ASTRA Multi-Agent System integrated with Unity3D.

Petar Stefanov, Bachelor of Computer Science

A thesis submitted to University College Dublin in fulfilment of the requirements for the degree of

Master of Science

College of Engineering, Mathematical and Physical Sciences
School of Computer Science and Informatics

Ireland

Supervisor: Assistant Professor Abraham Campbell

Supervisor: Assistant Professor Rem Collier

Date: December 17, 2017

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master in Computer Science, is entirely my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: Date:

Acknowledgements

I would first like to thank Prof. Abraham Campbell and Prof. Rem Collier for working with me for over a half year and helping me come up with a finished project. I especially appreciate them help as I worked through many topics, trying to find the right one, and them patience in answering all my different questions. Second, I want to thank my family for supporting me through this entire experience for the past two years. Without the help of those above I do not know how I would have completed my master's in Computer Science and the quality project that is before you.

Contents

Abstract

In recent years interest is growing in applying Artificial Intelligence(AI) to Virtual Reality (VR) by using software agents. This paper aims to provide an overview of the rapidly developing area of VR/AR technologies and the potential of the Agent-Oriented Programming (AOP) paradigm for bridge between AI and VR/AR. In this paper a novel approach for connecting Agent-Oriented Programming (AOP) to Virtual Reality (VR) and Augmented Reality (AR) world is suggested. The effort to connect ASTRA language to Unity3D, by building an API for direct access is reported. This paper will outline how ASTRA Agent-Oriented Programming Language has been brought to the game engine Unity3D and the possible benefits of it.

Chapter 1

Introduction

1.1 Motivation

Until recent years, the world of Virtual Reality (VR) belonged in the fantasy realm of science fiction movies. The iconic 90s movie *The Lawnmower Man*, *Star Trek*, *Brainstorm*, and more recent like *The Matrix*, *Inception*, *Total Recall*. All these movies illustrate the concept of being trapped within the machine/cyberspace, or as a form of advanced technology, or use of VR to enhance the mental skills, or utilization of VR technology to enables the person to experience any situation they want - holodeck in *Star Trek*.

These days Virtual/Augmented Reality technologies can be embedded in reality to provide people with more understanding of the world around them. Increasingly it is being used as a tool by military, education, media, health-care, entertainment and businesses in general.

Military has adopted the technology for training processes - flight simulation[19], medic training, battle field simulation. The VR training is conducted using head-mounted displays (HMD) and data gloves to enable interaction within the environment. Education use VR for teaching and learning, thus enables groups of students to interact with each other as well as within a 3D environment[20,21,39]. Medicine use VR/AR for diagnostic, robotic surgeries and skills training[22].

The VR/AR technologies are still in early stages of its development, but the possibilities

are huge. VR/AR could be a primary form of communication technology in the near future, it could eventually impact all of the senses. VR/AR enable humans to enter environments that do not exist in the real world and interact with them. Businesses in many industries adopted these technologies to gain a competitive advantage. Immersive experience, visualization, remote guidance, content augmentation, marketing and sales, education and training, these are some of the benefits that VR/AR technologies will bring to the businesses, enable them project real life situations[25].
On the other hand, there is the Artificial Intelligence (AI) - the simulation of human intelligence processes by computer systems. The processes that include ability to learn - the acquisition of information and rules for using the information, ability to reasoning - using the rules to reach definite conclusions[35].
In recent years interest is growing in combining together AI with VR/AR for more benefits[3,10,24]. Many examples that bridge the two fields have emerged recently - intelligence in digital games, utilization of computer graphics hardware for machine learning and AI, emotional interaction. While AI and VR/AR went rather distinct research pathways, the attempt to bring them together is obvious. Question raise: what would happens when the artificial world start to think and humans interact with it? How humans can interact and communicate with AI through better interfaces? I believe VR/AR are going to be the humans interface to AI. One possible answer is to combine AI with VR/AR world is using Agent Oriented Programming (AOP) paradigms and bringing the Multi-Agent System approach.
VR is composed of number of technologies - 3D display, various input devices, motion tracking hardware, software platforms. Game engines such as Unity3D, Unreal, Cryengine, Lumberyard have become development tools of choice for creating VR/AR[36,37]. Unity3D is a high-performance end-to-end development platform used to create rich interactive VR/AR experiences. The potential of Unity3D game engine as a platform for AI development is huge. In most cases the AI is hard coded as part of the VR/AR world programs. Segregation of VR/AR world programs and programs building and driving the AI is what is needed. This will make the software better and more maintainable, it will reduce complexity.

With development of VR/AR technologies gaining more and more speed, it is important to realize what value AI can add/bring to these technologies, to make the end product more complete, engaging and intelligent. Imagine VR/AR world as a place that humans can interact, learn, engage with an intelligent autonomous avatars embedded within this world. Agents capable of performing tasks without any supervision, agents that posses knowledge which can be shared. It is crucial to have an infrastructure that will enable these technologies to work together. This infrastructure can be build by using game engine like Unity3D as a platform, which enables the VR/AR development, and the intelligence can be brought in Unity3D by adopting the approach of implementing Multi-Agent Systems using Agent-Oriented Programming languages.
Agent-Oriented Programming (AOP) is a programming paradigm introduced by Shoham in the early 90s [26]. Its introduces the concept of mental states of an agent, beliefs, decisions, capabilities, and obligations. An agent can be defined as a computational entity that sense the environment, deliberate and then act. This paradigm fits very well with VR/AR world, where AOP approach can be used by incorporating agents embodied within a virtual character.

1.2 Objectives

The primary objective of this paper is to investigate how to embed AOP languages into the Unity3D game engine. To outline how the Agent-Oriented Programming language - ASTRA has been connected directly with Unity3D via an API deployed directly into the Unity3D environment. This has been done without the need of client/server network communication approach.
Secondary objectives:

- Show the ability of Multi-Agent System to communicate and interact within the VR/AR world.
- Discus the tools used to integrate Java - based AOP language directly to Unity3D game engine. The potential of IKVM for embedding Java - based AOP languages

in .NET applications. IKVM is a JVM for the Microsoft .NET Framework and Mono. It can both dynamically run Java classes and can be used to convert Java jars into .NET assemblies. It also includes a port of the OpenJDK class libraries to .NET.

- Outline the simplicity of the API created. In this objective the design and the implementation of the API is discused and how is used.
- Show the use of the API by describing the prototypes created.
- Provides a comparison between the current approach with the previous work that has already be conducted for combining the game engines with Agents.

1.3 Project planning

From the beginning, a plan was set up to fulfill the necessary requirements of the project.

- Initial reading - to read through several articles and papers, to cover the new trends in AI, VR/AR.
- Develop understanding of how Unity3D game engine works.
- Investigate the existing approaches of bringing Agents in game engines in particular Unity3D.
- Implementation - planning, design and implementation.
- Testing the system - test-driven development(TDD).
- Writing down the report using LaTeX.

1.4 Contribution

Considering Unity3D as our VR/AR environment the main goal of this work was to find the most efficient way of connecting ASTRA - an Agent Oriented Programming language to the VR/AR environment. Number of different approaches were examined initially as a possible solution - using RESTful Web Service over TCP/IP for communication between the agents and the environment, using Java Native Interface (JNI) to call directly methods within the environment and using Unity3D Network API for 'hooking' ASTRA. None of these was as efficient as deploying directly ASTRA into Unity3D

The bridge between both, outlined in this paper is based on the fact that ASTRA is compiled to a Java byte code, hence IKVM.NET was introduced to allow the conversion of the Java code to .NET dll libraries. This way ASTRA can be deployed directly into Unity3D.

This novel approach of connecting ASTRA, and in general AOP to Unity3D, outlined in this paper, gives the ability to connect any Java-based Agent-Oriented Programming languages directly to C# scripting game engines and to access native C/C++ libraries. As the agent behaviour is programmed externally, not within the environment program, it still obey the principle of *separation of concerns*, which is important and crucial for reducing complexity. By integrating ASTRA directly within the environment the risk of lagging, when both interact was reduced to minimum.

Developing unit test for the API, makes the solution reported in this paper much more reliable, maintainable and scalable. This provides a platform to tackle much easy the impact that the API can have if any of, Unity3D API or game engines platform future changes happens.

1.5 Thesis structure

The thesis is structured as follows:

Chapter 2. Overview of Unity3D - game engine as a VR/AR environment and ASTRA - Agent-Oriented Programming (AOP) language.

Chapter 3. Embedding Agents in Unity3D, high level design, analysis of how Unity3D works and how agents can be linked to Unity3D.

Chapter 4. Developing AstraApi. Description of the implementation of the API. All public

	methods description. Maven as a building tool.
Chapter 5.	Compatibility and Performance, and finding the perfect balance between Unity3D environment and the API
Chapter 6.	Implementation of Unity3D prototypes.
Chapter 7.	Testing the system and managing the quality of code.
Chapter 8.	Conclusions are drawn based on the research and the final opinions are stated.
Appendix	Uml class diagrams, applications screen shots and guide how to build and deploy the API within Unity3D.

Chapter 2

Overview

2.1 What is Virtual/Augmented Reality?

Virtual Reality (VR) and Augmented Reality (AR) technologies have been around for some time. Virtual Reality allows the user to perceive and interact with a simulated environment, it replaces the real world with a simulated one[15]. Augmented Reality aims to enhance the experience of the environment by overlaying information onto a user field of view as it perceives the real world[38].

The roots of the modern Virtual Reality, the concept of alternative existence was introduced in late 1950s[16]. It was first introduced by the cinematographer Morton Heilig who invented Sensorama[17], a machine that combines immersive, multi-sensory technology. The machine was able to display 3D images, provide vibrations and stereo sound. This invention paved the road of the modern Virtual Reality technologies. In the early 60s, Ivan Sutherland invented the first actual VR head set - head-mounted display (HMD) *Sketchpad*[18]. It was innovative software that influenced alternative forms of interaction with computers.

Virtual Reality is defined by degrees of freedom - the number of ways a user can move around and interact with the virtual environment. There are a total of six degrees of freedom in a 3D space, about and along the x, y, and z axes. **Rotational movements** - pitch, yaw and roll, and **Translation movements** - left/right, forward/backward and

up/down.
The main accessory needed to enable the Virtual Reality is the **Headset** - stereoscopic displays, also known as head-mounted display (HMD), a device like a pair of goggles that goes over users eyes. This device uses a combination of multiple images and special lenses to produce a stereo 3D image. **Head tracking** technology that enables the picture in front of user wearing the headset, shifts as he/she looks up, down and side to side - these are the so called six degrees of freedom. Also **Motion tracking** hardware - gyroscopes, accelerometers used to measure acceleration forces and to sense when users body moves in such a way the application can update the users view inside the VR. **Data gloves** which allow user to interact with objects within VR world.[27]
Augmented Reality is different from Virtual Reality in that it does not remove the user from the surrounding world. While Virtual Reality systems will fully remove the world around the user while wearing the headset, Augmented Reality allows the user to look around, while interacting with data and objects that surrounds the user. Virtual Reality is a digital recreation of the real world, while Augmented Reality delivers a virtual world as an overlay to the real world.

2.2 Enabling VR/AR

There are number of tools and platforms used to develop VR/AR applications. Virtual Reality development is possible through:

- **Virtual Reality Modeling Language (VRML)** which is used to create a series of images, and specify what types of interaction is possible for them[29].
- **ARToolKit** is a software that allows programmers to develop Augmented Reality applications. ARToolKit has a plug-in for Unity3D[28].
- **Unreal Engine 4** is a suite of development tools for enterprise applications and cinematic experiences, it allows also to create games across different platforms and VR/AR[32].
- **ARKit** has been developed and introduced by Apple, a developer framework for

creating an AR on mobile devices and iPad. It provides a motion tracking, plane, scale estimation. It also provides support for Unity3D and Unreal platforms[40].

- **SketchUp**[34] and **Blender**[33] are among the best available tools for modeling 3D, UV mapping, rendering and animation.
- **Unity3D**[4] - last but not least, a cross-platform game engine, that supports 2D and 3D graphics and scripting through C#. It also supports different operating systems on different platforms, i.e. Windows, Linux, macOS as well as mobile devices - Android and iOS. It has a direct VR mode to preview the application in a HMD. Unity3D is the most widely used VR development platform[5][30][31].

2.3 Examples of VR/AR

The VR/AR applications can be experienced through the headset (HMD) and display on mobile devices. Many applications have emerged in recent years, addressing different areas, including but not limited to, the military, sports, education, health-care and entertainment industry.
Medical use - medical students use the VR/AR applications/simulators to practice surgeries in a controlled environment, these applications gives the surgeon improved sensory perception, thus it can reduce the risk of the operation performed. Another application is the 3D ability to image the human brain.
Militaryl use - It was adopted heavily by the miltiary for simulations and training. Through VR/AR soldiers are able to engage with a simulated battlefield and pilots can learn to fly aircraft in a battle without the cost of flying real aircraft. The military also uses VR/AR for medical purposes, for example for treating post-traumatic stress disorder in soldiers after suffering battlefield trauma.
Education and Entertainment - Google developed Expeditions[41], a Virtual Reality teaching tool. It is a virtual reality classroom that help engage students with immersive lessons. Expeditions allows students to join immersive virtual tips in any environment, on the ground, under the water and even in the space. There are more than 600 trips

available. To use the application all its needed is a mobile device and Daydream - virtual reality platform built into the Android operating system or Cardboard [42], affordable headsets of high quality that enable the virtual experience. Another mobile VR headset - Gear VR was developed by Samsung, and its compatible with most of the new Samsung smartphone devices. NASA partnered with Facebook's Oculus Rift to build an application that lets people to explore the International Space Station. The application allows to float down the Space station and event step out for a space walk.
Engineering and Design in Construction - DAQRI[50] provide a full hardware and software solution of AR through the DAQRI Smart Glasses and DAQRI Smart Helmet. These products incorporate digital and holographic data into physical environment. Make process of designing, constructing and visualization of structures more convenient and easy. It can be used for a view of the building plans, for materials and fixtures in order to be put into precise alignment, without the need to consult any paper documents. These tools allows virtually to explore the design of the structure, even before its build.

2.4 Agent-Oriented Programming

This is a short introduction into the concept of the agent systems and the features they posses.

2.4.1 Multi-Agent System

Multi-Agent System (MAS) is a collection of agents sharing the same environment, tasks, goals and behavior. MAS consists of multiple interacting agents that cooperate, coordinate and negotiate. MAS is an alternative to centralized problem solving, because problems are themselves distributed and because the distribution of problem solving between different agents reveals itself to be more efficient way to organize the problem solving - it can allow failures in the system. For solving the problem agents cooperatively exchanging information as the solution is developed. This itself will reduce the complexity when intelligence is build. Multi-Agent System (MAS) is an effective software engineering paradigm for designing and developing complex software systems [43,44].

2.4.2 What is AOP?

Agent-Oriented Programming (AOP) is a programming paradigm introduced by Shoham in the early 90s. Its introduces the concept of mental states of an agent, beliefs, decisions, capabilities, and obligations. An agent can be defined as a computational entity that sense the environment, deliberate and then act[26]. Since then many implementations of AOP languages have been developed. AOP language is derived from a logical model of commitment. AOP paradigm enables to program intelligent agents that are able of reasoning about what to do next on the basis of beliefs and desires[9].
Agent-Oriented Programming (AOP) can be viewed as a specialization of Object-Oriented programming[2,26]. There are some similarities between both, but in its core the AOP does not promote a consistent conceptual model. Different AOP languages are based around different approaches, therefore it is hard to compare with Object-Oriented programming.

2.4.3 Anatomy of an Agent

The term 'agent' has been a subject of a years of debate, many researchers have proposed a variety of definitions. The common understanding is that an agent is an entity that posses some degree of independence, performs actions in its environment by deliberating to achieve the goal. Wooldridge and Jennings stated in 1995; 'An agent is a computer system that is situated in some environment, and that is capable of flexible, autonomous action in this environment in order to meet its design objectives'[43].
Agents can be described in two broad categories: *weak notion of agency* and *strong notion of agency.* The behaviour characteristics of an agent - software entity, are described by Woolridge and Jennings[43, 45]:

- **Weak notion of agency**
  - *Autonomy* - agent operates without the direct intervention of humans or others and has some control over its own actions and internal state.
  - *Reactivity* - agents are able to perceive their environment and are able to respond to it in a timely fashion.
  - *Pro-Activity* - agents are able to take initiatives to perform actions and may set and pursue their own goals.
  - *Social Ability* - agents have the ability to coordinate and cooperate while performing tasks. Agents are able to interact with other agents via communication.
- **Strong notion of agency** - is an agent system, build on top of the characteristics described above but also mental human-like characteristic such as: beliefs, desires, intentions, knowledge, commitments and obligation.
  - *Mobility* - an agent ability to move around.
  - *Benevolence* - every agent will always try to do what is asked to do, no conflicting goals.
  - *Rationality* - agent is goal orientated, assuming it will not act is such a way as to prevent its goals.
  - *Adaptivity* - an agent should adjust itself to the habits, methods and preferences of its user.
  - *Intentionality* - an agent reasons about its activities through the application of mental notions such as beliefs, goals, obligations, commitments, intentions.

The agent architectures are very complex, because of the dynamic aspects that must be dealt with and the richness of the agent-oriented world. However, the basic of all architectures is **perceive - reason - act** cycle, a useful abstraction that provides a high level view on the agent architecture in general. Most common and popular model of agent architecture is Belief, Desire and Intention (BDI). It was first introduced by Rao and Georgeff in 1991[46], in this model the rational agent possessed particular 'mental attributes':

- **Beliefs** - represent the informational state of the agent, in other words its beliefs about the environment.
- **Desires** - represent the motivational state of the agent, objectives or situations that the agent would like to accomplish or bring about in the environment.
- **Intentions** - is a desire that has been adopted for active pursuit by the agent, in essence desires that agent is committed to achieve.

2.4.4 Characteristics of MAS

Multi-Agent System (MAS) consists of number of agents with the ability to cooperate, coordinate, and negotiate with each other, in essence to interact with each other. The tasks in MAS are distributed between the agents, where each agent is considered as a member of the organization. Distribution of tasks involves assigning roles to each of the agents. The agents in MAS have several important characteristics:

- **Autonomy** - agents are independent, self-aware, autonomous.
- **Local views** - no agent has a full global view of the system.
- **Decentralization** - in the system there is no controlling/main agent, all agents have a particular task and they interact with each other.
- **Cooperation and Coordination** - agents are assumed to be acting autonomously, however they are capable of dynamically coordinating their activities and cooperating with others in order to achieve common goal. Agents are capable of doing it so through partial global planning, joint intentions and mutual modelling[45].

There are number of advantages in the concept of MAS. Ability of sufficient control and responsibilities shared among agents within MAS, such that tolerating failures of one or more agents is possible and is an option. It is easy to add a new agents to a MAS with new capabilities compare to a monolithic software system.

