# Agent Seek Final Report

**(v. 0.1.0)**

Francesco Magnani
francesco.magnani@studio.unibo.it

Thomas Capelli
thomas.capelli@studio.unibo.it

December 1, 2024

Agent Seek aims to develop a stealth-based video game where players have to complete several levels composed of obstacles and enemies, using the environment to avoid being detected. The player navigates a 2D arena with a top-down view seeking for the exit without alerting any enemy, implemented as intelligent agents. Agents will be provided with algorithms that analyze the state of the game and make optimal decisions independently, in order to make the user's gaming experience challenging.

## 1 Goal/s of the project

Starting from an old but fully functioning game project written in Java [2], the objective is to completely refactor it from the foundation to develop a MAS (Multi-Agent System) using Jason-lang [4] as the main technology. First of all, the main goals regard the restructuring of the old project:

- **Convert**: convert the project from Java to Kotlin and take advantage of all the functionalities of the new language.
- **Clean**: redesign, organize and remove everything that certainly will not be used in the new game.
- **Re-design**: adapt the project integrating the presence of Jason.
- **GUI**: create a GUI where agents and the user can interact with each other.

After that, the remaining goals completely regard the intelligent part, with the idea of implementing a system that includes different types of agent (Camera, Guard, Hearing), in which an agent should have the following properties:

- **Dynamic arena navigation**: agents should be able to dynamically navigate the map, avoid obstacles, and adjust routes autonomously, based on their perception or on signals received by allies.
- **Enemy identification**: agents should recognize an enemy, the user, and take different actions according to their type.
- **Agent-to-Agent communication**: agents should be able to communicate with other agents and react to signals coming from others, in order to coordinate and plan how to arrest the player.

## 1.1 Usage scenarios

1. **Launch**:
   - **User Action**: the user starts the game application.
   - **Response**: a GUI with a menu appears and the user can choose to start a new game, start a *scene editor*, load a scene from a file, or simply quit the game.
   - **Why**: the user lands in a menu with different possibilities and to ensure that the game starts only when wanted.
2. **Start the game**:
   - **User Action**: the user clicks the *start* button.
   - **Response**: the menu is changed with the GUI of the game.
   - **Why**: the user wants to begin playing Agent Seek.
3. **Play**:
   - **User Action**: the user interacts with the GUI by pressing W, A, S and D to move respectively Up, Left, Down, Right and Right. In addition, SHIFT can be used to reduce the player's speed.
   - **Response**: the GUI changes in real time and updates its parameters.
   - **Why**: the user interacts to progress through the game, managing to reach a destination to win.
4. **Level progression**:
   - **User Action**: by interacting with the GUI the player needs to reach a destination without being detected by the enemies.
   - **Response**: the level is completed and a new one is loaded.
   - **Why**: the player applies strategies to progress in the game
5. **How to win**:

- **User Action**: by interacting with the GUI the player needs to reach a destination without being detected by the enemies.
- **Response**: the level is completed and a new one is loaded. After N successful levels the game is considered won.
- **Why**: the player applies strategies to win the game

### 1.2 Definition of done

The project will be considered completed when the following aspects will be covered:

- User can interact and play several levels of the game through a GUI
- Agents of different types can percept, adapt and interact with the environment and with each other to achieve a consistent behavior in real-time.
- The intelligence of the agents makes the game challenging and appealing to the users.

The functionalities of the system can be tested in the following ways:

- **Unit tests**: code level tests to verify that complex and relevant aspects of the application work as expected.
- **Logger and debugger**: Jason includes a built-in debugger and logger. It is useful to track all the events that happen at a certain time.

The produced *artifact* will be an executable JAR (*Java archive*) that contains the whole game including all the needed dependencies.

## 2 Requirements Analysis

### 2.1 Business requirements

- **The game must include several levels**: the user can play several levels throughout the game. A level represents a unique "arena" where the player can move and act to reach an objective
- **Each entity within the game must have a unique representation**: the user must be able to distinguish the types of entities of the game.
- **The game should present a meaningful challenge**: players must encounter a certain level of difficulty to successfully complete the game.

