

cookbook

Desenvolvimento de jogos

com Unity



Guia prático dicas e mecânicas

Wiliam Nascimento

CookBook para desenvolvimento de jogos com Unity



Todos os direitos reservados à: Wiliam Trindade Nascimento - WTN Cursos e Games,
promowtnkursose@gmail.com.br - wtncursosegames.com

CookBook para desenvolvimento de jogos com Unity

Apresentação



Sobre o autor	05
Introdução	06

Parte 1 – Mecânicas

Capítulo 1 - O Pulo perfeito.....	07
Capítulo 2 - Cronometrando o jogo.....	16
Capítulo 3 - Movimento Inteligente com o uso do Mouse.....	20
Capítulo 4 - Movimento Simples e com RigidBody.....	33
Capítulo 5 - Detectando corpo em movimento	41
Capítulo 6 - Efeito páraquedas.....	44
Capítulo 7 - Pausando o Game.....	56
Capítulo 8 - Efeito Estilingue.....	59
Capítulo 9 - Plataforma Movel.....	64
Capítulo 10 - Efeito Corrente.....	69
Capítulo 11 - Veículo 2D.....	72
Capítulo 12 - Tiro 2D.....	80
Capítulo 13 - Impulso 2D.....	87
Capítulo 14 - Bomba 2D.....	89
Capítulo 15 - Água 2D.....	95

Capítulo 16 - Objeto Encoberto.....	97
Capítulo 17 - NockBack.....	103
Capítulo 18 - Dash Effect.....	111
Capítulo 19 - Drop Down	117
Capítulo 20 - RagDoll 2D.....	120
Capítulo 21- Animação com BlendTree.....	126
Capítulo 22 - Puxar e empurrar objetos.....	133
Capítulo 23 - Trajetória de lançamento.....	139
Capítulo 24 - Movimento Tile by Tile.....	149
Capítulo 25 - Camera Follow.....	153
Capítulo 26 - Camera Confinada.....	156
Capítulo 27 - Object Pooling.....	159
Capítulo 28 - AI Patrulha.....	166
Capítulo 29 - Sistema de Combo.....	172
Capítulo 30 - Trigonometria básica para games.....	177

Parte 2 - Principais erros no desenvolvimento de games

Capítulo 31 - Problemas com nome de classes.....	180
Capítulo 32 - Problema com Variaveis não encontradas.....	181
Capítulo 33 - Matando quem deveria estar vivo.....	183
Capítulo 34 - Chamando mais pessoas do que a festa suporta.....	184
Capítulo 35 - Ao infinito nem sempre vai ao alem.....	185
Capítulo 36 - Recolhendo o lixinho.....	186

Sobre o Autor

Analista de Sistemas, Desenvolvedor de jogos eletrônicos e Professor.

Trabalho com desenvolvimento de jogos e ensinando a desenvolver com Unity e Unreal há mais de 10 anos, porém iniciei minha carreira há cerca de 13 anos, quando comecei programando em Visual Basic e logo o interesse pelos jogos eletrônicos me fez ir além de jogar e comecei a estudar para me tornar desenvolvedor, o interesse em saber como eram feitos os jogos que eu tanto gostava me levou a iniciar minha carreira.

A partir daí não parei mais de me dedicar e aprender, não somente sobre desenvolvimento de jogos, mas também sobre tudo que diz respeito à área de T.I. Então me graduei, fiz vários cursos de especialização, conclui a primeira pós-graduação e atualmente já estou na segunda e já pensando também no mestrado. Nem preciso dizer que sou apaixonado pela minha profissão.

Há cerca de 7 anos comecei a sentir que eu poderia ir além e compartilhar meu conhecimento e minhas experiências desenvolvendo cursos. Desde então comecei a trabalhar no desenvolvimento de cursos e consultoria para o desenvolvimento de projetos de jogos.

Atualmente já tenho mais de 20.000 alunos inscritos em meus cursos.



Introdução

Nesse e-book você irá aprender alguns dos mais importantes efeitos utilizados dentro do mundo dos games.

Você terá um guia para te direcionar na hora de adicionar aquele efeito legal, que você sempre vê nos seus jogos preferidos, mas nunca encontrou a maneira correta de reproduzi-lo.

Reuni os efeitos mais pedidos e procurados de forma direta e objetiva ensino a mecânica de cada um desses efeitos.

Aproveite e tenha sempre esse guia à mão durante o processo de desenvolvimento dos seus projetos.

Agora mãos à obra!

Tire as ideias do papel e transforme-as em games!

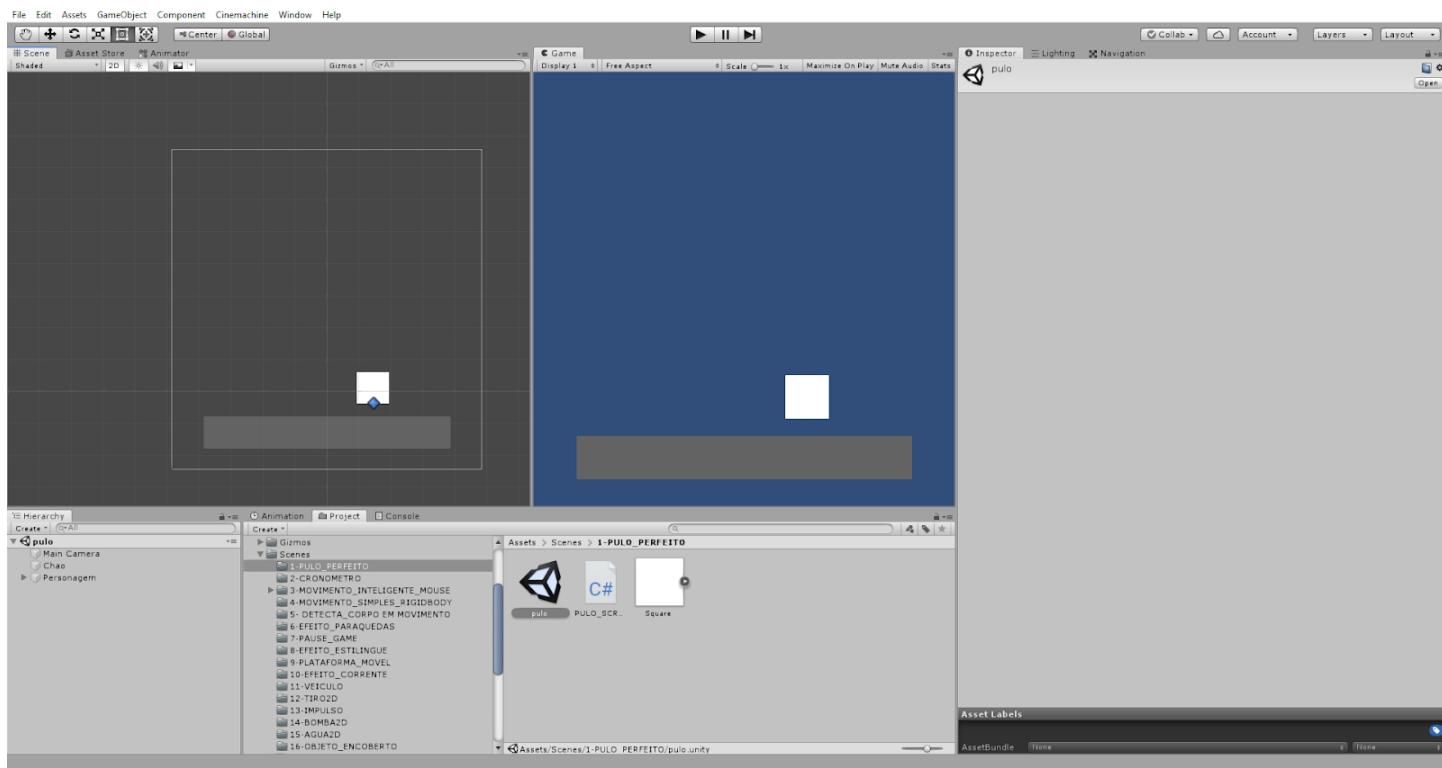
O Pulo Perfeito

Quando estamos criando jogos onde o personagem principal precisa pular, sempre tentamos chegar o mais perto possível dos efeitos de pulo que são executados em jogos como Super Mario Bros. e Rayman.

E isso tem uma razão muito simples, esses jogos mandam muito bem na jogabilidade. O efeito de pulo é simplesmente lindo, você controla a altura do pulo e isso ajuda muito em games que exigem velocidade.

Que tal aprender a reproduzir esse pulo?

A primeira coisa que precisamos fazer é criar uma cena como a apresentada logo abaixo:

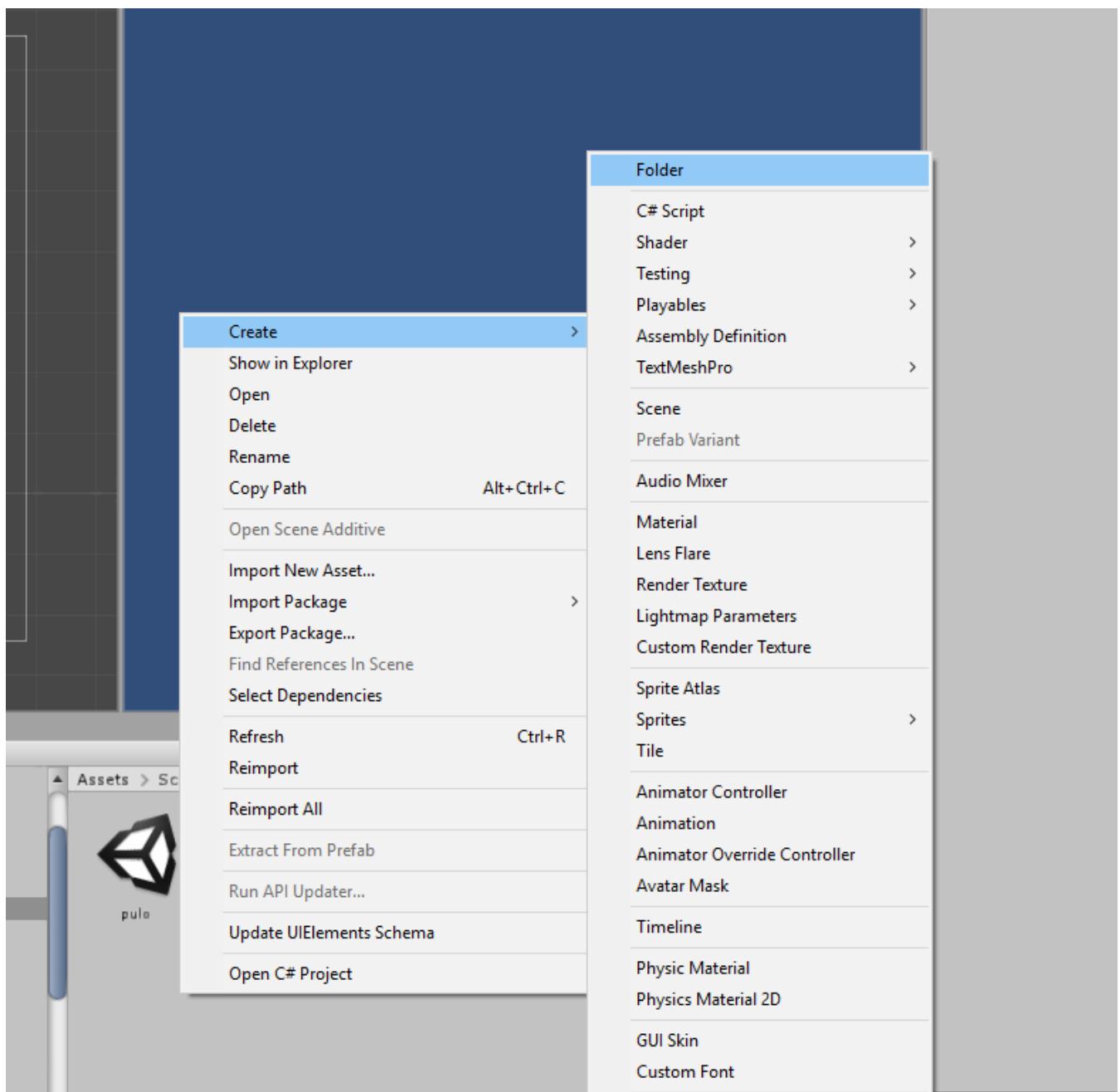


Veja que essa cena é muito simples, temos apenas uma sprite que representa o chão e outra sprite que representa o personagem.

Essas sprites foram criadas dentro de uma pasta que serve para organizar os arquivos desse exemplo.

Para criar a pasta é muito fácil, basta clicar com o botão direito do mouse sobre a pasta **Assets** ou **Scenes** e depois nas opções que vão surgir escolher a opção **Create** seguida de **Folder**.

Veja abaixo:

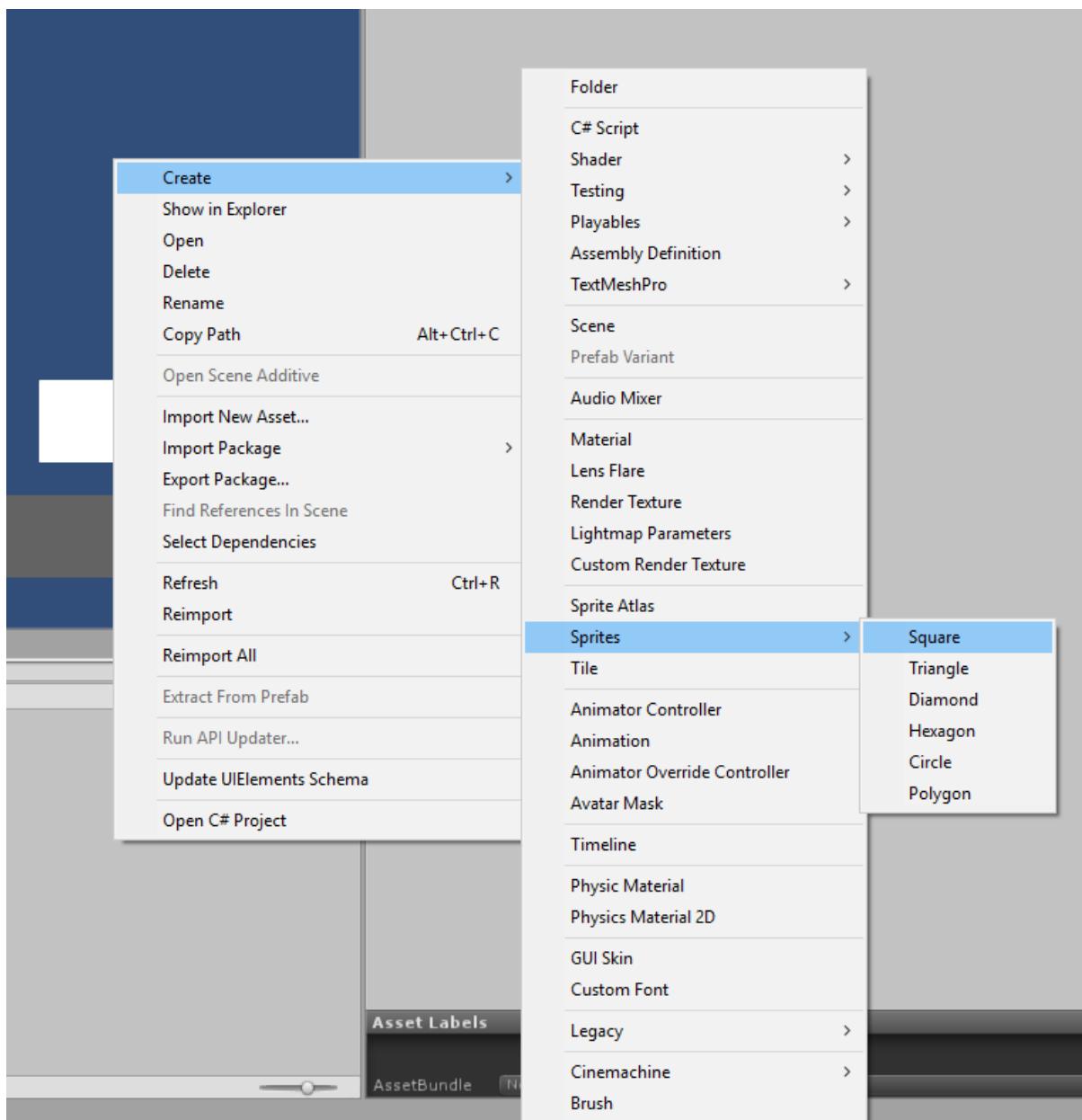


Depois, dentro desta pasta vamos adicionar os arquivos que serão usados nesse exemplo.

O primeiro arquivo é uma sprite que foi criada da seguinte forma:

- Dentro da pasta clique com o botão direito do mouse e nas opções que vão surgir escolha **Create** seguida de **Sprites** e por último **Square**.

Veja:

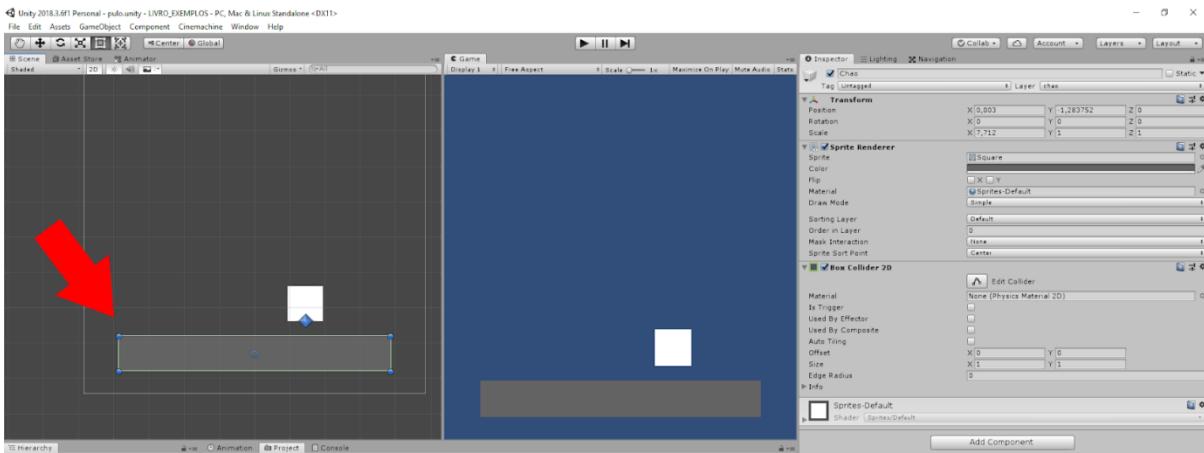


- Agora precisamos criar o arquivo de código do personagem, então para fazer isso, mais uma vez clique com o botão direito do mouse sobre a pasta e escolha a opção **Create** seguida de **C# Script**.

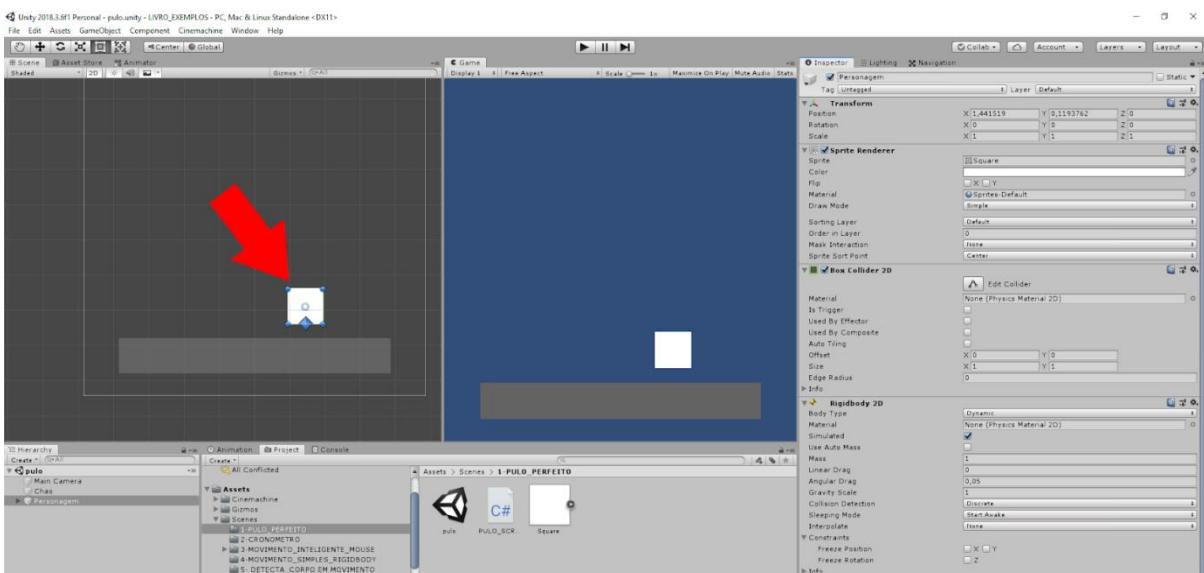


Feito isso é necessário ajustar as sprites em cena, primeiro arraste a sprite para dentro da cena e logo depois ajuste sua escala para que fique maior em seu eixo X.

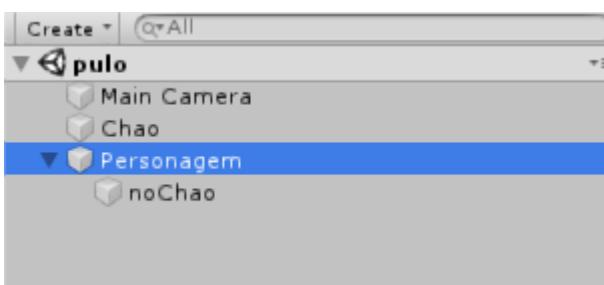
Em seguida adicione um **BoxCollider2D** a essa sprite para obter o seguinte resultado:



Com o chão configurado agora precisamos ajustar o personagem e para fazer isso é muito simples basta puxar a sprite para a cena e simplesmente adicionar um **BoxCollider2D** e um **RigidBody2D** como mostra a imagem abaixo:

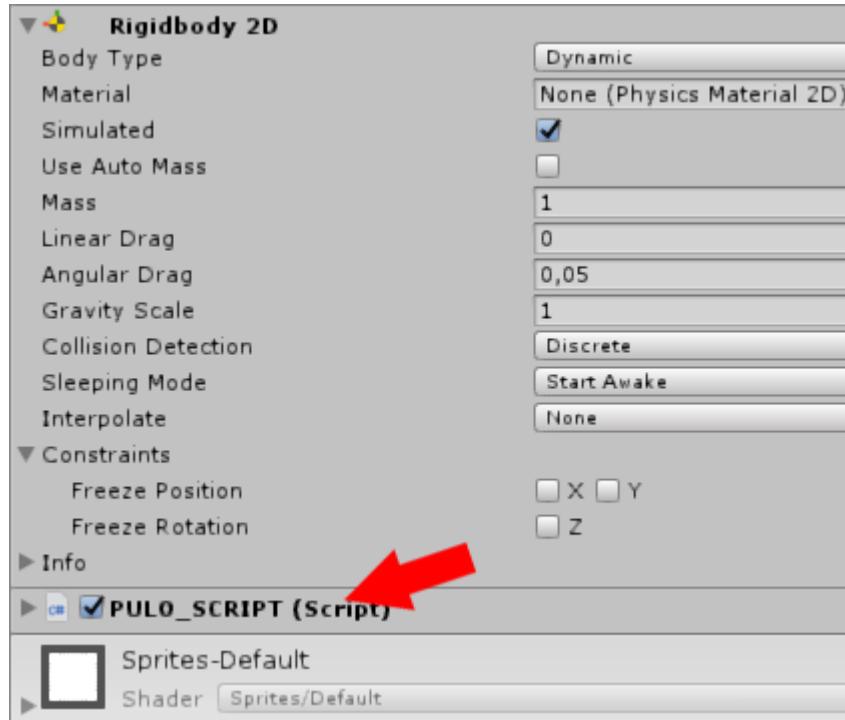


Claro que isso é só o início precisamos fazer mais alguns ajustes como, por exemplo, adicionar o código que vai ser executado nesse objeto e criar um GameObject vazio como filho desse objeto. Veja:



Repare que na imagem acima existe um objeto definido como filho do objeto Personagem é o noChao. Não se preocupe em saber o que esse cada faz, mais adiante vamos ver isso.

Agora adicione o código que criamos ao objeto Personagem dessa forma:



Agora precisamos escrever o código desse arquivo, caso contrário nada vai acontecer. Então clique duas vezes com o botão esquerdo do mouse sobre esse arquivo de código e escreva dentro desse arquivo o seguinte:

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

public class PULO_SCRIPT : MonoBehaviour
{
    private Rigidbody2D rigidbody;
    public float velocidade;
    public float forcaPulo;
    public float moveInput;
    public bool noChao;
    public Transform posPe;
    public float raio;
    public LayerMask chaoMask;
    public float contTempoPulo;
    public float tempoPulo;
    private bool pulando;

    // Start is called before the first frame update
    void Start()
    {
        rigidbody = GetComponent<Rigidbody2D>();
```

```

        }
    }

    // Update is called once per frame
    void Update()
    {
        noChao =
Physics2D.OverlapCircle(posPe.position,raio,chaoMask);

        if(noChao && Input.GetKeyDown(KeyCode.Space))
        {
            pulando = true;
            contTempoPulo = tempoPulo;
            rigidbody.velocity = Vector2.up * forcaPulo;
        }

        if(Input.GetKey(KeyCode.Space) && pulando)
        {
            if(contTempoPulo > 0)
            {
                rigidbody.velocity = Vector2.up * forcaPulo;
                contTempoPulo -= Time.deltaTime;
            }
            else
            {
                pulando = false;
            }
        }

        if(Input.GetKeyUp(KeyCode.Space) )
        {
            pulando = false;
        }
    }

    void FixedUpdate()
    {
        moveInput = Input.GetAxisRaw("Horizontal");
        rigidbody.velocity = new Vector2(moveInput
* velocidade,rigidbody.velocity.y);
    }
}

```

Veja que nesse código inicio criando as variáveis que serão utilizadas no exemplo, como um Rigidbody2D para controlar efeitos físicos.

A variável de velocidade que como o próprio nome diz controla a velocidade que será aplicada no personagem.

Temos a variável `forcaPulo` para controlar a força aplicada no efeito, temos `moveInput` para controlar o movimento horizontal do personagem.

Também temos a variável booleana noChao que verifica se pisamos no chão, a variável posPe é importante na verificação do estado de estar no chão.

A variável raio é usada para definir o raio da esfera de verificação de colisão com o chão, a variável chaoMask nos permite definir com exatidão o layer de quem vamos interagir.

Depois temos variáveis para controle de tempo do pulo assim como uma outra booleana que diz se estamos ou não pulando.

```
private Rigidbody2D rigidbody;
public float velocidade;
public float forcaPulo;
public float moveInput;
public bool noChao;
public Transform posPe;
public float raio;
public LayerMask chaoMask;
public float contTempoPulo;
public float tempoPulo;
private bool pulando;
```

Agora que definimos quais variáveis vão ser utilizadas no exemplo podemos fazer o primeiro ajuste dentro do método Start:

```
void Start()
{
    rigidbody = GetComponent<Rigidbody2D>();
}
```

Veja que dentro desse método apenas passamos para a nossa variável rigidbody o Rigidbody do objeto que contém esse arquivo de código, ou seja, nosso personagem.

Continuando vamos para o método Update ver o que está acontecendo ali.

```
void Update()
{
    noChao =
Physics2D.OverlapCircle(posPe.position, raio, chaoMask);

    if(noChao && Input.GetKeyDown(KeyCode.Space))
    {
        pulando = true;
        contTempoPulo = tempoPulo;
        rigidbody.velocity = Vector2.up * forcaPulo;
    }

    if(Input.GetKey(KeyCode.Space) && pulando)
    {
        if(contTempoPulo > 0)
        {
            rigidbody.velocity = Vector2.up * forcaPulo;
            contTempoPulo -= Time.deltaTime;
        }
    }
}
```

```

        {
            pulando = false;
        }

    if (Input.GetKeyDown(KeyCode.Space))
    {
        pulando = false;
    }
}

```

Veja que a primeira coisa que fizemos dentro desse metodo é garantir que seremos capazes de saber quando o nosso personagem esta pisando no chão.

E isso é feito de uma forma muito simples usando o Physics2D.OverlapCircle, esse cara verifica se um colisor esta dentro de uma area circular dessa forma podemos passar o retorno dessa verificação para uma variavel booleana que vai dizer sim se estamos pisando no chão ou não se não estamos em contato com o solo.

```
noChao = Physics2D.OverlapCircle(posPe.position, raio, chaoMask);
```

Depois dessa importante verificação é necessário criar uma estrutura condicional que verifica justamente se estamos pisando no chão e se apertamos a tecla de Espaço que será responsável por desencadear o pulo.

