listener and if it fails a path could be calculated to look if a root between them can be found. If one is found the distance could be used to determine the decrease of the volume. Another possibility is to add an additional description to the level, by dividing it into different zones. Sounds can only be heard when the source and listener are in the same or in adjacent zones (see figure 3.5).

Because this model allows for a number of compelling features it is especially suitable for stealth games and supports the implementation of gameplay elements this genre offers.

3.5 Sensory Information Tree

This section describes the sensory system implementation of the game “Thief: The Dark Project”, based on one of the first articles to be found when start-

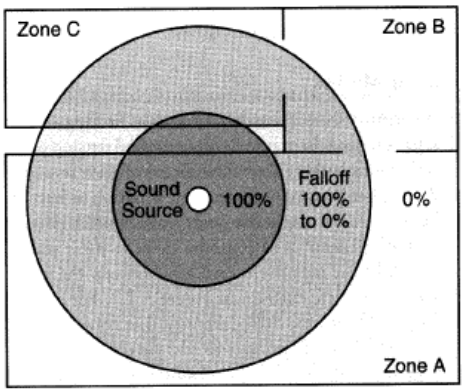

Figure 3.5: Illustrating a hearing model with intensity falloff and zones [1, section 3.2].

ing to research this field: "Building an AI Sensory System: Examining The Design of Thief: The Dark Project" by Tom Leonard [7]. At its core the system uses a view cone and raycast for the agents vision as well as a common hearing system.

The interesting part is how the senses are working, they are framed in discrete awareness levels (low, medium, high), which represents the AI's certainty of the perception of an game object. The awareness is stored in sense links, which associate an agent to another game object or to a position in the level (illustrated in figure 3.6).

Awareness Pulses

The process how sense links are created and maintained starts with the vision and hearing systems, their results are received as periodic pulses and stored as an awareness result in a sense link. There are three steps, to create, update or expire sense links:

- *Create:* The vision and hearing system output is saved in a sense link.
- *Update:* If the awareness pulse increases compared to the previous one, the awareness increases.
- *Expire:* In the case that the new pulse value is below the current result, the awareness decreases.

The sensory system output is a single awareness value for each game object that is of interest to the AI. This discrete states are the only part of the internal workings of the system the designers are exposed to, and because the output is also easy to understand, agent behaviours can be created without having to comprehend how the system computes those results. With sense links a great deal of stealth gameplay could been implemented in "Thief", like hiding in shadows or concealing bodies, and it provided the AI with a

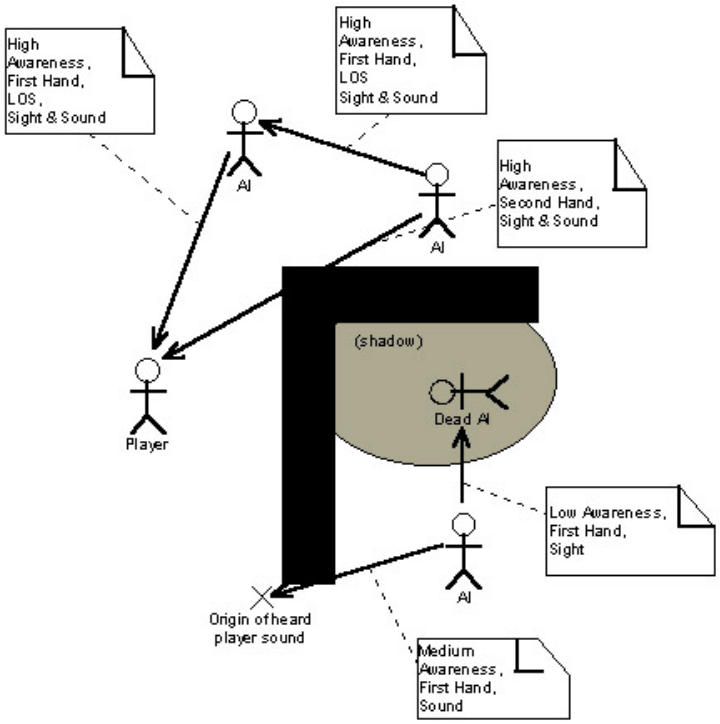

Figure 3.6: Sense Links [7].

range of environmental input that made the game more interesting.

3.6 Basic Design Rules for Sensory Systems

There are a few things that should be considered when building an sensory system, that will make it more reusable, faster and more scalable. The bulk of them are quite simple but nonetheless very effective optimization (the following rules in this section are based on the article “Sneaking Behind Thief’s AI: 14 Tricks to Steal for Your Game” [8]).

Focus on vision and hearing

This two senses already cover the better part of all gameplay elements and furthermore form the basis for most other senses that might be implemented later, even fictional senses. That said, vision is of upmost importance and should be developed first, since it is the most obvious sense players can tell pretty quickly if it is simulated poorly.

Optimize for Player Proximity and Importance

To ensure the performance of the system doesn't fall off, a simple level of detail logic should be implemented. That allows to reduce the need for sensory checks by a large amount, because the system doesn't have to check if an agent can see the player, if he isn't even near and a cheap distance check suffices. The same thing goes for the importance of perceived events, when the AI already spotted the player there is no need any more to still react to his footsteps.

Do the cheapest check first

As much events as possible should be filtered out with cheap checks, e.g., a distance check is cheap but a raycast is expensive. A reasonable order of checks for a basic vision system would be:

1. Distance check,
2. View Cone check,
3. Final visibility check with raycast.

If this simple culling technique is kept in mind, the performance of the system can already be increased.

Allow fantasy senses

The basic senses can be modified to create a magical aura which allows tougher enemies to have some kind of sixth sense, with which they can detect the player even if they don't see or hear him. Or the agents can be equipped with spells that enable them to enhance their senses for a short time.

Use discrete level of awareness

It is more intuitive to work with discrete values like low, medium or high awareness, instead of continuous ones. It doesn't really matter much what values the inner workings of the system are using, it suffices if they are converted into discrete values when creating the output the designers are exposed to.

Make view cones configurable

To allow them to be used to create a number of different agents that perceive differently, the key parameters of a view cone should be adjustable. For example to create elite guards that are more aware of their surroundings, or drunken ones that are less aware, or security cameras which do not have the same peripheral vision as humans.

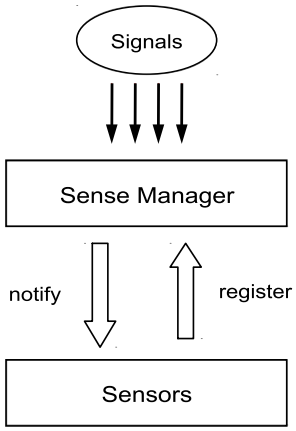

Figure 3.7: Basic functionality of a sense manager.

Context Sensitive Alertness

If the system allows the awareness level of the AI to be fed back into it, it's possible to modify the behaviour of the system dynamically. E.g. the senses of an agent could become more sharpened if he is alerted.

3.7 Region Sense Manager

This algorithm uses a sense manager as a central hub to administrate sensors and signals, it was described by Ian Millington in "Artificial Intelligence for Games". Agents have sensors they register with the sense manager, which describe their sensing abilities and their interests (game objects they want to sense), game objects that can be sensed have a signal added which is processed by the sense manager. A sensor is only notified of the signal, if he is interested in it and it passes all tests that check if the signal reaches the sensor (see figure 3.7), there is a simple example of the process is shown in figure 3.8. To sum up there are three main parts:

- *Sensors:* Models senses like vision and enables agents to detect game objects.
- *Signals:* An event that something happened in the level, it has a specific modality that describes how the signal is sent.
- *Sense Manager:* Administers sensors and signals, and notifies sensors of signals.