2.4.5 Example AOP languages

Agent-0

Shoham developed the first Agent-Oriented Programming language - *Agent-0* in 1991[47]. It is a simple Multi-Agent programming language with a particular architecture. The agent state has two components - *Beliefs* - agent actions can update its beliefs or can be updated as a result of being 'told' by other agent; and *Commitments* - activities that agent is committed to perform, which implicitly define the future actions for the agent. The language design is centered around the notion of 'commitment to action as opposed to the more goals driven BDI approach which has become more common' [48]. In essence the agent act based on its beliefs to achieve it commitments.
Given the architecture of the language, Shoham defines three fundamental modalities: belief, obligation and capability. *Beliefs* - enables agent to have beliefs about what it thinks the current state of the environment is and beliefs about other agents beliefs. *Obligation* - the notion of commitment to action, in that it represents the commitment of one agent to another at a given time to believe a certain state of affairs. *Capability* - defines situational conditions an agent can bring about at a given time.[48]

AgentSpeak(L)

AgentSpeak(L) was proposed by Rao[9] and its based upon BDI model and practical reasoning architecture - Procedural Reasoning System(PRS)[49]. PRS system can be described as a 'framework for constructing real-time reasoning systems'[49] and its based on the notion of rational agent using BDI model. The behaviour of the agent and its interactions with the environment are controlled by a set of first-order logic statements with events and actions. The language has the following components:

- *Beliefs* - the information that the agent has perceived from its environment, represented in first-order logic.

- *Goals* - predicate formulae, identify what the agent wished to achieved.
- *Events* - drives the agent behaviour, set of external and internal events that have occurred during the interpreter cycle are added to the event queue and act upon based on the plan rules.
- *Plans* - or plan library, which identifies what action to be pursed when event occur.

Jason

Jason [56,57] is Java-based implementation for an extended version of AgentSpeak(L). Jason obeys the BDI model and provides the possibility to run a MAS distributed over a network using Java Agent Development Framework (JADE)- platform for peer-to-peer agent based applications[58]. Jason enables developing external environments.

ASTRA

ASTRA[1] language implementation is based upon AgentSpeak(L), language based on Belief-Desire-Intention (BDI) theory[9]. ASTRA stands out from the other AOP languages with a set of important features that make ASTRA more practical. ASTRA provides a strict static type system, mutual exclusion in plans execution, extended set of control statements (if statements, for-each loops and while statements), message receiving plan rules.
ASTRA is closely integrated with Java via typed variables, annotations used to specify actions, terms, events, formulae and sensors, thus used directly in agent code. Its supports multiple inheritance which enables re-usability, allows combination, overriding and partial declaration of agent behaviours. All ASTRA code is compiled to Java, thus it is offering a simple mechanism to deploy and integrate ASTRA agents with other systems.

2.5 Agents and VR/AR

Agents already are used in combination with the VR/AR world. Agents in VR/AR systems not only enhance a human perception of the real world, but also increase the human interaction capabilities. Example of agents embodied in VR/AR environment is the high-end computer animation and AI software package used for generating crowd-related visual effects for film industry - MASSIVE [11]. Agent-based approach helps to enable agents, part of the crowd to have a degree of intelligence, hence they can react to each situation on their own based on a set of decision rules. This makes possible for simulating realistic crowd behavior.

Another example is NeXuS - an inherently multi-agent based framework that uses Agent-Factory [63] agents to have full control of the environment. NeXuS demonstrates how goals driven BDI approach can be achieved by incorporate agents within VR/AR world. It allows to develop context-aware systems with high modularity by allowing agents to have plans for any situations, steps needed to achieve the desired goal.

In recent years an increased interest in research and development of applications with agents embodied in a Mixed Reality (MR) environment is observed. MR was first defined by Milgram and Kishino in 1994 as "...anyware between the extrema of the virtuality continuum." [59]. It enables the creation of environments where the VR/AR world and the physical world can coexist. Thomas Holz et al. [60] conducted a very comprehensive survey of applications with agents embodied in VR/AR/MR environments as part of the taxonomy they offered in 2011. More than 25 different applications were discussed in detail. The most relevant ones are outlined below.

One of the best examples of such a system is the MagicBook [61]. It's an interface which utilizes a smooth 'transportation' between a physical book and a VR/AR world. The book can be handled and read like any normal book, but using the VR/AR display the users can fly inside and can fully immerse themselves in the virtual scene that is in the real book. The system allows multiple users to experience this VR/AR world and interact with each other.

Another example is in the construction sector, with combining the digital content with

physical buildings. Similar to DAQRI[50] as discussed above, virtual and real buildings are combined together to enable planning and early evaluation of construction development projects.
Application of agents embedded in the VR/AR world are popular in the education sector. There already exist systems that enables autonomous learning. The agents embedded in VR/AR can play a key role for bringing the learning process to the next level. It can be used for developing an interactive, fully autonomous systems/'classrooms', where the tutor is an intelligent agent.

2.6 Discussion

Summing up, the rise of the VR/AR technologies, brings a lot of opportunities along with number of questions that need to be answered. The application of VR/AR with embedded agents technologies has potential to grow up rapidly. This is evident by the intention of the biggest high-tech companies like Google, Apple, Facebook, Samsung etc., to invest in this field, which they are doing it already. This will change how humans interact with the world and there is an urgent need of more effort which will lead to more complete and better solutions. The huge potential in applying AI to Virtual/Augmented Reality world is obvious, but how AI will interface with VR/AR? The answer possibly lies in applying the Agent-Oriented Programming paradigm with Multi-Agent System approach to make that bridge.
AOP paradigm is more than 25 years old, it is already mature and despite it being not very popular in the software industry, I still believe it is capable of making the bridge between AI and VR/AR world. It will provide such needed interface between AI and VR/AR.
Establishing a bridge between ASTRA and Unity3D is not something new, as previous work has already been conducted for bringing Agents to game engines. Hindriks et al[10], connected GOAL agents with Unreal Tournament 2004 game engine using EIS. However, very little work has been done in connecting AOP particularly with Unity3D

game engine. The potential of Unity3D as a VR/AR development environment is clear, hence the possibility of a bridge between AOP and Unity3D requires urgent investigation to identify if this embodiment of agents in the VR/AR world is possible. This thesis explores how to make this connection between AOP and Unity3D game engine and outlines in detail the path taken to implement it.

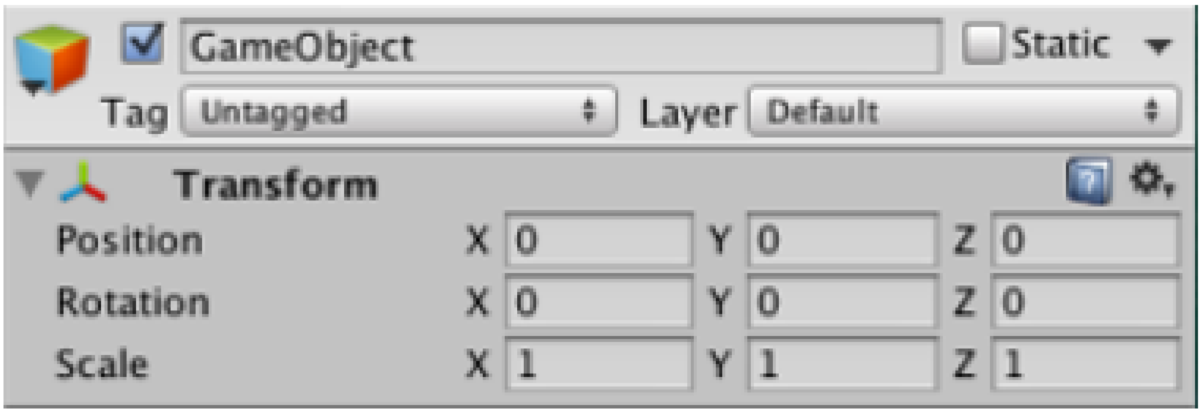
Chapter 3

Embedding Agents in Unity3D

3.1 Unity3D Overview

Unity3D[4] is a cross-platform game engine, supports 2D/3D graphics, and scripting through C# and JavaScript. The core Unity3D engine itself is written using C/C++ native language. The graphics, sound, physics are coded in C++[51]. The interaction/access to the core Unity3D engine is done through C# wrappers, which runs on the Mono virtual machine. Mono project provides the environment to run C# applications on any device.

The *GameObjects* is the most important concept in the Unity3D, it is the fundamental Object that represent characters, prop or scenery. They act as containers for *Components*. Most common and important *Component* is the *Transform Component*, which defines the *GameObject* position, scale and rotation of the object in the game world. It is added automatically to every *GameObject*. Figure 3.1 shows the Transform Component.


GameObject

Static

Tag: Untagged

Layer: Default

Transform

Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Figure 3.1: Transform Component

Usually the *GameObject* has several *Components*. They are used to define the physic, layout, effects, events, audio ect. Importantly they provide the option to add scripts to the *GameObject*, such that it can be controlled, influenced programmatically.

The behaviour of the *GameObject* is controlled by the *Components* attached to it. All the logic in Unity3D is written inside *MonoBehaviour* - so called behaviour scripts. All the behaviour scripts must derived from the built-in class *MonoBehaviour*. These scripts allow to modify properties of the *Components*, to trigger events and response to the user input. Every *MonoBehaviour* script have just one functionality/responsibility, and do not operate outside the *GameObject* itself. They are designed with modularity in mind, thus they can be reused independently on number of different *GameObject*. Their design obviously follows the basic concepts of Object-Oriented Programming (OOP).

Important note, Unity3D provides the option to use any .NET languages if they can compile to compatible DLL[52]. This fact was exploited to link ASTRA to Unity3D engine.

Through inheritance form *MonoBehaviour* (main, built-in class) number of important methods/functions are provided for controlling and modifying the *GameObject*. The *Update* function is the place where to handle the frame update for the *GameObject*. For every frame, movement, triggering events, position, rotation and responding to user input, should be handled inside this function. Also *Start* method is provided which runs

before game starts and is the ideal place to do all initialization needed.
To have controll over the *GameObject* in the Unity3D world, all what is needed is access to its basic properties, the ones that determines its position, scale, and rotation. They can be obtained from the mandatory *Transform Component.* The information that each property contains is: *position* in X, Y, and Z coordinates; *rotation* around the X, Y, and Z axes, measured in degrees; *scale* along X, Y, and Z axes, where value '1' is the original size of the object.
More information can be extracted from the *Components* attached to the *GameObject.* Information related to the physic of the object is provided via the different *Collider* components available. Some of the properties that can be amended programmatically are: *Is Trigger* - if enabled, it is used for triggering events, and is ignored by the physics engine; *Material* - determines how the *Collider* interacts with others; *Center* - position of the *Collider* in the *GameObject* local space; *Size* - in the X, Y, Z directions.

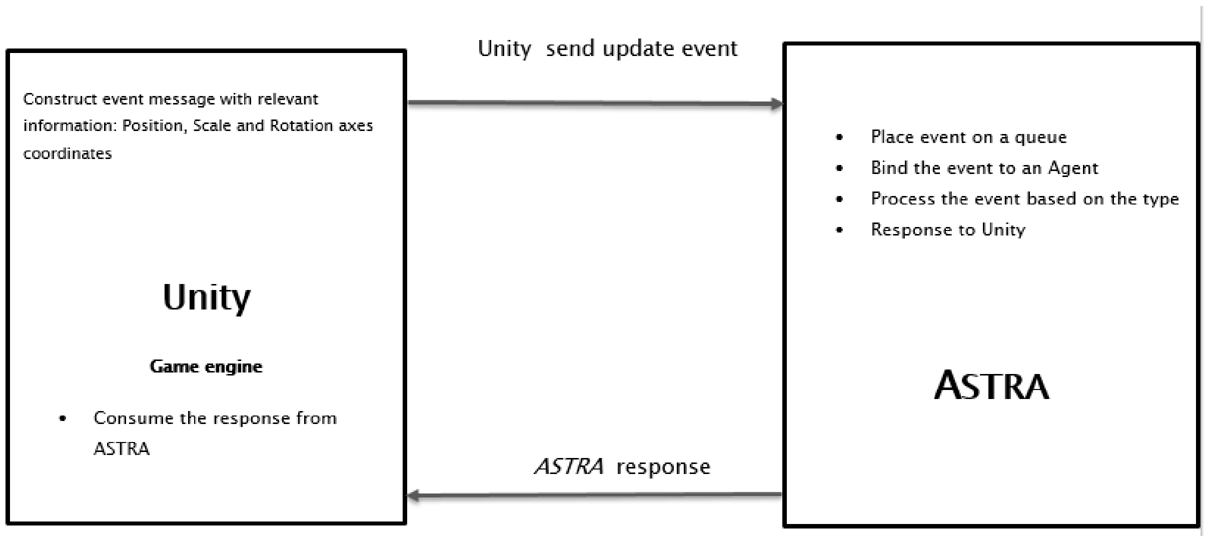
3.2 Agents and Unity3D

To embodied agent into Unity3D *GameObject* access to Unity3D scripts is required. There are number of different ways to gain access to Unity3D scripts.
Unity3D provides multiplayer and networking capabilities through the High Level API (HLAPI). HLAPI provides services for using multiplayer games such as: message handlers, general purpose high performance serialization, distributed object management, synchronization, and network classes - Server, Client, Connection, etc. This approach would require to build a complex infrastructure in order to gain access to Unity3D - setting up and configuring a Server, creating, potentially two clients, establishing connection, keeping this connection alive during the entire duration of the game. There are many pitfalls in this approach - complex design, possibly a dedicated machine for the Server, codding a server-client interface etc. No matter that Unity3D uses UDP/IP based protocol for transferring the data there still would be a delay in the network stream. 'UDP packets does not require any knowledge that the destination received the packets, UDP is relatively immune to latency. The only effect that latency has on a UDP stream

is an increased delay of the entire stream.'[53] HLAPI could be used in conjunction with Photon Unity Networking[54] which will provide a network framework and API to link to Unity3D, therefor less coding required, but still the network latency issue will exist.
Another option is to link the agents using Java Native Interface (JNI) and call directly methods within the Unity3D environment. This will require to write native methods to handle all functions that need to be supported. It means extra layer between Java-based ASTRA and Unity3D. Native code is called from Java using JNI, which comes standard with the JDK. JNI supports code portability across all platforms and allows code written in C/C++/Assembler to run alongside the JVM[55]. Downside is that these functions calls to JNI methods are very expensive, especially when methods are called repeatedly, what would be the case in Unity3D, the game engine updates the state of the *GameObject* every frame. Also extra amount of codding is required as Java classes need to be written with embedded calls to native code and also should include static code that references the native library in which the methods it refers to can be found.
Option also is to use RESTful Web Service over TCP/IP for communication between the Agents and Unity3D. This approach was explored recently by a researcher in UCD labs, where Jason AOP language was linked with Unity3D. It requires an interface to be developed to handle all RESTful calls. An issue heres is that exposing API over TCP/IP definitely will introduce a delay which will lead to a poor performance in Unity3D environment.
The most efficient way to link agents with Unity3D is if the agents language/platform can be deployed directly into Unity3D environment and communicate with it without any networking layer. This will give the best performance, factor that is so important in the gaming environment, in general when two different systems interact with each other. This can be done by converting the Java-based AOP language to a dll, compatible with Unity3D environment, for this purpose IKVM.NET tool is used as described in details in Section 5.3

3.3 Architectural Design

How the bridge can be achieved? As stated above, for controlling the Unity3D *GameObject*, access to the *Transform Component* properties - position, rotation and scale, is required. This can be achieved by building an API that provides an option for creating agent from Unity3D and transferring data from and to Unity3D. An agent will be embodied into *GameObject* via the Unity3D scripting. As every *GameObject* is allowed to have script attached to it, the script will have a reference to the API and will have an unique instance of AOP agent. The API should supports transfer of the data in a json format with the current state of the *GameObject*. Consider the following simple diagram for communication between both, where data is transmitted in a json format:



Unity send update event
Construct event message with relevant information: Position, Scale and Rotation axes coordinates
Unity
Game engine
• Consume the response from ASTRA
• Place event on a queue
• Bind the event to an Agent
• Process the event based on the type
• Response to Unity
ASTRA
ASTRA response

Figure 3.2: Communication

The API designed will be converted into dll assemblies and deployed/placed inside Unity3D scripting structure, thus it will be available for direct access within the *GameObject* via the attached C# script. An instance of the API will have a global access across all *GameObjects*. Each script in its *Start* method/function should initialize an unique (unique name) instance of an agent, which will be embodied into the *GameObjects*. Using the *Update* method/function data will be transmitted to the API. Consider the integration diagram bellow and a high level view of ASTRA deployed in Unity3D:

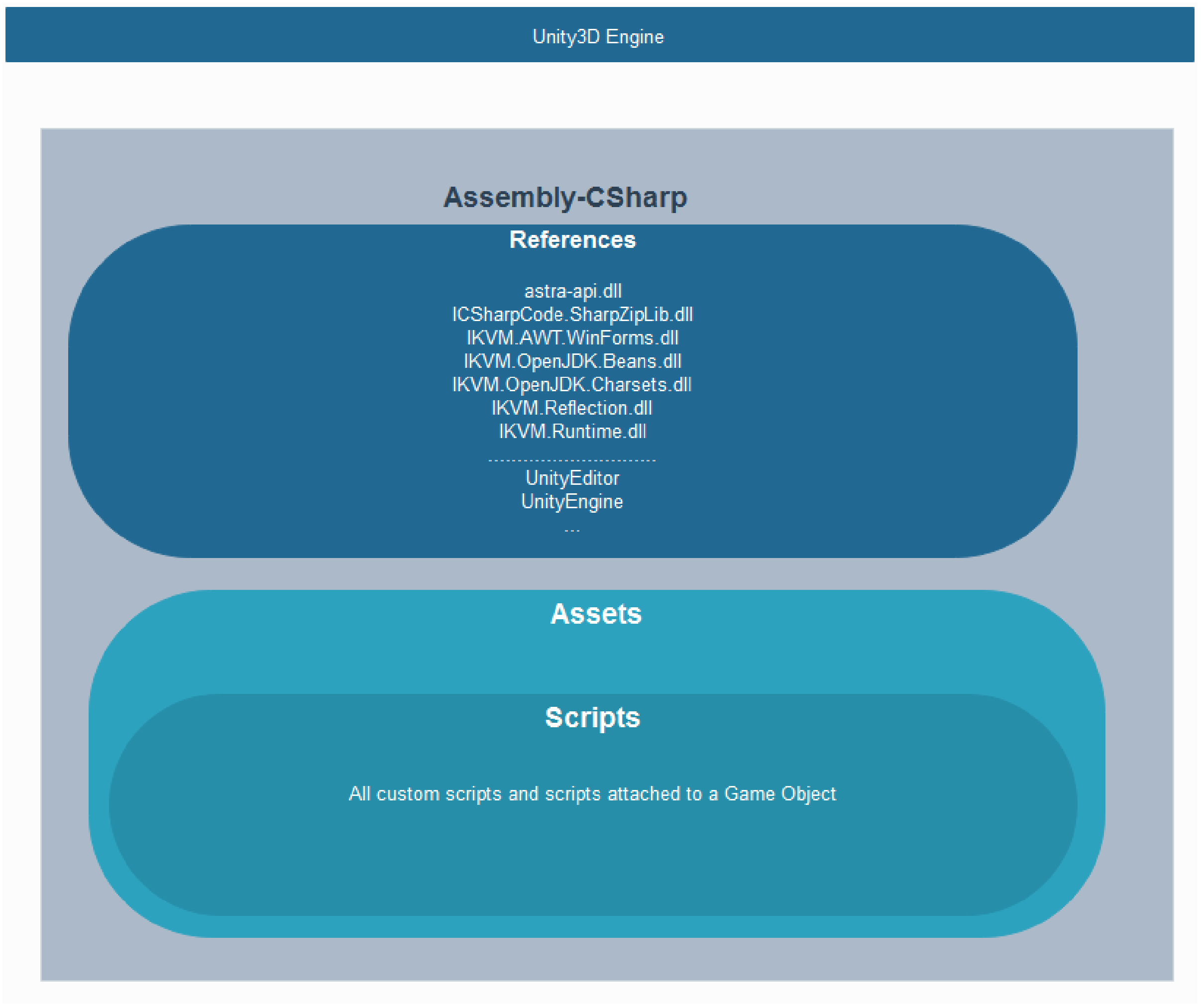


Figure 3.3: Structure of the API after it was deployed in Unity3D

This is a high level structure view of all required assemblies after the game is built. This diagram shows where exactly the API is located - under the References, thus any *GameObjects* script has a direct access to it. The *astra-api.dll* is part of reference libraries, also all IKVM dll's required are placed there. Basically it shows that the *astra-api.dll* along with IKVM dll's became part of the Unity3D compile and build process. The ASTRA API is fully incorporated inside Unity3D engine.