```

if (noChao && Input.GetKeyDown(KeyCode.Space))
{
    pulando = true;
    contTempoPulo = tempoPulo;
    rigidbody.velocity = Vector2.up * forcaPulo;
}

```

Se a verificação acima for satisfeita veja que ajustamos a variável pulando para verdadeira afinal estamos pulando e logo depois passamos para a variável contTempoPulo o valor que determinaremos mais adiante que é o tempoPulo.

Por último ajustamos a velocidade do nosso rigidbody para que o mesmo possa pular multiplicando a força do pulo pelo nosso Vector2.up.

Agora temos mais uma estrutura condicional que verifica se apertamos a tecla Espaço e se estamos pulando se isso for satisfatório passamos para a próxima verificação que analisa se nossa variável contTempoPulo é maior que zero, se for aplicamos a força de pulo ao nosso rigidbody e decrementamos o valor da variável contTempoPulo para controlar até onde o pulo vai ter força para nos levantar no ar.

Por último temos um else que simplesmente deixa a variável pulando como false.

```

if (Input.GetKey(KeyCode.Space) && pulando)
{
    if (contTempoPulo > 0)
    {
        rigidbody.velocity = Vector2.up * forcaPulo;
        contTempoPulo -= Time.deltaTime;
    }
    else
    {

```

```
        pulando = false;
    }
}
```

Agora ainda dentro do método Update temos mais uma estrutura condicional, nela verificamos se tiramos o dedo da tecla de pulo.

Se isso aconteceu passamos o valor false para a variável pulando.
Com isso fechamos o método Update.

```
if (Input.GetKeyUp(KeyCode.Space))
{
    pulando = false;
}
```

Mas agora temos o movimento horizontal do personagem que é controlado dentro do FixedUpdate, veja:

```
void FixedUpdate()
{
    moveInput = Input.GetAxisRaw("Horizontal");
    rigidbody.velocity = new Vector2(moveInput *
velocidade, rigidbody.velocity.y);
}
```

Veja que nesse método apenas passamos para a variável moveInput nosso axis horizontal e usamos ele para mover o personagem no sentido horizontal usando a velocidade do nosso rigidbody.

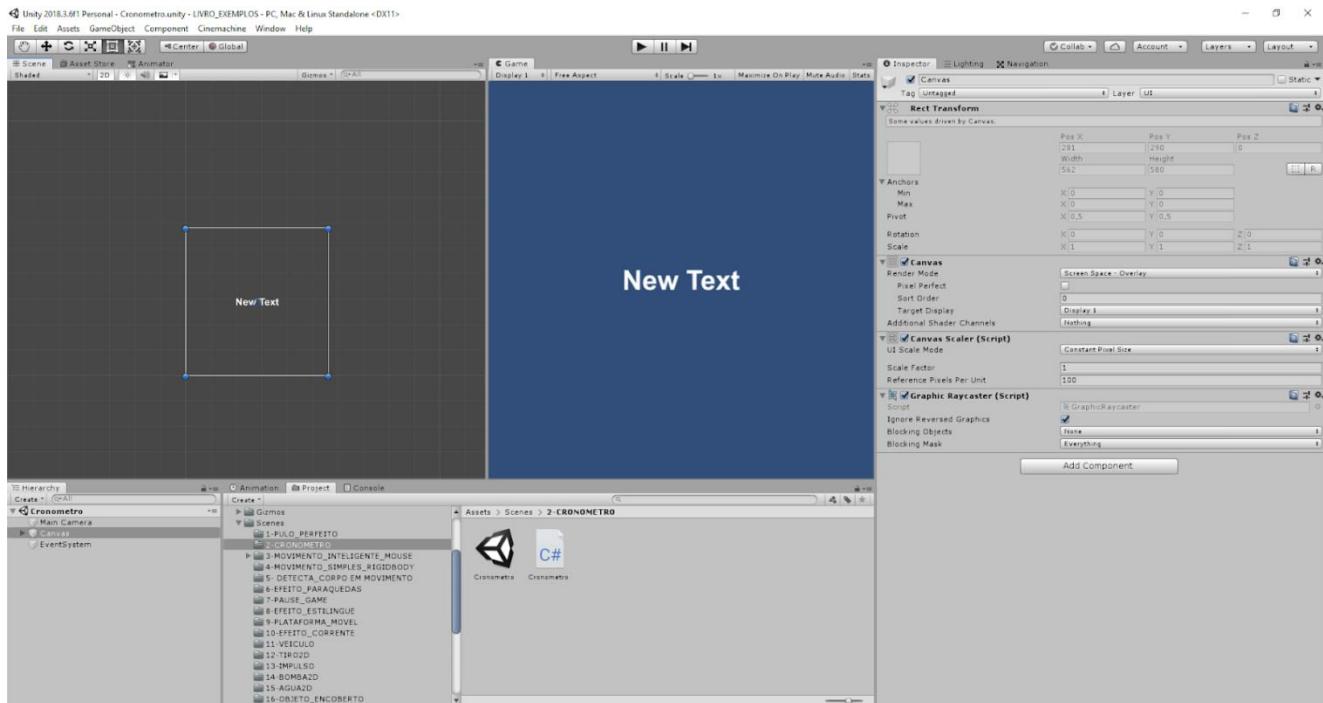
CRONÔMETRO

Outro assunto muito procurado por desenvolvedores é como criar cronômetros, os cronômetros são muito legais e podem ser usados para as mais diversas finalidades.

Seja para cronometrar o tempo de um ataque de vilões, seja para definir o tempo entre dia e noite, tempo de vida do personagem enfim da pra fazer muita coisa legal.

E agora vamos ver como criar um cronômetro simples e muito funcional dentro do Unity.

A primeira coisa que você precisa fazer é criar a sua cena conforme mostra a imagem abaixo:



Veja que essa cena é muito simples, nela temos apenas um Canvas e dentro desse canvas um Text que é responsável por exibir o contador de tempo.

Então com essa estrutura de Canvas definida basta criar um arquivo de código e escrever o seguinte conteúdo dentro dele:

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

public class Cronometro : MonoBehaviour
{
    public Text timerTxt;
    private float tempoInicio;

    // Start is called before the first frame update
    void Start()
    {
        tempoInicio = Time.time;
    }
}
```

```

}

// Update is called once per frame
void Update()
{
    float temp = Time.time - tempoInicio;

    string minutos = ((int) temp / 60).ToString("0#");
    string segundos = ((temp % 60).ToString("0#"));

    timerTxt.text = minutos + ":" + segundos;
}
}

```

Veja que nesse código a primeira coisa que foi feita é escrever a seguinte linha de código:

```
using UnityEngine.UI;
```

Isso é necessário para trabalhar com os elementos da nossa UI.

Logo depois disso dentro da nossa classe Cronometro foi necessário definir as variáveis que serão utilizadas.

```
public Text timerTxt;
private float tempoInicio;
```

Veja que temos apenas duas variáveis uma do tipo Text que serve para manipular nosso objeto Text dentro do Canvas e outra variável float que é o tempo que vamos manipular.

Dentro do método Start é necessário fazer uma pequena atribuição para nossa variável tempoInicio que é justamente o valor de Time.time.

```
void Start()
{
    tempoInicio = Time.time;
}
```

Dessa forma seremos capazes de calcular o tempo dentro do método Update que é o trecho de código abaixo:

```
void Update()
{
    float temp = Time.time - tempoInicio;

    string minutos = ((int) temp / 60).ToString("0#");
    string segundos = ((temp % 60).ToString("0#"));

    timerTxt.text = minutos + ":" + segundos;
}
```

Veja que nesse método criamos uma variável auxiliar chamada de temp e passamos para ela a subtração de Time.time e tempInicio.

Com isso só precisamos calcular os minutos e segundos baseado no valor de temp.

Veja que tenho variáveis do tipo string para me auxiliar nesse processo uma variável tem o nome de minutos e a outra de segundos.

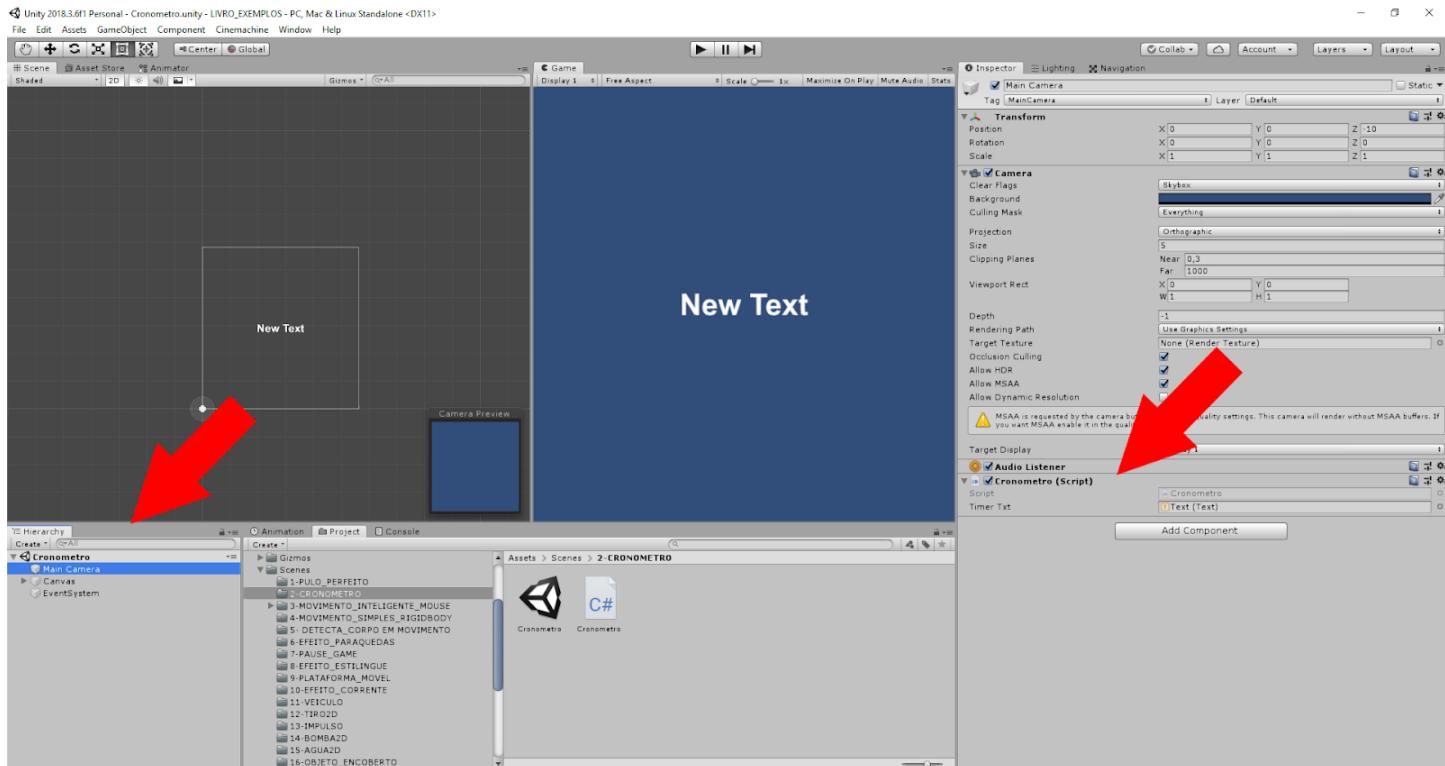
E para obter cada valor existe um cálculo específico, por exemplo para conseguir os minutos é necessário dividir temp por 60 e passar esse valor como uma string para a variável minutos.

Logo depois é necessário calcular segundos usando temp modulo 60 e passando esse valor como uma string também.

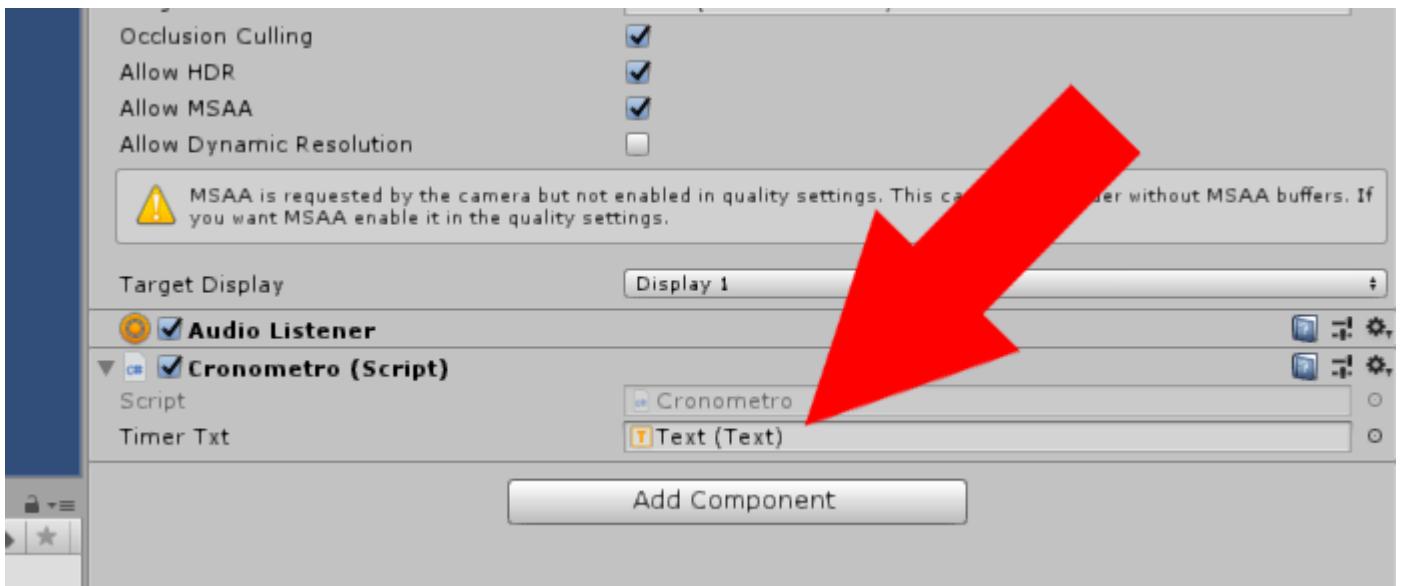
Repare que uso "0#" dentro de ToString isso é feito para que meu efeito visual tenha dois dígitos mesmo quando a contagem usar apenas um dígito por exemplo contagem de 0 até 9.

E por último passamos esses valores para a variável timerTxt para garantir que os valores serão exibidos na tela.

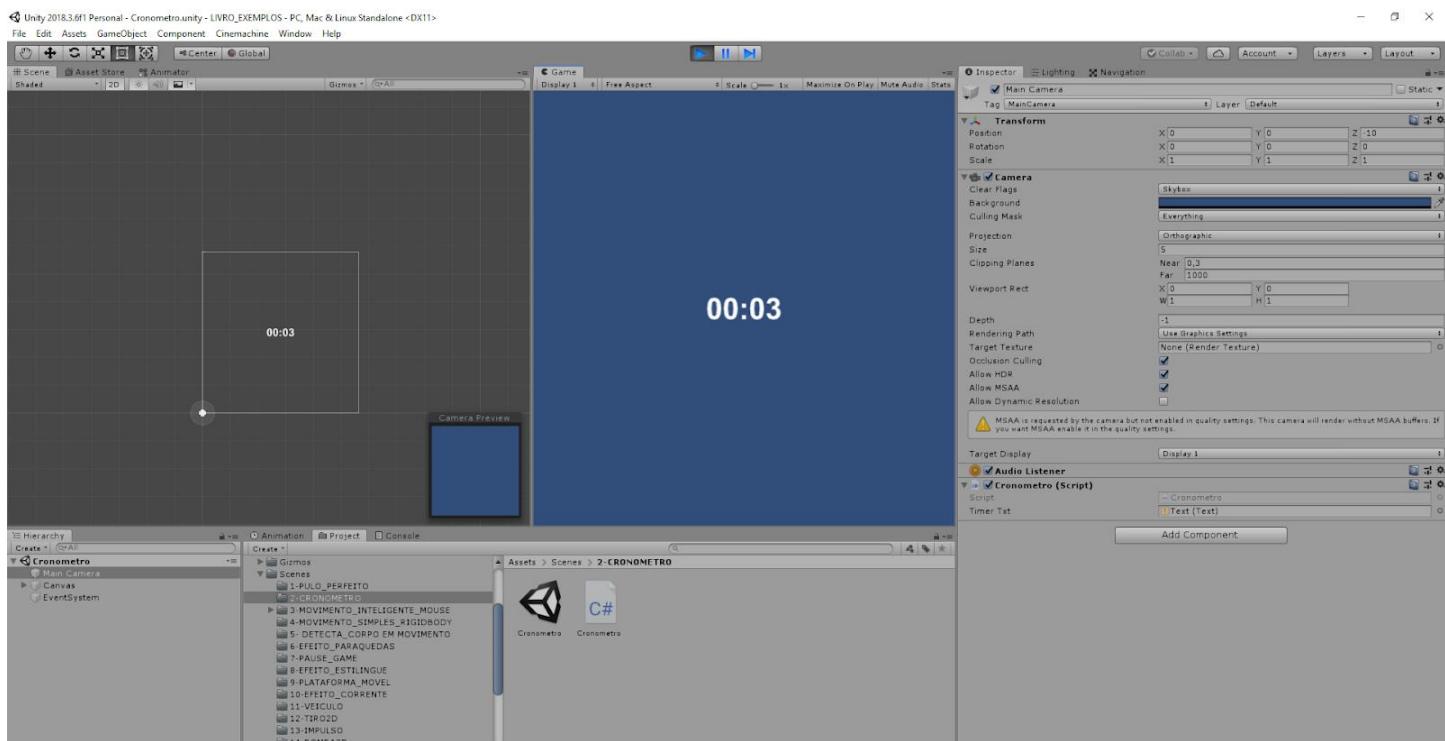
Mas não execute o exemplo ainda precisamos adicionar esse código em algum objeto da cena, no caso ele foi adicionado na câmera



E veja que também foi necessário passar o objeto Text para o nosso código caso contrário teríamos um erro de execução.



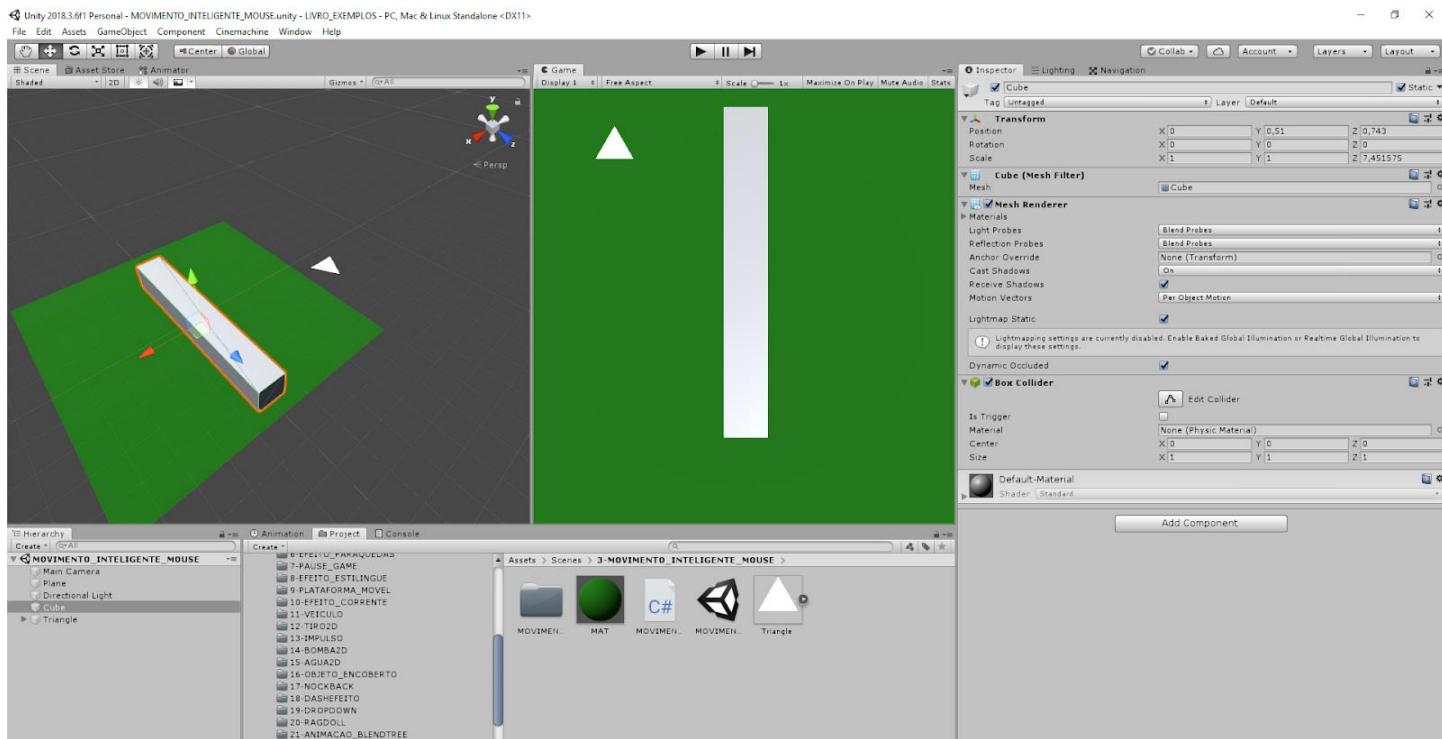
Prontinho depois de tudo isso feito basta executar o exemplo e ver o resultado:



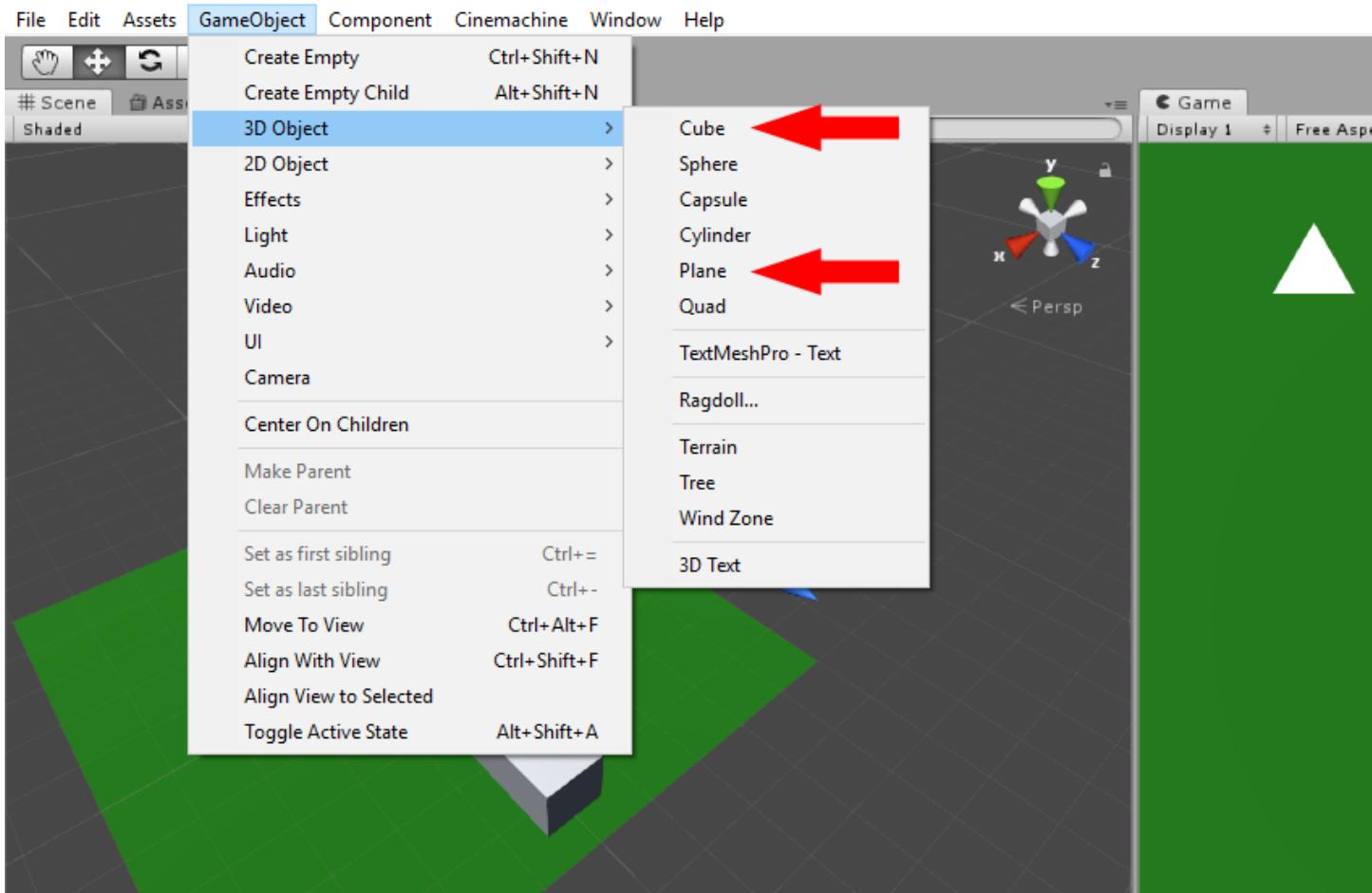
MOVIMENTO INTELIGENTE COM O MOUSE

É muito comum ver desenvolvedores querendo criar movimento inteligente com o mouse, aquele movimento legal que acontece quando você clica com o mouse em algum lugar da cena e o personagem vai até aquele local desviando perfeitamente de todos os obstáculos no caminho. Criar algo desse tipo não é difícil basta utilizar alguns recursos do nosso 3D seja em um projeto 3D mesmo ou escondido dentro de um projeto 2D + 3D.

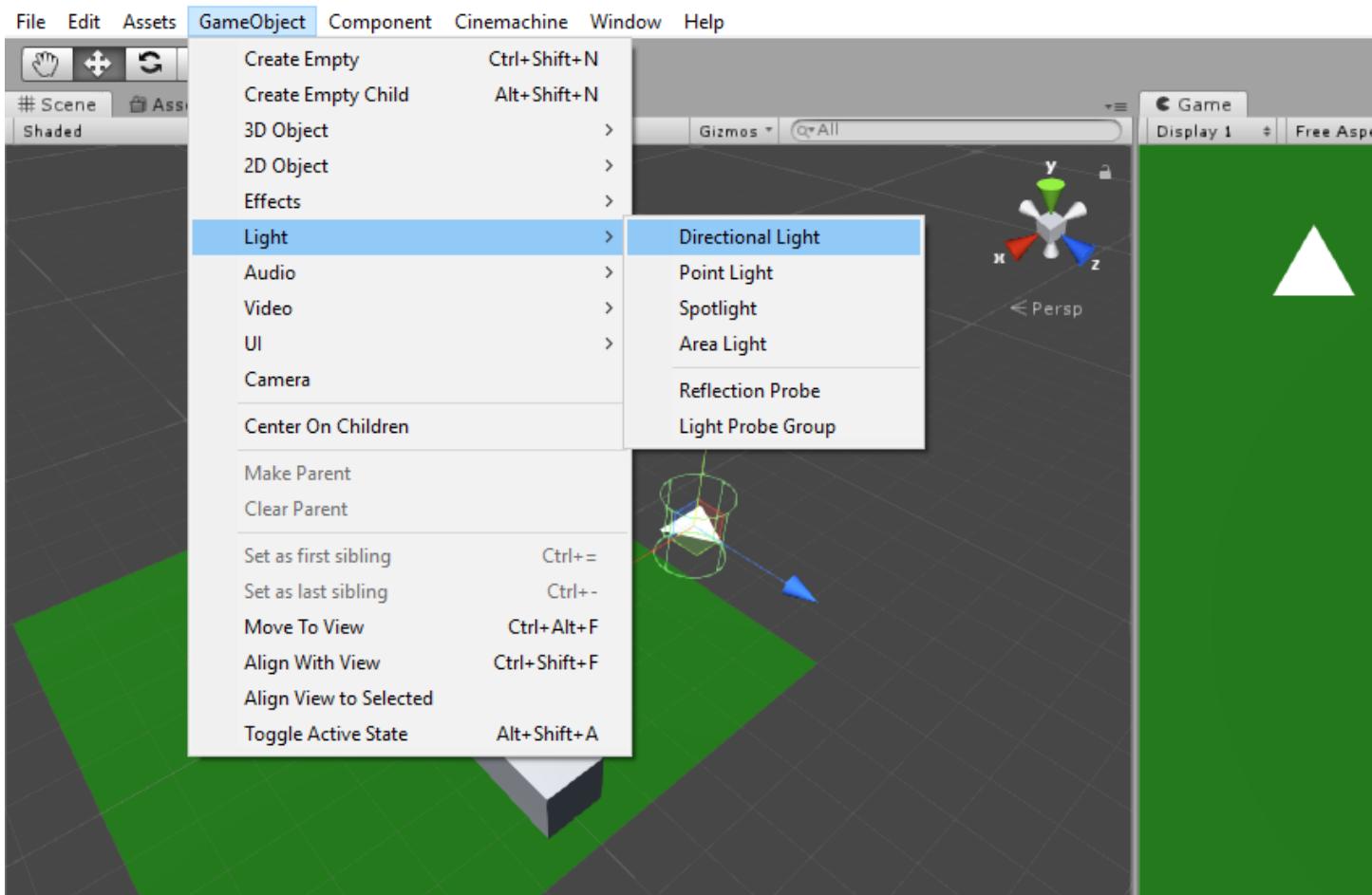
Para isso vamos criar uma cena como a que é apresentada logo abaixo:



Veja que nessa cena temos a mistura de objetos 3D e objetos 2D, no caso o chão e o obstáculo no centro do chão são elementos 3D o triângulo que aparece no canto superior é apenas uma sprite. Para criar os objetos 3D é muito fácil basta clicar no menu GameObject e em 3D Object escolher os objetos necessários.

