Toward MAS in Game Engines: A Simple BDI Architecture in Unity3D

Nicola Poli, Mattia Cerbara

Alma Mater Studiorum – Università di Bologna
via Sacchi 3, 47023 Cesena, Italy
nicola.poli2@studio.unibo.it, mattia.cerbara@studio.unibo.it,

Table of Contents

List of Figures

1 Introduction

What game engines usually lack is the expressive power to define behaviours of agents using a more high-level approach like seen in Jason. What game programmers tend to do is create AI for the NPCs through FSMs and that could be a fine approach until a more complex behavior is desired.
Nowadays, the game production has become more and more accessible to everyone especially for amateurs without any deep knowledge of IT or even programming in general so it is quite difficult to find rigorous and formal models to solve AI problems and as for now, there is not a built-in functionality in game engines that can help programmers with these kind of situations.

With this project we will try to bring some of the ideas of the BDI architecture, into Unity3D creating agents that can act according to their desires, what they believe and what they can sense from the environment.
We will start analyzing some of the BDI architectures available, strip them down to the bone and, from there, create a simplified version that can be used in a Unity3D project.

Since game engines like Unity3D usually work on loop, the computation on each frame must be optimized so that we can have a smooth simulation, for this reason it is also necessary to take into account the complexity of a BDI agent and how fast it can complete its phases (sense, plan, act) or find some different approach that can permit the agent to do its reasoning asynchronously.

2 Related Work

Disclaimer: the majority of the related work we have analyzed is part of the Autonomous Systems course of the University of Bologna [1].

The concept of Multi-Agent System evaluated and applied in Game Engines was discussed in our previous work [2], where it was analyzed how much the gap between these two worlds could be tackled and solved by using approaches like a proper coordination model (like LINDA) and the exploiting of what reminds the typical abstraction of a MAS within the Game Engines (in this case Unity3D), that are agents, society and environment.
It was shown how Unity3D already provides some useful concepts and abstractions which brought to a first implementation of a tuple space (with LINDA primitives) and its application to the typical case study of the Dining Philosophers, as a demonstration of the possibility to exploit the Game Engines framework and functionalities (like rendering, physics, ...) to create agents living and interacting with each other and the world (that is what happens in a MAS).

An important logic-based agent programming language is AgentSpeak, based on the BDI architecture and become relevant in the agent systems literature: it was an abstract agent programming language aimed to help the understanding of the BDI architecture as well as its description and programming and it is inspired by Procedural Reasoning System (PRS) and extended in order to become a practical agent programming language (nowadays it is also one of the most popular agent-oriented languages because of the development of the Jason platform); although their concepts and ideas were indeed fundamental for our work, there is not an implementation of such technologies in C# so not directly exploitable in Unity3D.

A wide used approach to realize complex AI in Unity3D is RAIN [3]. It provides a form of perception that can permit an agent to be aware of its surroundings, take decision based on the information acquired and on a behavioural tree that defines its responses to environmental changes. It is still very FSM-oriented but it's the closest thing you can found to a BDI architecture.

3 Requirements

What we want to realize is a basic version of a BDI architecture in Unity3D that follows all of the fundamental aspects characterizing a classic BDI agent: an agent should be able to sense environmental changes, to update its beliefs based on the information acquired and the interaction with other agents, to choose a behaviour (a plan) after a reasoning about its desires and its beliefs and finally to actuate the actions that correspond to the plan chosen.

4 A Brief Study of Autonomy

4.1 Autonomy and Intelligent Agents

The concept of autonomy is pervasive and it flows from different contexts (such as philosophy, military, social sciences, PL, AI and software agents) with many different notions.
In our case, we talk about autonomy in intelligent agents, a computational system capable of observing some environment, perceiving information and making decisions, with an eye to the software agent as the place of computational autonomy (that is encapsulation of control, no interfaces, no external control, capable of exchanging information between them within a MAS), so the agent is able to act independently, in according to the features of the agent behaviour like situatedness, reactivity, proactivity, social ability that are already known to be part of the autonomy itself.
Many ideas to how could be implemented an agent: agents with internal states, agents referring to human attitudes as intentional notions living in an intentional system, that is reducing the behaviour of the agent as moved by some kind of folk psychology (in the same way in which human behaviour could be explained, with believing, wanting, and so on) so agents which behaviour could be predicted in terms of beliefs, desires and rational acumen.

