

Curso de Pós-Graduação em Cloud Computing e Mobile

Disciplina DM 117 – Introdução a Desenvolvimento de Jogos com Unity

Professor: Phyllipe Lima

Aula 1

Objetivos:

Ao final desse relatório você terá:

- Se familiarizado com a interface do Unity
- Entender a organização básica de um projeto Unity
- Criado o primeiro projeto
- Criado um jogador (*player*)
- Criado um tile básico
- Ter feito a câmera seguir o jogador
- Usado dos *attributes* do C#
- Fazer o jogo ficar infinito
- Criado obstáculos

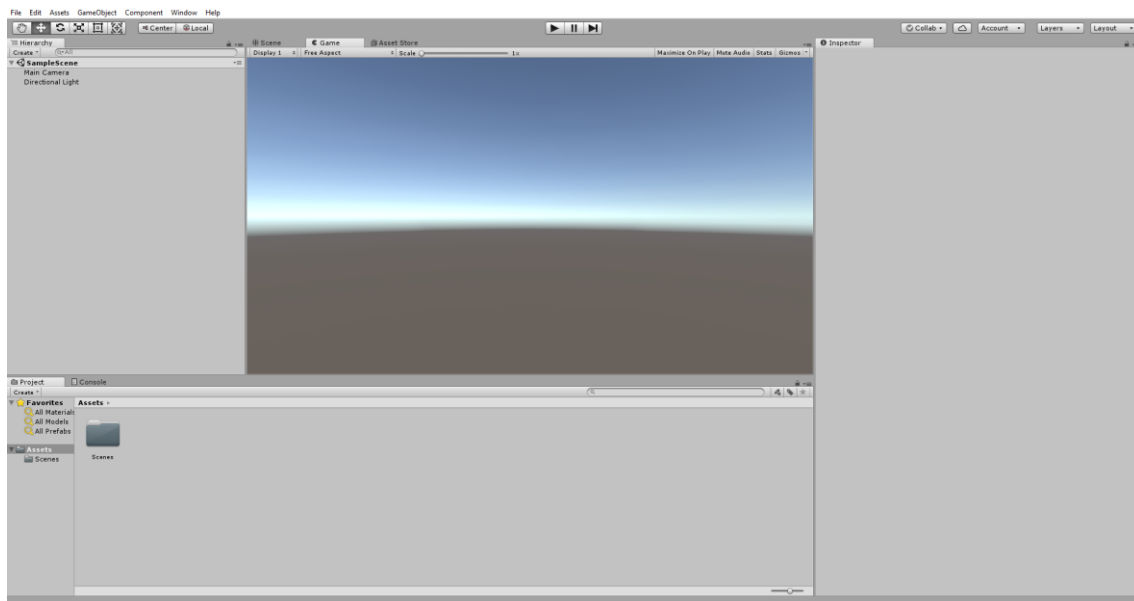
1 - Criando o projeto:

- Crie um projeto 3D
- Escolha um nome (eu coloquei DM117-1)
- Desligue a opção “Enable Unity Analytics” (mais adiante usaremos isso)

The screenshot shows the Unity Hub interface for creating a new project. At the top, there are links for 'Projects' and 'Learn'. On the right, there are icons for 'New', 'Open', and 'My Account'. The main area contains the following fields and controls:

- Project name:** A text input field containing 'DM117-1'.
- Template:** A dropdown menu set to '3D'.
- Location:** A text input field containing 'C:\Menethil\UnityGames\DM117-Aula' with a three-dot menu icon to its right.
- Add Asset Package:** A button next to the location field.
- Enable Unity Analytics:** A toggle switch currently set to 'OFF'.
- Create project:** A blue button at the bottom right.

Depois de criado o projeto o Editor do Unity irá aparecer conforme Figura 2.



Esse é o chamado *Layout* padrão do Unity. Se o seu estiver diferente vá no canto direito superior na aba Layout e clique em Default conforme mostra a Figura 3.

Se for sua primeira vez no Unity gaste algum tempo observando a interface e se familiarizando. Ao longo do curso iremos entender melhor cada uma delas

Quero chamar atenção para as seguintes abas:

Hierarchy: É a tela que apresenta a relação hierárquica entre os game objects presentes na aba Scene.

Scene: É onde construímos o jogo. Essa aba permite a manipulação dos game objects, como movimentar, rotacionar, escalar e etc.

Game: Essa aba é o que o jogador irá ver quando estiver jogando.

Inspector: Nessa aba podemos inspecionar cada game objects, observando os componentes que o compõe. Podemos também manipular esses componentes. Todo game object é composto de componentes, e todos eles possuem no mínimo um componente chamado Transform que define a sua posição no game world (mundo do jogo).

Project: Nessa aba vemos todos os assets (podemos pensar como arquivos) de todo o projeto, mesmo os que não estão sendo utilizados. Tudo que iremos importar para dentro do Unity (como sprites, textures, normal maps) podemos simplesmente arrastar para essa aba.

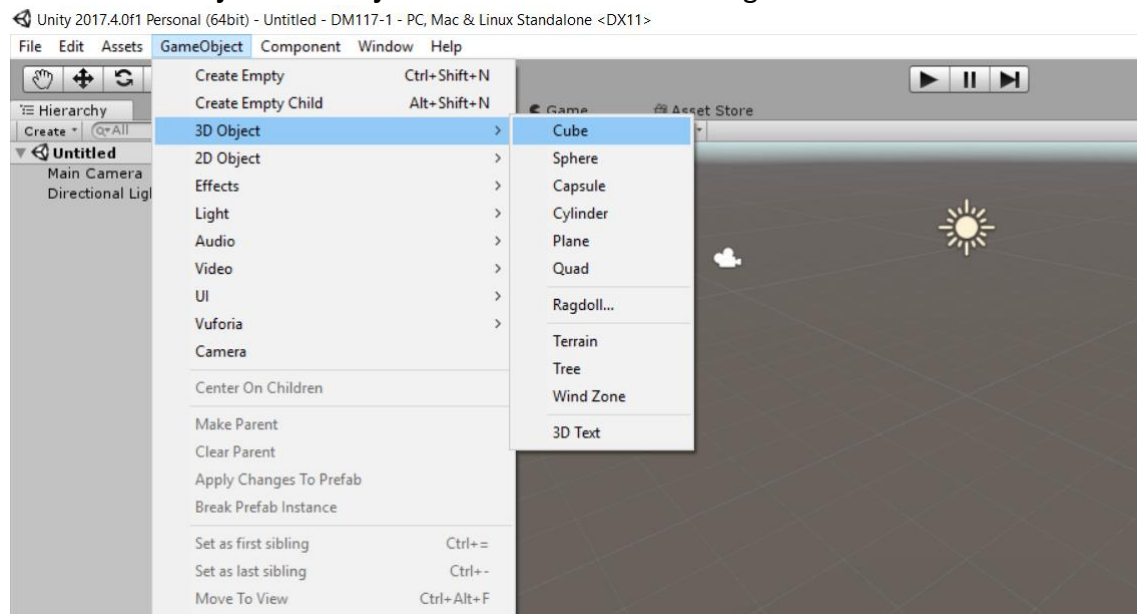
2 - Criando o jogador:

O jogo criado será do tipo *endless runner*. Onde teremos um jogador em formato de esfera rolando infinitamente. O primeiro passo é criar um terreno ou chão para nosso jogador.

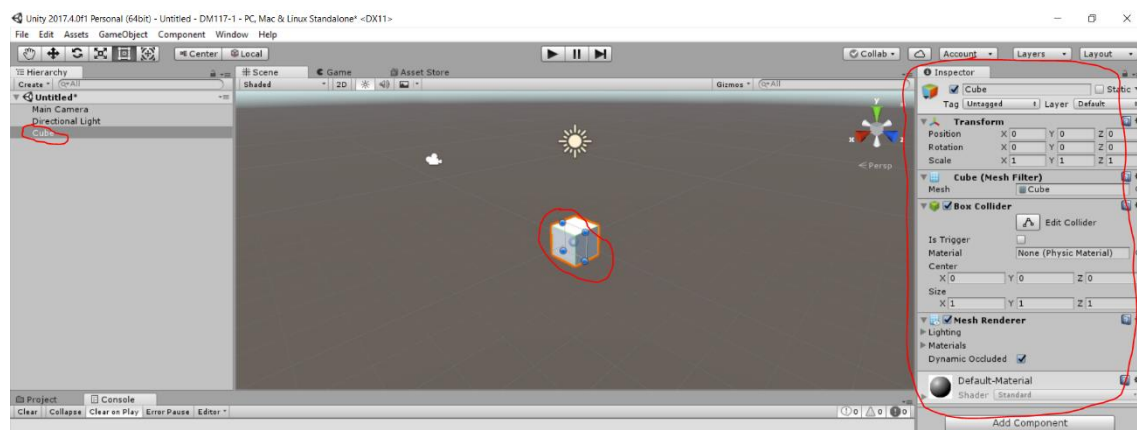
2.1 – Criando o chão

Usaremos um cubo para ser o chão.

Vá em **GameObject->3D Object->Cube** como mostra a figura abaixo

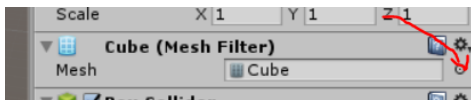


Observe o cubo que foi criado na Figura abaixo:

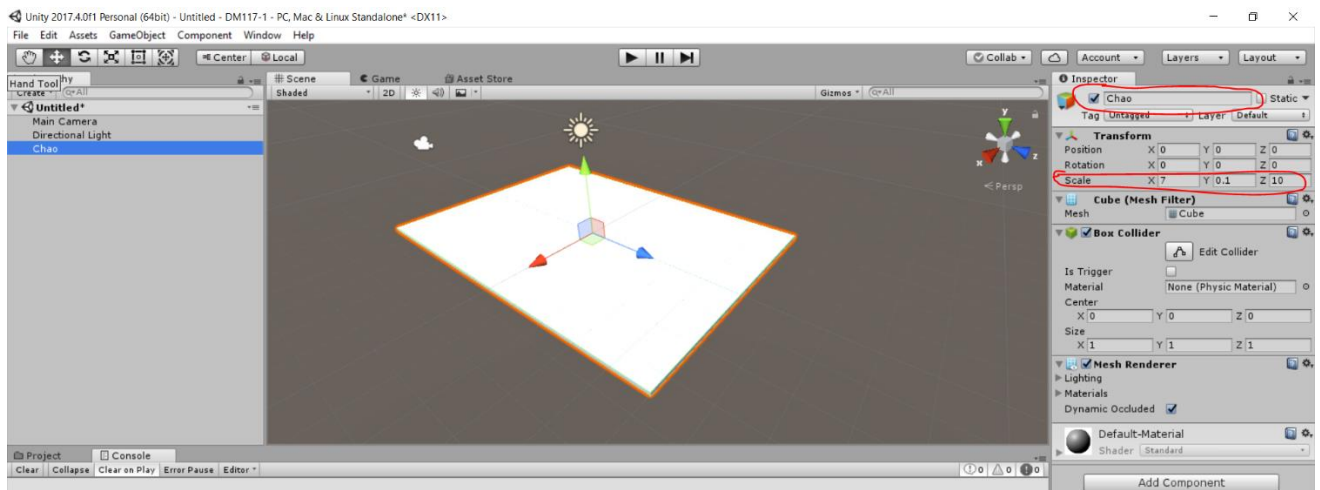


Olhe primeiro a aba Scene. Ela mostra o Cubo que acabou de ser criado. Veja agora a aba Hierarchy. Mostra que o Cubo não possui game objects filho. Agora observe a aba Inspector, mais à direita. Ela está mostrando todos os componentes que fazem parte deste Cubo.

- Transform: Diz aonde no game world esse cubo se encontra. Vemos que está na origem (0,0,0), possui rotação zero em todos os eixos e escala valendo 1, ou seja tamanho original.
- Cube (Mesh Filter): Lembre-se da aula “Gráficos”. Todo jogo 3D começa com o modelo 3D, chamado também de Polygon Mesh. Esse componente permite indicar qual o mesh para esse game object 3D. O Unity já possui alguns modelos já criado, como o “Cube”. Se você clicar no ponto ao final do campo “Mesh” (Embaixo da engrenagem, como mostra a figura abaixo), você poderá ver outros tipos de mesh que o Unity possui, ou mesmo selecionar os seus próprios modelos.
- Box Collider: Responsável por detectar colisão. Logo mais trataremos isso
- Mesh Renderer: Lembrando mais uma vez da aula sobre Gráficos. O modelo 3D precisa de um Material, onde podemos colocar informações como Normal Maps, Texturas e iluminação. Esse componente trata sobre isso.

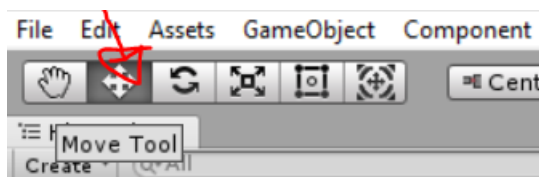


Na aba Inspector mude o nome para “Chao”. Agora vá no componente Transform e altere a escala para (7, 0.1, 10). Observe na Figura abaixo o resultado.



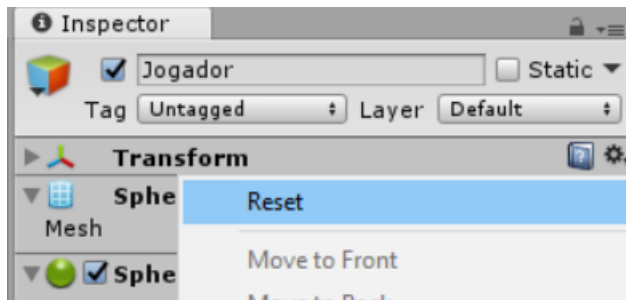
No padrão do Unity, 1 unidade de espaço significa 1 metro no mundo real. O que temos então é no eixo X (vermelho) e Z (azul) temos um chão mais largo. E no eixo Y (de verde) colocamos uma escala de 0,1 para reduzirmos a altura.

Se você não consegue ver os eixos em cima do game object (como na Figura) é porque você não está no modo “Movo”. Para isso clique no game object e pressione a tecla “W”. Veja a figura abaixo. Observe o desenho indicando o modo “Move Tool”.

