



**REALIDADE VIRTUAL E INTELIGÊNCIA  
ARTIFICIAL**

**INTELIGÊNCIA ARTIFICIAL, MÁQUINA DE  
ESTADO**

**Paulo Salvatore**

## RESUMO

A Máquina de Estado Finita, ou FSM é uma das técnicas de IA mais utilizadas, principalmente na construção de personagens de jogos que terão um comportamento automático baseado em tomadas de decisão que foram previamente planejadas e programadas. Nesse capítulo aprenderemos como planejar e desenvolver uma FSM, criando um cenário tridimensional, adicionando personagens e obstáculos que navegam de maneira inteligente e interagem entre si.

**Palavras-chave:** Inteligência Artificial; Máquina de Estado Finita; Navigation Mesh; NavMesh.

## LISTA DE FIGURAS

Figura 1.1 – Criação do Projeto.....	5
Figura 1.2 – Standard Assets.....	6
Figura 1.3 – Criação do chão. ....	7
Figura 1.4 – Pilar adicionado.....	7
Figura 1.5 – Alterar visualização de pivot de objetos na Unity. ....	8
Figura 1.6 – Posição final do pilar. ....	9
Figura 1.7 – Posição do pilar menor.....	9
Figura 1.8 – Inserindo o jogador.....	10
Figura 1.9 – Testando a movimentação do Jogador. ....	11
Figura 1.10 – Definindo target da câmera. ....	12
Figura 1.11 – Definindo posição e rotação da câmera. ....	12
Figura 1.12 – Testando a nova câmera.....	13
Figura 1.13 – Inserindo uma IA. ....	14
Figura 1.14 – Definindo target do AI Character Control. ....	14
Figura 1.15 – Erros de NavMesh ausente. ....	15
Figura 1.16 – Navigation Window.....	16
Figura 1.17 – Navigation Static. ....	17
Figura 1.18 – NavMesh Display .....	17
Figura 1.19 – IA seguindo o jogador. ....	18
Figura 1.20 – NavMesh Obstacle. ....	19
Figura 1.21 – Script Easing organizado na pasta Scripts.....	19
Figura 1.22 – Movimentação do obstáculo atualizando NavMesh. ....	22
Figura 1.23 – Todo o espaço de movimentação do obstáculo demarcado pela NavMesh. ....	23
Figura 1.24 – Representação da Máquina de Estado desse projeto.....	24
Figura 1.25 – Empty Game Object: Waypoints. ....	25
Figura 1.26 – Criando objeto Waypoint. ....	26
Figura 1.27 – Definindo um ícone para o Waypoint. ....	27
Figura 1.28 – Criando o Prefab do Waypoint. ....	27
Figura 1.29 – Posicionando e renomeando os waypoints. ....	28
Figura 1.30 – Script Controlador IA criado e organizado.....	29
Figura 1.31 – Valores atribuídos automaticamente para cada item do enumerator.	31
Figura 1.32 – Waypoints atribuídos nas variáveis da IA.....	39
Figura 1.33 – Ativar modo Debug no Inspector. ....	40
Figura 1.34 – Visualização das variáveis privadas no Inspector. ....	41
Figura 1.35 – IA seguindo waypoint. ....	44
Figura 1.36 – Definir tag Player no objeto do Jogador. ....	46
Figura 1.37 – Alteração dos valores de Steering da IA. ....	49

## SUMÁRIO

7	INTELIGÊNCIA ARTIFICIAL, MÁQUINA DE ESTADO .....	5
7.1	Introdução.....	5
7.2	Criando o Projeto.....	5
7.3	Criando o Cenário .....	6
7.4	Inserindo o Jogador .....	10
7.5	Inserindo uma Inteligência Artificial .....	13
7.6	Construindo uma NavMesh .....	15
7.6.1	Obstáculos na NavMesh	18
7.7	Máquina de Estado.....	23
7.7.1	Waypoints	25
7.7.2	Esperar	28
7.7.3	Patrulhar	37
7.7.4	Perseguir	44
7.7.5	Procurar	49
8	REFERÊNCIAS .....	54

## 7 INTELIGÊNCIA ARTIFICIAL, MÁQUINA DE ESTADO

### 7.1 INTRODUÇÃO

### 7.2 CRIANDO O PROJETO

Para o desenvolvimento da aplicação, utilizarei a Unity na versão 2018.2.11f1. Crie um projeto com o nome que preferir, selecionando o Template 3D.

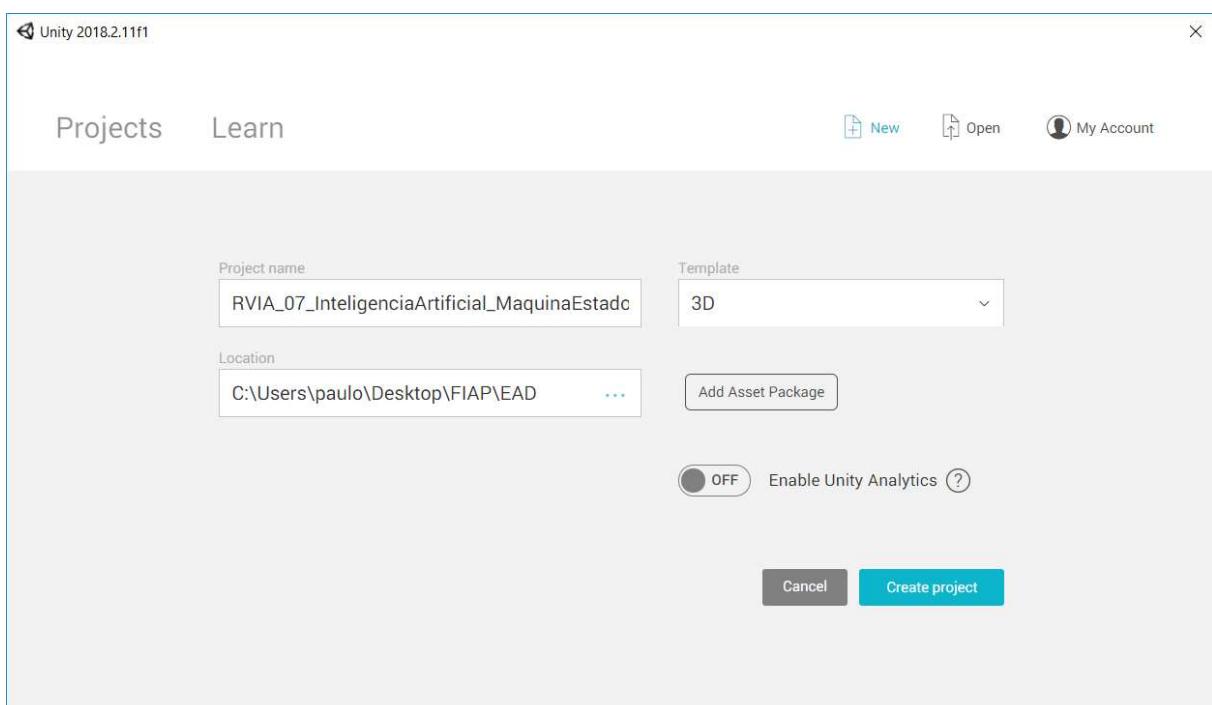
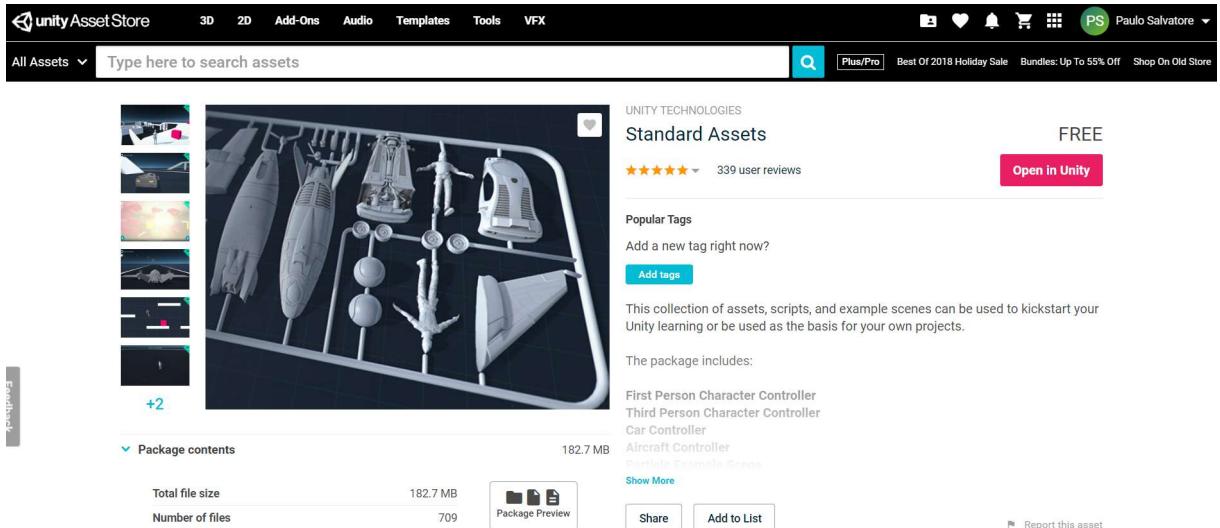


Figura **Erro! Fonte de referência não encontrada..1** – Criação do Projeto.  
Fonte: Unity (2018.2.11f1).

Com o projeto criado, adicione o asset “Standard Assets” disponibilizado gratuitamente pela Unity Technologies através do link:

<https://assetstore.unity.com/packages/essentials/asset-packs/standard-assets-32351>

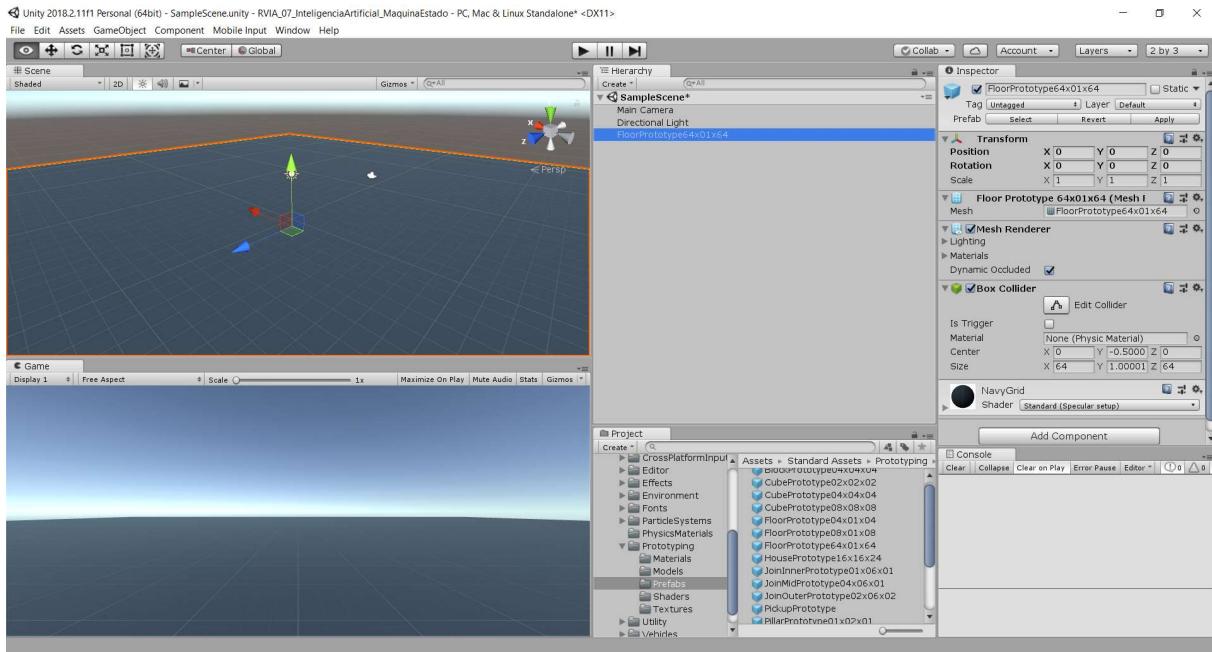
Esse processo deve demorar alguns minutos.



**Figura Erro! Fonte de referência não encontrada..2 – Standard Assets.**  
Fonte: Unity Asset Store (2018).

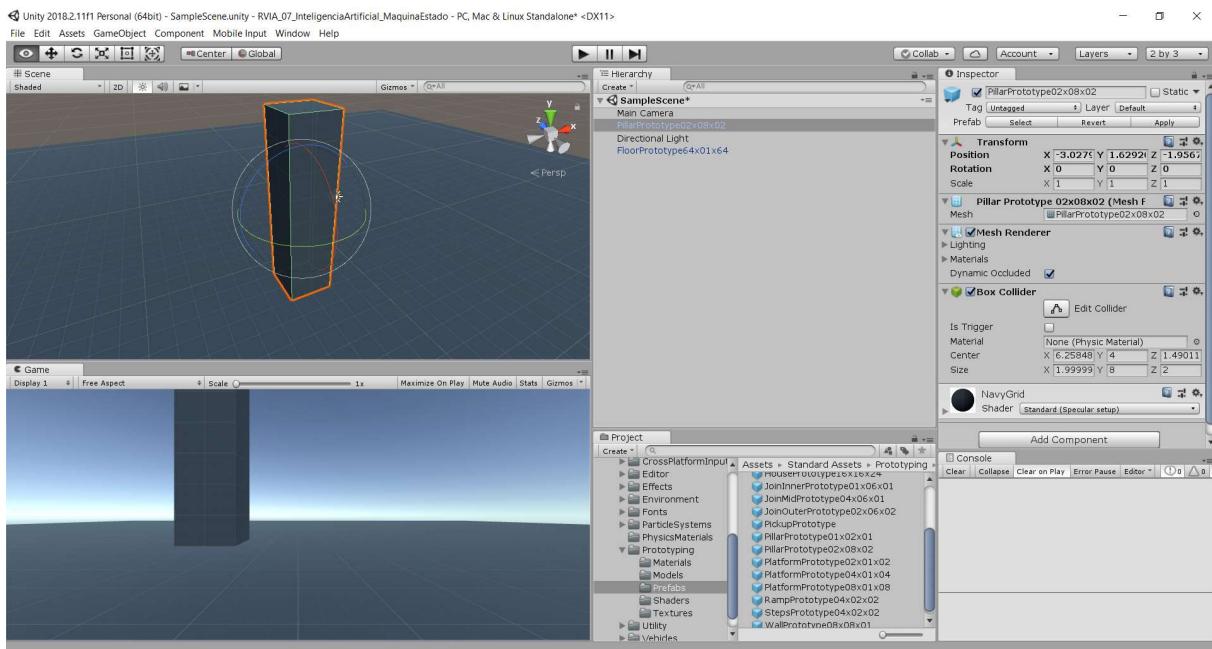
### 7.3 CRIANDO O CENÁRIO

Para iniciar, vamos construir um cenário que servirá como base para o nosso projeto e a movimentação dos elementos de inteligência artificial. Dentro da Unity, navegue até a pasta “Assets/Standard Assets/Prototyping/Prefabs”, busque pelo arquivo de prefab “FloorPrototype64x01x64” e adicione-o na cena. Certifique-se de que os valores do transform do GameObject adicionado estão zerados, como na Figura Erro! Fonte de referência não encontrada..3 – Criação do chão.



**Figura Erro! Fonte de referência não encontrada..3 – Criação do chão.**  
Fonte: Unity (2018.2.11f1).

Na mesma pasta, procure pelo prefab “PillarPrototype02x08x02” e adicione-o na cena. Selecione o novo GameObject criado e aperte o atalho E, para ativar a ferramenta de rotação da Unity.



**Figura Erro! Fonte de referência não encontrada..4 – Pilar adicionado.**  
Fonte: Unity (2018.2.11f1).

Note que no meu caso, a ferramenta de rotação está centralizada no objeto, no entanto, esse objeto possui um pivot na base. Para visualizarmos esse pivot,

procure pela opção “Center” na parte superior esquerda da Unity, conforme mostra na Figura **Erro! Fonte de referência não encontrada..5** – Alterar visualização de pivot de objetos na Unity.