4.2 Intentions and Practical Reasoning

With the growth of complexity the intentional system's abstraction could be exploited in order to describe in an easy way the behaviour of complex systems (and agents), so it was tackled it describing and predicting the agent behaviour with intentional and internal states (like the mental attitudes abstraction).
In fact, modelling the agents in this way allow the developer to depict in an easily way the agent complex behavior in something understandable and explainable; moreover the mimic of cognitive (human) mental states permit an internal representation in terms of epistemic states (agent's knowledge, beliefs) motivational states (agent's objectives, goals, desires) and processes of selecting the action to be executed (Practical Reasoning, with deliberation and means-end reasoning consisting in decide what the agent is going to achieve and how to do it).
The outcome of the deliberation phase of Practical Reasoning, the process to figure out what to do in order to achieve what I belief, are the intentions, so the work-flow of actions the agent is intended to adopt in order to fulfill their desires.

4.3 BDI Agents

Among the implementations of Practical Reasoning Agents, one of the most popular and successful framework for agent technology is defined by Rao and Georgeff where the concepts of beliefs, desires and intentions are the core ones

and agents within this framework are typically referred as BDI agents.
The BDI framework is composed of 3 main concepts:

- **beliefs**, representing the main knowledge of the agent about the world and itself (percepting the current state of the environment and evaluating internal states and information from other agents) and they are expected to change in the future as well as the world changes
- **desires**, motivational state of the agent, the state of the world and objectives the agent would like to accomplish
- **intentions**, deliberative state of the agent, representing what the agent has chosen to do, implemented in terms of plans and post-conditions: they are emerged properties actualized by choosing a plan for achieving a given goal

Other components included in the abstract model are:

- **plans** (plan library), a sequence of actions executable by the agent, so its procedural knowledge about what to do in order to achieve something and they are generated at design time by the programmer and selected by the agent at runtime

- event queue, with events representing triggers for reactive activity by the agent, so they could be either external (i.e. coming from the environment or other agents) or internal (i.e. from reflexive actions), they are supposed to be atomic and they even may update beliefs, trigger plans or modify goals.

The reasoning procedures of deliberation of the agent can be decomposed in two phases, during which the agent generates a set of possible plans to be executed in according to its beliefs and intentions and commits the intention to achieve one of them:

- **options generation**, it understands what are the available alternatives to be executed

- **deliberation**, it chooses an option (which are intentions, a plan) to apply from the available ones and filter the unadoptable ones

The agent control loop proposed by Rao and Georgeff is the following:

Algorithm 1 BDI agent control loop

```
initialize-state();
while true do
   options := option-generator(event-queue);
   selected-options := deliberate(options);
   update-intentions(selected-options);
   execute();
   get-new-external-events();
   drop-successful-attitudes();
   drop-impossible-attitudes();
end while
```

The following figure represents the basic architecture of a BDI agent:

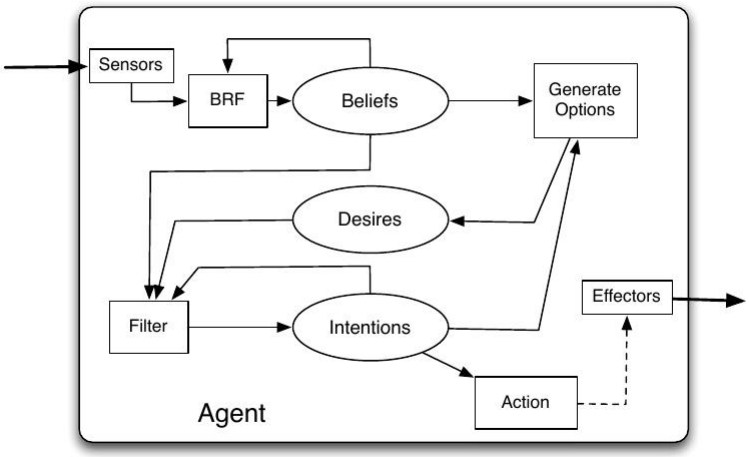

Fig. 1: BDI basic architecture

5 Project

This section will present the project's architecture and some diagrams that explain how the BDI model has been integrated and implemented in Unity3D. The project will follow the representation of the BDI architecture depicted in Fig. 1 as an initial step: then it will be analyzed how these concepts could be mapped to Unity3D abstractions, finally in the next section we will present a possible implementation of a BDI architecture.