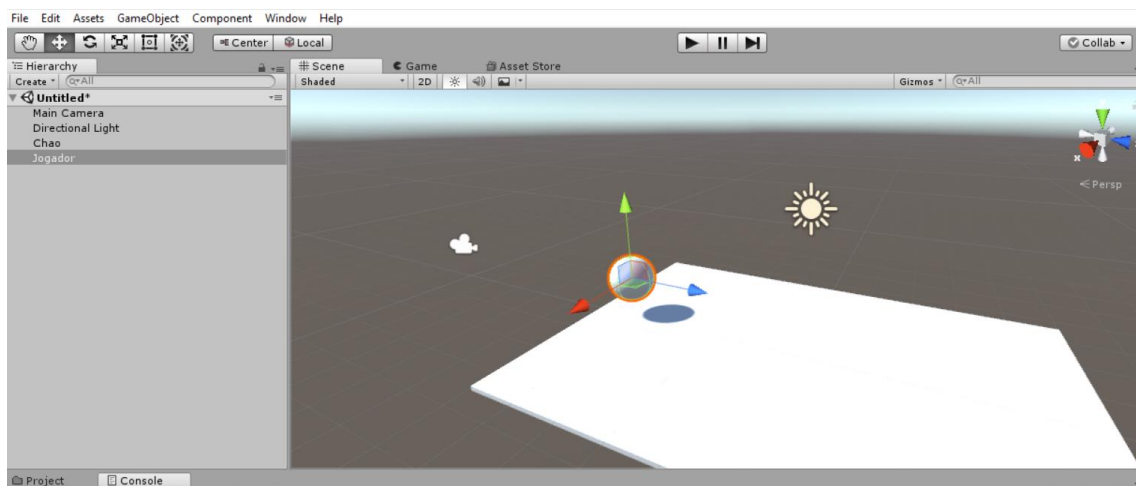


Vamos agora criar o jogador. Vá em **GameObject->3D Object->Sphere**
Altere a posição para (0,1,-4)

#DICA. Se você deseja resetar a posição de um GO (GameObject) clique na engrenagem no componente Transform e aperte Reset como na Figura abaixo

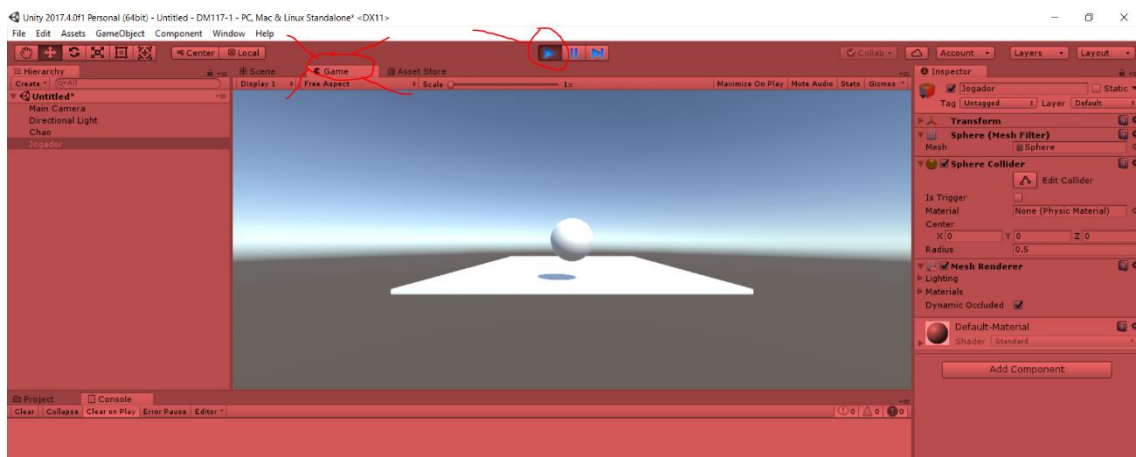


A Figura abaixo apresenta o estado atual do jogo.

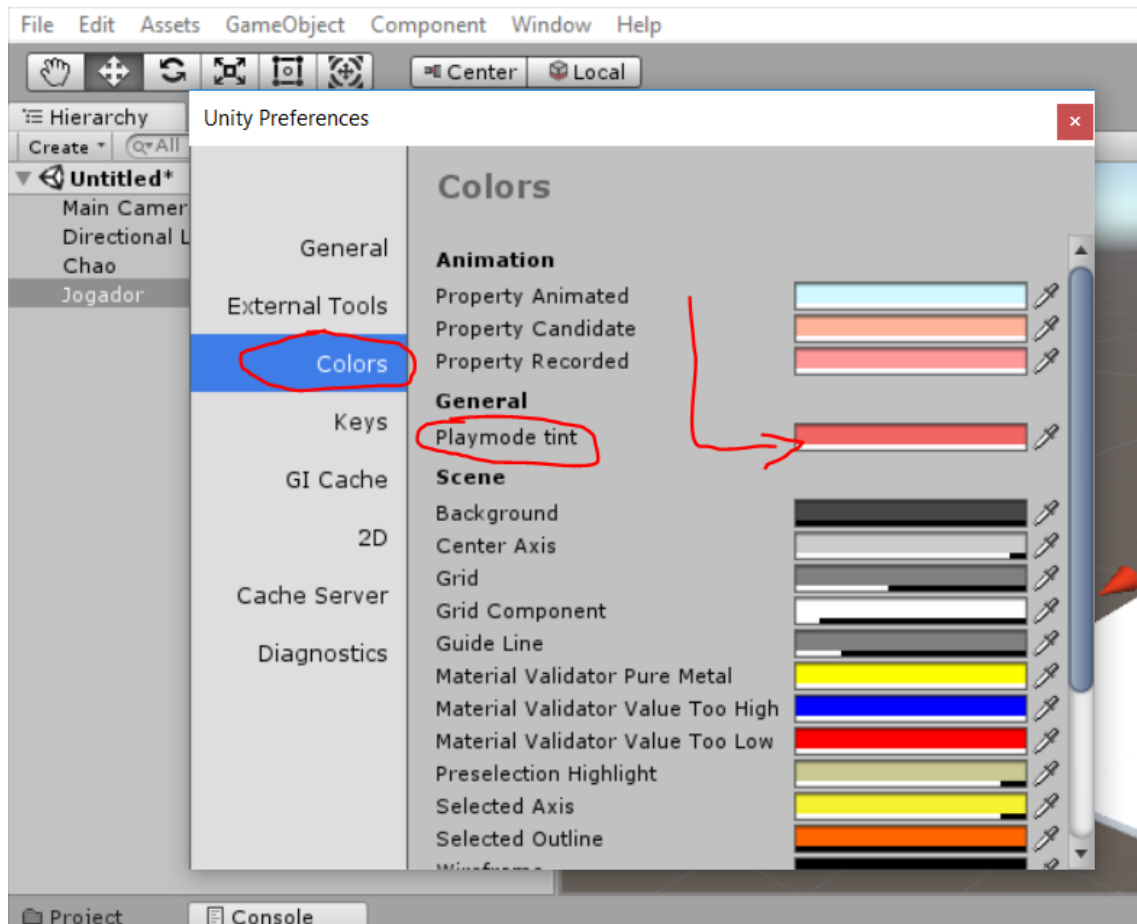


Como colocamos o valor 1 na coordenada Y, o jogador ficará um pouco acima do chão.

Vamos fazer um teste. Aperte o botão play e veja o que ocorre, conforme mostra a Figura abaixo.



#DICA2: Você deve estar pensando, “Ó Meu Deus, por que o seu Unity está vermelho e o meu não?”. É muito simples. Eu ajustei para ficar vermelho para eu estar ciente de que estamos no “Play Mode”. Esse modo permite alterarmos dados nos componentes dos GOs para testarmos, porém, quando saímos desse modo, os valores são perdidos. Então eu coloquei em vermelho para ressaltar o modo. Para fazer isso basta ir em Edit->Preferences clique em Colors e depois altere a cor na opção Playmode tint, como mostra a figura abaixo:

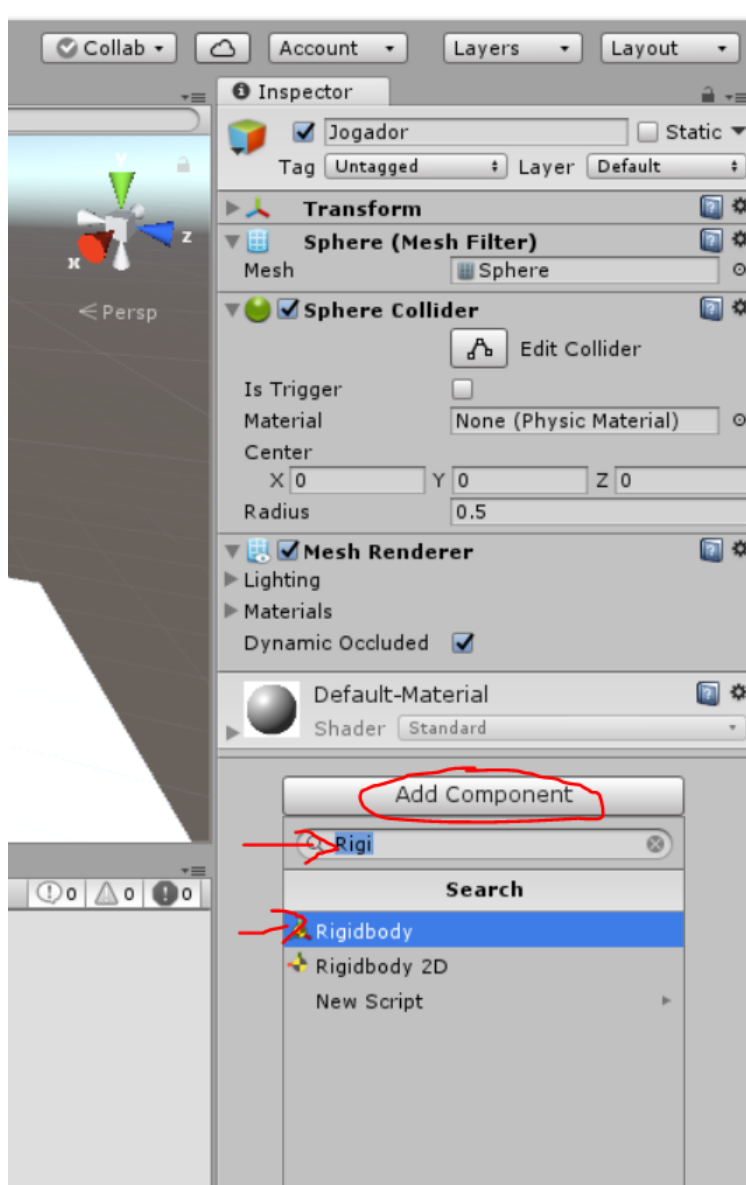


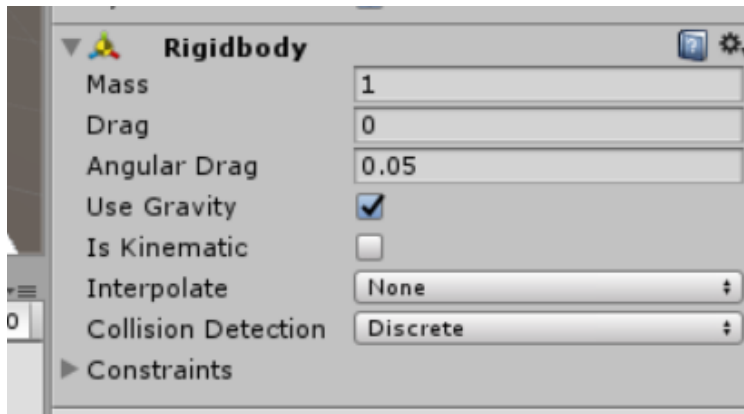
Voltando ao Play Mode. Automaticamente a aba Game foi selecionada, e o que você vê é exatamente o que o jogador veria se fizesse o deploy desse jogo em algum dispositivo. Nada acontece, pois ainda não fizemos nada para que algo acontecesse.

Aperte o botão Play novamente para sair do Play Mode. E vamos dar sequência ao nosso jogo.

Queremos mover o jogador(bola). Iremos fazer isso através da física que o Unity já possui. O que queremos é que a bola saia rolando. Então, como na vida real, iremos aplicar uma força nessa bola. Mas para aplicar força o Unity precisa de saber que este GO estará sujeito as leis da física, e a forma de fazer isso é transforma-lo em um corpo rígido.

Já sabemos que todo GO é formado por componentes, então não é surpresa que iremos adicionar um componente chamado “RigidBody” neste GO. Com o “Jogador” selecionado, na aba Inspector clique em Add Component, comece a escrever RigidBody e clique em RigidBody. É muito importante não clicar em RigidBody 2D (não se preocupe que irá dar erro se clicar), pois este possui física para objetos 2D, que não é o caso do nosso jogador. Observe as duas Figuras abaixo





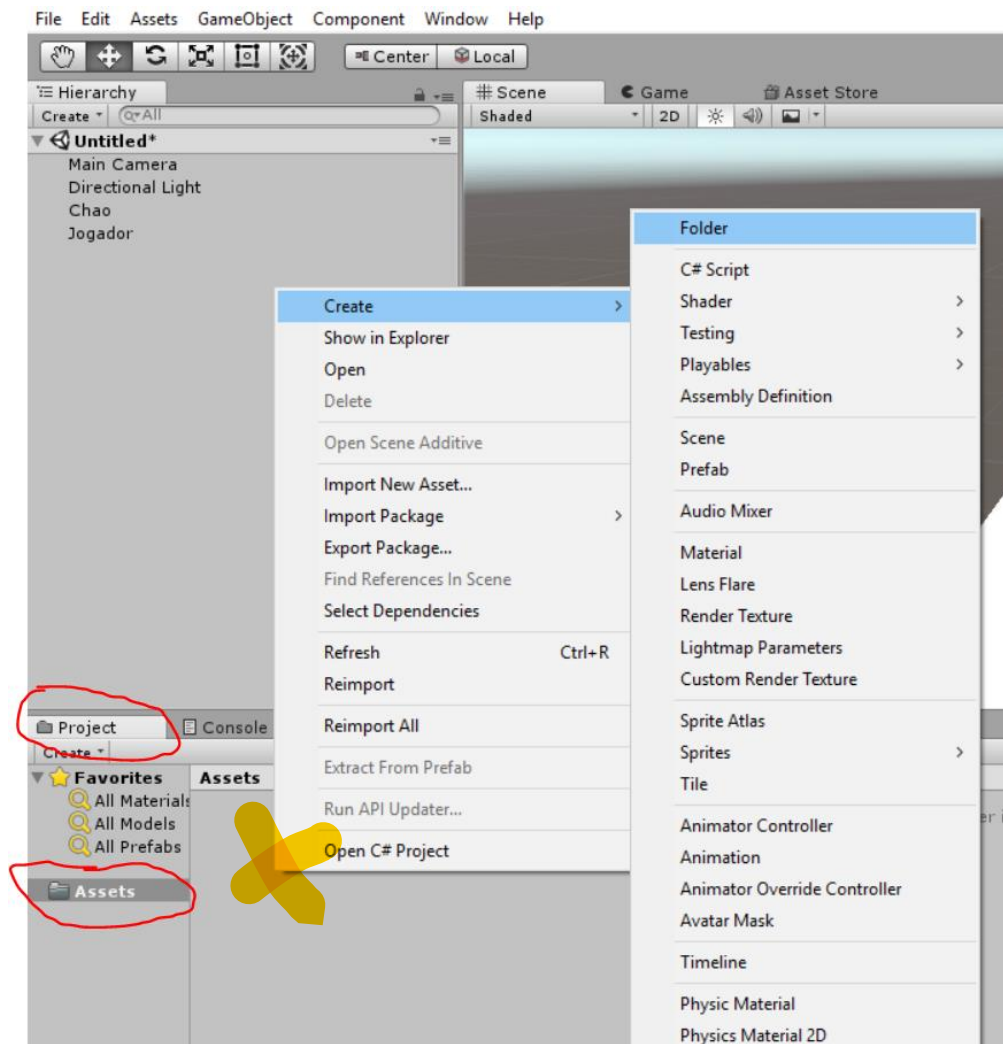
A primeira Figura apresenta o processo de adicionar o Rigidbody. A segunda já mostra o Rigidbody como componente do “Jogador”. O componente Rigidbody permite manipular como a física irá influenciar esse GO. Ao longo do curso iremos entender melhor como usamos o Rigidbody. Agora clique em Jogar e veja o que ocorre.

