AI4U Unity Edition

Guia do Desenvolvedor – Versão 0.5.0 (20 de dezembro de 2022)

Gilzamir Gomes (Programador/Analista)

Eduardo Nogueira (Game Designer)

Sumário

[Introdução 1](#_Toc122465963)

[A Modelagem do Ambiente 2](#_Toc122465964)

[Modelagem do Agente 3](#_Toc122465965)

[A classe *ControlRequestor* 5](#_Toc122465966)

[Atuadores 7](#_Toc122465967)

[A Classe *Actuator* 8](#_Toc122465968)

[A classe *MoveActuator* 8](#_Toc122465969)

[A Classe Sensor 9](#_Toc122465970)

[As Classes *Brain*, *RemoteBrain* e *LocalBrain* 12](#_Toc122465971)

[Um *Brain* do Tipo *RemoteBrain* 12](#_Toc122465972)

[Um *Brain* do Tipo *LocalBrain* 12](#_Toc122465973)

[Funções de Recompensa 13](#_Toc122465974)

[O uso de um controlador local do Agente 13](#_Toc122465975)

[O uso de um controlador remoto de Agente. 14](#_Toc122465976)

[Treinamento e manipulação externa do Agente 15](#_Toc122465977)

[Limitações 16](#_Toc122465978)

[Créditos 16](#_Toc122465979)

# Introdução

AI4U Unity Edition (AI4UUE) permite controlar personagens virtuais na Unity. Isso pode ser feito para a criação de experimentos de inteligência artificial interessantes. O treinamento dos personagens pode ser realizado usando *frameworks* como *stable-baselines3* ou qualquer outro *framework* baseado em Python. Inclusive, pode-se usar qualquer algoritmo de controle e resolução de problemas implementado em Python. Com isso, você pode usar o potencial de *frameworks* de aprendizado de máquina que proveem uma interface em Python na criação de modelos de controladores de NPCs (*Non-Player Characters*). Pode-se conectar qualquer script em Python baseado em AI4UUE para controlar itens de jogos.

AI4UUE foi refinada para o desenvolvimento de experimentos de desenvolvimento de NPCs usando inteligência artificial. Para isso, fornece uma forma lúdica e transparente de modelagem de NPCs por meio de uma abstração de agentes inteligentes. Para isso, sobre a Unity, implementamos o *Agent Abstraction Framework* (AI4U-AAF). Com este framework, você constrói o NPC adicionando atuadores e sensores, como se estivesse construindo um robô, contudo, com um corpo virtual. Isso nos provê, além de um escopo de experimentação, um apelo educacional poderoso. O AI4U-AFF também possibilita a criação de componentes que podem ser reutilizados em diversos experimentos/jogos.

Portanto, as principais características de AI4UUE são:

* desenvolvimento de controles de jogos baseados no paradigma de agentes em Inteligência Artificial;
* *multi-engine*, suporta jogos modelados na *Unity* ou na *Godot*, sem a necessidade de reprogramação dos scripts Python; e
* Modelagem de ambiente baseado no framework *Gym*.

# A Modelagem do Ambiente

A modelagem do ambiente, apesar do apelo visual, deve enfatizar os aspectos físicos do ambiente, como obstáculos, pisos, construções, passagens e colisões. A AI4UUE possui sensores de colisão com as formas padrões de corpo rígido disponíveis na *Unity*. Portanto, essa modelagem é muito dependente do motor de jogos que você usa.

*Unity* possui uma interface agradável para modelagem do ambiente. Na Figura 1, pode-se visualizar uma cena com alguns elementos: paredes, um cubo azul (que representa o alvo), um cubo vermelho (que representa radiação), uma câmera e um piso (um terreno plano simples). A radiação pode ser perceba pelo agente por meio de um *sensor* específico criado por meio de *scripts* da Unity. O que não se pode ver é o motor de física que adiciona gravidade ao ambiente, fixa o piso como um chão impedindo os objetos de caírem para sempre e permite que objetos colidam entre si. Todos estes elementos estão presentes na cena *Donut* [[1]](#footnote-1)(disponível como uma amostra no pacote de instalação da AI4UUE). Iremos nos basear neste projeto para explicar os diferentes componentes da AI4UUE.

Tela de computador com página de internet informando algo

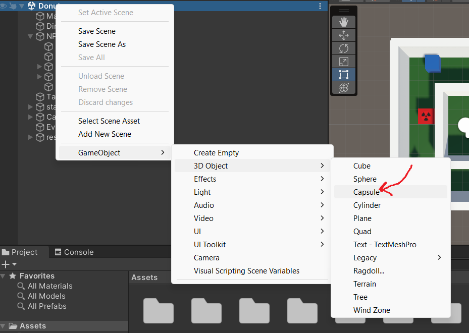
Descrição gerada automaticamente com confiança média

Figura . Uma cena muito simples modelada usando Unity.

O ambiente pode ser modelado como em qualquer jogo, exceto que os aspectos de interação do agente com o ambiente devem ser especificados. Portanto, o próximo passo para se compreender a modelagem do ambiente é compreender a modelagem do agente.

# Modelagem do Agente

O primeiro passo para a criação de um agente com um corpo físico é criar uma hierarquia com um objeto de jogo (*GameObject*) como raiz dessa hierarquia, como mostrado na Figura 2. Nesta figura, à esquerda é mostrado como criar a capsula que representa o corpo do agente; à direita, é mostrada a hierarquia com o objeto de jogo que representa o agente nomeado de NPC. Esse objeto pode ter um corpo físico (usamos uma capsula para representar o corpo físico do agente. Não há restrição *à priori* para o objeto de jogo do agente, contudo, AI4UUE foi testada apenas com objetos que tem um corpo rígido (*Rigidbody*) e um colisor (*Collider*). Uma capsula na Unity é criada automaticamente cum *CapsuleCollider*, que é um colisor no formato de uma capsula.