3.4 Conclusions

The bridge between agents using AOP paradigm and Unity3D environment suggested here is at a machine level, it is a direct connection between both. This approach gives the best performance and efficiency, as ASTRA API along with the IKVM dll became part of the compile/build process of Unity3D. IKVM.NET tool enables the bridge between agents and Unity3D, it is done with very little effort.
The available data that is required for embedding the intelligence, by using ASTRA, into the *GameObjects* in Unity3D is mainly in the coordinate format. Unity3D system use three axes: x,y and z, for representing the rotational movement - pitch, yaw and roll, and translation movement - left/right, forward/backward and up/down. This data can be represented and transmitted in a simple json format, thus allows the design of the API to avoid the support of a complex data structure, which simplifies the development of the API. ASTRA API design, with all public methods available is described in details in the following chapter.

Chapter 4

Developing ASTRA API

4.1 Design

The design of the API was driven from the fact that ASTRA is an event-driven programming language. All what is needed is to provide a mechanism for creating agents and passing information from and to Unity3D.
Events in ASTRA drive the behaviour of the agent through the activation of plan rules. Three types of event are supported - goal events, belief events and message events. ASTRA is designed to support custom event types also. This allows to create events that model changes in the environment. The API designed uses the concept of custom event. Custom event, called *UnityEvent* was created. It takes as arguments the *Term eventIdentifier* - event type, *ListTerm arguments* - value as an Object and has a signature *$ue*. By using Factory Method pattern in Java code the desired Object is derived for each event type from *AstraCommand* base class and provided different implementation for each event, see Figure A.2. This allows the API, based on the event unique identifier to construct the appropriate event object, bound it to the agent and process the event. In ASTRA, custom event is handled by creating rules for every different event, as its shown in the code snippet below, e.g. position event:

```
rule $unityModule.event("position", [string event]) {
    !position("position", event);
}
```

Every event is mapped to the correct agent, this is achieved by storing every event in a concurrent map data structure with a key constructed by concatenation of the agent name and the event type. All event's data is passed from and to Unity3D is in a json format.

4.2 ASTRA Modules

Modules are Java classes that can be used to extend the basic functionality of ASTRA agent/s in the form of libraries. They can be included in ASTRA code with a sintax similar to Java (import), declaration goes on top, first thing inside the class body:

```
...
module MyModule myModule;
...
```

The Java Modules classes make the link with ASTRA via a set of annotated methods. ASTRA supports five basic annotations: @EVENT - used in the triggering event of a rule; @TERM - used to return a value; @FORMULA - used to create and return logical formulae; @ACTION - represent internal actions that can be performed, returning a boolean value indicating if the action was successfully performed; @SENSOR - used to generate beliefs. This ASTRA feature was used in the API design.

4.3 Public Methods

Seven public methods were developed and exposed through the API for use in Unity3D environment. See Figure A.1

- **createAgent** - create an agent with a given, unique name and Class name to match the desirable capabilities of this agent. The methods allows to verify if agent was successful created by returning the provided agent name. If agent with the same name already exists or an Exception is thrown during the the process of agent creation an appropriate message is returned.
  - **Parameters accepted:** String - agent name, String - className.
  - **Returns:** - String - agent name or a message.
- **removeAgent** - an ability to terminate and remove agent form the registry. If an agent exists in registry, will be removed and appropriate message is sent back. If no agent with this name in registry, no action taken and appropriate message is returned.
  - **Parameters accepted:** String agent name.
  - **Returns:** - String - message.
- **asyncEvent** - an ability to add an asynchronous event from Unity3D to an agent based on the event type. The type used in Unity3D should match the one supported by the API. If the agent doesn't exists bind the event to the current agent. Event is added to the Queue and once is processed can be retrieved form the Queue using receive method.
  - **Parameters accepted:** String - agentIdentifier, String - eventIdentifier, Object[] eventArgs.
  - **Returns:** - void.
- **syncEvent** - an ability to add synchronous event from Unity3D to an agent based on the event type. The type used in Unity3D should match the one supported by the API. If the agent doesn't exists bind the event to the current agent. The method returns the processed event.
  - **Parameters accepted:** String - agentIdentifier, String - eventIdentifier, Object[] eventArgs.

  - **Returns:** - String - json with the processed event.

- **submitCommand** - an ability to send a command from ASTRA code to ASTRA Modules, this method is used to support the link between ASTRA Modules and Java, but also can be used directly in Unity3D.
  - **Parameters accepted:** String - agentIdentifier, String - eventIdentifier, String - command.
  - **Returns:** - void.

- **receive** - retrieve an event form the Queue stored in a Map with a key - the agent identifier concatenated with the event type.
  - **Parameters accepted:** String - agentIdentifier, String - eventIdentifier.
  - **Returns:** - String - json with the processed event.

- **clear** - an ability to remove a specified event from the agents event Map, if such entry exists in the Map, by removing all of the elements from Event Queue. The Event Queue will be empty after this method returns.
  - **Parameters accepted:** String - agentIdentifier, String - eventIdentifier.
  - **Returns:** - void.

4.4 Events type

Four main events are supported by the API: *position*, *collision*, *position_vector* and *messaging* events. Per each event type separate ASTRA Module was created to handle the requirements. An Unity3D Module was created also to link the API to Unity3D in terms of ASTRA. Base class was created in ASTRA, which provides the link to all Modules and the API, it needs to be inherited by every agent:

```
agent Unity {
    module api.modules.Unity unityModule;
    module api.modules.Position positionModule;
    module api.modules.Collision collisionModule;
    module api.modules.PositionVector positionVectorModule;
    module api.modules.Messaging messagingModule;

   rule +!unity(api.AstraApi api) {
      unityModule.linkToUnity(api);
   }
}
```

All four events type are explained in details in the following sections.

4.4.1 Position event

The most important event type to be handled of course is the *position* event. It takes as arguments from Unity3D an event message in a json format, where *cardinalDirection* is optional:

```
{
  "type":"position",
  "scale":{"x":0.5,"y":1.0,"z":0.5},
  "rotation":{"x":0.0,"y":0.0,"z":0.0},
  "position":{"x":1.649999976158142,"y":1.0,"z":2.700000047683716},
  "cardinalDirection":"South"
}
```

How it works? It is important to know that the last seven positions information sent from Unity3D are recorded in a Linked List data structure, therefore API is capable of preserving the insertion order. For the very first time an event is sent API expect *cardinalDirection* to be part of the json in order to determine which axis to amend. If the data structure holding the last seven position is not empty a comparison is performed, between the current and the very last coordinates, of the absolute values with accuracy of 0.1, base on the result the coordinates are mended and an appropriate sign to all coordinate axes - x-axis, y-axis and z-axis is added. The value changed is done by adding or subtracting the predefined API change rate, which is 0.7. This value was determine by considering a lot of factors, the speed the agent is moving in Unity3D world, the number of frames updated per second and the frequency of position update. In short the new position is determined based on the previous coordinate axes (x,y and z), in an event there is no difference between both, the same coordinates are send back to Unity3D.

4.4.2 Collision event

Similar to position event, collision event takes as arguments from Unity3D, an event message in a json format. Difference here is that json contains two extra attributes - *instanceId* and *astraCardinalDirection* which are used in Unity3D:

```json
{
  "type":"collision",
  "position":{"x":1.149999976158142,"y":1.0,"z":2.700000047683716},
  "scale":{"x":0.5,"y":1.0,"z":0.5},
  "rotation":{"x":0.0,"y":0.0,"z":0.0},
  "cardinalDirection":"South",
  "instanceId":9670,
  "astraCardinalDirection":"West"
}
```

How it works? In the same manner as has been done for the *position* event the last three event messages are recorded. This time they are stored in a Map data structure with a

key constructed by concatenation of *instanceId* and *cardinalDirection*, where *instanceId* is the id of the Object agent collided with in the Unity3D world (it is an unique) and a Linked List data structure with event message as Map value. This way all different collisions are bound to the same agent. The *cardinalDirection* is obtained in Unity3D based on how the Unity3D game Object face it the object with which it collided.
The decision of what direction to assign to the agent after the collision occurred is made based on previous/last five collisions. The logic used to make the choice is as follow:

- **Trivial/base case - empty map or no value for current key**:
  - Remove (rule out) the current *cardinalDirection* passed from Unity3D from the list with **base_cardinal_directions**.
  - Get a random index for the remaining cardinal direction.
  - Set a random cardinal direction to be passed to Unity3D called *astraCardinalDirection*.
  - Store the decision data with the new entry *astraCardinalDirection* - the one used in Unity3D.
- **If map not empty and map contains a value for this key**.
  - Remove (rule out) the current *cardinalDirection* passed from Unity3D from the list with **base_cardinal_directions**.
  - Iterate through the list in reverse order.
  - For each iteration check if two left from the **base_cardinal_directions**, if so break the loop.
  - For each iteration remove from the list of **base_cardinal_directions** if not yet removed. This will remove the last choices made for the same collision. Leaving two directions choice that yet to be made, different from the last one.
  - Get a random from the remaining **base_cardinal_directions**, at least two should be and set it to *astraCardinalDirection*.
  - Store the decision data and check if the list exceed three remove from the tail.

We do not want to store more than last tree decisions.

For clarity - **base_cardinal_directions** are: North, South, West and East.

Two options are given for use in Unity3D. Use directly in Unity3D code the *astraCardinalDirection* or use the position json attribute which has the updated axes coordinate with a rate of change of 0.7 as in position event. In short, what is done here is that API guarantee if an agent collides with the same object/obstacle again, it cannot have the same outcome in terms of *astraCardinalDirection* as the last result.

4.4.3 Position direction event

Handles event of type *position_vector* sent from Unity3D. The difference with position event type is in the json object returned to Unity3D. Instead precise coordinate axes for position attribute it contains position attribute with values for coordinate axes either 1.0 or 0.0. This is dictated from the way Unity3D use the position coordinate axes for moving the game object (applying physics). The same could be done using the same event by adding extra attribute to the json response, but decision was taken to keep both options completely separated for simplicity. The snippet below shows how the response json looks:

```
{
  "type":"position_vector",
  "scale":{"x":0.5,"y":1.0,"z":0.5},
  "rotation":{"x":0.0,"y":0.0,"z":0.0},
  "position":{"x":1.0,"y":0.0,"z":0.0},
  "cardinalDirection":"South"
}
```

In general the same logic applies here as with the *position* event type (based on the recorded last seven events), only when constructing the position json attribute the values are added based on the change made compare to the previous position.

To show the difference in two different ways of consuming position coordinate axes in Unity3D, consider the C# script below:

```
//position assigned directly
Vector3 position = new Vector3 (2.343212, 0.0f, -3.568903);
this.position = movement;

//use position vector to move the game object by applying physics
Vector3 direction = new Vector3 (0.0, 0.0f, -1.0);
this.transform.Translate (direction * moveSpeed * Time.deltaTime, Space.Self);
```

4.4.4 Message event

Provides the ability to send and receive message and act depends on the content of the message. Can be used for communication from ASTRA to Unity3D and the other way around, by sending simple text context.

```
{
  "type":"messaging",
  "message":"Any text message can be placed here!"
}
```

4.5 Problems and challenges

Problem was encountered during development which we overcome by fixing the core ASTRA language API. Test was created to reproduce and track down the issue. Test consisted by two sequential calls, made to ASTRA for the same event - *position* with different parameters. In ASTRA side all was looking correct.

```
...
rule $unityModule.event("position", [string position]) {
   string direction = positionModule.getDirections(position);
   console.println("direction: " + direction + " for position: " + position);
    !sendDirection(direction);
}

rule +!sendDirection(string Directions) {
   unityModule.sendCommand("position", [Directions]);
}
...
```

To verify the ASTRA side was working as we expected, text was printed out in the development environment. Both calls for the rule *sendDirection* were producing different parameters. This was the expected outcome as the position was different every time:

```
[agentOne]direction:
{"horizontal":-1.0,"vertical":-0.3299777} for position: HorizontalRight
[agentOne]direction:
{"horizontal":-0.15952972,"vertical":-1.0} for position: VerticalUp
```

The problem appears to be when the rule was invoked from the Java code:

```
  public boolean invoke(Intention intention, Predicate predicate) {
    return ((api.astrasupport.Unity)
        intention.getModule("Player","unityModule")).sendCommand(
       (java.lang.String) intention.evaluate(predicate.getTerm(0)),
       (astra.term.ListTerm) intention.evaluate(predicate.getTerm(1))
    );
  }
```

This was the output from the JUnit test in Java, basically wrong as the values for both calls were identical:

```
syncEventPosition:
    {"value":["{\"horizontal\":-1.0,\"vertical\":-0.3299777}"],"type":"position"}
asyncEventPosition:
    {"value":["{\"horizontal\":-1.0,\"vertical\":-0.3299777}"],"type":"position"}
```

Issue was identified as the Intention for both calls was different, where the Predicate stays the same with the value from the first call. The issue was addressed by fixing the core API. It was a bug related to the passing of a list containing a variable into an action (the variable was being replaced with the associated value, but was not 'released' after the method was executed).
This problem was found and fixed in a very early stage, thanks to the adoption of test-driven development. In essence thanks to the unit test written the bug was caught.

Chapter 5

API Usage, Compatibility and Performance

5.1 API Usage

Unity3D provides scripting API for both C# and JavaScript. Usage of the API is well documented, there are many tutorials and online support form the huge Unity3D community. For this work C# language was used. Let see how the ASTRA API is used within Unity3D environment.

As explained in section 4.3 seven public methods are available. The information from and to ASTRA is transferred in a json format. Good approach is to have a high level declaration and instantiating of the ASTRA API for easy access and avoid duplication. In the snippet below class created instantiates the API and declares all the event types supported and the additional parameters.

```
public class GameManager : MonoBehaviour {

  public static string EVENT_POSITION = "position";
  public static string EVENT_COLLISION = "collision";
  public static string EVENT_MESSAGE = "messaging";
```

```
    public static string EVENT_POSITION_VECTOR = "position_vector";

    public static string CARDINAL_DIRECTION_NORTH = "North";
    public static string CARDINAL_DIRECTION_SOUTH = "South";
    public static string CARDINAL_DIRECTION_WEST = "West";
    public static string CARDINAL_DIRECTION_EAST = "East";

    public static int MAZE_END_INSTANCE_ID = 7777777;

    public static api.AstraApiImpl api = new api.AstraApiImpl ();
}
```

5.1.1 Create and terminate Agent

ASTRA supports Multi-Agent System, therefor agents created require unique name and also the 'type' of the agent - ASTRA class that implements the agent behaviour. Mechanism could be put in place if multiple agents are required in Unity3D to ensure the agent name is unique. On creation of the agent, the API checks if agent exist with the same name and if does error message is returned, otherwise API returns the name of the agent created. Basically response is provided back to Unity3D. Creating an agent from Unity3D is straightforward, consider the script below:

```
string response = GameManager.api.createAgent (agentName, "Player");
```

For removing an agent the API requires only the name of the agent. Similar to the creation a response from API can be used to validate if the agent was terminated successfully. Name of the method - *removeAgent*.

5.1.2 Send and receive events

The API accepts events (both synchronous and asynchronous) with arguments - the type of the event and the event value in json format. Based on the type of the event a relevant API implementation is used. Four types of events supported: position, collision, position_vector and messaging. The structure of the json needs to be unified across both, ASTRA and Unity3D, simple class created in Unity3D to address the json serialization and deserialization:

```
[Serializable]
public class AstraJson {
   public Position position;
   public Scale scale;
   public Rotation rotation;
   public string type;
   public int instanceId;
   public string cardinalDirection;
   public string astraCardinalDirection;
   public string message;

   public AstraJson (){}

   public AstraJson (Transform transform){
      this.position = new Position(transform);
      this.scale = new Scale(transform);
      this.rotation = new Rotation(transform);
   }
}
```

When event is send to ASTRA three parameters are passed - *agentIdentifier*, *eventIdentifier* and *eventArgs*. Each event is bound to an agent, placed on a queue and processed in a separate thread in ASTRA. Sending events from Unity3D can be done programatically in many different ways, however Unity3D provides a nice way to manage delays and frequency of the call with the method API *InvokeRepeating(string methodName, float time, float repeatRate)*. Invokes the method *methodName* in time seconds, then repeatedly every *repeatRate* seconds. So, in the example below, it will call *UpdateAgentPosition* two seconds after the *InvokeRepeating* call and then every 0.5 second thereafter. This is how normally you should call your method responsible with sending updates to ASTRA (Unity3D transform object consists of Position, Scale and Rotation):

```
InvokeRepeating ("UpdateAgentPosition", 2.0f, 0.5f);
.............................................
private void UpdateAgentPosition () {
   AstraJson directions = new AstraJson (this.transform);
   directions.type = GameManager.EVENT_POSITION;

   string json = JsonUtility.ToJson (directions);
   GameManager.api.asyncEvent (agentName, GameManager.EVENT_POSITION_VECTOR,
      new object[] { json });
}
```

In Unity3D, the events normally are received inside the *Update()* or *FixUpdate()* methods, such a way the game object's - Position, Scale or Rotation attributes can be altered simultaneously inside the same frame. Main difference between both *Update()* - runs once/every frame, where frame to frame time is vary and *FixUpdate()* - can run once, zero, or several times per frame, depends on how many physics frames per second are set in the time settings, every fixed frameRate frame, regular time-line. Consider a use of the API *receive* method:

```
string response = GameManager.api.receive(agentName,
   GameManager.EVENT_POSITION);
```

Both synchronous and asynchronous events use the *receive* API method, one directly from Unity3D code (example from above, synchronous), the other one in the background when the API is called (asynchronous).