All the source code of this project will be available in the github page: https://github.com/conner985/Unity-BDI

5.1 Domain Model

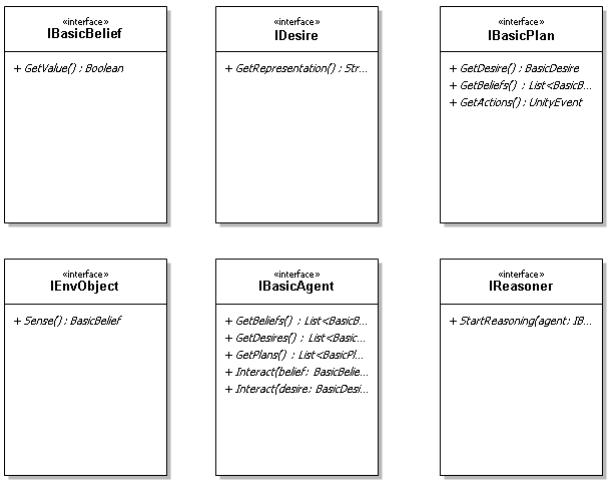

Fig. 2: Domain Model

- `IBasicAgent`: the BDI agent abstraction, composed of IBasicBeliefs, IDesires and IBasicPlans, from now on it will be simply called "Agent";
- `IBasicBelief`: it represents a generic belief as a boolean object;
- `IDesire`: it represents a desire within the agent, it will have a representation that is basically the name of a belief the agent wants to be true;
- `IBasicPlan`: it contains a set of actions the agent is able to perform, an IDesire and a list of IBasicBelifs that are needed in order to execute those actions. It is the equivalent of the Intention;

– `IEnvObject`: it is the representation of a “sensable” object inside the scene, an agent can sense them and retrieve the belief that they contain;

– `IReasoner`: it reasons on the plans of the agent and will return the first one that is executable based on a complete match between the beliefs and the desire of that plan and those of the agent.

These concepts are the representation of the components depicted in Fig. 1 and they are the basic of the BDI architecture: the IBasicAgent is the entity called ”Agent“ in the scene, able to interact with the environment and other agents, but with the internal logic of the BDI model (with IBasicBelief, IBasicDesire and IBasicPlan) and with an IReasoner, which is responsible of the deliberation phase during the Update() cycle (that is, when the Update() function of the Unity3D script is called every frame, the reasoner decides which plan to execute in according to beliefs and desires); all details are described in the next subsection, while in the section Case Study we will present an application in Unity3D of this architecture.

5.2 Architecture

We adapted the high-level BDI Agent Architecture depicted in the Fig. 1 to be more suitable for the integration with the game engine, moreover we have simplified some of the concepts that are present in that architecture in order to realize in the fastest way possible a working prototype that can be easily extended in the future.

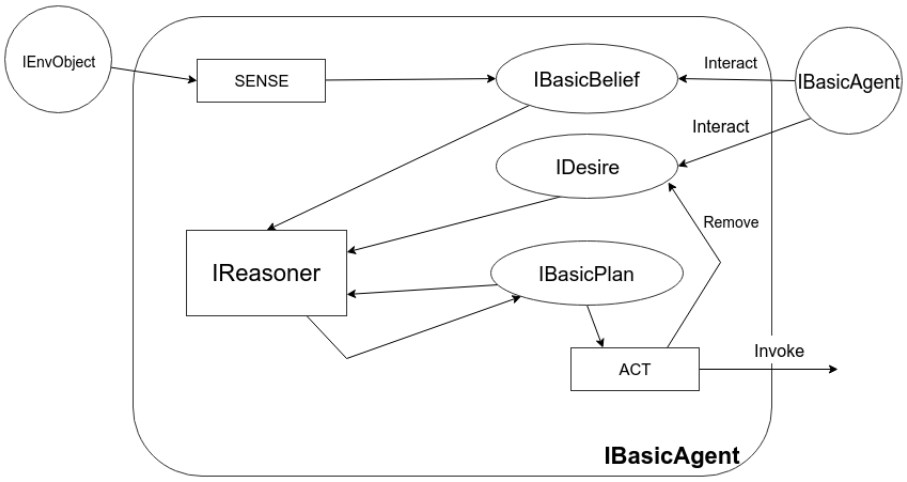

Fig. 3: BDI Agent Architecture - High Level

After the high-level architecture has been decided, we have further deepened the concepts emerged, in a low-level architecture as follows:

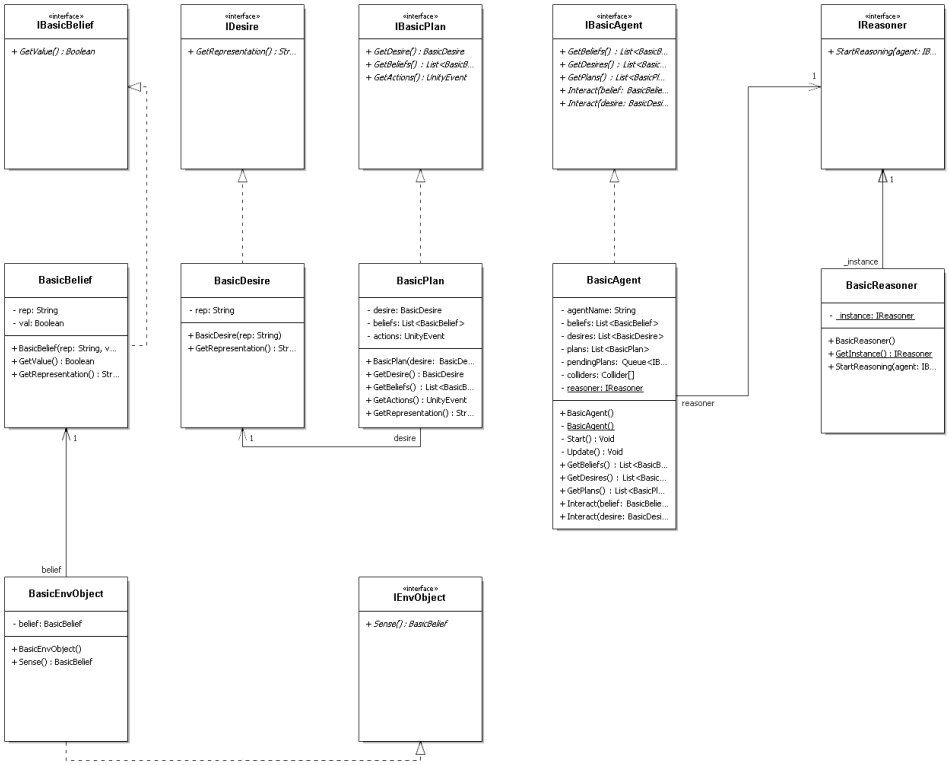

Fig. 4: BDI Agent Architecture - Low Level

We now analyze some of the choices made in this architecture:

- `BasicReasoner`: since we have seen the reasoner as a shared entity, every agent need one and it basically exhibits the same functionalities to each one of them, we designed it using the Singleton pattern and every agent has an instance of it;

- `BasicAgent`: we designed a fully inspectionable agent, it is possible to retrieve its beliefs, desires and plans. This choice has been made mainly to simplify the communication with the reasoner. Every agent can interact with the environment through our EnvObjects and with other agents, for the latter we decided to add a couple of methods that, when called, will add beliefs or desires to the agent. The interaction is always procedure call, there is no other way to enable agents communication, not with the default APIs.

– `BasicPlan`: since for us was important to have the possibility to define an agent (i.e. its beliefs, desires and intentions) without altering the code, it was necessary to define the actions of a plan as a UnityEvent (UnityEvents are a way of allowing user driven callback to be persisted from edit time to run time without the need for additional programming and script configuration, so they are used to seamlessly call functions during the edit phase without coding), in this way it was possible to choose actions from a separate script at runtime. The actions are the only thing that must be defined in a separate script in order to associate them to a plan (MasterActions and BotActions are the classes representing the plan library for each agent). Beliefs and Desires are definable from the Inspector, as shown in the Fig. 5;

– `BasicEnvObject`: we represent every sensable object in the environment with this class, so an agent is able to interact with all IEnvObject within its range and retrieve the belief associated to them (calling the Sense() function of each BasicEnvObject).

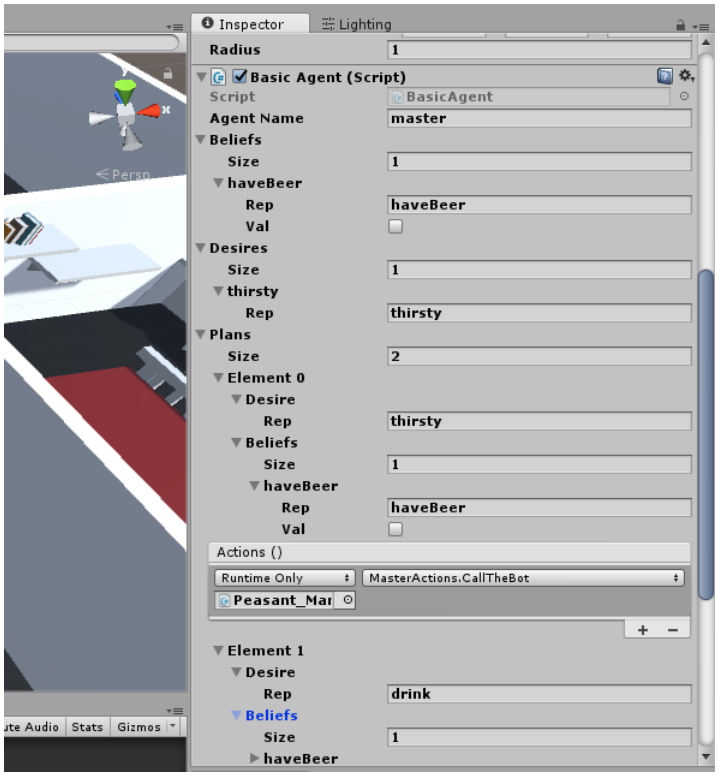

Fig. 5: A BasicAgent in the Unity Inspector, usually found in the right side of the Unity window