To determine if a sensor perceived a signal there are three distinct phases, the algorithm works through:

1. **Aggregation Phase:**
   If a signal is emitted it is registered by the sense manager, whereupon

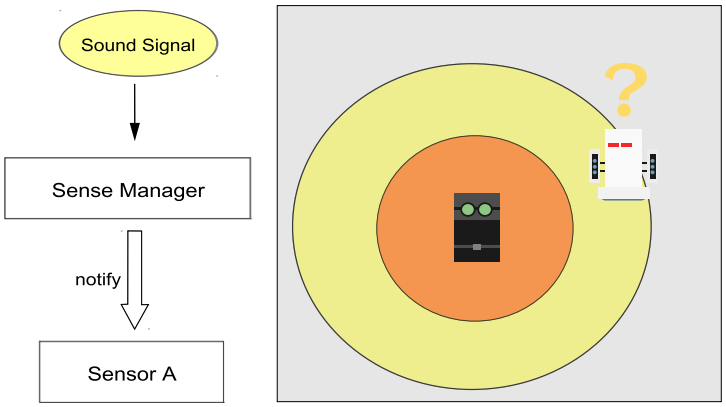

Figure 3.8: Illustrating an example for a sound signal that is received by the sense manager, which notifies the sensor of the agent.

all sensors are checked if they are within the maximum radius of the corresponding modality. This is in essence a first basic distance check to cull all sensors, that are out of range and can be disregarded.

2. **Testing Phase:**
   In this step all checks are conducted to determine if an agent actually can perceive the signal, starting with an intensity check which calculates the intensity of the signal when it's reaching the agent and compares it with a specified threshold. Additionally depending on the modality some extra checks are performed, e.g. view cone check for vision. If all tests pass the signal can be sensed and the sensor needs to be notified.
3. **Notification Phase:**
   The sense manager checks if there are any sensors to be notified of signals, if so a notification is delivered.

The regional sense manager is essentially a standard sensory system with reasonable culling techniques coupled with an event manager. Although the algorithm is simple, it's nevertheless fast and powerful. Another benefit would be the fact that debugging is made easier because all checking is done at a central point. That makes it easier to reconstruct an intricate order of decisions made by an agent.

And it can be extended to at least give the impression of avoiding its limitations. A weakness of this implementation is for example the manner sound is dealt with, because it's treated strictly as a spherical area of influence without taking level geometry into account. This can lead to some noticeable artefacts, but this is a common problem and there are a couple of solutions to improve the sound system, which can be used depending on the requirements of the gameplay and the demands regarding the performance.

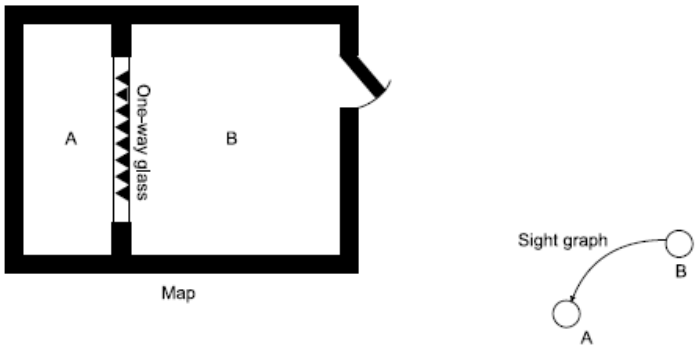

Figure 3.9: One node for each room and one edge connecting room B to room A, because of the one-way glass the sight modality only can pass in one direction [2, section 10].

3.8 Finite Element Model Sense Manager

An attempt to model senses more accurately is the use of a finite element model (FEM), which describes a technique to split up a large problem into many small subdomains, it has been proposed by Ian Millington in "Artificial Intelligence for Games" as an alternative to the regional sense manager. The complex problem is tackled by solving the subdomains and create an approximated solution which is sufficient enough, in the case of perception the subdomains are single parts the level is divided into. That means the sense manager has an additional piece of information to work with which aims at improving the AI perception, the sense graph.

Sense Graph

The game level is transformed into a directed acyclic graph (DAG), each node represents an area where a signal can pass unobstructed and the connections (edges) of the graph describe if a modality can pass between two areas (see figure 3.9). This means to determine between which areas a modality can pass, a graph is traversed, which is a simpler task than doing an elaborate and exact simulation.

At first sight there might be some confusion caused by the fact that it doesn't automatically mean, that by the connection of two nodes in the sense graph, everything in room A can be seen from every position of room B. Some locations of room A (the destination node in this case) cannot be seen from several locations in room B (illustrated in figure 3.10). Those cases are handled by the sight tests of the system just like in every other sensory system implementation. For example when a line of sight raycast fails, the target can't be detected, even if the nodes are connected, if they are not then the tests won't be performed. The algorithm works the same way as

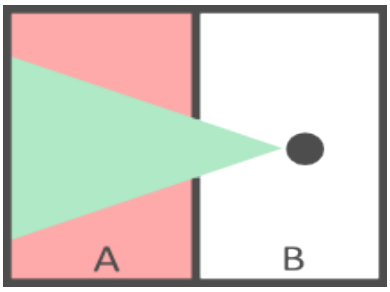

Figure 3.10: Line of sight issue with a pair of nodes.

the regional sense manager (section 3.7) with three phases: aggregation, testing and notification. Additionally the algorithm has the sense graph to work with, but first sensors and signals need to be incorporated into the nodes. To further illustrate how the algorithm operates, the handling of a sight signal will be described:

1. **Aggregation Phase:**
   The list of potential sensors consists of all sensors that are in the same node as the sight signal, as well as all sensors in nodes that have a connection to that node.
2. **Testing Phase:**
   As already mentioned before the test work the same as in the regional sense manager implementation for example.
3. **Notification Phase:**
   The notifications are delivered by the sense manager.

Concluding one of the weaknesses of this approach is outlined, the complexity. Because the algorithm is more complex it is more time consuming to implement, and if the gameplay doesn't specifically requires an elaborate sense simulation a simpler solution should be considered.

Chapter 4

Implementation

The basic approach was the implementation of a sensory system based on the "Regional Sense Manager" for the Unity3D engine to aid the development of stealth games. The system is built with editor scripts and thereby seamlessly integrated in the user interface and workflow of the engine and without the need of an installation, it's simply dragged and dropped into a specific subdirectory of a game project and instantly ready to use. Which is one of the reasons to use Unity for this project, it is possible to create custom tools inside the engine.

4.1 Requirements

There were some self-imposed requirements defined to guide the development:

Target Genre

The type of game that usually needs a sophisticated sensory system is a stealth game, because they use senses as a gameplay focus and that requires more elaborate sensory systems which cover more than the basics of perceiving, so the system is build keeping the requirements of this genre in mind.

Scalability

The system should be subservient to the game design and allow the construction of small levels with a low number of agents as well as big areas with a large amount of agents without causing performance issues.

Reusability

The implementation shouldn't be constricted to one specific project, but should be usable for many different types of games and it should offer the possibility to apply it's features to a broad spectrum of problems, so that it functions as an basis to work with, that can be utilized in new projects right from the spot, and without the need to be extended.

Level of Detail Logic

On one hand there should be the possibility to configure different levels of details for the sensors of the agents, to be able to implement more novel gameplay, and on the other hand the sensors need a level of detail logic for perception, to cull signals with the least amount of checks.

Usability

The integration of the system in the engine should be done while keeping the fact in mind, that the user interface has to be at least somewhat intuitive to use and the code needs to be clearly written to be extendable and maintainable.

4.2 Unity Sensory System Implementation

The Regional Sense Manager is integrated as a custom tool inside Unity through Editor Scripts written in C-Sharp, to extend the functionality of Unity to support the use of a sensory system. The system follows the workflow of the Unity Engine and enables the user to add sensors and signals as components to game objects, which can be configured in the inspector (see figure 4.3). There is no need to write any additional code for the AI perception, only the reaction of the agents needs to be coded.