Se tudo deu certo, você viu a bola caindo e parando no chão.

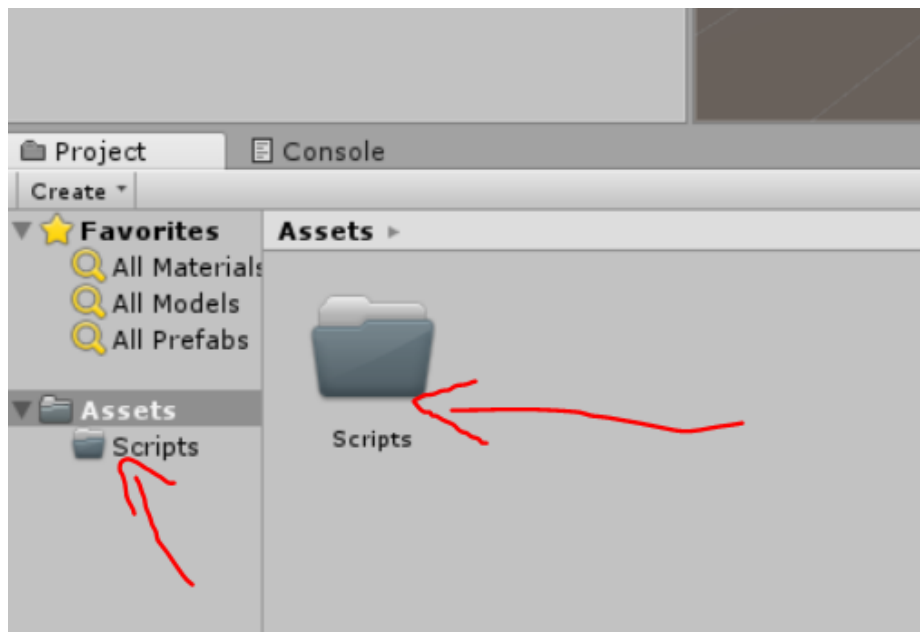
- 1-) Por que a bola caiu?
- 2-) Por que parou no chão?
- 3-) Por que o chão não caiu?

- 1-) Porque agora a bola possui um Rigidbody, ou seja, está sujeita às leis da física
- 2-) Porque tanto a bola, quanto o chão possuem um componente Collider, que identifica colisões. O Rigidbody fez a bola cair, mas ela encontrou o chão que possui um Collider.
- 3-) Porque o chão não possui um Rigidbody

Agora vamos fazer a bola se movimentar. Iremos criar nosso primeiro script. Mas antes vamos criar uma pasta Script dentro de Assets. Fazemos isso pela aba Project, como mostra a Figura abaixo.



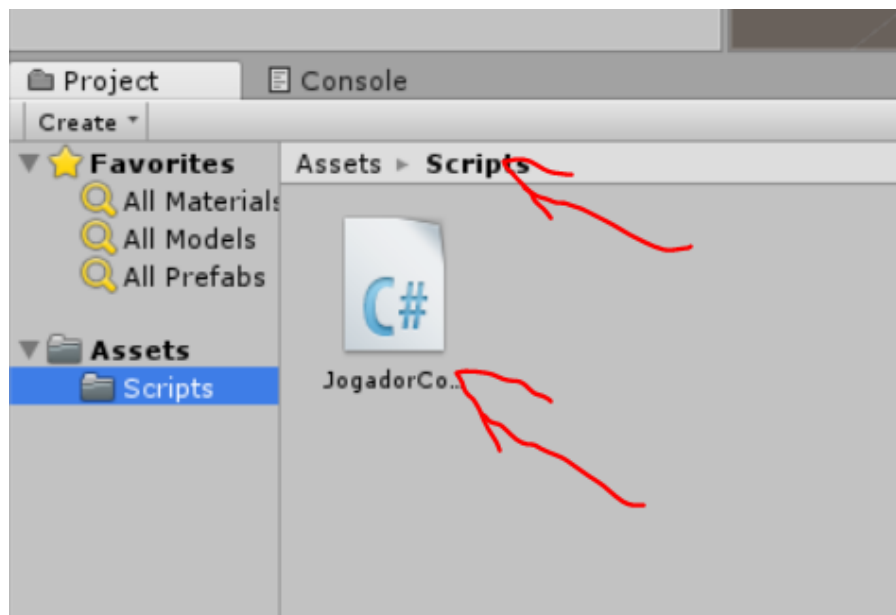
Na aba Project, clique com o botão direito onde temos um X na Figura. Estamos dentro da pasta Assets. A pasta Assets não pode ser deletada e é criada automaticamente pelo Unity. Crie dentro uma pasta chamada Scripts. Todo os scripts serão colocados dentro desta pasta. A Figura abaixo mostra o resultado.



Agora dentro da pasta Script crie um C# Script. Botão Direito->Create->C# Script

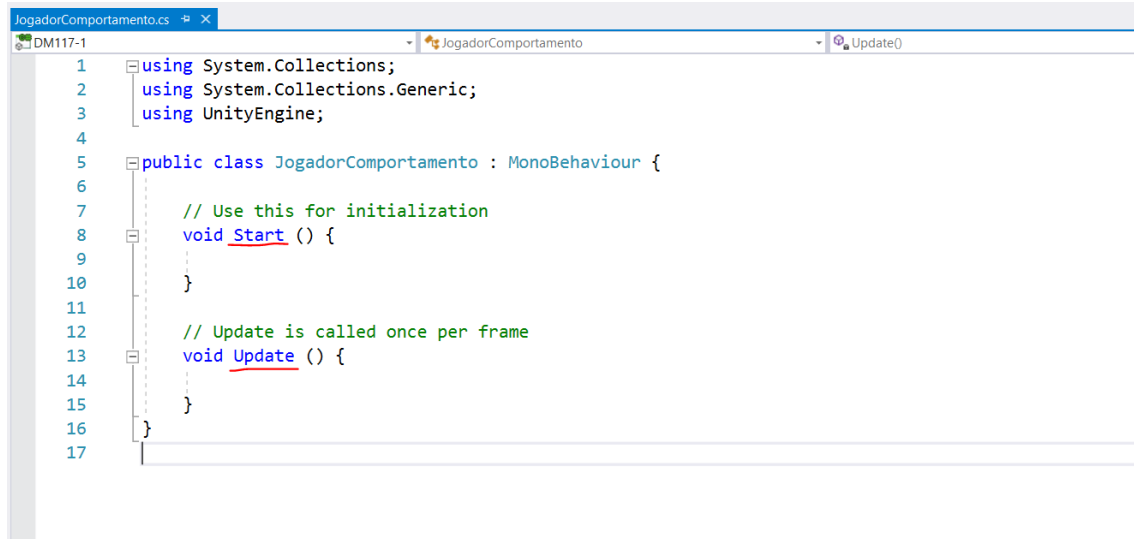
#DICA: “Ah, eu gosto de programar em JavaScript e fiquei sabendo que o Unity dá suporte. Portanto quero usar JavaScript”. O suporte será encerrado na Versão 2018, tanto é que nem existe mais a opção de criar script em JS na 2017. Vida que segue. Vamos de C#.

Chame o Script de “JogadorComportamento”, afinal esse script irá controlar o comportamento da bola, que é o nosso jogador. A aba Project ficará conforme figura abaixo.



De um clique-duplo nesse arquivo e aguarde o Visual Studio 2017 abrir. A primeira vez pode demorar um pouco. Caso o MonoDevelop abra também, somente ignorar e fechar. O VS pode pedir para você fazer login, caso não queira só clicar no “Agora não, talvez mais tarde”.

Observe na Figura abaixo um Script C# recém-criado. Perceba os métodos Update() e Start(). Na aula de introdução ao desenvolvimento de jogos eu comentei que uma game engine é vista como um framework, e este por sua vez que controla a aplicação. Este método Start e Update deixa isso muito explícito.



```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class JogadorComportamento : MonoBehaviour {
6
7     // Use this for initialization
8     void Start () {
9
10    }
11
12    // Update is called once per frame
13    void Update () {
14
15    }
16 }
17
```

Quando você roda uma aplicação feita com o Unity, é o Unity que chama o método Start (para inicializar o GO) e o Update (para ficar constantemente atualizando o GO), como visto em sala, isso faz parte do Game Loop.

Vamos agora escrever algumas variáveis no nosso código. Veja a próxima figura.

```

7      //Uma referencia para o componente Rigidbody
8      private Rigidbody rb;
9
10     //A velocidade que a bola irá esquivar para os lados
11     public float velocidadeEsquiva = 5.0f;
12
13     //Velocidade com qual a bola irá se deslocar para a frente
14     public float velocidadeRolamento = 5.0f;
15
16     // Use this for initialization
17     void Start () {
18         //Obter acesso ao componente Rigidbody associado a esse GO
19         rb = GetComponent<Rigidbody>();
20     }
21
22     // Update is called once per frame
23     void Update () {
24         //Verificar para qual lado o jogador deseja esquivar
25         var velocidadeHorizontal
26             = Input.GetAxis("Horizontal") * velocidadeEsquiva;
27         //Aplicar uma força para que a bola se desloque
28         rb.AddForce(velocidadeHorizontal, 0, velocidadeRolamento);
29     }
30 }

```

Vamos entender o que foi feito.

Na linha 8, declaramos uma variável privada do tipo Rigidbody. É através dela que iremos acessar o componente Rigidbody associado a este componente.

Na linha 11 e 14 declaramos duas variáveis públicas que irão controlar a velocidade da bola.

Na linha 19, dentro do método Start() usamos o método GetComponent() para obter acesso ao Rigidbody deste GO, no caso a bola (jogador). Isso faz sentido, pois o método Start() é chamado uma única vez quando o GO é instanciado. E só precisamos pegar essa referência uma vez.

Na linha 25, dentro do método Update() verificamos para qual lado o jogador está querendo se deslocar na horizontal. A classe Input possui um método GetAxis() que retorna um valor float entre -1 e 1. Se nenhuma tecla está sendo pressionada, este método retorna 0. Se pressionamos a tecla "A" isso indica que desejamos fazer o movimento para a esquerda, e esta função começa a retornar um valor negativo até parar em -1. Se soltarmos a tecla "A" o valor de retorno da função decai até chegar em 0. E se pressionarmos a tecla "D" o valor de retorno cresce até parar em 1.

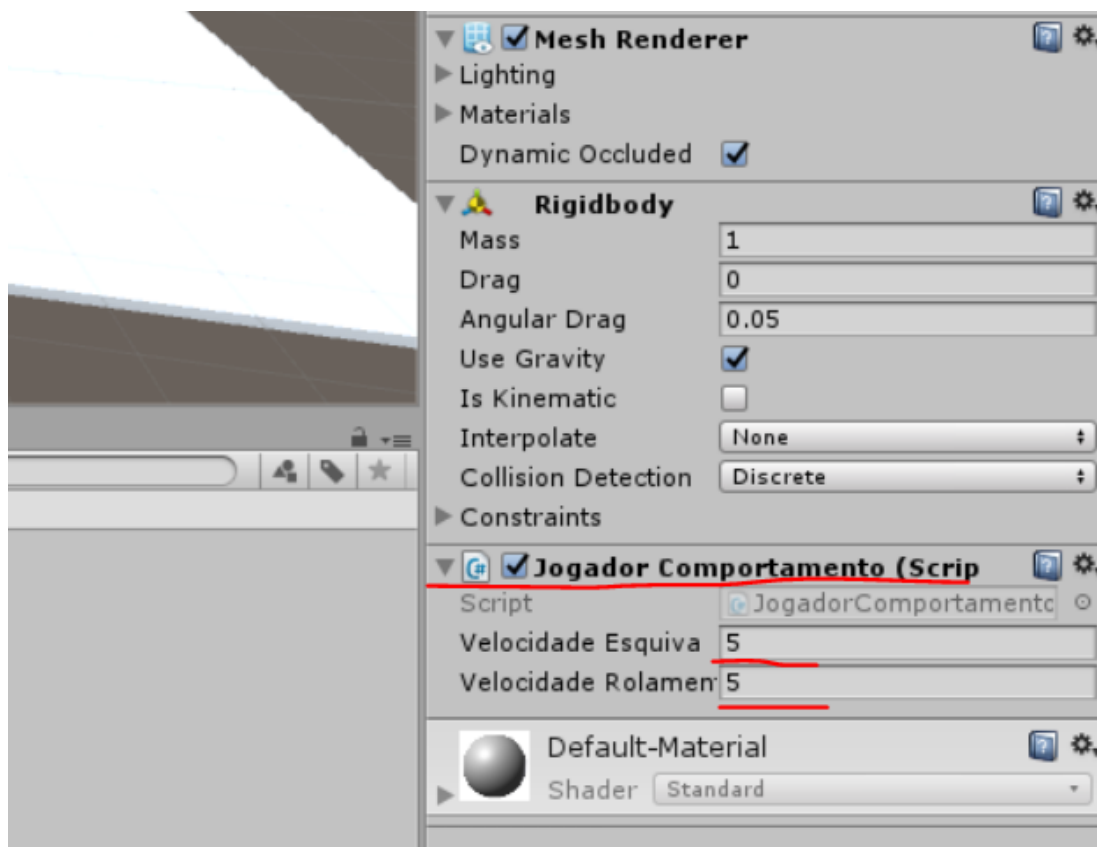
#DICA: Se quiser ver valores impressos no console use a função print().

