AI4U Connect Godot Edition

Developer Guide - Alpha Version (November 25, 2022)

Gilzamir Gomes (Programmer/Analyst)

Eduardo Nogueira (Game Designer)

Summary

Introduction

AI4U Godot Edition (AI4UGE) enables transparent python connection with Godot for the creation of interesting artificial intelligence experiments. With this, you can use the potential of machine learning frameworks that provide a Python interface in creating non-player characters (NPCs) or *neural network-based* virtual characters models. In addition, you can connect any AI4UGod-based Python script to control game items.

AI4UGE has been refined for the development of NPCs development experiments using artificial intelligence. To this, it provides a playful and transparent way of modeling NPCs through an abstraction of intelligent agents. You build NPC by adding actuators and sensors, as if you were building a robot, however, with a virtual body. This provides us, in addition to a scope of experimentation, a powerful educational appeal.

Therefore, the main features of AI4UGE are:

- development of game controls based on the artificial intelligence agents paradigm;

- *multi-engine*, supports games modeled *on Unity* or *Godot*, without the need to reschedule Python scripts; and
- Environment modeling based on the *Gym* framework.

Environmental Modeling

The modeling of the environment, despite the visual appeal, should emphasize the physical aspects of the environment, such as obstacles, floors, constructions, passages and collisions. The AI4UGE has collision sensors with the standard rigid body shapes available on *Godot*. Therefore, this modeling is very dependent on the game engine you use.

*Godot*Figure 1, you can visualize a very simple scene, but with the necessary elements in an environment: two objects (a capsule with arrow representing the agent body and a cube that represents an item that the agent has to capture), a camera, and a floor (a simple flat terrain). What you can't see is the physics engine that adds gravity to the environment, fixes the floor like a floor preventing objects from falling forever and allows objects to collide with each other. All these elements are present in the AI4UGTesting project (available *in examples/serverside/Godot/AI4UGTesting*). We will build on this project to explain the different components of AI4UGE.

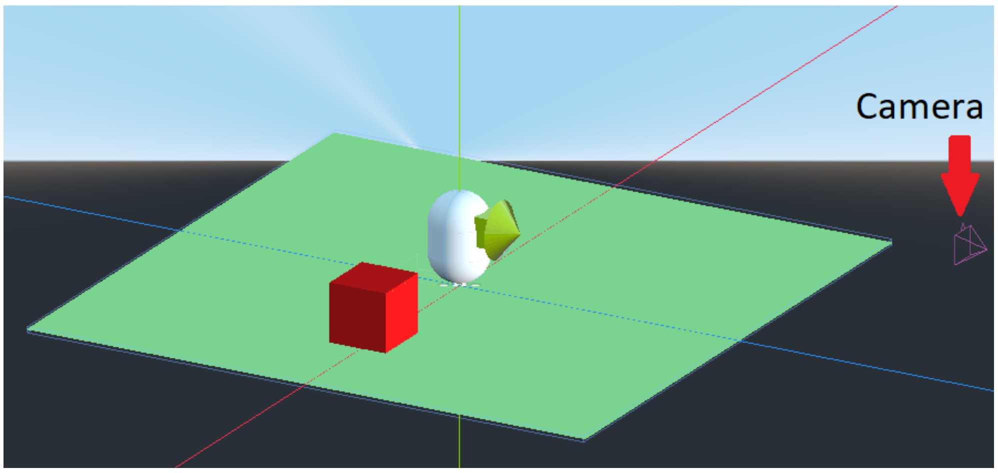
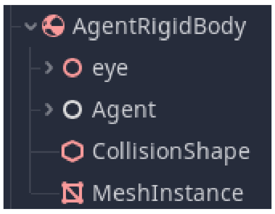


Figure 1. A very simple scene modeled using Godot.

The modeling of the environment in *Godot was* designed to favor the architecture of this game engine. The environment can be modeled within any game, except that aspects of agent interaction with the environment must be specified. Therefore, the next step to understanding environment modeling is to understand agent modeling.

Agent Modeling

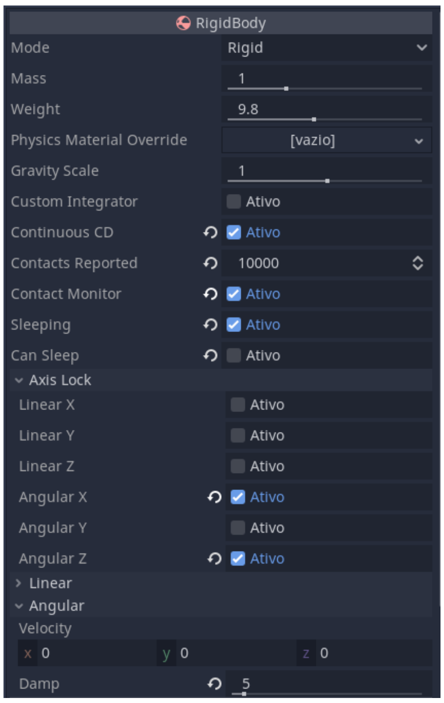
The first step in creating an agent with a physical body is to create a hierarchy with a *node of type RigidBody as* the root of that hierarchy, as shown in Figure 2.



```
AgentRigidBody
  eye
  Agent
  CollisionShape
  MeshInstance
```

Figure 2. Setting up an Agent in Godot.