## 2.2 Functional requirements

- **The game must provide a graphical interface**: the user must be able to play the game by the means of a GUI.
- **The game must include several types of environments**: including obstacles, walls, doors that can lead to other levels.
- **The game must incorporate a concept of *noise***: some behaviors, like the player's movement for example, can produce noise. Other entities can perceive the produced noise and react to it.
- **The game must incorporate a concept of *vision***: enemies should detect the player through "cones of vision".
- **The game must include at least three types of enemies**: including a *Guard Agent*, an enemy that can move, see and hear the player and communicate with other agents, a *Hearing Agent*, an enemy that can only hear the player and a *Camera Agent*, that cannot move but can alert other agents about the position of the player when it sees him/her.
- **The player must be able to move in a two dimensional space**: the player can easily move inside a free 2D environment, through keyboard input, interacting with the environment in several ways, for example clicking on the screen to generate noise or slowing down movement to produce less noise when walking.

## 2.3 Non-functional Requirements

- **Save/load game levels through files**: simplify the creation of levels by including the possibility to create and save them using *.yaml* files.
- **The system should be modular**: allowing for extensions or updates in an easy way.
- **The system should be sufficiently efficient**: the game must be playable at a sufficient frame-per-seconds ratio ($\approx 30$) without the need of a powerful machine.
- **The system's components must be re-usable**: the software must be designed in order to reuse components as much as possible.

## 2.4 Implementation Requirements

- **AI for agents inside the game must be implemented using Jason**: the Jason language must be used to implement the internal intelligence of each agent.

# 3 Design

Initially, the project was designed as a single, monolithic application organized into packages. However, upon further analysis, this structure was considered inadequate for the project's evolving needs. The system required greater flexibility to support extensions, incorporate new elements, and integrate intelligent components.

As a result, the project was redesigned into a modular structure with two distinct modules, each further organized into packages. This modular design was motivated by the need to separate concerns effectively:

1. **Engine Module**: Focused on converting existing functionality to the new architecture.
2. **Agent Seek Application Module**: Introduced a completely new user interface and incorporated all necessary components to enable intelligent agent behavior within the system.

This separation of concerns not only streamlines the development process but also makes the project easier to maintain and extend in the future. Each module and package is designed to encapsulate specific responsibilities, ensuring clarity and scalability.

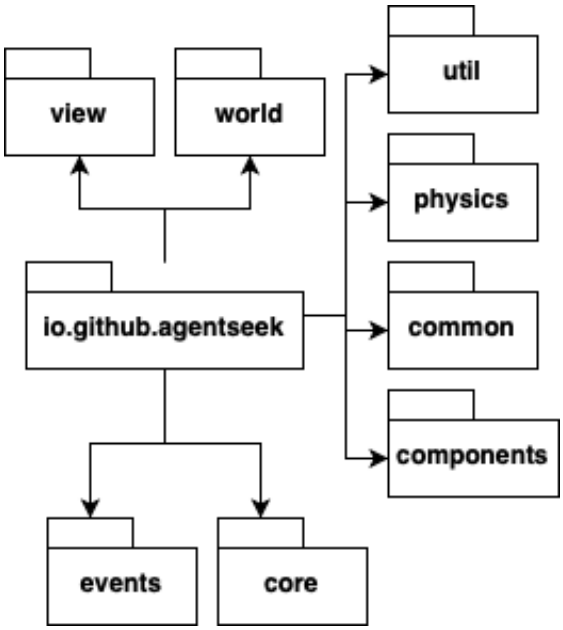



Figure 1: Engine sub-project

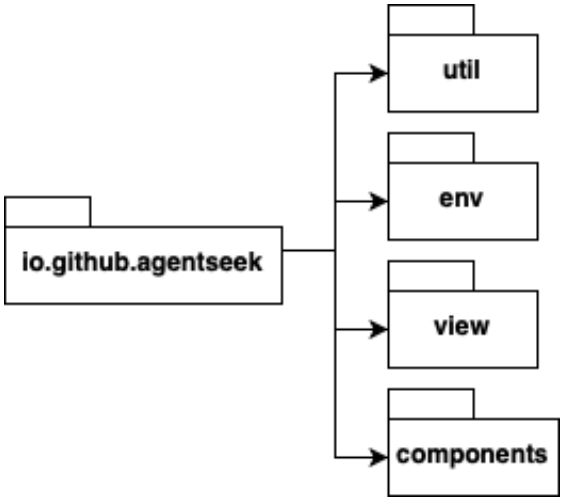



Figure 2: Agent Seek sub-project

## 3.1 Engine

As already stated, the engine part was adapted and enhanced from another project. The general structure, already adopted in the previous project, is similar to an **entity-component-system** architecture, and heavily inspired to the architecture used in the popular *Unity* engine [5].