4.2.1 Component Overview

The system consists of a manageable amount of components to realize AI perception:

Modality

A sensory modality characterizes the type of a signal and the sense organ that can perceive it (e.g. sound, smell).

Sensor

Sensors detect specified modalities and represent the eyes and ears of the AI.

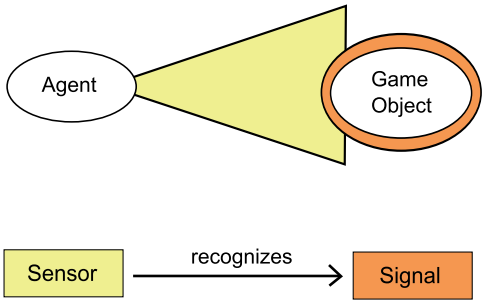

Figure 4.1: Illustration of the basic sight perception functionality, a sight sensor is added to an AI agent and a sight signal to a game object. Thereupon the sensor of the agent can recognize the signal of the game object.

Signal

A signal is a message that something of interest to the AI, happened in the level.

Regional Sense Manager

The sense manager administers all signals, does all the necessary tests and notifies sensors if needed.

Notification Queue

This queue stores notifications which consist of the emitted signal and the sensor to notify.

The class diagram in figure 4.2 gives an overview of the system, illustrating it as a whole, while figure 4.1 gives a simple illustrative example of its functioning. In the following the implementation of each component is described in more detail.

4.2.2 Sensor

There are two types of sensors implemented, the sight sensor and the sound sensor while a touch sensor was not realized because touching comes down to collision detection, which is already supported by most engines and easily incorporated in a sensory system. Whereas seeing and hearing on one hand are more complex to implement efficiently and on the other hand almost every other sensation can be actualized by altering the implementation of those two senses, if the sensory system needs to be extended. The sensor has one distinct parameter:

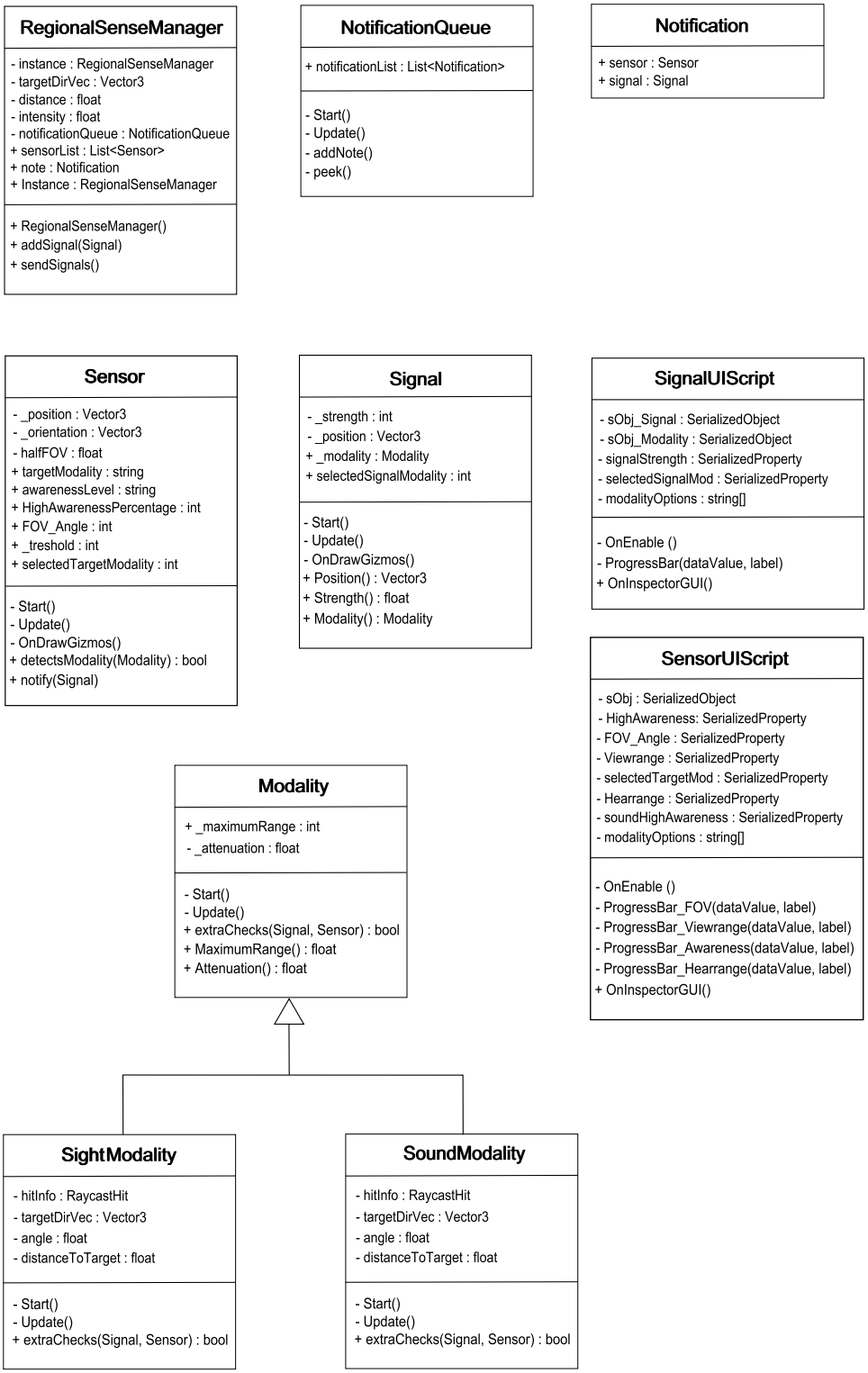

Figure 4.2: A class diagram describing the sensory system.

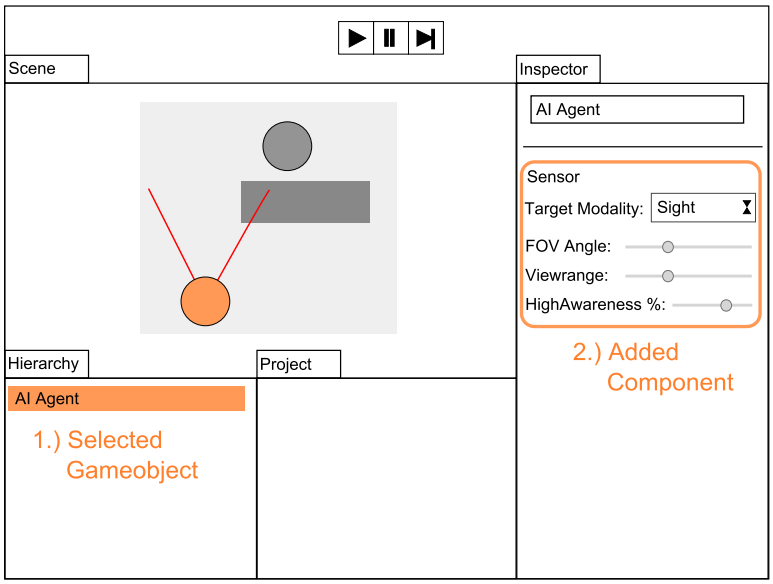

Figure 4.3: A sketch of the Unity UI showing the intended workflow of the sensory system implementation.

- *Target Modality:* The type of signal the sensor should detect (visual or audio signals).

Depending on what target modality is chosen, a different set of additional settings is shown in the UI, to further adjust the sensor. A sensor is composed of three different parts:

- *Sensor:* The "Sensor.cs" script is the structural basis, it contains the variables, provides the modality selection and is responsible to display the visual aid.
- *Modality:* In the "SightModality.cs" respectively the "SoundModality.cs" script the actual detection code is executed.
- *UI:* In the "SensorUIScript.cs" the user interface for the sensor is pieced together.