Texto

Descrição gerada automaticamente

Figura . Configuração de um Agente na Unity.

A Unity é baseada em componentes. Pode-se assim adicionar vários *scripts* a objetos de jogos. *Scripts* são componentes criados pelo usuário/desenvolvedor de jogo. Mas também há componentes pré-definidos na Unity, como o componente *Rigidbody*, que faz com que o objeto de jogo tenha um corpo rígido para interagir com o mundo físico. A Figura 3 mostra, na janela *inspector*, a configuração inicial do objeto NPC. Este objeto ainda não é um agente, pois não adicionamos os componentes (*scripts*) que transformam esse objeto em um agente.

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Figura . Exemplo de configurações adequadas de um corpo rígido.

Para transformarmos o objeto NPC em um agente, precisamos associá-lo ao *script* *ai4u.BasicAgent*, que representa o “cérebro” do agente. Portanto, para criarmos um agente, adicionamos um componente *BasicAgent* ao objeto NPC *AgentRigidBody*. Na janela *inspecto*, clique em *Add Component* e selecione o *script BasicAgent*. Este *script* vai se encarregar de conectar os sensores e atuadores ao corpo rígido. Um objeto do tipo *BasicAgent* depende de 4 tipos de componentes para funcionar:

* Actuator;
* Sensor;
* ControlRequestor; e
* Brain.

Estes componentes são criados da mesma forma que criamos o objeto NPC, mas os associamos a *scripts* diferentes, por exemplo, um objeto atuador pode ser do tipo *ai4u.MoveActuator* e um exemplo de sensoré a classe *ai4u.RaycastingSensor*. Para serem reconhecidos como parte do agente, os sensores podem ser adicionados como filhos de NPC(desde que a propriedade *Children Sensors* de *BasicAgent* esteja ativa). Alternativamente, os sensores podem ser adicionados manualmente na lista de sensores (apresentada no campo *Sensors*) do componente *BasicAgent.* Já os atuadores somente podem ser adicionados na lista de atuadores (apresentada no campo *Actuators*) do componente *BasicAgent*. Quando o componente *BasicAgent* é adicionado a um objeto de jogo, automaticamente vários outros componentes (sensores e atuadores essenciais) são criados automaticamente.

Na Figura 4, mostra-se todas as configurações do agente NPC da cena Donut e todos os outros componentes adicionados. É importante observar que os componentes *RemoteBrain* e ControlRequestor poderiam ser adicionados em outro objeto de jogo. No caso do componente *RemoteBrain* ser definido em outro objeto, a propriedade *Agent* do componente *RemoteBrain* deve apontar para o agente a ser controlado. Já no caso do *ControlRequestor*, o campo *Control Requestor* de *BasicAgent* deve apontar para o objeto que contém um componente do tipo *ControlRequest* (no caso da Figura 4, este objeto é o próprio NPC).

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

Figura . Componentes do agente NPC na cena Donut.

Nas próximas seções, os principais componentes de um agente são explicados separadamente.

## A classe *ControlRequestor*

Um agente possui sensores e atuadores e, além disso, uma forma de contagem de tempo própria, sincronizada com o tempo físico do motor de física do jogo. *Unity* provê o método *FixedUpdate* de atualização da física do jogo. Ou seja, *FixedUpdate* executa o laço de atualização física do jogo (por exemplo, a 60 fps). Dentro deste laço de atualização física do jogo, rodamos o laço de tomada de decisão, representado pela classe *ControlRequestor*. Em analogia a um cérebro biológico, o componente associado ao *script* *ControlRequestor* funciona como o hipotálamo e componente do agente (associado ao *script* *BasicAgent*) como o próprio cérebro.

Durante o laço físico do *ControlRequestor*, ocorre um ciclo de decisão. Em um ciclo de decisão, o agente envia uma mensagem para um objeto do tipo *Brain* requisitando alguma informação de controle (sincronização de dados do ambiente ou de ações) e recebe de volta informações de controle sobre o ambiente ou uma ação. Portanto, um ciclo de decisão pode ocorrer ao longo de várias iterações do laço de atualização física do jogo. A forma exata como este ciclo de decisão opera depende de um atributo da classe *ControlRequestor* e de 4 atributos da classe *Brain*:

|  |  |  |
| --- | --- | --- |
| **Atributo** | **Classe** | **Descrição** |
| *Skip Frame* | *Brain (LocalBrain ou RemoteBrain)* | O número de iterações do laço de física (execuções do método *FixedUpdate*) que deve ser ignorado entre ciclos de decisão. |
| *Repeat Action* | *Brain (LocalBrain ou RemoteBrain)* | Se a ação escolhida no início de um ciclo de decisão deve ser repetida nas iterações ignoradas (definidas pelo atributo *skip frame*) do laço de física. |
| *Default Time Scale* | *ControlRequestor* | O valor do atributo Time*.timeScale*. Isso tende a diminuir o intervalo de tempo entre iterações físicas. Veja mais sobre isso na documentação oficial da Unity. |
| *Host* | *Brain (RemoteBrain)* | O IP do *script* remoto que controla o agente (apenas válido se o *Brain Mode Path* apontar para *RemoteBrain*). |
| *Port* | *Brain (RemoteBrain)* | A porta do *script* remoto que controla o agente (apenas válido se o *Brain Mode Path* apontar para *RemoteBrain*). |

O modo de execução do laço de decisão pode ser remoto ou local. Para ser remoto, na cena tem que ter um objeto do tipo *RemoteBrain* habilitado. Para ser local, a cena tem que ter um objeto do tipo *LocalBrain* habilitado. É importante habilitar apenas um *RemoteBrain* ou apenas um *LocalBrain*. Caso dois objetos do tipo *Brain* sejam habilitados na cena, o comportamento é indeterminado. Se o *script RemoteBrain* for carregado, um protocolo de comunicação baseado em UDP é aberto com um *script* remoto, geralmente codificado em *Python*.