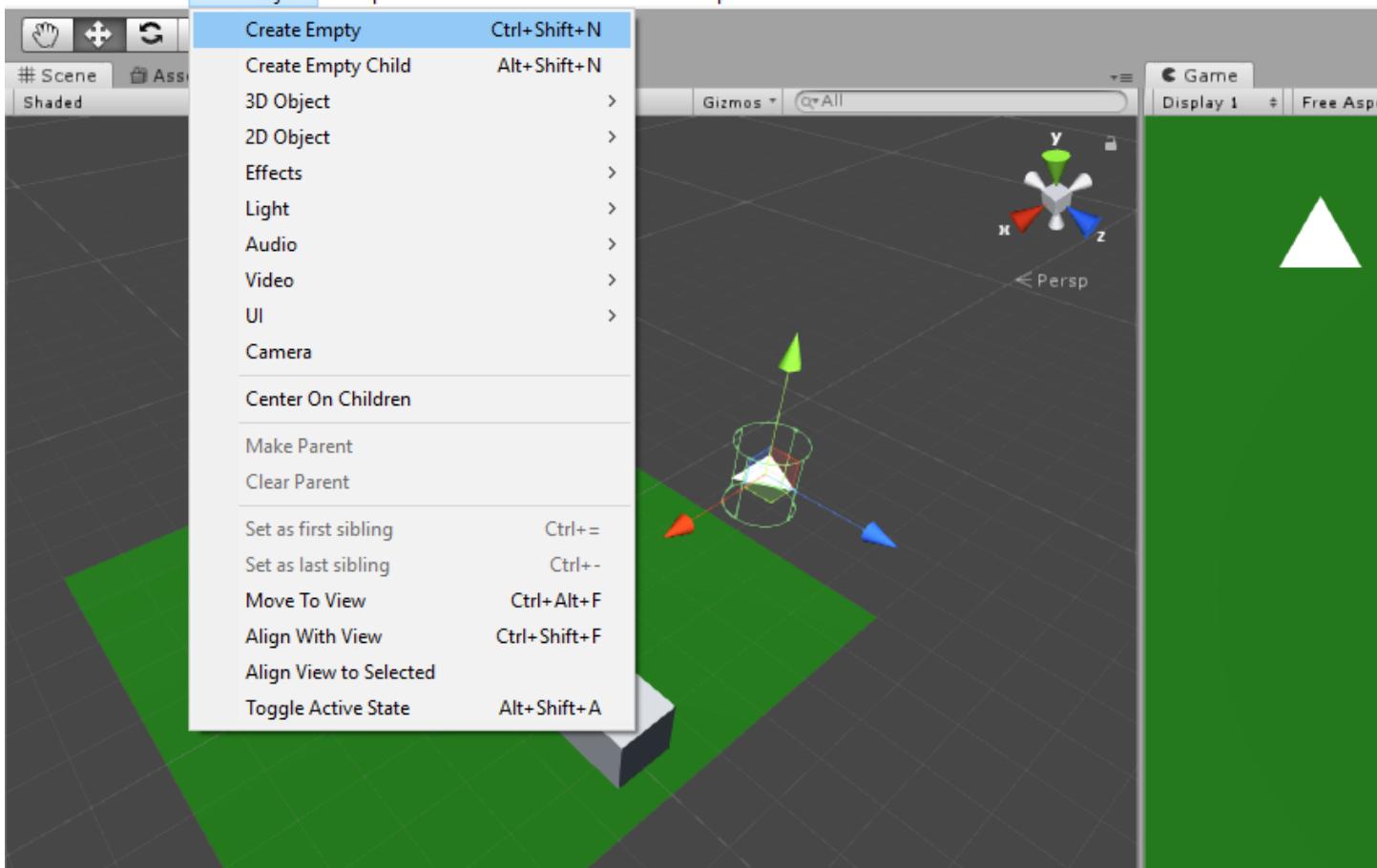


Feito isso você vai perceber que sua cena esta muito escura então crie uma fonte de luz indo a GameObject depois em Light e escolhendo a opção DirectionalLight.
Prontinho agora você vai conseguir ver a cena com mais nitidez.



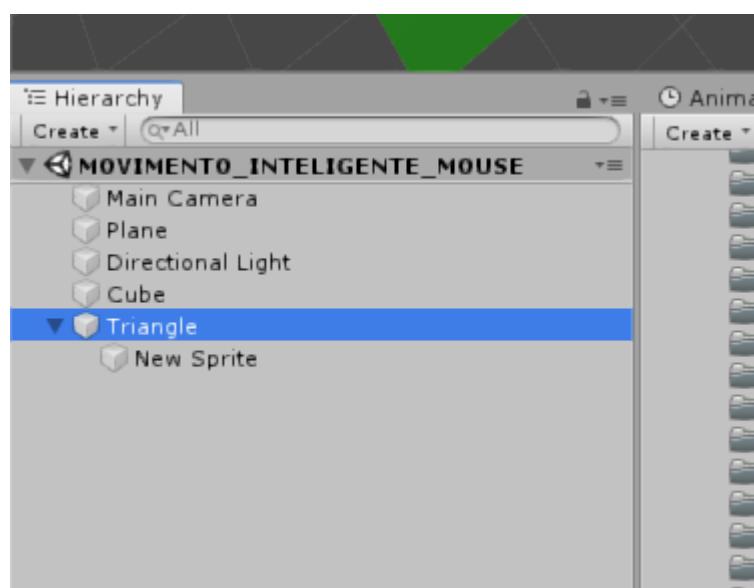
Agora que criamos esses objetos de jogo precisamos criar o objeto que será definido como sendo o personagem desse exemplo.

E para fazer isso a primeira coisa que precisamos fazer é criar um GameObject vazio dentro da cena. Para fazer isso é muito simples basta ir ao menu GameObject e escolher a opção Create Empty, feito isso você já vai ter acesso a um GameObject vazio dentro da sua cena.

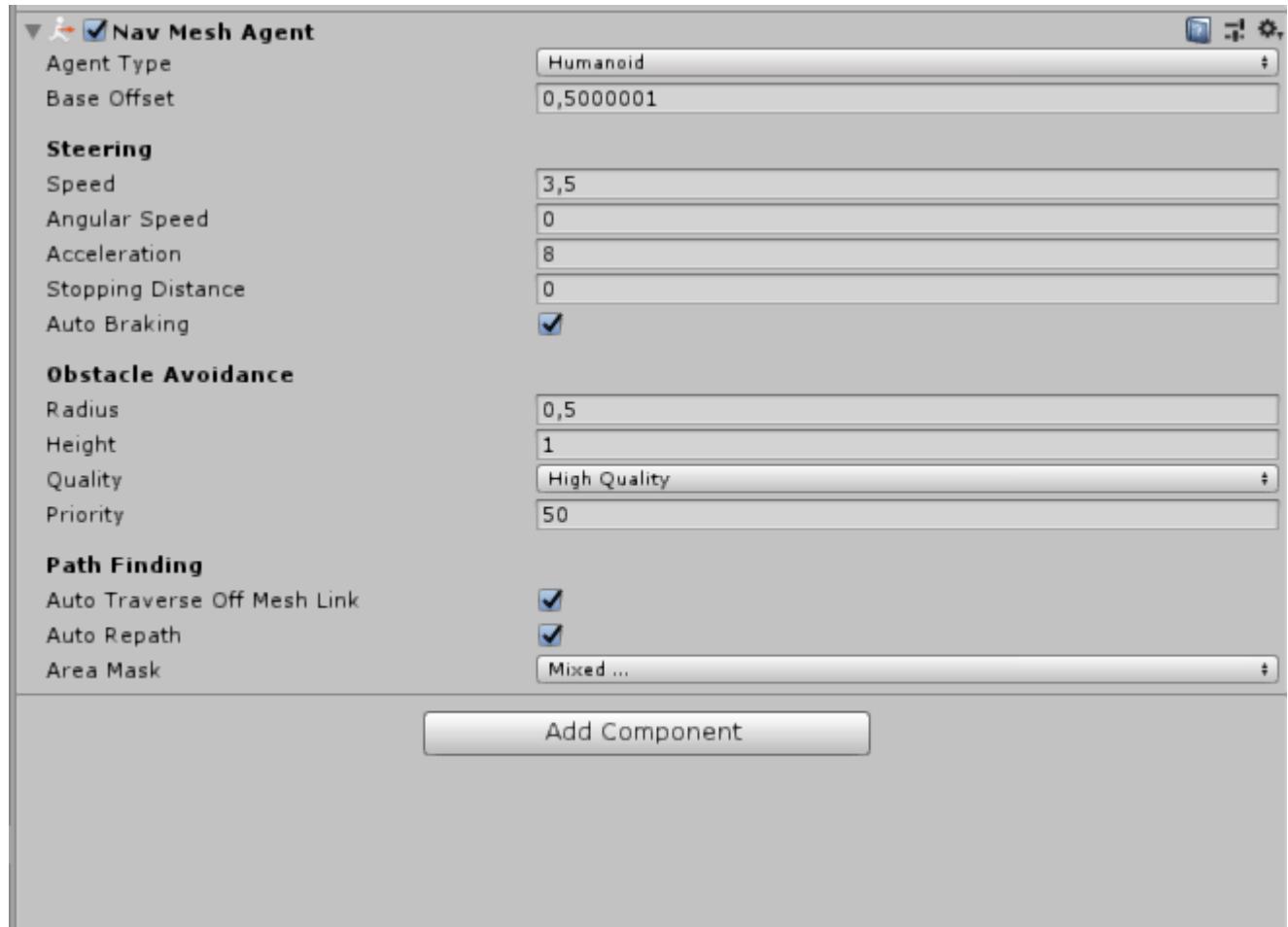


Com todos os elementos em cena agora precisamos configurar esses objetos para que o efeito possa funcionar sem nenhum problema.

Então primeiro selecione o GameObject vazio e adicione dentro dele uma sprite que pode ser criada repetindo o processo que vimos anteriormente.



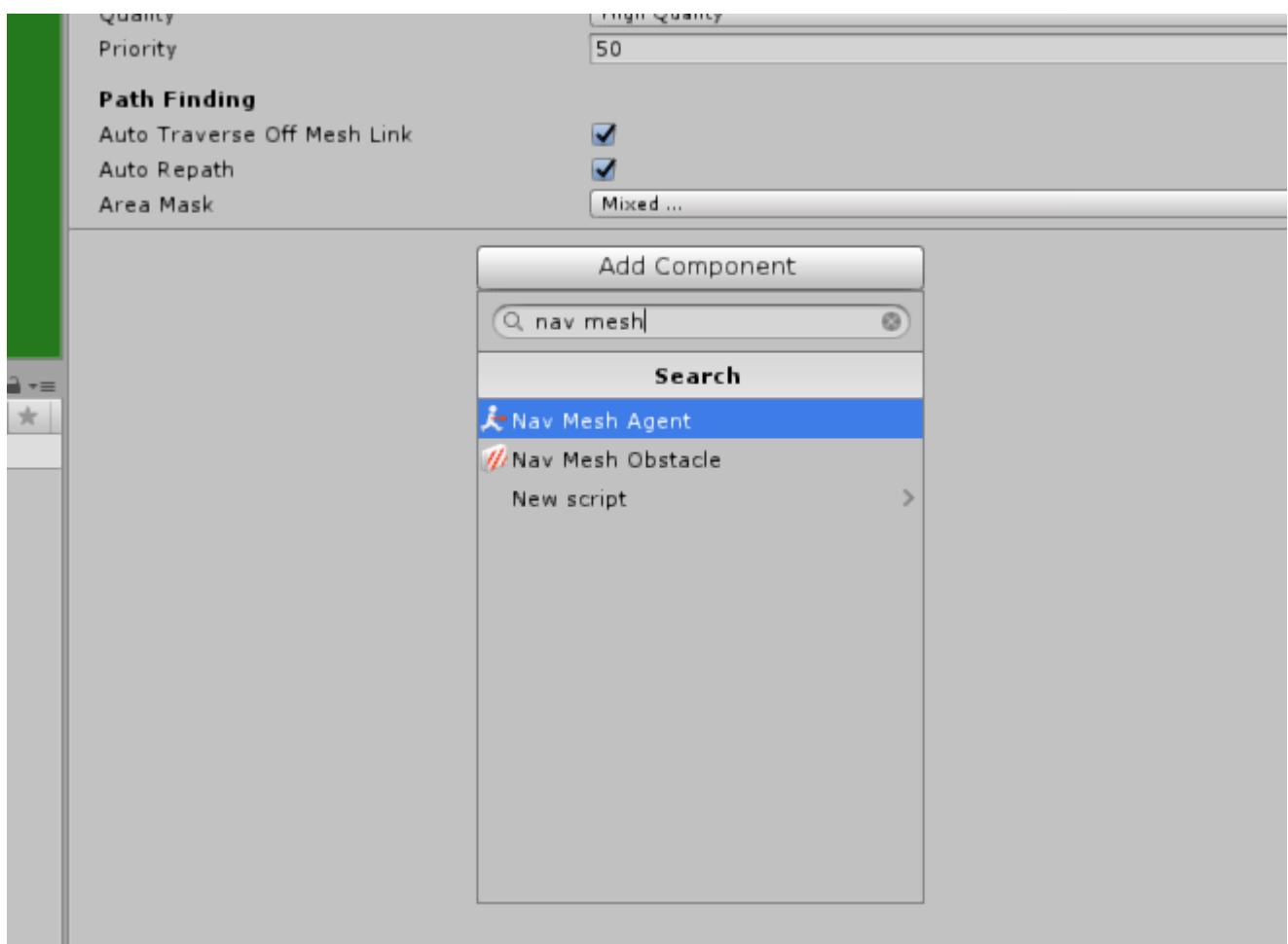
Pronto agora precisamos deixar o objeto vazio que demos o nome de **Triangle** selecionado e ir até o Inspector para adicionar o nav mesh.



Veja que no Inspector temos um **Nav Mesh Agent** adicionado ao nosso objeto isso é necessário para que o mesmo se movimente de forma inteligente pela cena.

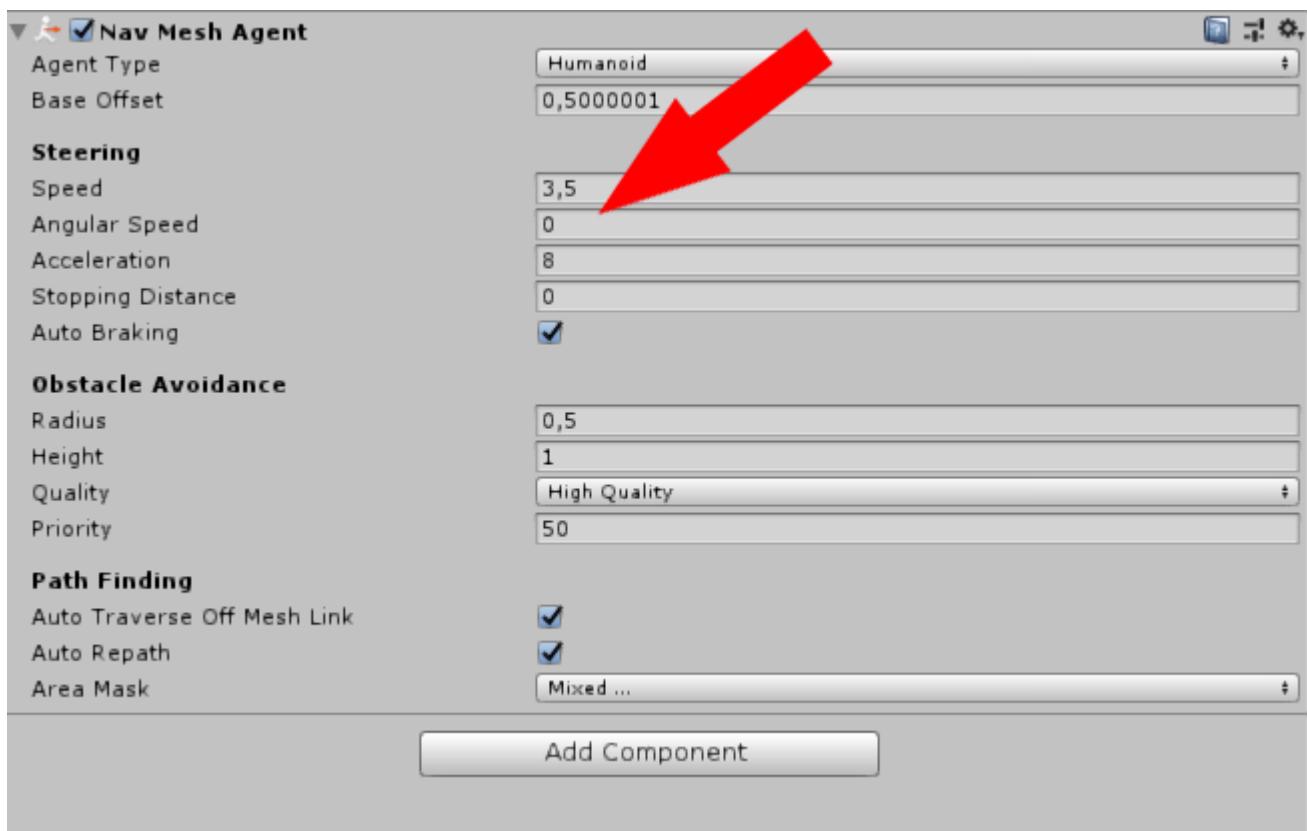
Agora é para adicionar esse componente como devemos proceder?

É muito simples com o objeto selecionado basta clicar sobre o botão **Add Component** e escrever na busca nav mesh.



Pronto veja que a opção já aparece, então agora é só selecioná-la para que a mesma seja adicionada ao objeto.

Muito bom agora temos o nosso **NavMeshAgent** adicionado no objeto que representa o personagem em cena, então vamos ajustar as configurações desse componente dentro do Inspector.

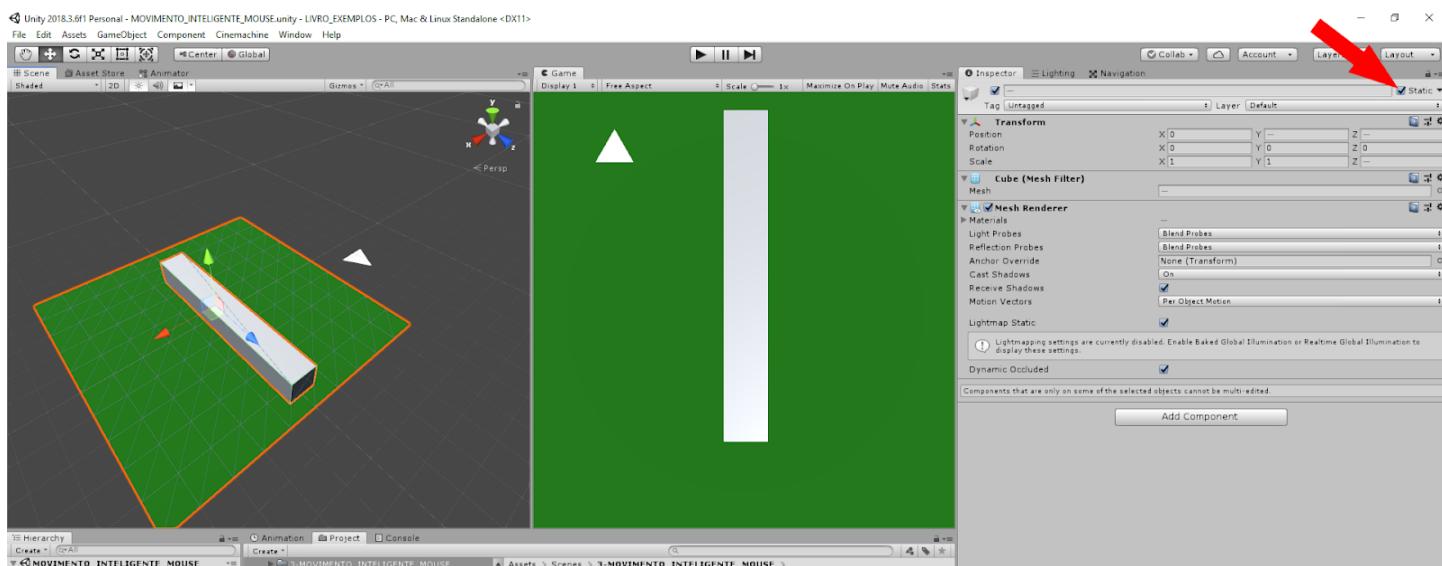


Veja na imagem acima que a única alteração feita foi zerar o **Angular Speed** isso foi feito para não rotacionar o objeto afinal queremos ter uma visualização 2D do efeito então, as rotações desse componente não vão ser interessantes aqui.

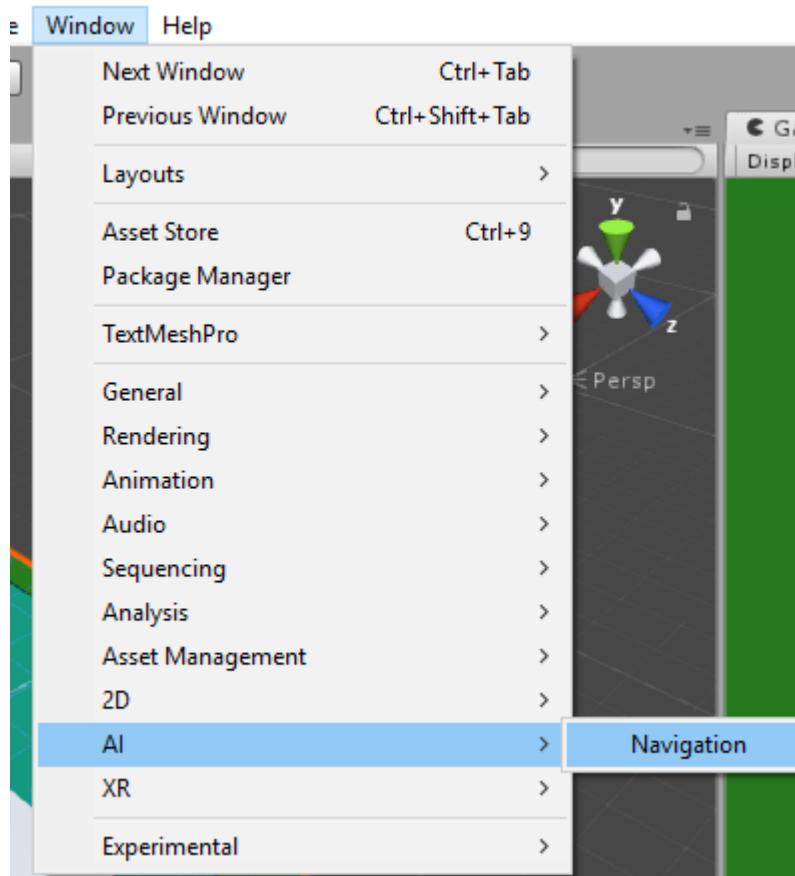
Agora mais um ajuste que precisamos fazer é definir os objetos 3D da cena como sendo Static.

Claro que estou falando de objetos que não vão se mexer durante o exemplo como o chão e o cubo que representa uma espécie de parede ou barreira.

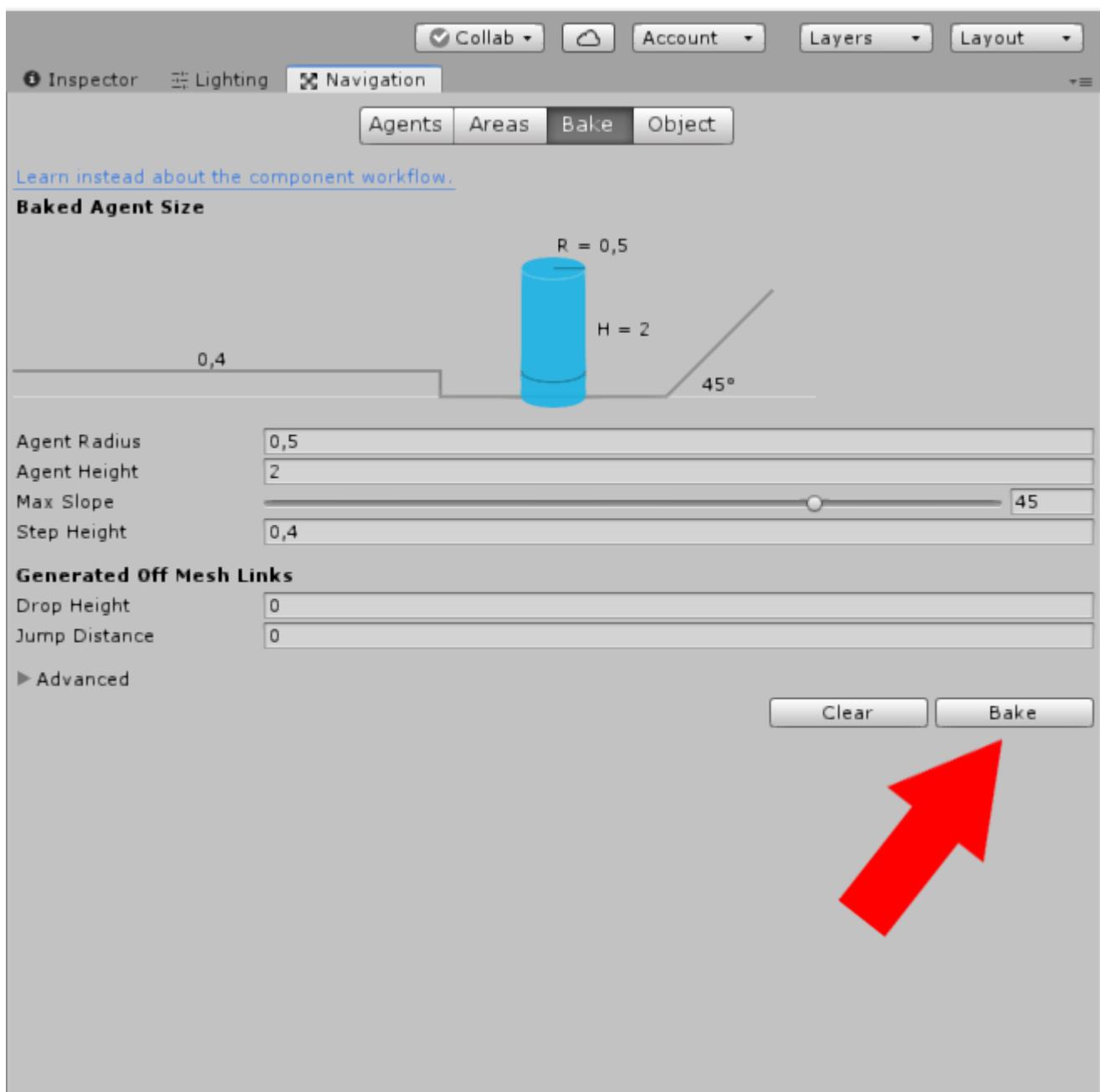
Para definir que esses dois objetos são **Static** basta seleciona-los e depois ligar a opção static na aba **Inspector**.



Agora vamos definir o caminho que o nosso personagem pode utilizar para se movimentar em cena. Para fazer isso precisamos ter acesso a janela **Navigation**, que pode ser adicionada a interface do Unity indo em **Window** depois **AI** e **Navigation**.



