**Appendix**

# Developing digital environments in Unity

Every application build with Unity is made of *Scenes, GameObject’s, components* and *scripts*. An application can contain an arbitrary number of scenes, and the application shown in figure 6 contains one, namely *SensorEnvironment-4.2.* Every scene contains GameObject’s, in which components and scripts are attached, to sustain any form of behaviour imaginable. Components enables the use of all the built-in functionalities in the Unity Engine, and scripts provides the researcher with the option to take full control.

Developing an environment, formally known as a *Scene* in Unity, to facilitate the possibilities within the ML-agents toolkit requires some standard objects; an actual *environment* to explore, an *academy*, an *agent* and a *target*. Figure 6 is an example of how everything could be organised within a scene.

*Figure 6 – A scene in Unity containing the components of the ML-Agents toolkit*A screen shot of a computer

Description automatically generated  
*The scene contains all available elements within the environment.*

One thing to note from figure 6 is that all necessary elements are contained in a *prefab* named *Area\_EnvX*[[1]](#footnote-1). Prefabs are pre-defined GameObject’s, and the use of prefabs are a neat way of altering similar objects simultaneously. For the environment in question, containing all necessary elements in a prefab, is a way to utilise a parallelised set-up, allowing for faster training.

The following describes key components of the entire environment, divided into the four categories made up by the basic objects required by the ML-Agents toolkit. All components left out of the following description can be found in the appendix, to ensure reproducibility.

## Environment

The walkable area is labelled *ground* in figure 6, and it serves two important purposes. Firstly, it defines the extend of the area through its scale, seen in figure 7. The extend of the area is used to ensure that random placing, through scripting, of objects happens within the bounds of the traceable area.   
Secondly, it serves as a container for the objects belonging to this training area[[2]](#footnote-2). Initialising new GameObject’s as children of another GameObject is a way to ensure interaction with intended GameObject’s. It allows the researcher to write generic scripts and not instances specific scripts, which are in general good practice, and especially desirable when working with parallelised set-ups.

## Walls & Obstacles

The walls as well as obstacles are, as GameObject’s, identical to the *ground* GameObject, with a different tag along with the obviously different size and position. The walls and obstacles have different materials, to indicate that they represent something different.

*Figure 7 – The ground object  
A screenshot of a cell phone

Description automatically generated*

##### Pedestrians

The pedestrian GameObject is a prefab, which is attached to the academy from where it is initialised. For further description, see section 5.4.1.2.

##### Crowded areas

The crowded area is a prefab and is initialised from the academy object.

#### Academy

Up until now, none of the outlined parts of the environment been specific to the ML-Agents toolkit. The academy is one of three cornerstones of the toolkit, and it serves to bridge actions and observations to the TensorFlow-based models in Python. The academy is a Python API, and bridging is done through the *brain*.  
The academy has only a single component attached, seen from figure 11.

***Brain***

There are two types of brains, learning brains and player trains.   
Learning brains learns the policy based on the net implemented in TensorFlow. See figure 10 for the configuration of a learning brain.   
The player brains allow the researcher to test before invoking the learning brain, by giving the researcher the option to control the agent with keys on the keypad.

*Figure 10 – A learning brain*  
A screenshot of a cell phone

Description automatically generated

A brain can take *vector observations* as well as *visual observations* as input, and it outputs an action vector. The size of both the observation vectors and the action vector is specified by the researcher, and it is problem specific.  
The learning brain used this case takes in vector observations, as visual observations requires more computational power than available for this paper. The size of the vector is determined by the observations collected, with an example using sensor information provided in section 2.3.1.3.

Stacking observation vectors equips the agent with short-term memory, which can be beneficial when dealing with dynamic obstacles.

*Figure 11 – The academy*  
A screenshot of a cell phone

Description automatically generated

It is possible to add more than one brain to a single agent; however, it is not possible to train multiple brains on a single agent – yet. Training multiple brains on an agent would present some interesting possibilities, discussed at the end of this paper.

***Varying complexity***

An important purpose of the academy is to initialise and alter the complexity of environment. The academy script, except for some helper functions, contains two methods; InitialiseAcademy() and AcademyReset().  
AcademyReset(), for this paper, is used with CL-based training, and the purpose is to update the environment with increased complexity at specific times.   
One could alternatively change location of certain objects from the academy, and thereby specify a maximum number of steps or call *Done* to reset. However, changing location of certain objects, in this study, is done through the agent script.