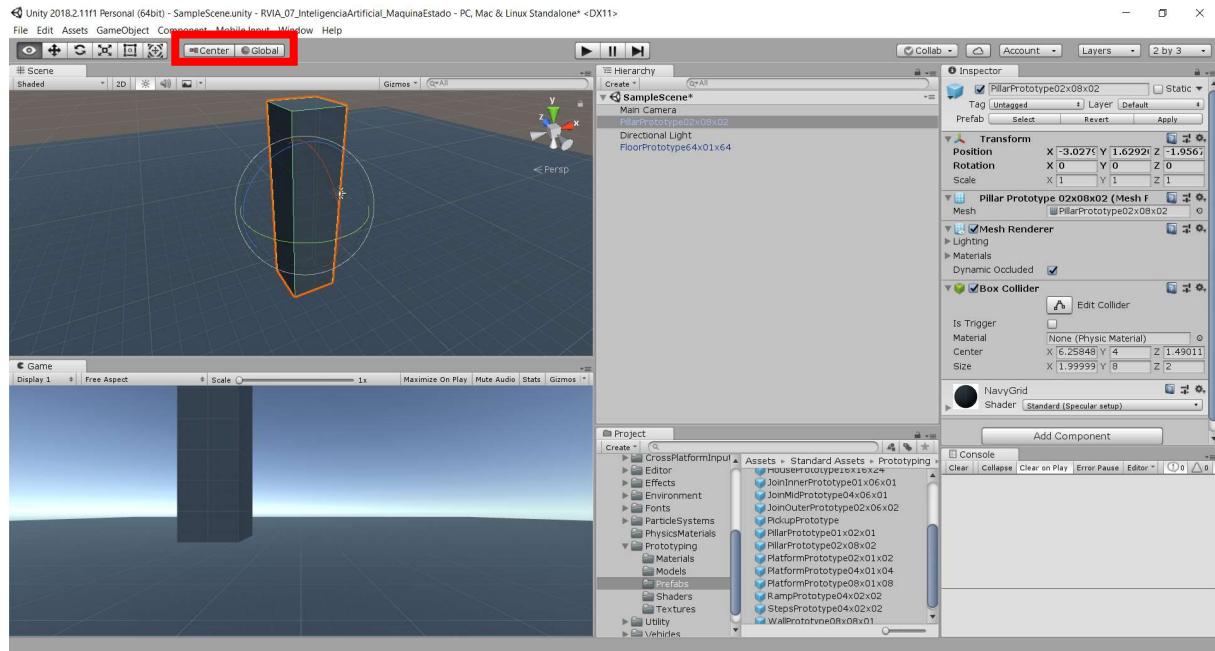


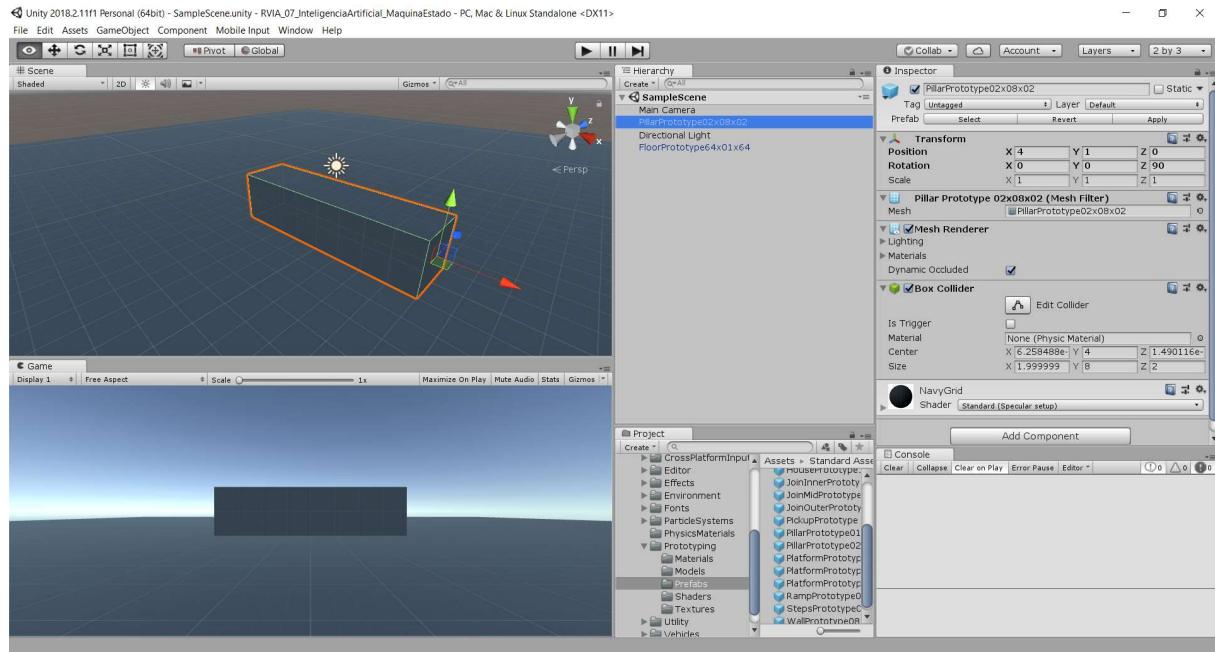
Figura **Erro! Fonte de referência não encontrada..5** – Alterar visualização de pivot de objetos na Unity.

Fonte: Unity (2018.2.11f1).

Pressionando a tecla “Ctrl”, rotacione o objeto para que ele fique alinhado com o chão. O atalho “Ctrl” fornecerá mais precisão na hora de rotacionar objetos. Faça o mesmo para posicionar o GameObject acima do chão e centralizado na cena, ativando o atalho “W” para movimentação de objetos, deixando o objeto da mesma forma que na Figura **Erro! Fonte de referência não encontrada..6** – Posição final do pilar. Se preferir, defina os valores manualmente para o objeto.

Position: x: 4, y: 1 e z: 0

Rotation: x: 0, y: 0 e z: 90

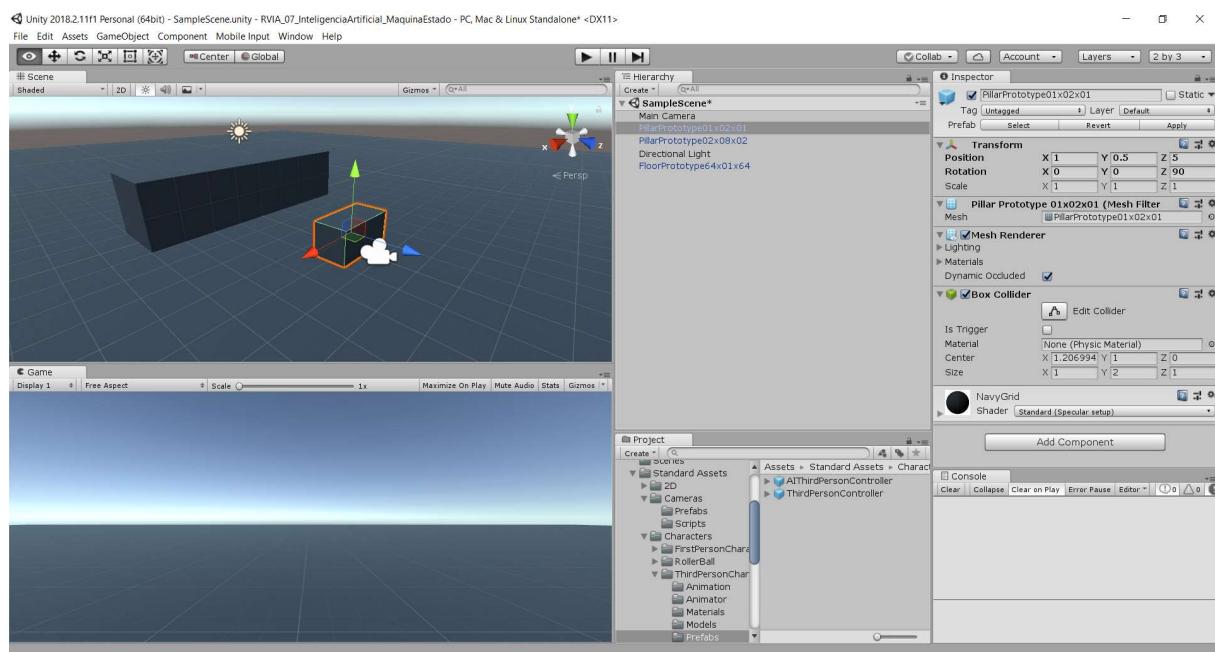


**Figura Erro! Fonte de referência não encontrada..6 – Posição final do pilar.**  
Fonte: Unity (2018.2.11f1).

Adicione também o prefab do pilar de tamanho menor, chamado “PillarPrototype01x02x01”, e posicione-o de acordo com a Figura **Erro! Fonte de referência não encontrada..7 – Posição do pilar menor**. Se preferir, altere os valores manualmente.

Position: x: 1, y: 0.5 e z: 5

Rotation: x: 0, y: 0 e z: 90



**Figura Erro! Fonte de referência não encontrada..7 – Posição do pilar menor.**

Fonte: Unity (2018.2.11f1).

## 7.4 INSERINDO O JOGADOR

Navegue até a pasta: “Assets -> Standard Assets -> Characters -> Third Person Character -> Prefabs” e adicione na cena o prefab “ThirdPersonController” de acordo com a Figura **Erro! Fonte de referência não encontrada..8 – Inserindo o jogador.** Se preferir, altere os valores manualmente.

Position: x: 0, y: 0 e z: -4

Rotation: x: 0, y: 0 e z: 0

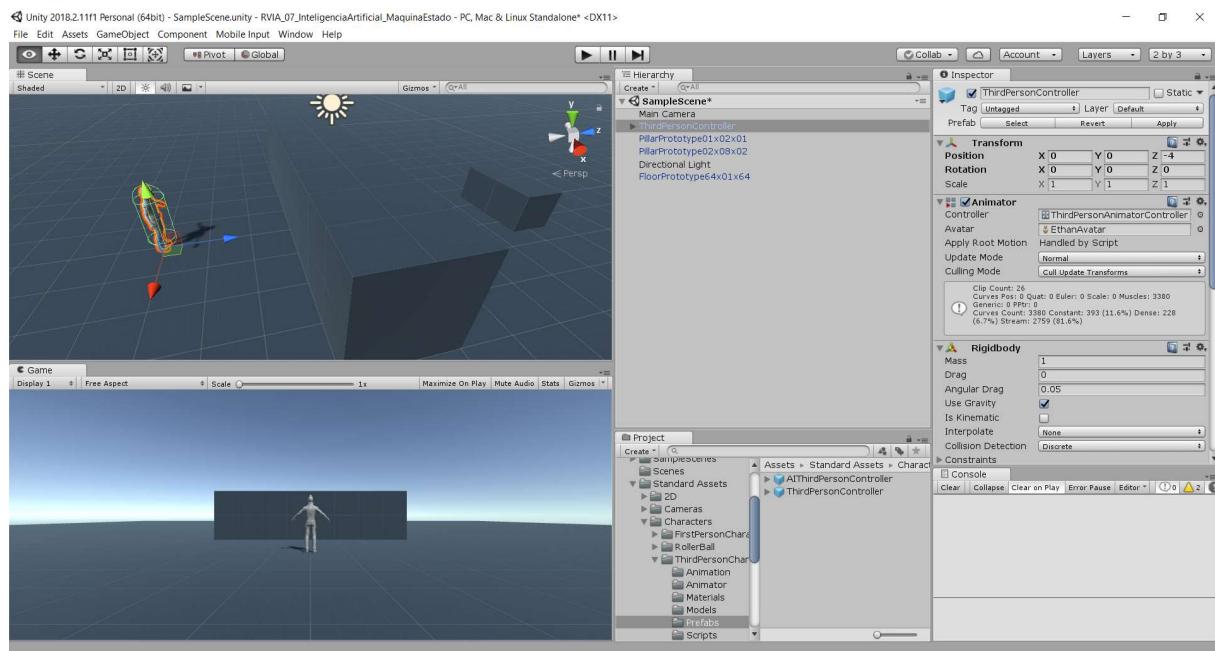
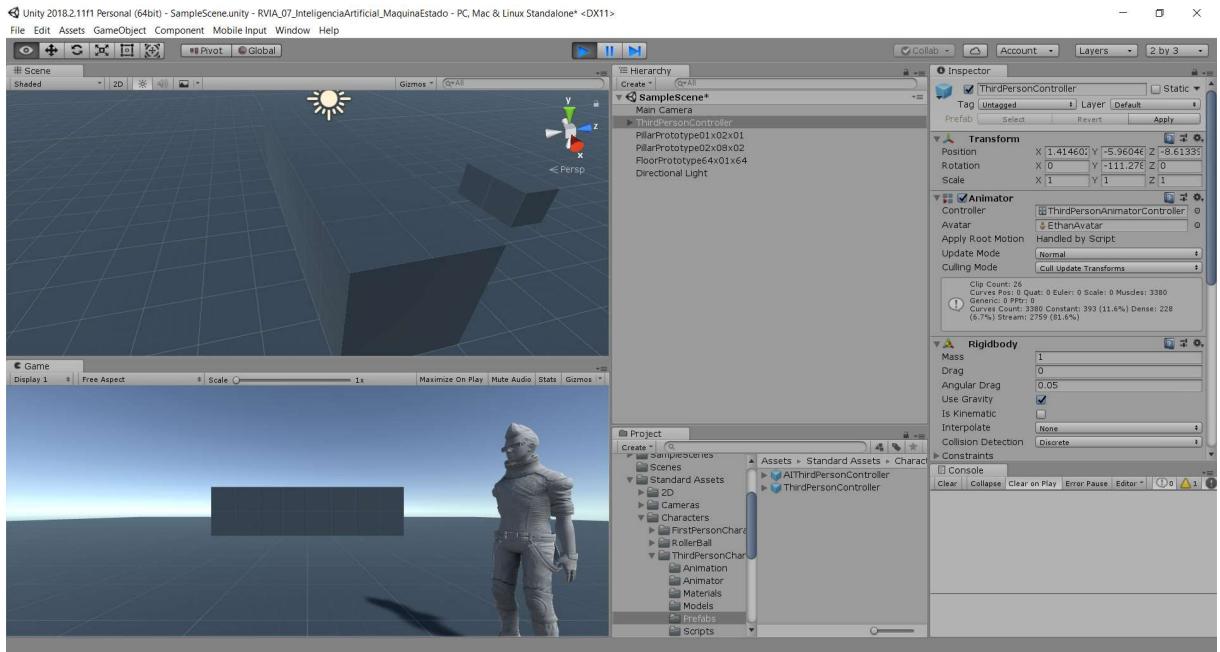


Figura **Erro! Fonte de referência não encontrada..8 – Inserindo o jogador.**

Fonte: Unity (2018.2.11f1).

Dê play no jogo pelo atalho “Ctrl + P” e movimente-se com o jogador. Note que a câmera está estática, ou seja, não está se movimentando. Dentro do pacote Standard Assets, além dos blocos de prototipação que utilizamos para o cenário e do personagem 3D que acabamos de inserir e testar, há também diversos prefabs de câmeras.

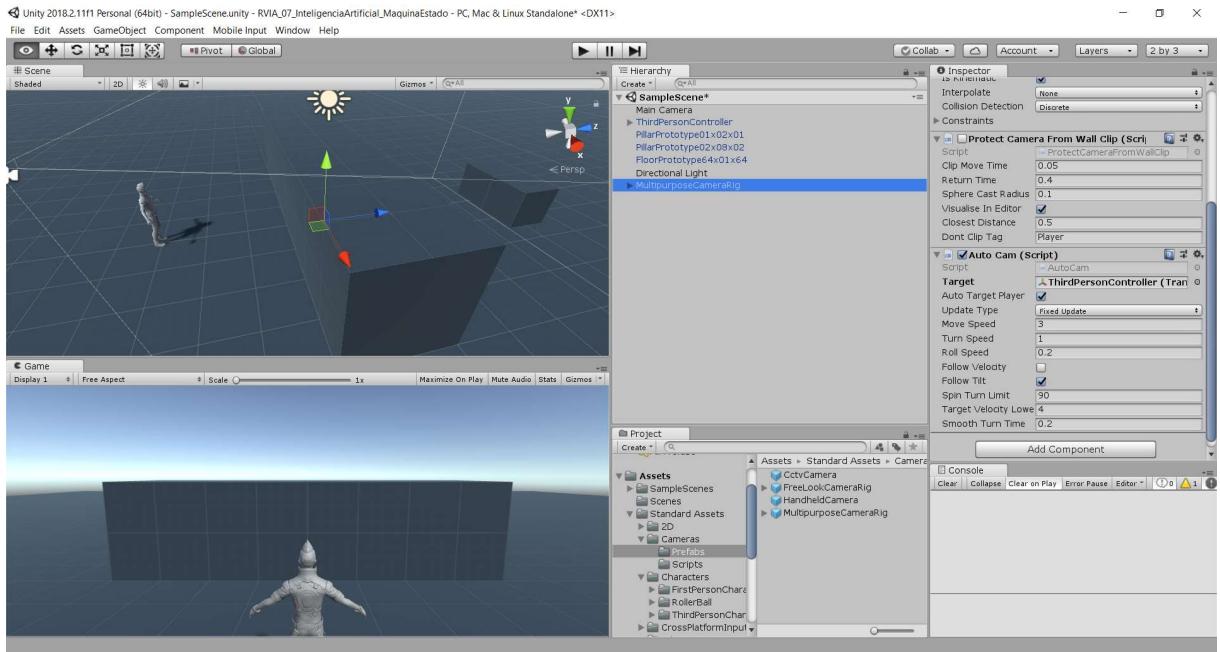


**Figura Erro! Fonte de referência não encontrada..9 – Testando a movimentação do Jogador.**  
Fonte: Unity (2018.2.11f1).

Saia do modo play, isso é bem importante para que suas modificações não sejam desfeitas. Dentro da pasta Standard Assets, procure pela pasta “Cameras/Prefabs” e adicione na cena o prefab chamado “MultipurposeCameraRig”.

No objeto que foi adicionado, busque pelo script chamado “Auto Cam” e note que há um campo chamado “Target” que está sem nenhum atributo.

Arraste o GameObject do “ThirdPersonController” na cena para esse campo, ficando como na Figura **Erro! Fonte de referência não encontrada..10 – Definindo target da câmera.**

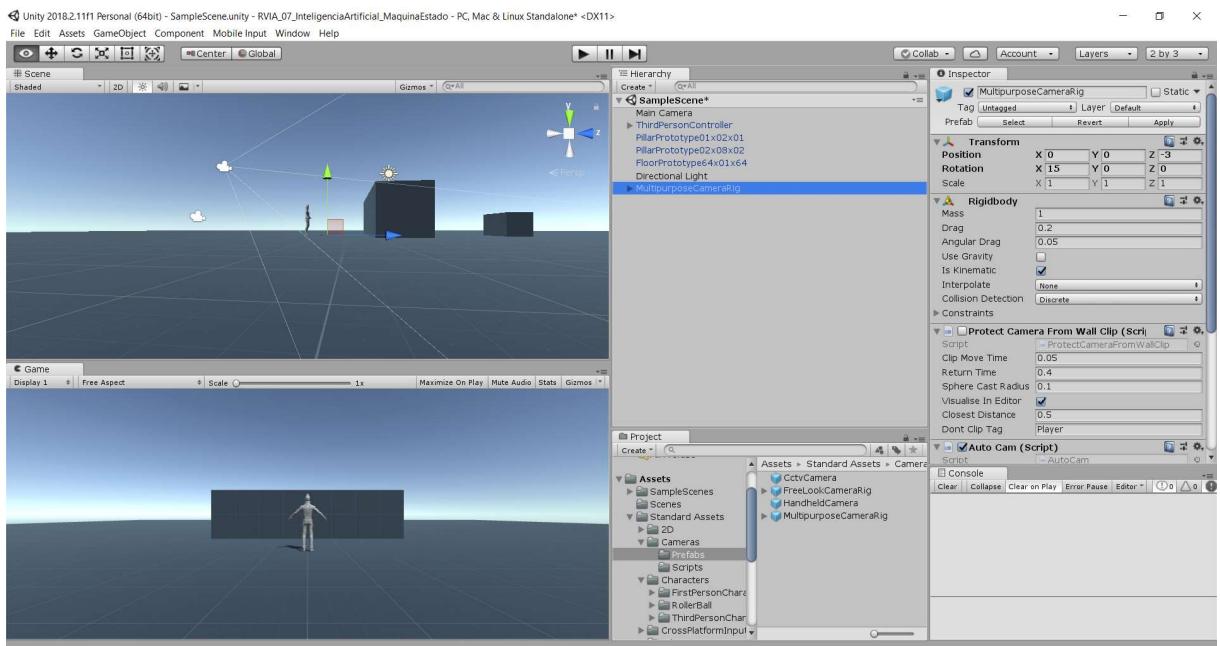


**Figura Erro! Fonte de referência não encontrada..10 – Definindo target da câmera.**  
Fonte: Unity (2018.2.11f1).

Para melhorar a visualização da câmera, altere os valores de posição e rotação de acordo com sua preferência ou de acordo com a **Figura Erro! Fonte de referência não encontrada..11 – Definindo posição e rotação da câmera**. Se preferir, altere os valores manualmente.