The core elements are described in the following diagram.

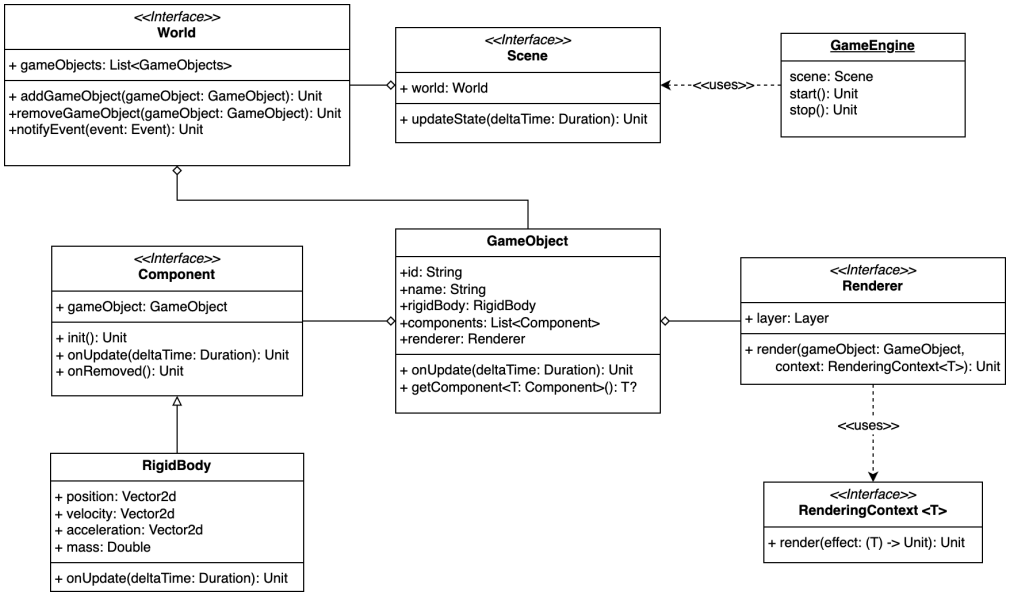



Figure 3: Engine core elements diagram

The public singleton `GameEngine` is used to load `Scenes`, which contains a `World` represent-

ing the physical 2D environment. `GameObjects` are the main entities of this environment, each representing a unique participant in the game / simulation. Game objects can have multiple `Components`, each characterizing the object in a specific way. For example, a component could specify a behavior for moving around, while another could fire projectiles in response to the player's input. The `RigidBody` is a special type of component, the one used to resolve physics-related events, such as collisions, movements, and forces.

The game engine updates the scenes during each iteration of the classic *game loop* structure. The scene updates the world, which updates each game object in it. An update of a game object consists of updating each component in order, with a specific call to the related method `onUpdate`. This structure allows for a highly customizable and reusable workflow. Since physics-related updates are particularly important, they are reserved for a specific update.

Finally, each `Renderer` can specify the graphical appearance of the game object. The renderer applies its effects through the use of a `RenderingContext` of type $T$, where $T$ specifies the type of object where graphical effects will be rendered. Since it obviously depends on the specific library used for rendering, the implementation of the renderer is left to the second module (the **Agent Seek application** one), which contains application specific requirements such as the graphics framework.