The *AgentRigidBody*Figure 2) is an object of *type RigidBody and* has four child nodes: *eye*, *Agent*, *MeshInstance*, and *CollisionShape*. The last nodepen serves to give a physical shape to the object and the latter serves to create a visual and geometric representation of this physical form. In short, creating *MeshInstance and CollisionShape nodes* as children of RigidBody is the way to model a rigid body in Godot. In addition, in the event that a rigid body interacts with the environment and produces the behavior expected by the AI4UGE,it is necessary to properly configure the s parameters of the *RigidBody class*. Figure 3 shows anFigure 3 of proper configuration of a rigid body that can be behaved like an NPC controlled by the *RBMoveActuator actuator actuator*.



RigidBody

Mode	Rigid
Mass	1
Weight	9.8
Physics Material Override	[vazio]
Gravity Scale	1
Custom Integrator	Ativo
Continuous CD	✓ Ativo
Contacts Reported	10000
Contact Monitor	✓ Ativo
Sleeping	✓ Ativo
Can Sleep	Ativo
Axis Lock	
Linear X	Ativo
Linear Y	Ativo
Linear Z	Ativo
Angular X	✓ Ativo
Angular Y	Ativo
Angular Z	✓ Ativo
Linear	
Angular	
Velocity	x 0 y 0 z 0
Damp	5

Figure 3. Example of proper settings of a rigid body.

The *Agent node represents* the agent's "brain". Becausewe create an agent, we need to create an object of type *BasicAgent*. To do this, we first add a *Node of type Node* as the child of the *AgentRigidBody* object, then we associate it with the *BasicAgent class script*. This *script* will take care of connecting the sensors and actuators to the rigid body. An object of type *BasicAgent* depends on 4 types of components to work:

- Actuator;
- Sensor;
- ControlRequestor; and
- Brain.

These components are created in the same way that we created *the Agent object*, but we associate them with *different scripts*, for example, an actuator node can be *of type RBMoveActuator* and a sensor example is the *RaycastingSensor class*. To be recognized as part of the agent, these components must be added as children of the BasicAgent node. These components will be explained separately.

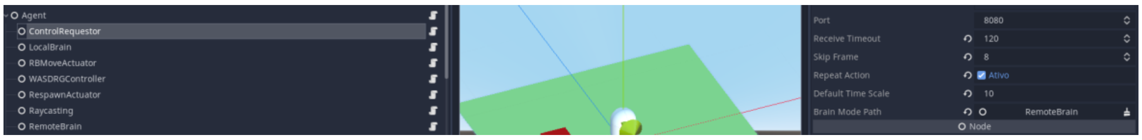
A *ControlRequestor* class

An agent has sensors and actuators and, in addition, a form of time counting of its own, synchronized with the physical time of the physics engine of the game. *Godot* provides the *_PhysicsProcess method* of updating the physics of the game. That is, *_PhysicProcess* performs the physical update loop of the game. Within this physical update loop of the game, we rotate the decision-making loop, represented by the *ControlRequestor class*. In analogy to a biological brain, *ControlRequestor* functions as the hypothalamus and the Agent node of type *BasicAgent* as the brain itself.

During the physical loop of *the ControlRequestor*, a decision cycle occurs. In a decision cycle, the agent sends a message to a *Brain-type object* requesting some control information (synchronization of environment data or actions) and receives back control information about the environment or an action. Therefore, a decision cycle can occur over multiple iterations of the game's physical update loop. The exact way this decision cycle operates depends on seven attributes of the *ControlRequestor class*:

Attribute	Description
Skip Frame	The number of physics loop iterations (executions of the *_PhysicProcess method*) that should be ignored between decision cycles.
Repeat Action	Whether the action chosen at the beginning of a decision cycle must be repeated in the ignored iterations (defined by the *skip frame attribute*) of the physics loop.
Default Time Scale	The value of the *Engine.TimeScale attribute*. This tends to decrease the time interval between physical iterations. See more about this in godot's *official documentation*.
Brain Mode Path	The decision loop execution mode: remote (loads *the RemoteBrain script*) or local (loads the *LocalBrain script*). If the *RemoteBrain script* is loaded, a UDP-based communication protocol opens with a remote *script*, usually encoded in *Python*.
Host	The IP of *the* remote script that controls the agent (only valid if brain *mode path* points to *RemoteBrain*).
Port	The remote *script port* that controls the agent (only valid if Brain *Mode Path points* to *RemoteBrain*).