Sight Sensor

In essence this sensor adds a view cone to an agent and offers some settings that can be adjusted in the user interface. Furthermore it provides a visual aid in the scene view[1] to support debugging. There are four parameters that can be adjusted by the user:

- *FOV Angle:* The angle of the field of view.

[1]The scene view is the part of the Unity interface where all game objects are positioned, it doesn't represent the final game the player is seeing.

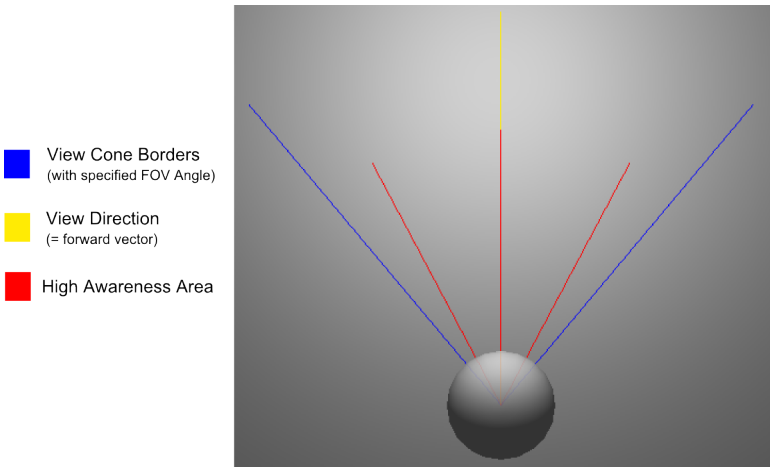

Figure 4.4: Screenshot of an agent with a sight sensor, showing its visual aid in the scene view.

- *View Range:* The length of the view cone, or how far the AI can see.
- *High Awareness Percentage:* How big is the area of the view field that constitutes the awareness level "high", as a percentage of the entire view field.

The "Sensor.cs" class consist of three important functions:

- *detectsModality(Modality):* Returns true if the sensor can detect the given modality, the argument is the modality that should be detected.
- *notify(Signal):* Notifies the sensor of the signal if it can be perceived.
- *OnDrawGizmos():* This function is responsible for the visualization of the view direction, cone borders and raycast.

The visualization of the view cone borders is done with quaternions[2], which are used by Unity internally to represent rotations and allow a shorter and slightly more elegant implementation (program 4.1), than calculations with Euler angles (illustrated in figure 4.4).

Sound Sensor

This sensor works with a spherical area of perception, describing the hearing range of an agent. A sound sensor is created by changing the target modality setting of the sensor to "Sound", and if a game object emits a sound signal within this area, the agent can perceive it. Same as the sight sensor a visual aid in the scene view is provided, while there are two parameters to adjust

[2]Quaternions are based on complex numbers and are used for calculating three-dimensional rotations, additionally they can be used as an alternative to Euler angles and rotation matrices.

Program 4.1: Implementation of the view cone borders with quaternions.

```
1 halfFOV = FOV_Angle / 2;
2 Quaternion leftRayRotation = Quaternion.AngleAxis( -halfFOV, Vector3.up
      );
3 Quaternion rightRayRotation = Quaternion.AngleAxis( halfFOV, Vector3.up
      );
4 Vector3 leftRayDirection = leftRayRotation * transform.forward;
5 Vector3 rightRayDirection = rightRayRotation * transform.forward;
6 // Draw Cone Borders:
7 Gizmos.color = Color.red;
8 Gizmos.DrawRay( transform.position, leftRayDirection * viewRange );
9 Gizmos.DrawRay( transform.position, rightRayDirection * viewRange );
```

this sensor:

- *Max Hear Range:* The radius of the hearing sphere.
- *High Awareness Percentage:* How big is the area of the hearing sphere that constitutes the awareness level "high", as a percentage of the entire sphere.

The perception is done by comparing the distance to the target, to the threshold of the sound sensor. With this sensor the AI can detect every game object that is emitting a sound signal in the level. Although the actual reaction of an agent isn't part of the sensory system itself, the system offers the possibility to set up and adjust different awareness levels (see figure 4.5) to support the development of more novel gameplay by implementing the perception with possible granularity in mind. With this sensor the behaviour of an agent can be changed by simply using sliders in the UI and this already can alter the gameplay tremendously.

4.2.3 Modality

A modality describes the type of the signal and the most important function is the execution of modality specific checks to determine if a signal can be perceived, it is comprised of the parameters and functions that are needed to check for signal perception. The script "Modality.cs" works as an interface which is implemented by the "SightModality.cs" and "SoundModality.cs" scripts, by realizing the modality specific checks in the extraChecks() function:

- *extraChecks(Signal, Sensor):* This function is responsible for the actual perception of signals and it returns true or false, if the passed signal can be perceived by the passed sensor or not (program 4.2).

The user is barely exposed to modality scripts through the user interface, because they work in the background. Only through the "Target Modality"

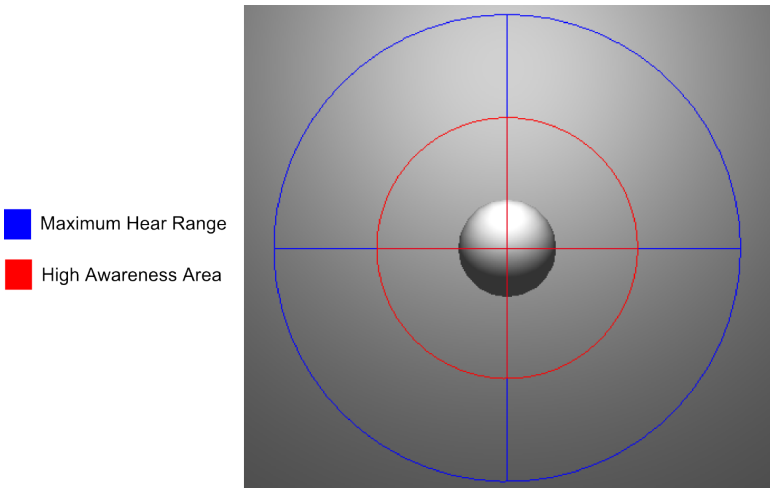

Figure 4.5: An agent with a sound sensor, consisting of two awareness levels.

selection the modality principle is pointed out, but its functionality is hidden and doesn't need additional adjustments to work. It is a crucial part of the sensory system, because the perception code is located here, encapsulated in the extraChecks() function. For a new modality (a fantasy modality for instance) the modality interface needs to be implemented and the new modality specific checks have to be coded in this function.

Sight Modality

The sight perception implementation is done with a standard technique of comparing two angles between the target direction vector and the forward vector of the agent (see program 4.2). The FOV angle describes how far to the agent can see to the left and to the right of his current forward vector, in other words this is the angle between the two borders of the cone. A FOV of 90 degrees would mean that the agent can see objects that are less than 45 degrees away from his forward vector (see figure 2.3). Of course this doesn't account for the view distance, but this is handled elsewhere by the sense manager. The next step of the visibility determination is to do a raycast from the agent to the game object, to check if any obstacles between them are blocking the sight. If this test checks out, the awareness level is determined which is a more detailed description of the alert state of the agent. The distinction between the two awareness levels works with a second smaller view cone inside the standard cone, whose size is defined by a percentage value specified in the sensor settings with the "High Awareness Percentage" slider. Depending in what area of the view cone the game object is located the awareness level is set accordingly and the reaction of the agent can be implemented with more detail if those levels are incorporated.