Na linha 28 iremos aplicar uma força na bola. Para fazermos isso precisamos usar a variável *rb* (ela representa o Rigidbody). Usando o método `AddForce()`, passamos três parâmetros, representando os eixos X,Y,Z.

No eixo Y, aplicamos uma força zero. Faz sentido pois não desejamos que bola suba. No eixo X (deslocamento horizontal) e no eixo Z passamos as variáveis que representam a força nestes eixos.

Salve o Script, volte para o Unity e aperte “Play”. O que acontece? Aperte a tecla A ou D e veja. Nada ainda certo? Que bom. Seria um milagre se acontecesse algo. Nós apenas criamos o script, não fizemos nenhuma associação deste script com a bola.

Lembre-se que o GO é composto por componentes, portanto um script também será um componente. Com a bola selecionada, vá na aba Inspector. Clique em Add Component e comece a digitar o nome do script. Assim que achar clique no script. Outra forma de adicionar é arrastando o arquivo script para a aba Inspector com a bola selecionada. De qualquer forma o resultado é mostrado na próxima Figura.



Observe as variáveis que foram declaradas como públicas. Elas aparecem no Editor do Unity e permite que você as manipule durante a execução do jogo. Essa funcionalidade visa ajudar também Artistas e Designer a fazerem testes no jogo sem a necessidade de irem no código fonte.

Agora sim. Aperte “Play” e comece a brincar. Aperte “A” e “D” e se divirta. Pena que dura pouco, logo a bola cai. Iremos arrumar isso mais adiante, mas antes algumas dicas para organizar o código.

3 – Usando comentário XML e Attributes

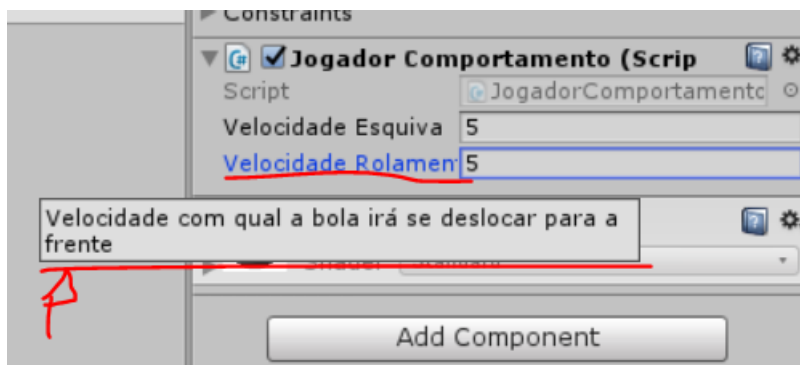
É sempre boa ideia comentar código fonte. Vamos ver aqui como a própria equipe dentro do Unity comenta seus códigos.

3.1 – Attributes

Semelhante ao Annotations que o JAVA dispõe, C# usa os Attributes para configurar metadados em elementos de programação. Vamos começar com o [Tooltip]. Veja a figura abaixo.

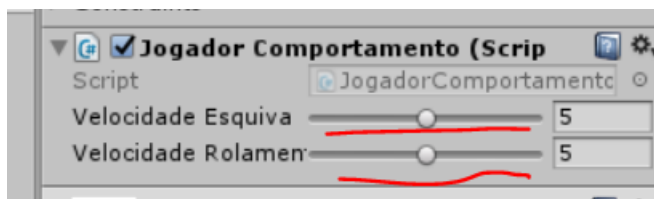
```
[Tooltip("A velocidade que a bola irá esquivar para os lados")]  
public float velocidadeEsquiva = 5.0f;  
  
[Tooltip("Velocidade com qual a bola irá se deslocar para a frente")]  
public float velocidadeRolamento = 5.0f;
```

Esse attribute faz com tooltips apareçam no editor do Unity, nessas variáveis. Isso pode auxiliar bastante outros membros da equipe que não sabem programar. Salve o script, retorne ao Unity e veja como fica quando você aproxima o mouse da variável.



Podemos facilitar ainda mais nossos colegas de trabalho, e minimizar a chance de problemas. Da forma como está, alguém pode inserir algum valor negativo nestes dois campos. Podemos usar o attribute [Range] para melhorar essa situação.

```
[Tooltip("A velocidade que a bola irá esquivar para os lados")]  
[Range(0,10)]  
public float velocidadeEsquiva = 5.0f;
```



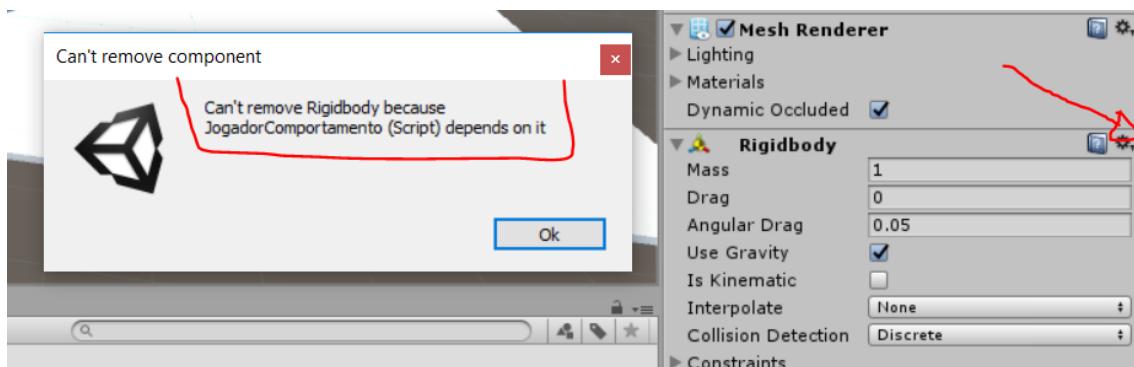
Depois que adicionamos o attribute Range, no Unity agora temos uma forma de escolhe apenas valores entre 0 e 10 através de um botão slide.

Nesse script, estamos fazendo acesso ao componente Rigidbody. O que aconteceria se alguém por acidente removesse esse componente? Teríamos uma série de NullPointers aparecendo. O [RequireComponent] adiciona uma camada de proteção.

```
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Rigidbody))]
public class JogadorComportamento : MonoBehaviour {

    //Uma referencia para o componente Rigidbody
    private Rigidbody rb;
```



Ao adicionarmos o [RequireComponent] especificando Rigidbody garantimos que o componente Rigidbody não poderá ser removido. E se o script for criado primeiro, o componente Rigidbody será adicionado automaticamente.

Para fechar essa parte de organização vamos falar sobre comentários XML. Usamos esses comentários para orientar outros programadores. Ao comentar uma variável usando XML, tooltips com o comentário irá aparecer ao longo do código. Observe a Figura abaixo.

```

/// <summary>
/// Uma referencia para o componente Rigidbody
/// </summary>
private Rigidbody rb;

[Tooltip("A velocidade que a bola irá esquivar para os lados")]
[Range(0,10)]
public float velocidadeEsquiva = 5.0f;

[Tooltip("Velocidade com qual a bola irá se deslocar para a frente")]
[Range(0, 10)]
public float velocidadeRolamento = 5.0f;

// Use this for initialization
void Start () {
    //Obter acesso ao componente Rigidbody associado a esse GO
    rb = GetComponent<Rigidbody>();
}
// Update is called once per frame

```

(campo) Rigidbody JogadorComportamento.rb
 Uma referencia para o componente Rigidbody

Pode parecer muito código, mas ao digitar `///` o Visual Studio já gera automaticamente um corpo para o comentário. Como regra geral, use o [Tooltip] para mensagens que devem aparecer no editor do Unity, e use XML para mensagens internas para outros programadores.

4 – Movendo a câmera

Para nosso jogo, queremos que a câmera siga a bola. Podemos fazer de duas formas. Fazendo com que ela se torne filho do GO, ou usando o método `LookAt()`. Lembre-se que quando fazemos um GO se tornar filho, ele passa a ter um espaço local referente ao objeto pai (a bola). E qualquer mudança na Transform no pai irá causar mudanças no filho. Porém, isso não irá funcionar, pois a bola irá rolar, e câmera irá rolar também, causando um efeito indesejado. Vamos com a segunda opção. Comece criando um script chamado `ControleCamera` como mostra abaixo.


```

5 public class ControleCamera : MonoBehaviour {
6
7     [Tooltip("O alvo a ser acompanhado pela camera")]
8     public Transform alvo;
9
10    [Tooltip("Offset da camera em relação ao alvo")]
11    public Vector3 offset = new Vector3(0,3,-6);
12
13    // Update is called once per frame
14    void Update () {
15        if (alvo != null) {
16            //Altera a posicao da camera
17            transform.position = alvo.position + offset;
18
19            //Altera a rotacao da camera em relacao
20            transform.LookAt(alvo);
21        }
22    }

```

Vamos analisar esse código.

Na linha 8 criamos uma variável do tipo Transform, ou seja, um GO. Esse será o nosso alvo, ou para quem a câmera irá “olhar”.

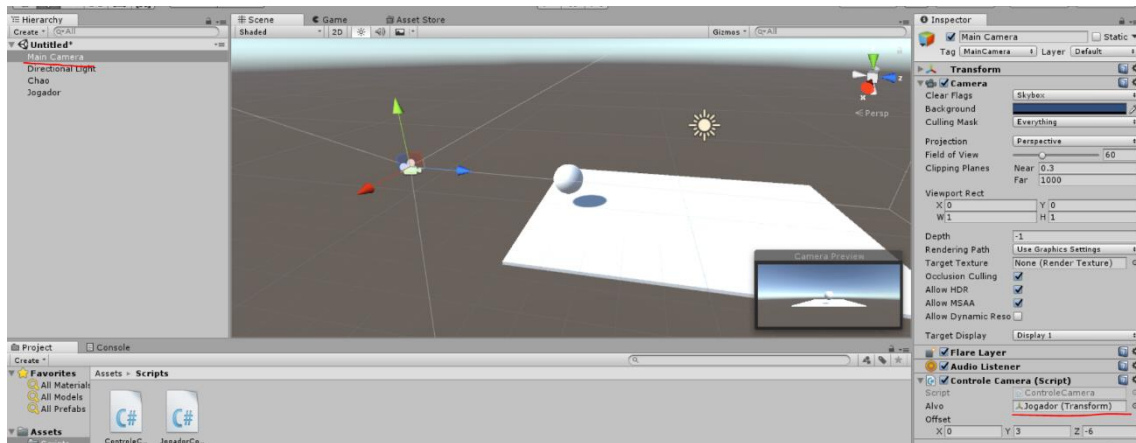
Na linha 11 declaramos um Vector3, que é simplesmente um vetor com 3 coordenadas, para ser a distância da câmera em relação ao alvo.

Dentro do método Update(), primeiro na linha 15 verificamos se realmente existe um alvo. Depois na linha 17 acessamos o componente Transform do próprio GO através do transform, e depois acessamos a posição através do position. No final temos: transform.position = um Vector3.

Note que para acessar o componente Transform não foi necessário usar o GetComponent() igual o Rigidbody, pois todo GO tem necessariamente um componente Transform. Veja que no Unity é tudo muito hierárquico e condizente com o que temos no editor. Olhando o componente Transform na aba Inspector, vemos que ele possui um campo Position. Portanto antes de acessarmos o Position, precisamos acessar o Transform. E no código vemos isso através do transform.position.

Agora precisamos garantir uma rotação adequada para a câmera. O método LookAt na linha 20 irá garantir isso. Esse método recebe uma variável do tipo Transform (ao longo do curso ficará claro a diferença entre transform, Transform e GameObject).

Salve o script e o adicione como componente da Câmera. Veja na figura abaixo



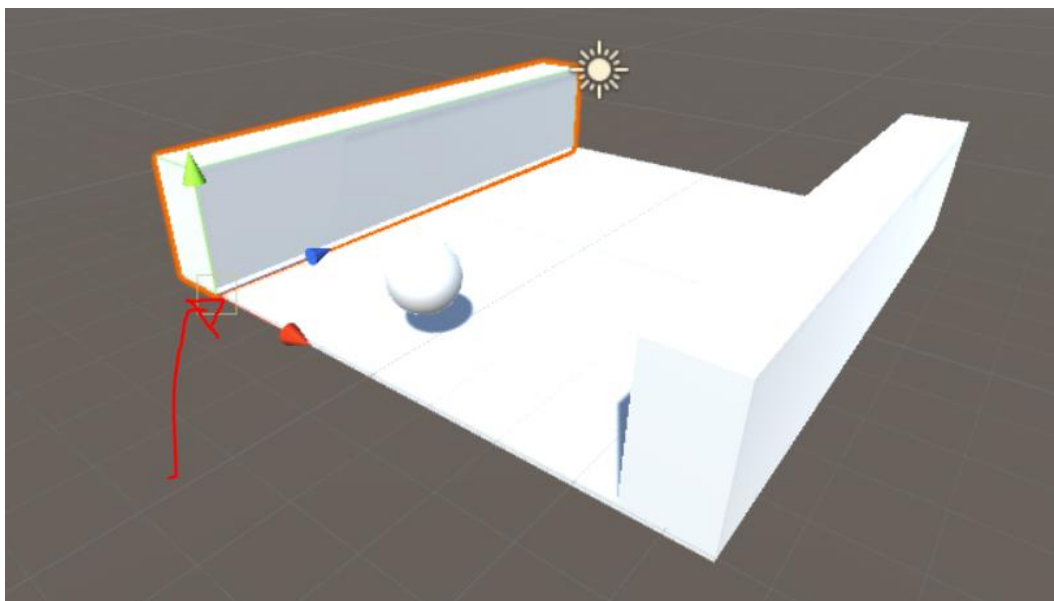
Primeiro selecionamos o GO MainCamera, adicionamos o script e veja que o campo Alvo estará vazio (olhe no Inspector). Agora arrastamos o GO Jogador para este campo “Alvo”. Salve e aperte Play. Agora a câmera irá acompanhar a bola!