Figure 4shows the properties of the ControlRequestor *used* in the *AI4UGTesting sample project*.



Agent
- ControlRequestor
- LocalBrain
- RBMoveActuator
- WASDRGController
- RespawnActuator
- Raycasting
- RemoteBrain

Port 8080
Receive Timeout 120
Skip Frame 8
Repeat Action ✓ Ativo
Default Time Scale 10
Brain Mode Path RemoteBrain
Node

Figure 4. Example of configuration of the ControlRequestor.

gilzamir@outlook.com

The *BasicAgent/ControlRequestor pair* establishes a call order for the Standard Methods of AI4UGE:

1. *The BasicAgent OnSetup* method runs once in scenario creation and performs *the OnSetup* method ofthe agent components, in this order: reward functions (*RewardFunc* objects),sensors (*Sensor* objects), and actuators (*Actuator objects*). Only the child active components of the *BasicAgent node* are effectively added to the agent.
2. Whenever an agent is rebooted (*the Reset method*) of the agent is called, all restartable components (usually sensors, actuators, and reward functions) are rebooted (the *OnReset function of* these components is performed).
3. At the beginning of a decision cycle, the agent triggers the sensors by calling *the Get<Type>Value() method*, *where <Type>* can be one of the supported sensor types: *Float*, *FloatArray*, *Bool*, *Int*, *IntArray*, *String*, *and ByteArray*.
4. After all sensor data is captured, this data is sent to a controller defined by a *Brain* object (*LocalBrain or RemoteBrain*). The controller sends back an action. This action has a name. An actuator that matches the action name fires. This means that the *actuator's Act* method fires at each physical cycle during the agent decision cycle (if *the Repeat Action property* is enabled) or only once at the beginning of the cycle (if *the Repeat Action property* is not enabled)." The triggered actuator canreceive r data sent by the controller by calling appropriate agent methods. For example, the *GetStateAsFloatArray() method of* agent class instances *returns* the data sent by the controller as an arrangement of actual numbers.
5. After all actions in the decision cycle are performed (usually one), the *OnUpdate method* is performed for each function of recompensto (*objects of the RewardFunc type*) of the agent. This is necessary for agent reward to be calculated as a consequence of the actions performed, maintaining the consistency of reinforcement learning algorithms that are based on the agent's decision cycle.

A programmer can also add event controllers available in *BasicAgent* that allow you to add sensor and actuator behavior and any other *Godot objects* between the steps presented:

Event	**Description**
beforeTheResetEvent	Event before any agent component is rebooted. Can be used to configure properties required for agent startup.
endOfEpisodeEvent	Event that occurs when an episode ends (*Agent Done* property changes to *true*).
beginOfEpisodeEvent	Event that occurs on the startup (and restart) of an agent.
endOfStepEvent	Event that occurs at the end of an agent decision cycle.
beginOfStepEvent	Event that occurs at the beginning of an agent decision cycle.
beginOfUpdateStateEvent	An event that occurs throughout the cycle before the sensors produce any value.
endOfUpdateStateEvent	Event that occurs in all cycles after all sensors have produced their cycle values.
beginOfApplyActionEvent	Event that occurs before any action is performed.

endOfApplyActionEvent	Event that occurs after all actions have been performed.

Actuators

An agent (*type BasicAgent object) must* have one or more actuators (*Actuator objects) before* it can function properly. By default, the Ge *AI4Ucontains* an *RBMoveActuator class* (which inherits from *Actuator*) that allows you to create instances that move objects of type *RigidBody*. The developer can create their own actuators by creating a class that inherits from *Actuator*.

A Classe *Actuator*

The *Actuator class* provides an abstraction to the meaning of anaction that affects the environment. In addition, this class provides the general information of theaction: the name, type, and shape.

The actuator name is unique (two actuators cannot have the same name), because it is through the name that determines what action the agent must perform at any given time. The actuator name is represented by *the actionName attribute* and can be defined both in the Godot editor property panel in a concrete class that inherits from *the Actuator* quanto class in the *OnSetup method* of the child class.

The type defines the type of data that the actuator receives to perform the action it represents and is represented by the *isContinuous attribute*. This attribute is protected and can only be defined by inheritance in the concrete classes that inherit from *Actuator*. It is recommended to set this value in the method or in the child class constructor in the *OnSetup method*. The type will always be numeric and can be continuous or non-continuous. For example, an actuator can operate by receiving real signals that indicate degrees of rotation or strength intensities that must be applied to a rigid body. Another case is the actuator represents a categorical action, such as choosing an option from many available options.