The Inspector it's used to explore and modify the public fields of a script attached to a GameObject that extends the MonoBehaviour class. It is also possible to explore the private fields but it is necessary to tag them as `[SerializeField]`. As seen in the Fig. 5, our agent is fully definable directly from the Inspector: Beliefs, Desires and Plans are lists so altering the Size parameter can let you add or delete them.
The actions, as explained before, are a little different, it was necessary to attach another script (that also extends MonoBehaviour) to the agent that contains a series of functions, in order to make them selectable from the Inspector.

6 The Domestic Bot: A Case Study

In this section it will be described an application of the BDI architecture in Unity3D with a Domestic Robot example.
The scene is composed of 2 agents acting inside a home-like environment: one agent is the master, he is tired and he needs a drink, so he calls the second agent (a domestic robot, which has the goal to serve the master) asking it to bring him a beer: the Domestic Robot goes to the fridge, takes a beer and finally brings it to the master.
Every time the master is thirsty and doesn't have a beer, he will call the bot (he knows its reference) adding a new beliefs to its knowledge (through Interact(belief)), this belief will activate a plan to make the bot walk toward the fridge to grab a beer, it will know when it have reached the fridge because it is a BasicEnvObject containing a belief that it can sense in order to stop and activate the animation to open the fridge. After the animation is complete, it will walk toward the master that also act as a BasicEnvObject to let the bot sense him, stop and activate the animation to handle the beer (the bot will use Interact(belief) on the master to add a belief and let him know that now he has a beer). Now the master can finally enjoy his beer.

In order to decide which plan is the chosen one to be executed, each agent will start the reasoner part: each BasicAgent has a sense-plan-act within its Update function, so every frame it will perform the following actions:

- **SENSE**: the agent casts a sphere and retrieve every BasicEnvObject within it, then it will call the Sense() function provided by every BasicEnvObject in order to update its beliefs with the sensed ones

- **PLAN**: the agent will ask the reasoner to start its computational phase (passing a reference to itself to the reasoner - StartReasoning(agent)) in order to select the right plan to be executed, so the reasoner will check if there is a desire in the agent that match the one in some agent's plan, then the chosen plan is analyzed in order to check if its beliefs are satisfied in the agent, finally if even this phase is completed, the chosen plan is returned to the agent, ready to be executed and it will be added to the internal queue (FIFO) containing the pending plans to run.

- **ACT**: the agent will now check its internal queue where pending plans are waiting, select the first one and execute its action, it will also delete the desire associated with the plan chosen from its internal desires because it has now been fulfilled.

Fig. 6: Part 1: the bot goes to take a beer for the master

Fig. 7: Part 2: the bot gives the beer to the master

7 Results and future work

With the architecture we designed it is really easy to design an agent without adding a single line of code, except for the actions, but it has some limitations. It is very basic, it is an initial deal with the problem of an integration of a BDI model in Unity3D and, as the name suggests, it can only handle boolean beliefs and desires as simple strings, basically a desire is just the name of a belief that the agent wants to be true.
The reasoner is not optimized, at every frame it will check every single plan of the agent, trying to find one that is executable and the reasoning is very object oriented, while a more logic-oriented one could be much more efficient.
EnvObjects can contain only a single belief and this could be too much restricted for a more general use.

The goal of this project wasn't to implement a fully general BDI architecture with 1OL reasoning but just to give a starting point to reach that and to show that can be a valid alternative to Finite State Machines that are largely abused in game programming.

References

1. A. Omicini and S. Mariani. Autonomous systems (2016-2017). http://apice.unibo.it/xwiki/bin/view/Courses/Sa1617, 2016.
2. N. Poli and M. Cerbara. Game engines and multi-agent systems: a marriage? http://apice.unibo.it/xwiki/bin/view/Courses/Sd1516Projects-GamingMasPoliCerbara, 2016.
3. RivalTheory. Rain. http://legacy.rivaltheory.com/rain/, 2015.