5.1.3 Deal with collisions

To listen for collision, either *OnCollisionEnter()* or *OnTriggerEnter()* can be used. In both cases, cardinal direction (North, East, South or West)can be obtain, depending where the Avatar is looking at - assuming it is on the XZ plane with positive Z as North. For this to work we do need to make the game Object (Avatar) face its movement direction. In Unity3D, the forward of any Game Object is considered the Z axis. If you want *transform.forward* to be in the same direction as your Avatar, just ensure whatever your Avatar front facing direction is along the Z axis.

```
this.transform.rotation = Quaternion.LookRotation(movement);
this.transform.Translate (movement * movementSpeed * Time.deltaTime,
    Space.World);
```

If this is the case and the Avatar always facing the direction of movement, it will be easy when collision occurs to find out from which direction the Avatar came, this information is needed by the API and its used to avoid returning the same direction for the same collision if happens again. This is how its done in Unity3D:

```
// Get a copy of the forward vector
Vector3 forward = this.transform.forward;
// Zero out the y component of the forward vector to only get the direction
    in the X,Z axes
forward.y = 0;
forward.Normalize();
```

```
//get the cardinal direction of the collision
string cardinalDirection = null;
if (Vector3.Angle(forward, Vector3.forward) <= 45.0) {
   cardinalDirection = "North";
} else if (Vector3.Angle(forward, Vector3.right) <= 45.0) {
   cardinalDirection = "East";
} else if (Vector3.Angle(forward, Vector3.back) <= 45.0) {
   cardinalDirection = "South";
} else {
   cardinalDirection = "West";
}
```

Second option is to use 'buffer' objects attached to the Avatar, as its done in all prototypes, such a way the cardinal direction is predefined per object, so no need to obtain the direction Avatar was coming from before the collision.
ASTRA API has record of all collisions and decision it took for the new direction after the collision occurred. This is done per agent for each object it collides. Moreover, for the same object hit from the same side, ASTRA API never will reply back with the same new direction, until all three possibilities are available. In short hitting the same obstacle multiple time it will response every time with a different new cardinal direction.

5.2 Concept of ASTRA API event queue size close to zero

Event queues act as a buffer between the subsystem sending data and the subsystem receiving it. In ASTRA language each agent has its own event queue with an unique event signature, thus allows to associate event to an agent. Internally the language use the scheduling mechanism to process these events in a fair fashion, by preserving the order. These will guarantee the agent will act on each event as its arrives, based on its arrival order.

In other hand after the agent processed the event, the ASTRA API method *submitCommand* is invoked by the agent and processed event, ready to be send to Unity3D is placed in a concurrent map data structure implemented in the API. This map is used by the API method *receive* to obtain the correct agent's response and send it to Unity3D on request. The API implementation addresses all stated above by using the strategy for preventing thread interference, in short ASTRA API is thread safe.
What could go wrong? There is a scenario that need to be considered - imagine if, Unity3D fires events so often that the ASTRA API cannot handle it. Both systems need to be synchronized.
As the events are received in Unity3D from within the *Update()* method and its runs once per each frame, where in the other hand the update is send with the help of *InvokeRepeating()* method. The default frame rate in Unity3D depends on the plaform, for example for mobile devices is 30 frames per second. The problem can occur if more than 30 event updates are send per second, which can be done by setting the last argument, the *repeatRate* in the *InvokeRepeating()* method to 0.02. This means every 20 milliseconds an update will be send from Unity3D to ASTRA API, or in total 50 events in a second. As the frame rate is set to 30 frames per second, in an ideal situation 20 events will still remain in the ASTRA event queue. Only the first 30 will be served, where the rest 20 will be transmitted in the next cycle (in the next second). This scenario will build up a very huge queue rapidly and is something that needs to be avoided. In an ideal world the Unity3D should be configured to fire and receive events, such that, the event queue per agent is close to zero, we say in this case the agent is fully utilized. This can be achieved by carefully setting the parameters for *InvokeRepeating()* method with regards to how many frames per second are set in Unity3D.

5.3 Unity3D and IKVM

Important part of this work is the ability to convert the core API - *astra-api.jar* to a dll assembly and bring it into .NET environment. IKVM.NET is a Java Virtual Machine (JVM) for the .NET and Mono runtimes. It was developed primarily by Jeroen Frijters.

Java and .NET are considered as mutually exclusive technologies, but IKVM.NET enables the unique possibility of bringing them together. It is a sophisticated collection of tools offering a variety of integration patterns between the Java and .NET languages and platforms.[7] It has full support for reflection, as well as JNI. IKVM use the GNU Classpath project, and therefore has as much support for the standard Java libraries as GNU Classpath. GNU Classpath is a project aiming to create a free software implementation of the standard class library for Java. Thus, IKVM today can run many complex server-side Java libraries. IKVM.NET consists of a compiler that can transform Java bytecodes into .NET dll representation. It includes the Java standard libraries, in the form of a version of GNU Classpath, pre-compiled into .NET. It also includes JNI providers for Mono and Windows, this enables Java code to access native C/C++ libraries.[8]
IKVM.NET plays a significant role in this approach of connecting ASTRA with Unity3D game engine, without the existence of this tool it would not be possible to make this connection so easily. From all of the available components, the option of converting a Java application to .NET was used, see Figure A.8 for detail explanation how IKVM.NET is used to convert jar to dll. IKVM.NET is not the only alternative, there are many tools that can be used to achieve this goal - JNBridge[14], IKVM.NET was chosen over the rest, because it is an open source project and its simplicity. Build guide explains everything that needs to be done in details in Appendix A.3.1 also deploying in Unity3D is shown step by step in Appendix A.4.

5.3.1 Issues

One thing to bear in mind, when game is build in Unity3D the following exception can be thrown:

```
ArgumentException: The Assembly System.Drawing is referenced by
    IKVM.OpenJDK.Media ('Assets/Assets/IKVM.OpenJDK.Media.dll'). But the dll
    is not allowed to be included or could not be found.
```

If this happens API compatibility needs to be changed in order to use external assemblies (dll). Do the following in Unity3D:
In Build Settings → Player Settings → API Compatibility Level, change the .NET setting.
Under Other settings for PC / Mac change the API Compatibility Level from ".NET 2.0 Subset" to ".NET 2.0" Recompile and build again, the error should goes away.

5.4 Maven as a building tool

Maven[13] is a project management tool that is based on project object model (POM). It is used for projects build, dependency and documentation. It simplifies the build process. Some of the Maven benefits are: standardized naming scheme for project dependencies; supports dependency management and provides tools for managing the complexity inherited dependencies; forces a standard directory structure; reduce the size of source distributions - jars can be pulled from a central location. See Appendix A.3.2 for detail explanation how maven is used for this project.

Chapter 6

Unity3D Prototypes

As the development of the API evolved, number of prototypes were built, to confirm the correctness of the API and to evaluate the performance. Building these prototypes was crucial in making the API fit for purpose. The prototypes answered the questions:

Validation: Are we building the right product? The API meets the application's/integration needs. To show the bridge between ASTRA and Unity3D is possible, API was built, by exposing seven public methods. These methods provide needed functionality to satisfy the requirements for showing the bridge between ASTRA and Unity3D is possible.

Verification: Are we building the product right? API was built according to the best standards and specifications. It follows very strictly the requirements outlined by Joshua Bloch in his book - "Effective Java Programming Language Guide"[12]. He led the design and implementation of number of Java API, including the Java Collections Framework, the java.math package, and the assert mechanism. Evident of that is that the integrated API solution works with Unity3D.

6.1 Roll a ball - Prove of concept

Before even starting building the API, prove of concept was required that the approach taken, using IKVM.NET will work at all. For this purpose most popular Unity3D game for beginners was created and used - Roll a ball. A simple rolling ball game that teaches many of the principles of working with Unity3D.
All ASTRA source code needed was compiled, bundled in a jar file and converted to dll libraries using IKVM.NET. See the output produced by IKVM - Figure A.10. The entire content of the bin folder was placed under the Unity3D project Assets folder.
The game itself is simple, it has number of collectible cubes constantly rotating and a rolling ball collecting them. Initially ASTRA API had implemented an agent creation functionality and a simple logic based on the four direction - Up, Down, Left and Right to return a new direction vector in a format of x,y and z axes, with random values. A synchronous event was implemented as the only functional method in the API to support transferring the data between ASTRA and Unity3D. Consider the code snippet below for synchronous communication and logic for new direction:

```
public String syncEvent(String agentIdentifier, String eventIdentifier,
    Object[] args) {
   asyncEvent(agentIdentifier, eventIdentifier, args);
   String json = null;
   while ((json = receive()) == null) {
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {
         e.printStackTrace();
      }
   }
   return json;
}
..............
```

```
@TERM
public String getDirections(String position) {
   Map<String, Object> result = new HashMap<String, Object>();
   Random rand = new Random();
   if (position.equals(DOWN)) {
      result.put("horizontal",(rand.nextFloat()*(max-rand.nextFloat())+min));
      result.put("vertical", 1.0f);
   } else if (position.equals(UP)) {
      result.put("horizontal",(rand.nextFloat()*(max-rand.nextFloat())+min));
      result.put("vertical", -1.0f);
   } else if (position.equals(LEFT)) {
      result.put("horizontal", 1.0f);
      result.put("vertical",(rand.nextFloat()*(max-rand.nextFloat())+min));
   } else if (position.equals(RIGHT)) {
      result.put("horizontal", -1.0f);
      result.put("vertical",(rand.nextFloat()*(max-rand.nextFloat())+min));
   } else {
      result.put("horizontal", 1.0f);
      result.put("vertical", 0.5f);
   }
   return gson.toJson(result);
}
```

In Unity3D on collision an information for the direction *gameObject* collided was sent and ASTRA was returning the new direction, thus the ball was rolling randomly and eventually was able to collect all cubes, which was the goal of the game. A json object returned contains the coordinate axes, mapped to a C # *Directions* in Unity3D. The json is deserialized in Unity3D before use. This is an example how in Unity3D the event is passed and the result from ASTRA is consumed:

```
private void SetPosition (string update) {
    string directionsFromAstra = apiAstra.syncEvent ("position", new object[]
        {update});

    Directions directions = new Directions ();
    directions = JsonUtility.FromJson<Directions> (directionsFromAstra);

    moveHorizontal = directions.horizontal;
    moveVertical = directions.vertical;
}
```

Simple, straight forward game, but was a prove that the approach of bringing ASTRA to Unity3D using IKVM.NET was working. What was confirmed with this prototypes was that ASTRA language can be executed and run within the Unity3D environment using the Java Virtual Machine (JVM) implemented in .NET. The work can continued for developing fully integrated API.

6.2 Maze game with fully integrated API

As the development of the API evolved and reached a stage where all public methods and basic events were supported a more complex game was created. A maze/labyrinth game was built with a single player (*gameObject*) and two exit point in a form of spinning coins. See Figure B.2 for the game screen shot. The aim of building this prototype was to show, that different and constant updates can be handled for all obstacles in the maze, that ASTRA is capable of responding fast and accurate. Basically to show that the API is fit for purpose.
Four supporting *GameObjects* were added/attached on each side of the Avatar - the main *GameObject*. This prevented the Avatar going into the walls on collision. The purpose of these supporting *GameObjects* was to act as a 'car bumper' - to detect the collision before the Avatar himself collides, and to have the responsibility for updating particular cardinal direction - North, South, West or East, depend on which side of the Avatar it

was attached. There is no direct communication between the four supporting *GameObjects* and the Avatar, they act on behalf of it by updating directly the collision events to ASTRA.
How it works? Two main concept: movement and collision. Two event types from the API were used for this prototype, *position_vector*, as well the *collision* event.

Movement: The Avatar sends asynchronous events every 0.1 second with the current position using *InvokeRepeating* functionality in Unity3D for method invocation - see section 5.1 for detail explanation. Inside Unity3D's *Update* method we do listen for response from ASTRA and if such exist the new direction vector is applied to the Avatar. Consider the following Unity3D script:

```
void Update () {
    //listen for response from Astra
   string positionVectorFromAstra =
       GameManager.api.receive(agentName,GameManager.EVENT_POSITION_VECTOR);

   if (positionVectorFromAstra != null && !isCollided) {
      AstraJson
          positionVector=JsonUtility.FromJson<AstraJson>(positionVectorFromAstra);
      Vector3 movement=new
          Vector3(positionVector.position.x,0.0f,positionVector.position.z);
      this.transform.Translate(movement*moveSpeed*Time.deltaTime,Space.Self);
   }
}
```

In this prototype the *'position_vector'* was used and therefore the Unity3D's transform was used to apply physics to the Avatar and make it move. See section 5.1 for distinguishing between position direct assignment and applying physics to a Game Object.

Collision: For detecting collisions and updating the *collision* event in ASTRA, the *GameObjects* attached to the Avatar were responsible. For each collision occurred the responsible *GameObject* send synchronous event to ASTRA, once the response was received with the new cardinal direction from ASTRA the 'bumper' *GameObject* aligns the new direction with the Avatar by invoking a method to send an update on behalf of it. Consider the script below for the 'bumper' *GameObject* attached to the South side of the Avatar, where *PlayerMovement* is an Unity3D script (class name) attached to the Avatar:

```
public PlayerMovement avatar;
...
void OnCollisionEnter (Collision collision) {
   avatar.currentCollider = GameManager.CARDINAL_DIRECTION_SOUTH;
   avatar.isCollided = true;

   AstraJson collisionDirection = new AstraJson (this.transform);
   collisionDirection.instanceId = collision.collider.GetInstanceID ();
   collisionDirection.cardinalDirection =
      GameManager.CARDINAL_DIRECTION_SOUTH;
   collisionDirection.type = GameManager.EVENT_COLLISION;

   string collisionDirectionJson = JsonUtility.ToJson (collisionDirection);
   string collisionFromAstra = GameManager.api.syncEvent (avatar.agentName,
      GameManager.EVENT_COLLISION, new object[] { collisionDirectionJson });

   AstraJson collisionResponse = JsonUtility.FromJson<AstraJson>
      (collisionFromAstra);

   //align the Avatar with the new direction
   avatar.UpdateAgentInitialVectorDirection
      (collisionResponse.astraCardinalDirection);
}
```

```
....................................
json send to ASTRA:
{"type":"collision","x":1.7858673334121705,"y":0.5019882321357727,"z":-1.017604947090149,
"instanceId":9962,"cardinalDirection":"South","astraCardinalDirection":""}
```

In the sample json from above *instanceId* is the Object id with which the Avatar collided and *cardinalDirection* - from where the Avatar came. As the direction sent back from ASTRA is different for the collisions with same *instanceId* and *cardinalDirection*, this prevents the Avatar to have the same new direction of movement if collides with the same wall again coming from the same direction (see section 4.3.2 for detail explanation). That way the Avatar is able eventually to reach the final destination in the Maze - the two spinning coins.

6.3 Maze with ASTRA AI

For this prototype more complex maze was created. Number of agents were deployed to walk freely around the maze and build a knowledge map by updating the position and obstacles they hit. The Avatar - main *GameObject* enters the game and relies on the knowledge gathered from the other agents to find the end of the maze.
The prototype built combines the concept of collaborative agents (Multi-Agent System), building a knowledge and using this knowledge to enable agent to operate autonomously - Artificial Intelligence (AI). See Figure B.3.

6.3.1 Build knowledge

In ASTRA Module called *GridMap* was implemented, see UML Class diagram in Figure A.5. A method *updateGridMap* was used to record each move made by the agent/s, thus a knowledge map was build. Position and collisions are processed, where when collision occurs the blocked side in the cell is recorded as well. For every collision event, a check was performed to find out if the end of the maze was reached. This was done by asserting an *instanceId* of the Unity3D *GameObject*.

```
@ACTION
public boolean updateGridMap(String eventType, String event) {

   //get current collision/position passed from Unity
   UnityJson requestFromUnity = gson.fromJson(event, UnityJson.class);
    Position position = requestFromUnity.getPosition();

     //if instanceId is passed and match the expected one
     //set the end flag to true, means the end of maze was reached
     int instanceId = requestFromUnity.getInstanceId();

     boolean isEnd = false;
     if (instanceId == endInstanceId) {
        isEnd = true;
     }
     //which direction is blocked in the cell, only if its a collision event
         record the data
     String cardinalDirection = eventType.equals(EventType.COLLISION) ?
         requestFromUnity.getCardinalDirection() : null;

     //store coordinates as integers
     Coordinates coord = new Coordinates((int) Math.round(position.getX()),
         (int) Math.round(position.getZ()), cardinalDirection, isEnd);
     breadCrumbs.add(coord);

     return true;
}
```

ASTRA requires every agent to have an unique name, for the purpose of this prototype agent was called 'smartyPans'. The name of the agent is important as in ASTRA in order to send a message to another agent, the agent name is needed, so it needs to match with the one created in Unity3D. In ASTRA collision and position events were overridden, thus the normal use of the API was preserved, with addition that before processing the event information the knowledge map was build and once the condition was met message was sent to trigger the 'smartyPans' agent.

```
rule +!collision(string collision, string event) {
    int count = gridMap.getFinishCount();
   if (count > 1) {
       !doNotify(finishCount(count));
   } else {
     //update on map
     gridMap.updateGridMap(collision, event);
   }
    Player::!collision(collision, event);
}
```

6.3.2 Find path

Prerequisite: The maze has certain dimensions - 20x20, all walls are positioned exactly one unit apart from each other. The coordinates are represented as an Integer - whole number in the ASTRA side, e.g. in Unity3D the horizontal walls will be located always with Z axis on 0.5, 1.5... half of a whole number. Respectively the vertical walls are on X axis with half of whole number as well. When data is passed to ASTRA the decimal number is rounded to a whole number. Why its done that way? As the maze is represented as a plane, its 'converted' to a grid with a 1x1 cells in regards to the ASTRA. Representation of a grid in ASTRA is more convenient to find the path. The data passed from Unity3D is in its standard form - a json containing: position, scale, rotation and optional parameters: cardinal direction and instance id. For both, spinning coins - end

of the maze, *instanceId* assigned was 7777777, which eventually will trigger ASTRA in the background to send a message once enough data is collected and the Avatar can start moving.
The two agents, so called 'workers' start to collect the data from the same position where the Avatar will start its adventure finding the end of the maze. On starting the game the Avatar is hidden and will become active once the 'workers' complete them job. The algorithm entirely relies on the data gathered from the 'workers'. As more data we have as preciser the algorithm will be. How it works?