The actuator shape defines how the data on which the actuator operates is organized and is represented by the protected shape *attribute*. This attribute is of type *int[] and* must be instantiated along with the actuator type in the constructor or in the *OnSetup* child class method. For example, an actuator that produces rigid body movements can receive data in the form (4,) that represents an *array* of four elements, such as the array$[f, t, j, jf]$, f where it represents the intensity of the forward/backward movement, t represents the rotation allocated angle of the rigid body around the axis itself, j represents a vertical jumpjf, and , a forward jump. This information can be used by a controller to produce the sensor input data accordingly.

The AI4UGE has a *builtin actuator* called *rbmoveactuator.* This actuator is capable of controller objects of type *RigidBody*.

A class *RBMoveActuator*

An object of type *RBMoveActuator* controls objects of type *RigidBody*, this is why it has the *acronym RB of* the beginning of its name. In addition, as in Godot a *RigidBody object has* four different modes, it is important to note that the *RBMoveActuator actuator* is specifically designed to control *RigidBody objects* in *RigidBody mode*. (See the field *mode* in Figure 3. Therefore, it is not recommended to use this type of actuator *in a RigidBody* with another mode type.

Therefore, to add an *RBMoveActuator to* the agent, below the Agent node (see Figure 2), you create a *node of type Node* and associate it *with the script of* the *RBMoveActuator class*.

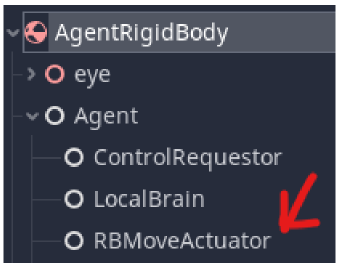


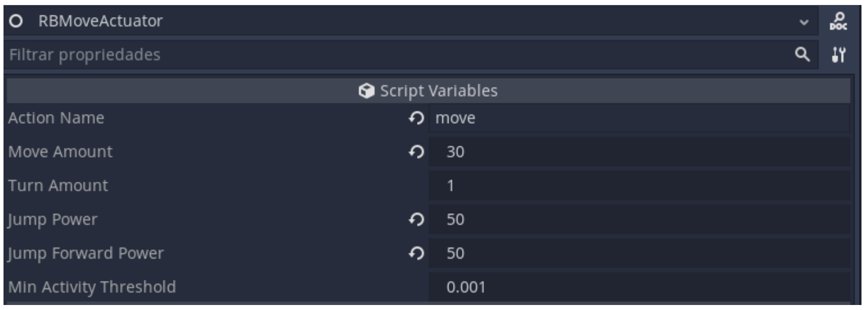
Figure 5. Example where an RBMoveActuator object was added to the agent.

An object of type *RBMoveActuator* has six fields:

Field	Description
Action Name	It must be a unique name for each actuator, as it is through this name that the actuator will be recognized by the agent.
Move Amount (ma)	The amount of forward movement.
Turn Amount (ta)	The intensity of the spin. It can be a value that varies continuously between -1 (left swivel) and 1 (right swivel).
Jumper Power (jp)	The maximum intensity of the jump (vertical only).
Jumper Forward Power (jpf)	The intensity of the leap forward.

An agent with an actuator of this type can receive actions *of the form [f, t, j, jf], where f* > 0 represents a forward movement (positive *direction of the object's z-axis*) *and f < 0* represents the opposite movement; *t* > 0 represents right-hand turn and *t < 0* represents left-hand turn; *j* represents the intensity of the jump and must be greater than or equal to zero; *and jf* represents a forward jump, it must also be greater than or equal to zero. The final intensity of the movements depends on the values of the *fields of the RBMoveActuator object* and will be equal to *[f*ma, t*ta, j*jp), jf *jpf]*, where * represents the scalar multiplication.

Finally, it is recommended that the *action name* be set to "move", because this is the name expected by the driver examples available in AI4U. Figure 5 displays a Figure 6 example of configuration of this type.


RBMoveActuator

Filtrar propriedades

Script Variables

Action Name	move
Move Amount	30
Turn Amount	1
Jump Power	50
Jump Forward Power	50
Min Activity Threshold	0.001

Figure 6. Configuration of an actuator type RBMoveActuator.