The general interaction between the previously described entities is described in the following diagram:

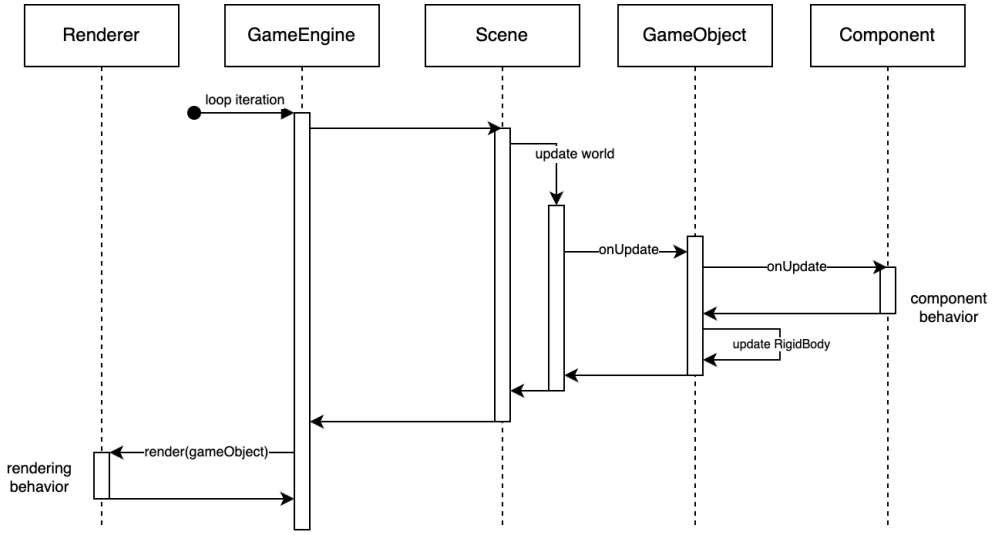



Figure 4: Interaction between the core elements of the engine

## 3.2 Agent Seek application

The actual application is implemented in this second module. All the abstractions and tools of the **Engine** module are used to implement the *Agent Seek* game.

The player is fully controlled by the user through interactions with the graphical user interface (GUI). In contrast, the game's intelligent system consists of three autonomous

agents, each with unique capabilities and varying levels of complexity.

These agents operate independently, updating their beliefs either by perceiving changes in the environment or through communication with other agents.

- **Hearing-agent**: As the name suggests, these agents rely solely on hearing to detect and respond to their environment. They represent the simplest type of agent in the system, updating their beliefs by listening for noise emissions. More accurately:
  1. If no noise is detected, the agent moves randomly through the environment, changing direction every 5 seconds.
  2. If a noise is perceived, the agent attempts to follow the source of the sound.

  A noise remains audible to the agent for up to 3 seconds after it is emitted. This straightforward behavior makes the Hearing Agent predictable yet effective at tracking the player when noise is present.

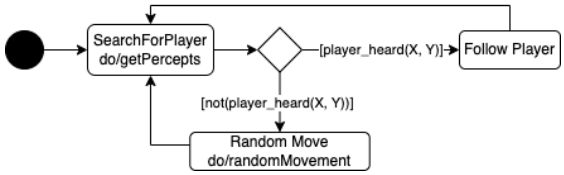



Figure 5: Hearing agent state diagram

- **Camera Agent**: This agent features a more advanced logic compared to the Hearing Agent. Its primary goal is to scan its surroundings to detect the player. While it cannot move, it is strategically positioned near walls, similar to real-world security cameras.
  1. **Idle Scanning**: If the player is not detected, the agent rotates every 5 seconds to a new direction, carefully avoiding positions blocked by nearby walls.
  2. **Player Detection**: Upon spotting the player, the Camera Agent sends a broadcast signal to alert other agents, providing the player's detected position.

  This agent's design ensures it serves as a static but important part of the *gameplay*, adding tension and strategy for the player to evade detection.

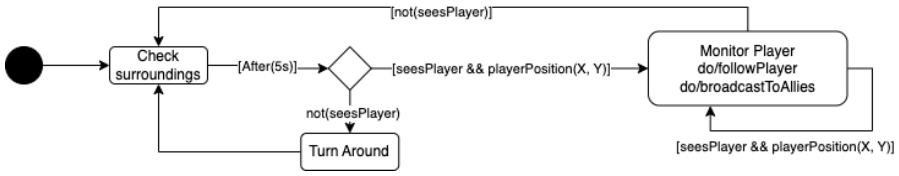



Figure 6: Camera agent state diagram

- **Guard Agent**: The Guard Agent is the most complex of the three, with six main goals to manage:

1. **Random Walk**: The default behavior is a pseudo-random walk used to explore the map and search for the player. The agent changes its walking direction every 4 seconds.
2. **Player Heard**: The agent can hear noises emitted within a certain distance. When it hears a noise, it moves toward the last known position of the sound and continues searching for 5 seconds.