5 – Criando o nosso Tile

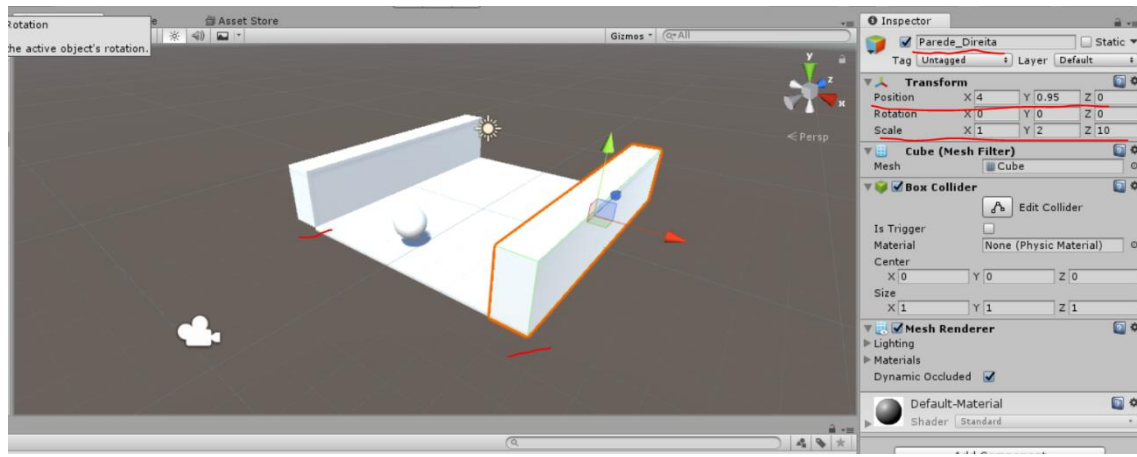
O jogo está chato, pois a bola fica caindo indefinidamente. Vamos arrumar isso criando um Tile básico. Esse tile será replicado ao longo da cena criando a nossa fase infinita. Lembre-se do conceito de tile apresentado na aula sobre Gráficos.

Vamos começar adicionando duas paredes para o nosso chão. Primeiro podemos duplicar o chão com o comando Ctrl+D e alterar a Scale para (1,2,10). Agora precisamos colocar no lugar.

Volte para o modo “Move” através da tecla W. Mesmo que você desloque esse GO pelos eixos talvez você não consiga deixar essa parede encaixada perfeitamente. Para isso temos o “VertexMode”. No modo “Move” aperte e segure a tecla V. Você vai perceber que o gizmo dos eixos se moveu para um vértice (o mais próximo do cursor do mouse). Agora você pode mover esse GO fazendo com o este vértice se encaixe no vértice de outro GO. Observe a figura abaixo.



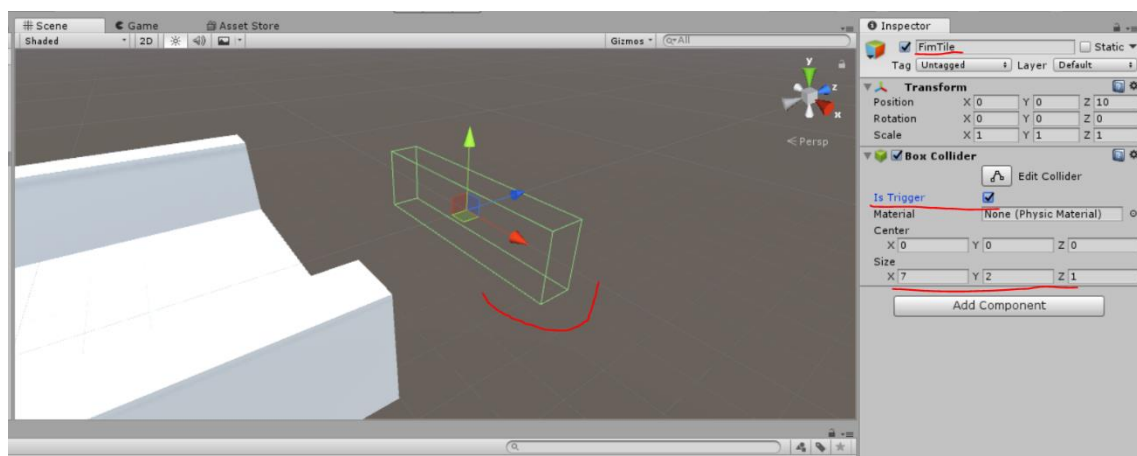
Agora que você já sabe usar o Vertex Mode. Termine de criar as paredes. O resultado deve ser algo como mostra abaixo.



Estamos montando um jogo do tipo *endless*. Então precisamos gerar esse Tile indefinidamente. Mas precisamos também eliminar os Tiles que ficam para trás, pois estes só irão consumir recursos desnecessário.

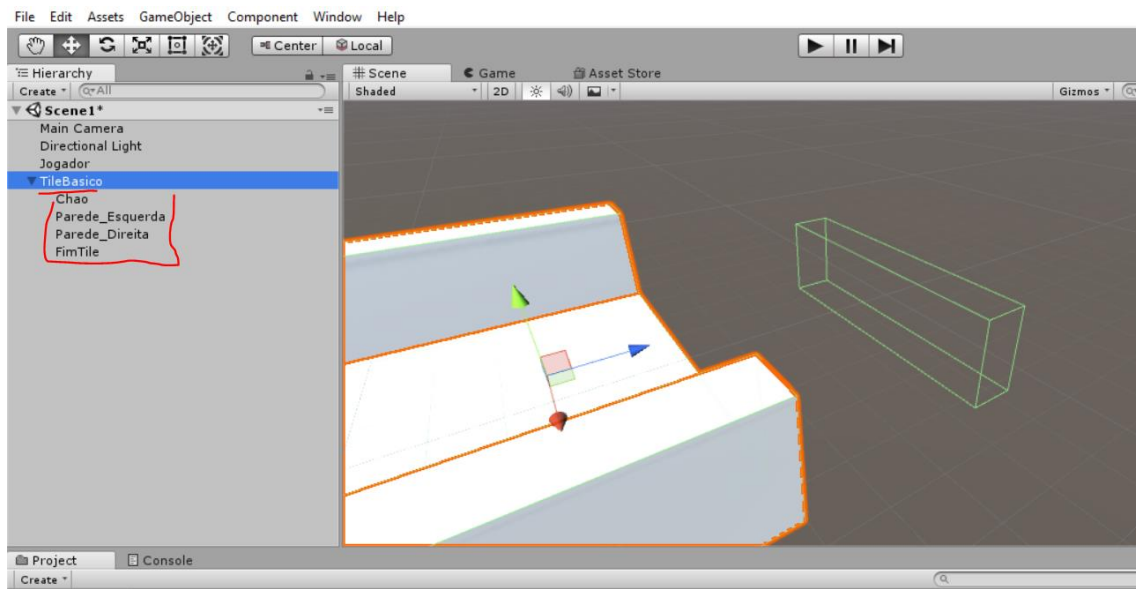
Vamos definir então um ponto para saber o fim do Tile. Assim que a bola passar deste ponto iremos destruir o Tile.

- Crie um GO vazio através de GameObject->Create Empty
 - Coloque o nome de FimTile
 - Coloque a Position (0,1,10)
 - Agora precisamos adicionar o componente BoxCollider (sem ser o 2D). O objetivo deste Collider é apenas detectar que a bola passou por este ponto. Ou seja, iremos detectar a colisão, mas não iremos impedir a bola de prosseguir. Para fazer isso basta marcar a opção *IsTrigger* dentro do componente BoxCollider.
 - Ainda dentro do componente BoxCollider mude o tamanho (size) para (7,2,1)
- Observe a figura abaixo:



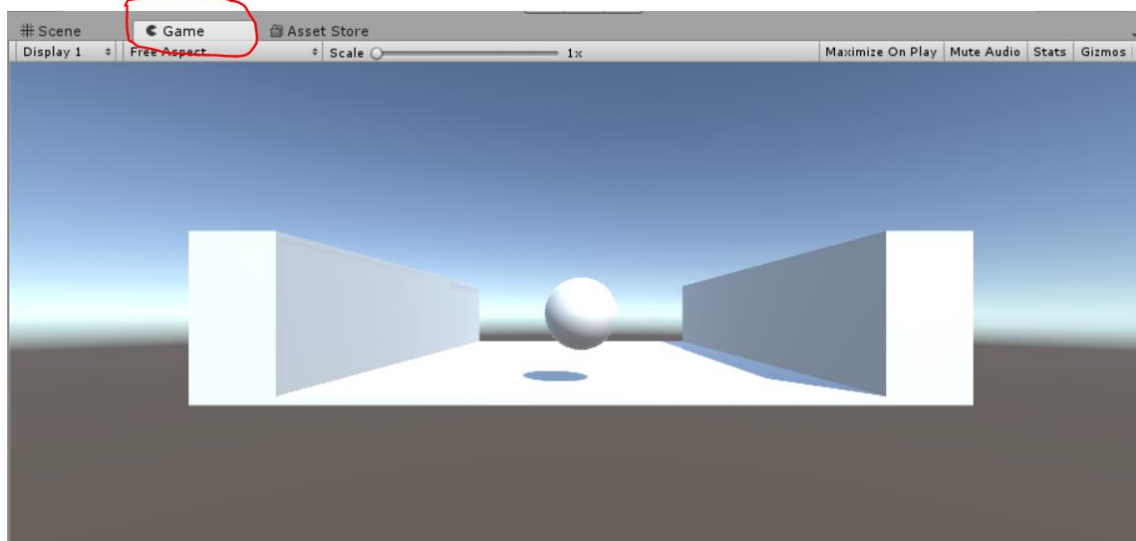
Quando usamos a opção `IsTrigger` dentro de algum Collider, estamos informando para o Unity que queremos apenas detectar a colisão, e através de código nós iremos decidir o que fazer. Essa técnica é muito usada em desenvolvimento de jogos.

Agora o nosso Tile está pronto. Vamos então agrupar todas as partes em um único GO, para ficar mais fácil duplicar este Tile pelo jogo. Crie um GO vazio, resete a posição, coloque o nome de `TileBasico` e arraste todos os GOs que compõem o Tile para dentro do GO `TileBasico`. Faça isso na aba `Hierarchy`. Veja a figura.



O que temos agora é uma relação hierárquica. Se movimentarmos o GO `TileBasico` todos os demais irão se movimentar juntos. Faça o teste.

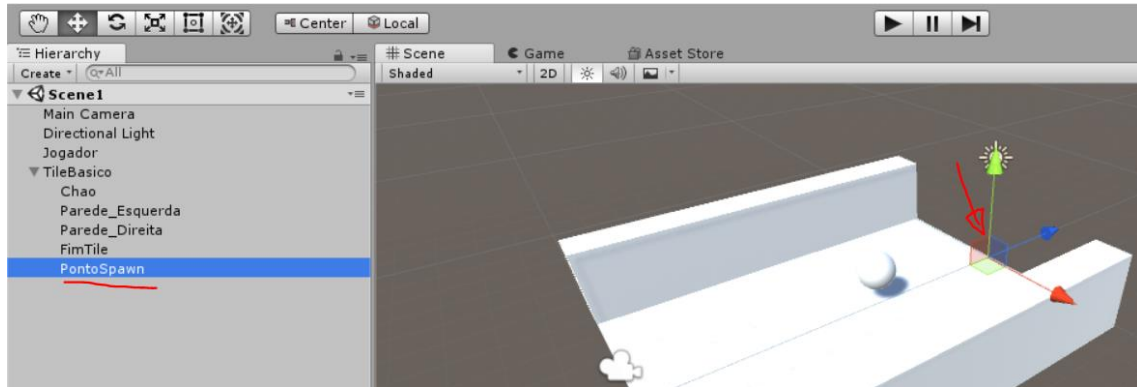
Se clicar na aba `Game`, vemos que a Câmera mostra o início do Tile.



Para arrumar isso para alterarmos a posição da Câmera, ou do `TileBasico`. Vamos com a segunda opção. Altere a posição para `(0,0,-5)`.

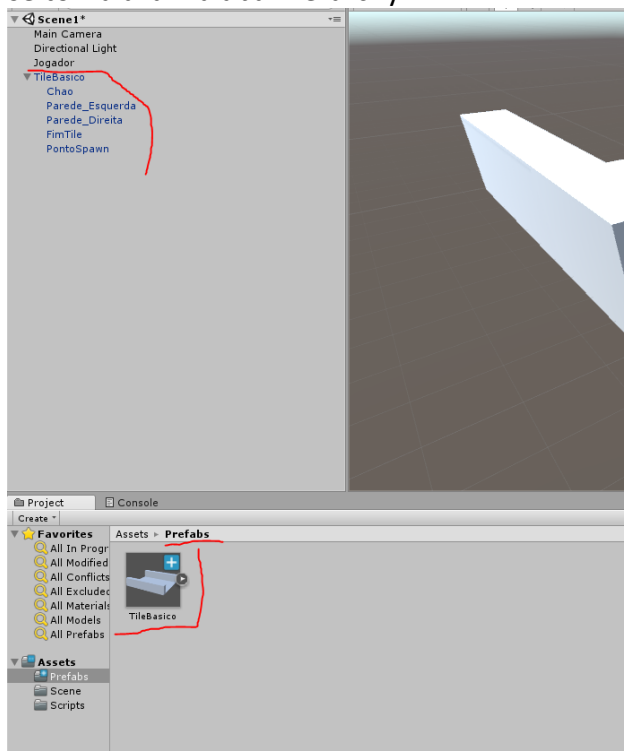
Para fechar nosso Tile, precisamos de um outro ponto que nos indique que devemos criar um novo Tile para jogo prosseguir. Vamos usar a mesma técnica usada no FimTile.

- Na aba Hierarchy, dentro do TileBasico clique com o botão direito -> CreateEmpty
- Coloque o nome de PontoSpawn
- Muda a sua posição para (0,0,5). Perceba que na verdade essa posição é em relação ao GO pai, e não ao *game world*.