The Sensor Class

An agent must have a way of perceiving the world. For this, we provide the *Sensor class*, which allows the agent to understand the properties of the environment in which it operates. The

Sensor class provides general attributes present on all sensors. These attributes can be defined in godot's *own* editor, but can also be defined in code through inheritance. In fact, the *Sensor class* does not represent a specific sensor, but rather agglomerates the general properties of sensors and methods that return all data types supported by AI4U. The general attributes are five:

Attribute	**Description**
perceptionKey	Unique sensor identifier. This information will be used by the agent to determine certain type of information expected by the agent.
stackedObservations	The amount of information that the agent receives in the decision cycle *t*. This amount must be greater than or equal to 1. If it is 1, the agent receives in the decision cycle *t only* the information captured at the beginning of the cycle. If it *is k > 1*, in *the decision cycle t*, the agent also receives the information captured in *the decision cycles t, t-1, t-2, ..., t-k+1*.
isActive	Turns the sensor on or off. If the sensor is disabled before the start of the game, the agent ignores this sensor while running the game. This may require changes to *the* agent's remote control script, because from the point of view it is external, this sensor is no longer visible.
normalized	Tells the sensor programmer whether or not the data should be normalized.
resetable	Indicates whether the *Sensor's* OnReset method should run when the agent is rebooted.

In addition to the public and external attributes (which can be modified in the Godot editor interface), a sensor has four other attributes that must be defined in the *sensor's OnSetup constructor* or method:

1. *Type*: The type of the sensor, which is of the SensorType enumeration *type*. *SensorType* can be:
    a. *SensorType.sint,*
    b. *SensorType.sfloat,*
    c. *SensorType.sbool,*
    d. *SensorType.sstring,*
    e. *SensorType.sfloatarray,*
    f. *SensorType.sbytearray,*
    g. *Sensortype.sintarray.*
2. *IsState*: Indicates whether sensor information should be incorporated into the state of the environment or indicates only an initial configuration of the environment (something that does not change over time).
3. *Shape*:The dimensions s and their respective sizes. If the sensor produces only one scalar value, the shape is empty, represented by an empty instance of an entire arrangement: *new int[0]*.
4. *agent*: a reference to the agent that the sensor is the owner. Only one agent can receive data from one sensor at a time.

In addition to these properties, when creating a sensor, you must implement the method that returns the data corresponding to the sensor type. For example, if the sensor is *of type*

SentorType.sfloatarray, you must implement the *GetFloatArrayValue*() method that returns an arrangement of actual numbers. The possible methods for each type of dataare :

Guy	**Method**
SensorType.sfloat	GetFloatValue()
SensorType.sstring	GetStringValue()
SensorType.sbool	GetBoolValue()
SensorType.sbytearray	GetByteArrayValue()
SensorType.sint	GetIntValue()
SensorType.sintarray	GetIntArrayValue()
SensorType.sfloatarray	GetFloatArrayValue()

AI4UGE provides some built-in sensors: *ActionSensor, DoneSensor, FloatArrayCompositeSensor, IDSensor, OrientationSensor, PositionSensor, RayCastingSensor, RewardSensor* and *StepSensor.*

ActionSensor: This sensor returns the last(s) action performed by the agent. In some learning problems by reinforcement, the history of actions facilitates the learning of the problem. This method is of type *SensorType.sfloatarray* and the amount of actions can be set through the *actionSize* (*Action Size*) property in the Godot editor.

DoneSensor: This sensor returns a boolean indicating whether the episode is over or not. It is of the *SensorType.sbool type* and does not support stacking observations, as it serves only to indicate to controllers that an episode has ended or not. The default value of the *perceptrionKey property* of this sensor is *done*. Every BasicAgent agent *has* this sensor.

FloatArrayCompositeSensor: This object behaves like a sensor, but is an aggregator of various types. This sensor is used to create a composition of sensors. To do this, you must add one or more *child nodes* d and a type that inherits from *the Sensor* to the FloatArrayType. The information of these sensors is converted into real numbers and placed in an arrangement of real numbers containing the values of all aggregated sensors. Therefore, this object must have one or more sensor children. Children can be of any numeric type. The type of this sensor *is SensorType.sfloatarray*. The Python *driver scripts* available in the sample directory assume that the *perceptionKey property* of this sensor has the array *value*.

IDSensor: Returns the agent identifier in the environment. This sensor returns a unique agent identification code in the environment and is available to all agents of type *BasicAgent*. The type of this sensor is *SensorType.sstring* and the default value of the *perceptionKey property* is *id*. This sensor cannot stack information over several decision cycles.