Position: x: 0, y: 0 e z: -3

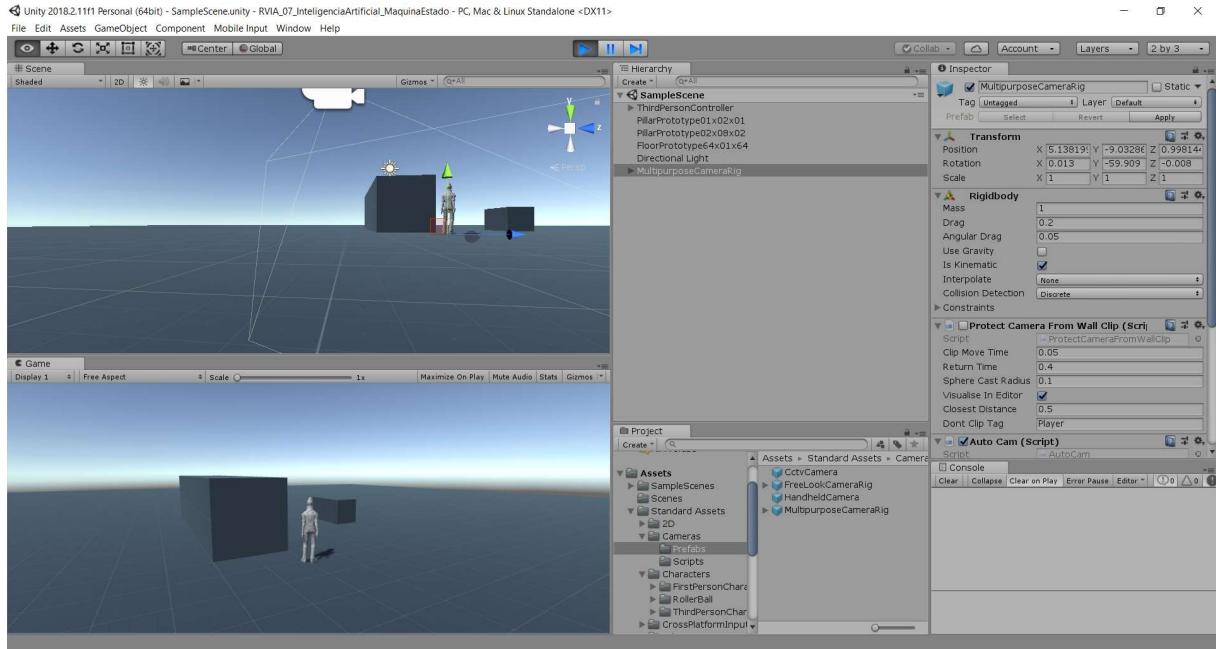
Rotation: x: 15, y: 0 e z: 0



**Figura Erro! Fonte de referência não encontrada..11 – Definindo posição e rotação da câmera.**

Fonte: Unity (2018.2.11f1).

Para que a câmera funcione corretamente, precisamos remover a câmera default criada pela Unity no início do projeto. Remova o GameObject “MainCamera”, salve a cena e dê play para testar o progresso do jogo.



**Figura Erro! Fonte de referência não encontrada..12 – Testando a nova câmera.**  
Fonte: Unity (2018.2.11f1).

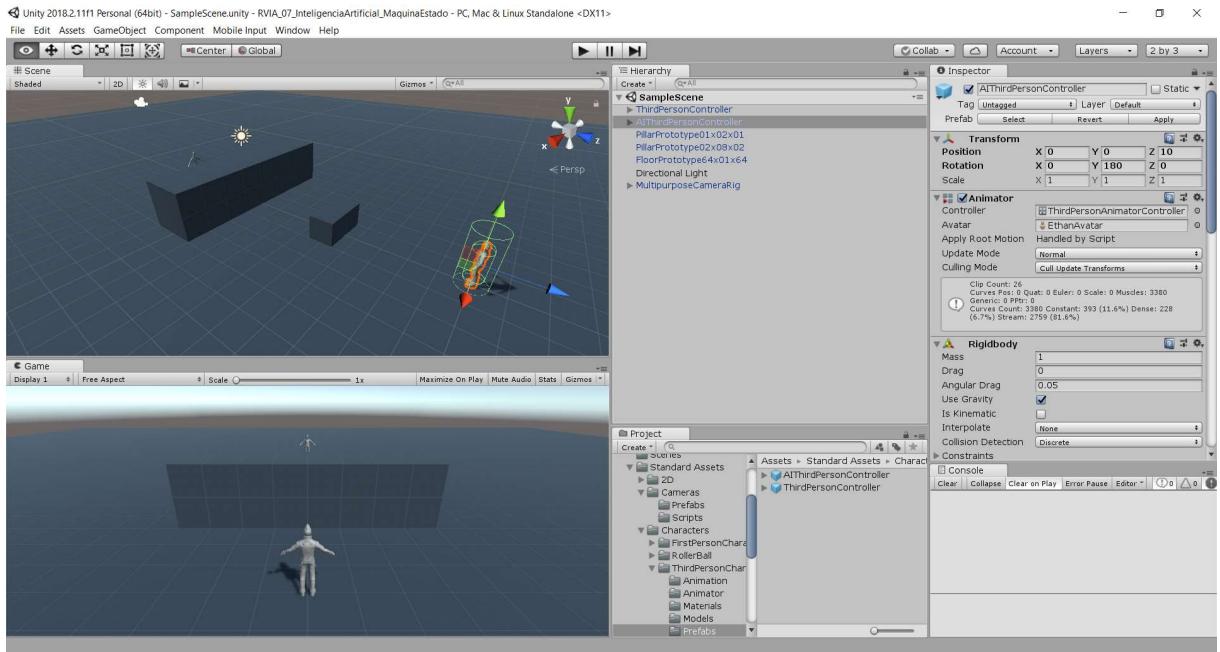
Para obter um melhor resultado da câmera, também é possível ativar o script “Protect Camera From Wall Clip”, que impedirá que a câmera fique na mesma posição de algum objeto com collider.

## 7.5 INSERINDO UMA INTELIGÊNCIA ARTIFICIAL

Na mesma pasta do prefab do jogador há um outro prefab chamado “AIThirdPersonController”. Adicione-o na cena e defina os valores de posição e rotação do objeto criado de acordo com a Figura **Erro! Fonte de referência não encontrada..13 – Inserindo uma IA.**

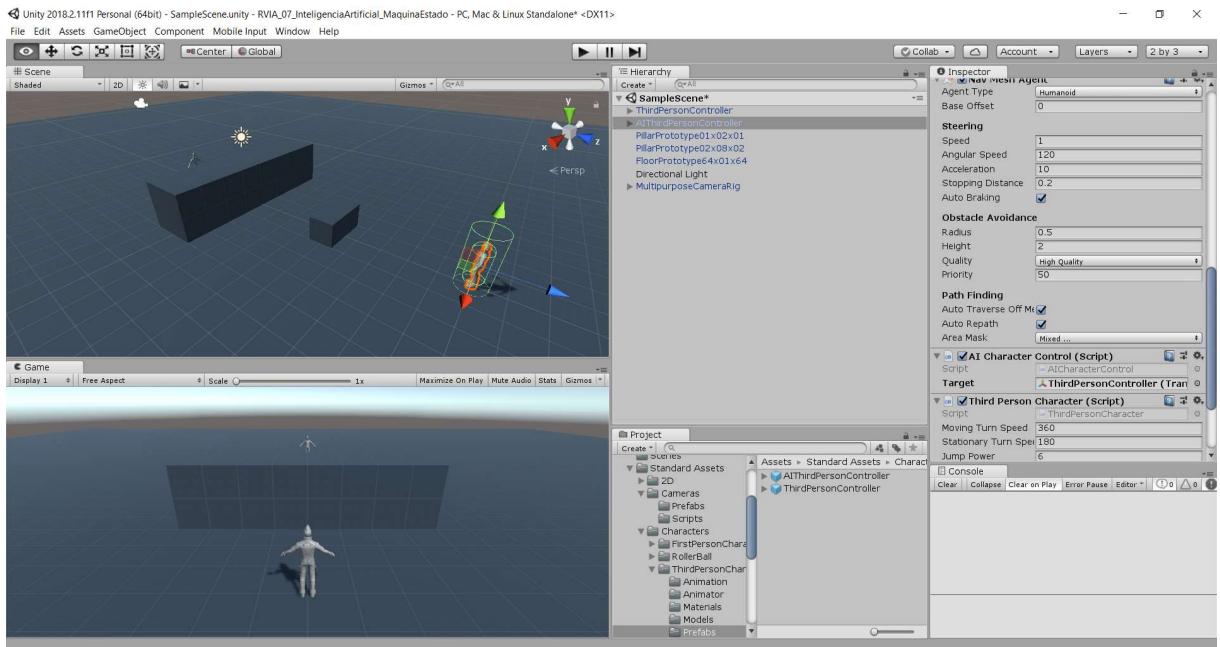
Position: x: 0, y: 0 e z: -10

Rotation: x: 0, y: 180 e z: 0



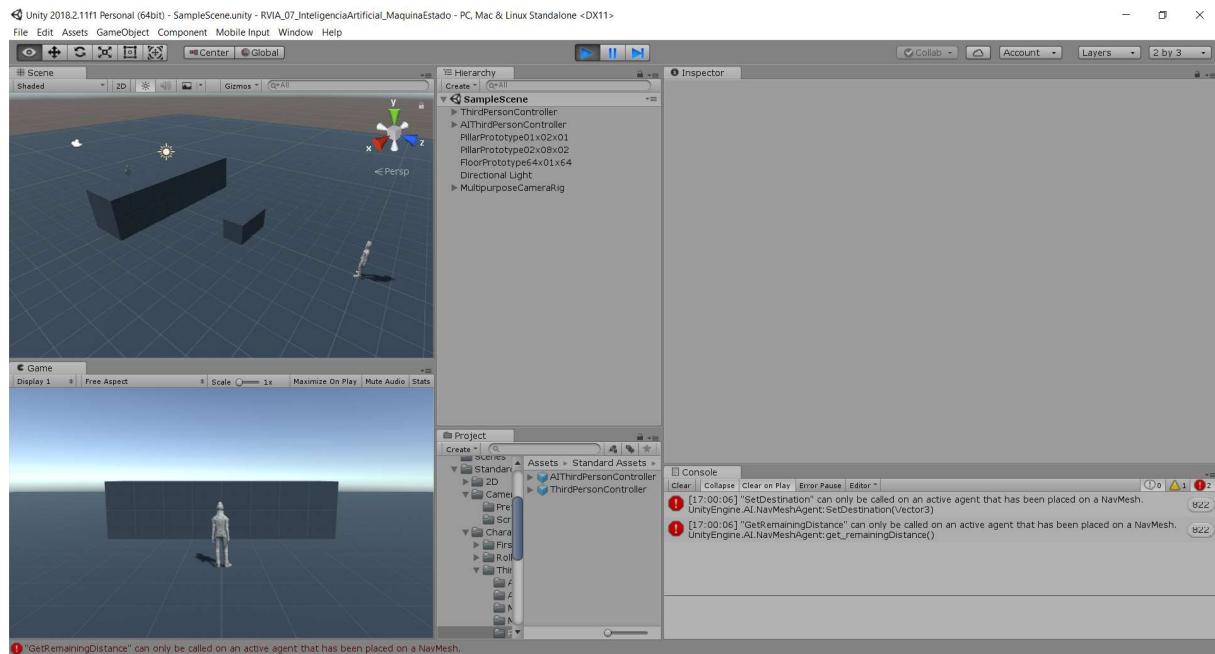
**Figura Erro! Fonte de referência não encontrada..13 – Inserindo uma IA.**  
Fonte: Unity (2018.2.11f1).

Selecione o novo objeto criado da IA e procure pelo script “AI Character Control”. Na propriedade “Target”, arraste o GameObject do jogador, como na Figura **Erro! Fonte de referência não encontrada..14 – Definindo target do AI Character Control.**



**Figura Erro! Fonte de referência não encontrada..14 – Definindo target do AI Character Control.**  
Fonte: Unity (2018.2.11f1).

Salve a cena e dê play.



**Figura Erro! Fonte de referência não encontrada..15 – Erros de NavMesh ausente.**  
Fonte: Unity (2018.2.11f1).

Logo no início da execução, duas mensagens de erros aparecem constantemente.

- "SetDestination" can only be called on an active agent that has been placed on a NavMesh.
- "GetRemainingDistance" can only be called on an active agent that has been placed on a NavMesh.

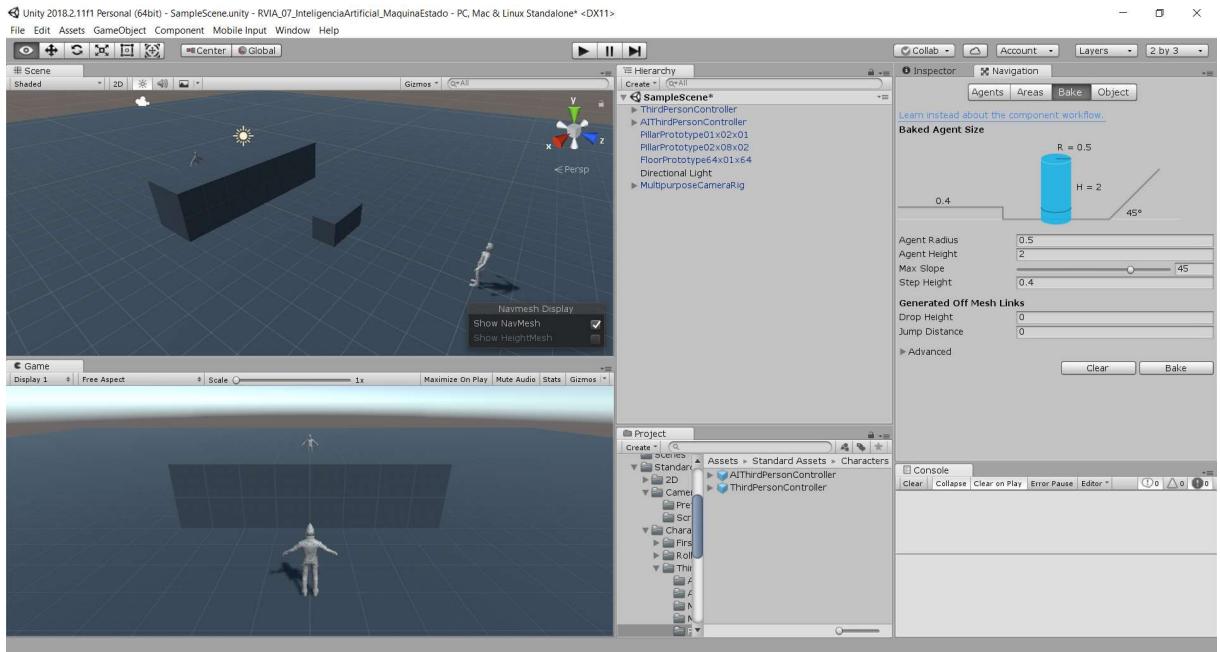
Para corrigir isso, precisamos criar uma malha de navegação do cenário, chamada de NavMesh.

Saia do modo play.

## 7.6 CONSTRUINDO UMA NAVMESH

Como dito em aulas anteriores, a NavMesh é um componente da Unity que calcula o cenário 3D e cria polígonos que definem a área de navegação disponível para os personagens que precisam dessa informação para se movimentarem.

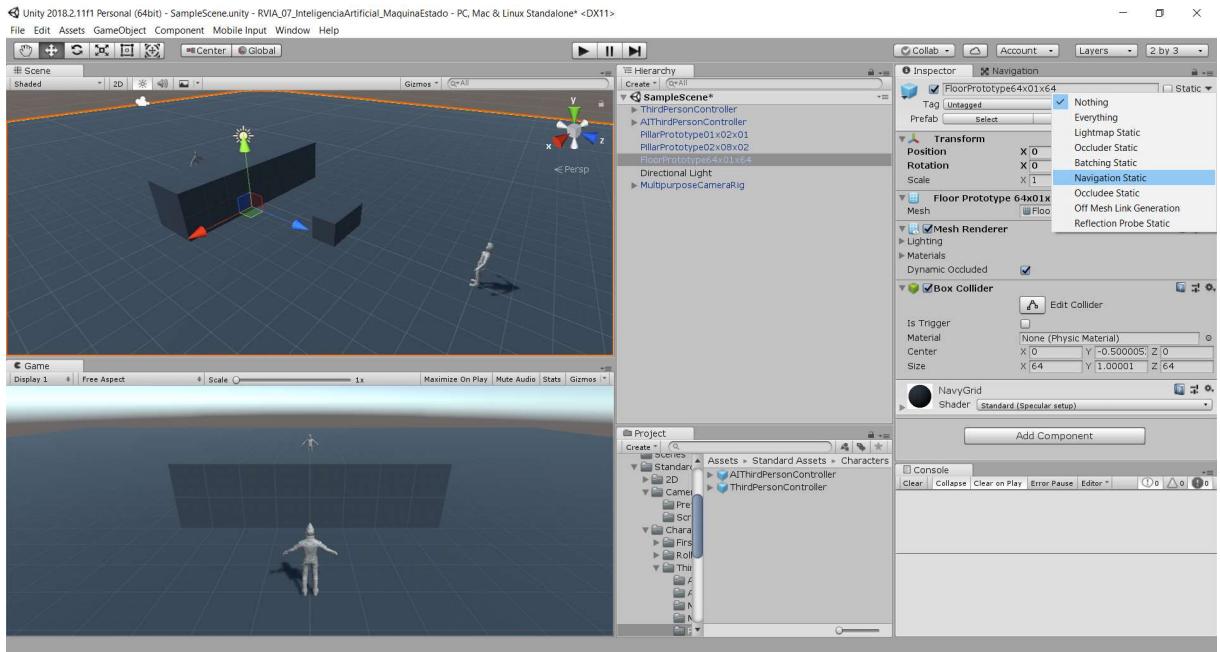
Para calculá-la é muito simples, basta navegar até o menu “Window | AI | Navigation” e nessa nova janela que apareceu, clicar na aba “Bake”, como na Figura Erro! Fonte de referência não encontrada..16 – Navigation Window.