The same Module that builds the knowledge is responsible for triggering the algorithm execution. As mentioned above, on every collision recorded in ASTRA we check if both end of the maze ('spinning coins') were reached. Consider the ASTRA code snippet:

```
rule +!doNotify(finishCount(int X)) : X == 2 & ~state("Agree") {
   send(request, "smartyPans", taskPathReady("Ready"));
}
```

When update is sent from Unity3D, no matter what the current position is, ASTRA returns the next step 'smartyPans' needs to make in order to reach the end of the maze. Normally for position update method with a name *getDirection* is used, here we called it *getNextMove*. The very first time the method is invoked a new instance of *PathFined* is created - the class responsible with finding the path. The path is generated and stored in *LinkedHashSet* - no duplicated values and preserved order, this is done on constructor invocation. Coordinates POJO holds all information for each step recorded by the workers, its consists of x,y as an integers, direction as a string and Boolean blocked and finish. So one position/step can have up to five related Objects - e.g

```
Coordinates [x=10, y=8, direction=, blocked=false, finish=false]
Coordinates [x=10, y=8, direction=East, blocked=true, finish=false]
Coordinates [x=10, y=8, direction=North, blocked=true, finish=false]
```

Coordinates with same x and y simply represent a cell with four sides, where for each side we can have a blocked direction and flag finish for the end of the maze. For clarity the supporting Coordinates POJO holds coordinates as x and y, where y is translated to z in Unity3D.

The Algorithm:

**path* is the holding data structure - LinkedHashSet

The algorithm looks rather complex, but imagine the agent is moving between two walls towards South, for all these cells it would have only one related Coordinate Object per cell, so no information for blocked directions. The algorithm relies heavily on the initial direction of movement, as we do not have all the data of the grid.

In Unity3D with this prototype, the position is used directly for the first time, so far always physic was applied:

```
Vector3 movement = new Vector3 (positionInitialVector.position.x, 0.0f,
    positionInitialVector.position.z);
this.transform.Translate (movement * moveSpeed * Time.deltaTime, Space.Self);
```

where now position is assigned directly:

```
Vector3 movement = new Vector3 (positionVector.position.x, 0.0f,
    positionVector.position.z);
this.position = movement;
```

With this prototype a communication between ASTRA and Unity3D was achieved, with

ASTRA initiating this communication. As shown above a message was sent once conditions were met. The message sent from ASTRA use the 'messaging' event type to propagate it to the Unity3D. How it works? Unity3D listen for an event type 'messaging' inside the *Update* method, once message is received, depend on its content appropriate action is taken.

```
string message = GameManager.api.receive(agentName,
    GameManager.EVENT_MESSAGE);
if (message != null) {
  AstraJson messageResponse = JsonUtility.FromJson<AstraJson> (message);
  if (messageResponse.message.Equals("Ready")) {
  .......
```

The interaction between ASTRA and Unity3D obeys the principal of 'send - receive' - message driven communication. This example shows the flexibility of using different approach of communication, in essence ASTRA initiates and dictates the communication, where no need to receive information in order to response, simply can fire a message anytime.

Also the prototype clearly shows that multiple number of agents can coexist and interact with Unity3D environment at the same time. Multi-Agent System is an alternative to a centralized problem solving, where agents cooperatively exchanging information as the solution is developed. This was shown clearly with this last prototype, agents working together towards common goal.

Algorithm 1 (PathFinder) Find path to the end of the maze

```
if path contains the new coordinates then return
end if
if end of the maze is reached then return
end if
1. Add to the path the current coordinate.
2. Check the same direction of movement - from all four cardinal direction.
for all path do
    1. Get next coordinates in the direction of movement, basically if we are on (10,10) it was already
added to the path, now we are trying to go one step further. If it was South, the new coordinate
would be (10,9).
    2. Check if for the new coordinate the current direction of movement is blocked. Basically the
logic is - you arrive on (10, 9), but (10,9) can have up to five entries: without blocked flag and blocked
for each direction, so as the Agent is here we need to decide where to go, can he continue the same
direction? This is the question we are trying to answer here.
    if new coordinate is blocked for the current direction then
        call changeDirectionOfMovement
    else
        recursive call
    end if
end for
    .............................................
changeDirectionOfMovement
1. Find all coordinates with the same x and y.
2. Get all blocked directions, at least the current direction and the one from where it comes will be
marked as blocked.
3.Get random index for the remaining cardinal direction and make a recursive call, with new direction
of movement.
```

Chapter 7

Testing

7.1 Test-driven development

For the implementation and development of the ASTRA API the Test-driven development process was employed. Test-driven development (TDD) is a software development process that relies on development cycle repetitions: you write your test first and after that you write the functional/actual code to fulfill the requirements from the test, refactor the new code to acceptable standards, repeat the same steps. [6]
This approach helps us to think always first for the specification, make sure we are developing the correct functional wise API. Developing in this manner gave us a better sense of progress, continuous feeling of achievement, help us to think through API requirements and design before writing any functional code, reduced the failures.

7.2 Unit testing

A unit is the smallest possible software component, it performs a single cohesive function. Having unit tests in place help us to reveal defects, to locate them and to repair them easy. The API was written in Java, therefore the obvious choice for us was to use JUnit testing framework to implement our unit testing. Tests written were separated in three main categories, different areas of the API were covered:

- General API tests - covered creation and removal of an Agent.
- Events API tests - multiple tests per each event's type API supports (collision, position, position_vector and messaging).
- ASTRA Modules tests - ensures all logic provided by ASTRA Modules was correct.

In total 75 test cases were written. For some of the tests, most of the event's types unit tests we used parameterized tests, thus we are allowed to run the same test over and over again using different values, we avoided writing duplicate coded and we achieved extensive coverage. The table below shows how tests are spread around different components of the API, what coverage we have achieved. Also see Figure A.7

Component	Test Class name	Tests	Parameterized Tests
Event	MessageEventTypeTest	4	-
Event	PositionEventTest	14	56
Event	PositionVectorEventTest	11	33
Event	CollisionEventTest	12	-
Modules	GridMapGeneratorTest	23	-
Modules	SmartyPansMovingTest	3	-
Common	AstraApiImplTest	6	-

Figure 7.1: Test cases coverage.

7.3 System and Integration testing

The best way to test the API is to develop Unity3D games. For this purpose number of prototype games were created. With help of these games system integration testing activities were performed during the development of the API. Agile testing approach was employed for testing the API, as most of the test effort was on-going during the

development process. Some unexpected results and problems occurred during the testing, which help us to improve the code and find the right solution. Several test cases were established and they were performed repetitively during this implementation. Prototypes addressed different aspects of the API functionality and capabilities. Games with single and multiple agents were developed. Games with and without interaction between the agents.
Number of game prototypes were build for the last maze, where we had agents collecting data in order this data to be used for navigating thought the maze by another agent. Game with up to seven 'workers' agents was created to speed up the process of gathering the data, where we had a free path and where the path to the end of the maze was blocked. No latency was observed, introducing more agents made the process much faster.

7.4 Quality of code

For testing quality of the code PMD was integrated to Eclipse IDE for analyse efficiency and quality of the code. It finds unused variables, empty catch blocks, unnecessary object creation, and so forth. This is simply programming mistake detector. PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements.
- Dead code - unused local variables, parameters and private methods. Suboptimal code - wasteful String/StringBuffer usage.
- Over complicated expressions - unnecessary if statements, for loops that could be while loops.
- Duplicate code - copied/pasted code means copied/pasted bugs.

After scan completion it generates reports which are placed in the project directory , based on the reports we improved the code. Great tool, help a lot to develop skills as a programmer, to avoid making silly mistakes and to write clean and readable code.

Chapter 8

Conclusions and Future Work

A novel approach was presented for connecting ASTRA Agent-Oriented Programming (AOP) with Unity3D game engine, in essence a bridge between AI and VR/AR world using software agents. The work reported here provides a starting point for establishing if the AOP paradigm is useful to develop immersive VR/AR world with embedded intelligence.

The connection between ASTRA and Unity3D is made at a machine level, thus it provides an efficent communication mechanism between both without any delay or latency. I believe the implementation of this direct approach makes it possible to connect any Java-based agent platform with relatively little effort to C/C++ environments which opens many opportunities for exploring AOP paradigm within the VR/AR world.

Number of simple game prototypes were built to verify the correctness and show how the API can be used. These prototypes helped for evaluating and improving the performance during the development.

Agent based systems have the potential of building complex, distributed software systems. The API provides a platform for building Unity3D games with more complex collaboration between agents embodied into Unity3D avatars. Building complex games by adopting the Multi-Agent System (MAS) approach is proven to be successful. Multi-Agent System approach within Unity3D can be explored fully using ASTRA by building systems with agents interacting on both levels - through the underlying ASTRA and propagating the decisions to Unity3D environment and also communication at Unity3D level where single agent is triggered to act and via Unity3D collaborates with the rest of the agents in the system.

The API can be used in many different scenarios within Unity3D world, it could expand to cover different event's type. The current work provides the ability to react and modify the position and collision of the Unity3D avatars, and passing a message to and from ASTRA. An important concept in Unity3D is the rotation and the scalability of the game Objects. The API already supports the transfer of this information from and to Unity3D. In ASTRA relevant Modules can be build and applicable logic to be attach to them, thus rotation and scale Unity3D attributes can be accommodated.

Appendix A

ASTRA API Implementation

A.1 UML class diagrams

All ASTRA Api and relevant ASTRA Modules class diagrams are shown below.

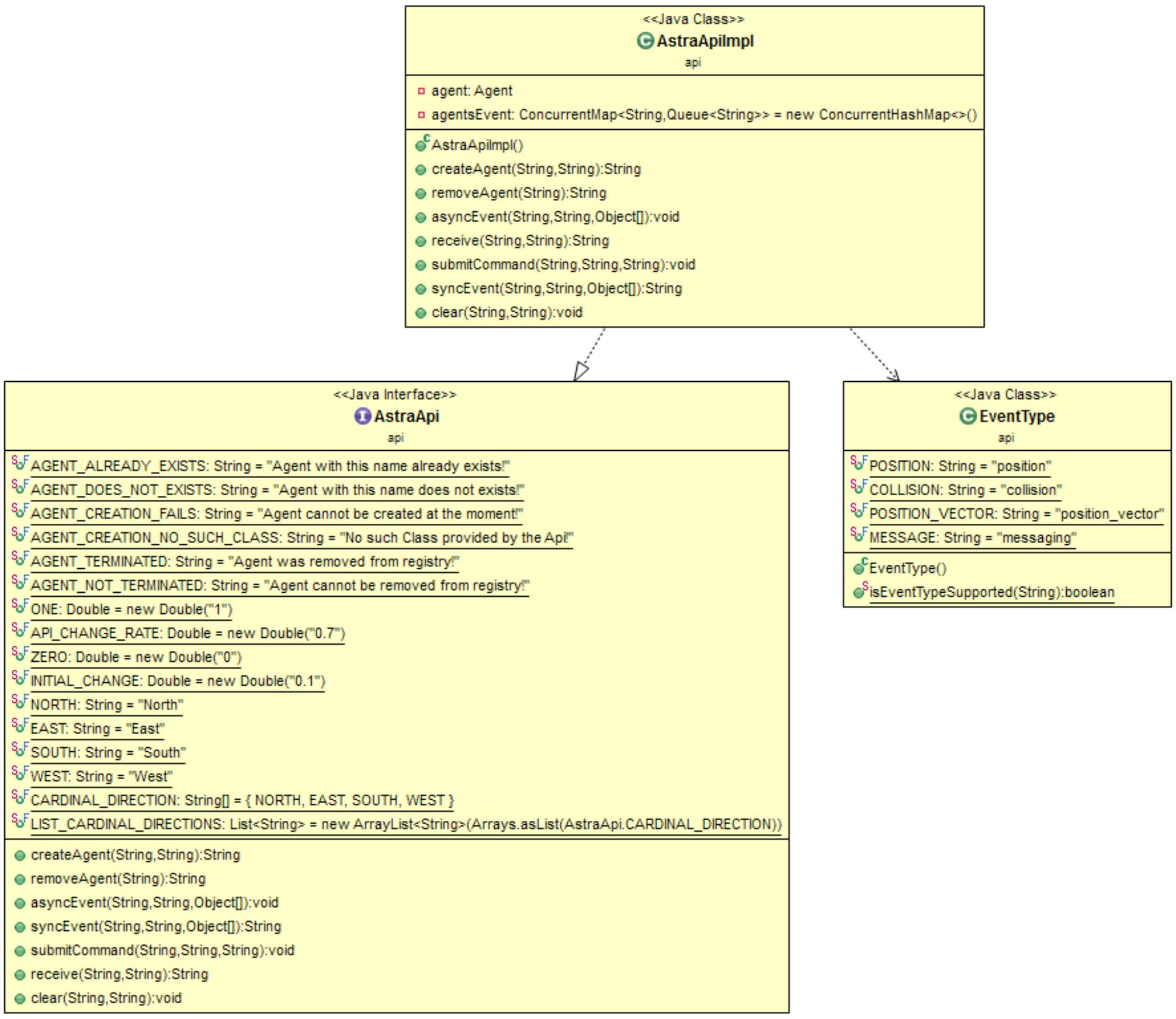

<<Java Class>>
AstraApiImpl
api
agent: Agent
agentsEvent: ConcurrentMap<String,Queue<String>> = new ConcurrentHashMap<>()
AstraApiImpl()
createAgent(String,String):String
removeAgent(String):String
asyncEvent(String,String,Object[]):void
receive(String,String):String
submitCommand(String,String,String):void
syncEvent(String,String,Object[]):String
clear(String,String):void
<<Java Interface>>
AstraApi
api
AGENT_ALREADY_EXISTS: String = "Agent with this name already exists!"
AGENT_DOES_NOT_EXISTS: String = "Agent with this name does not exists!"
AGENT_CREATION_FAILS: String = "Agent cannot be created at the moment!"
AGENT_CREATION_NO_SUCH_CLASS: String = "No such Class provided by the Api!"
AGENT_TERMINATED: String = "Agent was removed from registry!"
AGENT_NOT_TERMINATED: String = "Agent cannot be removed from registry!"
ONE: Double = new Double("1")
API_CHANGE_RATE: Double = new Double("0.7")
ZERO: Double = new Double("0")
INITIAL_CHANGE: Double = new Double("0.1")
NORTH: String = "North"
EAST: String = "East"
SOUTH: String = "South"
WEST: String = "West"
CARDINAL_DIRECTION: String[] = { NORTH, EAST, SOUTH, WEST }
LIST_CARDINAL_DIRECTIONS: List<String> = new ArrayList<String>(Arrays.asList(AstraApi.CARDINAL_DIRECTION))
createAgent(String,String):String
removeAgent(String):String
asyncEvent(String,String,Object[]):void
syncEvent(String,String,Object[]):String
submitCommand(String,String,String):void
receive(String,String):String
clear(String,String):void
<<Java Class>>
EventType
api
POSITION: String = "position"
COLLISION: String = "collision"
POSITION_VECTOR: String = "position_vector"
MESSAGE: String = "messaging"
EventType()
isEventTypeSupported(String):boolean

Figure A.1: ASTRA API - Class diagram.

Figure A.1 shows the UML-Class diagram of ASTRA API - All public methods available.

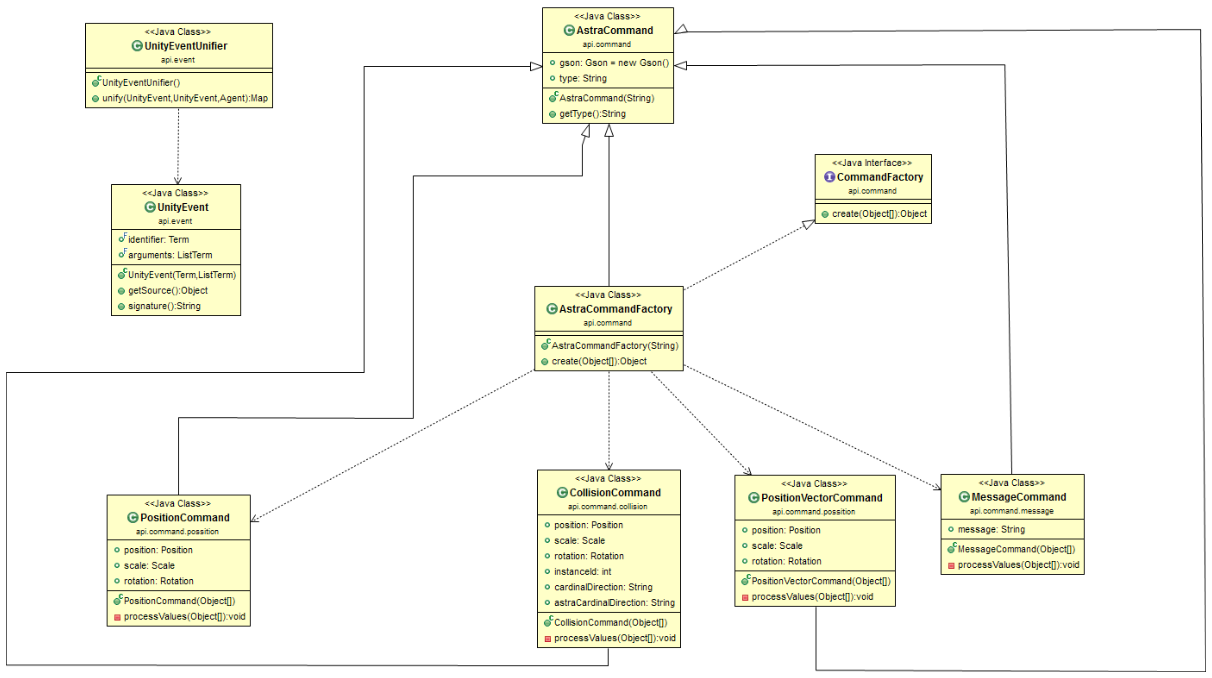


Figure A.2: ASTRA API - events factory method pattern.

Figure A.2 shows the UML-Class diagram of ASTRA API events factory method pattern.