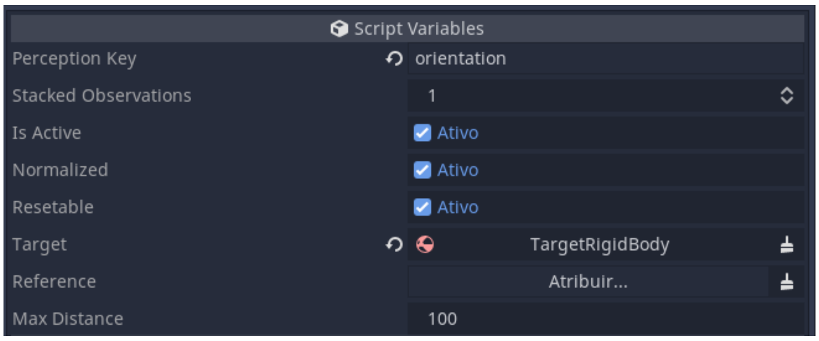
OrientationSensor: Returns two actual numbers that indicate the relative orientation and distance of the agent to a target defined by the target *property*. The orientation is calculated as the cosine of the angle between the vectors **u and v**, **in which u represents** the direction between the agent and the target **and v represents** the direction of the agent (axis that indicates the direction of the agent's axis of view). The distance is the Euclidean distance between the agent and the target. This sensor proved to be efficient for the generation of spatial navigation behaviors. The sensor type is *SensorType.sfloatarray*, which returns an arrangement$[o, d]$, where o it is orientation and d is the distance of the agent from thelvo. The value of the *perceptionKey* property is usually set by the user in the *Godot* interface, as shown in Figure 7.

PositionSensor: Returns the position of the agent at local coordinates or global coordinates. The sensor is of type *SensorType.sfloatarray and* returns an arrangement of real numbers in the form [x, y, z].

RayCastingSensor: Returns an *mxn dimension array* containing code from objects detected in the agent's field of view. Rays are cast in perspective from a position set on the eye *property of* the sensor. For each object intersected by the radius, either the object code or the agent distance to the object is recorded in the array at the position corresponding to the intersection of the radius in the plane between the observer and the object in the scene. This sensor is sensortype.sfloatarray *type and produces* an arrangement with m *x n elements*. In the controller, this arrangement is usually put back into matrix format.

RewardSensor: This sensor returns the last reward produced by the user. The type of this sensor is *SensorType.sfloat* and does not support the *stackedObservations property*.

StepSensor: Returns the agent's current decision cycle. The type of this sensor *is SensorType.sint*. This sensor does not support the *stackedObservations property.*


Script Variables

Perception Key	orientation
Stacked Observations	1
Is Active	✓ Ativo
Normalized	✓ Ativo
Resetable	✓ Ativo
Target	TargetRigidBody
Reference	Atribuir...
Max Distance	100

Figure 7. Example of setting up an OrientationSensor sensor.

As Classes *Brain*, *RemoteBrain* e *LocalBrain*

The *ControlRequestor component* depends on a Brain-type *object*, which can be *RemoteBrain* or *LocalBrain.* The Type of *Brain* defined in *ControlRequestor* determines the controller type of the agent.

A RemoteBrain *Brain*

A RemoteBrain type *brain allows* the agent to have a remote controller, implemented in any language that implements the AI4U's private protocol. For now, we keep this protocol implemented only in the Python language and tested specifically in Python *3.7 version*.

A Brain Of The *LocalBrain Type*

A *LocalBrain-type Brain*, *on the* other hand, allows the *agent to be controlled by scripts implemented in the game engine itself*, using C# or another language supported by *the game engine.* A *LocalBrain object* depends on an object of a type that inherits from *ai4u. Controller.*

For now, the only concrete controller built into the AI4UGE is *the ai4u. WASDRBMoveController*, which allows you to control the agent using *the WASD* keys on the keyboard of a PC or notebook.

However, the user could implement their own controller to, for example, run a neural network implemented directly in C#. This would allow you to train a controller using Python and use the neural network after being trained without the need to maintain the connection to the Python language. This is a future feature that we intend to add to AI4UGE.

Creating Your Own *Controller*

A controller needs to overwrite basically two methods: *GetAction* and *NewStateEvent*. The *GetAction method* runs when the *ControlRequestor* sends a control request, so *GetAction* must return an encoded action on a *string*. The programmer can use anaccess methods, such as the various versions of the *ai4u method. Utils.ParseAction*, which transforms a command into its encoding as string.

In the *NewStateEvent* method, the programmer can use several accessory methods (such *as GetStateSize, GetStateName*, and *GetStateAsFloat*) to check the current state of the agent.

While no more extensive documentation is produced, we recommend basing the controller code on the code of the *ai4u class. WASDRBMoveController*.