Program 4.2: The sight perception is implemented by comparing two angles between the target direction vector to the forward vector of the agent (see row 15), as illustrated in figure 2.3.

```
1  // Executes modality specific checks in the testing phase
2  public override bool extraChecks(Signal signal, Sensor sensor){
3
4    // calculate the vector agent->target (targetDirVec), then get the angle between
5    // it and the enemy forward direction, if it's less than half (!) the angle of
6    // view, the target is visible:
7
8    // vector agent -> target (=signal):
9    targetDirVec = signal.transform.position - sensor.transform.position;
10   // angle between targetDirVec ande AI forward vector:
11   angle = Vector3.Angle(targetDirVec, sensor.gameObject.transform.
       forward);
12   distanceToTarget = targetDirVec.magnitude;
13
14   // if target is visible, do raycast:
15   if(angle < sensor.FOV_Angle/2 && distanceToTarget <= sensor._threshold
       )
16   {
17     // if raycast hits target (and not e.g. a obstacle in between:
18
19          // determine and set awareness level:
20          ...
21   }
22   ...
23 }
```

Sound Modality

The sound implementation is quite simple because it works with spheres and doesn't account for obstacles like walls or signal attenuation, although it was considered and the modality class has the appropriate variable to support it, for a potential expansion. The check compares the length of the target direction vector, which is determined by calling the magnitude function for that vector (see program 4.3 in row 6) which returns the length of the vector, subtracted by the signal strength (= radius of the signal) to the threshold of the sound sensor (=radius of the sensor). The algorithm is illustrated in figure 2.6 for better understanding and simply put the sound sensor can perceive a sound signal when their respective spheres overlap.

Like the sight modality the sound modality offers awareness levels, which are also described as a percentage of the sound sphere (figure 4.5) and depending on that level the reaction of the agent can be implemented. As mentioned this solution doesn't account for obstacles, to incorporate this ray traces with the environment can be performed. But this is a simpli-

Program 4.3: For the sound perception the distance to the target is compared to the threshold of the sound sensor (additionally the strength of the sound signal is considered), see figure 2.6.

```
1 // Executes modality specific checks in the testing phase
2 public override bool extraChecks(Signal signal, Sensor sensor){
3
4   // vector agent -> target (=signal):
5   targetDirVec = signal.transform.position - sensor.transform.position;
6   distanceToTarget = targetDirVec.magnitude;
7
8     //Check if the sensor is within the maximumRange of the sound signal:
9     if(distanceToTarget-signal.Strength <= sensor._soundthreshold){
10
11          // determine and set awareness level:
12          ...
13  }
14  ...
15 }
```

fied solution, due to the fact that a sound signal shouldn't be disregarded because of an obstacle, but rather just damped. Additionally the type of obstacle need to be considered and if the sensor and the signal are in the same room. Because of this the culling of sound signals would require a more creative and elaborate solution.

4.2.4 Signal

Signals are constituting a message that something in the level has happened and they consist of three important pieces of data: the modality, the signal strength and the position. If a signal is added to a game object, it enables the AI to perceive this object with the appropriate sensor, but in order to do that the target modalities need to accord. Like a sensor, a signal has a certain modality, more specifically each signal has a particular modality and a sensor must be set up with the same target modality to be able to perceive the signal (see figure 4.6). While the strength value can be changed to alter the maximum distance the signal travels which is shown by a visual aid in the scene view for the sound signal. Overall the implementation of the signal class is pretty straight forward because its main task is to encapsulate events of interest to the perception mechanic and work as a corner stone for the sense manager.

4.2.5 Component Interaction Part 1

Until now three components of the sensory system has been described: sensors, signals and modalities. The process how these parts are used to enable

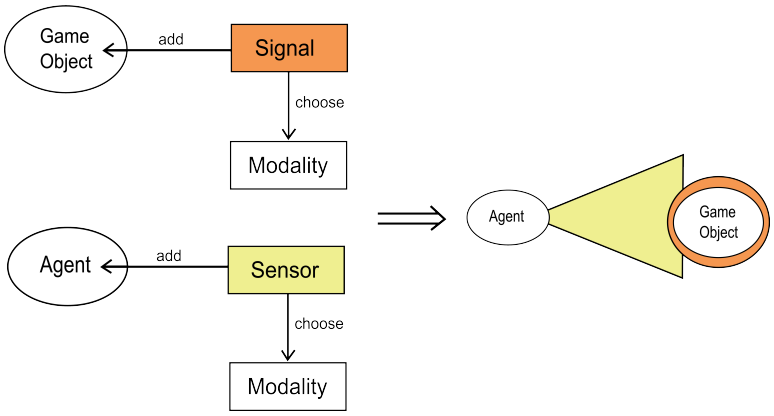

Figure 4.6: An illustration how a part of the component interaction works, a sight signal is added to a game object and a sight sensor to an agent.

the AI to perceive its environment works as followed:

- A sensor is added to an agent.
- The target modality of the sensor is specified.
- A signal is added to a gameobject.
- The target modality of the signal is specified.

After these components are added and set up, the agent with the added sensor can see or hear the game object with the added signal (see figure 4.6). With this system the AI can perceive the environment sufficiently and the reaction to perceived game objects is based on the type of the object, without having to enlarge the sensory system itself. Meaning that an agent can see every game object with an attached sight signal, and what he does when he is seeing it depends solely on its type. This approach separates the perception from the reaction and keeps the sensory system at a small size, which makes it easier to reuse.

4.2.6 Notification Queue

The notification queue contains a list of notifications that need to be passed to sensors, to notify them that they can perceive a signal (which is handled by the sense manager described in the next section). The class consists of three parts:

- *notificationList:* A list of notifications waiting to be send out to sensors.
- *addNote(Notification):* Adds a note to the queue (notes have the type Notification, defined by a struct) if a signal passes all checks and can be perceived by a sensor.
- *peek():* Is used when notifications are flushed from the queue, to notify a sensor about a signal.

A note is composed of a signal that passed all checks and a sensor that can perceive that signal and it is only added to the queue if the signal has passed all checks. The notification queue is used by the regional sense manager to administrate signals and the sensors which can actually perceive them.

4.2.7 Regional Sense Manager

All sensors and signals are administered by a sense manager, which is responsible for all checks and notifications. The algorithm is separated in three different phases (see chapter 3.7):

1. *Aggregation Phase:* All sensors are found.
2. *Testing Phase:* Perception checks are performed.
3. *Notification Phase:* Sensors are notified about perceived signals.

First a singleton pattern[3] is used to create a sense manager, to make sure that there is only one running because there is only one needed to manage the sensors and signals (program 4.4). To keep track of sensors and signals the sense manager uses two lists:

- *sensorList:* If a sensor is created it's added to this list (in the Start() function of the sensor).
- *NotificationQueue:* This queue holds notifications that include the sensors to notify and the corresponding signals.

The sense manager has two main purposes, the execution of perception checks for all agents at a centralized location and the notification of sensors if those checks are successful. This is implemented in the following two functions:

- *addSignal(Signal):* Checks all signals and creates a notification if one can be perceived by a sensor, it implements the biggest part of the three phases the sense manager algorithm consists of.
- *sendSignals():* This function notifies a sensor of a perceived signal, by using the peek() function of the notification queue to get the sensor which needs to be notified of a signal. Then the notify(Signal) function of that sensor is called to inform it about the signal.

The addSignal(Signal) function is shown in program 4.5 and the first step is the aggregation phase, where it iterates over all sensors that have registered themselves in the sense managers sensorList. For every sensor in that list, the testing phase checks if one of them can perceive a signal. The checks start with the cheapest one and get more expensive, due to the fact that the cheap checks are done at the beginning a lot of sensors can be culled, without much computing effort and thereby the performance can be increased easily. There are four different checks executed:

[3]The singleton pattern is a simple software design pattern, used to restrict the instantiation of a class to only one object, to ensure that not multiple instances are created.