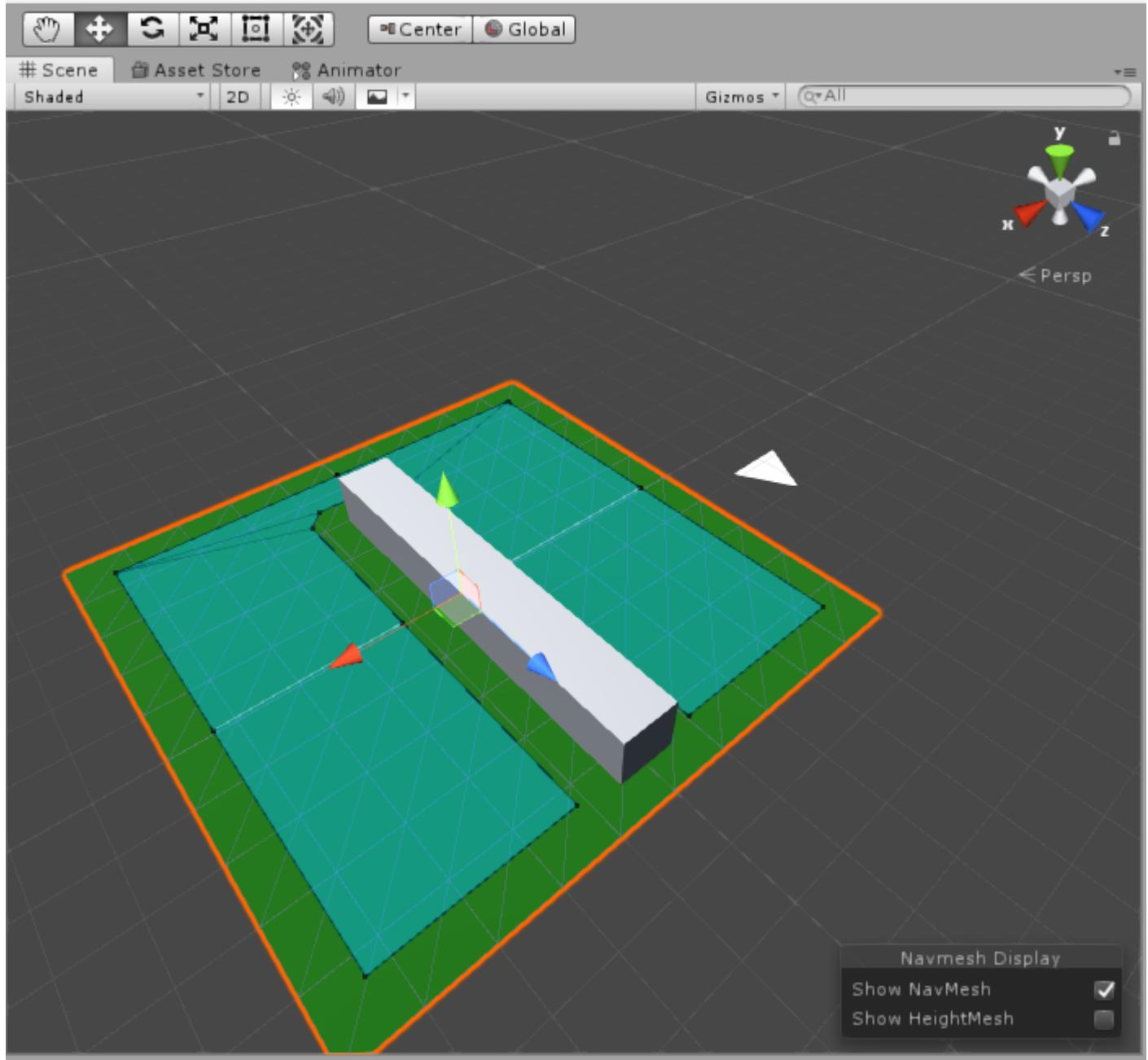
Com isso teremos a seguinte janela para trabalhar:



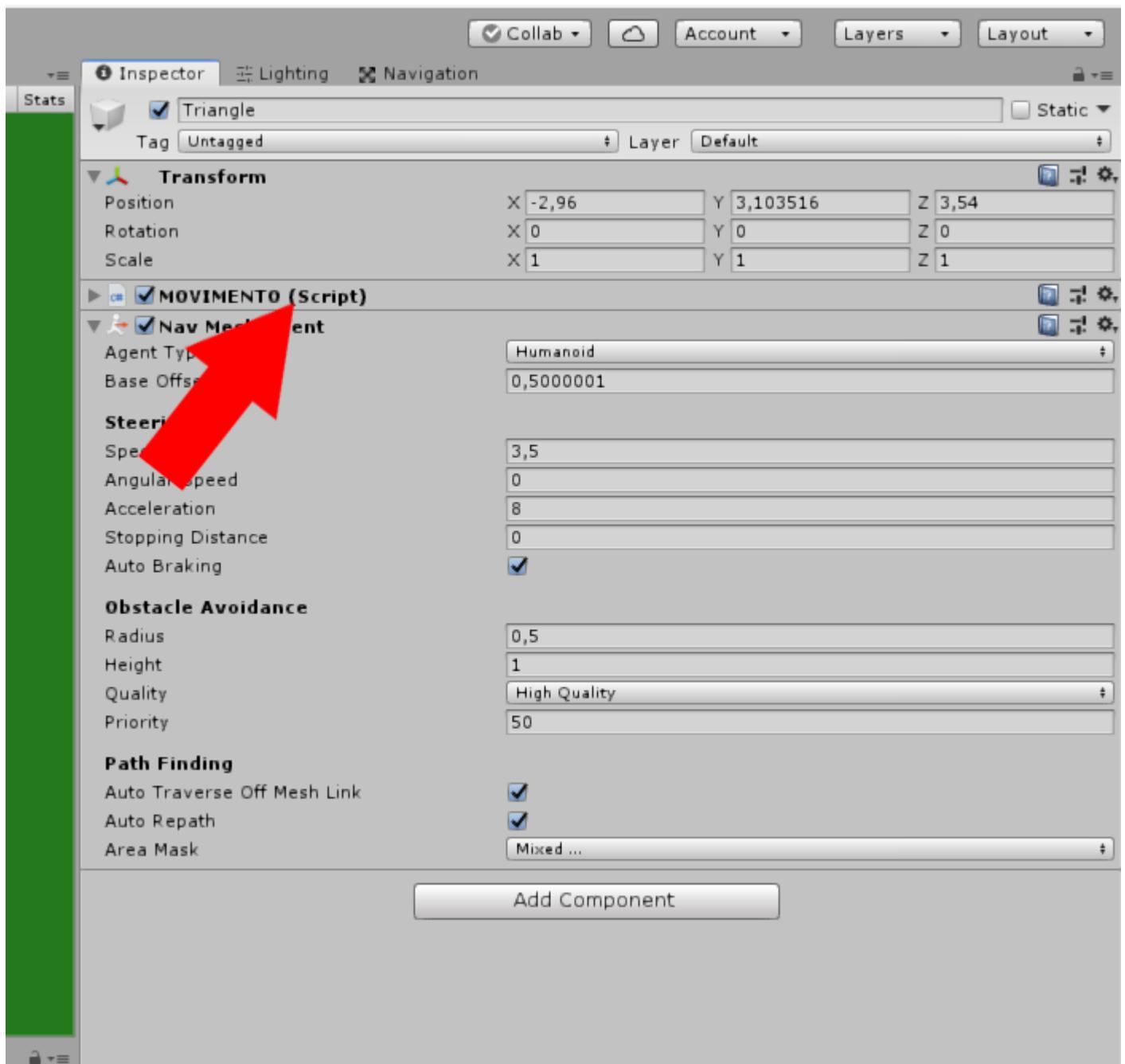
Para criar o nosso caminho basta clicar no botão **Bake** que se encontra no canto inferior direito da janela.

Feito isso os cálculos serão feitos e teremos o caminho definido através de uma malha azul que representa o caminho que pode ser trilhado pelo personagem que tem o **NavMeshAgent**.

Unity 2018.3.6f1 Personal - MOVIMENTO_INTELIGENTE_MOUSE.unity - LIVRO_EXEMPLOS - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Cinemachine Window Help



Maravilha agora que terminamos esses ajustes é necessário criar um arquivo de código e adiciona-lo no objeto que representa o personagem.



E com o arquivo de código criado e adicionado no objeto vamos escrever o código que vai fazer com que esse exemplo funcione.

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

public class MOVIMENTO : MonoBehaviour
{
    public NavMeshAgent agent;
```

```

Vector3 clickPos;
// Start is called before the first frame update
void Start()
{
}

// Update is called once per frame
void Update()
{
    if(Input.GetMouseButtonDown(0))
    {

        Ray raio =
Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        if(Physics.Raycast(raio,out hit))
        {
            clickPos = hit.point;
        }
        agent.SetDestination(clickPos);
    }
}
}

```

Certo veja que o código é pequeno e muito simples, só precisamos reparar nas pequenas coisas como por exemplo a adição da seguinte linha de código:

```
using UnityEngine.AI;
```

Essa linha de código nos possibilita trabalhar com o NavMeshAgent então, sua presença é mais que necessária.

Depois precisamos definir as variáveis que vamos usar que no caso são duas, uma do tipo NavMeshAgent que nos possibilita trabalhar diretamente sobre o objeto que contém esse componente e logo depois uma variável do tipo Vector3 que serve para definir um local de click de mouse para indicar onde o NavMeshAgent precisa ir.

```

public NavMeshAgent agent;
Vector3 clickPos;
```

Por último dentro do método Update criamos uma estrutura condicional que verifica antes de qualquer coisa se clicamos o botão esquerdo do mouse se isso aconteceu passamos para a próxima parte que é a criação de um raio que vai da câmera até um ponto onde clicamos na tela.

Com isso conseguimos verificar se houve alguma colisão no processo e dessa forma passamos o ponto de impacto para a variável clickPos que será usada sendo o destino do nosso NavMeshAgent.

```

void Update()
{
    if(Input.GetMouseButtonDown(0))
```

```

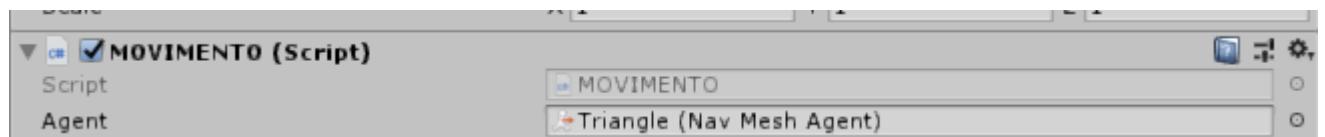
    {

        Ray raio =
Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        if(Physics.Raycast(raio,out hit))
        {
            clickPos = hit.point;
        }
        agent.SetDestination(clickPos);
    }
}

```

Prontinho agora para fechar precisamos passar o NavMeshAgent para o nosso código dentro do Inspector veja:



Com isso tudo vai funcionar.

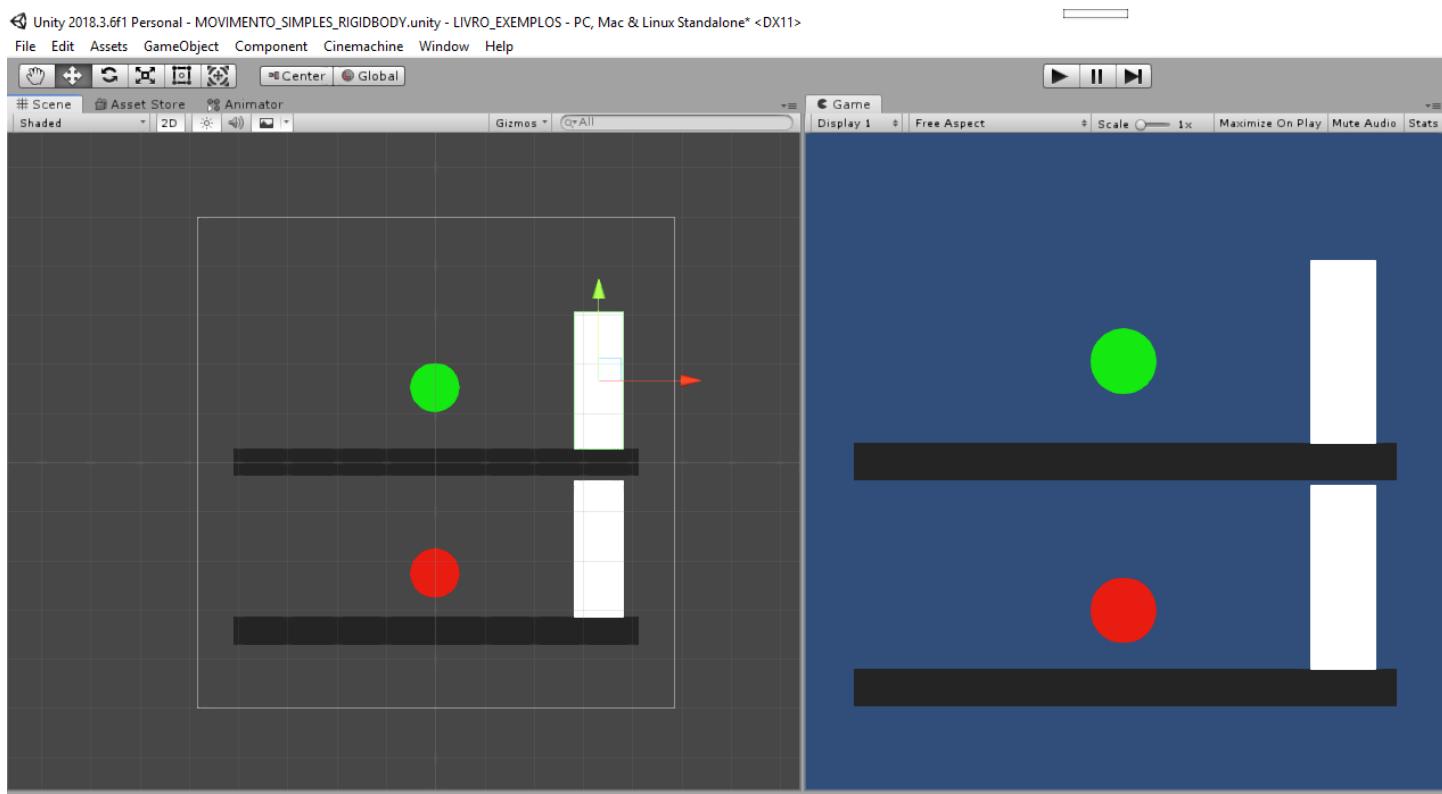
MOVIMENTO SIMPLES E COM RIGIDBODY

Outro assunto que ferve a cabeça de quem está iniciando no desenvolvimento de jogos é o movimento de personagens.

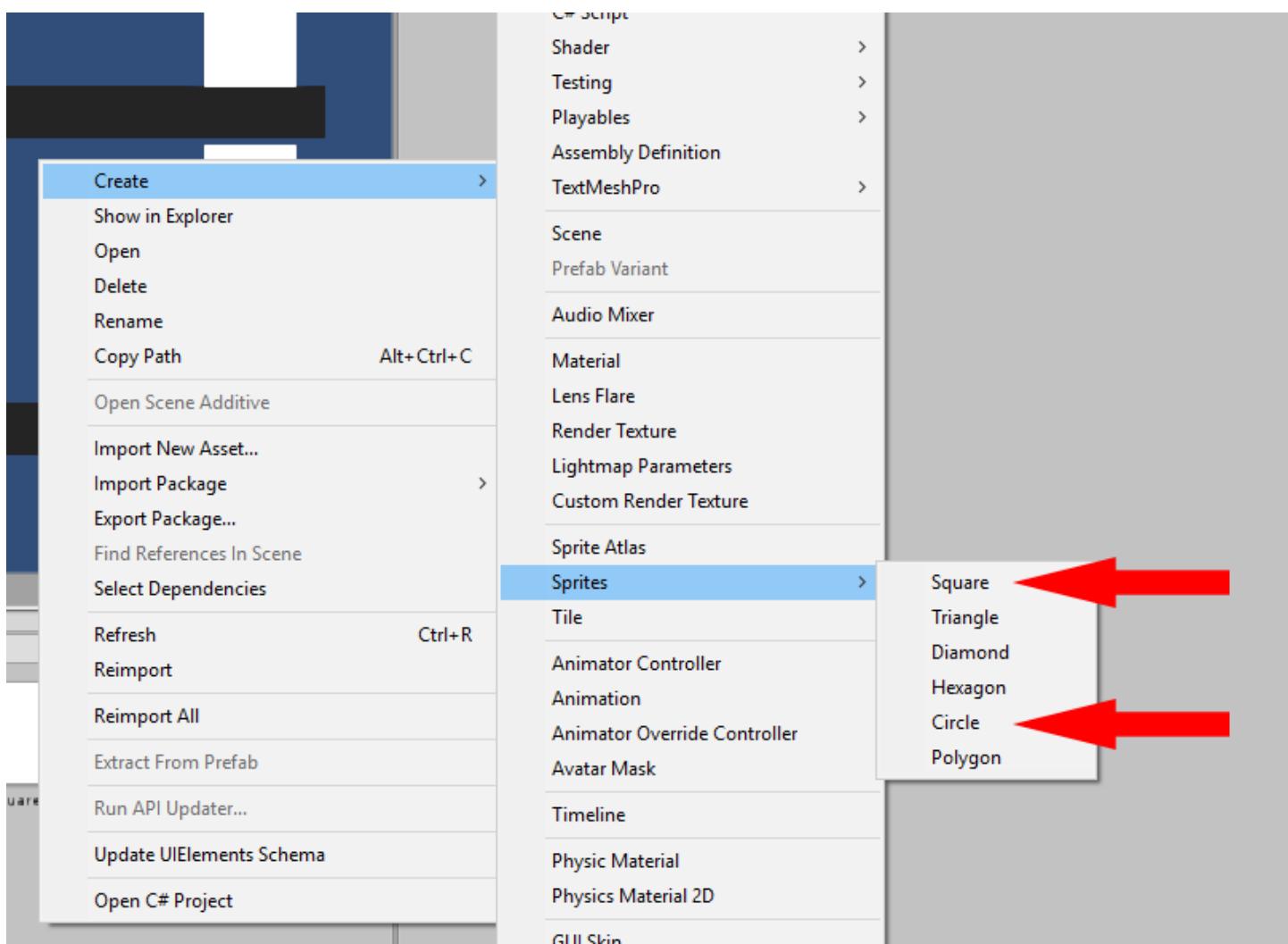
E isso é comum afinal conseguimos mover objetos em cena utilizando mais um método.

Nesse exemplo eu vou mostrar como movimentar um objeto usando o `Transform.Translate` e depois usando o `Rigidbody2D.velocity`.

Mas antes de qualquer coisa precisamos criar uma cena como a que aparece abaixo:



Veja que para criar essa cena não existe segredo basta criar dois tipos de sprites uma esférica e outra quadrada.

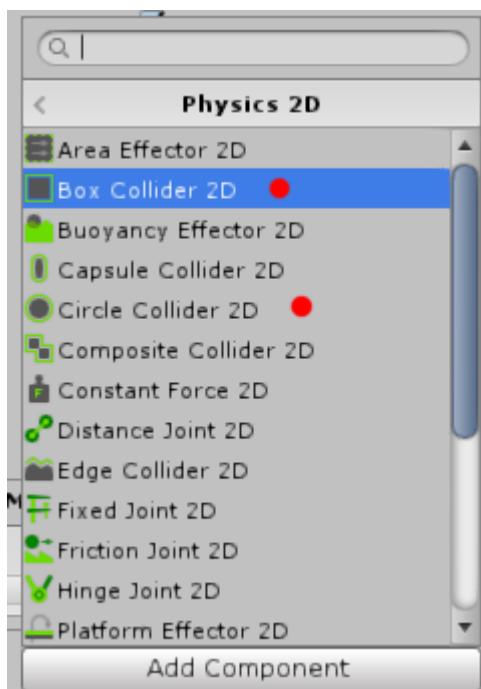


Depois arraste essas sprites para a cena e ajuste a escala e cor das sprites quadradas para que o seu resultado seja parecido com o apresentado aqui.

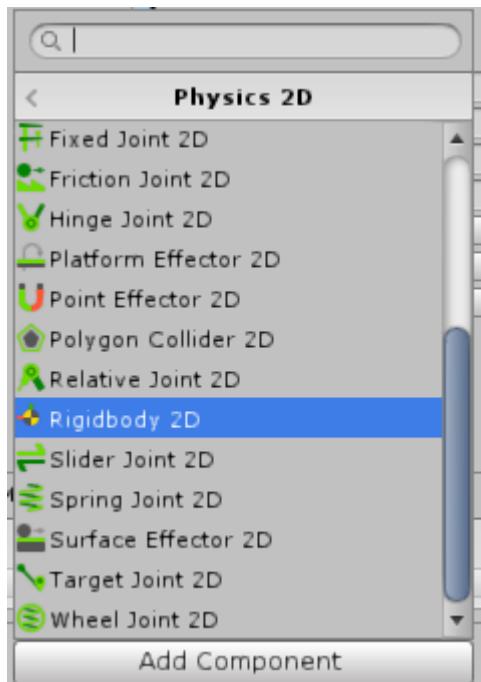
E depois arraste as sprites esféricas mudando apenas a sua cor.

Com isso basta adicionar corpos colisores a essas sprites lembrando que as sprites esféricas vão ter corpos colisores esféricos e sprites quadradas corpos colisores quadrados.

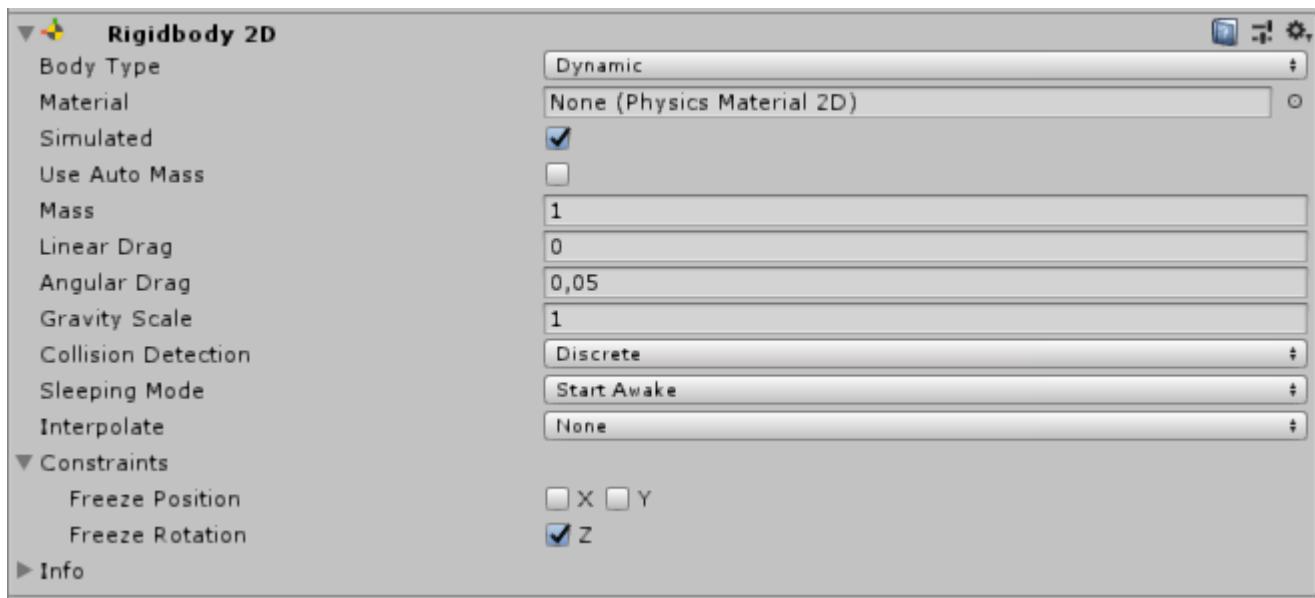
Para adicionar esses corpos colisores é simples basta selecionar a sprite em cena e depois no Inspector clicar sobre o botão Add Component opção Physics e escolher os colisores.



Pronto todos os objetos de cena tem um corpo colisor ideal, mas precisamos fazer mais um ajuste pois vamos movimentar as sprites esféricas então, precisamos adicionar um Rigidbody a elas. Para fazer isso é o mesmo processo de adição de corpo colisor mas nesse caso vamos escolher o Rigidbody2D.



Dessa forma nossas esferas vão ter o seguinte componente adicionado:



Maravilha! já ajustamos a nossa cena agora precisamos criar os arquivos de código que vão fazer com que cada esfera se movimente da mesma forma porém usando métodos diferentes.

Vamos iniciar com a criação do arquivo de código com o movimento transform.Translate.

Crie um arquivo de código com o nome de **MOVIMENTO_TRANSFORM** e dentro dele escreva o seguinte:

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

public class MOVIMENTO_TRANSFORM : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float velocidade = Input.GetAxisRaw("Horizontal") * 20 * Time.deltaTime;
        transform.Translate(velocidade, 0, 0);
    }
}
```

Veja que esse código de movimento é muito simples tudo o que é necessário para fazer com ele funcione esta dentro do método Update.

Primeiro uma variável com o nome de velocidade que recebe o input horizontal ou seja as teclas A e D ou setas para direita e esquerda.

Esse valor é passado depois de multiplicado por 20 e por Time.deltaTime.

Feito isso basta passar o valor calculado de velocidade para o transform.Translate que cuida do movimento do objeto em cena.

No caso veja que o movimento acontece apenas no eixo X por essa razão os últimos valores dentro do Translate são zerados pois não queremos mexer nos valores de Y e Z.

Pronto o primeiro método de movimento está finalizado agora vamos passar para o segundo.

Para esse exemplo crie outro arquivo de código mas agora com o nome de **MOVIMENTO_RIGIDBODY** e dentro dele adicione o seguinte:

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

public class MOVIMENTO_RIGIDBODY : MonoBehaviour
{
    public Rigidbody2D rigidbody;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void FixedUpdate()
    {
        float velocidade = Input.GetAxisRaw("Horizontal");

        if(velocidade > 0)
        {
            rigidbody.velocity = new Vector2(20,rigidbody.velocity.y);
        }
        else if(velocidade < 0)
        {
            rigidbody.velocity = new Vector2(-20,rigidbody.velocity.y);
        }
    }
}
```

Veja que nesse código a coisa muda um pouco à complexidade aumente porém, de uma forma leve. A primeira coisa que foi feita aqui é criar uma variável do tipo Rigidbody2D

```
public Rigidbody2D rigidbody;
```

Essa variável vai nos ajudar na movimentação do objeto nos proporcionando a possibilidade de mexer na velocidade do objeto.

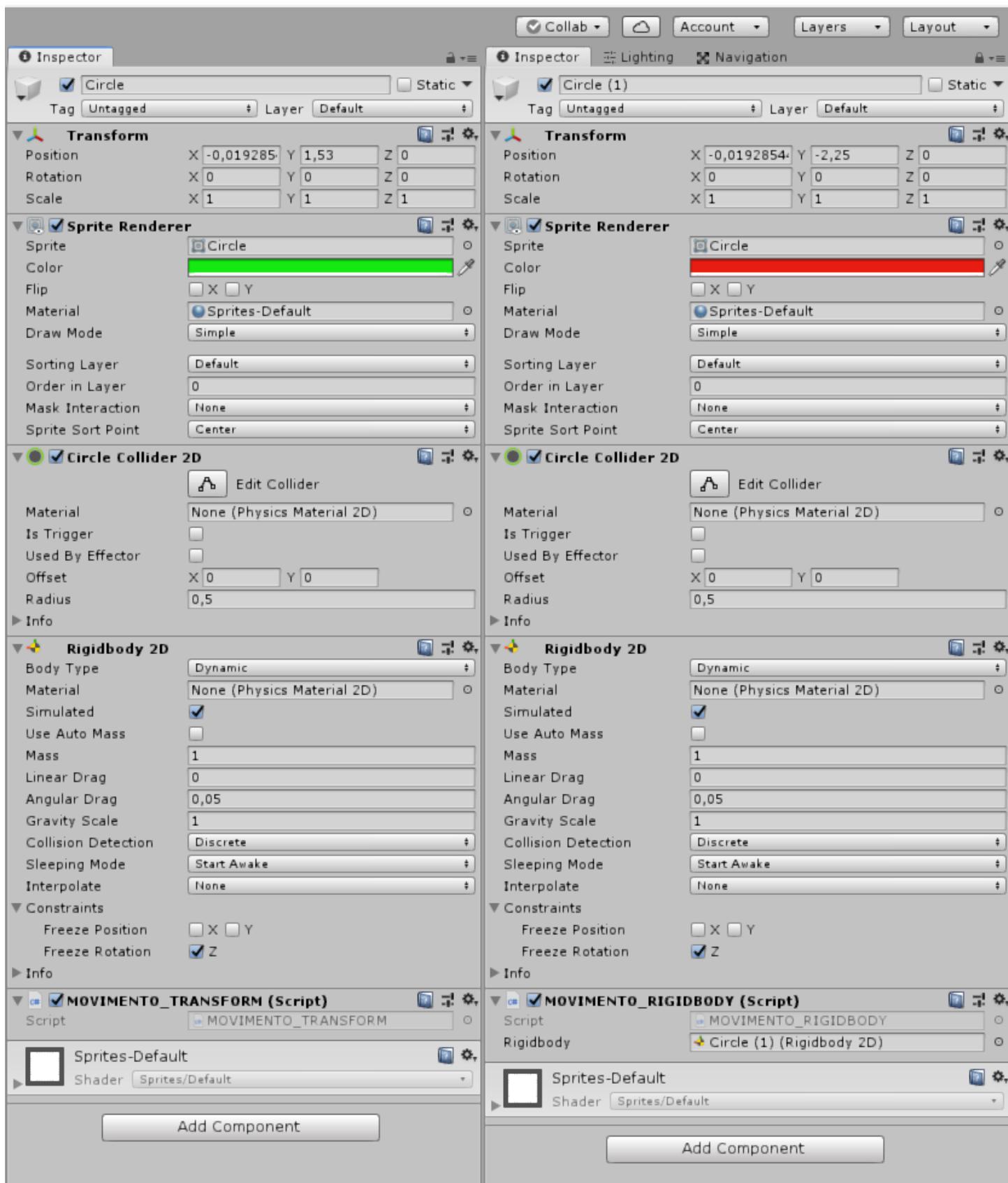
Logo depois temos dentro do método **FixedUpdate** à estrutura condicional que vai movimentar o nosso objeto, veja que aqui passamos nosso input horizontal para a variável velocidade e logo depois

verificamos o valor que essa variável pode ter, se for maior que zero ajustamos a velocidade do objeto de forma positiva, se for menor que zero ajustamos a velocidade de forma negativa.