**Figura Erro! Fonte de referência não encontrada..16 – Navigation Window.**  
Fonte: Unity (2018.2.11f1).

O processo é bem simples, nessa nova janela basta clicar no botão “Bake” que a Unity calculará o cenário. Entretanto, note que quando clicamos, simplesmente nada acontece. Isso se dá pelo fato de que a Unity só calcula uma malha de navegação para GameObjects que não se movem durante o jogo, e para que ela saiba essa diferença, devemos marcar esses objetos como “estáticos”.

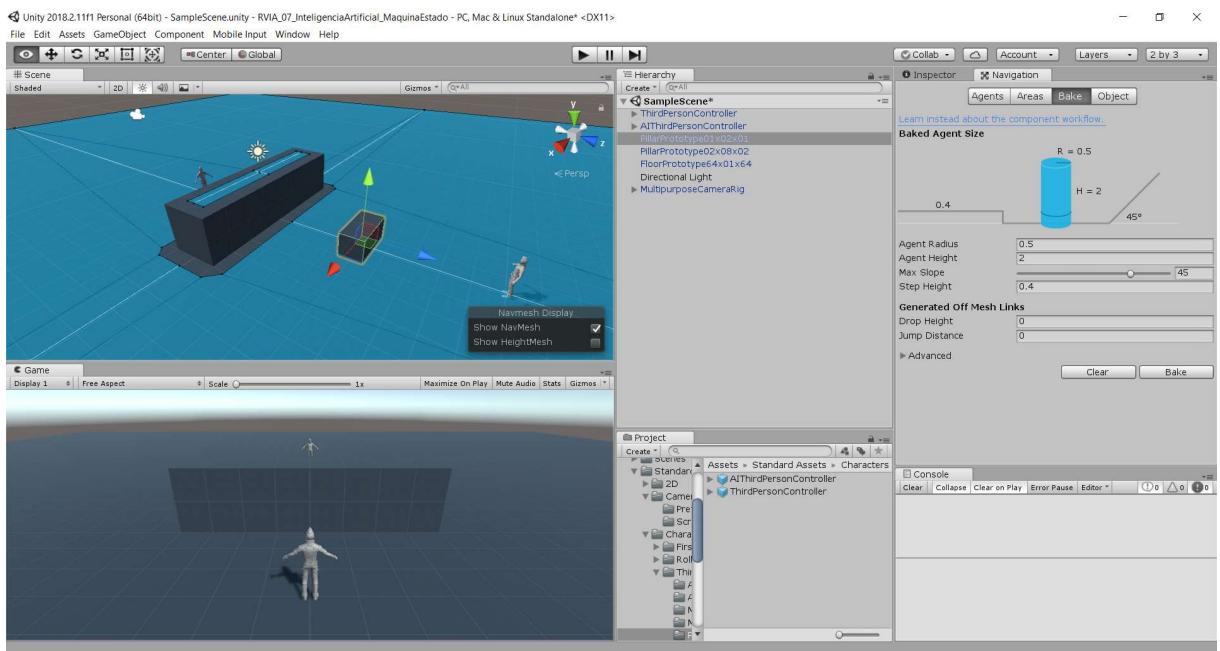
Para fazer isso, selecione o “Floor”, vá até o inspector e marque a opção “Navigation Static” para o objeto, de acordo com a Figura **Erro! Fonte de referência não encontrada..17 – Navigation Static**.



**Figura Erro! Fonte de referência não encontrada..17 – Navigation Static.**  
Fonte: Unity (2018.2.11f1).

Repita o mesmo processo para o pilar maior que está centralizado na cena, com o nome de “PillarPrototype02x08x02”.

Volte para a Navigation Window e clique no botão “Bake”. A Unity irá iniciar o cálculo e uma malha de navegação deverá aparecer sempre que a janela do Navigation estiver aberta, conforme a Figura **Erro! Fonte de referência não encontrada..18 – NavMesh Display**.



**Figura Erro! Fonte de referência não encontrada..18 – NavMesh Display.**

Fonte: Unity (2018.2.11f1).

Salve a cena e dê play no jogo. Note que a IA imediatamente começa a ir em direção do jogador.

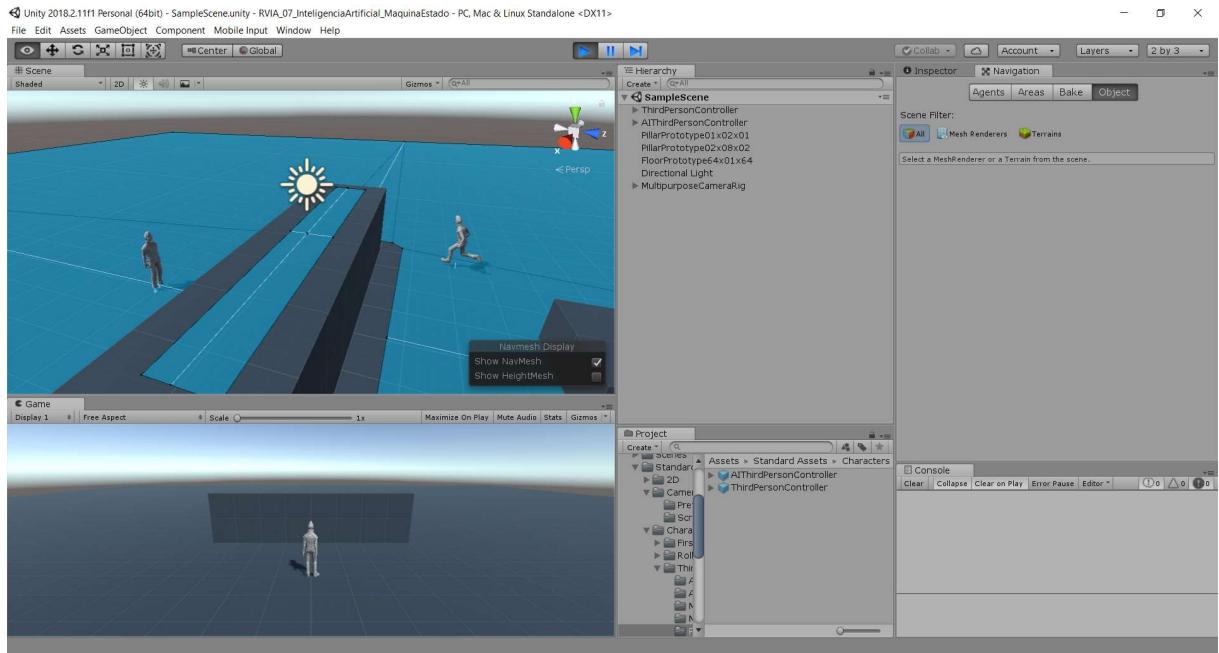


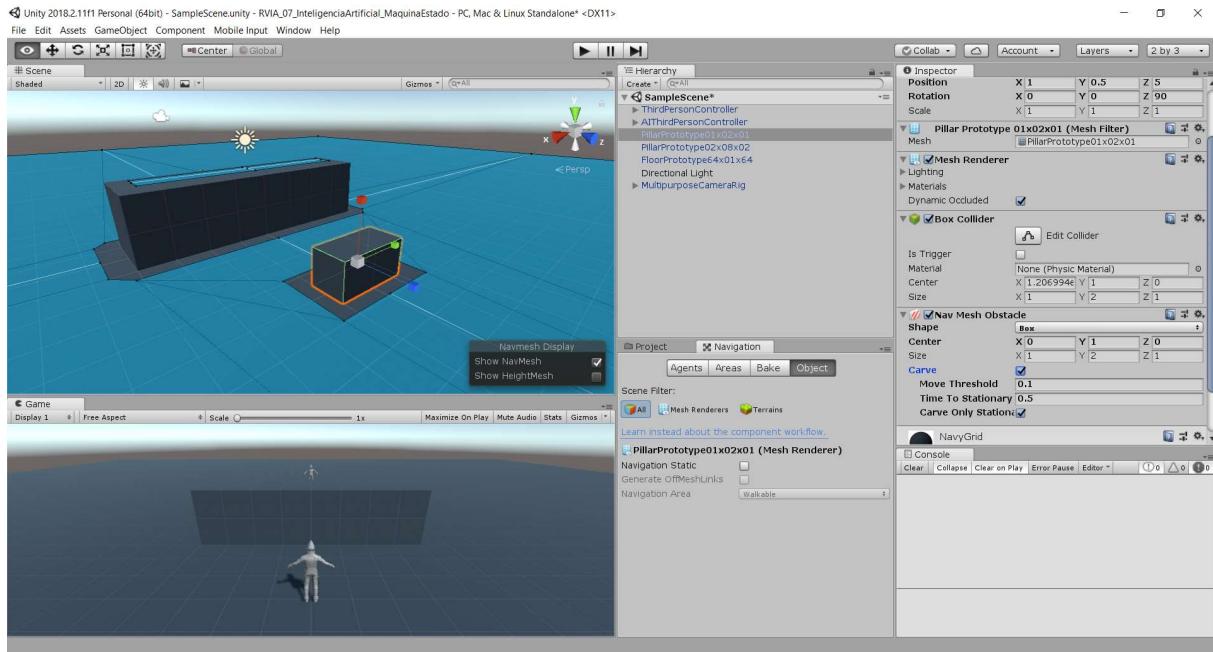
Figura Erro! Fonte de referência não encontrada..19 – IA seguindo o jogador.  
Fonte: Unity (2018.2.11f1).

### 7.6.1 OBSTÁCULOS NA NAVMESH

Na nossa cena atual, temos um chão principal e dois obstáculos. Marcamos apenas o maior obstáculo como Static, fazendo com que a malha de navegação leve em consideração a área desse objeto durante seu cálculo. Entretanto, o obstáculo menor não está sendo considerado na NavMesh. Isso poderia ser facilmente resolvido marcando-o como estático, porém, queremos que esse obstáculo se movimente pelo cenário, o que faz com que não possamos utilizar o atributo Static.

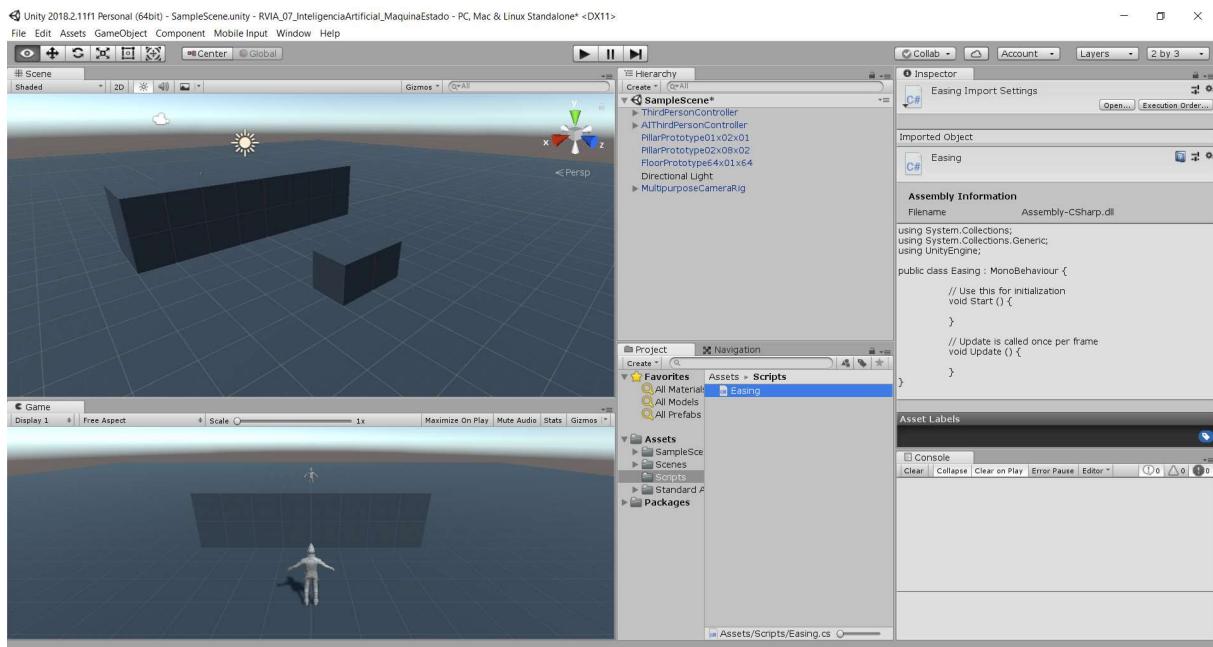
Para que a Unity entenda esse obstáculo e atualize a NavMesh de acordo com sua movimentação, devemos utilizar um componente chamado NavMeshObstacle.

Marque a opção “Carve”, para que a Unity atualize imediatamente a NavMesh, como na Figura Erro! Fonte de referência não encontrada..20 – NavMesh Obstacle.



**Figura Erro! Fonte de referência não encontrada..20 – NavMesh Obstacle.**  
Fonte: Unity (2018.2.11f1).

Para prosseguir, criaremos um script que movimentará o obstáculo lateralmente. Com o mesmo objeto selecionado, clique em “Add Component” e adicione um novo script chamado “Easing”. Aproveite e crie uma pasta chamada “Scripts” dentro da pasta Assets, e move o novo arquivo criado, como na Figura **Erro! Fonte de referência não encontrada..21 – Script Easing organizado na pasta Scripts.**



**Figura Erro! Fonte de referência não encontrada..21 – Script Easing organizado na pasta Scripts.**  
Fonte: Unity (2018.2.11f1).

Começaremos declarando as variáveis iniciais.

A primeira variável definirá o sentido do movimento, que pode ser qualquer valor para x, y e z, por tanto, seu tipo será um Vector3.

A segunda variável definirá a velocidade do movimento, portanto, será do tipo float, para que consigamos ter um ajuste melhor do valor.

A terceira variável definirá a cada quanto tempo o objeto inverterá a direção do movimento.

Todas as variáveis serão públicas para que os valores possam ser alterados direto pela Unity, sem abrir o script novamente. Os valores declarados direto no código são apenas para referência inicial quando a Unity carregá-los pela primeira vez, após esse carregamento, os valores só poderão ser alterados pelo inspector.

```
using UnityEngine;

public class Easing : MonoBehaviour
{
    public Vector3 sentidoMovimento = Vector3.left;
    public float velocidade = 2f;
    public float delayDirecao = 2f;

    void Start()
    {
    }

    void Update()
    {
    }
}
```

O funcionamento do script será muito simples, realizaremos uma movimentação constante do objeto manipulando seu transform direto no update. Para que o obstáculo não se movimente na mesma direção infinitamente, criaremos um método que inverterá a direção do movimento a cada x segundos, definido pela variável ‘delayDirecao’.

Por tanto, criaremos uma variável do tipo int, chamada ‘direcao’, e deixaremos ela como privada. No método Awake, chamaremos um ‘InvokeRepeating’, que recebe o nome de um método a ser executado, um valor em segundos para iniciar a execução e um outro valor, também em segundos, que definirá a cada quantos segundos a chamada será feita novamente. O método que será executado basicamente inverterá

o sinal da variável de direção, fazendo com que o movimento seja realizado na direção oposta.

Utilizaremos o método Awake em vez do Start pois as chamadas que serão executadas somente influenciam o próprio objeto do script. O Awake é um método do ciclo de vida da Unity que executa antes do Start e não necessariamente todos os outros objetos da cena estão carregados. Se tiver interessado em saber mais sobre a ordem de execução dos métodos, consulte o link da documentação “Execution Order of Event Functions”.

Ref.: <https://docs.unity3d.com/Manual/ExecutionOrder.html>

```
using UnityEngine;

public class Easing : MonoBehaviour
{
    public Vector3 sentidoMovimento = Vector3.left;
    public float velocidade = 2f;
    public float delayDirecao = 2f;
    private int direcao = 1;

    void Awake()
    {
        InvokeRepeating("InverterDirecao", delayDirecao, delayDirecao);
    }

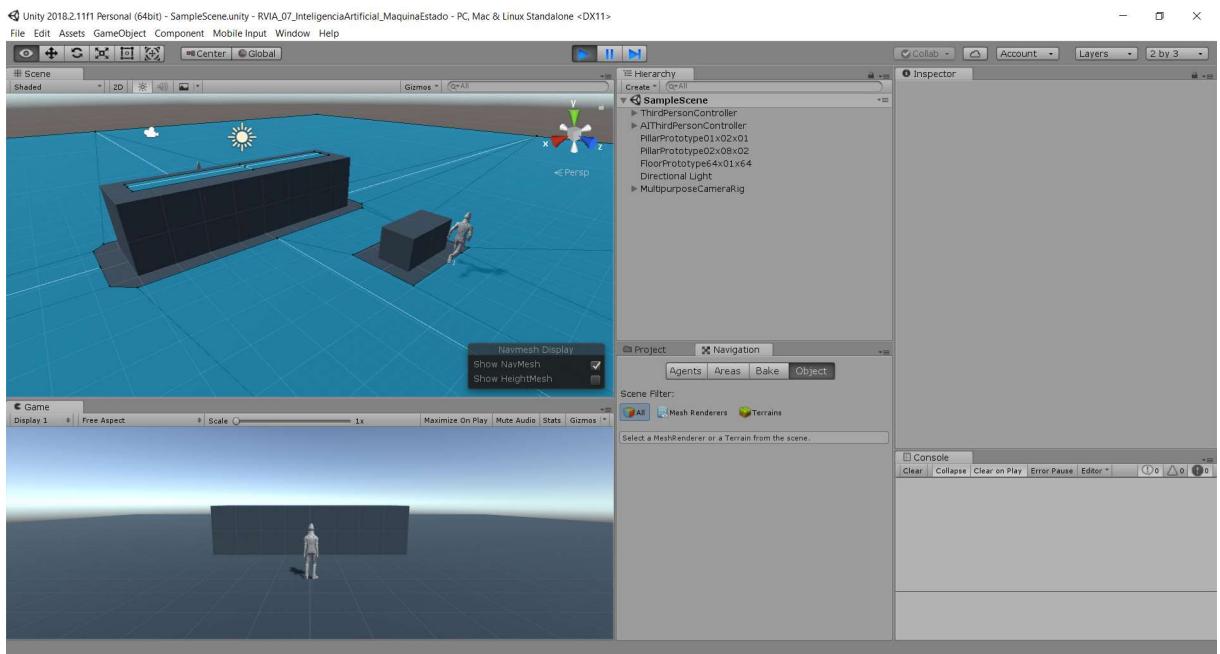
    void Update()
    {
        transform.Translate(
            sentidoMovimento * direcao * velocidade * Time.deltaTime,
            Space.World
        );
    }

    void InverterDirecao()
    {
        direcao *= -1;
    }
}
```

Salve o script e volte para a Unity. Note que em volta do obstáculo há um espaço demarcado na NavMesh. Rode o jogo e perceba que assim que o movimento se inicia esse espaço some. O componente NavMeshObstacle possui algumas variáveis de configuração que precisam ser ajustadas de acordo com o objeto e sua movimentação. Para esse caso, iremos realizar duas modificações, a primeira na variável “Move Threshold”, onde definiremos o valor 0. E a segunda desmarcando a

opção “Carve Only Stationary”, para que a Unity atualize a modificação na NavMesh enquanto o objeto estiver se movimentando.