Na Figura 5, mostra-se as propriedades do *ControlRequestor* e do *Brain* (*RemoteBrain*) usada na cena Donut.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Figura . Exemplo de configuração do ControlRequestor.

O par *BasicAgent/ControlRequestor* estabelece uma ordem de chamada dos métodos padrões da AI4UUE:

1. O método *OnSetup* de *BasicAgent* é executado uma vez na criação do cenário e realiza a execução do método *OnSetup* dos componentes do agente, nesta ordem: funções de recompensa (objetos do tipo *RewardFunc*), sensores (objetos do tipo *Sensor*) e atuadores (objetos do tipo *Actuator*). Apenas os componentes ativos filhos do nó *BasicAgent* são adicionados efetivamente ao agente.
2. Sempre que um agente é reinicializado (o método *Reset*) do agente é chamado, todos os componentes reinicializáveis (geralmente sensores, atuadores e funções de recompensa) são reinicializados (a função *OnReset* destes componentes é executada).
3. No início de um ciclo de decisão, o agente aciona os sensores chamando o método *Get<Type>Value()*, onde *<Type>* pode ser um dos tipos de sensores suportados: *Float*, *FloatArray*, *Bool*, *Int*, *IntArray*, *String* e *ByteArray*. Os dados são coletados e enviados em um dicionário (indexado pelo campo *perceptionKey* do sensor) para o controlador (que pode ser local ou remoto).
4. Depois que todos os dados dos sensores são coletados, estes dados são enviados para um controlador definido por um objeto do tipo *Brain* (*LocalBrain* ou *RemoteBrain*). Como resposta, o controlador envia de volta uma ação. Esta ação possui um nome (definido pelo atributo *actionName*). Um atuador que combina com o nome da ação é acionado. Isso significa que o método *Act* do atuador é acionado a cada ciclo físico durante o ciclo de decisão do agente (se a propriedade *RepeatAction* estiver habilitada) ou apenas uma vez no início do ciclo de decisão (se a propriedade *RepeatAction* não estiver habilitada). O atuador acionado pode receber os dados enviados pelo controlador chamando métodos adequados do agente. Por exemplo, o método *GetStateAsFloatArray()* de instâncias da classe *Agent* retorna os dados enviados pelo controlador como um arranjo de números reais.
5. Depois que todas as ações do ciclo de decisão forem executadas (geralmente uma), o método *OnUpdate* é executado para cada função de recompensa (objetos do tipo *RewardFunc*) do agente. Isso é necessário para que a recompensa do agente seja calculada como consequência das ações executadas, mantendo a consistência de algoritmos de aprendizado por reforço que se baseiam no ciclo de decisão do agente.

Um programador também pode adicionar controladores de evento disponíveis em *BasicAgent* que possibilitam acionar comandos (por exemplo, adicionar recompensa) em qualquer um dos momentos que o evento acionado representa. Os principais eventos são:

|  |  |
| --- | --- |
| **Evento** | **Descrição** |
| *beforeTheResetEvent* | Evento antes de qualquer componente do agente ser reinicializado. Pode ser usado para configurar propriedades necessárias para a inicialização do agente. |
| *endOfEpisodeEvent* | Evento que ocorre quando um episódio termina (propriedade *Done* do agente muda para *true*). |
| *beginOfEpisodeEvent* | Evento que ocorre na inicialização (e reinicialização) de um agente. |
| *endOfStepEvent* | Evento que ocorre no final de um ciclo de decisão do agente. |
| *beginOfStepEvent* | Evento que ocorre no início de um ciclo de decisão do agente. |
| *beginOfUpdateStateEvent* | Evento que ocorre em todo ciclo antes dos sensores produzirem qualquer valor. |
| *endOfUpdateStateEvent* | Evento que ocorre em todos os ciclos depois que todos os sensores produziram seus valores de ciclo. |
| *beginOfApplyActionEvent* | Evento que ocorre antes de qualquer ação ser executada. |
| *endOfApplyActionEvent* | Evento que ocorre depois de todas as ações terem sido executadas. |

## Atuadores

Um agente (objeto do tipo *BasicAgent*) deve ter um ou mais atuadores (objetos do tipo *Actuator*) para poder funcionar adequadamente. Por padrão, a *AI4UUE* contém uma classe *MoveActuator* (que herda de *Actuator*) e que permite criar instâncias que movimentam objetos que agem como corpo rígido (associados ao componente *Rigidbody*). O desenvolvedor pode criar seus próprios atuadores criando uma classe que herda de *Actuator*.

### A Classe *Actuator*

A classe *Actuator* provê uma abstração para o significado de uma ação que afeta o ambiente. Além disso, esta classe provê as informações gerais da ação: o nome, o tipo e a forma.

O nome do atuador é único (dois atuadores não podem ter o mesmo nome), pois é por meio do nome que se determina qual a ação o agente deve executar em determinado momento. O nome do atuador é representado pelo atributo *actionName* e pode ser definido tanto no painel de propriedades do editor da Unity em uma classe concreta que herda da classe *Actuator* quanto no método *OnSetup* da classe filha.