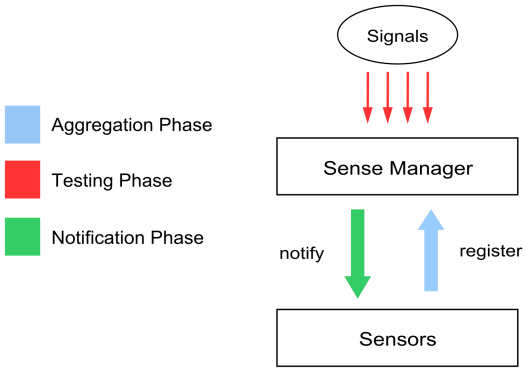

Figure 4.7: The different phases of the sense manager.

- *Modalitycheck:* Tests if it's the appropriate sensor for the given signal (e.g. a sight sensor can't detect a sound signal).
- *Rangecheck:* A level of detail logic, that verifies if the signal is within the maximum detection radius of the sensor (e.g. if it's within the maximum viewing distance of an agent).
- *thresholdcheck:* If signal attenuation is used a intensity value is computed, to test if the signal strength when it's reaches the agent is still above the threshold value to be perceived.
- *Modality specific checks:* Depending on the signal modality the extraChecks() function of the appropriate modality type is called to execute modality specific checks (e.g. the view cone logic for the sight modality and the raycast).

If a signal passes all four tests a notification is created and added to the queue, after that the last step is the notification phase where sendSignal() is called to actually notify sensors of the signals. This concludes the perception process and the primary task of the sensory system. At this point sensors that had been notified of perceived signals can execute their reaction logic.

This implementation is scalable and can be used for games with a small number of agents as well as for projects that need a higher amount of agents. Additionally the fact that the perception is managed at one central point, makes the system simpler to debug. While this approach isn't the most complex one, simple algorithms have proven to produce fewer artefacts that are noticed by players, which aid their purpose: to create the illusion of intelligence.

4.2.8 Component Interaction Part 2

The sense manager is the centre of the system and is responsible for keeping track of all the agent's sensors and to check if a signal emitted by a game

Program 4.4: The sense manager singleton.

```
1 public static RegionalSenseManager Instance{
2
3   get{
4     if(instance == null)
5       new RegionalSenseManager();
6
7     return instance;
8   }
9 }
```

object with a specific modality can be perceived by one of them. Provided that there is an instance of the regional sense manager running, this process starts with the creation of a sensor, which is adding himself to the sensor list of the sense manager. If a signal is created (which can happen in different intervals from only one time to continuous signals) it calls the addSignal function of the sense manager.

This leads to the actual perception process, where the algorithm iterates over all elements in the sensor list and tests for each sensor in that list, if it can perceive the signal which is the parameter of addSignal(). That is called the testing phase that conducts several different tests, starting with the cheapest one and culls the sensors if one test fails. If a sensor passes all tests a notification is created and added to the notification queue.

In the last step called the notification phase, sendSignal() is called to actually send the items saved in the notification queue. With this the sensor has been notified that he perceived a signal and the task of the sensory system is finished (illustrated in figure 4.8).

4.2.9 User Interface

There are two different options to build a custom user interface in Unity:

- *Editor Windows:* Used to create a custom window inside Unity, which needs to be opend with a new menu item located above the toolbar on top of the screen.
- *Inspector Windows:* Create custom controls in the game object inspector for frequently used components.

For this project the editor window option was chosen because the number of required UI elements is quite low, which means the size of an inspector window suffices while the benefit of an increased work speed can be utilized. Because agents and game objects are configured by the inspector window, it makes sense to incorporate the configuration of sensors and signals in that same work flow and not hiding it away in a separate editor window, which

Program 4.5: The addSignal() function is the heart of the system, it is responsible for the aggregation and testing phase, introduces signals into the game and creates the notifications for the sensors.

```
1  public void addSignal(Signal signal){
2
3    // AGGREGATION PHASE
4    // ————————————————————————————————
5
6    // Iterate over all sensors
7    for(int i = 0; i < sensorList.Count; i++){
8
9      // TESTING PHASE
10     // ————————————————————————————————
11
12     // Modalitycheck
13     if(sensorList[i].detectsModality(signal._modality)==true){
14
15       // Range check
16       targetDirVec = signal.transform.position - sensorList[i].transform
       .position;
17       distance = targetDirVec.magnitude;
18
19       if(signal._modality.MaximumRange < distance){
20
21         // threshold check
22         intensity = signal.Strength * Mathf.Pow(signal._modality.
       Attenuation, this.distance);
23
24         if(intensity < sensorList[i].threshold){
25
26           // Modality specific checks
27           if(signal._modality.extraChecks(signal, sensorList[i]) == true
       ){
28             // Create notification and add it to queue
29             Notification note = new Notification();
30             note.sensor = sensorList[i];
31             note.signal = signal;
32             notificationQueue.addNote(note);
33
34             this.sendSignals();
35           }
36         }
37       }
38     }
39   }
40 }
```

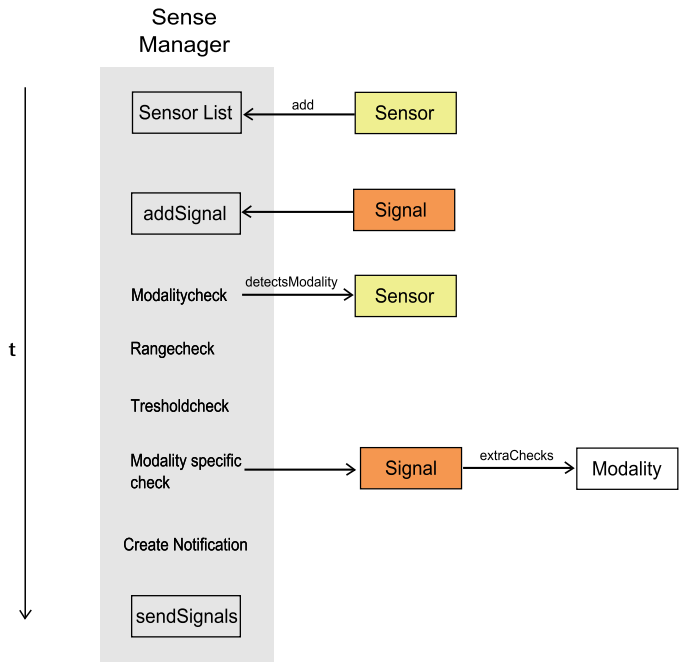

Figure 4.8: Abstract illustration of the component interaction during the perception process.

has to be opened with an additional menu item. At the same time the size of this editor window is not needed at this point to display the few controls.

Two UI scripts have been created, which are responsible for the UI of the sensor and signal components in the inspector window, both of them make use of serialized objects and properties.

Serialized objects and properties

These are needed for the custom inspector to be able to access the values from the script the UI is built for. For example, if a UI script has a slider to adjust the view range value this value must be handed from the UI script where this value is adjusted, to the script where this value is used to actually calculate the perception, the programs 4.6 and 4.7 are showing an example of this process.

Sensor UI

The UI script creates a popup selection field to select the target modality and depending on that selection a number of different sliders are displayed to adjust the values of the class the custom UI is built for. Along with those progress bars are displayed to give a better indication of the value in relation to its maximum (see figure 4.9).