   If no further noises are detected after this period, the agent resumes its random walk.
3. **Player Sight**: When the player is visible within the agent's cone of sight, it starts pursuing them. At the same time, the agent broadcasts a warning message to other agents.

   If the player escapes the agent's sight, the agent tries to regain visibility of the player for 5 seconds, before trying another strategy.
4. **Return to Base**: When the agent receives a signal from other Guard Agents, it moves directly toward the base to defend it. However, if the agent detects the player through sight or hearing while in route, it prioritizes pursuing the player instead.
5. **Defend Base**: Once the agent reaches the base and there is an active warning signal from other guards, it stops moving and waits for the player for up to 3 seconds.

   If the player does not appear, the agent adjusts its position slightly, pauses, and continues this behavior. This goal is maintained as long as the warning signal persists and the agent remains near the base.
6. **Follow Camera-Signaled Position**: The Guard Agent can react to broadcast messages sent by the Camera Agent. Upon receiving such a message, it moves toward the position where the camera last spotted the player.

The complexity of the Guard Agent arises from the integration of these goals. To manage them effectively, a priority-based design is implemented.

The priority hierarchy is as follows:

*Player sight* $\rightarrow$ *Player noise* $\rightarrow$ *Camera message* $\rightarrow$ *Defend base* $\rightarrow$ *Return to base* $\rightarrow$ *Random walk.*

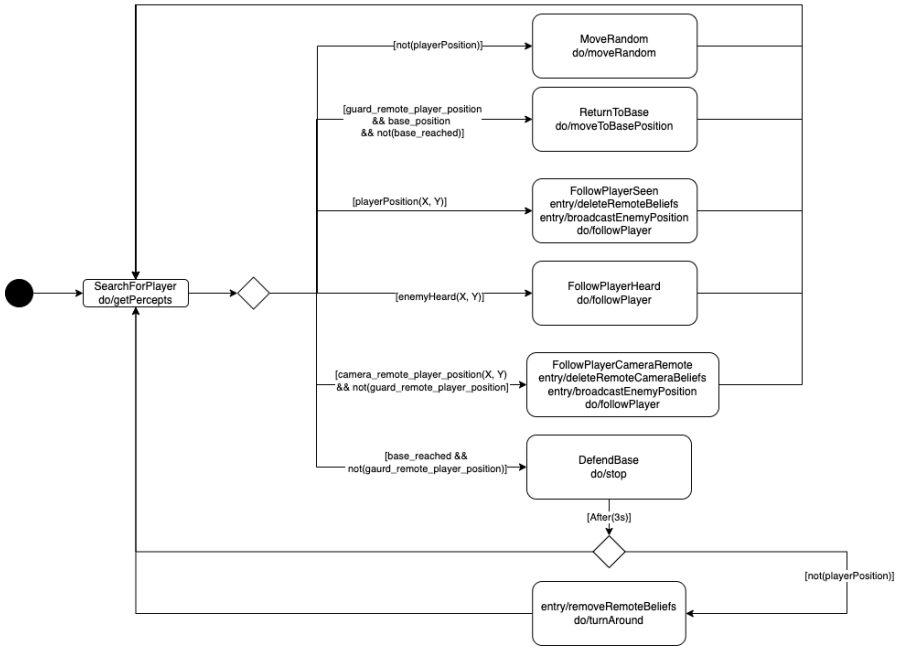



Figure 7: Guard agent state diagram

# 4 Salient implementation details

The following sections highlight specific implementations that we considered particularly relevant to the project.

## 4.1 JasonInitializerComponent

This class is one of the *components* (see Section 3.1 for more details) used by the **jason manager** object, responsible for the interaction between the `GameObjects` and the Jason environment.

It is mainly responsible for the launch of Jason [4] which is usually obtained by preparing a specific configuration file with the *.mas2j* extension that needs to be launched with an ad-hoc gradle task. The *JasonInitializerComponent* was implemented to simplify the whole process. It uses the function `generateTempFile` to setup the jason environment and to simplify the project structure by avoiding the creation of a permanent directory where `.mas2j` files should be stored.