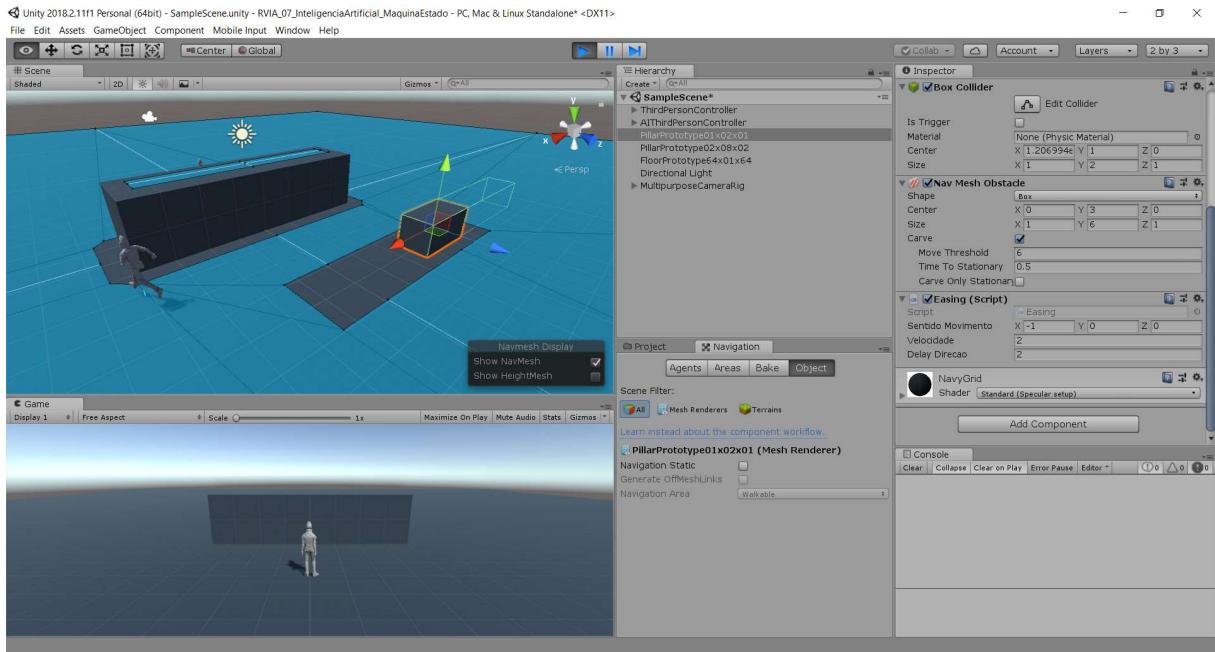
Salve a cena, rode o jogo e veja o resultado.



**Figura Erro! Fonte de referência não encontrada..22 – Movimentação do obstáculo atualizando NavMesh.**  
Fonte: Unity (2018.2.11f1).

Uma outra forma de configuração para esse obstáculo é fazer com que a Unity reserve todo o espaço da movimentação para a NavMesh. Para fazer isso, saia do modo de execução do jogo e selecione o NavMesh Obstacle.

Como pode notar, a primeira propriedade se chama “Shape”, e define o formato de uma espécie de colisor secundário que só é considerado pela NavMesh. Além disso, temos os valores de Center e Size desse colisor. Altere o valor da propriedade Center para x: 0, y: 3 e z: 0. Também altere o valor do Size para x: 1, y: 6 e z: 1. Além disso, precisamos aumentar o Move Threshold, para que a Unity não atualize a NavMesh enquanto o objeto se move. Mude o valor dessa propriedade para 6. O resultado deverá ficar semelhante ao da Figura **Erro! Fonte de referência não encontrada..23 – Todo o espaço de movimentação do obstáculo demarcado pela NavMesh.**



**Figura Erro! Fonte de referência não encontrada..23** – Todo o espaço de movimentação do obstáculo demarcado pela NavMesh.

Fonte: Unity (2018.2.11f1).

Além disso, há também a propriedade “Time To Stationary”, que funcionará em conjunto com a propriedade “Carve Only Stationary”. Ela basicamente define um valor de tempo que o obstáculo atualizará a NavMesh assim que estiver parado. Por exemplo, se um obstáculo está se movendo e encerra seu movimento, ele aguardará o tempo definido em “Time To Stationary” para atualizar a NavMesh, porém, isso só acontecerá se a propriedade “Carve Only Stationary” estiver ativa, caso contrário, o valor de “Move Threshold” será levado em consideração.

Com isso encerramos o capítulo sobre Obstáculos na NavMesh. Estamos prontos agora para aprender mais sobre máquina de estado e navegação na NavMesh.

## 7.7 MÁQUINA DE ESTADO

Como dito na aula de anterior, a máquina de estado finita, também conhecida como FSM – Finite State Machine, é uma das técnicas de IA mais utilizadas e consiste em um modelo com um número finito de estados, sendo que a máquina está em apenas um desses estados por vez, chamado de estado atual. Um estado armazena informações sobre estados anteriores, refletindo as mudanças desde a entrada num

estado. Uma transição indica uma mudança entre estados e ocorre após uma determinada condição ser realizada. Quando a máquina está em determinado estado, ela passa a executar ações que descrevem as atividades a serem realizadas nesse determinado momento, além de sempre verificar se alguma das condições de mudança de estados são atendidas.



Figura Erro! Fonte de referência não encontrada..24 – Representação da Máquina de Estado desse projeto  
Fonte: autor (2019)

Como é possível ver na Figura **Erro! Fonte de referência não encontrada..24** – Representação da Máquina de Estado desse projeto, no exemplo que estamos desenvolvendo iremos criar uma máquina de estado para a inteligência artificial com 4 estados diferentes: Patrulhar, Esperar, Perseguir e Procurar. Esses estados combinados irão tomar decisões que movimentarão o personagem baseado em dados recebidos pelo ambiente. Resumidamente nosso personagem começará em um estado de espera e iniciará uma leitura do cenário, para ver se encontra o jogador por perto ou não.

Caso não encontre nada por um tempo determinado, mudará para o estado de patrulha, que consiste em andar entre dois pontos pré-determinados, que chamaremos de waypoints. Sempre que a IA chegar em um waypoint, ela entrará novamente no estado de espera, alternando entre os waypoints até que o jogador apareça em seu campo de visão.

Quando o jogador aparecer, o estado da IA será alterado para perseguir, fazendo com que ela siga o jogador. Caso o jogador se afaste do campo de visão, a IA mudará para o estado de procurar, que consiste em se movimentar para a última posição conhecida do jogador. Chegando nessa posição sem encontrar nada, mudará para o estado de espera e seguirá o mesmo fluxo citado anteriormente.

### 7.7.1 WAYPOINTS

Antes de começar a programar a nossa IA, devemos preparar os waypoints que servirão para o estado de patrulha. Volte ao projeto, certifique-se de que está fora do modo play, salve a cena e crie um objeto vazio chamado “Waypoints”. Certifique-se de que os valores do Transform estão resetados, conforme a Figura **Erro! Fonte de referência não encontrada..25 – Empty Game Object: Waypoints**.

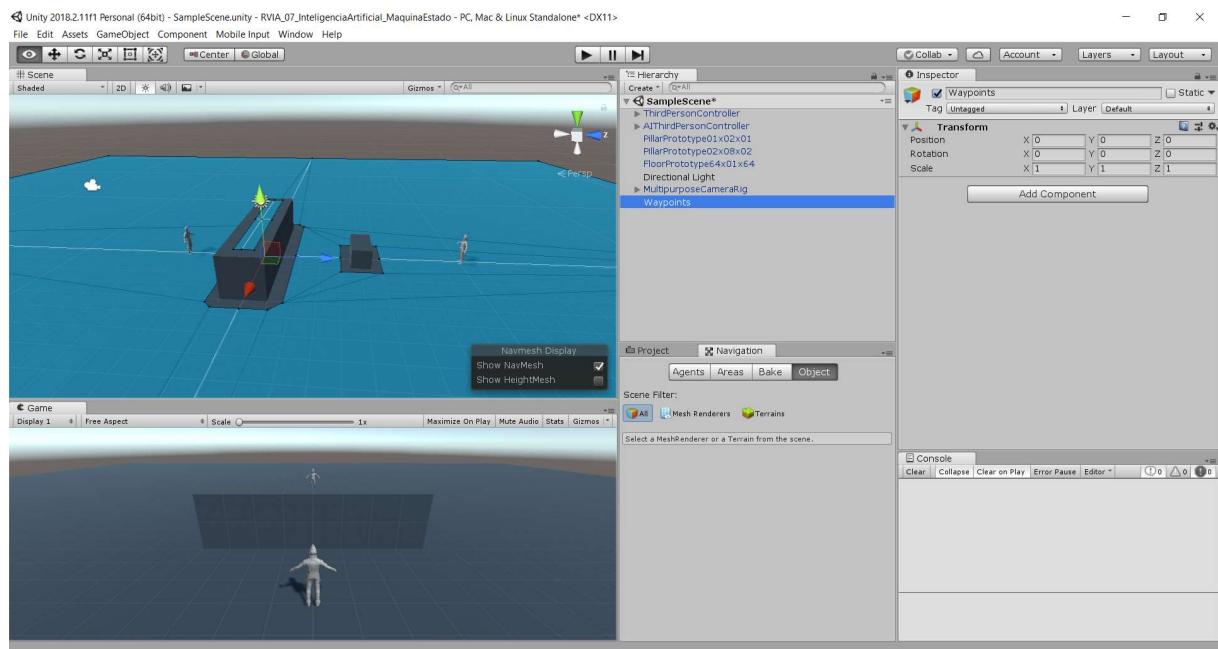
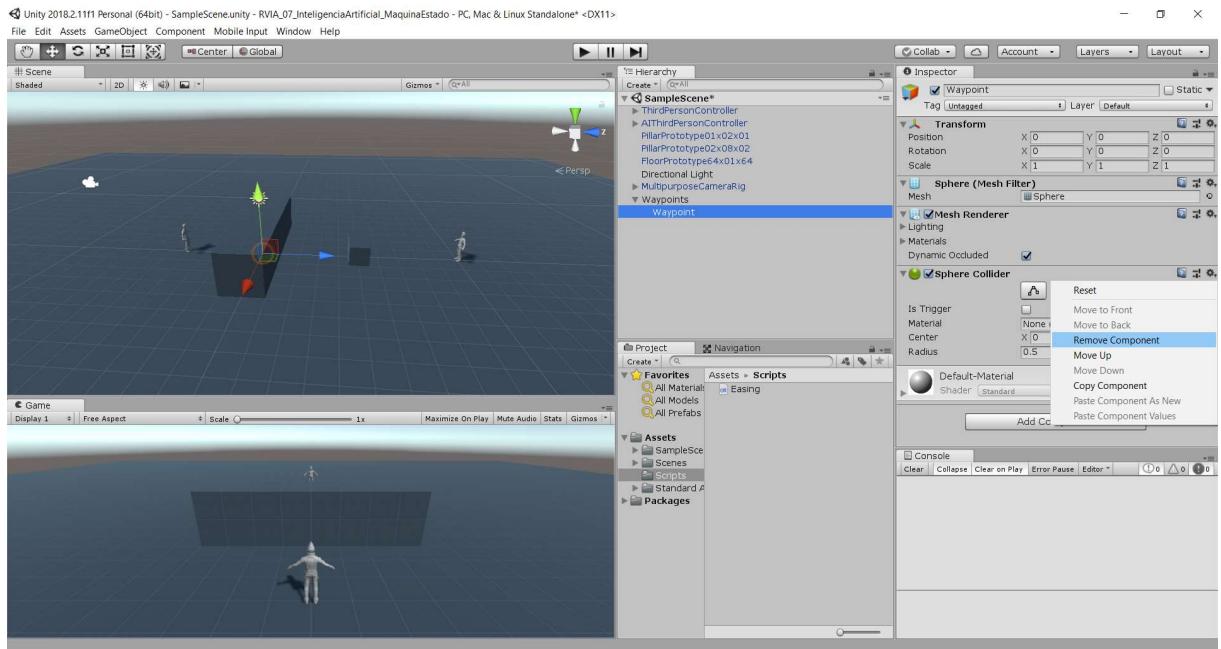


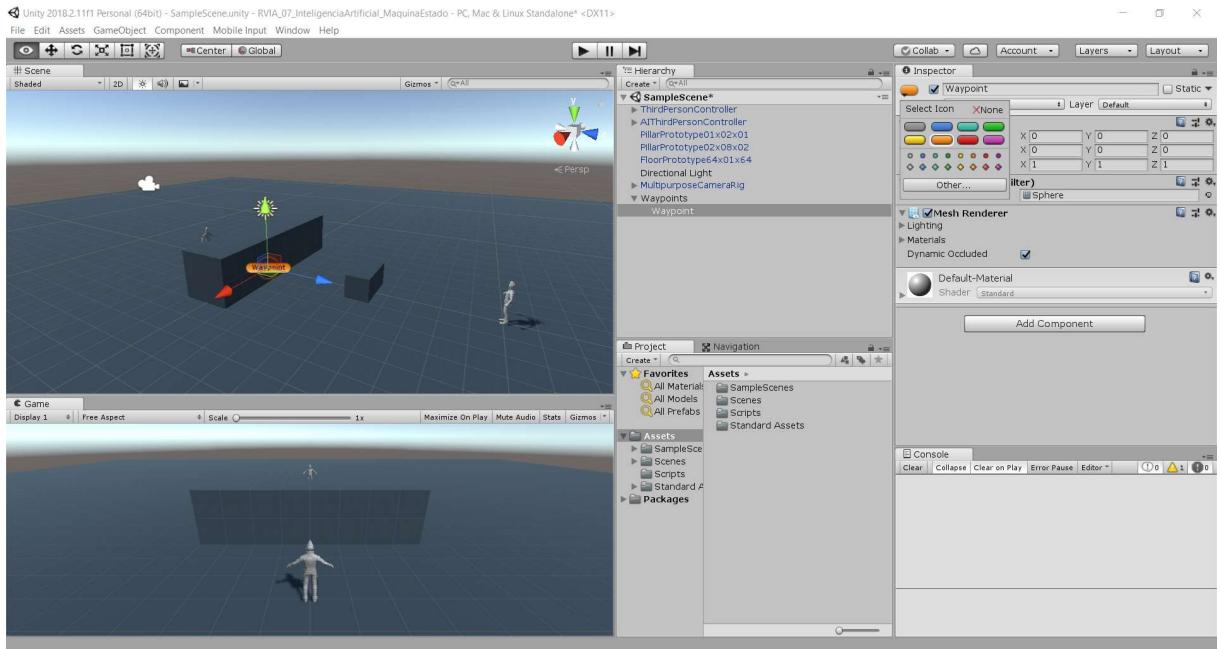
Figura **Erro! Fonte de referência não encontrada..25 – Empty Game Object: Waypoints**.  
Fonte: Unity (2018.2.11f1).

Dentro desse novo objeto, crie uma Sphere, através da Hierarchy, clicando em “Create | 3D Object | Sphere”. Mova ele para filho do objeto “Waypoints”, nomeie como “Waypoint”, resete os valores do Transform e remova o componente “Sphere Collider” desse objeto criado, procurando pelo componente, clicando no ícone da engrenagem e selecionando a opção “Remove Component”, conforme a Figura **Erro! Fonte de referência não encontrada..26 – Criando objeto Waypoint**.



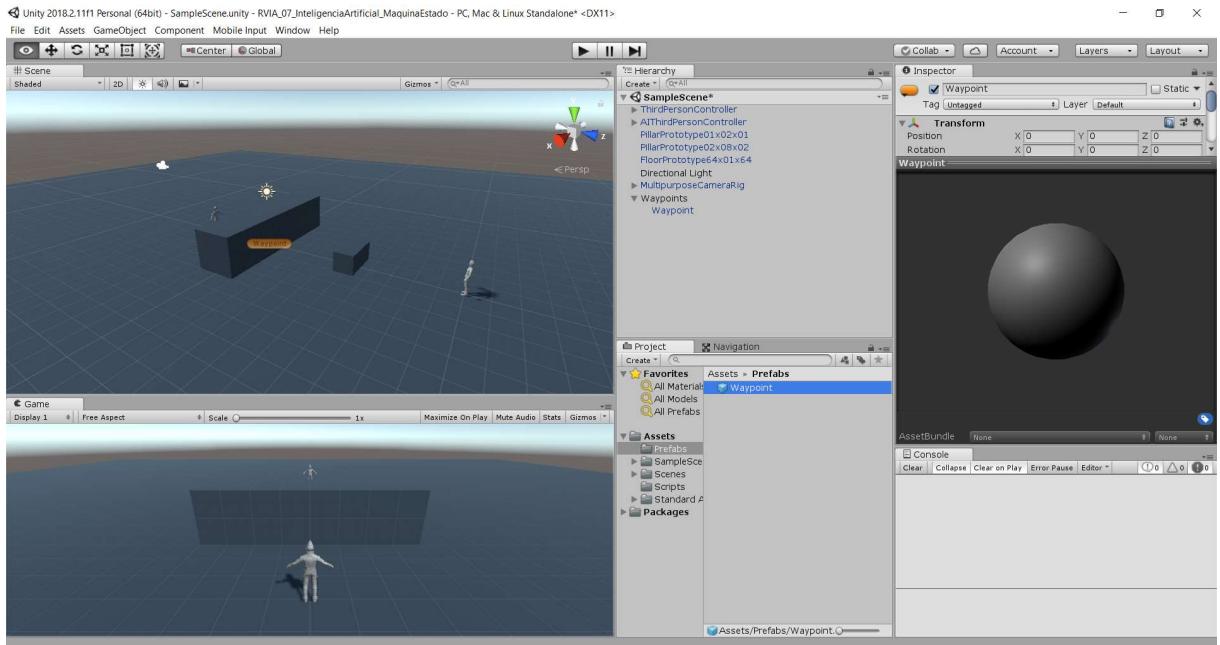
**Figura Erro! Fonte de referência não encontrada..26 – Criando objeto Waypoint.**  
Fonte: Unity (2018.2.11f1).