Program 4.6: A example UI script to build a custom inspector window for a class to adjust the view range value with a slider.

```
1 [CustomEditor(typeof(Sensor))]
2
3 public class SensorUIScript : Editor {
4
5   SerializedObject sObj;
6   SerializedProperty Viewrange;
7
8   void OnEnable () {
9
10    // Setup the SerializedProperties (from the Sensor class Serialized Fields)
11    // ————————————————————————————————
12    // Opens the reference in row 1, to the class you are building an editor for:
13    sObj = new SerializedObject(target);
14    // Access a property from this class:
15    Viewrange = sObj.FindProperty("_threshold");
16  }
17
18  public override void OnInspectorGUI(){
19
20    // Update the serializedProperty
21    serializedObject.Update ();
22
23    // Create the slider to adjust the view range
24    EditorGUILayout.IntSlider(Viewrange, 1, 2000, new GUIContent("
      Viewrange:"));
25
26    // Apply changes to the serializedProperty
27    sObj.ApplyModifiedProperties ();
28  }
29 }
```

Program 4.7: A example class using a custom UI script, all variables declared as serialized fields can be accessed by the UI script (see program 4.6) and due to the update of the serialized property the values changes in this class too.

```
1 [Serializable]
2 public class Sensor : MonoBehaviour {
3
4   [SerializeField]
5   public int _threshold = 400;     // = viewRange, how far the AI can see
6
7   ...
8 }
```

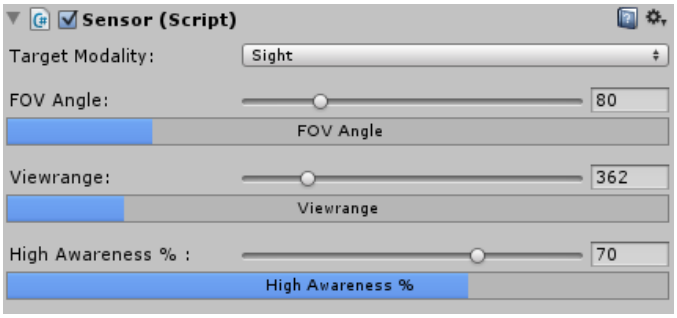

Figure 4.9: The custom inspector of the sensor with the controls for the selected target modality "Sight", created by the sensor UI script.

To create a custom inspector the function OnInspectorGUI() needs to be implemented, therein the actual layout is defined (program 4.6). With the UI the workflow of creating and adjusting a sensor is faster because the values can be adjusted with sliders, while paying attention to their visual representation in the scene view. In contrast to typing them into a text field and estimate the effects.

Signal UI

The script creating the custom inspector for signals works the same as the sensor UI script (figure 4.10). The implementation of the sight modality is worth mentioning because there are no adjustable parameters for it, the perception works just with the position of the signal.

Note on script compilation

The UI scripts are using the UnityEditor namespace because the Editor class is in it and this is the base class needed to derive custom Editors from. Due to that fact the script compilation order of the Unity engine has to be considered. Unity performs the script compilation in 4 steps and one important factor is that the scripts from one group can access scripts from previous groups, but not the other way around.

1. "Standard Assets", "Pro Standard Assets" or "Plugins".
2. "Standard Assets/Editor", "Pro Standard Assets/Editor" or "Plugins/Editor".
3. All other scripts outside "Editor".
4. All scripts in "Editor".

The sensory system scripts are located in a folder inside "Standard Assets" and scripts in this folder can't access scripts outside that folder. The UI scripts are placed in the "Editor" folder to use the UntityEditor namespace

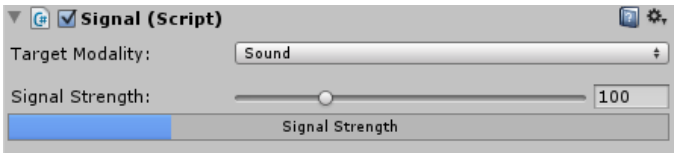

Figure 4.10: The custom inspector of the signal with the controls for the selected target modality "Sound", created by the signal UI script.

and because scripts in this folder can access all scripts from previously compiled groups. It is crucial that the UI scripts are placed in the right location otherwise they can't work because they simply cannot access the scripts they create a custom UI for.

4.3 Result

The implementation result is a sensory system for the Unity engine that can be used to equip the NPCs and the objects in a game with sensors and signals, to enable the agents to perceive the game world without further effort. The system works right from the spot and can be used without having to be familiar with it because it uses the standard workflow of Unity, sensors and signals are just added as components to game objects.

4.3.1 Setup

The setup of the system is clear and brief, it just has to be copied and pasted into a Unity project:

1. The folder "Sensory System" has to be copied into the Assets folder.
2. The scripts "SensorUIScript" and "SignalUIScript" has to be copied into the Assets/Editor folder.

For the reason the system is separated this way, refer to the note on script compilation on page 42 in section 4.2.9. When the system is included in a Unity project (see figure 4.11) it is ready to go and can be used.

4.3.2 Application

To equip an agent with a sensor, the sensor script has to be added by selecting the agent and then adding the script by drag drop or with the "Add Component" menu. Afterwards the sensor script appears in the inspector of the selected agent and can be adjusted. At this point the perception does works already and the AI can see and hear the game world without any additional work.

The user interface and possible adjustments has been described in detail already, the outcome of this is shown in figure 4.12. First off please note that

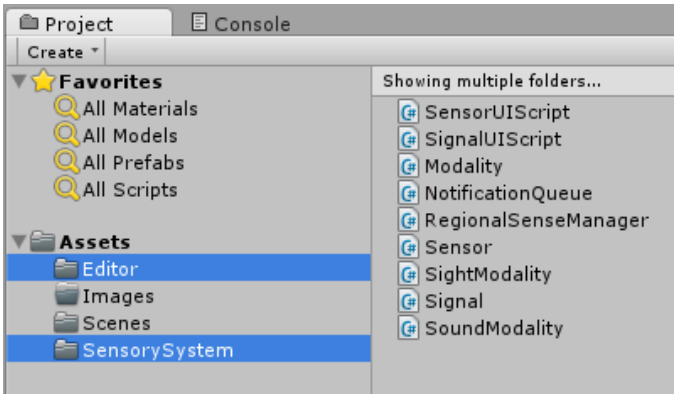

Figure 4.11: The sensory system included in the Asset directory of a Unity project, listing all scripts of the system.

the screenshot shows a test environment which is build for this exact purpose only, the green sphere represents an agent and the blue sphere the player (that can be controlled). The colour of the agent indicates his awareness level (green for idle, yellow for suspicious and red for alert) and serves as a simple AI reaction to the perception for testing purposes.

In the left picture the player is behind a wall and therefore the raycast that is executed by the agent because he is within the view cone fails, as a result the color of the agent is still green, indicating that the agent can't see the player. If the player moves and is no longer hidden behind the wall, the agent can see him and changes the color to yellow, indicating he is now suspicious. If the player would move even closer to the agent and enters the red view cone, the agent would change the colour to red, because the player is inside the high awareness area of the sight sensor.

Sound sensors and signals work similarly except they don't account for obstacles and it is possible to add more than one type of sensor or signal to a game object (figure 4.13). Many types of agents will need to be able to see and hear the player at the same time, for that they just need to have two sensors set up with different target modalities, while the game object in return needs two signals. If some other kind of fictional modality is needed for project, the system can be extended by modifying the given sight and sound classes without having to change the underlying sense manager code. That allows for the quick and easy incorporation of new senses that are still managed at a central point.

There is one factor left that has to be considered when working with a sensory system, the point in time at which a signal is actually send. A sight signal that is send from the player could be send in a update function, but the sound that the footsteps of the player make has to be linked to his movement. The same goes for many more game events that trigger a sound,

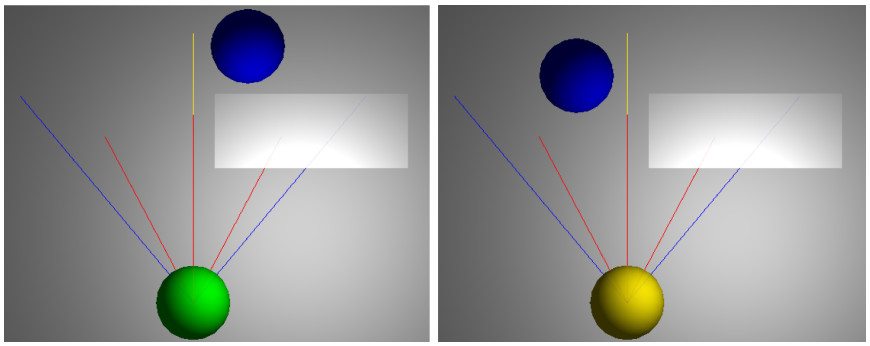

Figure 4.12: Screenshots of a test environment with an agent and a player (showing the scene view of Unity).