O tipo define o tipo de dado que o atuador recebe para executar a ação que representa e é representado pelo atributo *isContinuous*. Este atributo é protegido e pode apenas ser definido por herança nas classes concretas que herdam de *Actuator*. Recomenda-se definir este valor no método ou no construtor da classe filha ou no método *OnSetup*. O tipo sempre será numérico e pode ser contínuo ou não contínuo. Por exemplo, um atuador pode operar recebendo sinais reais que indicam graus de rotação ou intensidades de força que devem ser aplicadas em um corpo rígido. Outro caso é o atuador representa uma ação categórica, como escolher uma opção dentre muitas opções disponíveis.

A forma do atuador define como os dados sobre os quais o atuador opera estão organizados e é representada pelo atributo protegido *shape*. Este atributo é do tipo *int[]* e deve ser instanciado junto com o tipo do atuador no construtor ou no método *OnSetup* classe filha. Por exemplo, um atuador que produz movimentos de corpos rígidos pode receber dados na forma (4, ) que representa um *array* de quatro elementos, como o array , onde representa a intensidade do movimento para frente/para trás, representa o ângulo de rotação do corpo rígido em torno do próprio eixo, representa um salto vertical e , um salto para frente. Essas informações podem ser usadas por um controlador para produzir os dados de entrada do sensor adequadamente.

A AI4UUE possui um atuador *builtin* chamado de *RBMoveActuator.* Este atuador é capaz de controlador objetos do tipo *RigidBody*.

### A classe *MoveActuator*

Um objeto do tipo *MoveActuator* controla agentes com corpo rígido (contendo o componente *Rigidbody*). Portanto, para adicionar um *MoveActuator* ao agente, abaixo do objeto *NPC* (veja na Figura 6 o objeto selecionado e nomeado de *MoveActuator*), cria-se um objeto vazio e o associa ao *script* da classe *MoveActuator*.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamenteInterface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Figura . Exemplo em que se adicionou um objeto do tipo MoveActuator ao agente.

Um objeto do tipo *MoveActuator* possui seis campos:

|  |  |
| --- | --- |
| Campo | Descrição |
| Action Name | Deve ser um nome único para cada atuador, pois é por meio deste nome que o atuador será reconhecido pelo agente. |
| Agent | Uma referência ao agente que usa este atuador. |
| Move Amount (ma) | A quantidade de movimento para frente. |
| Turn Amount (ta) | A intensidade do giro. Pode ser um valor que varia continuamente entre -1 (giro à esquerda) e 1 (giro à direita). |
| Jumper Power (jp) | A intensidade máxima do salto (apenas vertical). |
| Jumper Forward Power (jpf) | A intensidade do salto para frente. |

Um agente com um atuador deste tipo pode receber ações da forma *[f, t, j, jf],* onde *f* > 0 representa um movimento para frente (direção positiva do eixo *z* do objeto) e *f < 0* representa o movimento oposto; *t* > 0 representa giro para a direita e *t < 0* representa giro para a esquerda; *j* representa a intensidade do salto e deve ser maior ou igual a zero; e *jf* representa um salto para frente, também deve ser maior ou igual a zero. A intensidade final dos movimentos depende dos valores dos campos do objeto *MoveActuator* e será igual a *[f \* ma, t \* ta, j \* jp), jf \* jpf]*, em que \* representa a multiplicação escalar.

Finalmente, é recomendado que o nome da ação (*Action Name*) seja configurado como “move”, pois este é o nome esperado pelos exemplos de controladores disponíveis na AI4U. Exibe-se na Figura 6 um exemplo completo de configuração deste tipo.

## A Classe Sensor

Um agente deve ter um modo de perceber o mundo. Para isso, provemos a classe *Sensor*, que permite o agente perceber as propriedades do ambiente no qual atua. A classe *Sensor* provê atributos gerais presentes em todos os sensores. Estes atributos podem ser definidos no próprio editor da *Unity*, mas também podem ser definidos em código, por meio de herança. Na verdade, a classe *Sensor* não representa um sensor específico, mas sim aglomera as propriedades gerais de sensores e os métodos que retornam todos os tipos de dados suportados pela AI4U. Os atributos gerais são cinco:

|  |  |
| --- | --- |
| **Atributo** | **Descrição** |
| *perceptionKey* | Identificador único do sensor. Esta informação será usada pelo agente para determinar determinado tipo de informação esperada pelo agente. O valor da propriedade *perceptionKey* geralmente é definido pelo usuário na interface da Unity, como mostrado na Figura 7. |
| *stackedObservations* | A quantidade de informação que o agente recebe no ciclo de decisão *t*. Essa quantidade deve ser maior ou igual a 1. Se for 1, o agente recebe no ciclo de decisão *t* apenas a informação capturada no início do ciclo. Se for *k > 1*, no ciclo de decisão *t*, o agente recebe também das informações capturadas nos ciclos de decisão *t, t-1, t-2, ..., t-k+1*. |
| *isActive* | Ativa ou desativa o sensor. Se o sensor estiver desativado antes do início do jogo, o agente ignora este sensor durante a execução do jogo. Isso pode exigir alterações no *script* de controle remoto do agente, pois do ponto de vista é externo, este sensor não fica mais visível. |
| *normalized* | Indica ao programador do sensor se os dados devem ser normalizados ou não. |
| *resetable* | Indica se o método *OnReset* do sensor deve ser executado quando o agente for reinicializado. |

Além dos atributos públicos e externos (que podem ser modificados na interface do editor da Unity), um sensor tem outros quatro atributos que devem ser definidos no construtor ou no método *OnSetup* do sensor:

1. *Type*: o tipo do sensor, que é do tipo de enumeração *SensorType*. *SensorType* pode ser:
   1. *SensorType.sint,*
   2. *SensorType.sfloat,*
   3. *SensorType.sbool,*
   4. *SensorType.sstring,*
   5. *SensorType.sfloatarray,*
   6. *SensorType.sbytearray,*
   7. *Sensortype.sintarray.*
2. *IsState*: indica se a informação do sensor deve ser incorporada ao estado do ambiente ou indica apenas uma configuração inicial do ambiente (algo que não muda ao longo do tempo).
3. *Shape*: as dimensões e seus respectivos tamanhos. Se o sensor produz apenas um valor escalar, a forma é vazia, representada por um instância vazia de um arranjo inteiro: *new int[0]*.
4. *agente*: uma referência para o agente dono do sensor. Apenas um agente pode receber dados de um sensor por vez.

Além destas propriedades, ao criar um sensor, deve-se implementar o método que retorna os dados correspondentes ao tipo do sensor. Por exemplo, se o sensor é do tipo *SentorType.sfloatarray*, deve-se implementar o método *GetFloatArrayValue*() que retorna um arranjo de números reais. Os métodos possíveis para cada tipo de dados são:

|  |  |
| --- | --- |
| **Tipo** | **Método** |
| *SensorType.sfloat* | GetFloatValue() |
| *SensorType.sstring* | GetStringValue() |
| *SensorType.sbool* | GetBoolValue() |
| *SensorType.sbytearray* | GetByteArrayValue() |
| *SensorType.sint* | GetIntValue() |
| *SensorType.sintarray* | GetIntArrayValue() |
| *SensorType.sfloatarray* | GetFloatArrayValue() |

A AI4UUE provê alguns sensores embutidos: *ActionSensor, DoneSensor, FloatArrayCompositeSensor, IDSensor, OrientationSensor, PositionSensor, RayCastingSensor, RewardSensor* e *StepSensor.*

**ActionSensor:** este sensor retorna a(s) última(s) ação(ões) realizada(s) pelo agente. Em alguns problemas de aprendizagem por reforço, o histórico de ações facilita a aprendizagem do problema. Este método é do tipo *SensorType.sfloatarray* e a quantidade de ações pode ser definida por meio da propriedade *actionSize* (*Action Size* no editor da Unity).

**DoneSensor:** este sensor retorna um booleano indicando se o episódio terminou ou não. É do tipo *SensorType.sbool* e não suporta empilhamento de observações, pois serve apenas para indicar aos controladores que um episódio terminou ou não. O valor padrão da propriedade *perceptrionKey* deste sensor é *done*. Todo agente do tipo *BasicAgent* possui este sensor.

**FloatArrayCompositeSensor:** este objeto se comporta como um sensor, mas é um agregador de sensores de diversos tipos. Este sensor serve para criar uma composição de sensores. Para isso, você deve adicionar ao nó do tipo *FloatArrayCompositeSensor* um ou mais nós filhos de um tipo que herda de *Sensor*. A informação destes sensores é convertida em números reais e colocadas em um arranjo de números reais contendo os valores de todos os sensores agregados. Portanto, este objeto deve ter um ou mais filhos do tipo *Sensor*. Os filhos podem ser de qualquer tipo numérico. O tipo deste sensor é *SensorType.sfloatarray*. Os *scripts* controladores em Python disponível no diretório de exemplos pressupõem que a propriedade *perceptionKey* deste sensortem o valor *array*.

**IDSensor:** retorna o identificador do agente no ambiente. Este sensor retorna um código de identificação único do agente no ambiente e está disponível para todos os agentes do tipo *BasicAgent*.O tipo deste sensor é *SensorType.sstring* e o valor padrão da propriedade *perceptionKey* é *id*. Este sensor não pode empilhar informações ao longo de vários ciclos de decisão.

**OrientationSensor**: retorna dois números reais que indicam a orientação relativa e a distância do agente para um alvo definido pela propriedade *target*. A orientação é calculada como o coseno do ângulo entre os vetores **u** e **v**, em que **u** representa a direção entre o agente e o alvo e **v** representa a direção do agente (eixo que indica a direção do eixo de visão do agente). A distância é a distância euclidiana entre o agente e o alvo. Este sensor se mostrou eficiente para a geração de comportamentos de navegação espacial. O tipo do sensor é *SensorType.sfloatarray*, que retorna um arranjo , onde é a orientação e é a distância do agente em relação ao alvo.

**PositionSensor**: retorna a posição do agente em coordenadas locais ou em coordenadas globais. O sensor é do tipo *SensorType.sfloatarray* e retorna um arranjo de números reais na forma [x, y, z].

**RayCastingSensor**: retorna uma matriz de dimensão *mxn* contendo códigos de objetos detectados no campo de visão do agente. Raios são lançados em perspectiva de uma posição definida na propriedade *eye* do sensor. Para cada objeto intersectado pelo raio, ou o código do objeto ou a distância do agente para o objeto é registrado na matriz na posição correspondente à intersecção do raio no plano entre o observador e o objeto na cena. Este sensor é do tipo *SensorType.sfloatarray* e produz um arranjo com *m x n* elementos. No controlador, geralmente esse arranjo é colocado novamente no formato matricial.

**RewardSensor**: este sensor retorna a última recompensa produzida pelo usuário. O tipo deste sensor é *SensorType.sfloat* e não suporta a propriedade *stackedObservations*.

**StepSensor:** retorna o ciclo de decisão atual do agente. O tipo deste sensor é *SensorType.sint*. Este sensor não suporta a propriedade *stackedObservations.*

Interface gráfica do usuário

Descrição gerada automaticamente

Figura . Exemplo de configuração de um sensor do tipo FloatArrayCompositeSensor.