Listing 1: Function generateTempFile in JavaInitializerComponent

```
private fun generateTempFile(
    name: String,
    environment: Class<*>,
    agents: List<Agent>
): File {
    val content = """
        MAS $name {
```

```
8
9               infrastructure: Centralised
10              environment: ${environment.name}
11              agents:
12              ${agents.joinToString(separator = "\n\t\t") { "${it.id} ${it.
                    aslAgentName};" }}
13
14              aslSourcePath:
15              "agentseek/src/main/asl";
16          }
17          """.trimIndent()
18
19      val tempFile = File.createTempFile("MAS_", ".mas2j")
20      tempFile.writeText(content)
21
22      return tempFile
23  }
```

After generating a temporary file, it uses the function `RunLocalMas` to start it instead of using the gradle task, resulting in a cleaner way of managing jason, that can coexist with the kotlin language and with the game engine implementation.

Listing 2: init function in JavaInitializerComponent

```
1  override fun init() {
2      val file = generateTempFile(mas2jName, Class.forName(environmentClassFQName),
           agents).absolutePath
3      jasonManager = this.gameObject
4      Thread {
5          RunLocalMAS.main(arrayOf(file))
6      }.start()
7  }
```

The configuration is obtained from a scene file that is created as a result of the serialization process. The result is a *.yaml* file, created using the library jackson [3]

The concept behind the provided function is that the configuration file is generated dynamically based on the parameters of the scene (such as the environment class and the agents).

## 4.2 JasonAgent

The intelligent part of the system is achieved by integrating jason-lang (in *.asl* files) and logic written in kotlin, two parts that must communicate to create a consistent behavior of the agents.

Starting from the kotlin part, the system was designed to have several combinations of agents, each extending the abstract class `JasonAgent`.

Listing 3: JasonAgent abstract class

```
1  abstract class JasonAgent(gameObject: GameObject) : AbstractComponent(gameObject)
       {
2      /**
3       * ID for this Agent, corresponding to a Jason agent ID
4       */
5      abstract val id: String
6
```

```
    /**
     * Handler for the actions performed by this agent.
     */
    abstract fun execute(action: Structure): Boolean

    /**
     * Gets the percepts for this Agent
     */
    abstract fun getPercepts(): MutableList<Literal>
}
```

The class takes a GameObject as a parameter to underline the link between a jason agent and the objects of the system. Moreover, it provides two main functions to extend:

- The `execute` function can be used to manage the commands coming from agents in the form of `Structure`.
- On the other hand, the `getPercepts` function can be used to manage the sensorial part to pass inputs to agents as beliefs.

Every agent in the project was created by extending the provided class in an appropriate way (and many more can be created following the same approach).

Then there is the part implemented via jason-lang. In particular, the focus goes on the *GuardAgent*, which is the most complex of our system's intelligences, because it has a lot of possible goals to follow. In order to understand the main objective, we decided to implement a sort of *thinking* phase(called `searchForPlayer`) in which the agent evaluates its beliefs and decides the actions to take.

Listing 4: Guard Agent evaluation

```
+!searchForPlayer : player_position(X, Y) <-
-guard_remote_player_position[source(_)];
-camera_remote_player_position[source(_)];
!alertAllies;
!followEnemy.

+!searchForPlayer : enemy_heard(X, Y) <-
!followEnemy.

+!searchForPlayer : camera_remote_player_position(X, Y) & not
    guard_remote_player_position <-
!followEnemy.

+!searchForPlayer : base_reached & guard_remote_player_position <-
!defendBase.

+!searchForPlayer : guard_remote_player_position & base_position(X, Y) & not
    base_reached <-
!returningToBase.

+!searchForPlayer : not player_position(X, Y) <-
!moveRandom.
```

Due to this approach, it was necessary to sort the thinking goals so that the agent follows a priority mechanism.

### 4.3 Agent movement

The implementation of the agents' movement is inspired by robotics, specifically *field-based movement*, following an approach similar to the *motor schemas* architecture [6].