Se preferir, marque o objeto criado com um ícone, selecionado o objeto e clicando ao lado da caixa de nome no cubo colorido, e selecionando um dos ícones retangulares, que farão com que o nome do objeto apareça na cena, como mostrado na Figura **Erro! Fonte de referência não encontrada..27 – Definindo um ícone para o Waypoint.**



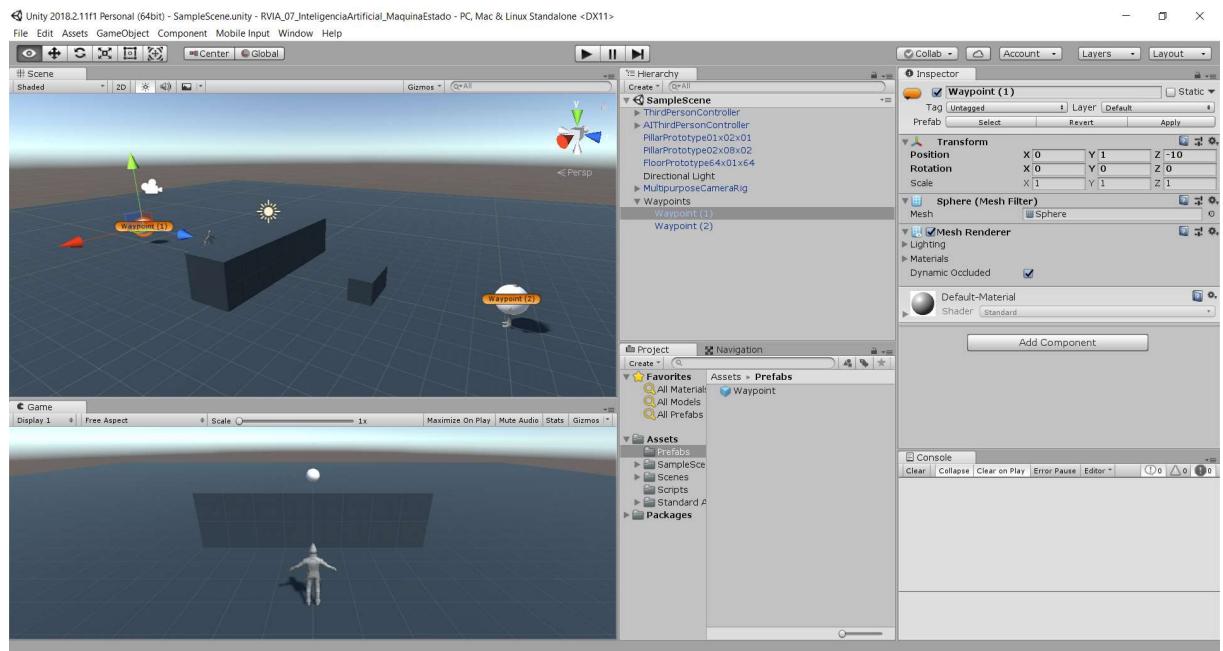
**Figura Erro! Fonte de referência não encontrada..27 – Definindo um ícone para o Waypoint.**  
Fonte: Unity (2018.2.11f1).

Crie uma pasta chamada “Prefabs” dentro da pasta Assets e crie um novo prefab do Waypoint que acabamos de criar. Para criar o Prefab basta arrasta o objeto da cena direto para a aba do Project, dentro da pasta que foi criada. O resultado deverá ser igual à Figura **Erro! Fonte de referência não encontrada..28 – Criando o Prefab do Waypoint.**



**Figura Erro! Fonte de referência não encontrada..28 – Criando o Prefab do Waypoint.**  
Fonte: Unity (2018.2.11f1).

Com o prefab criado, renomeie o Waypoint da cena para “Waypoint (1)”. Com o objeto renomeado, aperte o atalho “Ctrl + D”, que irá duplicar o objeto selecionado. Note que automaticamente a Unity deve nomear o novo objeto como “Waypoint (2)”. Com isso, posicione os Waypoints em locais onde deseja que a IA patrulhe, no meu caso, colocarei o Waypoint (1) na posição x: 0, y: 1 e z: -10, e o Waypoint (2) na posição x: 0, y: 1 e z: 10, conforme a Figura **Figura Erro! Fonte de referência não encontrada..29 – Posicionando e renomeando os waypoints.**



**Figura Erro! Fonte de referência não encontrada..29 – Posicionando e renomeando os waypoints.**  
Fonte: Unity (2018.2.11f1).

Pronto! Agora temos dois waypoints devidamente criados e posicionados. Posteriormente é possível criar scripts mais complexos de waypoints, com mais pontos de parada e diversas outras funcionalidades. Para o nosso exemplo esses dois irão funcionar muito bem. Com isso criado, podemos iniciar o desenvolvimento da nossa inteligência artificial, criando cada estado individualmente para facilitar a compreensão da criação do fluxo proposto.

### 7.7.2 ESPERAR

O primeiro estado, e o mais simples, é o da Espera, que consiste em fazer a IA esperar por uma atualização, passando para o estado de Patrulha ou para o estado

de Perseguição, dependendo das condições que forem satisfeitas durante o tempo em que ela está esperando.

Para começar, selecione o objeto “AIThirdPersonController” e adicione um novo componente que chamaremos de “ControladorIA”. Mova o script que foi criado para a pasta de Scripts, conforme a Figura **Erro! Fonte de referência não encontrada..30 – Script Controlador IA criado e organizado.**

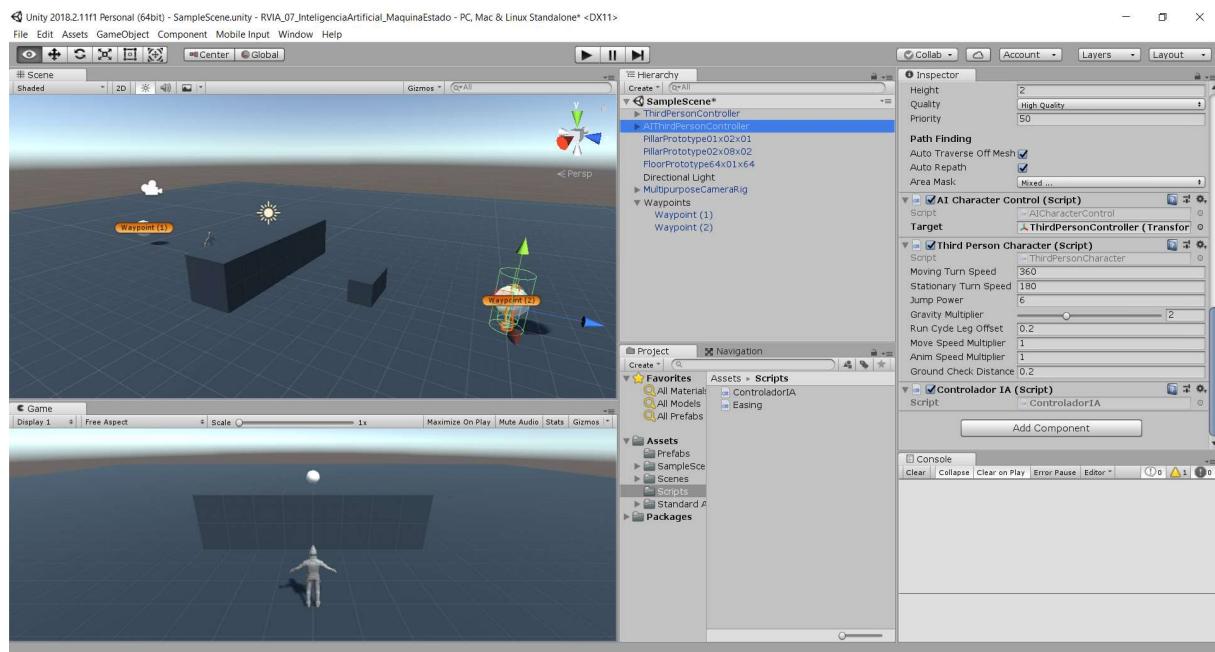


Figura **Erro! Fonte de referência não encontrada..30 – Script Controlador IA criado e organizado.**  
Fonte: Unity (2018.2.11f1).

Abra o script criado. Começaremos declarando os estados da nossa máquina de estados finita. Faremos isso utilizando um tipo de declaração chamado Enumerator. Com esse tipo é possível definir valores pré-determinados para aquela declaração, e a linguagem irá atribuir automaticamente uma sequência numérica para eles, de acordo com a ordem em que estão declarados.

Em outras palavras, podemos posteriormente criar uma variável que possua um tipo personalizado, que é o tipo que criamos o Enumerator, e automaticamente ela só poderá receber os valores que definimos durante sua criação. Dentro da classe criada, adicione as seguintes linhas para construir o enumerator.

```
using UnityEngine;

public class ControladorIA : MonoBehaviour
{
    public enum Estados
    {
```

```
        Esperar,
        Patrulhar,
        Perseguir,
        Procurar
    }
}
```

Depois, iremos declarar uma variável chamada “estadoAtual”, com o tipo “Estados”, que é o novo enumerator criado.

```
using UnityEngine;

public class ControladorIA : MonoBehaviour
{
    public enum Estados
    {
        Esperar,
        Patrulhar,
        Perseguir,
        Procurar
    }

    public Estados estadoAtual;
}
```

Salve o script e vá para Unity para que ele seja carregado. Assim que o carregamento terminar, selecione o objeto “AIThirdPersonController” e desça a barra de rolagem até encontrar o componente “ControladorIA”. Note que uma nova informação apareceu, e automaticamente a Unity criou um dropdown com as opções que configuramos. Por trás de cada valor que podemos escolher, há um valor inteiro que foi definido automaticamente, como é possível ver na Figura **Erro! Fonte de referência não encontrada..31** – Valores atribuídos automaticamente para cada item do enumerator.

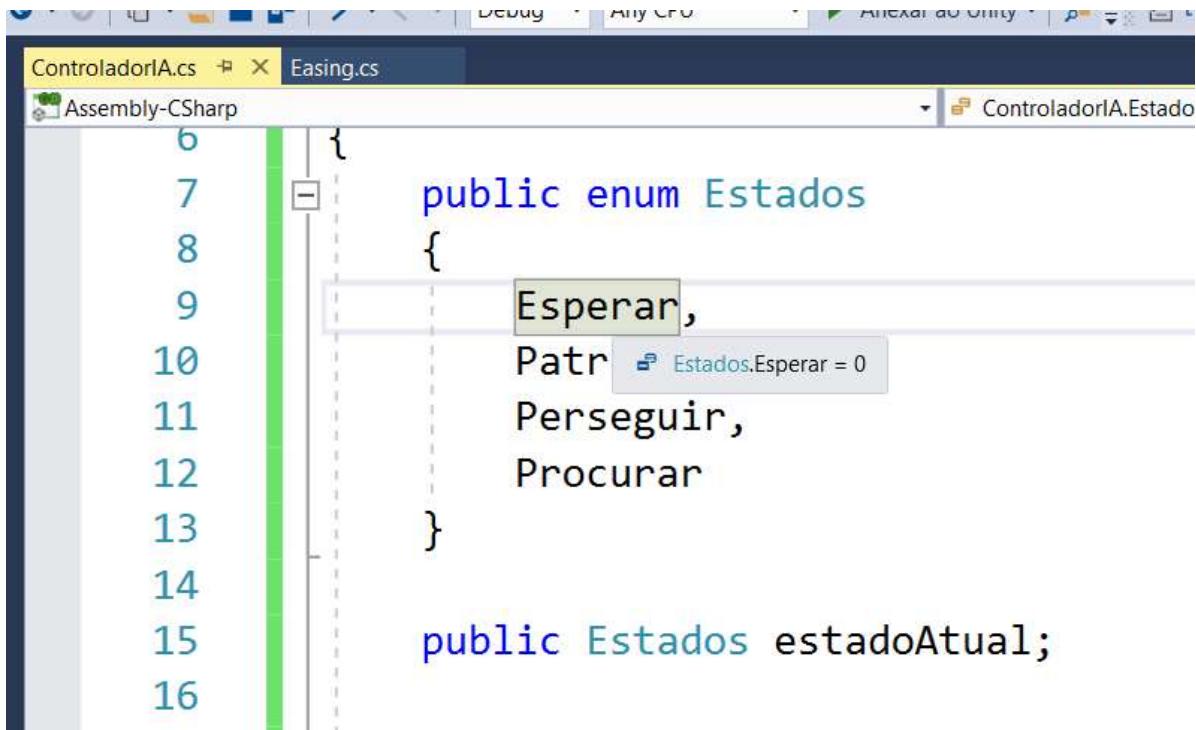


Figura Erro! Fonte de referência não encontrada..31 – Valores atribuídos automaticamente para cada item do enumerator.  
Fonte: Unity (2018.2.11f1).

Entretanto, a variável do estado atual não poderá ser alterada através do Inspector, pois quem fará a atualização dessa informação é o próprio script. Por tanto, alteraremos o nível de acesso dessa variável para “private”, em vez de “public”.

Note que não definimos nenhum valor para a variável “estadoAtual”, portanto, o valor que será atribuído é o primeiro enumerator disponível, que no caso é o “Esperar”.

O ato de esperar consiste no seguinte comportamento: temos um determinado tempo que a IA deve aguardar e devemos sempre checar se o tempo que a IA está esperando é suficiente. Caso seja, iniciamos a patrulha. Para isso, precisamos criar duas variáveis: uma que defina quanto tempo a IA deve esperar e outra que define qual o momento em que a IA iniciou a espera, para que consigamos comparar isso sempre e tomar uma decisão baseada nessas informações atualizadas.

O tempo que a IA irá esperar deve ser público, pois queremos alterar essa informação pelo inspector, já o tempo que a IA iniciou a espera será definido pelo script, então podemos deixá-lo como privado.

```
using UnityEngine;
public class ControladorIA : MonoBehaviour
```

```

{
    public enum Estados
    {
        Esperar,
        Patrulhar,
        Perseguir,
        Procurar
    }

    private Estados estadoAtual;

    [Header("Estado: Esperar")]
    public float tempoEsperar = 2f;
    private float tempoEsperando;
}

```

Como isso, iremos criar e chamar alguns métodos. Criaremos um método para checagem de estados, que chamaremos de “ChecarEstados()”, e outro para iniciar o estado de espera, que chamaremos de “Esperar()”, que atualiza o estadoAtual e grava o tempo que a espera começou. Para que a IA comece esperando e obtenha o funcionamento esperado, chamaremos o método dentro do Start() e para que a IA cheque os estados sempre que o jogo esteja acontecendo iremos chamar o método de checagem de estados dentro do Update().

```

using UnityEngine;

public class ControladorIA : MonoBehaviour
{
    public enum Estados
    {
        Esperar,
        Patrulhar,
        Perseguir,
        Procurar
    }

    private Estados estadoAtual;

    [Header("Estado: Esperar")]
    public float tempoEsperar = 2f;
    private float tempoEsperando;

    void Start()
    {
        Esperar();
    }

    void Update()
    {
        ChecarEstados();
    }

    private void ChecarEstados()
    {
    }
}

```

```
// Estados

private void Esperar()
{
    estadoAtual = Estados.Esperar;

    tempoEsperando = Time.time;
}
```

Com isso, precisamos criar uma rotina dentro do método de checagem de estados para fazer com que a IA fique parada enquanto estiver esperando. Primeiro, precisamos criar uma variável que será o alvo da IA. Esse alvo será do tipo Transform, e poderá ser inclusive a própria IA, quando quisermos que ela fique parada.

Além disso, o alvo real da IA é definido pelo script “AICharacterControl”, através da variável “Target”, portanto, também devemos saber como atualizar o componente em questão, que será feito no método Awake(), utilizando a declaração que conhecemos do “GetComponent”. Note que para utilizar o script AICharacterControl devemos importar o namespace “ThirdPerson” do StandardAssets, como o código abaixo indica.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    // Movimentação da IA
    private AICharacterControl aiCharacterControl;
    private Transform alvo;

    private void Awake()
    {
        aiCharacterControl = GetComponent<AICharacterControl>();
    }

    // ...
}
```

Agora que sabemos atualizar as informações do alvo, devemos checar em qual estado estamos e atualizar a propriedade “Target” com o valor que encontramos. Para iniciar a checagem de estados, vamos fazer uma declaração de “switch” dentro do método “ChecarEstados()”. Para facilitar, iremos utilizar alguns atalhos do Visual Studio (talvez não funcionem em outras IDEs).

O primeiro passo a se fazer é digitar “switch” e apertar a tecla “tab” duas vezes. Perceba que a IDE construiu a declaração e deixou selecionada a palavra “switch\_on”. Sem clicar em nada, digite no lugar de switch\_on a palavra “estadoAtual”, de acordo com o nome da variável que criamos. Aperte enter uma vez para selecioná-la no autocomplete e pressione enter mais uma vez para confirmar a inserção.

Perceba que a IDE deverá construir um caso para cada enumerator que declaramos na construção de “Estados” e um caso “default” que é uma declaração padrão do switch.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...
    private void ChecarEstados()
    {
        switch (estadoAtual)
        {
            case Estados Esperar:
                break;
            case Estados Patrulhar:
                break;
            case Estados Perseguir:
                break;
            case Estados Procurar:
                break;
            default:
                break;
        }
    }
    // ...
}
```

Geralmente é recomendado a utilização do caso “default”, porém, como essa variável é do tipo enumerator e todos os casos estão mapeados, não somos obrigados a deixá-lo, pois ele nunca será lido. Você removê-lo se quiser economizar algumas linhas ou deixá-lo se quiser utilizar como referência para lembrar dessa declaração quando quiser. Caso remova, apague a declaração do “break;” junto.