```
void FixedUpdate()
{
    float velocidade = Input.GetAxisRaw("Horizontal");

    if(velocidade > 0)
    {
        rigidbody.velocity = new Vector2(20,rigidbody.velocity.y);
    }
    else if(velocidade < 0)
    {
        rigidbody.velocity = new Vector2(-
20,rigidbody.velocity.y);
    }
}
```

Prontinho com esses ajustes meu objeto já pode se movimentar.
Então vamos para o Unity fechar os últimos ajustes desse exemplo.

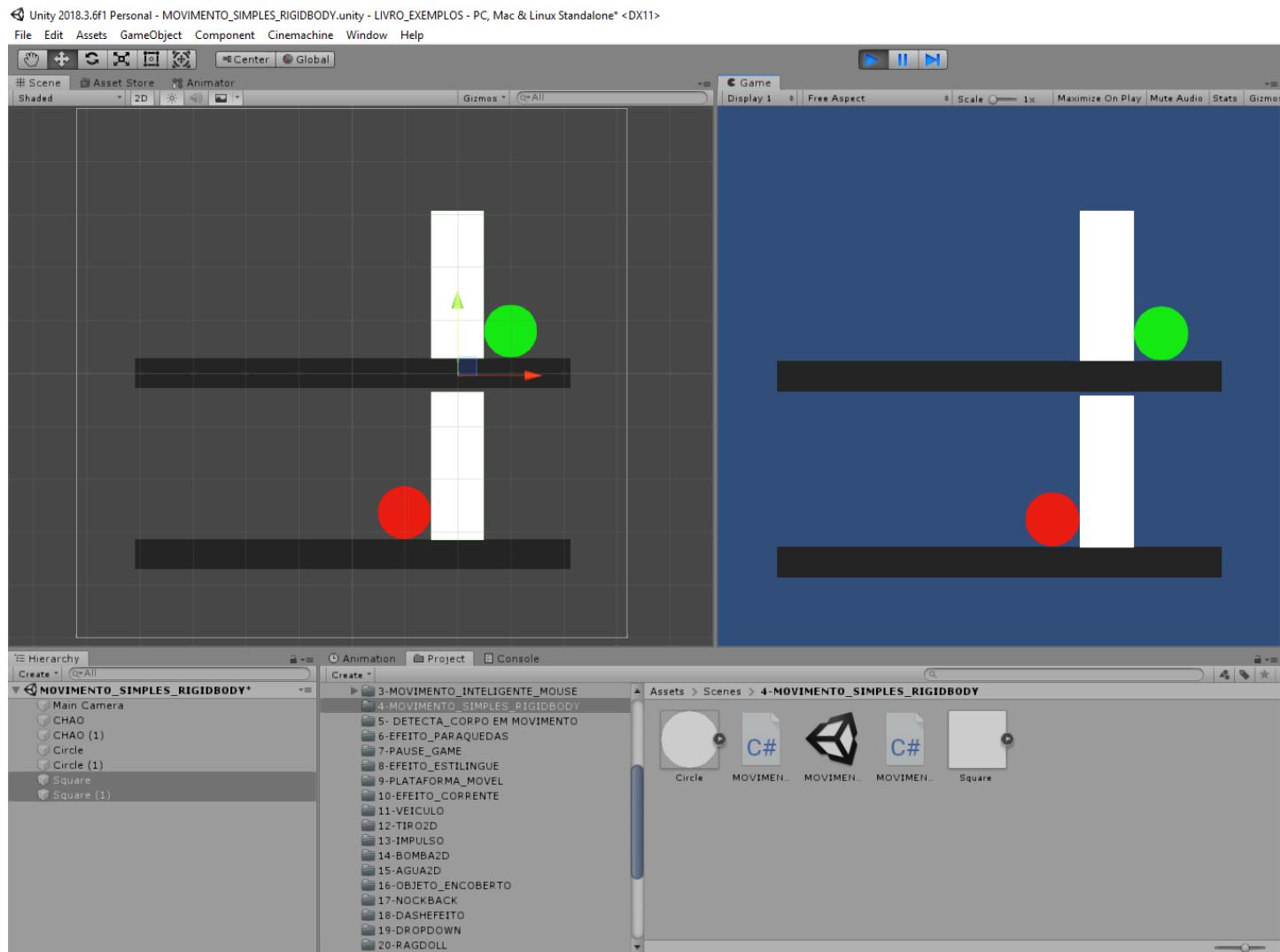


Veja que estou exibindo na imagem acima o Inspector de cada esfera e perceba que em cada uma tenho um código de movimento.

Na da esquerda o movimento pelo Transform e na direita o movimento pelo Rigidbody, no caso do movimento pelo Rigidbody é necessário passar o Rigidbody do objeto para o slot vazio do código caso contrário teremos um erro de execução.

Feito isso execute o exemplo e veja que as duas esferas vão se movimentar igualzinho mas com uma diferença muito importante.

O movimento por Transform vai furar as colisões com muita facilidade já o movimento por Rigidbody te dá uma segurança muito maior nessa questão veja:



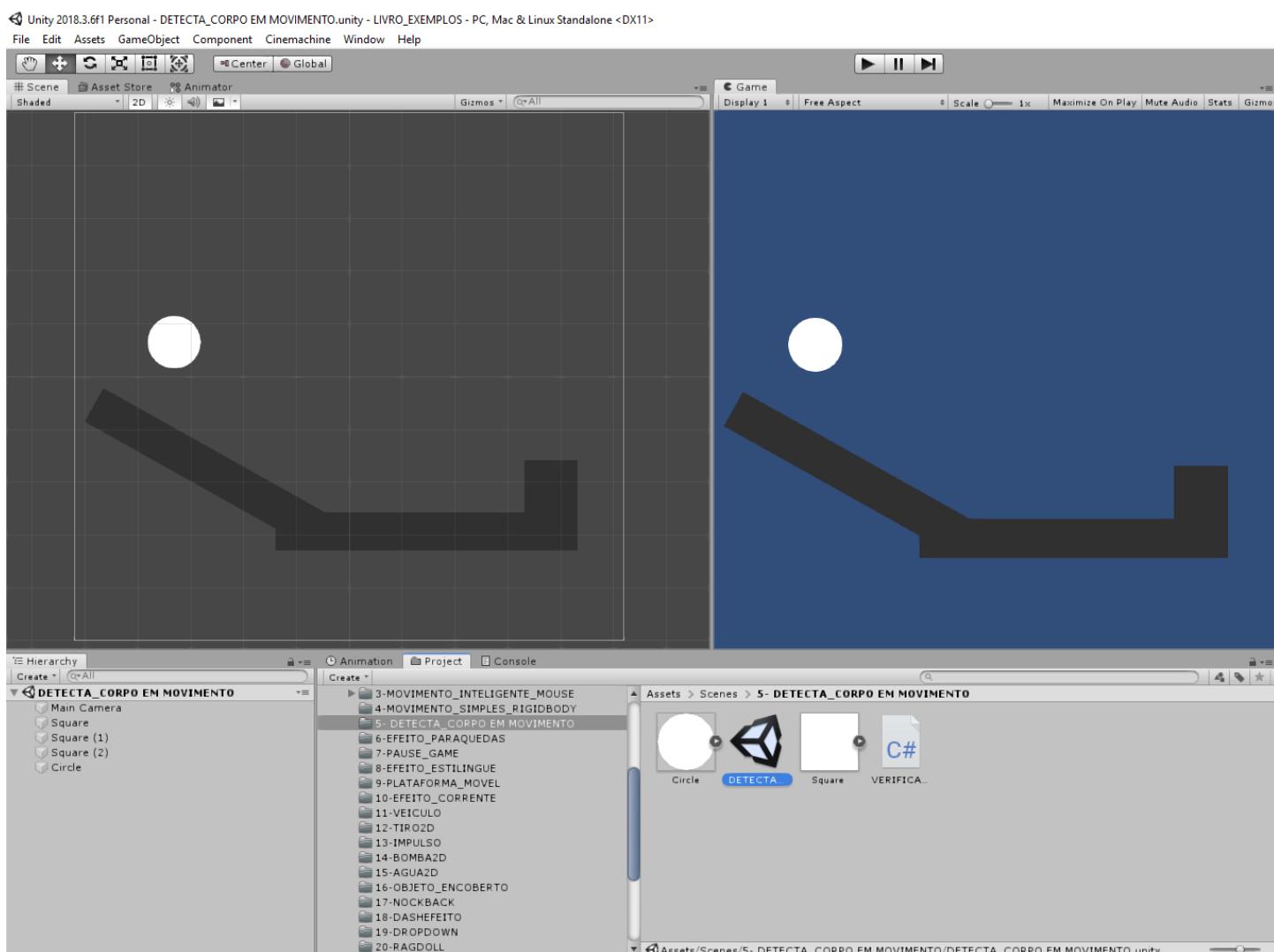
DETECTANDO CORPO EM MOVIMENTO

Outro assunto muito importante quando se trata de desenvolvimento de games é a verificação de movimento de corpos físicos.

Afinal é comum lançar um objeto em cena e ter a necessidade de saber se aquele objeto parou seja para criar outro objeto para ser lançado ou qualquer outra ação.

Então vamos ver esse exemplo.

Para isso é necessário criar uma cena onde um objeto tenha a possibilidade de se movimentar de forma natural usando apenas a gravidade veja:



Para criar essa cena foi necessário criar duas sprites uma quadrada e outra redonda.

Feito isso usando as sprites quadradas deixe sua cena parecida com a cena apresentada acima.

Não se esqueça de adicionar corpos colisores nesses objetos repetindo o processo que já mostrei anteriormente.

Não esqueça também de adicionar um Rigidbody2D na sprite esférica.

E feito isso crie um arquivo de código com o nome de **DETECTA_CORPO_EM_MOVIMENTO** e nesse arquivo escreva o seguinte:

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

```

public class VERIFICA_OBJ_PARADO : MonoBehaviour
{
    public Rigidbody2D rb;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if(rb.velocity.magnitude == 0)
        {
            print("Objeto parado");
        }
    }
}

```

Esse é o código e veja que para uma função tão importante temos algo muito simples escrito nessas linhas.

Veja que para a verificação de movimento funcionar precisamos antes de qualquer coisa criar uma variável do tipo Rigidbody2D

```
public Rigidbody2D rb;
```

Depois disso é necessário dentro do método Update verificar se o objeto está em movimento e para isso é muito simples.

Basta verificar a `velocity.magnitude` do nosso `rigidbody2D` se o valor obtido for igual a zero podemos dizer que o objeto está parado.

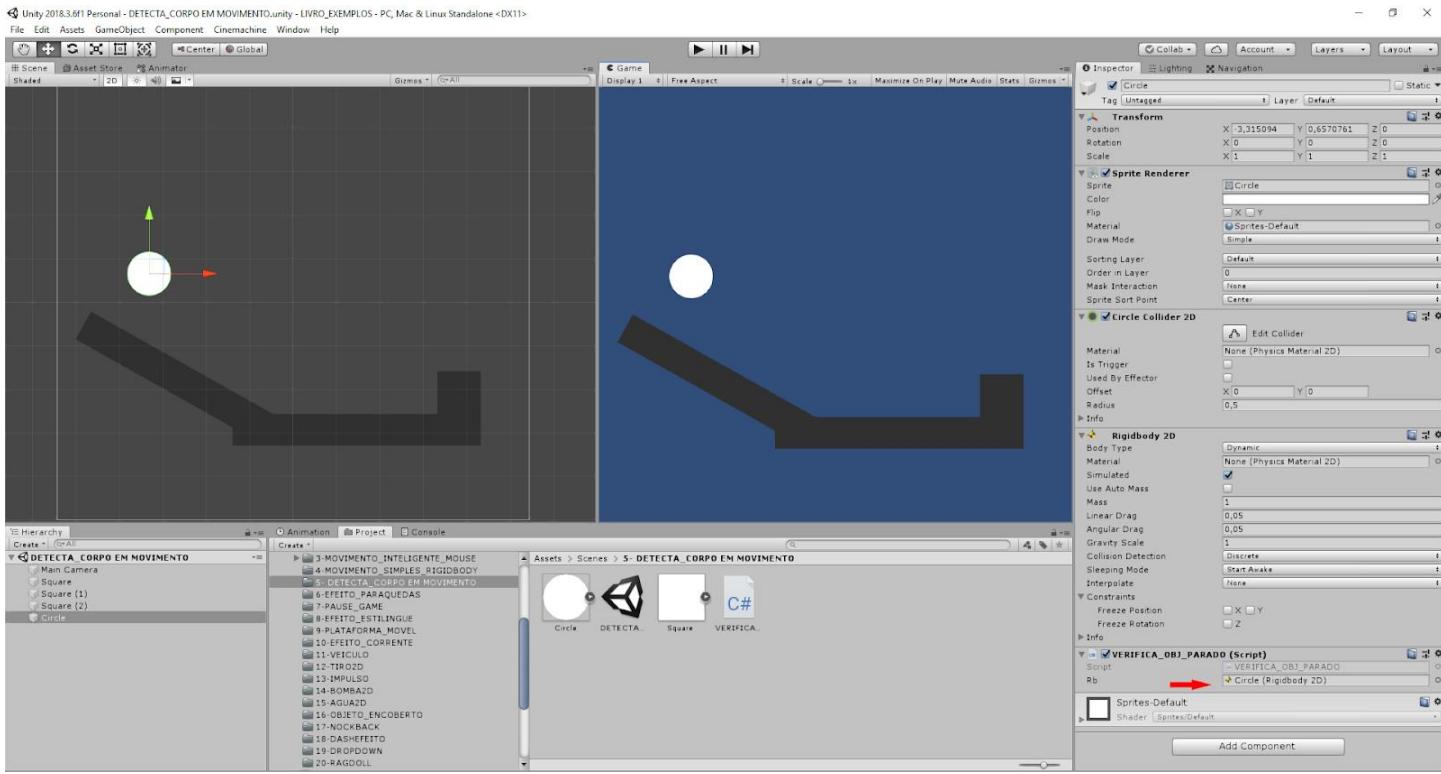
E com a certeza de que o objeto está parado podemos desencadear eventos dentro do nosso game como por exemplo Game Over.