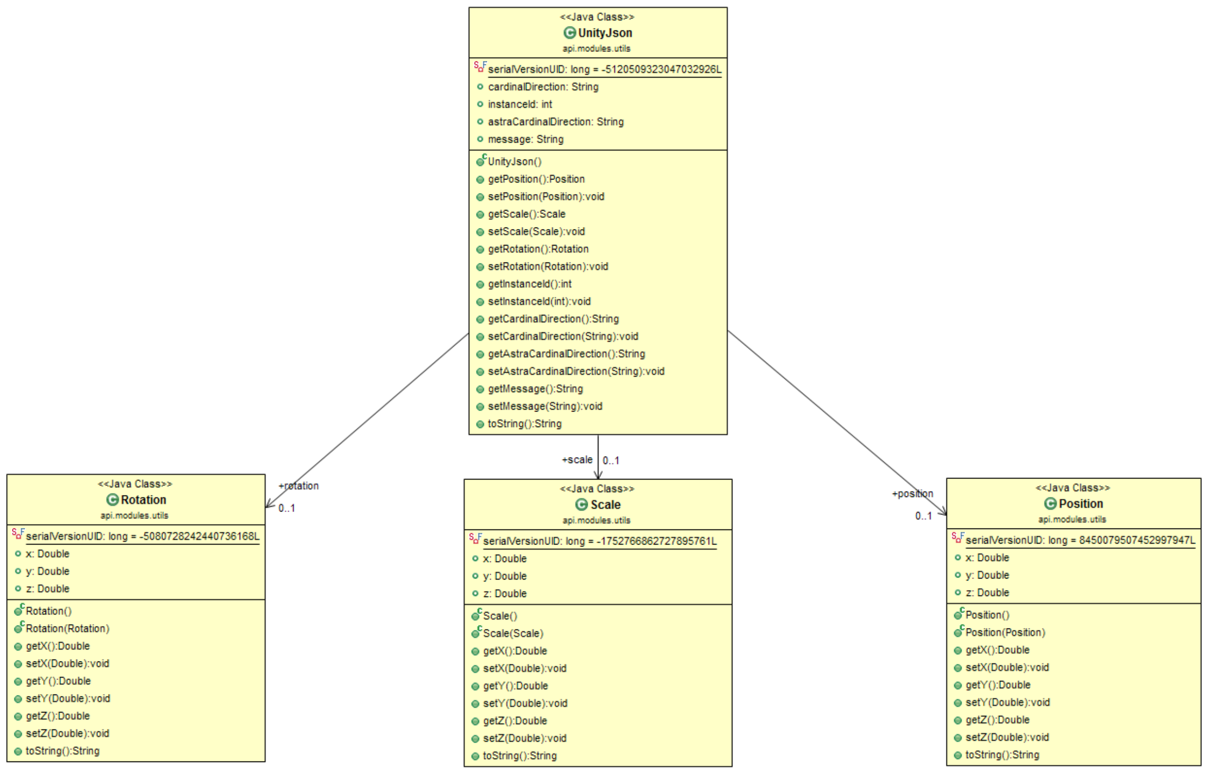
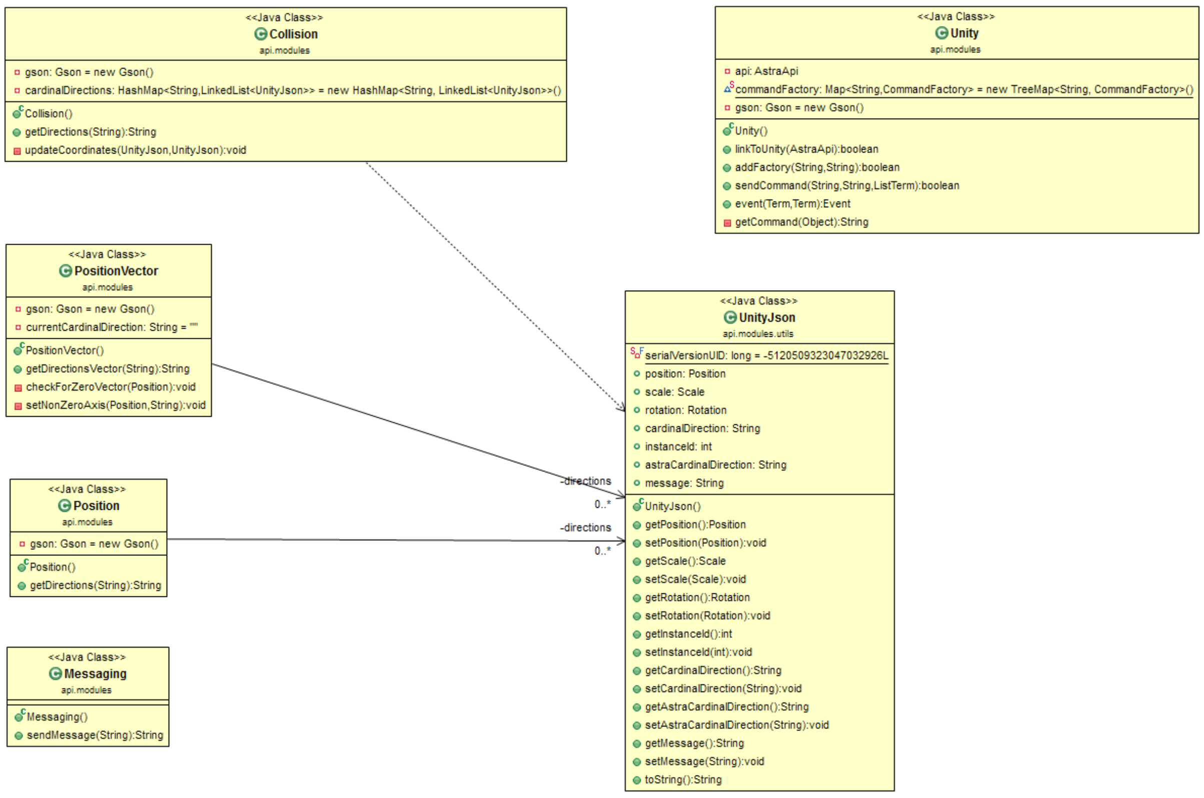


Figure A.3: ASTRA API - JSON support structure.

Figure A.3 shows the UML-Class diagram of JSON support structure. This is the JSON structure used for communication between ASTRA and Unity.



<<Java Class>>
Collision
api.modules
gson: Gson = new Gson()
cardinalDirections: HashMap<String,LinkedList<UnityJson>> = new HashMap<String, LinkedList<UnityJson>>()
Collision()
getDirections(String):String
updateCoordinates(UnityJson,UnityJson):void
<<Java Class>>
Unity
api.modules
api: AstraApi
commandFactory: Map<String,CommandFactory> = new TreeMap<String, CommandFactory>()
gson: Gson = new Gson()
Unity()
linkToUnity(AstraApi):boolean
addFactory(String,String):boolean
sendCommand(String,String,ListTerm):boolean
event(Term,Term):Event
getCommand(Object):String
<<Java Class>>
PositionVector
api.modules
gson: Gson = new Gson()
currentCardinalDirection: String = ""
PositionVector()
getDirectionsVector(String):String
checkForZeroVector(Position):void
setNonZeroAxis(Position,String):void
<<Java Class>>
UnityJson
api.modules.utils
serialVersionUID: long = -5120509323047032926L
position: Position
scale: Scale
rotation: Rotation
cardinalDirection: String
instanceId: int
astraCardinalDirection: String
message: String
UnityJson()
getPosition():Position
setPosition(Position):void
getScale():Scale
setScale(Scale):void
getRotation():Rotation
setRotation(Rotation):void
getInstanceId():int
setInstanceId(int):void
getCardinalDirection():String
setCardinalDirection(String):void
getAstraCardinalDirection():String
setAstraCardinalDirection(String):void
getMessage():String
setMessage(String):void
toString():String
-directions
0..*
<<Java Class>>
Position
api.modules
gson: Gson = new Gson()
Position()
getDirections(String):String
-directions
0..*
<<Java Class>>
Messaging
api.modules
Messaging()
sendMessage(String):String

Figure A.4: ASTRA Modules - general.

Figure A.4 shows the UML-Class diagram of ASTRA Modules.

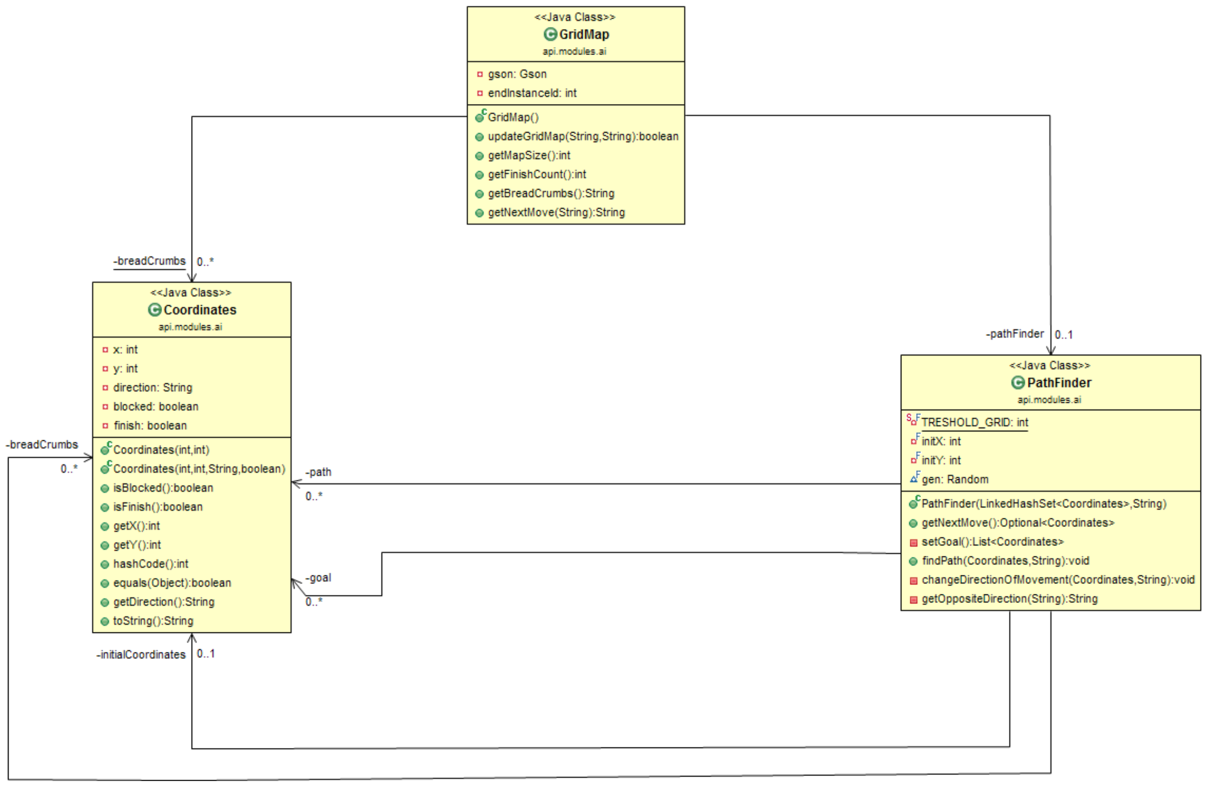


Figure A.5: ASTRA Modules - AI, path finder.

Figure A.5 shows the UML-Class diagram of ASTRA AI Modules. These modules are used for the very last prototype, where the position module is overridden and based on the gathered data the path to the end of maze is found.

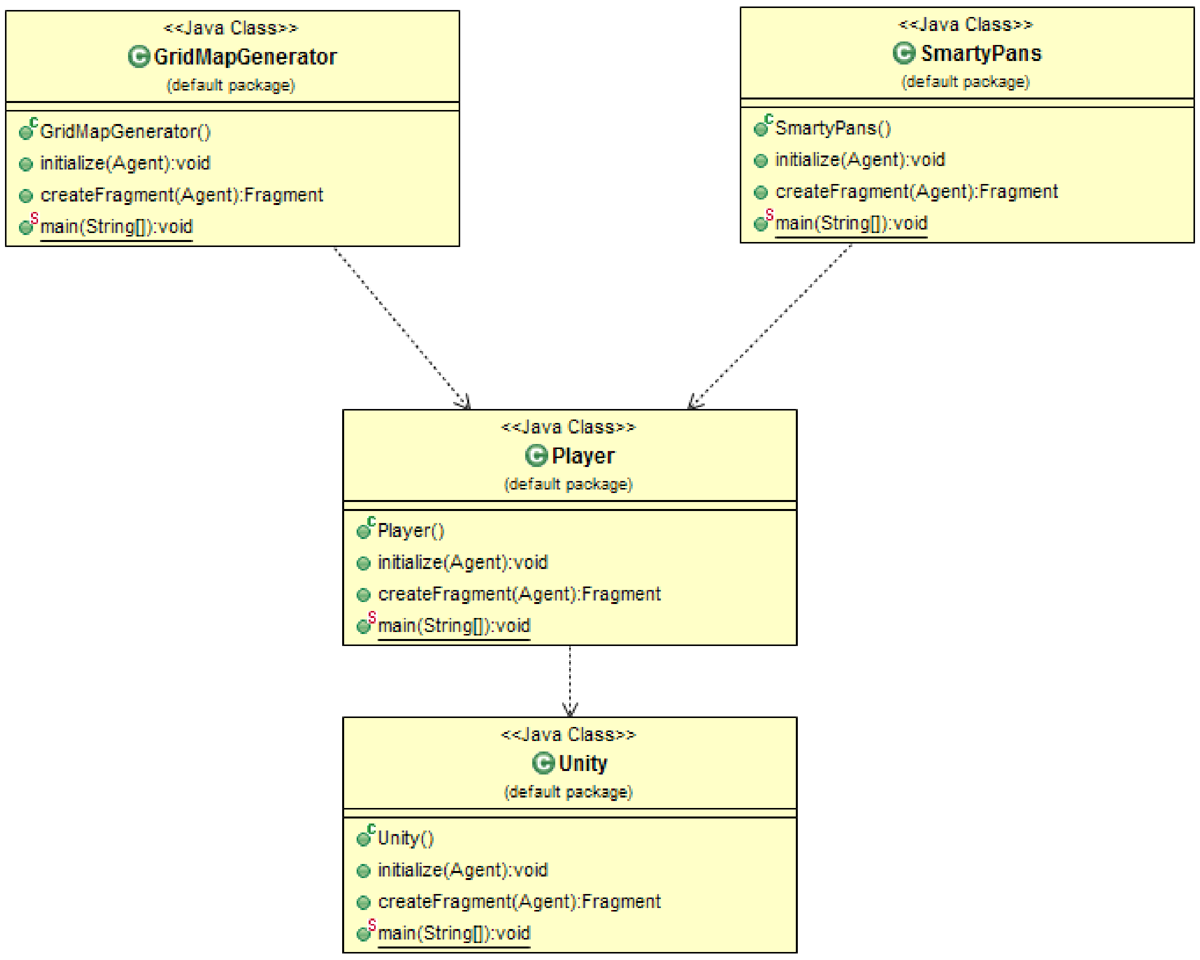
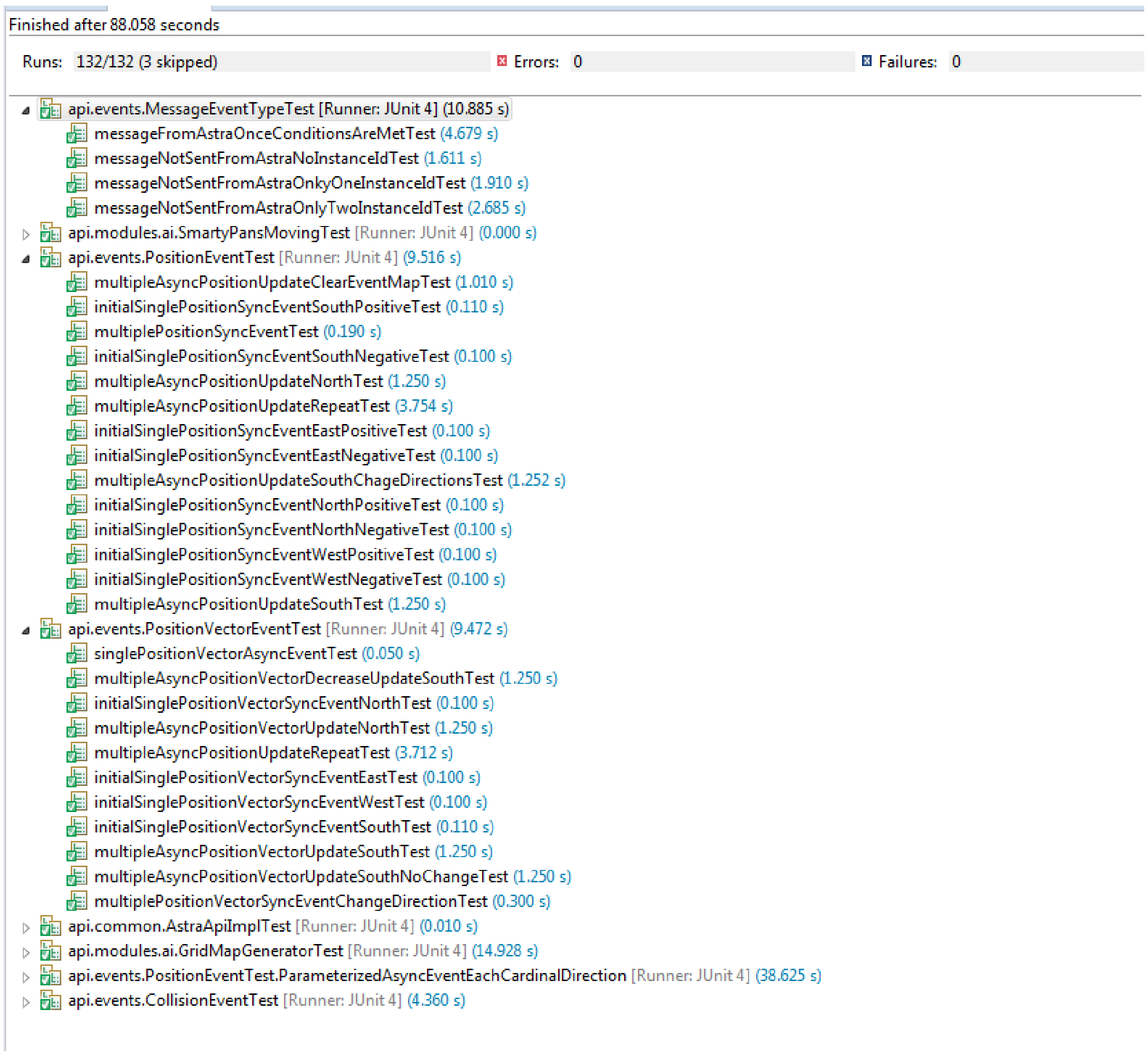


Figure A.6: ASTRA generated classes.

Figure A.6 shows the UML-Class diagram of ASTRA generated classes.