Prefabs: Antes de duplicarmos nosso TileBasico, vamos entender o conceito de prefabs no Unity. É como se fosse uma fábrica de objetos, objetos pré-fabricados (prefabs). Qualquer modificação feita no prefab, se propaga para todos os GOs que foram criados a partir do mesmo prefab.

Crie uma pasta chamada Prefabs, nela colocaremos todos os prefabs deste jogo. A forma mais rápida de criar prefabs no Unity é simplesmente arrastando o GO da Hierarchy para dentro da pasta Prefab. Quando um GO está associado a um prefab, ele se torna azul na aba Hierarchy.



6 – Se tornando infinito

Como acabamos de criar o prefab TileBasico, podemos deletar o TileBasico que está na Scene (cena). Precisamos agora definir a estratégia para criar infinitos TileBasico. Para isso iremos criar um GO vazio chamado ControladorJogo. Este GO ficará responsável por gerenciar informações sobre o jogo. Vamos ver como isso irá funcionar.

- Crie um GO vazio chamado ControladorJogo e resete sua posição (lembre-se, é no componente Transform que fazemos isso)
- Dentro da pasta Script crie um C# Script chamado ControladorJogo e abra no Visual Studio. A figura abaixo mostra os atributos desta classe.

```
7  /// Classe para controlar a parte principal do jogo
8  /// </summary>
9  public class ControladorJogo : MonoBehaviour {
10
11      [Tooltip("Referencia para o TileBasico")]
12      public Transform tile;
13
14      [Tooltip("Ponto para se colocar o TileBasicoInicial")]
15      public Vector3 pontoInicial = new Vector3(0, 0, -5);
16
17      [Tooltip("Quantidade de Tiles iniciais")]
18      [Range(1,20)]
19      public int numSpawnIni;
20
21      /// <summary>
22      /// Local para spawn do proximo Tile
23      /// </summary>
24      private Vector3 proxTilePos;
25
26      /// <summary>
27      /// Rotacao do proximo Tile
28      /// </summary>
29      private Quaternion proxTileRot;
30
31      // Use this for initialization
32      void Start () {
```

Vamos analisar os atributos

- Linha 12 é uma referência para o nosso TileBasico. Iremos arrastar o prefab TileBasico no Inspector para este campo
- Linha 15. O valor inicial onde o primeiro TileBasico irá aparecer
- Linha 20. Variável inteira definindo o número de tiles iniciais
- Linha 24 e 29 controlam a posição e rotação do próximo TileBasico.

Na próxima Figura apresento os métodos utilizados.

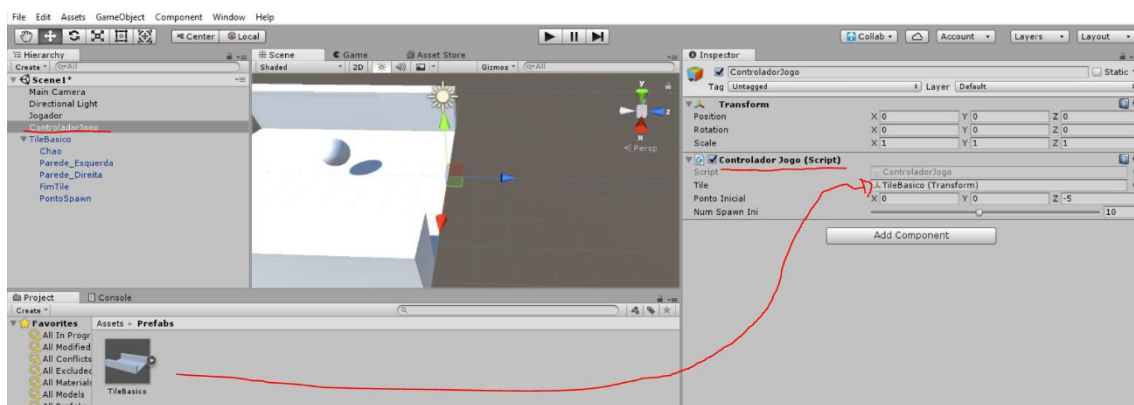
```

32 void Start () {
33     // Preparando o ponto inicial
34     proxTilePos = pontoInicial;
35     proxTileRot = Quaternion.identity;
36
37     for (int i = 0; i < numSpawnIni; i++)
38     {
39         SpawnProxTile();
40     }
41 }
42
43 public void SpawnProxTile()
44 {
45     var novoTile = Instantiate(tile, proxTilePos, proxTileRot);
46
47     //Detectar qual o local de spawn do prox tile
48     var proxTile = novoTile.Find("PontoSpawn");
49     proxTilePos = proxTile.position;
50     proxTileRot = proxTile.rotation;
51 }
52

```

Dentro do método Start(), nas linhas 34 e 35 definimos os pontos do primeiro TileBasico. Depois dentro do *for* chamamos o método SpawnProxTile() para de fato criar o TileBasico. A cada vez que este método é chamado a variável proxTilePos é atualizada, permitindo criar TileBasicos seguidos uns dos outros.

Salve esse script e o adicione como componente ao GO ControladorJogador. Arraste para o campo tile o prefab TileBásico.



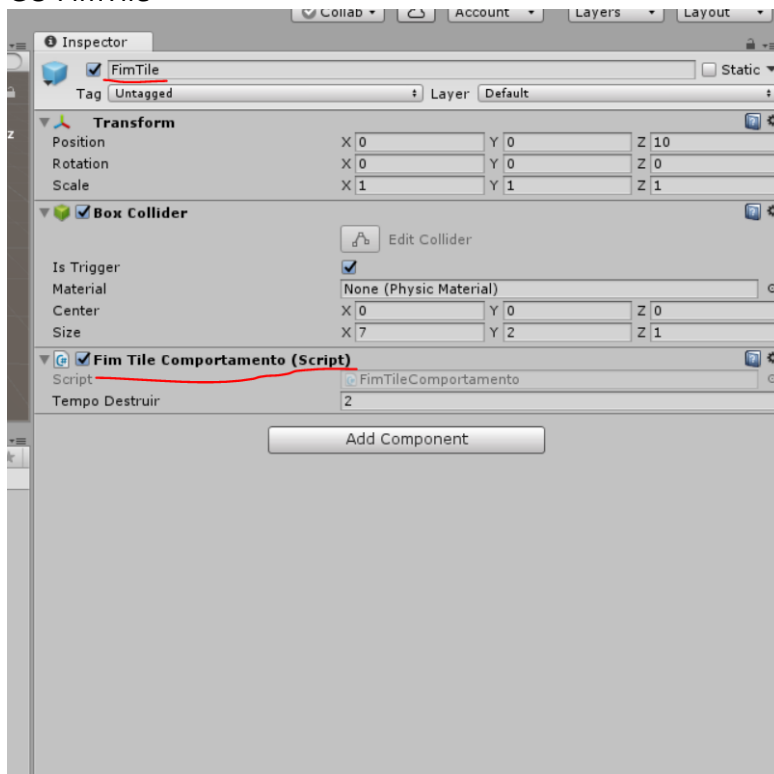
Agora aperte o "Play". O jogo ficou um pouco melhor, pois existe mais TileBasico para a bola rolar, mas ainda sim não está infinito. Isso faz sentido, pois usamos apenas o método Start(), que o Unity chama uma única vez.

Ainda precisamos criar TileBasico infinitamente, mas não podemos sair fazendo isso de forma descontrolada. Vamos primeiro cuidar de destruir o TileBasico que não é mais utilizado.

Crie um script chamado FimTileComportamento.

```
5 public class FimTileComportamento : MonoBehaviour {
6
7     [Tooltip("Tempo esperado antes de destruir o TileBasico")]
8     public float tempoDestruir = 2.0f;
9
10    // Use this for initialization
11    void Start () {
12    }
13
14    // Update is called once per frame
15    void Update () {
16    }
17
18    private void OnTriggerEnter(Collider other) {
19
20        //Vamos ver se foi a bola que passou pelo fim do TileBasico
21        if (other.GetComponent<JogadorComportamento>())
22        {
23            //Como foi a bola, vamos criar um TileBasico no proximo ponto
24            //Mas esse proximo ponto esta depois do ultimo TileBasico presente na cena
25            GameObject.FindObjectOfType<ControladorJogo>().SpawnProxTile();
26
27            //E agora destrói esse TileBasico.
28            Destroy(transform.parent.gameObject, tempoDestruir);
29        }
30    }
```

Veja na linha 18 o método OnTriggerEnter(). Esse método é o próprio Unity que chama quando algum GO que possui um Collider marcado com a opção IsTrigger atravessa outro GO com Collider. Esse script iremos associar a nossa prefab TileBasico, mas iremos buscar o FimTile. Para fazer isso basta ir na pasta prefab, expandir o TileBasico e clicar no FimTile. Agora vá na pasta script e adicione este FimTileComportamento ao GO FimTile

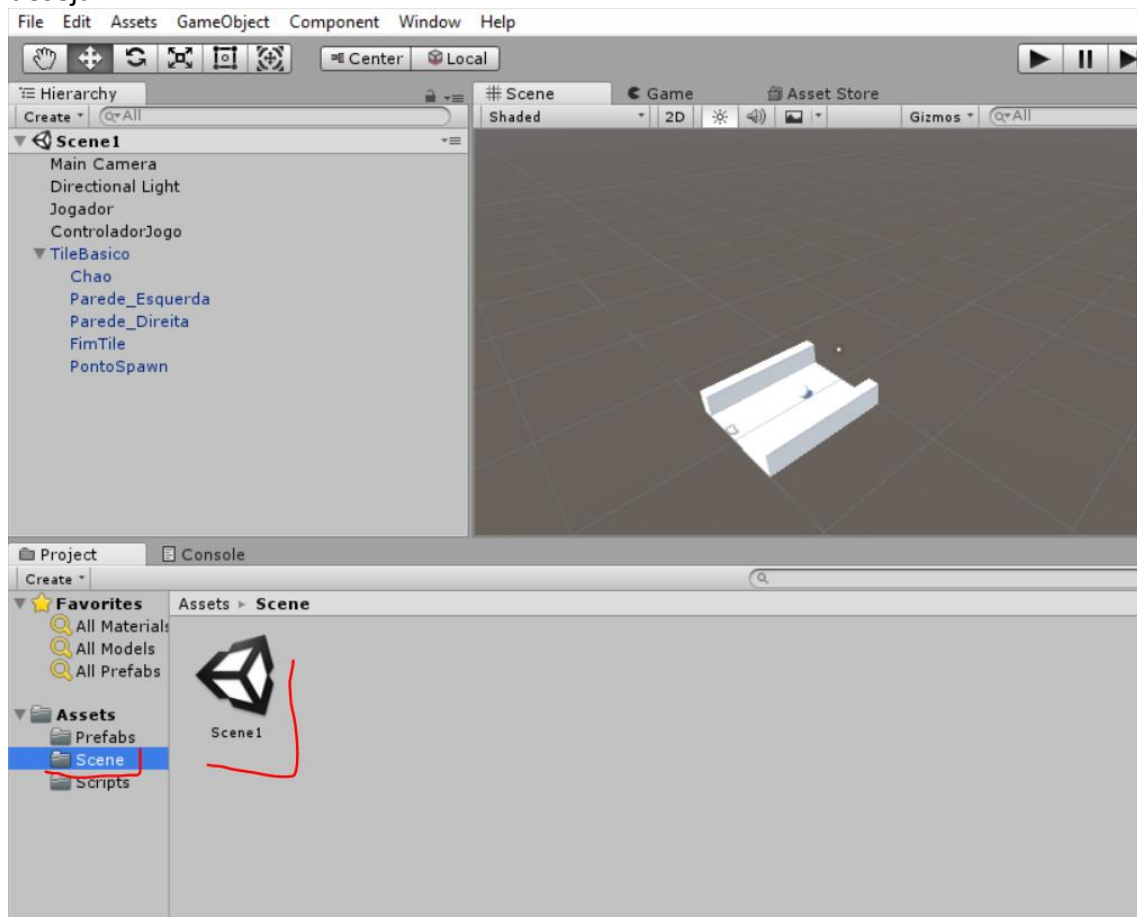


Voltando ao script, falando do método `OnTriggerEnter()`. Toda vez que algum GO com algum Collider e `IsTrigger` atravessar o `FimTile`, este método é chamado. A primeira coisa que fazemos é verificar se quem atravessou foi a bola na linha 21. Se tiver sido procuramos onde devemos criar um novo `TileBasico` (linha 25) e depois destruímos o `TileBasico` em questão.

Por que dentro do método `Destroy` usamos `transform.parent.gameObject`? Lembre-se que o Unity possui uma estrutura hierárquica. Como estamos no GO `FimTile`, que por sua vez é filho do `TileBasico`, o que desejamos é destruir o pai. Para isso fazemos o acesso pelo transform (componente Transform) . parente (estamos no pai agora) . gameObject (o GO que representa o pai).

Salve o script e aperte Play. Se quiser ver em tempo real o `TileBasico` ser construído e destruído alterne para a aba Scene.

Até agora não salvamos Scene dentro do Unity. Crie uma pasta Scene e vá em File->Save Scenes. Escolha um nome. Você pode usar este arquivo para abrir o Unity se desejar.

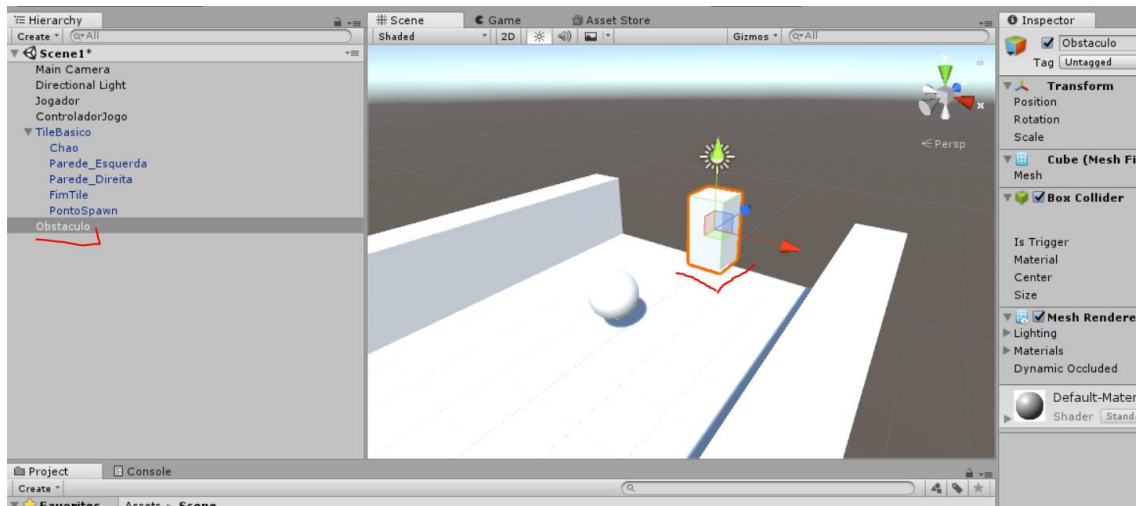


Tudo o que existe na Scene fica salvo através deste arquivo

7 - Criando Obstáculos

Já temos o nosso jogo infinito. Mas vamos colocar alguns desafios para nosso jogador. Se sua Scene não tiver nenhum TileBasico, arraste um para nos guiar na criação de um obstáculo.