```

void Update()
{
    if(rb.velocity.magnitude == 0)
    {
        print("Objeto parado");
    }
}

```

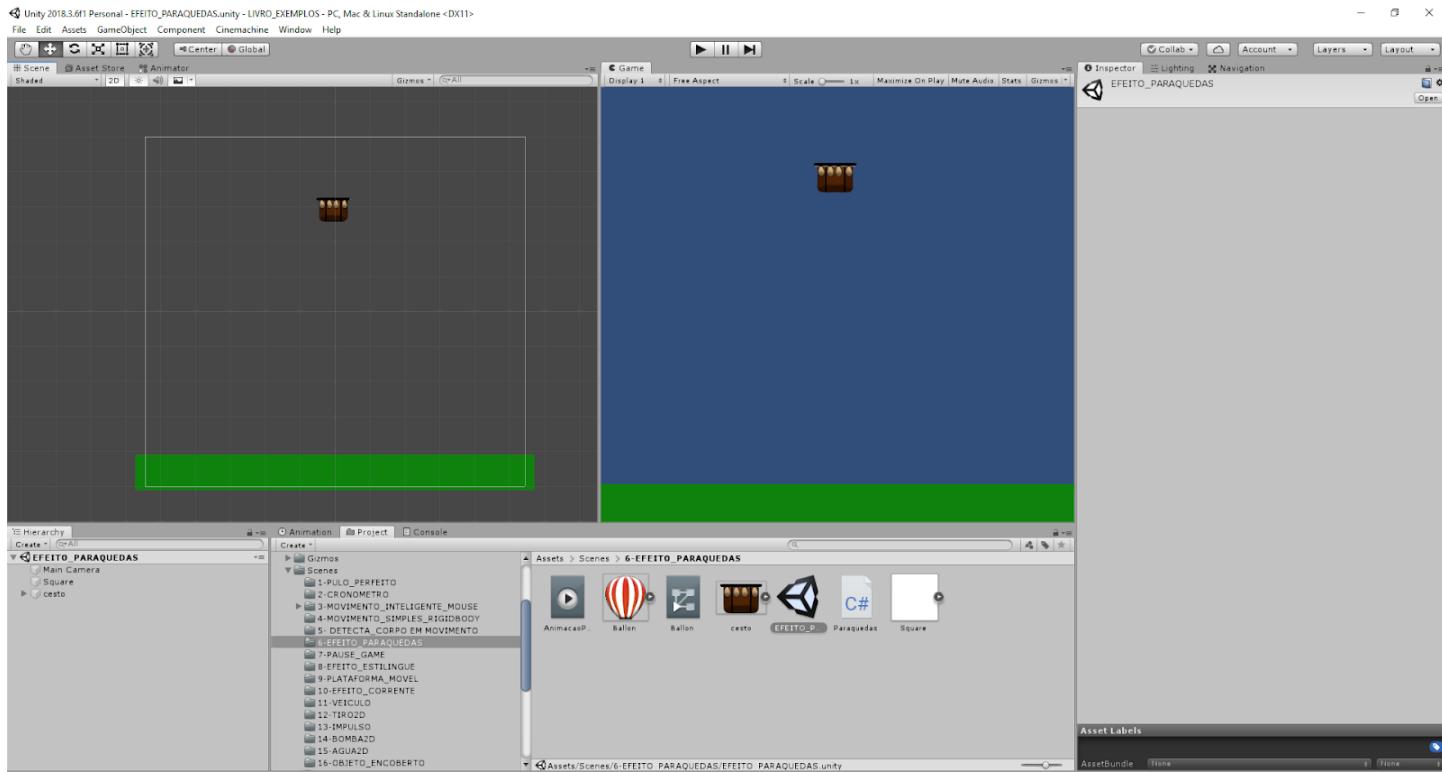
Agora para que isso realmente funcione basta dentro do Unity selecionar a esfera adicionar esse código a ela e passar para o código o `Rigidbody2D` que vamos usar que é o da própria esfera.



Prontinho agora é só executar e ver a magia funcionando.

EFEITO PARAQUEDAS

Um efeito muito legal que podemos usar nos nossos games é o efeito para quedas, esse efeito pode parecer simples mas envolve vários assuntos que juntos nos proporcionam um belo efeito. Para criar esse exemplo vamos criar uma cena parecida com a da imagem abaixo:



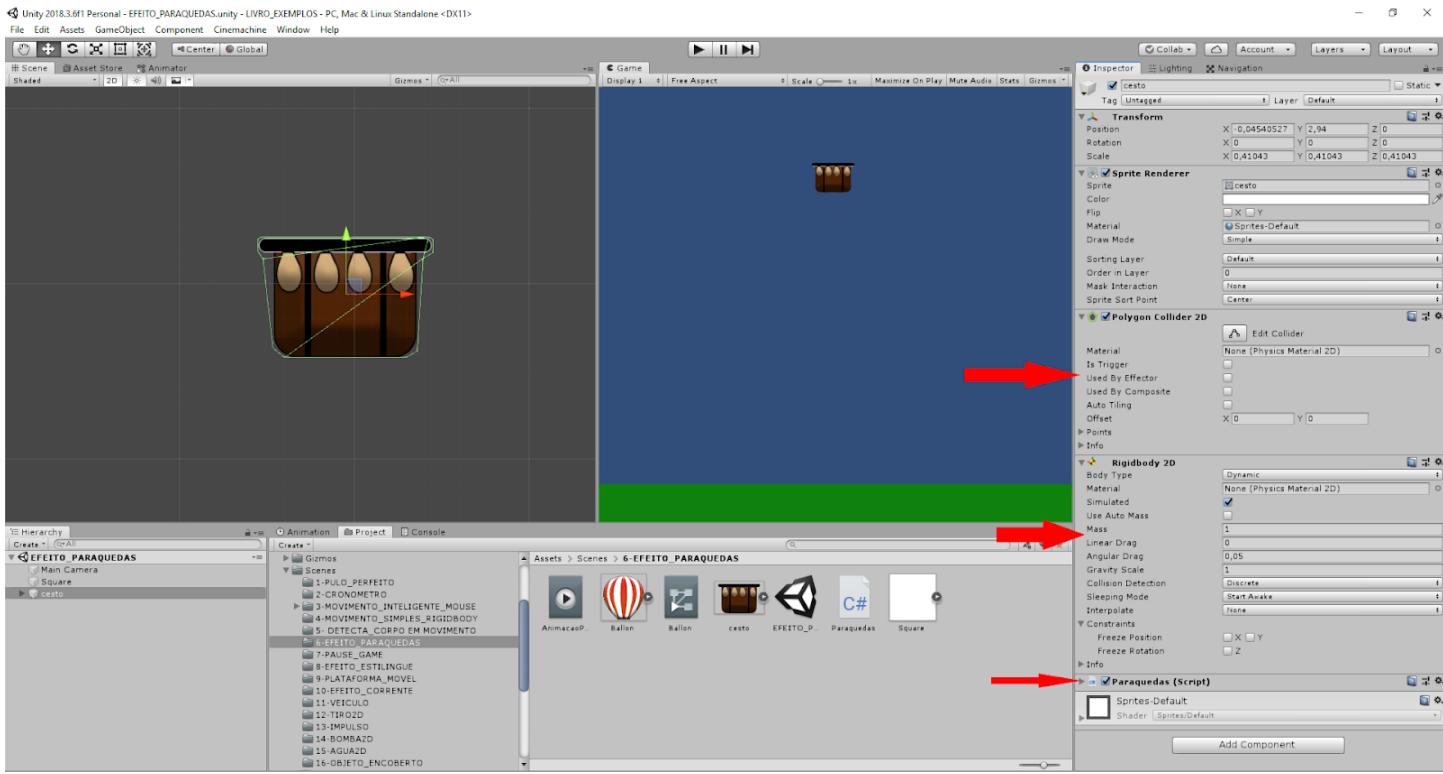
Veja que essa cena é simples temos visivelmente duas sprites em cena uma retangular que serve de chão essa sprite tem um corpo colisor para que nada passe através dela.

E temos o cesto na parte superior, é nesse cesto que vamos focar nesse momento.

Nesse caso eu importei essa imagem para dentro do meu projeto no Unity, esse processo é simples basta arrastar a imagem do local onde ela estiver na sua maquina para dentro da pasta do seu projeto dentro do Unity.

Isso já vai fazer com que a imagem fique disponível para uso no seu jogo.

Então depois disso basta arrastar a imagem para a nossa cena de game e adicionar a ela um corpo colisor do tipo **Polygon Collider**, depois um **Rigidbody2D** e por último um arquivo de código com o nome de **EFEITO_PARAQUEDAS**.

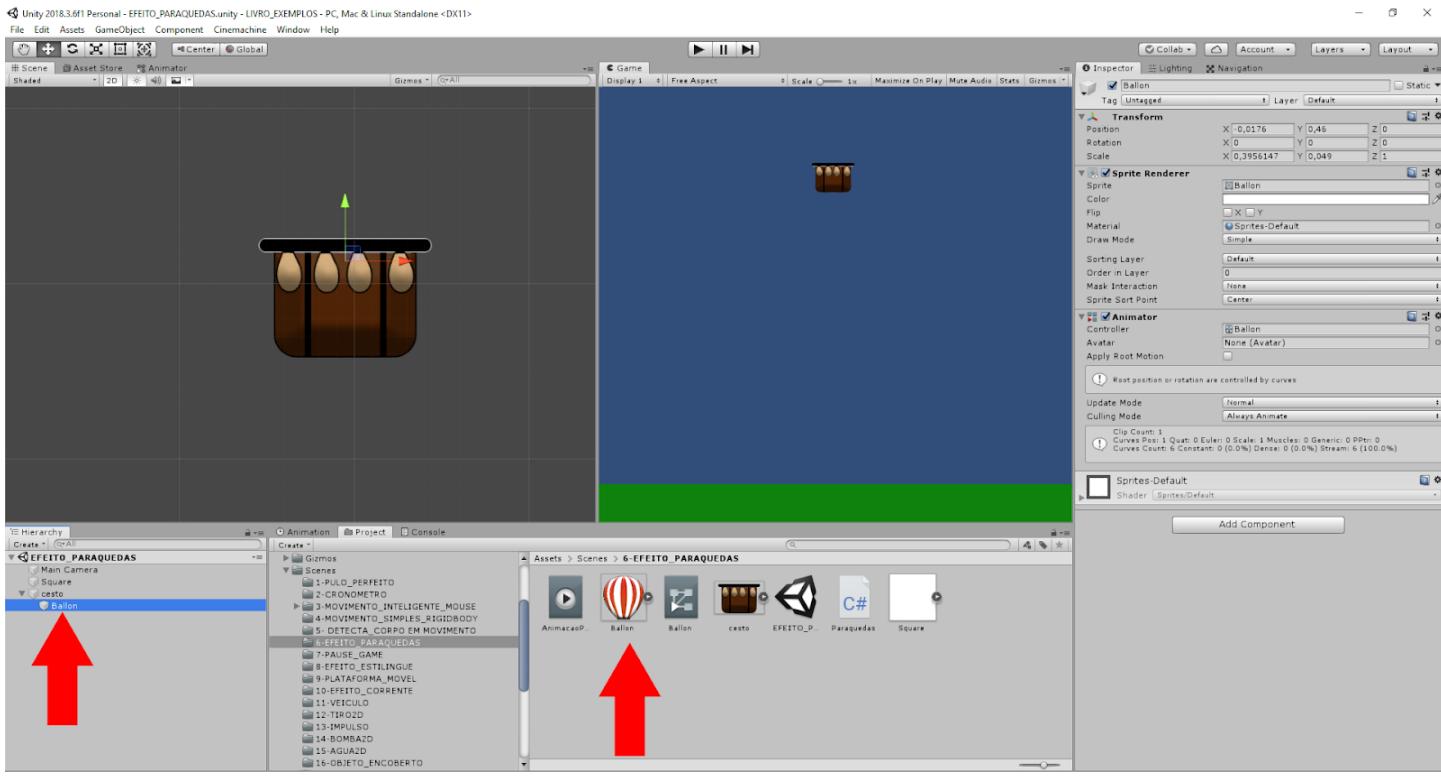


Esses ajustes já preparam essa sprite para se comportar como um objeto físico que vai ser afetado pela gravidade e vai colidir com outros objetos.

O código não vai fazer nada por enquanto pois vamos fazer mais um pequeno ajuste na sprite do cesto antes de escrever esse código.

Veja que não foi apenas a imagem do cesto que foi adicionada ao projeto, temos também a imagem de um balão que vai ser definido como sendo filho do objeto cesto.

Isso acontece para que uma animação do balão possa ser desencadeada quando o objeto se aproximar demais do chão.

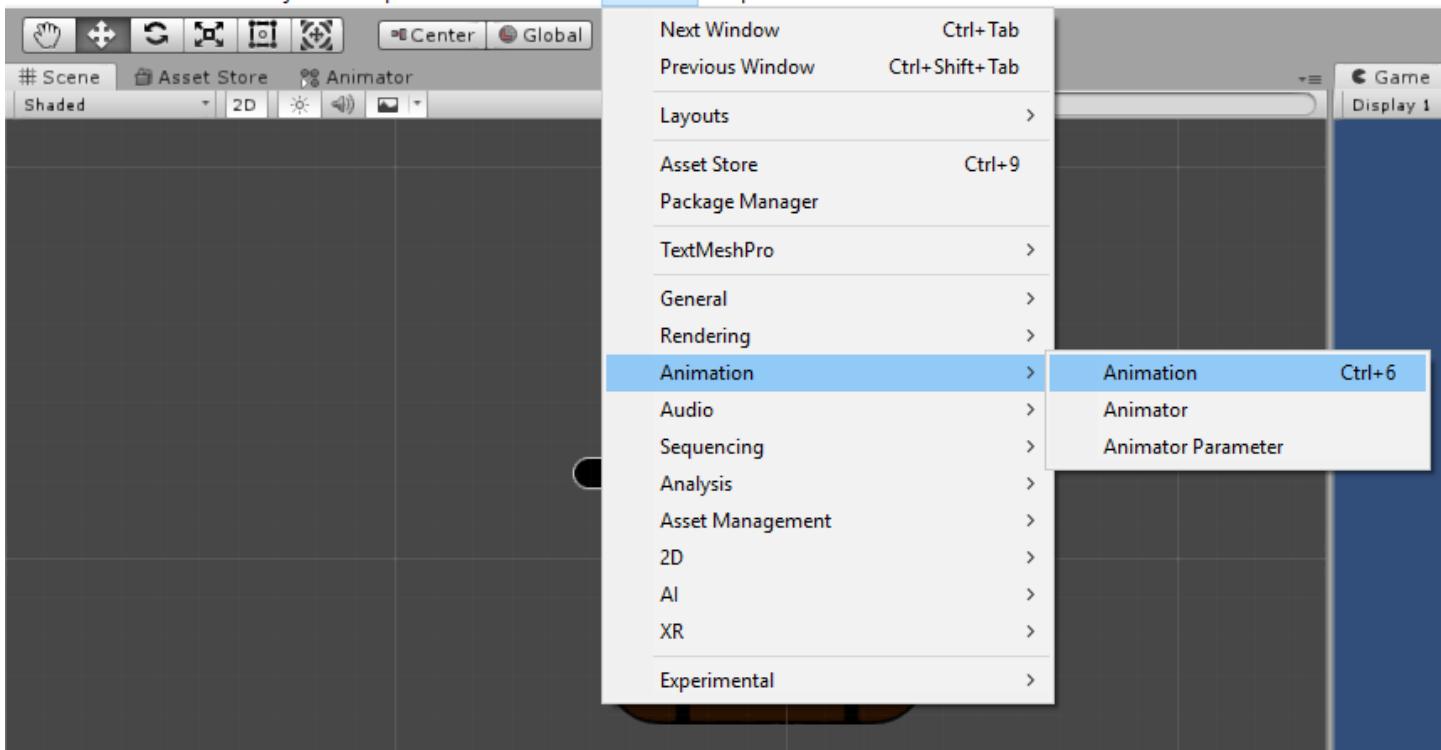


Então já que adicionamos a sprite do balão em nossa cena como filho do objeto cesto vamos selecionar o objeto balão e trabalhar na sua animação.

Para isso faça com que a janela de animação fique visível caso a mesma não esteja.

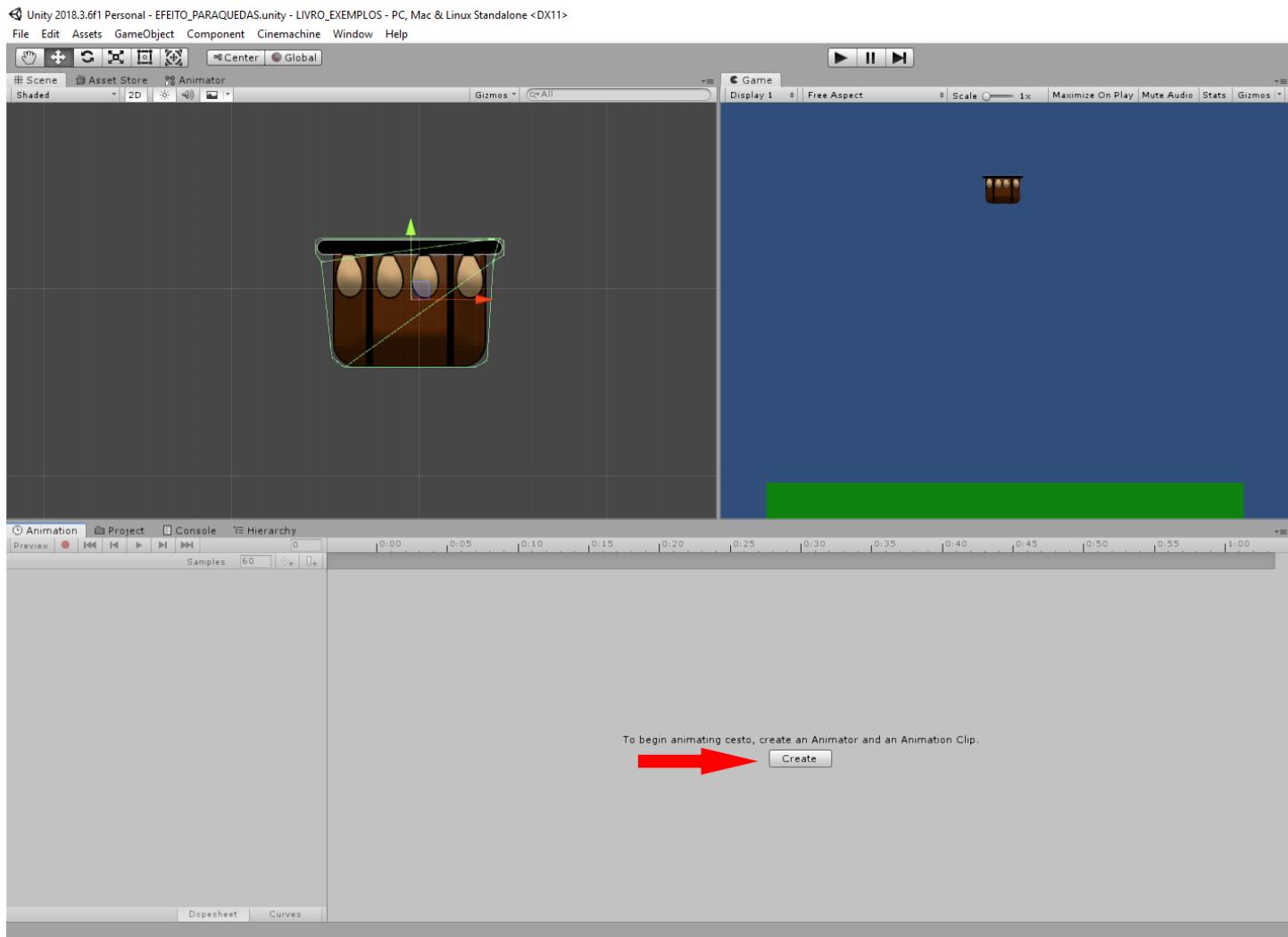
Basta ir em Window depois Animation e escolher a opção Animation.

Aproveite que você já está ai e chame a janela Animator também precisaremos das duas janelas.

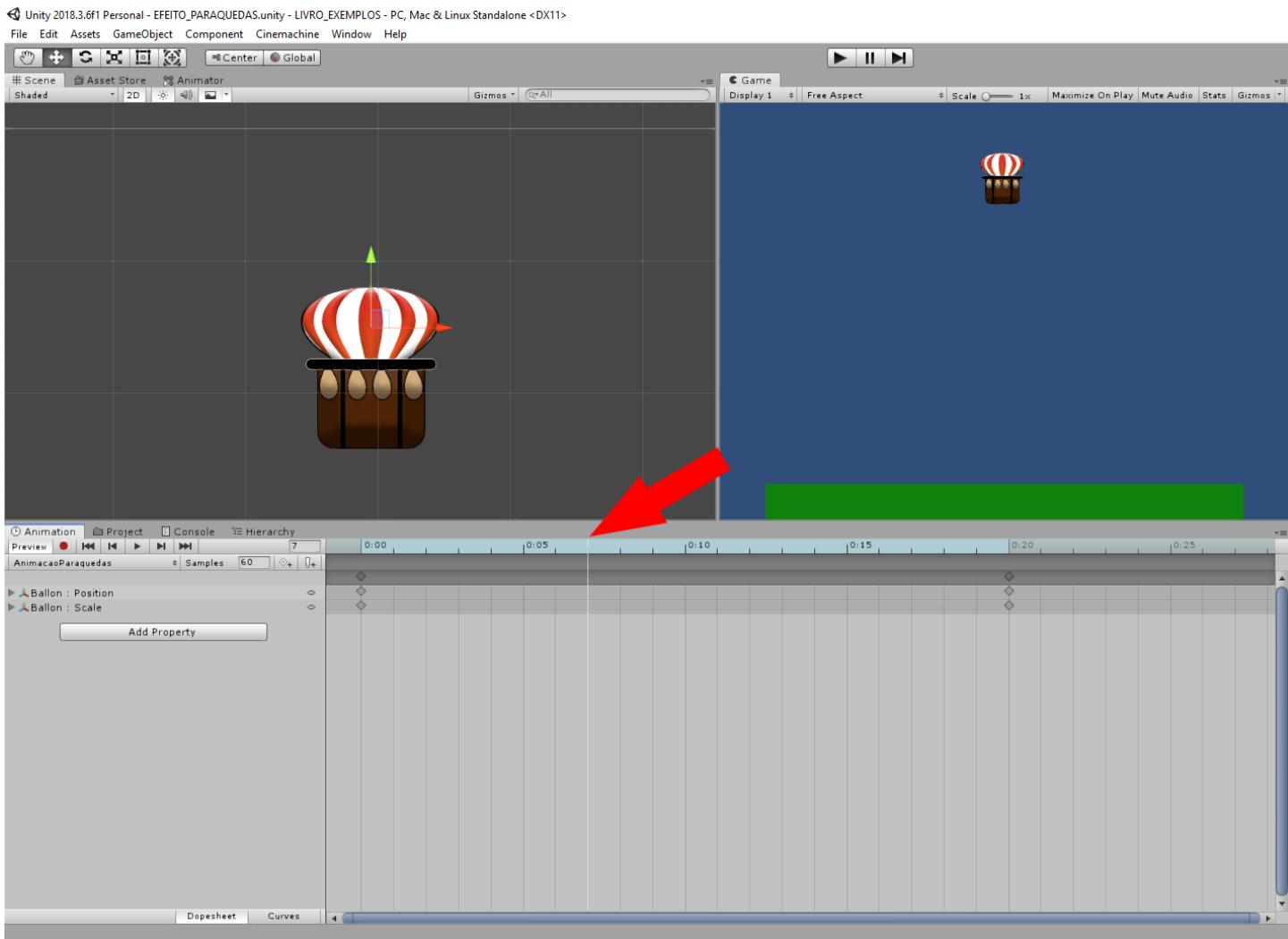


Agora que temos a janela Animation acessível clique sobre o objeto filho do cesto “o balão” e olhe a janela Animation:

Repare que existe um botão com o rotulo dizendo **Create**, esse botão é responsável por criar animações para o objeto selecionado.

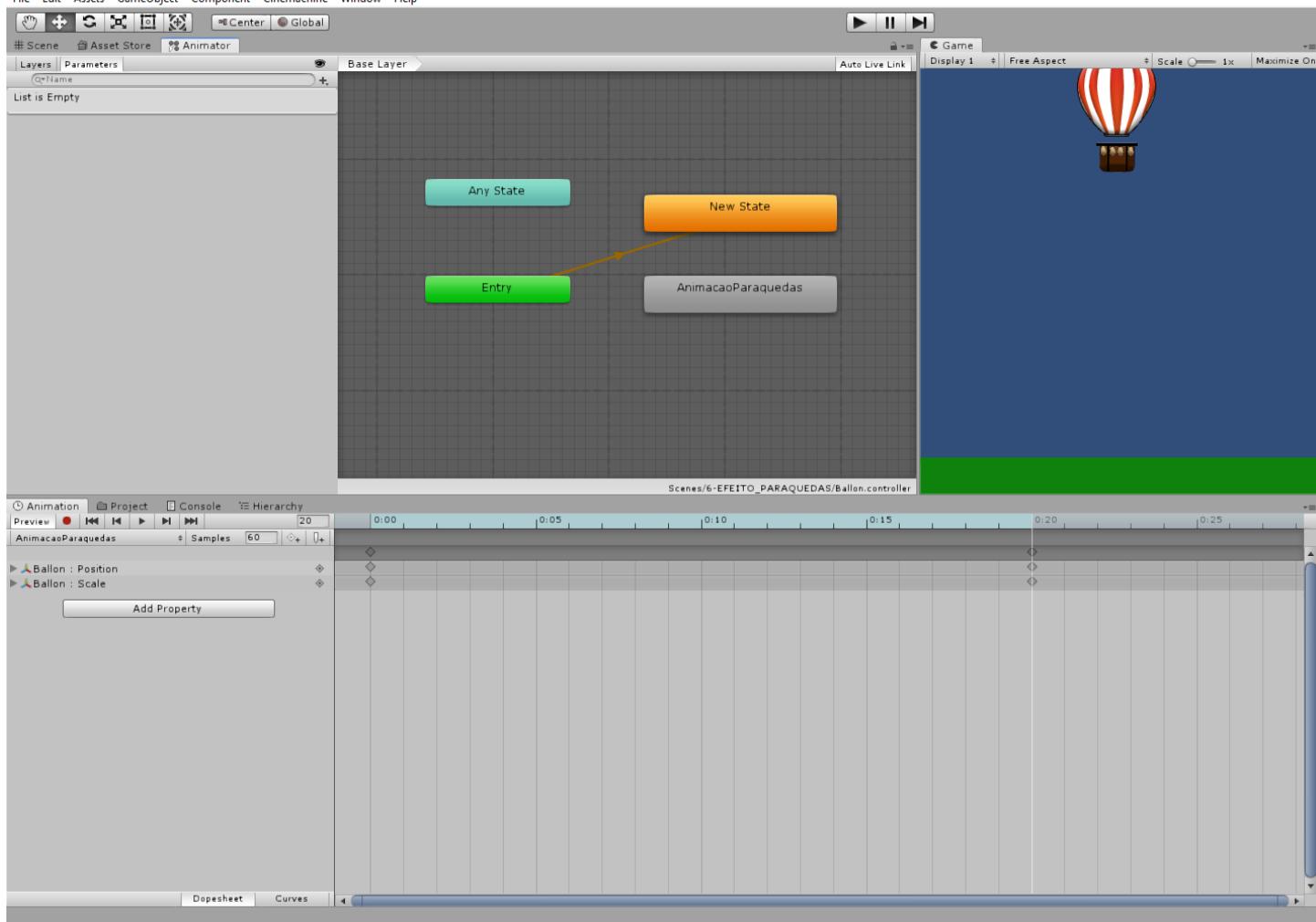


Vamos clicar nesse botão e criar uma animação que manipule a posição e escala do balão veja:



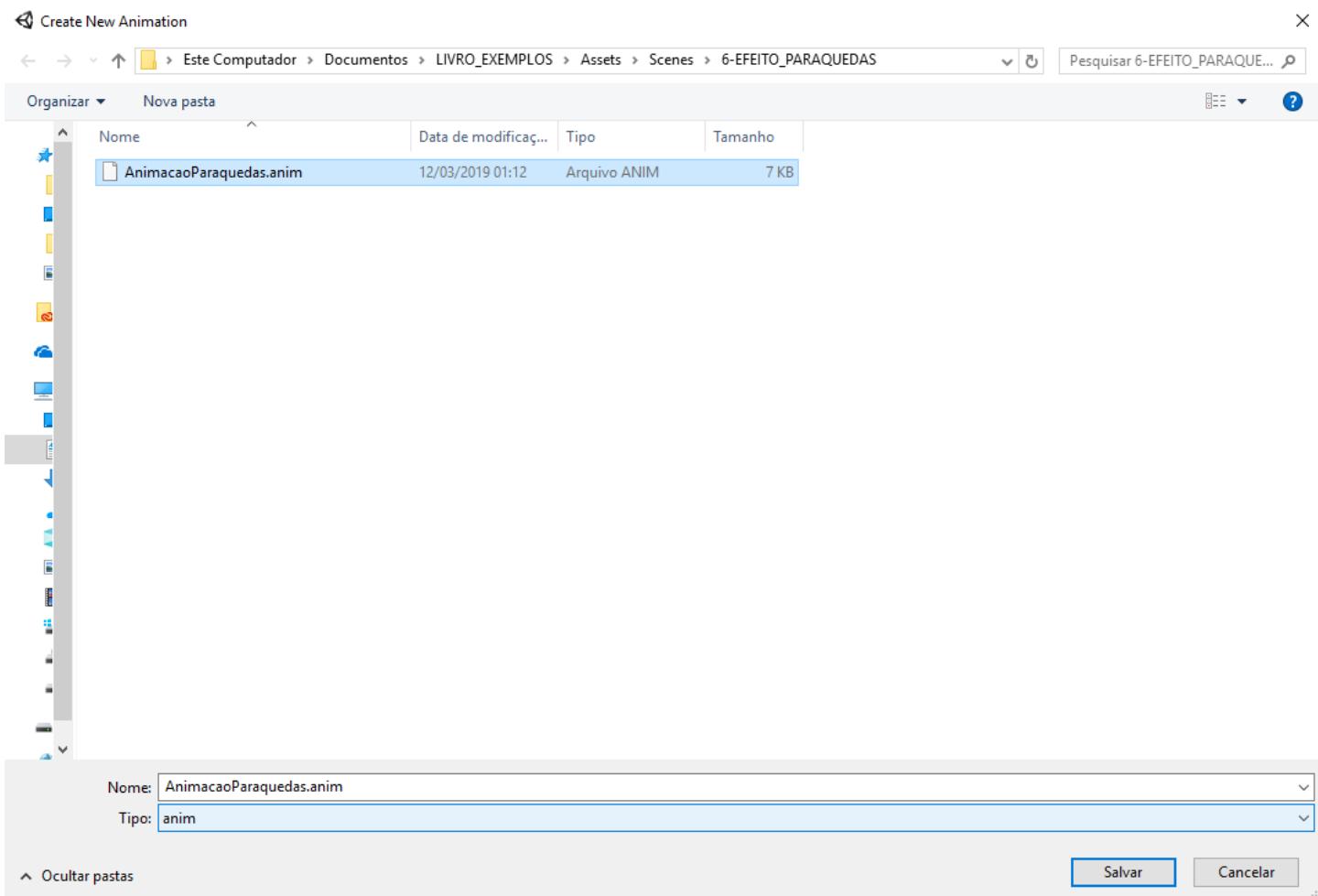
Repare que quando arrasto a barra vertical da Timeline tenho a animação sendo executada, essa animação tem dois pontos extremos o primeiro deixa o balão com sua escala mínima e posição levemente decrementada e no eixo Y, já o ponto máximo tem a escala definida como normal ou seja valor 1 para os eixos X, Y e Z e sua posição no eixo Y levemente incrementada.
 Com isso temos nossa animação criada e pronta para trabalhar, mas precisamos fazer mais um ajuste lembra que falei da janela Animator?
 Pois é agora é a hora de usa-la, abra a sua janela animator do jeito que foi explicado anteriormente e deixe as configurações internas da mesma forma que a imagem abaixo:

Unity 2018.3.6f1 Personal - EFEITO_PARAQUEDAS.unity - LIVRO_EXEMPLOS - PC, Mac & Linux Standalone <DX11>
File Edit Assets GameObject Component Cinemachine Window Help



Veja que dentro do Animator temos um arquivo que representa a animação que acabamos de criar o arquivo é o **AnimacaoParaquedas** que tem esse nome pois na janela Animation quando criamos uma animação damos um nome para ela.

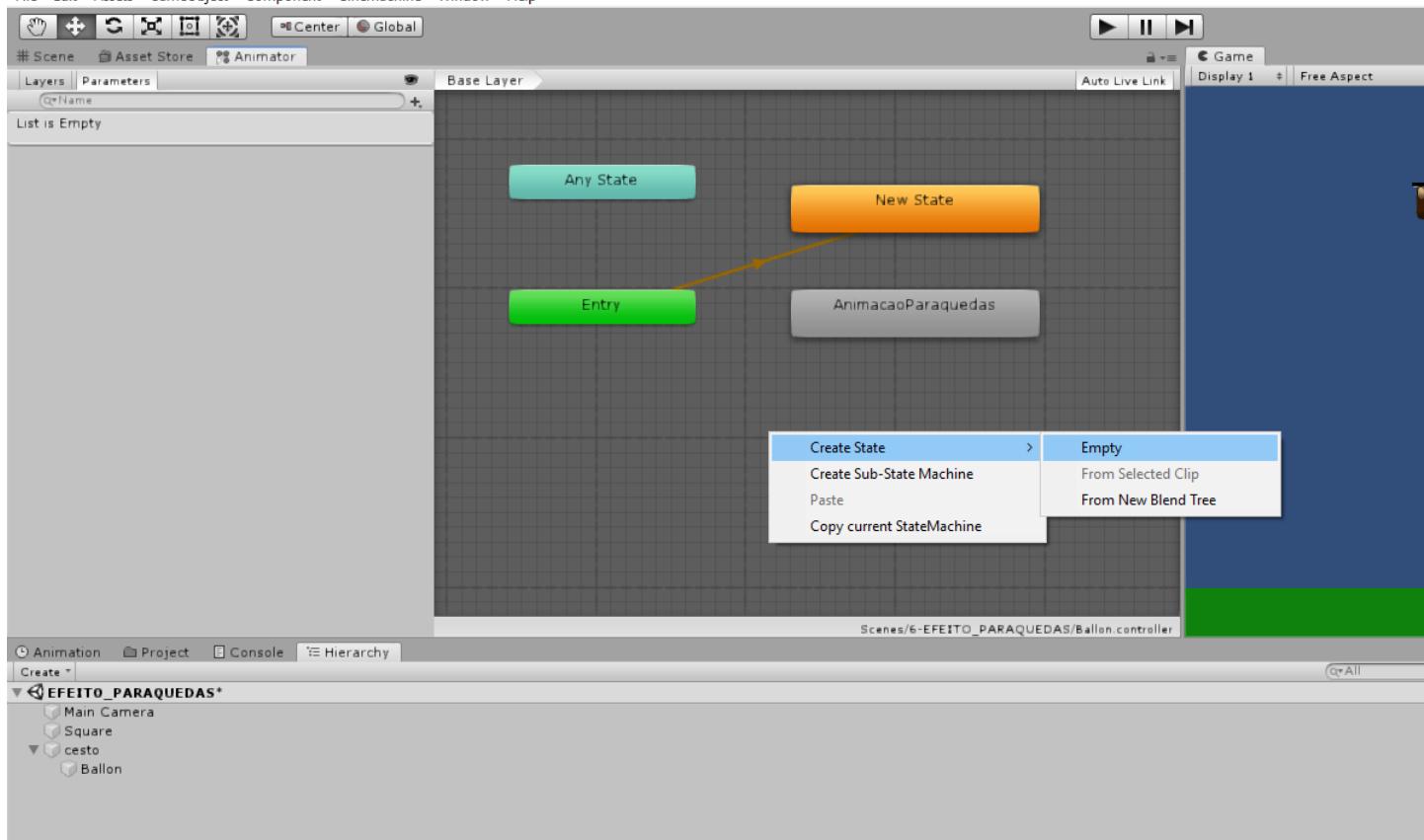
Veja:



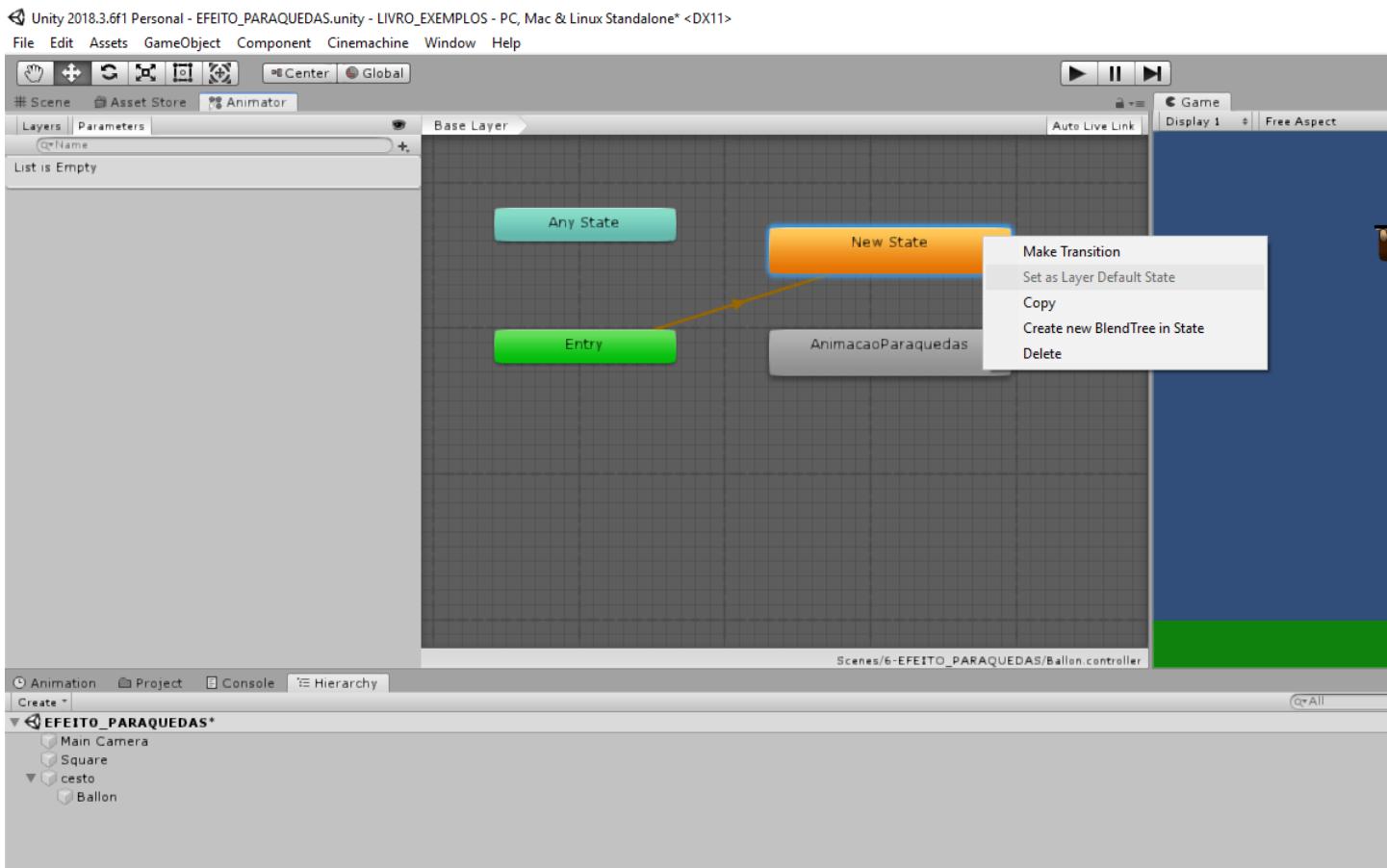
Então dessa forma a animação aparece com o seu nome certinho dentro do Animator, mas e agora como vamos garantir que essa animação vai ser reproduzida só no momento que for necessário? É muito simples para fazer isso vamos criar um estado de animação vazio. Para fazer isso basta clicar com o botão direito do mouse sobre a área de trabalho do Animator e escolher a opção **Create State** depois **Empty**.

Unity 2018.3.6f1 Personal - EFEITO_PARAQUEDAS.unity - LIVRO_EXEMPLOS - PC, Mac & Linux Standalone* <DX11>

File Edit Assets GameObject Component Cinemachine Window Help



Com esse estado de animação criado agora defina o como sendo o estado de animação padrão. Para fazer isso é muito fácil basta selecionar o estado vazio e depois clicar sobre ele com o botão direito do mouse nas opções que vão aparecer escolha a opção **Set as Layer Default State**.



Certo agora que já ajustamos a nossa cena e os elementos dentro dela vamos trabalhar no código que vai fazer toda a magica acontecer.

Então crie um arquivo de código com o nome de Paraquedas e adicione esse código no objeto cesto. Feito isso escreva o seguinte código no arquivo criado:

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

public class Paraquedas : MonoBehaviour
{
    public float distancia;
    public LayerMask mask;
    public Rigidbody2D rb;
    public Animator animator;
    public bool gatilho;

    // Start is called before the first frame update
    void Start()
    {

    }
}
```

```

// Update is called once per frame
void Update()
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, -Vector2.up, distancia, mask);

    if(hit)
    {
        rb.drag = 20f;
        animator.Play("AnimacaoParaquedas");
        gatilho = true;
    }
}

}

```

Veja que nesse código a primeira coisa que foi feita é escrever as variáveis que serão usadas.

```

public float distancia;
public LayerMask mask;
public Rigidbody2D rb;
public Animator animator;
public bool gatilho;

```

Veja que temos uma variável float que chamamos de distancia que é basicamente a distância entre a sprite do cesto e o chão.