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

A forma como o agente funciona quanto à origem do controle depende de um objeto do tipo *Brain*, que pode ser *RemoteBrain* ou *LocalBrain.* Um agente deve ter apenas um nó do tipo *Brain* habilitado na cena associado a ele. No caso de vários agentes, pode-se criar um *Brain* habilitado para cada agente.

### Um *Brain* do Tipo *RemoteBrain*

Um *Brain* do tipo *RemoteBrain* permite o agente ter um controlador remoto, implementado em qualquer linguagem que implemente o protocolo privado da AI4U. Por enquanto, mantemos este protocolo implementado apenas na linguagem Python e testado especificamente na versão *Python 3.7*.

### Um *Brain* do Tipo *LocalBrain*

Já um *Brain* do tipo *LocalBrain*, permite o agente ser controlado por *scripts* implementados na própria *engine* de jogos, usando C# ou outra linguagem suportada pela *engine* *de jogos.* Um objeto *LocalBrain* depende de um objeto de um tipo que herda de *ai4u.Controller*. Por enquanto, o único controlador concreto embutido na AI4UUE é o *ai4u.WASDMoveController*, que permite controlar o agente usando as teclas *WASD* do teclado de um PC ou de um notebook.

Contudo, o usuário poderia implementar seu próprio controlador para, por exemplo, executar uma rede neural implementada diretamente em C#. Isso permitiria treinar um controlador usando Python e usar a rede neural depois de treinada sem a necessidade de manter a conexão com a linguagem Python. Essa é uma funcionalidade futura que pretendemos adicionar à AI4UUE.

**Criando seu Próprio *Controller***

Um controlador precisa sobrescrever basicamente dois métodos: *GetAction* e *NewStateEvent*. O método *GetAction* é executado quando o *ControlRequestor* envia uma solicitação de controle, então *GetAction* deve retornar uma ação codificada em uma *string*. O programador pode utilizar métodos acessórios, como as várias versões do método *ai4u.Utils.ParseAction*, que transforma um comando em sua codificação como *string*.

No método *NewStateEvent*, o programador pode utilizar diversos métodos acessórios (como *GetStateSize, GetStateName* e *GetStateAsFloat*) para verificar qual o estado atual do agente.

Enquanto não se produz uma documentação mais ampla, recomendamos basear o código do controlador no código da classe *ai4u.WASDMoveController*.

# Funções de Recompensa

Um elemento centralizador de algoritmos de aprendizado de máquina é o tipo de *feedback* que se usa para treinamento dos controladores. AI4U provê os elementos básicos para o aprendizado por reforço.

A classe *BasicAgent* provê métodos que permitem ao programador adicionar recompensas ao agente em qualquer ponto de um ciclo de decisão do agente. Contudo, para garantir a consistência do laço de decisão e a propriedade *markoviana* de um processo de decisão, deve-se adicionar a recompensa correspondente à ação do agente em determinado estado no final do ciclo de decisão *t*. É importante cada recompensa ficar restrita ao ciclo de decisão na qual foi adicionada. Há duas formas de se fazer isso com AI4U:

* Implementar uma classe que herda de *ai4u.RewardFunc*; criar um nó do tipo dessa classe e filho do nó *BasicAgent*; neste caso, o deve-se sobrescrever o método *OnUpdate*, que é onde se deve adicionar recompensa do ciclo de decisão do agente; ou
* Adicionar um controlador do evento ao evento *endOfStepEvent* de *BasicAgent* e adicionar a recompensa neste controlador.

O interessante da segunda possibilidade é que é possível adicionar recompensa de qualquer componente do agente, seja dentro de um sensor ou de um atuador criados pelo usuário. Na Figura 8, observa-se uma função que adiciona uma recompensa quando o agente toca um cubo. Observe que a variável *acmReward* é zerada (linha 32 do respectivo código-fonte) depois que a recompensa é adiciona ao agente. Isso evita a propagação da recompensa nos ciclos de decisão subsequentes.

Texto

Descrição gerada automaticamente

Figura . Exemplo de função de recompensa.

# O uso de um controlador local do Agente

Uma vez que o agente e todos os seus componentes foram configurados, pode-se configurar o *ControlRequestor* para manipulação do corpo do agente por meio de um controlador local. Para isso, deve-se criar um nó do tipo *Node*, associá-lo ao *script LocalBrain*, definir o agente deste objeto *Brain* por meio da propriedade *agentPath* (*Agent Path*). Deve-se garantir que haja apenas este *LocalBrain* com este agente e habilitado.

O nó do tipo *LocalBrain* deve apontar para um controlador. Um controlador define a forma como o agente se comportará. Para isso, deve-se criar um nó do tipo *Node* e associá-lo ao *script* que implementa o controlador. Vamos usar como exemplo o controlador embutido *WASDMoveController*, que permite controlador o agente por meio das teclas *WASD* do telcado alfa-numérico. Na Figura 9, mostra-se como este controlador foi configurado no projeto *AI4UGTesting*.



Figura . Exemplo de configuração de um controlador WASDMoveController. Speed é o multiplicador de velocidade do agente quando a tecla de movimento para frente/traz é pressionada. Actuator Name é o nome do atuador que este controlador deve manipular.

Depois de o controlador ser configurado, deve-se configurar a propriedade *Controller* de *LocalBrain* para apontar para este controlador, como mostrado na Figura 10.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Figura . Configuração de um nó LocalBrain. O campo Is Enabled determina se este Brain está habilitado ou não. O campo Agent indica qual agente será controlado por este Brain. O campo Skip Frame determina a quantidade de quadros físicos que separam uma tomada de decisão de outra. O campo Repeat Action determina se uma ação deve ser repetida em cada quadro entre tomadas de decisão. Já o campo Controller aponta para o objeto que implementa a tomada de decisão do agente.