A.2 ASTRA API JUnit Tests



```
Finished after 88.058 seconds
Runs: 132/132 (3 skipped)          Errors: 0          Failures: 0

api.events.MessageEventTypeTest [Runner: JUnit 4] (10.885 s)
    messageFromAstraOnceConditionsAreMetTest (4.679 s)
    messageNotSentFromAstraNoInstanceIdTest (1.611 s)
    messageNotSentFromAstraOnkyOneInstanceIdTest (1.910 s)
    messageNotSentFromAstraOnlyTwoInstanceIdTest (2.685 s)
api.modules.ai.SmartyPansMovingTest [Runner: JUnit 4] (0.000 s)
api.events.PositionEventTest [Runner: JUnit 4] (9.516 s)
    multipleAsyncPositionUpdateClearEventMapTest (1.010 s)
    initialSinglePositionSyncEventSouthPositiveTest (0.110 s)
    multiplePositionSyncEventTest (0.190 s)
    initialSinglePositionSyncEventSouthNegativeTest (0.100 s)
    multipleAsyncPositionUpdateNorthTest (1.250 s)
    multipleAsyncPositionUpdateRepeatTest (3.754 s)
    initialSinglePositionSyncEventEastPositiveTest (0.100 s)
    initialSinglePositionSyncEventEastNegativeTest (0.100 s)
    multipleAsyncPositionUpdateSouthChageDirectionsTest (1.252 s)
    initialSinglePositionSyncEventNorthPositiveTest (0.100 s)
    initialSinglePositionSyncEventNorthNegativeTest (0.100 s)
    initialSinglePositionSyncEventWestPositiveTest (0.100 s)
    initialSinglePositionSyncEventWestNegativeTest (0.100 s)
    multipleAsyncPositionUpdateSouthTest (1.250 s)
api.events.PositionVectorEventTest [Runner: JUnit 4] (9.472 s)
    singlePositionVectorAsyncEventTest (0.050 s)
    multipleAsyncPositionVectorDecreaseUpdateSouthTest (1.250 s)
    initialSinglePositionVectorSyncEventNorthTest (0.100 s)
    multipleAsyncPositionVectorUpdateNorthTest (1.250 s)
    multipleAsyncPositionUpdateRepeatTest (3.712 s)
    initialSinglePositionVectorSyncEventEastTest (0.100 s)
    initialSinglePositionVectorSyncEventWestTest (0.100 s)
    initialSinglePositionVectorSyncEventSouthTest (0.110 s)
    multipleAsyncPositionVectorUpdateSouthTest (1.250 s)
    multipleAsyncPositionVectorUpdateSouthNoChangeTest (1.250 s)
    multiplePositionVectorSyncEventChangeDirectionTest (0.300 s)
api.common.AstraApiImplTest [Runner: JUnit 4] (0.010 s)
api.modules.ai.GridMapGeneratorTest [Runner: JUnit 4] (14.928 s)
api.events.PositionEventTest.ParameterizedAsyncEventEachCardinalDirection [Runner: JUnit 4] (38.625 s)
api.events.CollisionEventTest [Runner: JUnit 4] (4.360 s)
```

Figure A.7: JUnit Tests results.

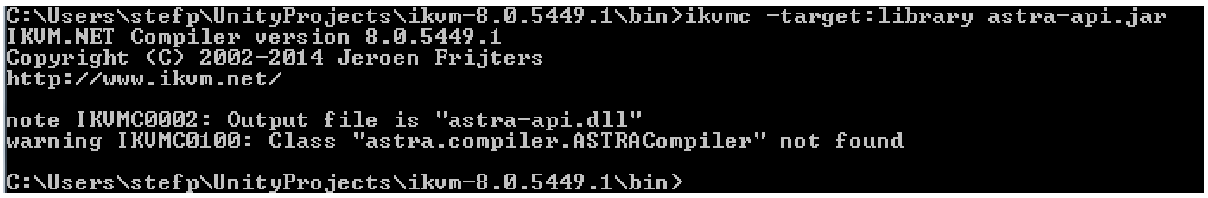
Figure A.7 shows the JUnit test results.

A.3 Build guide

A.3.1 IKVM

There are other options out there that supports the link between Java and C#. e.g. jni4net, MONO (which use IKVM under the hood). Our choice was to use IKVM.NET. For the purpose of building this API IKVM.NET version - 8.0.5449.1 was used and can be downloaded here or here.
Steps to convert jar to dll. We assumed that the jar is already built and will use the name 'astra-api.jar' for reference below.

- Place your jar under the <path to your ikvm version>\ikvm-8.0.5449.1\bin
- Open command line and navigate to the path from step 1
- Execute the command - ikvmc -target:library astra-api.jar
- You are done. Everything what you need now is under the bin folder, where you placed your jar file from step 1



```
C:\Users\stefp\UnityProjects\ikvm-8.0.5449.1\bin>ikvmc -target:library astra-api.jar
IKVM.NET Compiler version 8.0.5449.1
Copyright (C) 2002-2014 Jeroen Frijters
http://www.ikvm.net/

note IKVMC0002: Output file is "astra-api.dll"
warning IKVMC0100: Class "astra.compiler.ASTRACompiler" not found

C:\Users\stefp\UnityProjects\ikvm-8.0.5449.1\bin>
```

Figure A.8: IKVM - jar to dll

Figure A.8 shows the command line for converting jar files to dll assemblies.

A.3.2 Maven

When building a Mavens project, Maven will check pom.xml file, to identify which dependency to download. First, Maven will try to get the dependency from your local repository - .m2, if not found, it will try to get it from the default Maven central repository .

Note - For this project Java version - 1.8.0_141 was used. All ASTRA core language code was pre-compiled with this Java version before bundling it to a jar. The jar was named *astra-core*. It contains the main two components from ASTRA language - *astra.apis*, *astra.interpreter*. This jar is included in *astra-api.jar*, the one used in Unity.

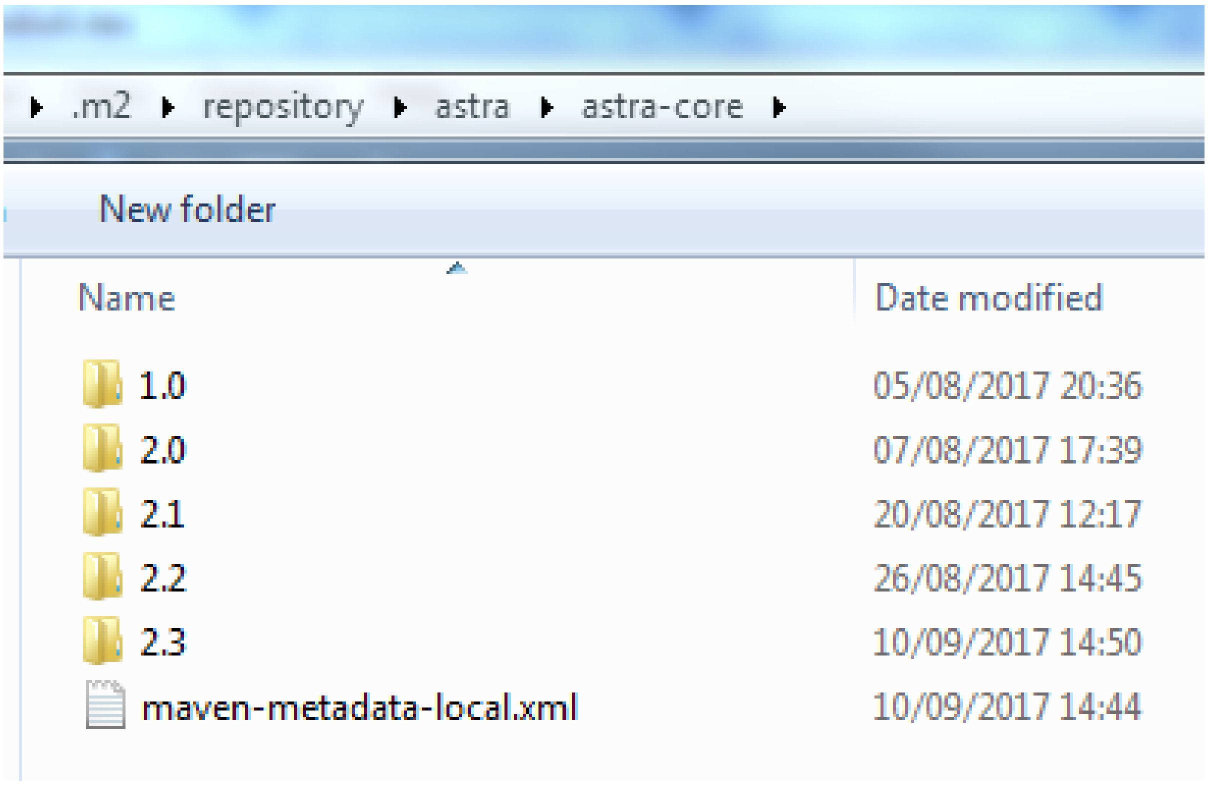
Steps for building with Maven:

- Compile *astra.apis*, *astra.interpreter* with current Java version and bundle it to a jar, using Eclpse IDE or command line. Called whatever you want, on bringing it to your local Maven repository an appropriate name will be defined - *astra-core* , with a version number appended at the end.
- Include the jar in your local repository by executing the following in command line. See Figure A.9 for the repository structure after execution of the command.

```
mvn install:install-file
    -Dfile=C:<path_to_your_jar>\astra-core-8.141_10-09.jar
    -DgroupId=astra -DartifactId=astra-core -Dversion=2.0 -Dpackaging=jar
```

- Include in Eclipse IDE the ASTRA API project as a Maven project.
- Write your extra Java Modules and ASTRA classes, if needed for extra functionality.
- Build the main jar used in Unity, by executing the following in command line or do the same in Eclipse. The name of the jar is defined in the pom.xml - *astra-api*

```
    mvn clean install
    ............. or build without JUnit tests...............
    mvn clean install -DskipTests
```



▸ .m2 ▸ repository ▸ astra ▸ astra-core ▸

New folder

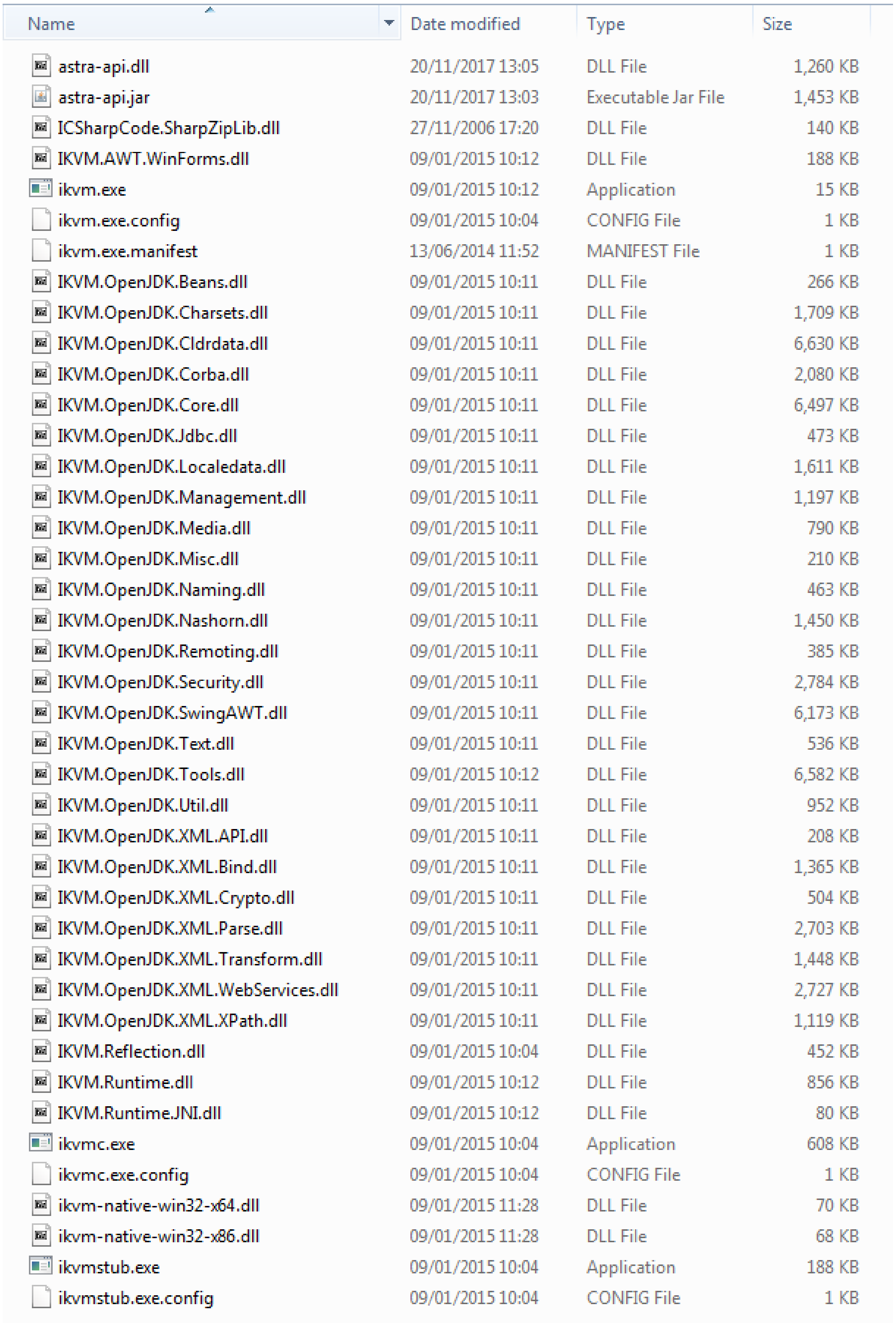
Name	Date modified
1.0	05/08/2017 20:36
2.0	07/08/2017 17:39
2.1	20/08/2017 12:17
2.2	26/08/2017 14:45
2.3	10/09/2017 14:50
maven-metadata-local.xml	10/09/2017 14:44

Figure A.9: Maven local repository

Figure A.9 shows the local Maven repository structure.

A.4 Deploy to Unity3D

Once all ASTRA source code is compiled, bundled as a jar and converted into a dll format, copy all the content from the bin folder and place it in Unity3D, under - your_project/Assets/Assets. This is how it looks under the bin folder of IKVM, the content that needs to be coppied:



Name	Date modified	Type	Size
astra-api.dll	20/11/2017 13:05	DLL File	1,260 KB
astra-api.jar	20/11/2017 13:03	Executable Jar File	1,453 KB
ICSharpCode.SharpZipLib.dll	27/11/2006 17:20	DLL File	140 KB
IKVM.AWT.WinForms.dll	09/01/2015 10:12	DLL File	188 KB
ikvm.exe	09/01/2015 10:12	Application	15 KB
ikvm.exe.config	09/01/2015 10:04	CONFIG File	1 KB
ikvm.exe.manifest	13/06/2014 11:52	MANIFEST File	1 KB
IKVM.OpenJDK.Beans.dll	09/01/2015 10:11	DLL File	266 KB
IKVM.OpenJDK.Charsets.dll	09/01/2015 10:11	DLL File	1,709 KB
IKVM.OpenJDK.Cldrdata.dll	09/01/2015 10:11	DLL File	6,630 KB
IKVM.OpenJDK.Corba.dll	09/01/2015 10:11	DLL File	2,080 KB
IKVM.OpenJDK.Core.dll	09/01/2015 10:11	DLL File	6,497 KB
IKVM.OpenJDK.Jdbc.dll	09/01/2015 10:11	DLL File	473 KB
IKVM.OpenJDK.Localedata.dll	09/01/2015 10:11	DLL File	1,611 KB
IKVM.OpenJDK.Management.dll	09/01/2015 10:11	DLL File	1,197 KB
IKVM.OpenJDK.Media.dll	09/01/2015 10:11	DLL File	790 KB
IKVM.OpenJDK.Misc.dll	09/01/2015 10:11	DLL File	210 KB
IKVM.OpenJDK.Naming.dll	09/01/2015 10:11	DLL File	463 KB
IKVM.OpenJDK.Nashorn.dll	09/01/2015 10:11	DLL File	1,450 KB
IKVM.OpenJDK.Remoting.dll	09/01/2015 10:11	DLL File	385 KB
IKVM.OpenJDK.Security.dll	09/01/2015 10:11	DLL File	2,784 KB
IKVM.OpenJDK.SwingAWT.dll	09/01/2015 10:11	DLL File	6,173 KB
IKVM.OpenJDK.Text.dll	09/01/2015 10:11	DLL File	536 KB
IKVM.OpenJDK.Tools.dll	09/01/2015 10:12	DLL File	6,582 KB
IKVM.OpenJDK.Util.dll	09/01/2015 10:11	DLL File	952 KB
IKVM.OpenJDK.XML.API.dll	09/01/2015 10:11	DLL File	208 KB
IKVM.OpenJDK.XML.Bind.dll	09/01/2015 10:11	DLL File	1,365 KB
IKVM.OpenJDK.XML.Crypto.dll	09/01/2015 10:11	DLL File	504 KB
IKVM.OpenJDK.XML.Parse.dll	09/01/2015 10:11	DLL File	2,703 KB
IKVM.OpenJDK.XML.Transform.dll	09/01/2015 10:11	DLL File	1,448 KB
IKVM.OpenJDK.XML.WebServices.dll	09/01/2015 10:11	DLL File	2,727 KB
IKVM.OpenJDK.XML.XPath.dll	09/01/2015 10:11	DLL File	1,119 KB
IKVM.Reflection.dll	09/01/2015 10:04	DLL File	452 KB
IKVM.Runtime.dll	09/01/2015 10:12	DLL File	856 KB
IKVM.Runtime.JNI.dll	09/01/2015 10:12	DLL File	80 KB
ikvmc.exe	09/01/2015 10:04	Application	608 KB
ikvmc.exe.config	09/01/2015 10:04	CONFIG File	1 KB
ikvm-native-win32-x64.dll	09/01/2015 11:28	DLL File	70 KB
ikvm-native-win32-x86.dll	09/01/2015 11:28	DLL File	68 KB
ikvmstub.exe	09/01/2015 10:04	Application	188 KB
ikvmstub.exe.config	09/01/2015 10:04	CONFIG File	1 KB

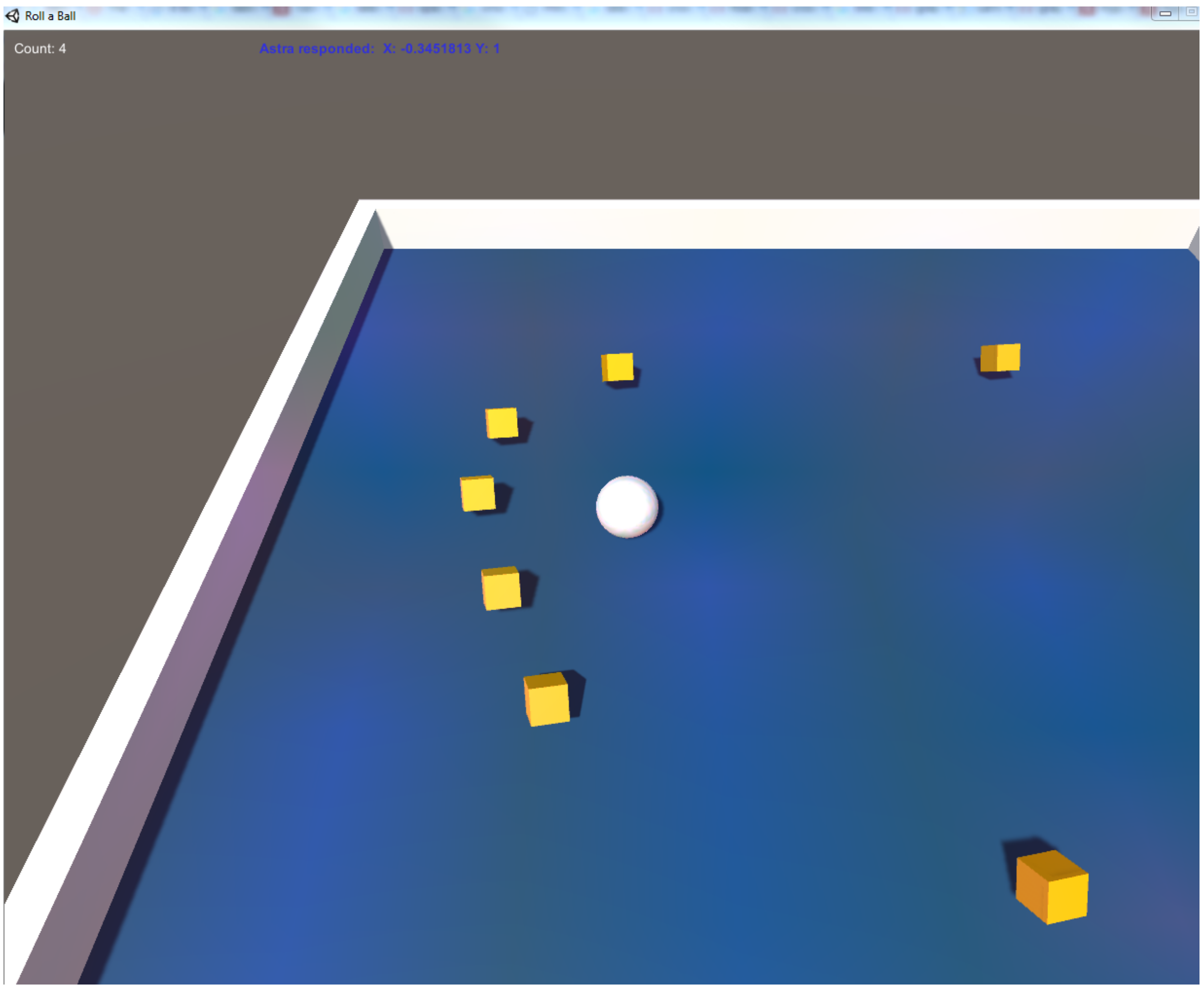
Figure A.10: IKVM - bin content after conversion.