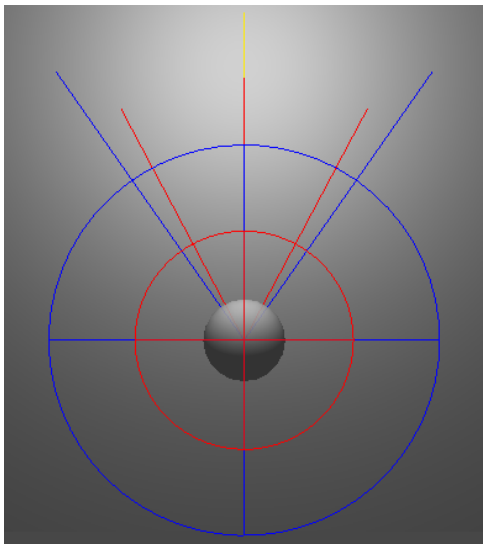

Figure 4.13: Agent equipped with both a sight and a sound sensor.

they need to send a sound signal too, assuming the event is relevant to the AI. Although the sensory system is set up quickly the sending of signals has to be incorporated into the rest of the game code, which takes time.

4.4 FEM Implemenation Proposal

An alternative to the regional sense manager implementation is the finite element model approach, but it has some distinct disadvantages which makes it arguable hard to justify this approach for many games. One of them would be the higher implementation effort, one reason for it is the fact that the FEM sense manager works with a sense graph. And the creation of that

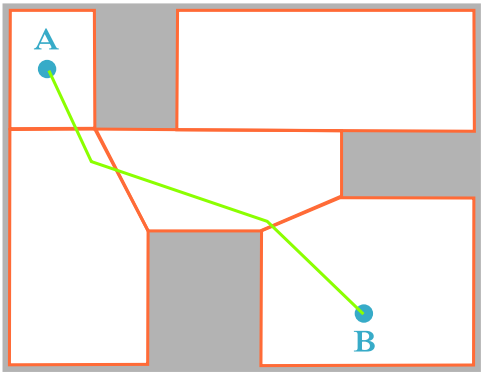

Figure 4.14: Example for a navigation mesh and the resulting AI movement.

graph is affiliated with a high effort, because to get the source data additional tool support is needed. There has to be a way to describe the parts of the level as nodes as well as the edges connecting them, and that in a quick and intuitive manner because the creation of the graph shouldn't already need to much time and effort, it rather should be kept as minimal as possible to battle the big disadvantage of this model, the implementation costs.

One possible option to tackle this issue, is to make use of a feature that is offered by many 3D Game Engines to reduce the amount of additional tools that needs to be implemented and at the same time ensure that the usage is intuitive because it builds on an already widely-known system, the navigation mesh.

4.4.1 Navigation Meshes

A navigation mesh or navmesh, describes the representation of all walkable areas in a game world to support pathfinding algorithms for agents that need to navigate through this world. Those areas are created by dividing the level in many different small parts, while each part consists of a number of polygons or squares. Every part represents a node in a connected graph and an agent can get from any point within that node, to any other position in that same node. If the AI needs to walk from point A to point B, a navigation mesh provides a path of a series of polygons the agent needs to walk through, to reach point B. But in some engines, there is not only a movement mesh, but also an obstacle mesh which stands for the obstacles in the level. Those meshes are usually implemented as a graph, which would offer the possibility to utilize the navmesh system for the FEM model.

4.4.2 Creating a sense graph from a navmesh

The navmesh system of an engine could be expanded and adapted to incorporate the creation of a sense graph, but to create a sense graph there are some requirements that need to be fulfilled:

Requirements

- *Sense Graph Nodes:* The nodes represent areas where a signal can pass and the edges describe the directions in which modalities can pass. A navmesh already is a graph describing the level but it has to be refined to be transformed into a sense graph.
- *Sense Graph Edges:* The edges of the navmesh need to have additional information describing the different modalities and their directions, and they need to be administered to be able to make changes later on.
- *Sense Graph Partitioning:* It must be possible to manage the different modalities while still maintaining an overview of the whole sense graph, for example by providing the option to divide the graph in different parts or by blending out specific modalities to be able to attend to one at a time.

Overall the navmesh system offers a good option to implement the tool support to create a sense graph and additionally mitigating one of the most difficult elements of the FEM model, the generation of the source data.

Chapter 5

Conclusion

5.1 Summary

In this thesis the implementation of a sensory system for Unity has been presented, which can be used by adding sensors and signals as components to game objects, to allow for the incorporation of AI perception in a game project right from the spot. The system is seamlessly integrated in the Unity user interface and uses a sense manager which works in the background and administers all sensors and signals. The perception works basically in three phases, first the manager finds all sensors, next it checks if a signal can be perceived and then the sensors are notified.

With just a few clicks it is possible make NPCs see and hear the game world which is demonstrated with a small test environment that illustrates how the system operates and the results it produces. With the custom build user interface, whose implementation and usage is explained in detail, it is possible to add AI perception in a quick and easy manner.

5.2 Result

Artificial intelligence can help developers to create more fun and more immersive games but its still neglected at times, because resources are spent on different aspects like graphics. Respectively some developers don't have enough time and resources to spend on a sophisticated AI system, but one fact of game AI development works in their favour: most times a simple solution offers the better result than a complicated one. The use of sensory systems is quite rare and additionally there is not much documentation to find even if it has been used in game. But there are numerous ways to implement such a system and it can greatly benefit the gameplay, especially in the genre of stealth games. The developed result shows that a pre-build sensory system can be set up and used quite easily and offers small developers to utilize a perception system to create more novel gameplay.

5.3 Prospect

Although the build sensory system works, it certainly can be improved further. In a game with a large number of signals and sensors it would be get increasingly difficult to keep track of all of them. Therefore it would be helpful to implement an appropriate menu where all sensors and signals are listed and where they can be altered at one central location, when needed.

Additionally sound signals are not affected by obstacles, it makes no difference if there is an obstacle between the source of a sound signal and the sensor, there is currently no attenuation implemented. A raycheck could be integrated quite quickly but sound attenuation has to be sophisticated to work well and thus would need a lot more additional effort to get right.

Appendix A

Content of the CD-ROM

A.1 Masterthesis

Pfad: /

Sommeregger_Bernd_2013.pdf Masterthesis

A.2 Literature

Pfad: /Literature/

*.pdf . . . . . Copies of the online sources

A.3 Projectfiles

Pfad: /Project/

Sensory System - Unity Toolkit/ All source files of the sensory system.

StealthSensorySystem/ Unity-Project folder with a test environment.

References

Literature

[1] S. Jacobs. *Game Programming Gems 7*. Game Programming Gems Series Bd. 7. Charles River Media/Course Technology, 2008.

[2] I. Millington and J.D. Funge. *Artificial Intelligence for Games*. Morgan Kaufmann. Morgan Kaufmann/Elsevier, 2009.

[3] S. Rabin. *AI Game Programming Wisdom 1*. AI Game Programming Wisdom. Charles River Media, Incorporated, 2002.

[4] S. Rabin. *AI Game Programming Wisdom 4*. AI Game Programming Wisdom. Charles River Media, 2008.

Films and audio-visual media

[5] Curve Studios. *Stealth Bastard*. Download. 2011.

[6] Looking Glass Studios. *Thief: The Dark Project*. CD. 1998.

Online sources

[7] URL: http://www.gamasutra.com/view/feature/2888/building_an_ai_sensory_system_.php?print=1 (visited on 06/19/2013).

[8] URL: http://aigamedev.com/open/review/thief-ai/ (visited on 06/19/2013).