# O uso de um controlador remoto de Agente.

Pode-se definir um controlador remoto de agente por meio de um objeto do tipo *RemoteBrain*. Deve-se garantir que este nó seja o único do tipo *Brain* habilitado e associado ao agente alvo. Para isso, primeiro, deve-se criar e configurar um nó do tipo *RemoteBrain*. Na Figura 12, observa-se a configuração de um nó do tipo *RemoteBrain*. Cria-se um nó desse tipo do mesmo modo que se cria um nó do tipo *LocalBrain*, exceto que o *script* usado é *RemoteBrain*.

Tela de um aparelho eletrônico

Descrição gerada automaticamente

Figura . Configuração de um componente do tipo RemoteBrain. Além das mesmas propriedades de um LocalBrain, possui os campos Host, que contém o IP ou nome da máquina que executa o script de controle do agente; Port, que indica qual porta será usada para comunicação entre o ambiente da Unity e o script remoto de controle; e Receive Timeout, que indica o tempo que deve passar antes da conexão falhar com time out. Mantenha o valor de Receive Timeout igual ou acima de 200.

Um objeto do tipo *RemoteBrain* se conecta com um *script* na rede e recebe comandos de um controlador externo, passando estes comandos ao agente por meio do *ControlRequestor*. As configurações de conexão remota são feitas no próprio *RemoteBrain*, como mostrado na Figura 11. A Linguagem suportada pelo AI4U para controle externo do agente é Python. Esta linguagem pode ser usada para treinar uma rede neural que controla o agente. A linguagem Python foi usada devido à sua facilidade e à disponibilidade de *frameworks* e ferramentas de aprendizado de máquina que a suportam.

# Treinamento e manipulação externa do Agente

O treinamento e manipulação externa (por meio de ferramentas fora da *engine de jogos*) pode ser feita por meio de *scripts* em Python. A forma de implementar este controle é a mesma tanto para a *Unity* quanto para a *Godot*. Por isso, toda a documentação referente a controle externo está disponível em arquivos com extensão \*.md disponíveis no diretório *doc* da AI4U.

Contudo, é importante considerar as configurações adequadas da Unity para que o treinamento do agente seja bem-sucedido. Durante o treinamento, configure a propriedade *Default Time Scale* para 10, dessa forma o treinamento ocorrerá bem mais rápido. Esse valor parece ser adequado para computador com GPU dedicada *Nvidia 1650* de 4G de VRAM, processador Intel Core i5 da geração 11 e memória RAM de 8GB.

Depois de configurar adequadamente o ambiente e o agente, garanta que um e apenas um componente do tipo *RemoteBrain* esteja habilitado apontando para o agente a ser testado. Então, entre no diretório *examples/scene\_samplescene* (localizado no repositório da AI4U) e execute o comando:

*>> python appgym\_sb3train.py*

Este comando aciona um *script* que usa o algoritmo SAC disponível no *framework* *stable-baselines3* para treinar um modelo controlador do agente. Este modelo consegue alcançar o cubo sem cair da plataforma. Para testar o controlador, execute o comando:

*>> python appgym\_sb3test.py*

Ao executar este comando, você escolherá se quer executar o modelo treinado para controlar um agente da Unity, da Godot ou o último modelo treinado (como mostrado na Figura 13).

Texto

Descrição gerada automaticamente

Figura . Inicializando um script que carrega uma rede neural para controlar um agente na Unity/Godot

# Criando Novos Componentes (Sensores, Atuadores, Agentes e Controladores)

É fácil criar sensores, atuadores e controladores usando AI4UUE. Vou mostrar como se faz isso em uma implementação própria do jogo Pong. Vou usar o máximo possível a implementação de física da Unity para jogos 2D.

## O Jogo Pong

Vou criar uma mecânica muito simples do Pong. O visual do jogo é mostrado na Figura Figura 13. O objetivo é criar um jogo que possa ser jogado por uma Inteligência Artificial. Uma partida do jogo começa com a bola no centro e lançada aleatoriamente para a direita ou para a esquerda. O jogador da esquerda é manipulado por um controlador pré-programado. O programador da direita é controlado por um *script* *remoto* (em Python) e será usado para treinar o agente da direita. Um episódio termina quando a bola consegue passar por um dos jogadores, sendo que o jogador do lado aposto vence e ganha um ponto, enquanto o jogador que sofreu a derrota perde um ponto. Este ponto vai ser usado como recompensa pelo algoritmo de treinamento.

Interface gráfica do usuário, Retângulo

Descrição gerada automaticamente com confiança média

Figura . As barras amarelas representam os jogadores. O jogador da esquerda é um NPC pré-programado. O jogador da direita é controlado por uma rende neural. O círculo branco é a bola. As pontuações são mostradas nos cantos superiores.

**O atuador**

Criamos um atuador para o agente chamado de *PongActuator*. Esta é uma classe que herda da classe *ai4u.Actuator*, conforme mostrado na Figura Figura 14.

**Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente**

Figura . Esta classe implementa um novo atuador, específico para o jogo Pong. Geralmente, três métodos devem ser implementados: OnSetup, Act e OnReset.

No método *Setup*, inicializa-se as configurações iniciais do atuador, por exemplo, a quantidade de ações pode ser definida por meio do atributo herdado *shape* (equivalente ao *shape* de *arrays* *numpy*). Um atuador possui um *shape*, um domínio definido pelas variáveis *rangeMin* e *rangeMax* e um tipo (*contínuo ou discreto*).