Para começar a criar o obstáculo, crie um Cube (GameObject -> 3D Object -> Cube), chame de Obstaculo (sem acento mesmo). Coloque a Scale (1,2,1) e Position(0,1,0.025).



Se jogar o jogo nesse ponto, a bola irá rolar até chegar no obstáculo. Como ele não está marcado como IsTrigger, a bola parar. Mas nada mais acontece. No jogo, queremos que o jogador seja derrotado (game over), e reinicie o jogo. Vamos criar um script ObstaculoComp para tratar disso.

```
6 public class ObstaculoComp : MonoBehaviour {
7
8     [Tooltip("Quanto tempo antes de reiniciar o jogo")]
9     public float tempoEspera = 2.0f;
10
11     private void OnCollisionEnter(Collision collision) {
12
13         //Verifica se eh(acho boa pratica nao usar acento) o jogador
14         if (collision.gameObject.GetComponent<JogadorComportamento>())
15         {
16             Destroy(collision.gameObject);
17             Invoke("ResetaJogo", tempoEspera);
18         }
19     }
20     /// <summary>
21     /// Reinicia o level
22     /// </summary>
23     void ResetaJogo()
24     {
25         // Reinicia o level
26         SceneManager.LoadScene(SceneManager.GetActiveScene().name);
27     }
28 }
```

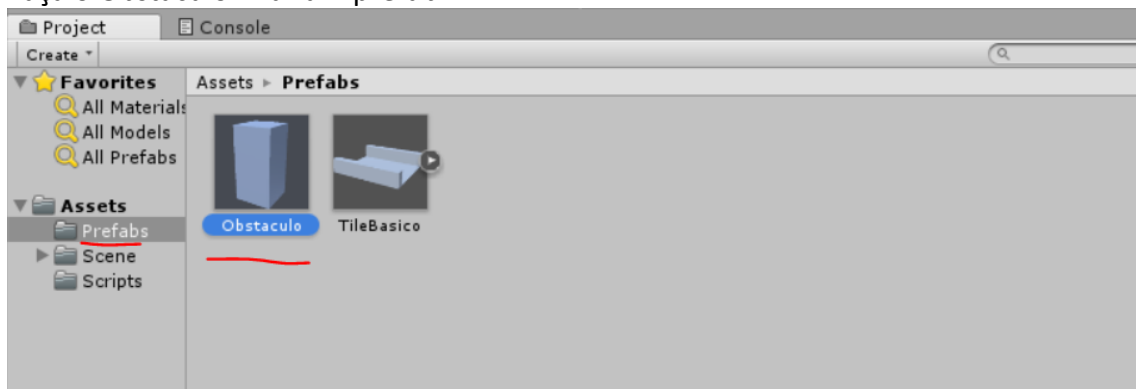
Adicione esse script ao Obstaculo. Agora quando você jogar, verá que a bola para no obstáculo e o jogo é reiniciado. Está feio, mas por enquanto é o que precisamos.

#DICA: Você deve ter notado algo de errado com a iluminação. Isso é um problema do Unity. Para resolver isso vá em Window -> Lighting -> Settings. Desmarque a opção Auto Generate e clique em Generate Lighting.

Vamos analisar o script. Na linha 11 temos o método OnCollisionEnter(). A diferença deste método para o OnTriggerEnter é que este detecta colisões em objetos que não estão marcados com o IsTrigger. Ou seja, a física irá entrar em ação e o objeto irá colidir como se fosse no mundo real. Faça um teste, no componente BoxCollider do Obstaculo, marque-o como IsTrigger e veja o que acontece.

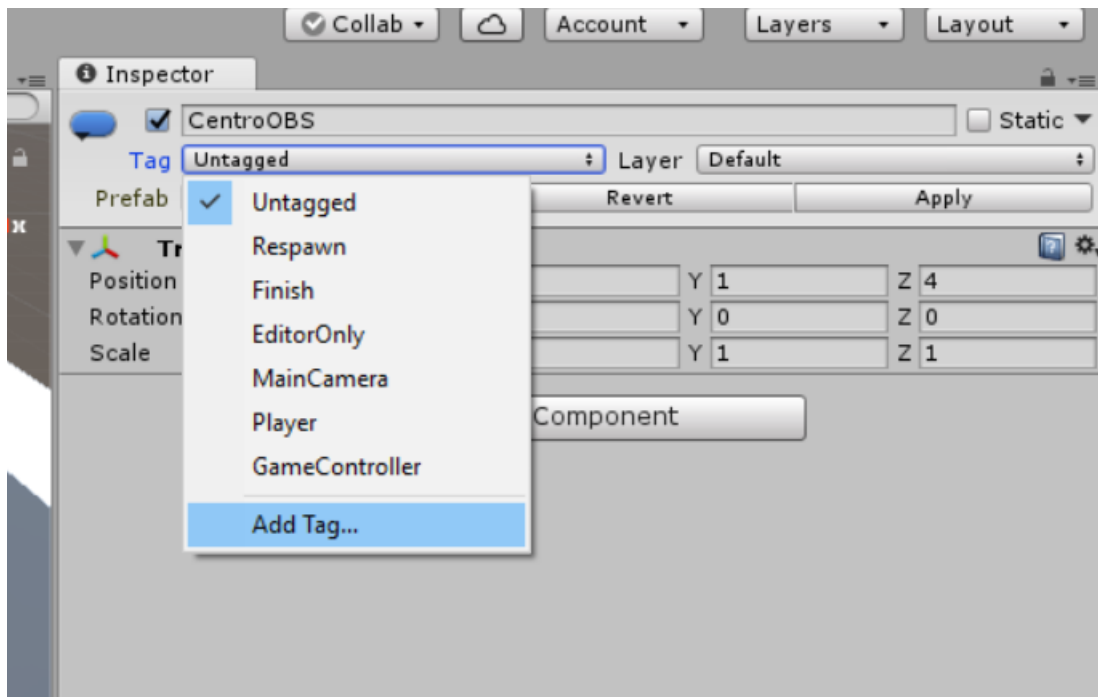
Na linha 17, usamos o Invoke() para chamar o método Reset() depois de tempoEspera. Para reiniciar a cena, ou level, fase, scene (tantos nomes) usamos a classe SceneManager (não esqueça de importar o namespace). Essa classe possui vários métodos para manipular Scenes no Unity. Estamos usando o LoadScene(). Este método pode receber o nome da Scene que queremos carregar. Queremos a própria Scene em questão, por isso usamos o GetActiveScene().name como parâmetro.

Faça o Obstaculo virar um prefab.

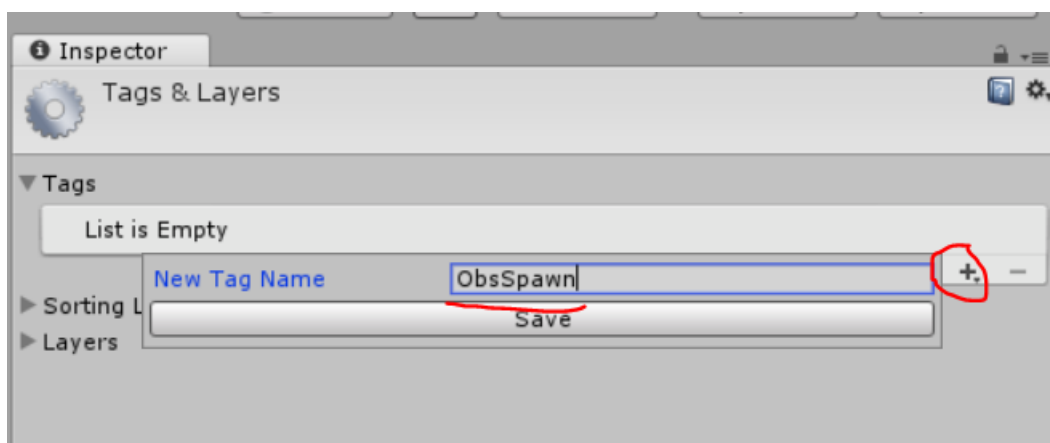


Vamos agora criar 3 pontos possível onde obstáculo podem aparecer no TileBasico. Comece colocando um TileBasico na Scene. Duplicue o GO PontoSpawn, renomeie para CentroOBS. Coloque na posição (0,1,4).

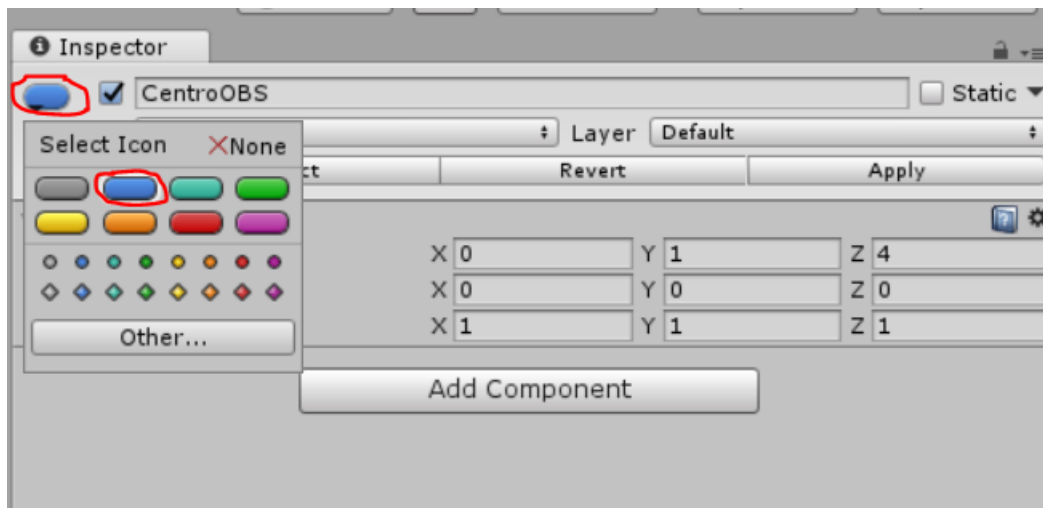
Outra forma buscar GO por scripts é através de tags (etiquetas). Com o CentroOBS selecionado vá até o Inspector e clique em Tag -> Add tag



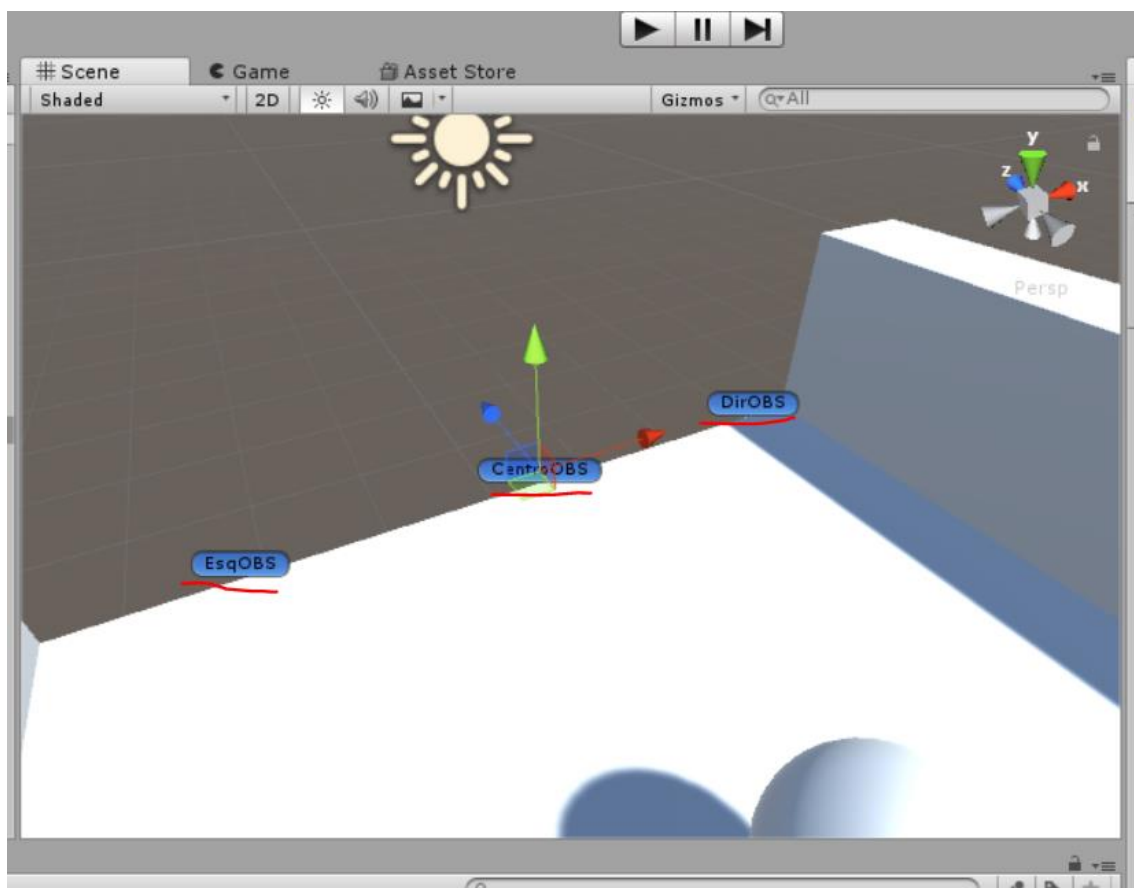
Depois clique no + e coloque como nome ObsSpawn e pressione Save.