In this mechanism, agents are equipped with a **proximity sensor** that detects the distances to nearby objects in 8 directions around the agent. Each direction generates a vector that is combined to produce a **resultant distance vector**, whose magnitude increases as the agent approaches obstacles. The field-based movement uses this resultant vector to avoid collisions as follows: the vector is rotated by approximately 120° to obtain a nearly tangential direction, scaled by a `danger coefficient` (higher values correspond to stronger responses to obstacles), and finally added to the **objective vector**. The objective vector is derived from the agent's internal behavior, representing the next target point the agent aims to reach. The sum of these vectors determines the agent's next velocity.

This approach allows for a simple movement behavior driven by the agent's internal logic while ensuring collision avoidance, without over complicating the agent's specification with unnecessary calculations.

## 5 Deployment Instructions

To play Agent Seek, follow these instructions:

1. Go to the GitHub page of Agent Seek [1]
2. Go to the **Release** page
3. Download the JAR from the latest release
4. Run the jar doing `java -jar agentseek-<VERSION>.jar`
5. In the GUI click *Start Game*

## 6 Usage Examples

The game starts from a simple menu (figure 8) where the user can choose between multiple options:

- **Start game**: pressing this button causes the switch to the game GUI (figure 9).
- **Start game with REPL**: let the user start the game with the addition of a REPL (Read-Eval-Print Loop) that can be used for debug and testing.
- **Start selected**: the button becomes clickable once the user has selected one scene (on the left side of the menu). If clicked, the user can play the selected scene.
- **Start editor**: after the selection of a scene, the user can press this button to start editing the scene by moving, adding, or changing game objects to customize a level.

- **Start editor with empty scene**: starting from an empty scene, the user can create a whole new level.