Depois temos uma variável do tipo LayerMask que nos auxilia na identificação do chão.

Temos a variável do tipo Rigidbody2D que é o rigidbody do nosso objeto cesto.

Temos uma variável do tipo Animator que cuida da animação do objeto filho que é o balão.

E por último temos uma variável do tipo booleana que é apenas didática a única serventia dessa variável é mostrar quando o paraquedas deve abrir.

Continuando temos o método Update onde usamos um Raycast na posição do cesto apontado para baixo que verifica quando o raio colide com o chão.

Quando essa colisão acontece ajustamos o valor de drag do nosso Rigidbody2D para 20 deixando a queda mais suave e já na sequência disparamos a animação do balão e mudamos o valor do gatilho para verdadeiro.

```

void Update()
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, -Vector2.up, distancia, mask);

    if(hit)
    {
        rb.drag = 20f;
        animator.Play("AnimacaoParaquedas");
    }
}

```

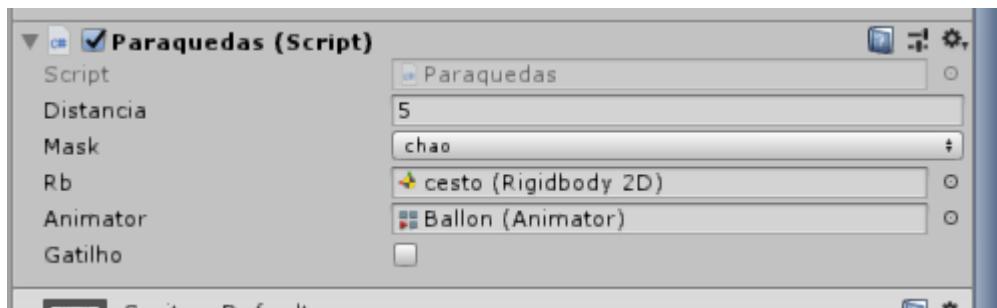
```

        gatilho = true;
    }

}

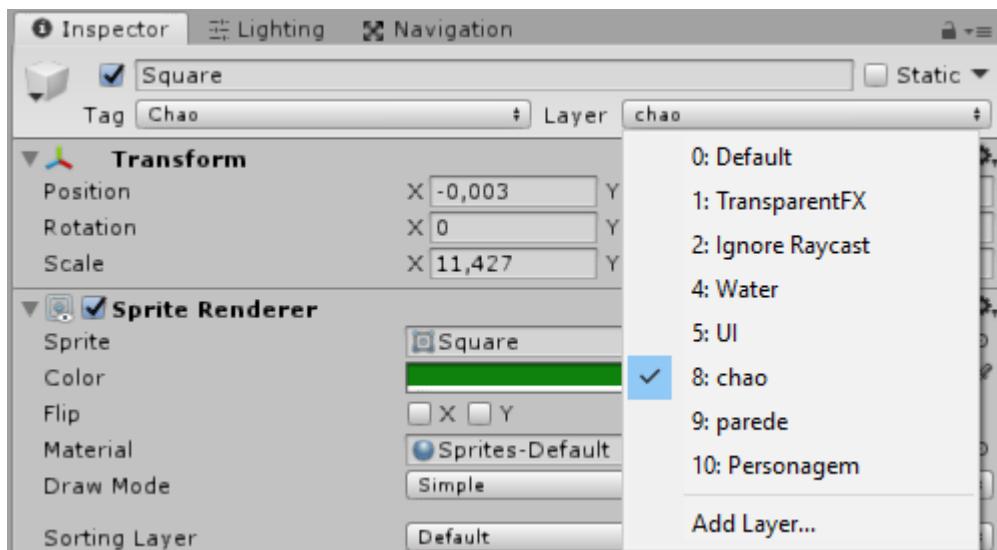
```

Com isso o exemplo vai funcionar perfeitamente porém, precisamos fazer mais alguns ajustes no código lá dentro do Inspector.



Veja que passamos os valores que vamos usar tanto para distância do raio, mascara que esta definida como layer chão o Rigidbody2D e o Animator usado nesse exemplo.

Agora mais um detalhe importante é para a criação do layer usado aqui veja:



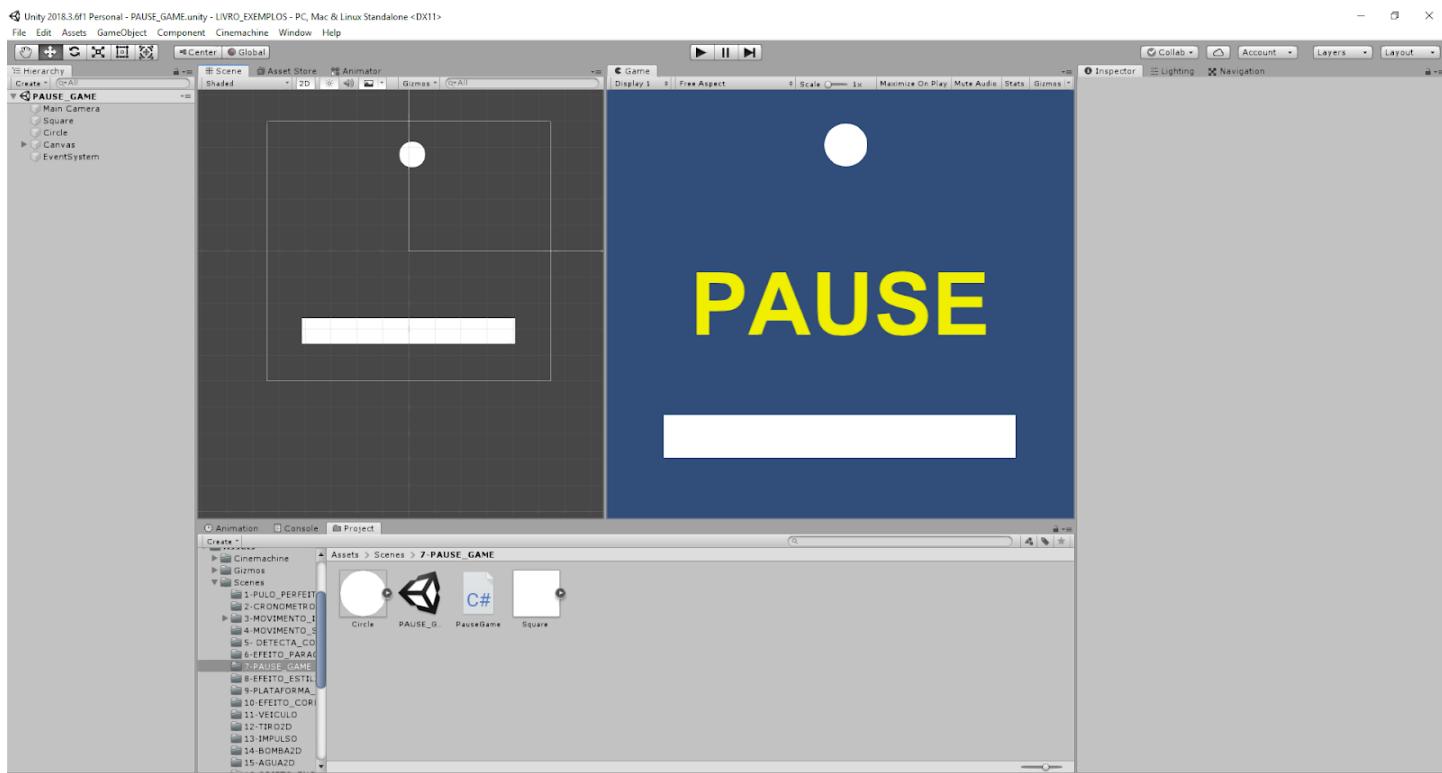
Repare que foi criado um layer com o nome de chão e esse layer foi aplicado ao objeto chão da cena. É isso agora é só executar o exemplo e ver tudo funcionando.

PAUSE GAME

Agora vamos ver mais um exemplo muito legal e que todo mundo quer adicionar ao seu game que é o pause.

Quem nunca falou “Dá um pause ai”, pois é o pause é o pause e nesse exemplo vamos ver como é simples criar um pause game.

Para isso antes de qualquer coisa precisamos de uma nova cena e nela vamos adicionar elementos dessa forma:



Veja que nessa cena temos duas sprites uma esfera e um retângulo e temos um Canvas e dentro desse Canvas temos um Text.

O retângulo que representa o chão tem apenas um corpo colisor, já a esfera tem um corpo colisor e um RigidBody2D.

Com isso vamos ter uma visão maior do efeito de pause sendo executado.

Então vamos criar um código com o nome de PauseGame e vamos adicionar esse código dentro da câmera da nossa cena.

Feito isso vamos escrever o seguinte código:

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

public class PauseGame : MonoBehaviour
{
    private bool paused;
    public Text pauseTxt;
```

```

// Start is called before the first frame update
void Start()
{
    pauseTxt.enabled = false;
}

// Update is called once per frame
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        if (Time.timeScale == 1)
        {
            Time.timeScale = 0;
            pauseTxt.enabled = true;
        }

        else if (Time.timeScale == 0)
        {
            Time.timeScale = 1;
            pauseTxt.enabled = false;
        }
    }
}
}

```

Veja que nesse código temos a seguinte linha que nos permite trabalhar com UI.

```
using UnityEngine.UI;
```

Depois disso no método Start apenas desabilitamos o nosso text para que o mesmo não fique visível o tempo, mas que aparece somente quando o pause for ativado.

```

void Start()
{
    pauseTxt.enabled = false;
}

```

Já no método Update temos uma estrutura condicional que verifica se apertamos a tecla espaço se apertamos verificamos na sequencia se o Time.timeScale é igual a 1 se for mudamos esse valor para zero e habilitamos novamente a visualização do nosso text pois nesse momento nosso pause já esta funcionando.

Agora caso contrario se o timeScale for igual a 0 quer dizer o jogo esta pausado então vamos fazer com que ele rode novamente mudando o valor de timescale para 1 e desabilitando a visualização do text.

```
void Update()
```

```

    }

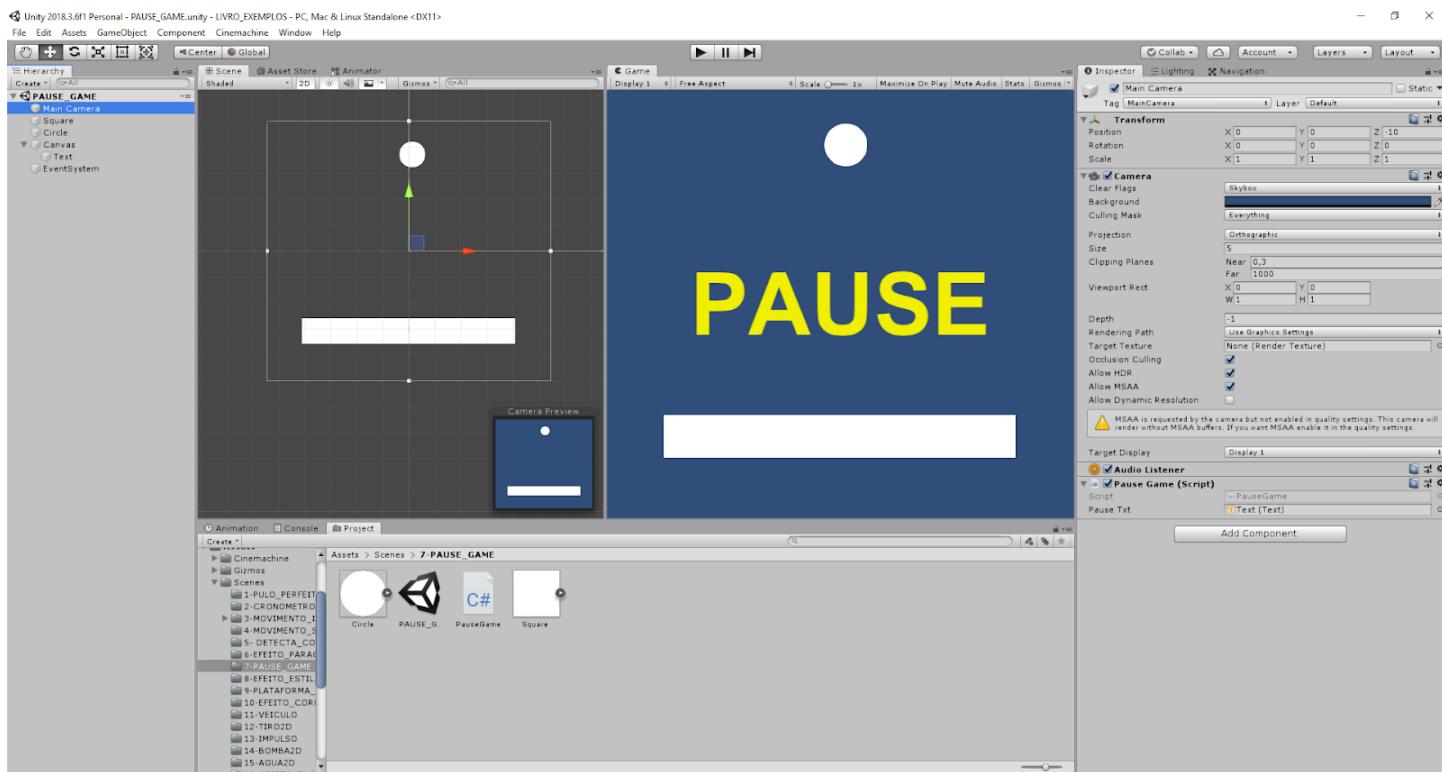
    if (Input.GetKeyDown(KeyCode.Space))
    {
        if (Time.timeScale == 1)
        {
            Time.timeScale = 0;
            pauseTxt.enabled = true;
        }

        else if (Time.timeScale == 0)
        {
            Time.timeScale = 1;
            pauseTxt.enabled = false;
        }
    }
}

```

Com isso o exemplo já vai funcionar perfeitamente.

Só precisa fazer um ajuste no código que está adicionado na câmera, precisamos passar o nosso text para o slot do código que esta no Inspector.



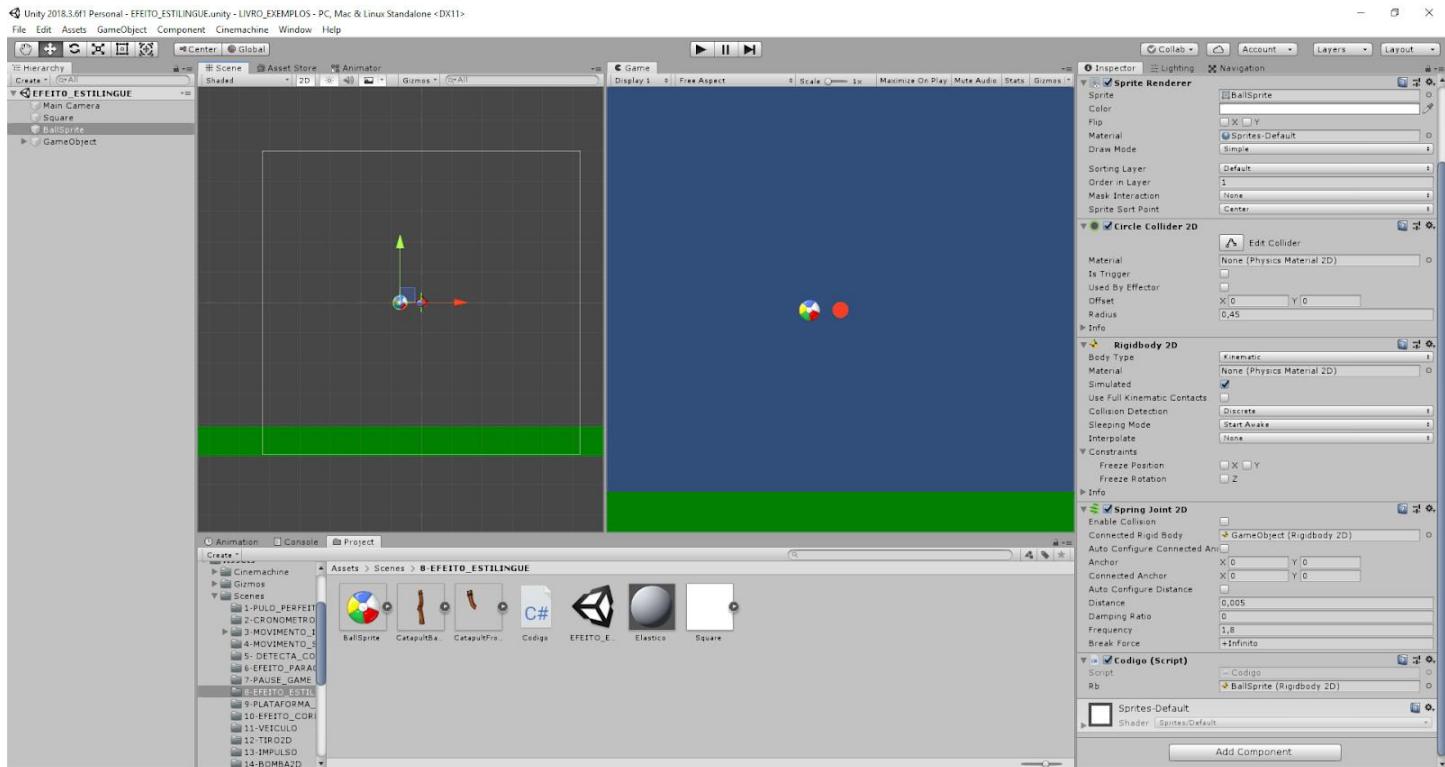
Agora basta executar e ver o exemplo funcionando.

EFEITO ESTILINGUE

Mais um exemplo muito legal é o estilingue, nada melhor que poder disparar objetos como se estivesse manipulando um estilingue.

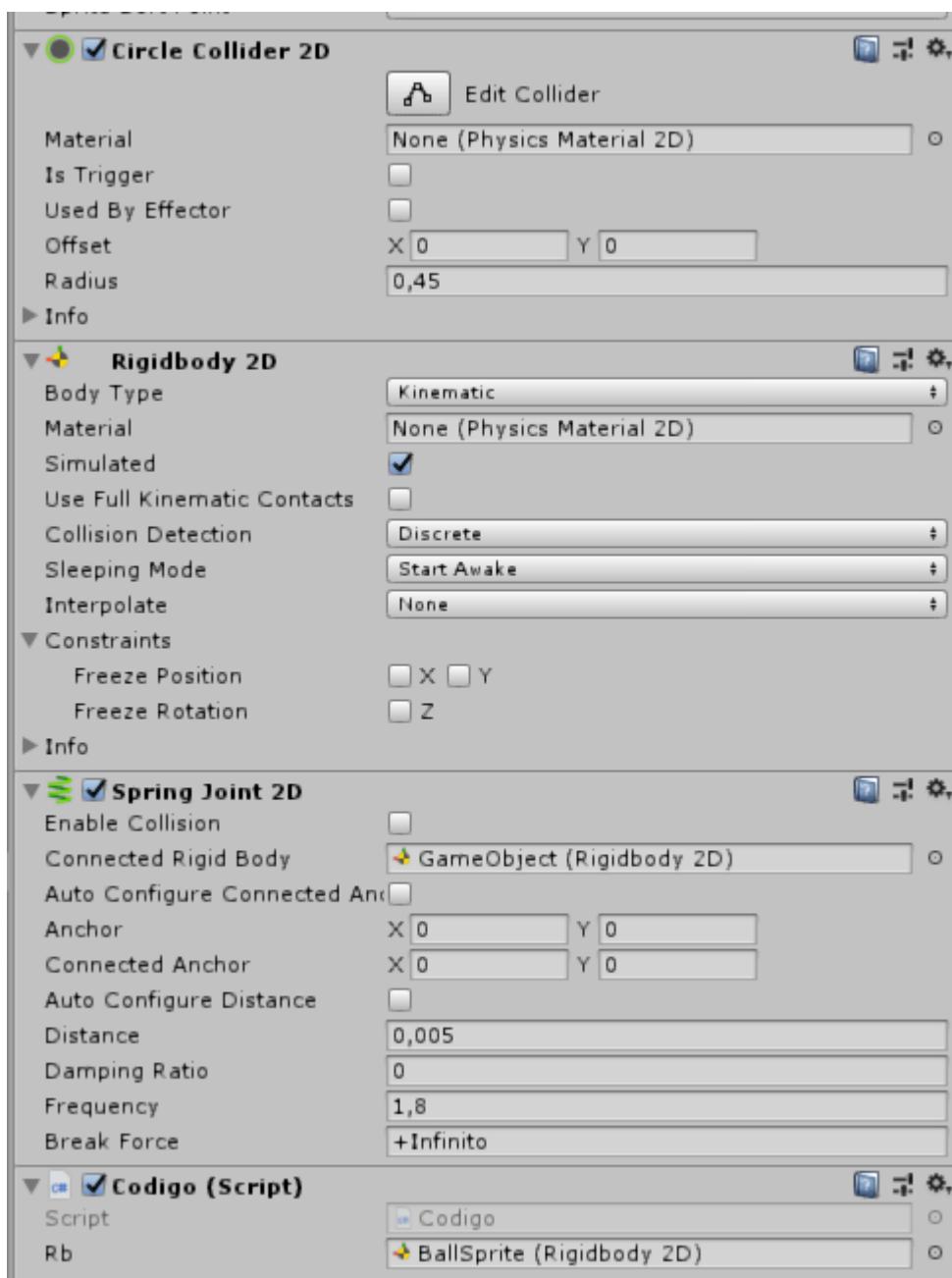
E esse efeito é muito fácil de ser reproduzido, claro que podemos deixar ele cada vez mais complexo mas o principal dessa mecânica é muito fácil de fazer.

Para isso precisamos criar uma cena parecida com a da imagem abaixo:



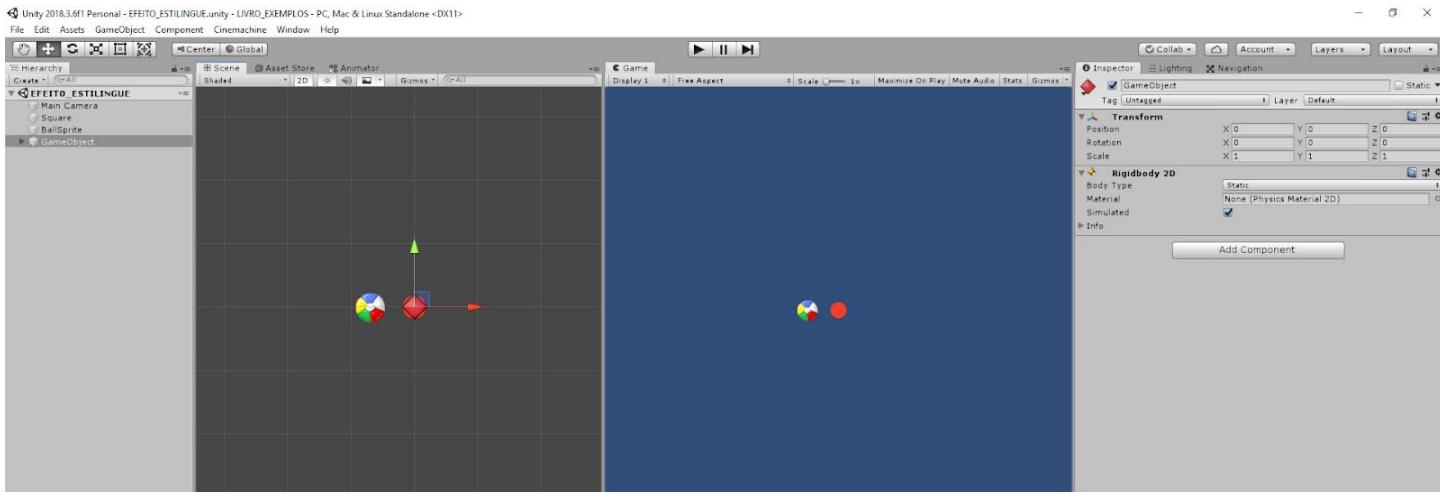
Veja que a cena é realmente muito simples temos apenas uma sprite representando o chão, uma sprite da bola colorida que é justamente o objeto que vai ser arremessado e temos um gameobject vazio que serve de suporte para o efeito estilingue.

Agora com a bola selecionada vamos dar uma olhadinha no Inspector.



Veja que esse objeto tem um colisor um Rigidbody2D e um código que se chama código mesmo. Até aqui nada demais porém, perceba que agora temos um novo componente na jogada que é o Spring Joint 2D.

Esse componente age como uma mola que é justamente o que queremos nesse momento. Repare que nas configurações desse componente temos um campo chamado Connected Rigid Body que nada mais é que o Rigidbody que vamos usar para conectar esse componente. Como queremos que o componente se ligue ao game object vazio na cena então esse game object precisa ter um Rigidbody2D veja:



Perceba que o Game object tem um Rigidbody2D mas o mesmo está definido como static, então ele não vai se mexer.

Legal agora que terminamos esses ajustes podemos trabalhar no código que foi adicionado dentro do objeto bola.

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

public class Codigo : MonoBehaviour
{
    public Rigidbody2D rb;
    private bool selecao = false;
    private WaitForSeconds tempo = new WaitForSeconds(0.15f);

    void Update()
    {
        if(selecao)
        {
            rb.position =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
        }
    }

    void OnMouseDown()
    {
        selecao = true;
        rb.isKinematic = true;
    }

    void OnMouseUp()
    {
        selecao = false;
        rb.isKinematic = false;
        StartCoroutine("QuebraSpring");
    }
}
```

```

IEnumerator QuebraSpring()
{
    yield return tempo;
    GetComponent<SpringJoint2D>().enabled = false;
}
}
}

```

Agora vamos entender o nosso código veja que a primeira coisa que fizemos foi definir as variáveis que serão usadas nesse exemplo.

Temos aqui uma variável do tipo Rigidbody2D que serve para manipular a nossa bola, em seguida temos uma outra variável do tipo booleana que verifica a seleção do objeto bola. E por último temos a definição do tempo que será usado dentro de uma corotina que está mais adiante no código.

```

public Rigidbody2D rb;
private bool selecao = false;
private WaitForSeconds tempo = new WaitForSeconds(0.15f);

```

Agora que definimos as variáveis desse exemplo vamos trabalhar dentro do nosso método Update. Veja que a primeira coisa que fizemos aqui foi criar uma estrutura condicional que verifica se eu selecionei a bola.

Se essa condição for satisfeita, vamos passar para a posição do nosso rigidbody a posição do espaço de tela como espaço de mundo.

```

void Update()
{
    if(selecao)
    {
        rb.position =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
    }
}

```

E continuando temos alguns métodos que vão nos auxiliar para fazer com que o efeito funcione corretamente, o primeiro é o método OnMouseDown que é onde verifico se selecionei a bola se estou com o botão esquerdo do mouse apertado sobre o objeto bola.

Nesse caso deixo seleção como verdadeiro e ativo o isKinematic do meu rigidbody pois nesse momento estou manipulando a bola definindo até onde vou esticar o elástico que vai lança-la.

Depois em OnMouseUp é onde deixo de apertar o botão esquerdo do mouse soltei a bola para que a mesma possa ser arremessada.