***Reset parameters***

The reset parameters are the variables that enables changing the environment on reset, and they are input variables to the agent script as well, used when changing locations.   
The values of the reset parameters are being set by the *curriculum[[3]](#footnote-3)*.

For a description of the configuration of the academy, see section 5.4.2.

#### Agent

The agent object is by far the most complex, in terms of the number of components and methods contained in the attached script. The content of the agent object is seen figure 12.

The agent GameObject contains a ray perception component along with two custom scripts, other than the familiar components as of the mesh, the rigidbody and the collider components.

Other than the mesh components as well as the rigidbody and collider component, which are previously described, contains the agent a ray perception component along with two custom scripts; one to draw trails and one to hold the necessary methods needed for any agent, to leverage the ML-Agents toolkit.

***Ray perception***

The ray perception component enables the agent to cast rays in specified length and direction and are here used to collect observations about the state of the environment. The rays resemble with LIDAR sensors commonly used for robots (Georgiev and Allen, 2004; Kümmerle et al., 2013; Starship, 2019; FedEx, 2019).

***Agent Script***

Any ML agent needs an agent script, to hold the agent-specific methods, just as the academy needed an academy script. An agent script contains some default variables and options, listed above the grey line in figure 12.

*Figure 12 – The content of the agent object  
A screenshot of a cell phone

Description automatically generated*

***Brain***

An agent needs a brain to control the movement of the agent, and it is the same brain as specified in the academy. Certain methods control the movement, more specifically; CollectObservations, AgentAction and MoveAgent.

CollectObservations serves to provide the agent with a vector of *observations*, sensor information in this case,and is only needed when the brain uses vector observations*[[4]](#footnote-4)*.   
The agent has 180 degrees sensor vision, in steps of 10 degrees, spanning in front with a length of 50 meters.   
The agent is provided with five tags to recognise, and a ray is casted for each of the tags to recognise, for each of the degrees specified. Furthermore, it keeps track of distance to the objects, and if an object has been missed.   
The observation vector has the dimension , which for the specific case here means that the observation vector is .

Describe partial/full observability.

The agent chooses an action, based on the observations about the current state of the environment. The action/-’s is chosen by the brain, and facilitated to the agent through the AgentAction method, in which the action signal is translated to actual movement via the MoveAgent method.  
The action of the agent is degrees to turn, as the agent is moved forward with a constant speed. The speed of the agent is subject to change, via the public ***speed*** variable.   
The speed of the agent is, by default yet subject for change, set equal to 1 m/s, which is roughly equal to a speed of 3.5 km/t – the average chilled walking speed.

The direction of the movement could, as well, be an action for the agent to learn, subject of investigation in section 3.3.1.

***Max step***

As with the academy, the option to specify a maximum number of steps is present. The agent will be reset when the number of steps surpasses the specified number, which is useful to break unfavourable movement patterns, as is shown later.

***Reset on Done***

Another way to reset the agent is to call *Done*at some point, which usually is after colliding with an object in the environment.

Both *max step* and *reset on Done* is used in this paper. Collisions with a wall, a static obstacle, a pedestrian or the target results in Done being called, to reset the agent, and start a new episode.

On reset, not only the position of the agent resets, but also the position of the target and the crowded areas. Re-positioning the crowded areas causes re-drawing the densities.   
For this reason, the option to specify levels and number of possible **densities** is available under the agent script.

***Decisions***

Decisions can be done either at a specific interval or on demand, and this paper here uses decision at a specific interval. The decision interval (DI) should be chosen with the complexity of the environment and the speed at which the agent moves in mind.

As with many of the other parameters of the environment, it is of interest to choose the level at a level which generalises. The default level of DI is chosen to be every fifth step but can be subject to change. It seems natural that greater complexity/speed should benefit from lower DI.

1. X referrers to the current version of the implementation. [↑](#footnote-ref-1)
2. Made up of the *Area* prefab. [↑](#footnote-ref-2)
3. See section 2.4. [↑](#footnote-ref-3)
4. Visual observations are provided directly to agent. [↑](#footnote-ref-4)