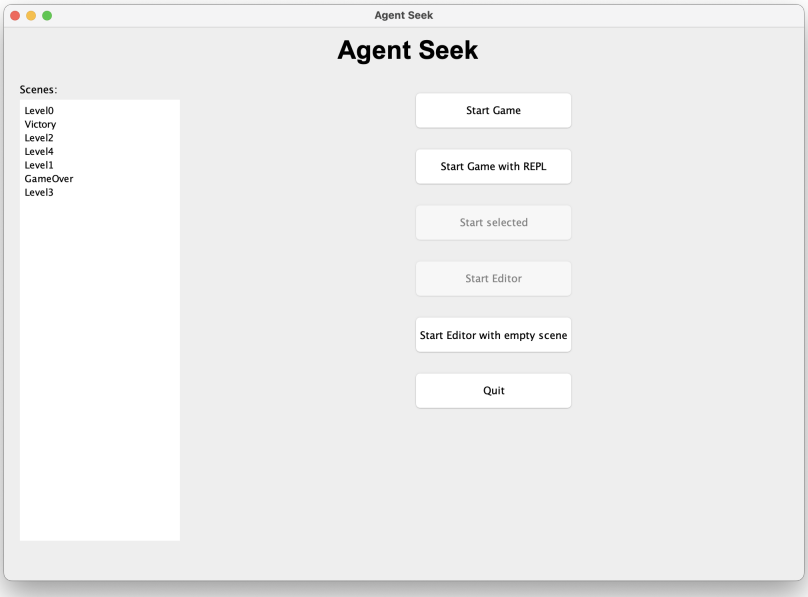



Figure 8: Starting menu

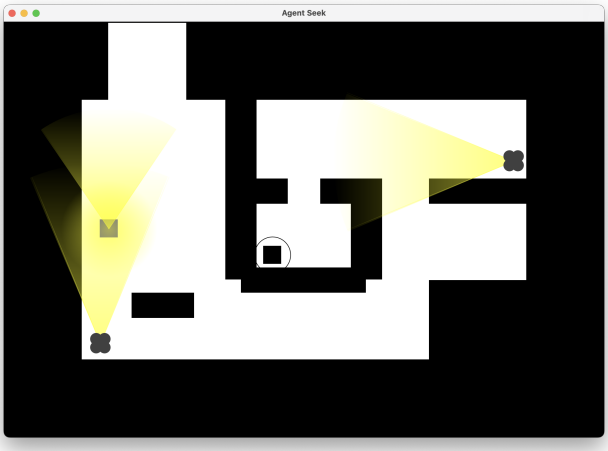
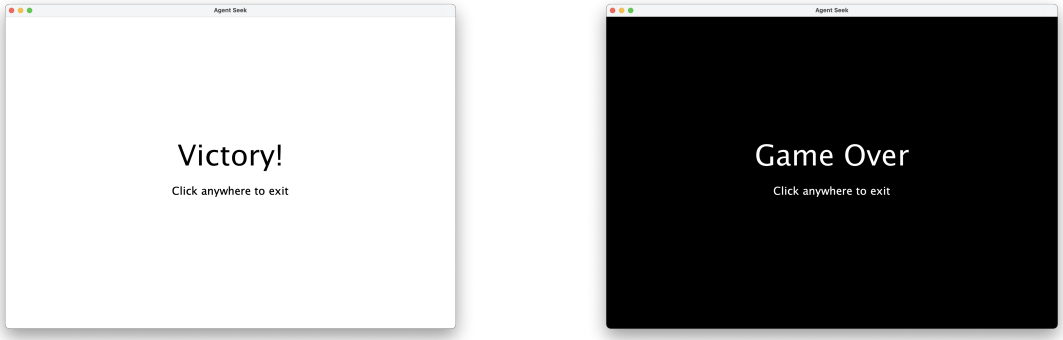



Figure 9: Example level

After starting the game, the player has to win several 2D levels (figure 9) with enemies (interacting using W, A, S, D and SHIFT keys). The user can also distract enemies by

clicking around the scene: this action will generate noise that is detected by agents. Each arena was designed to achieve an increasing level of difficulty. While playing, if the player gets caught by an enemy, the game is lost and the game-over screen is shown (figure 10b). Otherwise, after winning all the levels the victory screen is shown to the user (figure 10a).




(a) Victory screen

(b) Game Over screen

Figure 10: Ending screens

# 7 Conclusions

In conclusion, the *Agent Seek* project provided a valuable opportunity to explore the integration of two distinct environments: a Kotlin game engine and a Jason multi-agent system. The primary challenge lay in combining an already functional game engine with a separate environment like Jason, identifying "connection points" and shaping the agents' design accordingly. The agents' behavior has been specified taking the engine's components mechanism into consideration, so the major effort has been put into "bringing Jason towards a custom environment", rather than "building a custom environment on top of Jason". This approach allowed the team to develop a scalable strategy for leveraging Jason in pre-existing projects, exploring a way for its broader applicability and usability.

## 7.1 Future Works

As the built environment is highly flexible and scalable, an intriguing extension to the *Agent Seek* project would be the development of **more complex agents**, potentially shifting the focus towards simulation aspects rather than solely game-related features. Additionally, enhancing the **interaction** between agents—currently limited to basic messaging about the player's position—could enable the creation of nature-inspired simulations, effectively visualized in the free, continuous environment implemented in this project.

Finally, future work could concentrate on **enhancing the game engine** by incorporating more physics-based elements, such as gravity and friction, to introduce an additional layer of complexity in defining agent behaviors.

### 7.2 What did we learned

As previously mentioned, one of the most interesting and challenging aspects of the project was the **integration** between the two distinct environments. For instance, using *actions* implemented in Kotlin from Jason proved to be both intriguing and complex, particularly in terms of achieving a clean and efficient design and implementation. Leveraging the strengths of two programming languages simultaneously is no easy task, but this investigation provided valuable insights into a potential approach, specifically through the adoption of an architecture like the *entity-component-system*.

The approach adopted for the definition and implementation of the agents aimed to leverage the declarative nature of a language like Jason. Having previously worked on robotics projects, where the behavior of robots was defined in a much more direct and low-level manner, using this language allowed us to focus more on the "what" rather than the "how" during the development of agent behaviors. To support this, efforts were made to extract aspects not considered inherently declarative from Jason code into Kotlin code, keeping the former clearer and more readable.

## References


[1] Agent seek github page. `https://github.com/FreshMag/AgentSeek`.

[2] Isaac-rge project. `https://github.com/FreshMag/OOP20-isaac-rge`.

[3] Jackson github page. `https://github.com/FasterXML/jackson`.

[4] Jason language. `https://jason-lang.github.io/`.

[5] Unity engine. `https://unity.com/`.

[6] R. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Proceedings. 1987 IEEE International Conference on Robotics and Automation*, volume 4, pages 264–271, 1987.