Figure A.10 shows the the content of IKVM's bin folder after jar conversion to dll.

Appendix B

Unity3D prototypes implementation

B.1 Applications screen shots



Count: 4

Astra responded: X: -0.3451813 Y: 1

Figure B.1: Roll a ball prototype.

Figure B.1 shows the roll a ball prototype where we have an agent created and simple logic of rolling a ball and collection spinning cubes.

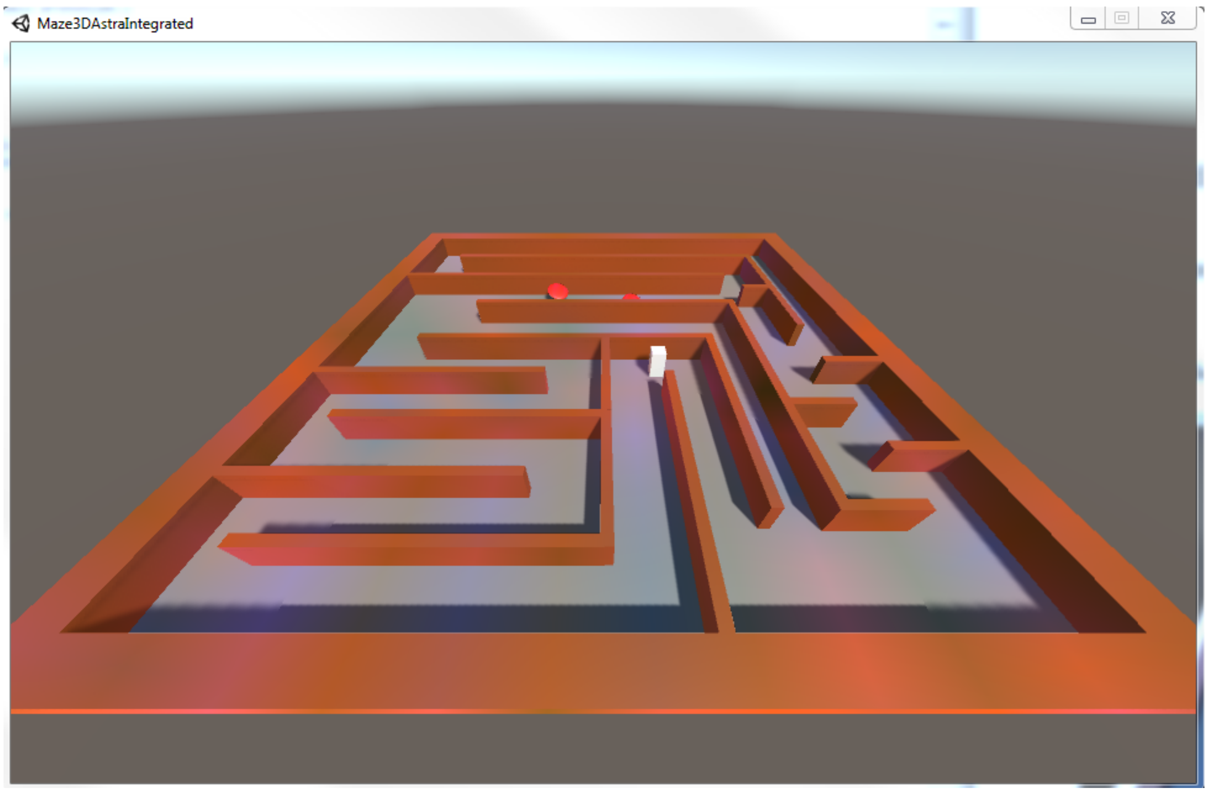


Figure B.2: Maze prototype with fully integrated ASTRA.

Figure B.2 shows the simple maze prototype with ASTRA fully integrated. On collisions random direction is chosen without repeating the same twice, such way the end of the maze is reached.

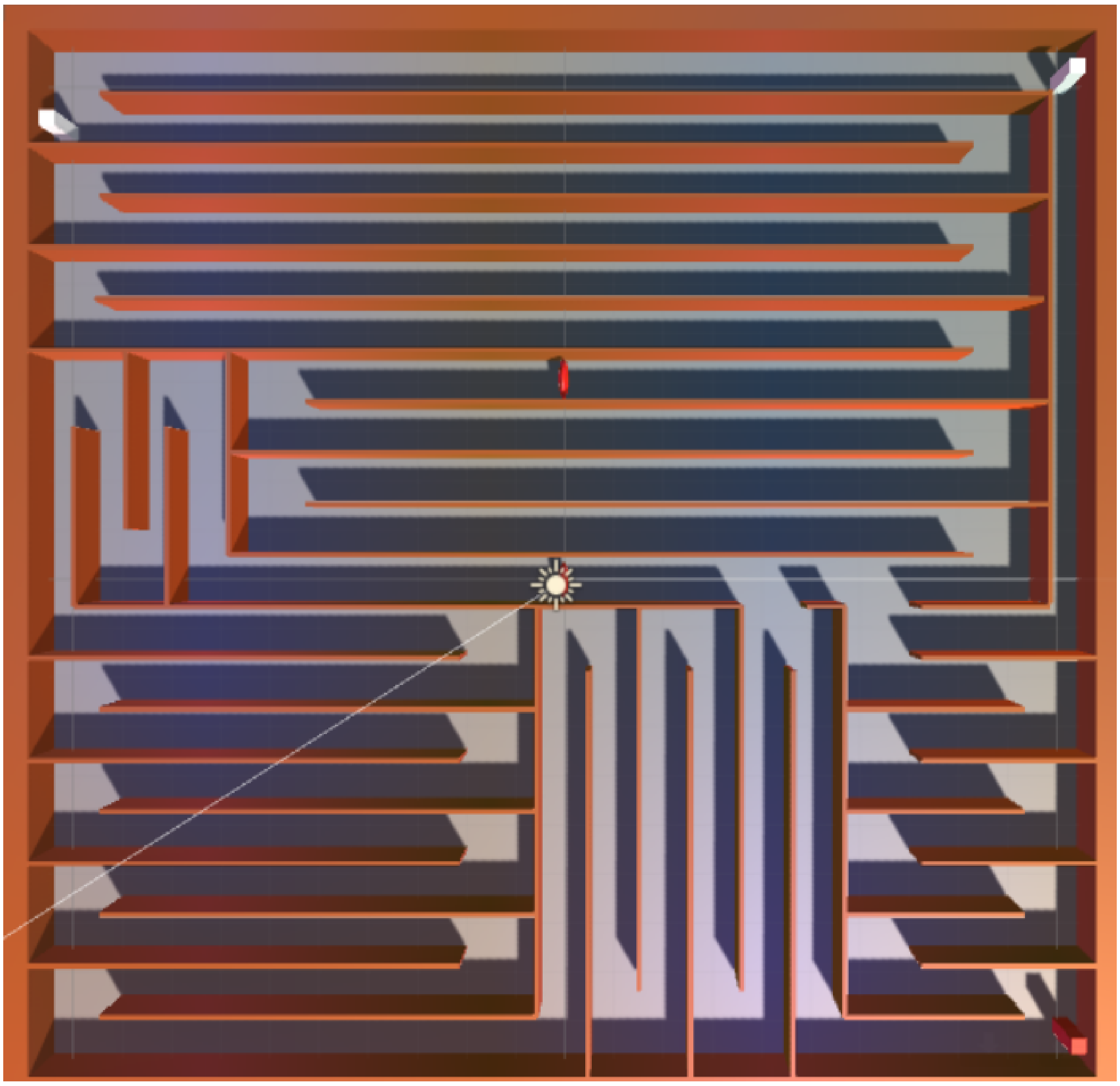


Figure B.3: Maze prototype with an intelligent agent.

Figure B.3 shows the maze prototype where we have two 'working' agents gathering data which is used by the 'smarty' agent to find the end of the maze.

Bibliography

[1] *ASTRA*, http://www.astralanguage.com/

[2] Collier, R., Russell, S., Lillis, D. *(2015) Exploring AOP from an OOP Perspective, in Proceedings of the 5th International Workshop on Programming based on Actors, Agents and Decentralized Control (held at SPLASH 2014), Pittsburgh, Pennsylvania, USA*

[3] Campbell,Abraham G.,& Stafford,John W.,& Thomas,Holz.,& OHare,G. M. P. *Why, When and How to use AuRAs (Augmented Reality Agents).*

[4] *Unity3D*, https://unity3d.com/

[5] *Unity VR*, https://unity3d.com/unity/features/multiplatform/vr-ar

[6] Scott W. Ambler *Introduction to Test Driven Development (TDD)*, http://agiledata.org/essays/tdd.html

[7] Jeroen Frijters *IKVM.NET*, https://www.ikvm.net/index.html

[8] Avik Sengupta, *An Introduction to IKVM* , http://www.onjava.com/pub/a/onjava/2004/08/18/ikvm.html

[9] Anand S. Rao. 1996. *AgentSpeak(L): BDI agents speak out in a logical computable language.* In Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away (MAAMAW '96), Walter Van de Velde and John W. Perram (Eds.). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 42-55.

[10] Koen V. Hindriks, Birna Van Riemsdijk, Tristan Behrens, Rien Korstanje, Nick

Kraayenbrink, Wouter Pasman, and Lennard De Rijk. *Unreal goal bots. In Agents for games and simulations II*, pages 1-18. Springer, 2011.

[11] Thalmann, Daniel and Hery, Christophe and Lippman, Seth and Ono, Hiromi and Regelous, Stephen and Sutton, Douglas, *Crowd and group animation*, SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes, 34, ACM, New York, NY

[12] Joshua Bloch, *How to Design a Good API and Why it Matters*, https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/32713.pdf

[13] Apache Maven, *Apache Maven Project*, https://maven.apache.org/

[14] JNBridge, LLC, *JNBridge*, https://jnbridge.com/about/us

[15] Steuer, Jonathan., *Defining Virtual Reality: Dimensions Determining Telepresence*, Department of Communication, Stanford University. 15 October 1993. , https://web.archive.org/web/20160524233446/http://ww.cybertherapy.info/pages/telepresence

[16] Luke Dormehl, *8 virtual reality milestones that took it from sci-fi to your living room* , https://www.digitaltrends.com/cool-tech/history-of-virtual-reality/

[17] Morton Heilig, *SENSORAMA SIMULATOR* , http://www.mortonheilig.com/SensoramaPatent.pdf

[18] Ivan Sutherland's 1963 Ph.D. Thesis from Massachusetts Institute of Technology republished in 2003 by University of Cambridge as Technical Report Number 574, *Sketchpad, A Man-Machine Graphical Communication System.*, http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf

[19] Lintern, Gavan (1980). *Transfer of landing skill after training with supplementary visual cues.*, Human Factors. 22: 8188

[20] Lubrecht, Anna., *Augmented Reality for Education*, The Digital Union, The Ohio State University 24 April 2012.

[21] Kaufmann, Hannes., *Collaborative Augmented Reality in Education*, Institute of Software Technology and Interactive Systems, Vienna University of Technology.

[22] *Augmented Reality Revolutionizing Medicine*, Health Tech Event.,

https://www.healthtechevent.com/technology/augmented-reality-revolutionizing-medicine-healthcare/

[23] *The Future of Virtual Reality*, https://www.fool.com/investing/2017/03/22/the-future-of-virtual-reality-5-things-to-know.aspx

[24] Michael Luck, Ruth Aylett, *Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments*, Centre for Virtual Environments, University of Salford, Department of Computer Science, University of Warwick, United Kingdom

[25] Yustyna Velykholova , *VR and AR in Insurtech: Applications, Use Cases, Prospects*, https://www.n-ix.com/vr-ar-insurtech-applications-use-cases-prospects/

[26] Yoav Shoham, *Agent-oriented programming*, Artifcial intelligence, vol. 60, no. 1, pp. 51-92, 1993.

[27] *Explained: How does VR actually work?*, https://www.wareable.com/vr/how-does-vr-work-explained

[28] *ARToolKit*, https://www.artoolkit.org/documentation/

[29] *VRML*, http://graphcomp.com/info/specs/sgi/vrml/spec/

[30] *Unity or Unreal Which Is the Best VR Gaming Platform?*, https://appreal-vr.com/blog/unity-or-unreal-best-vr-gaming-platforms/

[31] Jerad Bitner, *Tools for VR Developers*, https://www.lullabot.com/articles/11-tools-for-vr-developers

[32] *Unreal Engine 4 Documentation*, https://docs.unrealengine.com/latest/INT/

[33] *Blender*, https://www.blender.org/support/

[34] *SketchUp*, https://www.sketchup.com/

[35] Herbert A.Simon, *Artificial intelligence: an empirical science*, (1995) Artificial Intelligence, 77 (1) , pp. 95-127. Department of Psychology, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA

[36] S. G. Ma *The Research of Next-Gen Game Engine Virtual Reality in Actual Project*,

Applied Mechanics and Materials, Vol. 443, pp. 39-43, 2014

[37] *Getting Started With VR: The Best Software Tools Are Free*, https://makezine.com/2016/03/24/makers-introduction-vr-best-software-tools-free/

[38] *What is augmented reality (AR)?*, http://whatis.techtarget.com/definition/augmented-reality-AR [Accessed 29 Sep. 2017]

[39] Martn-Gutirrez, Jorge, Mora, Carlos Efrn, Aorbe-Daz, Beatriz, Gonzlez-Marrero, Antonio, *Virtual Technologies Trends in Education*, Eurasia Journal of Mathematics, Science and Technology Education, Vol. 13, pp. 469-486, 2017

[40] *Introducing ARKit*, https://developer.apple.com/arkit/

[41] *Expeditions*, https://edu.google.com/expeditions/ar/

[42] *Google VR*, https://vr.google.com/

[43] M. Wooldridge, N.R. Jennings, *Intelligent Agents: Theory and Practice*, Knowledge Engineering Review, vol. 10, no. 2, pp. 115-152, 1995.

[44] Jennings, N.R. (2001)., *An agent-based approach for building complex software systems.* Communications of the ACM, 44(4), 35-41.

[45] M. Wooldridge, *An Introduction to MultiAgent Systems*, John Wiley and sons, 2002

[46] A. S. Rao, M. P. Georgeff., *Modeling Rational Agents within a BDI-Architecture.*, In Proc of the 2nd International Conference on Principles of Knowledge Representation and Reasoning, pages 473484, 1991.

[47] Yoav Shoham, *Agent0: An agent-oriented programming language and its interpreter*, Journal of Object Oriented Programming (1991)

[48] R. Collier, G.M.P O'Hare, *Modeling and programming with commitment rules in agent factory.*

[49] Georgeff, M., Lansky, A., *Reactive reasoning and planning.*, In: Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87), pp. 198204, (1987)

[50] *PROFESSIONAL GRADE AR*,https://daqri.com/about/

[51] *How do we use code coverage for Unity?*, https://blogs.unity3d.com/2014/02/27/how-do-we-use-code-coverage-for-unity/

[52] *Managed Plugins*, https://docs.unity3d.com/560/Documentation/Manual/UsingDLL.html

[53] *Network Latency*, http://smutz.us/techtips/NetworkLatency.html

[54] *Photon Unity Networking*, https://doc-api.photonengine.com/en/pun/current/index.html

[55] Judith Bishop, R. Nigel Horspool, Basil Worrall, *Experience in integrating Java with C# and .NET*, http://webhome.cs.uvic.ca/ nigelh/Publications/ccpe03.pdf

[56] Rafael H. Bordini, Jomi F. Hbner, and Michael Wooldridge, *Programming multiagent systems in AgentSpeak using Jason.*, John Wiley & Sons, 2007.

[57] *JasonAPI*, http://jason.sourceforge.net/api/

[58] *JADE*, http://jade.tilab.com/

[59] Paul Milgram, Fumio Kishino, *A Taxonomy of Mixed Reality Visual Displays*, IEICE trans. inf. & Syst., vol. E77-D, NO. 12 December 1994

[60] T. Holz, A. G. Campbell, G. M. P. OHare, J. W. Stafford, A. Martin, M. Dragone., *MiRA Mixed Reality Agents.*, International Journal of Human-Computer Studies, 69:251268, April 2011.

[61] Billinghurst, M., Kato, H., Poupyrev, I., *The magicbook: a transitional ar interface.*, Computers & Graphics 25 (5), 2001., 745753.

[62] Wang, X., Jeong Kim, M., *Exploring presence and performance in mixed reality-based design space.*, In: Wang, Xiangyu; Schnabel, M. A. (Ed.), Mixed Reality In Architecture, Design, And Construction. Springer.

[63] OHare, G. M. P., Collier, R., Conlon, J., Abbas, S., August 1998., *Agent factory: An environment for constructing and visualising agent communities.*, In: Proceedings of the Ninth Irish Conference on Artificial Intelligence and Cognitive Science - AICS 98. Dublin, Ireland, pp. 249261.