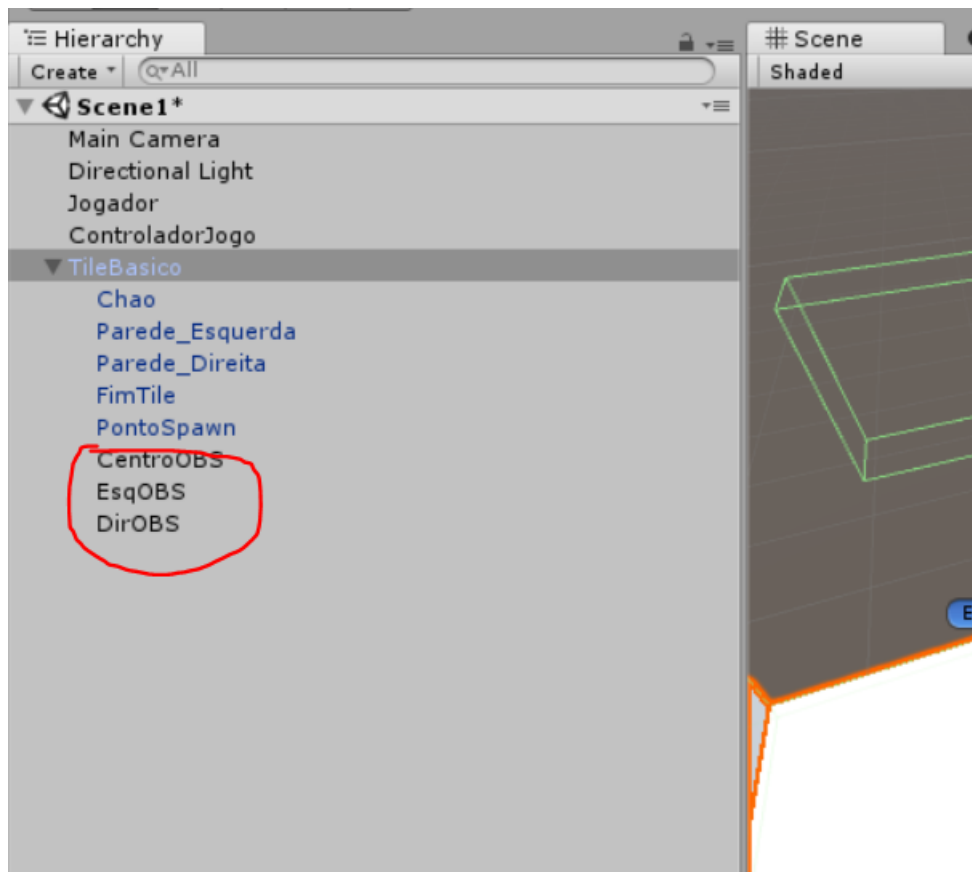
Faça mais duas cópias desse GO, chame de EsqOBS com posição (-2,1-4) e o outro DirOBS na posição (2,1,-4). Para facilitar visualizar estes GOs na Scene podemos habilitar para que seu nome apareça. Clique no CentroOBS, vá até o Inspector e clique no canto esquerda no botão azul, e selecione azul. Faça como na figura abaixo



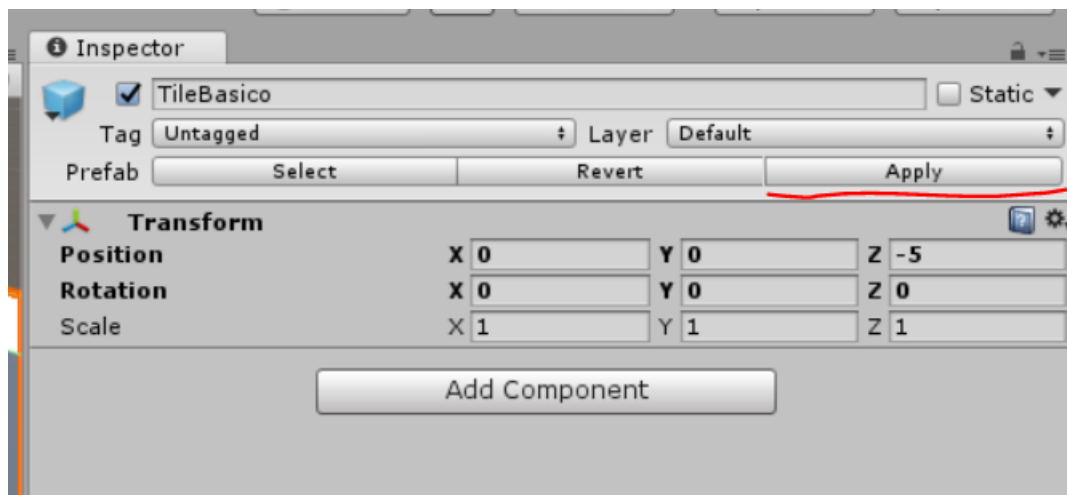
O resultado fica como na figura abaixo



Perceba na aba Hierarchy, que estes novos GOs estão em preto. Isto quer dizer que o prefab TileBasico não contém essas informações, apenas o GO local dentro da Scene.



Para isso, clique no TileBasico na Scene e pressione Apply como mostra abaixo



Agora todos os filhos do TileBasico estão com o nome em azul.

Para finalizar vamos trabalhar no Script ControladorJogo para manipular esses obstáculos. Vá no Visual Studio e faça as seguintes modificações.

```

| /// </summary>
| public class ControladorJogo : MonoBehaviour {
|
|     [Tooltip("Referencia para o TileBasico")]
|     public Transform tile;
|
|     [Tooltip("Referencia para o Obstaculo")]
|     public Transform obstaculo;
|
|     [Tooltip("Ponto para se colocar o TileBasicoInicial")]
|     public Vector3 pontoInicial = new Vector3(0, 0, -5);
|
|     [Tooltip("Quantidade de Tiles iniciais")]
|     [Range(1, 20)]
|     public int numSpawnIni = 15;
|
|     [Tooltip("Numero de Tiles sem obstaculos")]
|     [Range(1,4)]
|     public int numTileSemOBS = 4;
|

```

Adicione uma referência para o prefab Obstáculo, e um variável inteira para determinar o número de TileBasicos iniciais sem obstáculos. Essa é uma medida para o jogador não começar o jogo imediatamente com obstáculos.

Agora observe o método Start()

```

38  | void Start () {
39  |     // Preparando o ponto inicial
40  |     proxTilePos = pontoInicial;
41  |     proxTileRot = Quaternion.identity;
42  |
43  |     for (int i = 0; i < numSpawnIni; i++)
44  |     {
45  |         SpawnProxTile(i >= numTileSemOBS);
46  |     }
47  | }

```

Agora o método SpawnProxTile(bool spawnObstaculos = true) recebe um parâmetro booleano dizendo se o TileBasico terá um obstáculo. Dentro desse *for*, queremos que somente o TileBasico depois do numTileSemOBS (atualmente vale 4) passe a ser criado com obstáculos.

Vamos analisar o método SpawnProxTile()

```
59 //Verifica se ja podemos criar Tiles com obstaculo.
60 if (!spawnObstaculos)
61     return;
62
63 //Podemos criar obstaculos
64
65 //Primeiro devemos buscar todos os locais possiveis
66 var pontosObstaculo = new List<GameObject>();
67
68 //Varrer os GOs filhos buscando os pontos de spawn
69 foreach (Transform filho in novoTile)
70 {
71     //Vamos verificar se possui a TAG PontoSpawn
72     if (filho.CompareTag("PontoSpawn"))
73         //Se for o adicionamos na lista como potencial ponto de spawn
74         pontosObstaculo.Add(filho.gameObject);
75 }
76
```

Na linha 60 verificamos se podemos criar obstáculos.

Na linha 66 criamos uma Lista que recebe os possíveis pontos de Spawn para os obstáculos.

Na linha 69 varremos todos os GOs filho do TileBasico que acaba de ser criado em busca dos três pontos de Spawn. Percebe que o filho é do tipo Transform. Quando percorremos um GO buscando Transform, o retorno são os filhos deste GO.

Depois verificamos se o filho possui a TAG que o caracteriza como PontoSpawn. Se possuir a TAG o adicionamos na Lista. Continuando no método SpawnProxTile()

```
77 //Garantir que existe pelo menos um spawn point disponivel
78 if(pontosObstaculo.Count > 0){
79
80     //Vamos pegar um ponto aleatório
81     var pontoSpawn = pontosObstaculo[Random.Range(0, pontosObstaculo.Count)];
82
83     //Vamos guardar a posicao desse ponto de spawn
84     var obsSpawnPos = pontoSpawn.transform.position;
85
86     //Cria um novo obstaculo
87     var novoObs = Instantiate(obstaculo, obsSpawnPos, Quaternion.identity);
88
89     //Faz ele ser filho do TileBasico.PontoSpawn (centro, esq ou direita)
90     //Outra forma de fazer isso eh no proprio Instantiate. Ja existe uma sobrecarga para adicionar
91     //um parent.
92     novoObs.SetParent(pontoSpawn.transform);
93 }
```

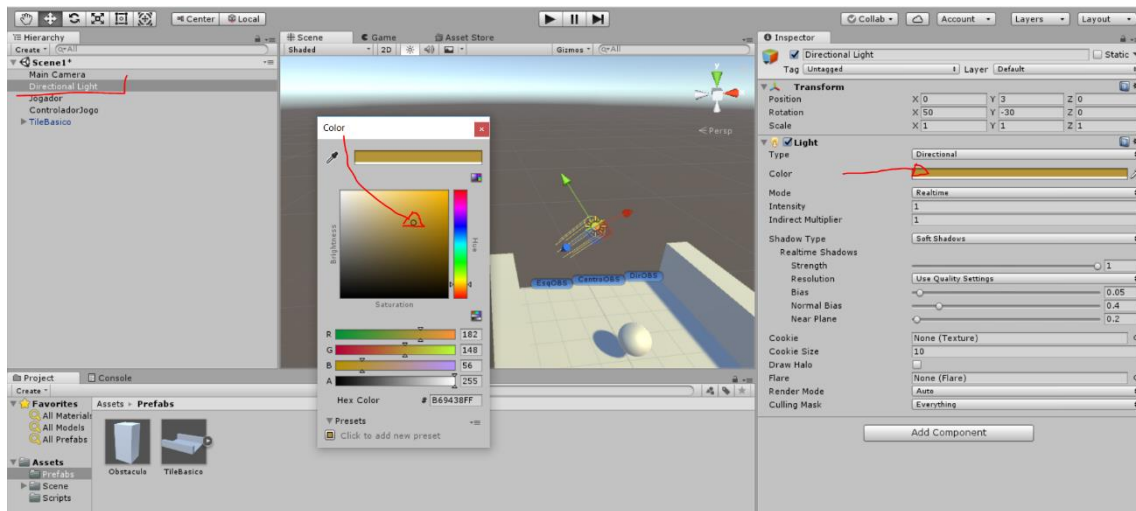
Na linha 80 pegamos um ponto aleatório da Lista de pontos de Spawn

Na linha 84 pegamos a posição que usaremos (na linha 80 pegamos o GameObject, agora que pegamos o position, que é dado em Vector3)

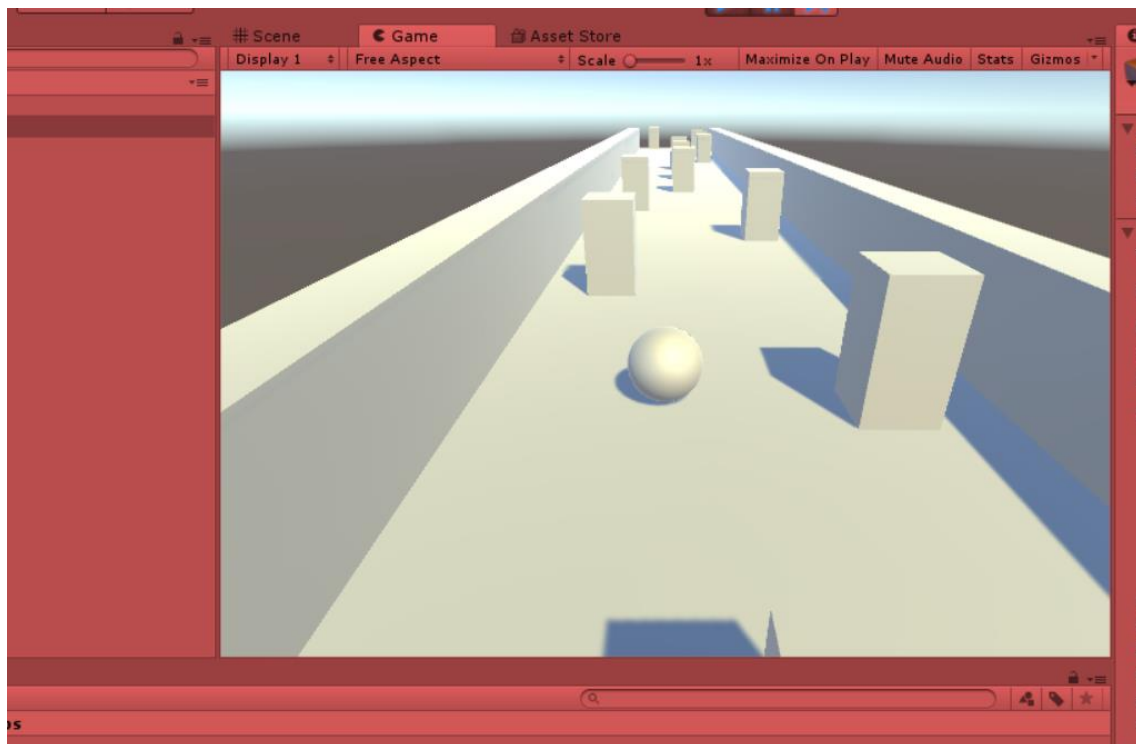
Na linha 88 criamos o obstáculo e na linha 92 colocamos ele como filho do ponto de Spawn.

Pronto, jogue e seja feliz!.....

Ainda tem um problema. A iluminação está atrapalhando. O brilho está excessivo. Na aba Hierarchy clique na DirectionalLight (funciona como o sol, irradiando para todos os lados). Na aba Inspector, componente Light clique em color e pega uma cor um pouco mais escura. Veja a figura



Agora apertando o Play temos



Agora sim pode sair jogando!