Prosseguindo com o estado Esperar, devemos checar se a IA esperou tempo suficiente, para isso, criaremos um método chamado “EsperouTempoSuficiente()”, que retorna verdadeiro ou falso. Caso ela tenha esperado, alteramos para o estado de Patrulha, caso contrário, alteraremos o alvo para o próprio transform da IA (para que ela fique parada) e encerramos a checagem do estado.

Por fim, no final do método “ChecarEstados()”, assim que o comportamento do estado atual foi processado, iremos atualizar o alvo da IA, para que ela se movimente ou não.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void ChecarEstados()
    {
        switch (estadoAtual)
        {
            case Estados.Esperar:
                if (EsperouTempoSuficiente())
                {

                }
                else
                {
                    alvo = transform;
                }

                break;
            case Estados.Patrulhar:
                break;
            case Estados.Perseguir:
                break;
            case Estados.Procurar:
                break;
        }

        aiCharacterControl.target = alvo;
    }

    // ...

    // Métodos do Estado Esperar

    private bool EsperouTempoSuficiente()
    {
        return true;
    }
}
```

Prosseguindo com a implementação do método “EsperouTempoSuficiente()”, devemos pensar o que representa em números que a espera da IA terminou.

Temos duas variáveis, tempoEsperar, que determina quando tempo a IA deve esperar, e tempoEsperando, que determina quando que a IA iniciou a espera. Ambas são do tipo float, dentro que a variável tempoEsperando guarda a informação que

estava na variável “Time.time” no momento que a espera foi iniciada. Time.time representa o tempo em segundos desde que o jogo foi iniciado.

Para saber se a IA esperou tempo suficiente, devemos comparar o tempo do jogo atual com o tempo que a IA iniciou a espera somado com o tempo que ela deve esperar. Por tanto, vamos supor:

Tempo que a IA iniciou a espera (tempoEsperando): 4 segundos

Tempo que a IA deve esperar (tempoEsperar): 2 segundos

Tempo atual do jogo (Time.time): 5 segundos

`tempoEsperando + tempoEsperar <= Time.time`

`4 + 2 <= 5`

`6 <= 5` é falso.

Essa expressão retornaria “falso”, ou seja, para que a IA espere tempo suficiente o valor da soma entre “tempoEsperando” e “tempoEsperar” deve ser menor ou igual ao “Time.time”.

Como o método deve retornar uma informação booleana (true ou false), iremos retornar direto o valor da expressão, que corresponde à informação que queremos inserir no if que foi chamado dentro do método “ChecarEstados()”.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    // Métodos do Estado Esperar

    private bool EsperouTempoSuficiente()
    {
        return tempoEsperando + tempoEsperar <= Time.time;
    }
}
```

Com isso, caso a IA tenha esperado tempo suficiente a informação retornada por esse método será true e o if será executado. Quando esse if for executado, significa que a transição para o estado da Patrulha está autorizada.

### 7.7.3 PATRULHAR

O estado da patrulha defina para a IA um dos waypoints que criamos como alvo da movimentação. Assim que a IA chegar nesse waypoint, alteramos o estado para Esperar, e atualizamos para que ela siga o próximo waypoint caso entre em modo de patrulha novamente.

Antes de começar a construção do comportamento durante esse estado, devemos iniciar sua chamada na checagem do estado Esperar, que criamos no capítulo anterior.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void ChecarEstados()
    {
        switch (estadoAtual)
        {
            case Estados.Esperar:
                if (EsperouTempoSuficiente())
                {
                    Patrulhar();
                }
                else
                {
                    alvo = transform;
                }
                break;
            case Estados.Patrulhar:
                break;
            case Estados.Perseguir:
                break;
            case Estados.Procurar:
                break;
        }

        aiCharacterControl.target = alvo;
    }

    // ...
    // Estados
    // ...

    private void Patrulhar()
    {
        estadoAtual = Estados.Patrulhar;
    }
}
```

```
// ...
}
```

Quando o estado da Patrulha estiver ativo, iremos fazer com que a IA siga um determinado waypoint. Nesse nosso exemplo, temos apenas dois waypoints. Para começar a programar o comportamento da IA nesse estado precisamos declarar algumas variáveis.

Duas variáveis para saber quem são os waypoints 1 e 2, sendo que elas serão do tipo Transform, pois apenas precisamos de informações relacionadas a posicionamento desses objetos. O modificador de acesso das variáveis deve ser público, pois queremos atribuí-las pelo Inspector da Unity. Além de saber os 2 waypoints disponíveis, precisamos manter um controle de qual é o waypoint atual que a IA está seguindo, para saber quando alternar para o outro ponto demarcado.

Quando a IA estiver em direção ao ponto que mandamos, precisamos saber se ela chegou até o waypoint. Como estamos trabalhando com dois pontos em um espaço 3D, esse valor não é preciso, ou seja, pode ser que a IA chegue apenas perto do waypoint e pare o movimento. Para certificar de que a IA está próxima do waypoint, utilizaremos um valor float que definirá a distância mínima que deve ser atingida para sabermos que a IA está naquele waypoint.

Por fim, teremos uma variável global que irá armazenar o valor da distância entre a IA e o waypoint atual, para que consigamos visualizar o valor dessa informação pelo Inspector. Essa variável será privada, portanto só poderemos visualizá-la quando o modo Debug do Inspector estiver ativado.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    [Header("Estado: Esperar")]
    public float tempoEsperar = 2f;
    private float tempoEsperando;

    [Header("Estado: Patrulha")]
    public Transform waypoint1;
    public Transform waypoint2;
```

```

private Transform waypointAtual;
public float distanciaMinimaWaypoint = 2f;
private float distanciaWaypointAtual;

// ...

}

```

Salve o script e volte para a Unity. Espere o código compilar e selecione o objeto da IA. As variáveis públicas declaradas devem aparecer no Inspector. Atribua o waypoint 1 e 2 para as variáveis correspondentes, conforme mostrado na Figura **Erro! Fonte de referência não encontrada..32** – Waypoints atribuídos nas variáveis da IA.

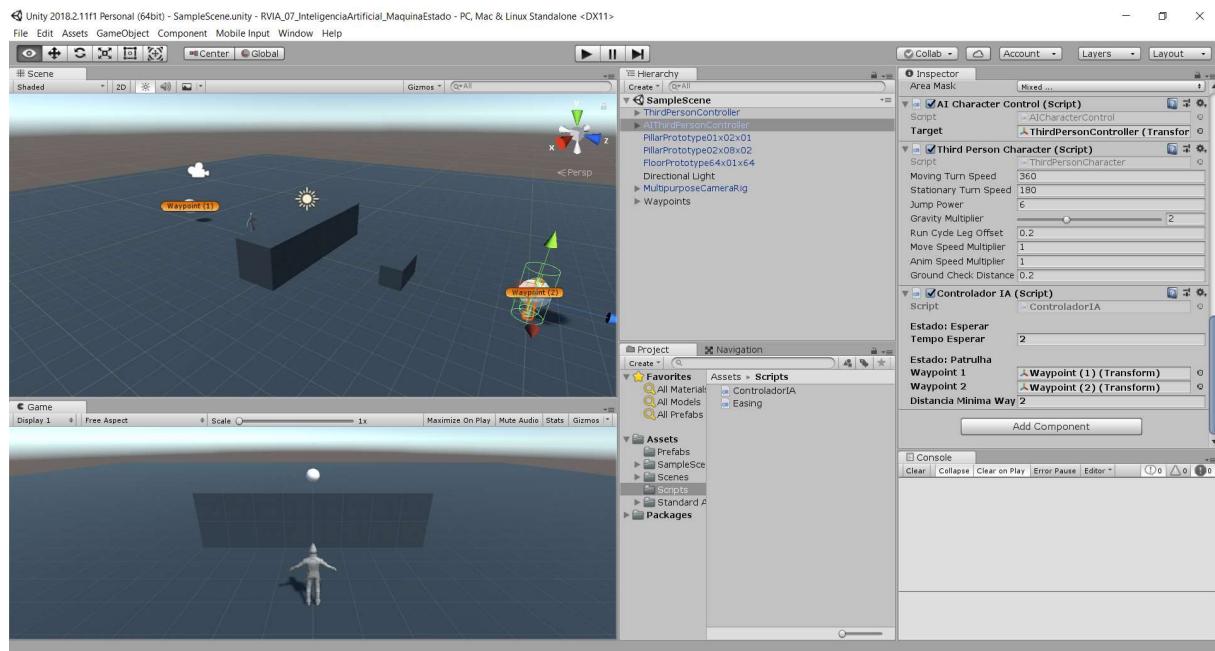
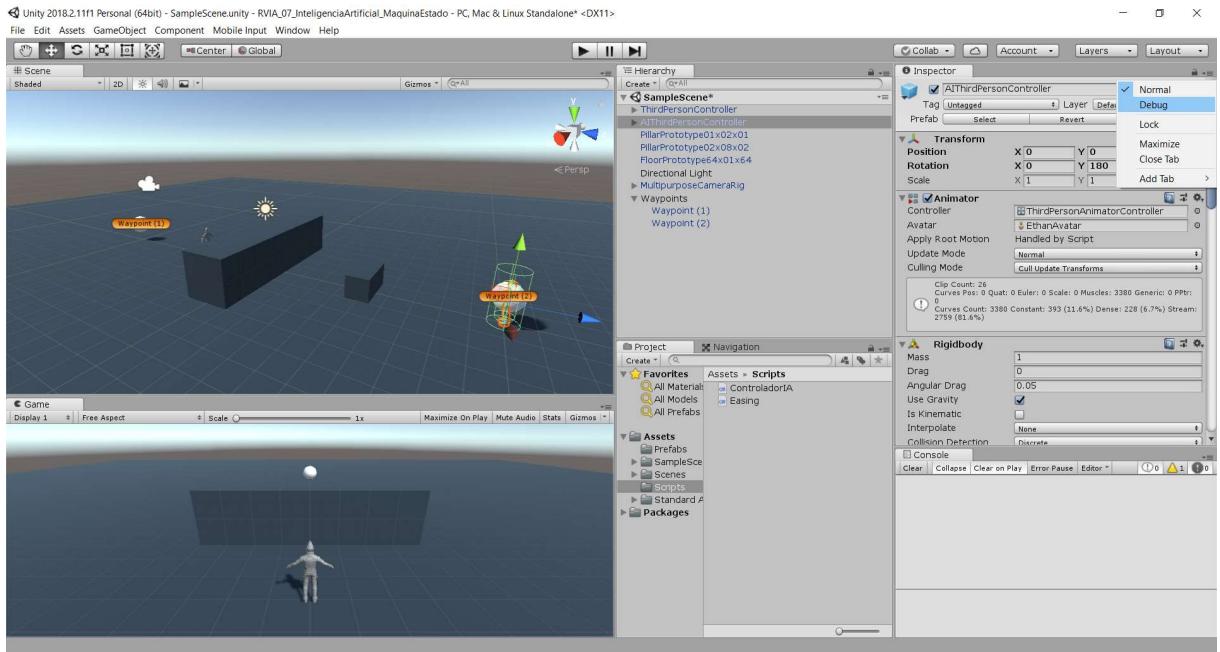


Figura **Erro! Fonte de referência não encontrada..32** – Waypoints atribuídos nas variáveis da IA.  
Fonte: Unity (2018.2.11f1).

Note que as variáveis privadas não aparecem, como já sabemos. Entretanto, podemos visualizar o valor que a Unity está armazenando para elas se alterarmos o tipo de visualização do Inspector. Para fazer isso, basta ir no Inspector e clicar no ícone das 3 barrinhas, ao lado do cadeado, e selecionar a opção “Debug”, conforme mostrado na Figura **Erro! Fonte de referência não encontrada..33** – Ativar modo Debug no Inspector.



**Figura Erro! Fonte de referência não encontrada..33 – Ativar modo Debug no Inspector.**  
Fonte: Unity (2018.2.11f1).

Quando ativamos esse modo, fica muito mais fácil entender o que está acontecendo com os nossos componentes, pois visualizamos todas as informações que antes ficavam escondidas, sem precisar ficar exibindo-as no console, basta visualizar direto no Inspector. Note que a variável de distância do waypoint atual aparece logo abaixo da distância mínima do waypoint, isso nos ajudará na hora de saber qual valor de distância define se a IA está próxima do waypoint ou não, o que pode variar por vários motivos.

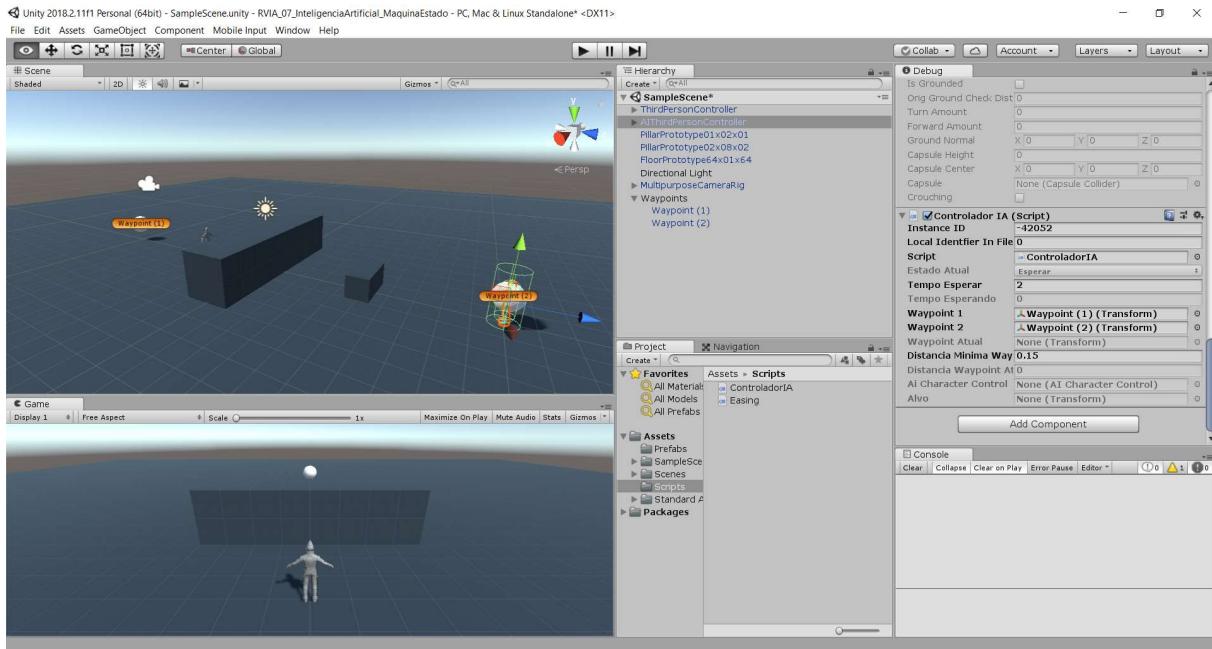


Figura Erro! Fonte de referência não encontrada..34 – Visualização das variáveis privadas no Inspector.

Fonte: Unity (2018.2.11f1).

Vamos começar a escrever o funcionamento do estado de Patrulha. Começaremos definindo o waypoint atual padrão da IA, que será o waypoint 1, sendo que isso deverá ser feito assim que jogo começar, portanto, declararemos essa chamada no método Start().

No método de checagem de estados, criaremos a lógica do estado Patrulhar, que irá checar se a IA está perto do waypoint atual, se estiver, iremos mandá-la esperar e alteraremos o waypoint, para que ela vá em direção ao outro waypoint. Caso contrário, apenas definiremos seu alvo como o waypoint atual. Para a lógica de checagem se está perto, criaremos um método chamado “PertoWaypointAtual()”, que retornará um booleano informando se está perto ou não. Já para a alteração do waypoint, criaremos um outro método chamado “AlterarWaypoint()”.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    void Start()
    {
        waypointAtual = waypoint1;
        Esperar();
    }
}
```

```

// ...

private void ChecarEstados()
{
    switch (estadoAtual)
    {
        // ...
        case Estados.Patrulhar:
            if (PertoWaypointAtual())
            {
                Esperar();

                AlterarWaypoint();
            }
            else
            {
                alvo = waypointAtual;
            }

            break;
        // ...
    }

    aiCharacterControl.target = alvo;
}

// ...

// Métodos do Estado Patrulhar

private bool PertoWaypointAtual()
{
    return true;
}

private void AlterarWaypoint()
{
}
}

```

Para construir o conteúdo dos métodos é muito simples. Iniciaremos pelo método que checa se a IA está perto do waypoint atual. Para fazer isso precisaremos saber a distância entre os dois. Sempre que precisarmos obter a distância entre dois pontos de Vector3 podemos usar um método chamado “Vector3.Distance()”, que recebe dois pontos de Vector3 e retorna um valor único de float já calculado. Armazenaremos esse cálculo na variável que chamamos de “distanciaWaypointAtual”.

O método “PertoWaypointAtual()” precisa retornar um valor booleano, ou seja, true ou false, sendo que true representa que a IA está perto do waypoint. Quanto menor o valor recebido pelo Vector3.Distance(), mais próximo os pontos estão,

portanto, para sabermos se a IA está perto do waypoint, devemos checar se o valor armazenado em “distanciaWaypointAtual” é menor ou igual ao valor armazenando em “distanciaMinimaWaypoint”.

```
// ...
private bool PertoWaypointAtual()
{
    distanciaWaypointAtual = Vector3.Distance(
        transform.position,
        waypointAtual.position
    );
    return distanciaWaypointAtual <= distanciaMinimaWaypoint;
}
// ...
```

Para o método de alteração do waypoint, devemos checar se o waypoint atual é o waypoint 1, caso seja, mudamos o valor da variável para o waypoint 2, caso contrário, mudamos para o waypoint 1. Como estamos utilizando um if/else com apenas uma linha e ambas declarações estão alterando o valor de uma variável, podemos utilizar uma expressão conhecida como “if ternário”, que consiste em um if de uma linha.