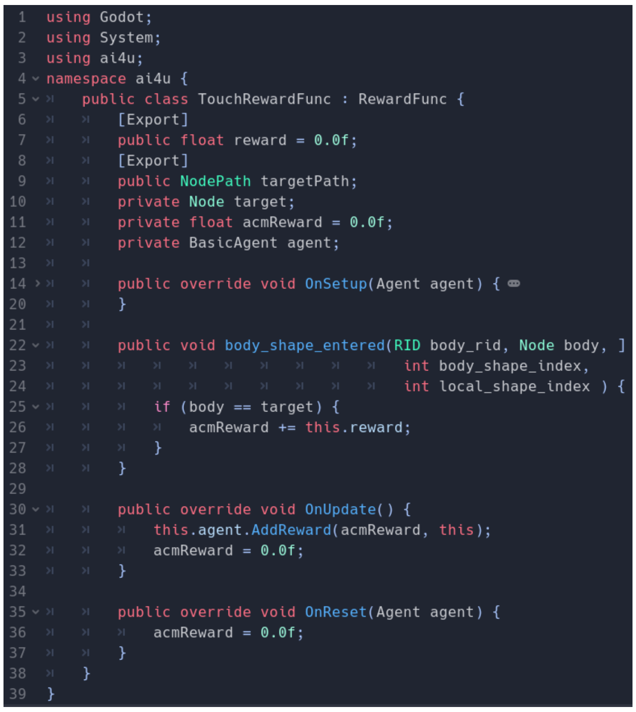
Reward Functions

A centralizing element of machine learning algorithms is the *type* of feedback that is used for controller training. AI4U is provides the basic elements for reinforcement learning.

The *BasicAgent class* provides methods that allow the programmer to add rewards to the agent at any point in an agent decision cycle. However, to ensure the consistency of the decision-loop *and the Markovian property* of a decision process, the reward corresponding to the agent's action in the t-decision cycle *at the* end of the *t-decision cycle should be added*. It is important for each reward to be restricted to the decision cycle in which it was added. There are two ways to do this with AI4U:

- Implement a class that inherits from *ai4u. RewardFunc*; create a node of the type of this class and child of the *BasicAgent* node; in this case, you must overwrite the *OnUpdate method*, which is where you should add agent decision cycle reward; or
- Add an event controller to the *basicAgent endOfStepEvent* event and add the reward on this controller.

The interesting thing about the second possibility is that it is possible to add reward from any component of the agent, whether inside a sensor or actuator created by the user. Figure Figure 8a function that adds a reward when the agent touches a cube. Note that the *variable acmReward* is zeroed (line 32 of its source code) after the reward is added to the agent. This prevents the spread of the reward in subsequent decision cycles.



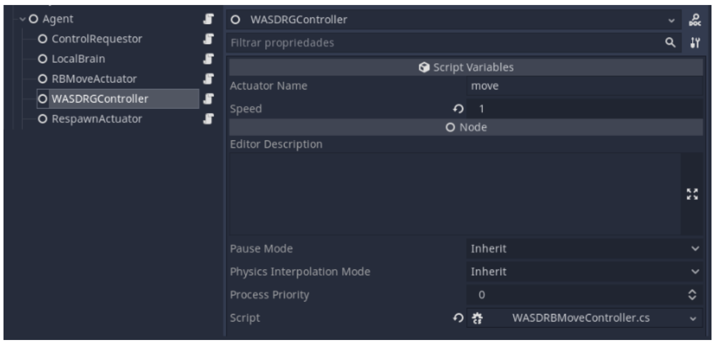
```
using Godot;
using System;
using ai4u;
namespace ai4u {
    public class TouchRewardFunc : RewardFunc {
        [Export]
        public float reward = 0.0f;
        [Export]
        public NodePath targetPath;
        private Node target;
        private float acmReward = 0.0f;
        private BasicAgent agent;

        public override void OnSetup(Agent agent) { ⚭
        }

        public void body_shape_entered(RID body_rid, Node body, ]
                                        int body_shape_index,
                                        int local_shape_index ) {
            if (body == target) {
                acmReward += this.reward;
            }
        }

        public override void OnUpdate() {
            this.agent.AddReward(acmReward, this);
            acmReward = 0.0f;
        }

        public override void OnReset(Agent agent) {
            acmReward = 0.0f;
        }
    }
}
```

Figure 8. Example of reward function.

Using a local Agent controller

Once the agent and all its components have been configured, you can configure *the ControlRequestor* for handling the agent body through a local controller. To do this, you must create a *Node of type Node*, associate it *with the LocalBrain script*, and then configure the *Brain Mode Path property* of *ControlRequestor* to point to this node of the *LocalBrain type*.

The *LocalBrain node must* point to a controller. A controller defines how the agent will behave. To do this, you must create a Node of type *Node* and associate it with the *script* that implements the controller. Let's use as an example *the built-in WASDRBMoveController* controller, which allows the *agent to controller through the WASD keys* of the alpha-numeric telcado. Figure Figure 9shows how this controller was configured in the AI4UGTesting *project*.



```
Agent
    ControlRequestor
    LocalBrain
    RBMoveActuator
    WASDRGController
    RespawnActuator
```