O *shape* determina a quantidade de ações do agente e deve ser um *array* de tamanho um com o único elemento indicando a quantidade ações do atuador.

O *domínio é* atribuído para cada ação do atuador. O *rangeMin* é um array contendo o valor mínimo de cada ação e *rangeMax* é array contendo o valor máximo de cada ação. O domínio é realmente apenas necessário quando o atuador é do tipo *contínuo*. Contudo, *rangeMin* e *rangeMax* devem ser inicializados como um *array vazio* se o atuador é discreto.

O *tipo* do atuador é *discreto (inteiro)* ou *contínuo* (ponto-flutuante). A propriedade herdade *isContinuous* determina o tipo do atuador. Por padrão, esta propriedade tem o valor *false*, ou seja, um atuador é discreto.

Na Figura 15, mostra-se a implementação do método *OnSetup* do atuador do jogo Pong. No jogo *Pong*, o jogador tem três opções (ações): zero (0) significa não fazer nada, um (1) significa subir e dois (2) significa descer.

Texto

Descrição gerada automaticamente

Figura . Implementação do método OnSetup do atuador do jogo Pong.

A aplicação efetiva da lógica de funcionamento do atuador é feita no método *Act* (Figura 16), onde usamos os elementos de física 2D da Unity para implementar o movimento da barra controlada pelo agente.

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Figura . Implementação da Lógica de atuação do agente no jogo Pong.

Apesar do atuador do tipo discreto ser inteiro, recomenda-se utilizar vetores numéricos (ponto-flutuante ou inteiro) para a obtenção dos parâmetros do atuador por meio do agente. Estes parâmetros podem ser enviados para diferentes tipos de controladores, portanto, por conveniência, vetores de numéricos são usados como moeda única de troca. Ou seja, em vez de usar a função *agente.GetActionArgAsInt*, usamos a função *agente.GetActionArgAsFloatArray* (Figura 16, linha 26) ou a função *agente.GetActionArgAsIntArray* mesmo que a ação seja discreta ou contínua com um único parâmetro.

O método *OnReset* é importante para restabelecer as configurações iniciais do jogo quando reinícios foram acionados. A implementação deste método no atuador *PongActuador* é mostrada na Figura 17.

Uma imagem contendo Texto

Descrição gerada automaticamente

Figura . Implementação do método OnReset do objeto PongActuator.

**O Sensor**

**Para o jogo Pong, usamos o sensor *ScreenSensor* (**Figura 18**) disponível na AI4UUE. Este sensor exige que se especifique uma câmera extra onde será renderizada a cena para o agente. Observe, esta câmera extra não precisa e nem deve ter alta resolução, pois a imagem gerada somente será “vista” pelo agente. É recomendado manter a resolução desta imagem abaixo de 81 pixels por dimensão. A partir desta resolução, a quantidade de memória necessária em alguns tipos de algoritmos (*Deep Q-Learning*, por exemplo) será muito grande (acima de 18 GB). Deve-se usar o *ScreenSensor* somente em casos muito específicos, pois este sensor tem alta-exigência computacional.**

**Interface gráfica do usuário

Descrição gerada automaticamente**

Figura . Sensor de tela do agente Pong.

O sensor do tipo *ScreenSensor* exige a definição de uma câmera, como mostrado na Figura 18**. E esta câmera tem que apontar para uma textura alvo (campo *target texture*), como mostra a** Figura 19**.**

Interface gráfica do usuário

Descrição gerada automaticamente

Figura . Campo "Target Texture" da câmera do sensor ScreenSensor definida com uma textura de baixa resolução.

A textura da câmera do agente deve ter baixa resolução. Além disso, prefira sempre manter ativada a propriedade *Gray Scale* de *ScreenSensor* e não colocar um valor maior do que três no campo *stackedObservations* de *ScreenSensor*, a não ser, claro, que você tenha um computador muito bom (talvez o *Hall*). A Figura 20 mostra as configurações da textura usada neste caso.

Tela de computador com texto preto sobre fundo branco

Descrição gerada automaticamente

Figura . Configurações da textura do sensor do tipo ScreenSensor usado no jogo Pong.

# 

**Agentes e Controladores**

**Definimos dois agentes para o jogo Pong (**Figura 21**). Os dois agentes possuem as mesmas configurações gerais (um atuador e um sensor), mas enquanto o agente *player1* pode ser controlado por uma rede neural, o agente *player2* foi pré-programado com um *script* C# local. Ou seja, o *Brain* de *player1* é do tipo *RemoteBrain*, enquanto que o *Brain* de *player2* é do tipo *LocalBrain* que aponta para um controlador do tipo *PongBotController*.**

**Texto

Descrição gerada automaticamente**

Figura . Agentes do jogo Phong.

O projeto completo do jogo Pong baseado em AI4UUE pode ser encontrado [AQUI](https://github.com/gilzamir18/PhongDemo.git).

# Limitações

Até o momento, a AI4UUE suporta agentes que controlam corpos físicos do tipo *RigidBody* em ambientes tridimensionais e possui um conjunto limitado de sensores. Os *scripts* de treinamento e teste apenas foram testados no Windows 10 e no Windows 11.

# Créditos

Gilzamir Gomes ([gilzamir@outlook.com](mailto:gilzamir@outlook.com)): *design* e desenvolvimento.

Eduardo Nogueira (não informado): dicas de *game design*.

1. *Donut é uma cena que nos lembra um donut, exceto que é quadrada e não redonda que nem um donut😊.* [↑](#footnote-ref-1)