```
if (waypointAtual == waypoint1)
    waypointAtual = waypoint2;
else
    waypointAtual = waypoint1;
```

No if ternário colocamos a lógica de checagem na mesma linha que atribui o valor da variável, ficando:

```
waypointAtual = (waypointAtual == waypoint1) ? waypoint2 : waypoint1;
```

Atribuímos no waypointAtual o que sair do if ternário. A primeira informação é um booleano, que checa se o waypoint atual é igual ao waypoint 1. O ponto de interrogação é chamado caso essa condição seja verdadeira, e o valor depois do ponto de interrogação é imediatamente armazenado na variável do começo da linha. Caso esse valor seja falso, o valor depois do símbolo de “dois pontos” é imediatamente solicitado, e armazenado na variável do começo da linha. Finalizando a construção do método, fica:

```
// ...
```

```
private void AlterarWaypoint()
{
    waypointAtual = (waypointAtual == waypoint1) ? waypoint2 : waypoint1;
}

// ...
```

Salve o código, volte para a Unity e teste o funcionamento. Deixe a janela de Navigation aberta que é possível observar uma linha vermelha entre a IA e o seu ponto de destino, como mostrado na Figura **Erro! Fonte de referência não encontrada..35** – IA seguindo waypoint.

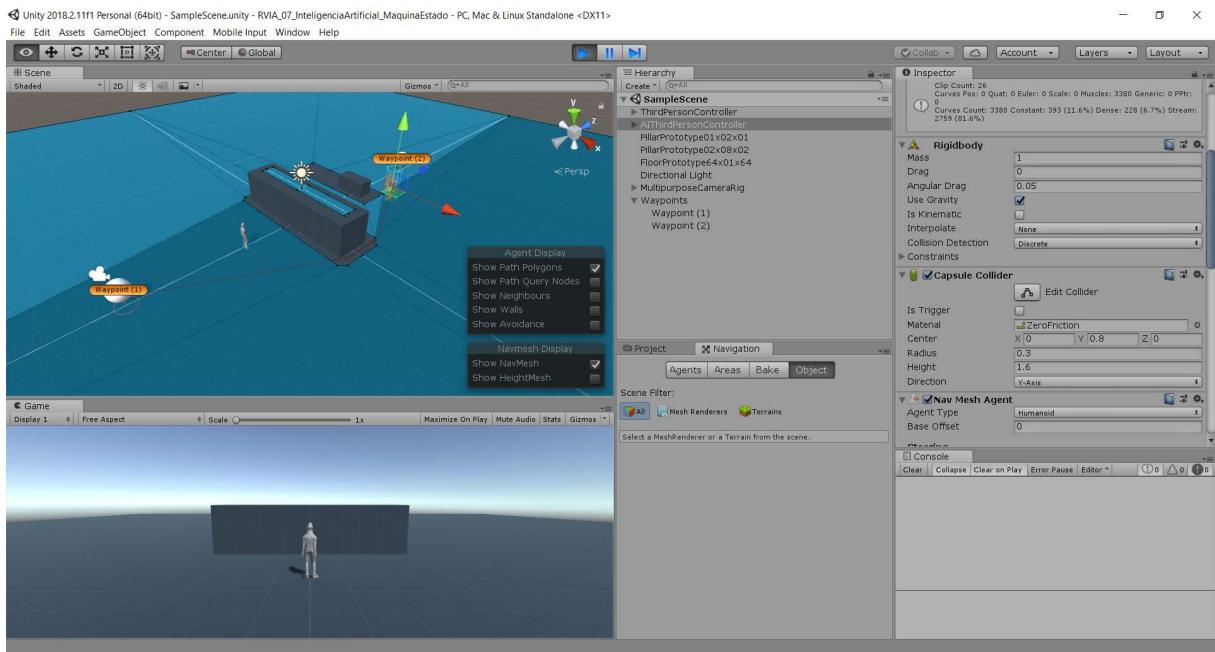


Figura **Erro! Fonte de referência não encontrada..35** – IA seguindo waypoint.  
Fonte: Unity (2018.2.11f1).

Com isso terminamos o estado de Patrulha. Precisamos agora fazer com que a IA enxergue o jogador e inicie o modo de perseguição caso ele esteja em seu campo de visão.

#### 7.7.4 PERSEGUIR

O estado de perseguição é provavelmente o mais importante dessa máquina de estados finita que estamos construindo. Ele é o responsável por procurar pelo jogador e fazer a IA seguir em sua direção. Como podemos perceber, no estado de

Esperar temos a transição para o de Patrulhar sendo chamada baseada em algumas condições atendidas.

No entanto, o estado de Perseguir não está sendo chamado em nenhum estado que programamos anteriormente. Isso ocorre, pois, esse estado não se importa com o estado que a IA estava anteriormente, portanto, sua checagem deve acontecer independente de como a IA está. Para que isso ocorra, dentro do método de checagem de estados iremos adicionar um trecho de código logo no começo.

As condições que precisamos para transicionar é se a IA possui visão do jogador, que receberemos a informação através de um novo método chamado “PossuiVisaoJogador()” e também devemos verificar se o estado atual da IA não é o estado de Perseguir, caso seja qualquer outro faremos a checagem. Se ambas forem atendidas, faremos a transição para o estado de Perseguir e encerramos a checagem de estados através do comando “return”, que encerra a execução do método.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void ChecarEstados()
    {
        if (estadAtual != Estados.Perseguir
            && PossuiVisaoJogador())
        {
            Perseguir();

            return;
        }

        switch (estadAtual)
        {
            // ...
        }
    }

    // ...

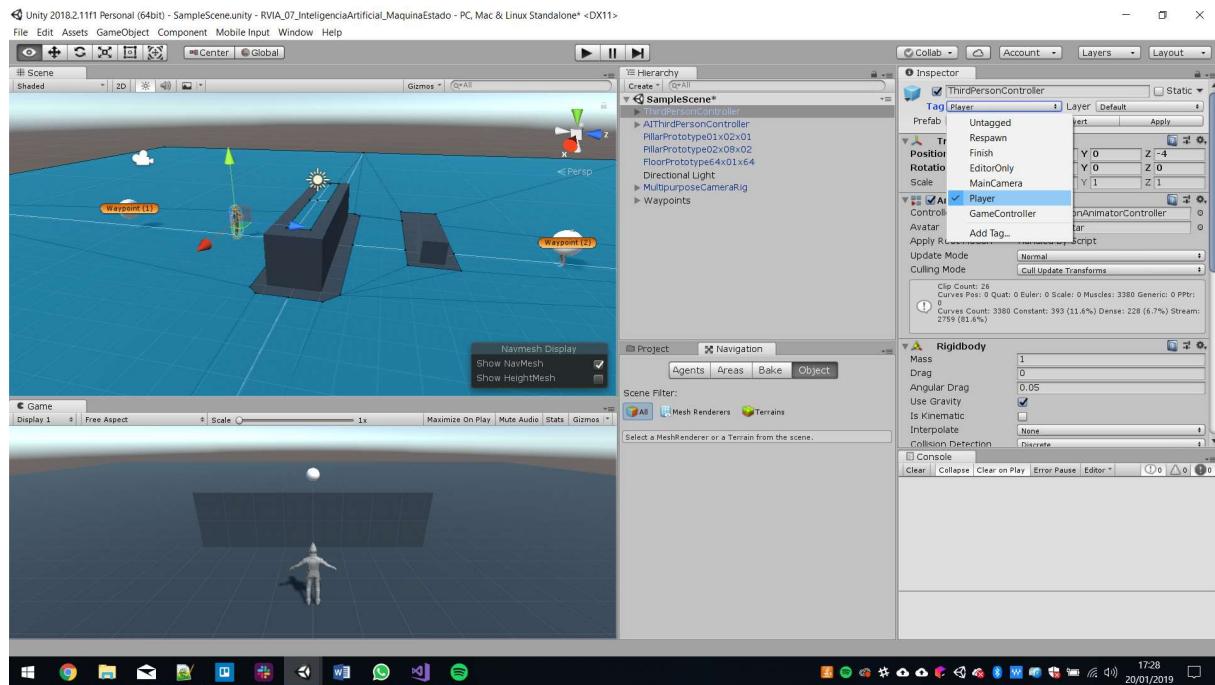
    private void Perseguir()
    {
        estadAtual = Estados.Perseguir;
    }

    // ...
}
```

```
// Métodos do Estado Perseguir

private bool PossuiVisaoJogador()
{
    return true;
}
```

Com a transição feita, podemos programar o comportamento do estado, que será feito dentro do “case Estados.Perseguir”, do switch(). O comportamento desse estado é bem simples, mas definir o alvo da IA como o jogador. No entanto, não sabemos nesse script quem é o jogador. Para simplificar, vamos definir o nosso jogador com a tag “Player” direto na Unity como mostrado na Figura **Erro! Fonte de referência não encontrada..36 – Definir tag Player no objeto do Jogador.**



**Figura Erro! Fonte de referência não encontrada..36 – Definir tag Player no objeto do Jogador.**  
Fonte: Unity (2018.2.11f1).

Agora criamos uma variável do tipo GameObject para armazenar o jogador encontrado e definimos o Transform desse objeto encontrado como target.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{

    // ...

    [Header("Estado: Perseguir")]
}
```

```

private GameObject jogador;

// ...

void Start()
{
    jogador = GameObject.FindGameObjectWithTag("Player");
    // ...
}

// ...

private void ChecarEstados()
{
    // ...
    switch (estadoAtual)
    {
        // ...
        case Estados.Perseguir:
            alvo = jogador.transform;

            break;
        // ...
    }
    // ...
}

// ...
}

```

Agora que temos as condições e a execução definida, a IA precisa saber como fazer para enxergar o jogador. O código é o mesmo utilizado no estado de Patrulhar. Porém, agora pegaremos a distância entre o jogador e a IA.

Repetiremos o processo que fizemos anteriormente, criando uma variável que armazenará a distância do jogador e uma outra variável com o campo de visão da IA, para sabermos em quais momentos a IA possui visão do jogador.

```

using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    [Header("Estado: Perseguir")]
    public float campoVisao = 5f;
    private float distanciaJogador;
    private GameObject jogador;

    // ...

    // Métodos do Estado Perseguir
}

```

```

private bool PossuiVisaoJogador()
{
    distanciaJogador = Vector3.Distance(
        transform.position,
        jogador.transform.position
    );
    return distanciaJogador <= campoVisao;
}

```

Salve o script e rode o jogo. Observe que a IA inicia seu estado de Patrulhar e assim que chega próxima ao jogador imediatamente começa a perseguir o jogador. No entanto, podemos notar alguns problemas:

- Não conseguimos nos afastar da IA, pois, a velocidade dela é a mesma que a do jogador;
- Mesmo que nos afastemos da IA, ela continuará nos perseguinto, mesmo que não tenha mais visão do jogador, afinal, não definimos uma condição para troca de estado.

Para corrigir o primeiro problema, vamos alterar alguns valores do componente “NavMesh Agent” da IA. Podemos trabalhar com os valores da categoria de “Steering”, alterando Speed para 0.5, Angular Speed para 20 e Stopping Distance para 1.5, conforme a Figura **Erro! Fonte de referência não encontrada..37** – Alteração dos valores de Steering da IA.

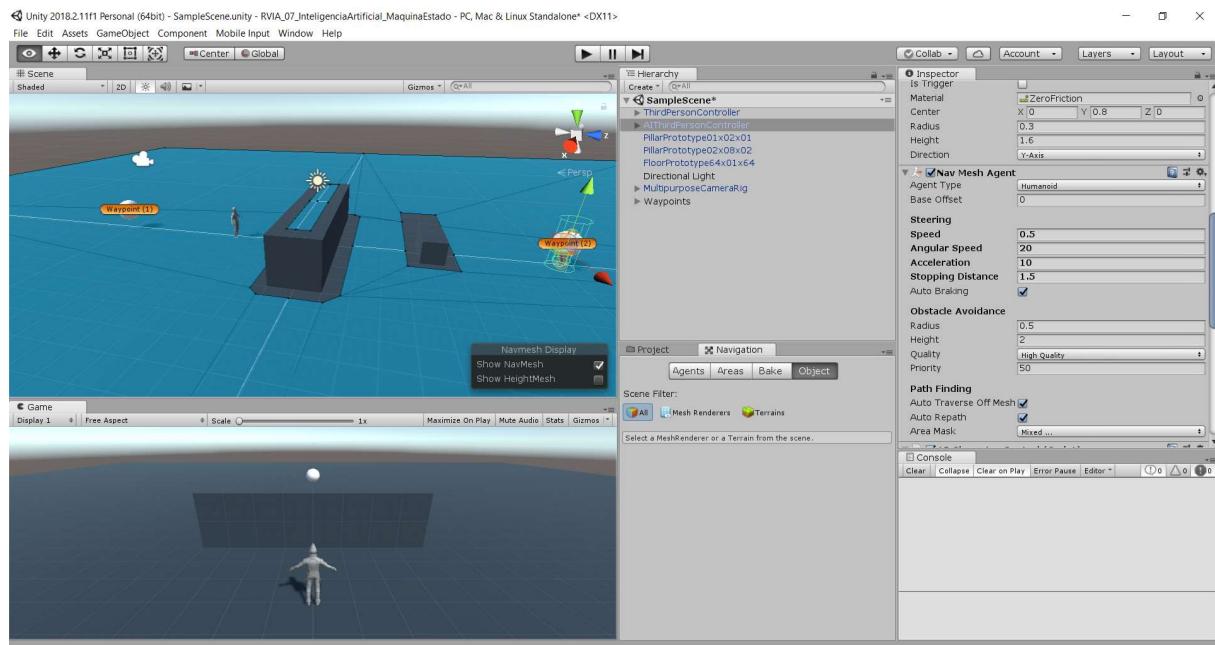


Figura Erro! Fonte de referência não encontrada..37 – Alteração dos valores de Steering da IA.  
Fonte: Unity (2018.2.11f1).

Com os valores alterados já podemos melhorar nosso fluxo de testes, afinal, a IA não está mais se movimentando na mesma velocidade do jogador. Para resolver o segundo problema precisamos definir a transição para o estado de Procurar, que faremos a seguir.

### 7.7.5 PROCURAR

O estado de Procurar é acionado quando a IA não possui mais visão do jogador mas ainda irá se movimentar para última posição que conhece dele, podendo encontrá-lo novamente. Caso a IA tenha procurado tempo suficiente, ela irá voltar para um estado de Esperar, que consequentemente leva para o estado de Patrulhar quando a IA esperou por um tempo suficiente.

Para iniciar o desenvolvimento desse estado, iremos construir a sua transição vinda do estado de Perseguir, que desenvolvemos anteriormente. Dentro da checagem do estado vamos adicionar uma condição utilizando o método que criamos “PossuiVisaoJogador()”, porém, nesse caso queremos saber se a IA não possui visão do jogador. Como esse método retorna true em caso de visão positiva, queremos que o if seja executado caso a visão seja negativa, portanto, utilizaremos o ponto de exclamação para inverter o sinal recebido.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void ChecarEstados()
    {
        // ...
        switch (estadoAtual)
        {
            // ...
            case Estados.Perseguir:
                if (!PossuiVisaoJogador())
                {
                    Procurar();
                }
                else
                {
                    alvo = jogador.transform;
                }
        }
    }
}
```

```

        }

        break;
    // ...
}

// ...

private void Procurar()
{
    estadoAtual = Estados.Procurar;
}

// ...

}

```

Com a transição feita, podemos iniciar a construção do comportamento do estado de Procurar. Se analisarmos como a IA irá se comportar durante esse estado, chegamos à conclusão de que ela ficará um determinado tempo procurando pelo jogador antes de mudar seu estado. Para isso, criaremos uma variável que chamaremos de “tempoPersistencia”. Além disso, assim que a IA entra no estado de Procurar, devemos guardar o valor de “Time.time” do início, para conseguir construir algo semelhante ao que foi feito durante o estado de Esperar, portanto, criaremos uma variável para isso que chamaremos de “tempoSemVisao”.

```

using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    [Header("Estado: Procurar")]
    public float tempoPersistencia = 6f;
    private float tempoSemVisao;

    // ...

}

```

A variável “tempoSemVisao” precisa ter seu valor atualizado assim que a IA entrar no estado de Procurar, por tanto, adicionaremos a declaração no método que realiza tal ação.

```

using UnityEngine;

```

```

using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void Procurar()
    {
        estadoAtual = Estados.Procurar;
        tempoSemVisao = Time.time;
    }

    // ...
}

```

Agora podemos iniciar a construção do comportamento do estado de Procurar. Assim que o estado iniciar devemos remover o valor da variável alvo, afinal, não temos mais um Transform para perseguir. Podemos apagar a informação armazenada na variável “alvo”, simplesmente atribuindo o valor null.

```

using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void Procurar()
    {
        estadoAtual = Estados.Procurar;
        tempoSemVisao = Time.time;
        alvo = null;
    }

    // ...
}

```

Salve o código, volte para a Unity e teste. Note que assim que a IA encontra o jogador ela passa a persegui-lo normalmente, quando afastamos o jogador da IA ela continua andando até a última posição conhecida. Caso ela encontre o jogador novamente, ela volta para o estado de perseguição.

Entretanto, mesmo que a IA fique muito tempo parada, ela não volta para o estado de Esperar/Patrulhar. Isso ocorre pois precisamos configurar qual é o momento

que ela deve tomar a decisão de voltar. Para isso, vamos configurar na checagem de estados.

Criaremos um método chamado “SemVisaoTempoSuficiente()” que irá retornar uma informação booleana que verifica quanto tempo a IA está sem visão e se ela pode mudar para o estado de Esperar ou não.

```
using UnityEngine;
using UnityStandardAssets.Characters.ThirdPerson;

public class ControladorIA : MonoBehaviour
{
    // ...

    private void ChecarEstados()
    {
        // ...
        switch (estadoAtual)
        {
            // ...
            case Estados.Procurar:
                if (SemVisaoTempoSuficiente())
                {
                    Esperar();
                }
                break;
            }
            // ...
        }
        // ...

        // Métodos do Estado Procurar

        private bool SemVisaoTempoSuficiente()
        {
            return true;
        }
    }
}
```

A lógica do método “SemVisaoTempoSuficiente()” é a mesma que fizemos para o método “EsperouTempoSuficiente()”, só que dessa vez utilizaremos as variáveis “tempoSemVisao” e “tempoPersistencia” no lugar.

```
private bool SemVisaoTempoSuficiente()
{
    return tempoSemVisao + tempoPersistencia <= Time.time;
}
```

Pronto! Terminamos o script! Salve, vá para a Unity e rode. Note que assim que a IA perde visão do Jogador, ela vai até a última posição conhecida, aguardar por um tempo, inicia o estado de Espera e depois de mais alguns segundos retorna para a Patrulha.

Com isso finalizamos nossa máquina de estado. No entanto algumas melhorias ainda são possíveis em um mundo infinito de possibilidades. Experimente planejar sua própria máquina de estado e desenvolver com os conhecimentos adquiridos nessa aula!

## 8 REFERÊNCIAS

Execution Order of Event Functions, Data do conteúdo desconhecida  
<<https://docs.unity3d.com/Manual/ExecutionOrder.html>> Acessado em 14 de Janeiro de 2019