WASDRGController

Filtrar propriedades

Script Variables

Actuator Name	move
Speed	1

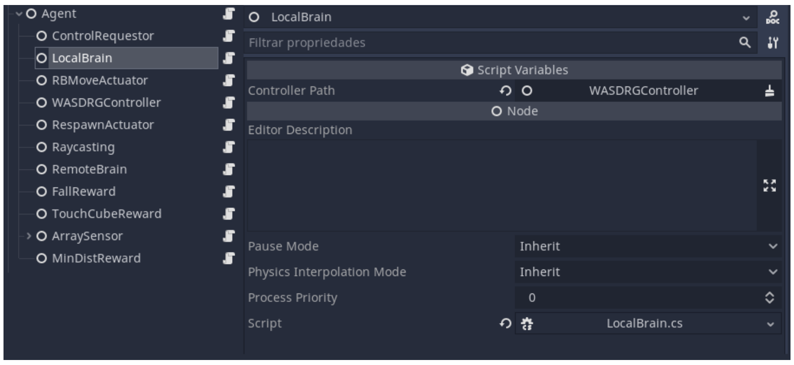
Node

Editor Description

Pause Mode	Inherit
Physics Interpolation Mode	Inherit
Process Priority	0
Script	WASDRBMoveController.cs

Figure 9. Example of setting up a WASDRBMoveController controller. Speed is the agent's speed multiplier when a motion key is pressed.

After the controller is configured, you must configure the *Controller Path property of LocalBrain*Figure 10.

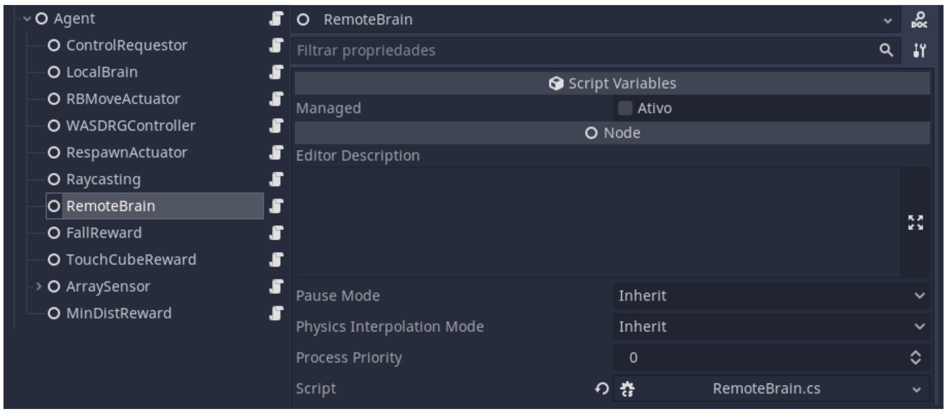

Agent
ControlRequestor
LocalBrain
RBMoveActuator
WASDRGController
RespawnActuator
Raycasting
RemoteBrain
FallReward
TouchCubeReward
ArraySensor
MinDistReward

LocalBrain
Filtrar propriedades
Script Variables
Controller Path WASDRGController
Node
Editor Description
Pause Mode Inherit
Physics Interpolation Mode Inherit
Process Priority 0
Script LocalBrain.cs

Figure 10. Setting up a LocalBrain node.

The use of an Agent remote controller.

You can define an agent remote controller using a *RemoteBrain type object, which* must replace the *LocalBrain object in* the *Brain Mode Path configuration* of the *Agent ControlRequestor* component. To do this, you must first create and configure a *RemoteBrain node*. Figure Figure 11shows the configuration of a *RemoteBrain node*. You create a node of this type in the same way that you create a *LocalBrain node*, except that the *script* used is *RemoteBrain*.


Agent
ControlRequestor
LocalBrain
RBMoveActuator
WASDRGController
RespawnActuator
Raycasting
RemoteBrain
FallReward
TouchCubeReward
ArraySensor
MinDistReward

RemoteBrain
Filtrar propriedades
Script Variables
Managed Ativo
Node
Editor Description
Pause Mode Inherit
Physics Interpolation Mode Inherit
Process Priority 0
Script RemoteBrain.cs

Figure 11. Configuration of a RemoteBrain component.

An object of type *RemoteBrain* connects to a *script on* the local network and receives commands from an external controller, passing these commands to the agent through the *ControlRequestor*. Remote connection settings are made in the *ControlRequestor itself*, as shown in Figure 4. The language supported by AI4U for external control of the agent is Python. This language can be used to train a neural network that controls the agent. The Python language was used because of its ease and availability *of frameworks* and learning tools inwhich they support it.

Agent Training and External Manipulation

External training and manipulation (through tools outside the game *engine)* can be done through Python *scripts*. The way to implement this control is the same for *both Unity* and *Godot*. Therefore, all documentation regarding external control is available in files with extension *. available in the AI4U *doc* directory.

Credits

Gilzamir Gomes (gilzamir@outlook.com): *design* and development.

Eduardo Nogueira (not informed): *game designer*.