Nesse caso ajusto o valor da variável seleção para false, deixo isKinematic como false e chamo uma corotina que vai ser a responsável por quebrar o link entre a bola e o ponto de conexão da mola.

```

void OnMouseDown()
{
    selecao = true;
    rb.isKinematic = true;
}

```

```
void OnMouseUp()
{
    selecao = false;
    rb.isKinematic = false;
    StartCoroutine("QuebraSpring");
}
```

Agora vamos dar uma olhada na nossa corotina, veja que ela é muito simples. Apenas definimos um tempo para desabilitar o SpringJoint fazendo com que a bola seja arremessada. Por essa razão essa corotina é executada quando soltamos o botão do mouse.

```
IEnumerator QuebraSpring()
{
    yield return tempo;
    GetComponent<SpringJoint2D>().enabled = false;
}
```

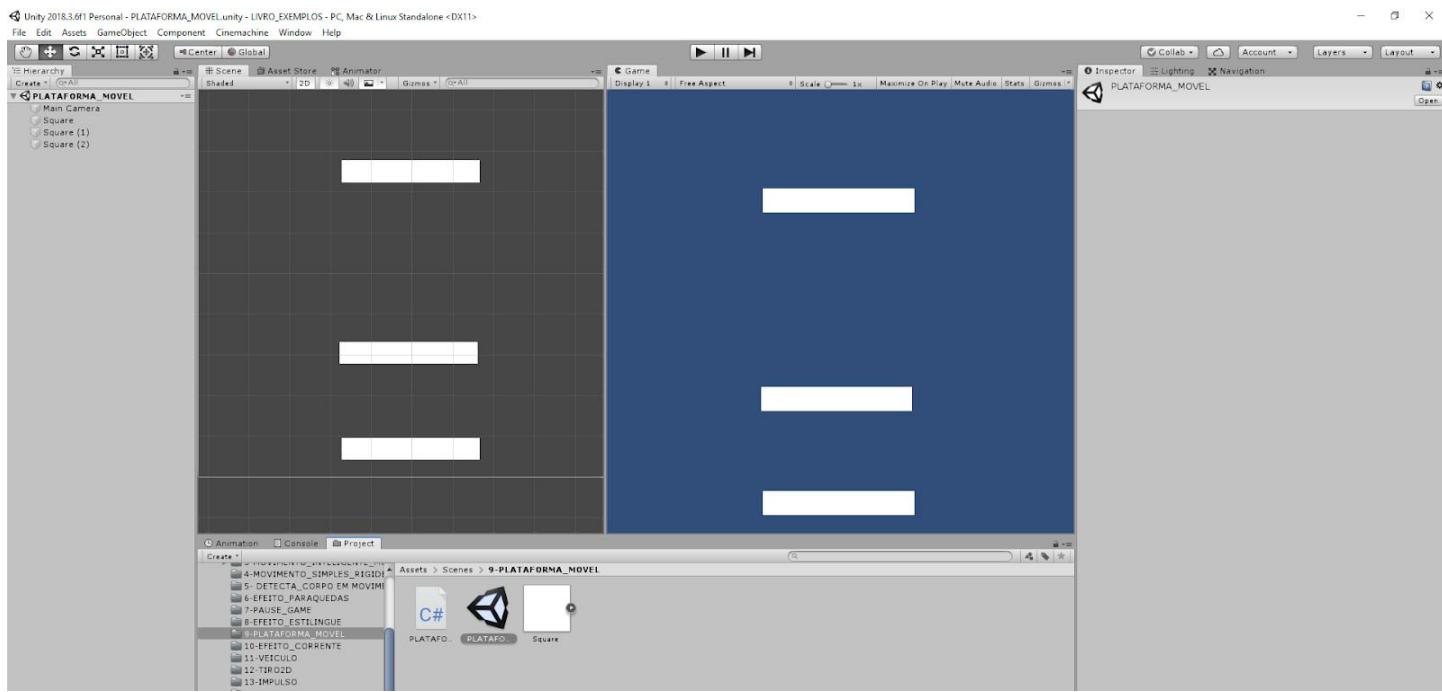
Agora sim terminamos mais esse exemplo.

PLATAFORMA MÓVEL

Outro efeito comum dentro dos jogos é o movimento de plataformas, seja na vertical, na horizontal ou até mesmo em diagonal ou circulares.

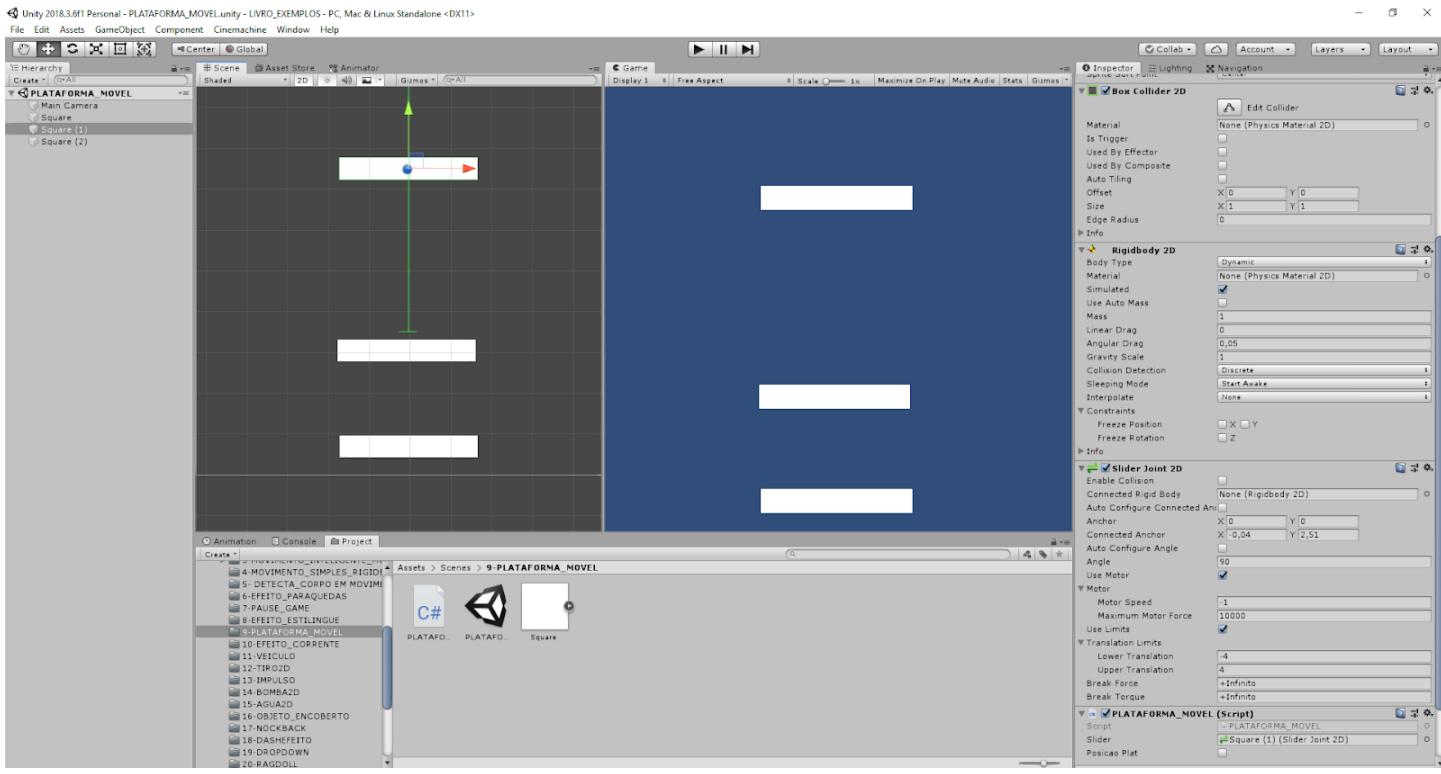
Enfim é uma mecanica que valoriza muito o jogo já que aumenta a dificuldade de uma fase principalmente se a mesma exigir velocidade.

Para criar esse efeito vamos criar uma cena parecida com a da imagem abaixo:



Veja que a cena é muito simples temos apenas sprites retangulares que vão representar as plataformas.

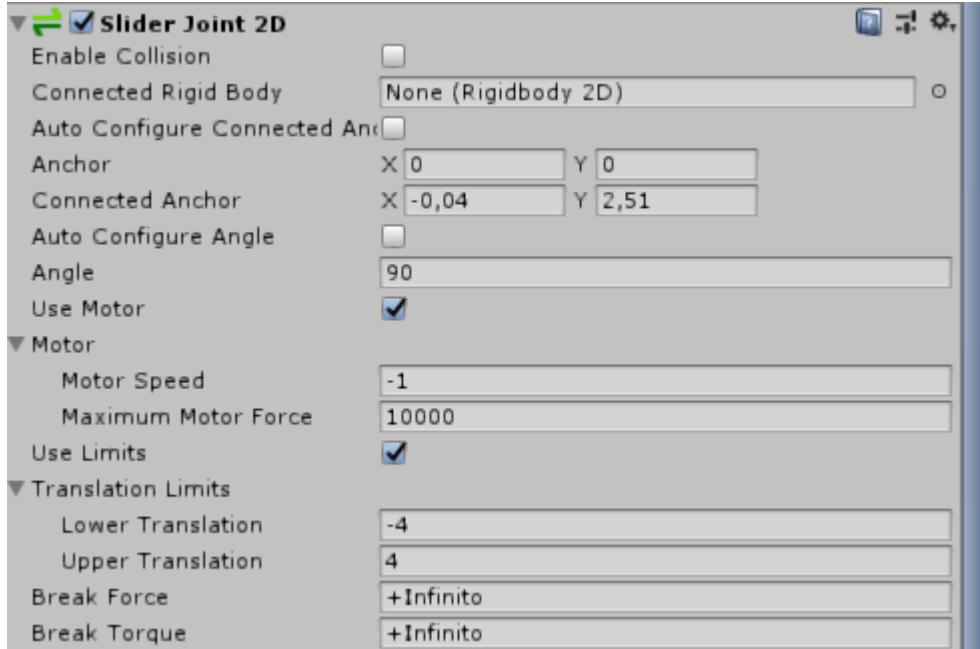
Então vamos analisar profundamente como estão configuradas essas plataformas.



Veja que as plataformas tem um corpo colisor, um **rigidbody2D** e um **SliderJoint2D** alem do seu código que chamamos de **PLATAFORMA_MOVE**

Veja que o Slider joint 2D adicionado nesses objetos tem algumas configurações importantes que foram ajustadas.

Por exemplo o Angle que define o angula de movimento da plataforma, o Use Motor que define se vamos ou não usar o motor para mover a plataforma e por ultima o Use Limits que define limites para o movimento sendo um limite mínimo e outro máximo de movimento.



Visto isso podemos então trabalhar no nosso código que segue abaixo:

```

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

public class PLATAFORMA_MOVEL : MonoBehaviour
{
    public SliderJoint2D slider;
    public JointMotor2D motor2D;
    public bool posicaoPlat;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

        MovimentoPlat(Random.Range(500,10000),Random.Range(0.5f,2f));
    }

    void MovimentoPlat(float torque, float vel)
    {
        string posicao = slider.limitState.ToString();

        if( posicao == "UpperLimit" && posicaoPlat)
        {
            motor2D.motorSpeed = -vel;
            motor2D.maxMotorTorque = torque;
            slider.motor = motor2D;
            posicaoPlat = false;
        }
        else if(posicao == "LowerLimit" && !posicaoPlat)
        {
            motor2D.motorSpeed = vel;
            motor2D.maxMotorTorque = torque;
            slider.motor = motor2D;
            posicaoPlat = true;
        }
    }
}

```

Veja que nesse código a primeira coisa que foi feita é criar as variáveis que vão ser necessárias para o efeito funcionar.

Temos uma variável do tipo SliderJoint2D que serve para manipular o Slider do objeto, depois temos uma variável do tipo JointMotor2D que serve para controlar o motor que cera_2 usado no efeito e por último uma variável booleana que é usada para fazer a plataforma ficar em um looping infinito de movimento.

```
public SliderJoint2D slider;
public JointMotor2D motor2D;
public bool posicaoPlat;
```

O próximo passo é trabalhar no método Update aqui perceba que a coisa é bem simples apenas chamamos o método MovimentoPlat passando alguns parametros para ele.

Sendo o primeiro um range entre valores de 500 até 10000 e depois outro random entre 0.5 e 2 vamos entender o porquê disso mais adiante.

```
void Update()
{
    MovimentoPlat(Random.Range(500,10000),Random.Range(0.5f,2f));
}
```

Agora sim vamos analisar o método que faz toda a magica funcionar, veja que nesse método precisamos de parâmetros para trabalhar na verdade dois parâmetros um de torque e outro de velocidade.

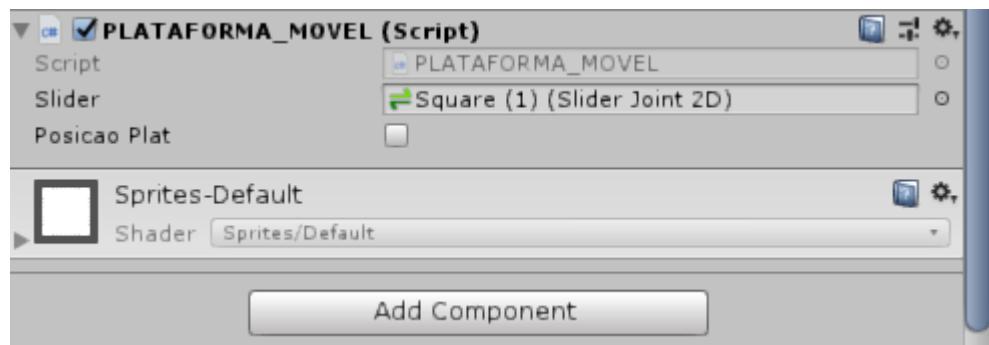
Dentro desse método a primeira coisa que fizemos foi criar uma variável do tipo string com o nome de posição e atribuímos a ela o string de slider.limitState que vai nos retornar dois valores muito importantes que são o limite máximo "UpperLimit" e mínimo "LowerLimit" do efeito.

Em cada condição ajustamos os valores de velocidade do motor assim como o seu torque máximo dessa forma temos o movimento de vai e vem da plataforma.

```
void MovimentoPlat(float torque, float vel)
{
    string posicao = slider.limitState.ToString();

    if( posicao == "UpperLimit" && posicaoPlat)
    {
        motor2D.motorSpeed = -vel;
        motor2D.maxMotorTorque = torque;
        slider.motor = motor2D;
        posicaoPlat = false;
    }
    else if(posicao == "LowerLimit" && !posicaoPlat)
    {
        motor2D.motorSpeed = vel;
        motor2D.maxMotorTorque = torque;
        slider.motor = motor2D;
        posicaoPlat = true;
    }
}
```

Terminado esse código agora só precisamos ir até o Inspector do Unity com cada uma das plataformas selecionada e adicionar o seu Slider no código.



Agora é só executar o exemplo para ver tudo funcionando.

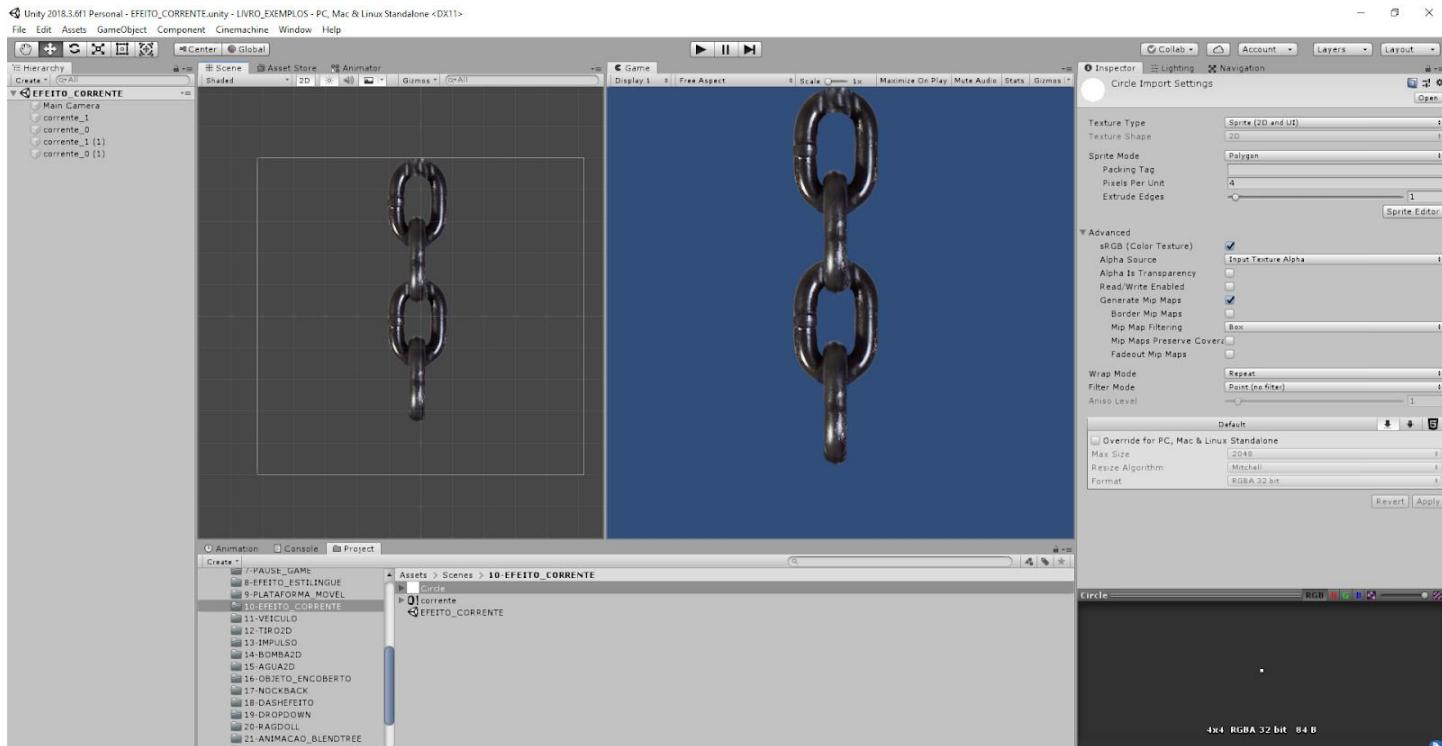
EFEITO CORRENTE

Um efeito muito legal que podemos ver em games que gostam de abusar de efeitos físicos é o efeito de correntes.

Nada mais legal do que ver aqueles objetos se movendo como se realmente estivessem ligados por elos de metal.

Então vamos ver como criar esse efeito.

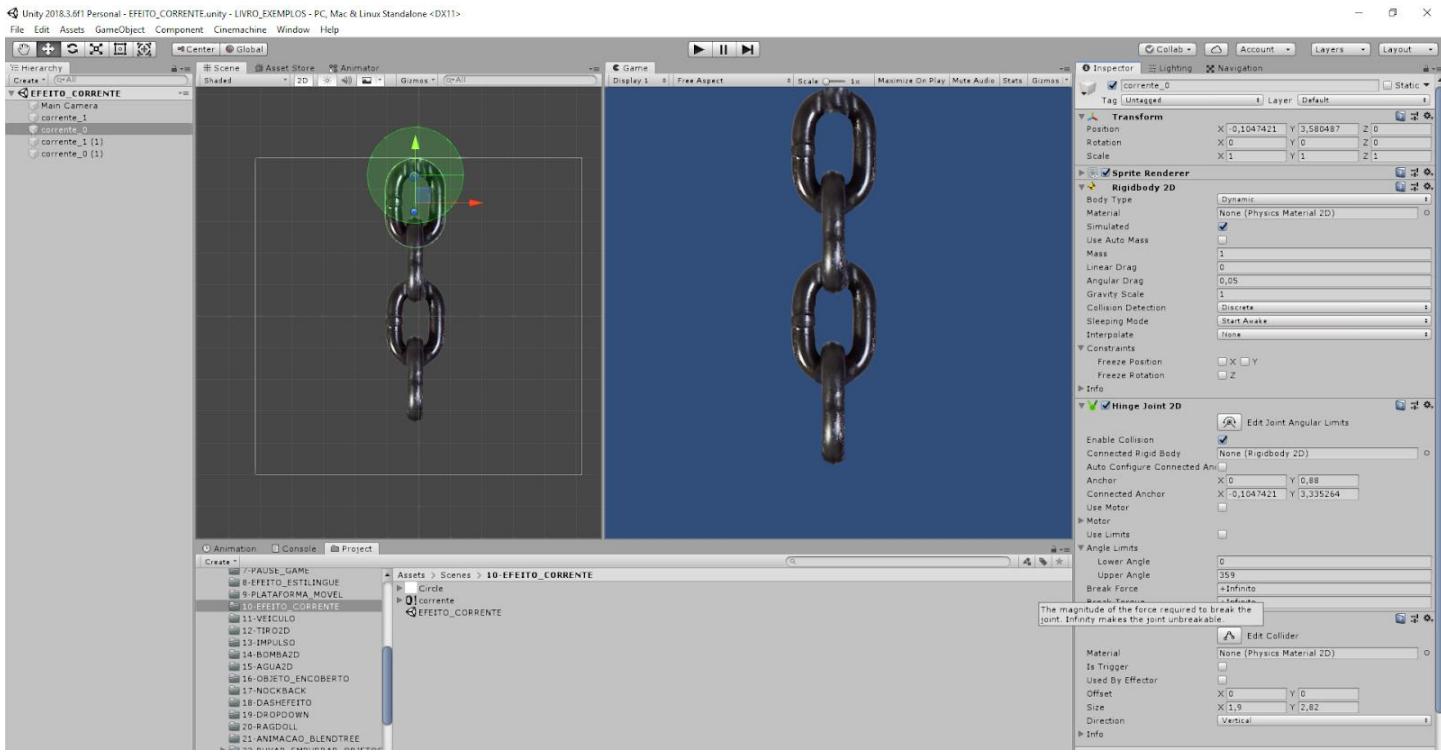
Para isso crie uma cena parecida com a da imagem abaixo:



Veja que nesse caso estamos usando imagens reais de elos de uma corrente, no caso precisamos apenas de dois elos de ligação.

Dessa forma podemos arrastar essas sprites para a cena e ajustar cada uma delas.

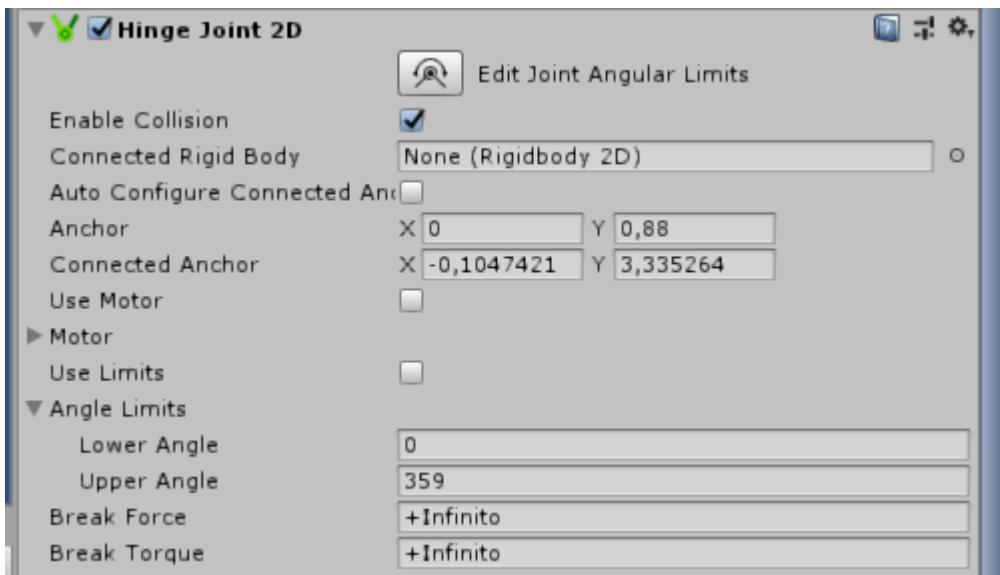
Veja que na imagem abaixo selecionei o primeiro elo da corrente e dessa forma conseguimos ver seus componentes.



Veja que aqui temos um corpo colisor que seja indicado para a sprite em questão, no caso o colisor que melhor se encaixa nessa imagem é uma capsula.

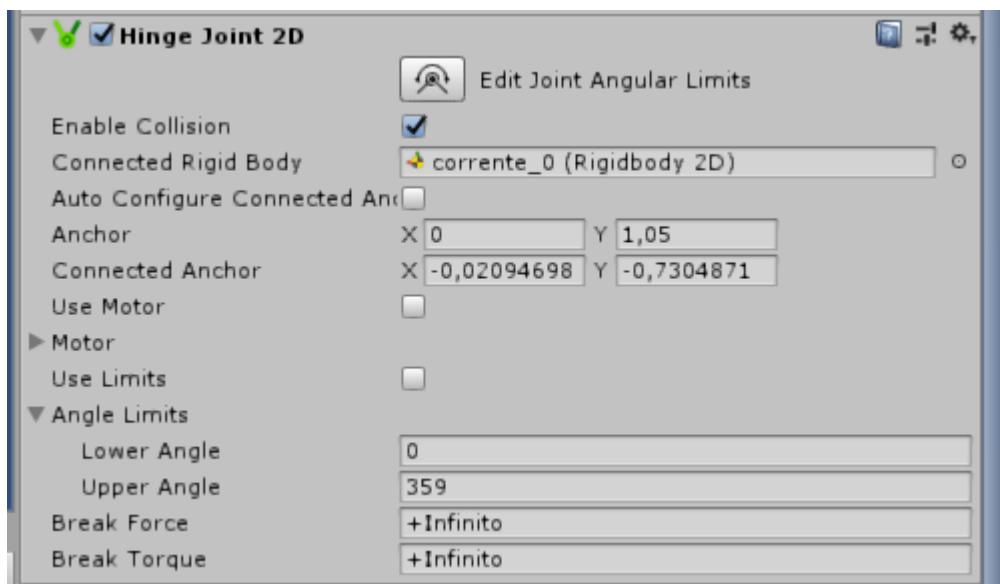
Depois temos nosso Rigidbody2D e por último temos o HingeJoint2D que é quem nos dá a possibilidade de ligar os objetos de uma forma que o efeito de corrente seja possível.

Agora vamos dar uma olhada nas configurações do HingeJoint2D.



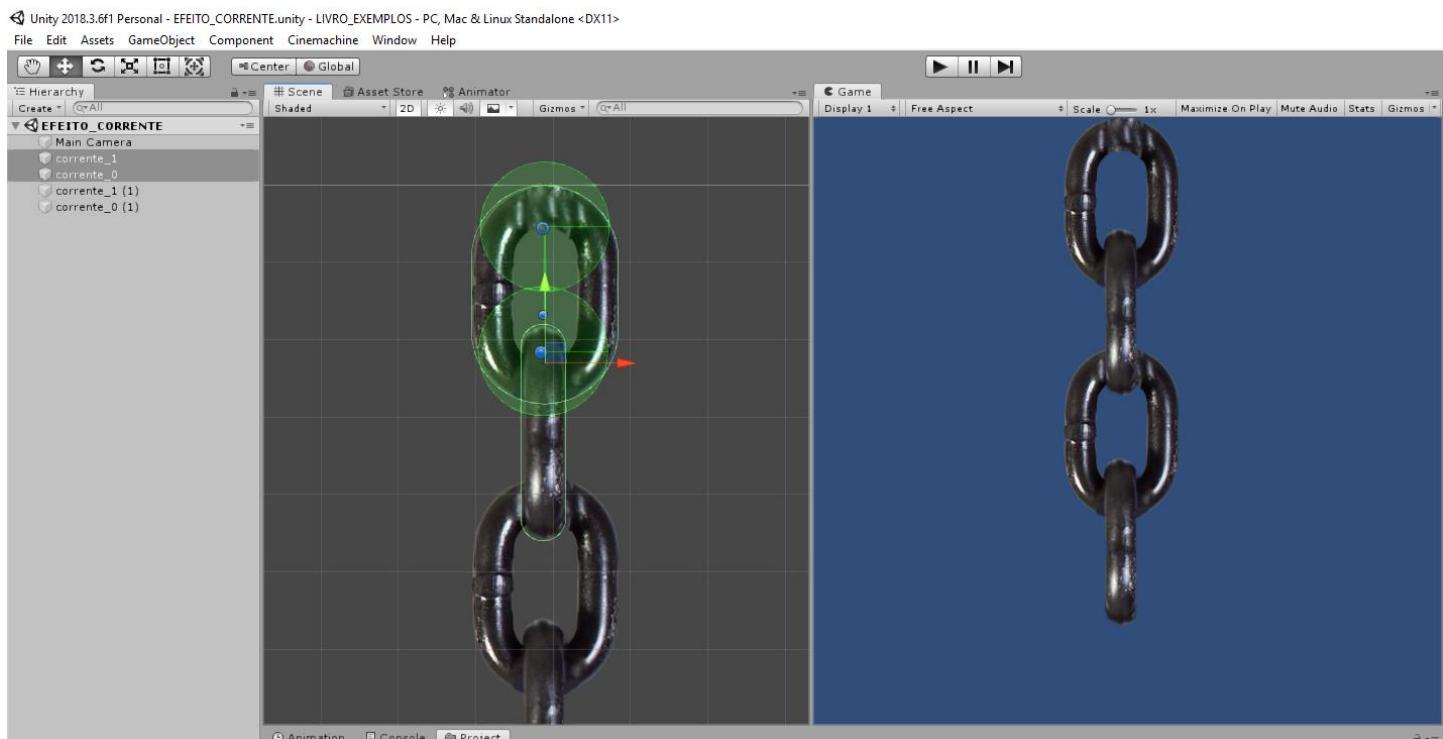
Perceba que nessas configurações definimos Enable Collision como verdadeiro pois queremos que a colisão seja respeitada entre os elos da corrente.

Mas esses ajustes foram feitos no primeiro elo de corrente então existem diferenças entre essas configurações e as configurações dos outros elos, veja:



As configurações da imagem acima são do segundo elo de corrente, perceba que nessas configurações também temos a opção de colisão habilitada porém em Connected Rigid Body temos um objeto definido como objeto de ligação.

No caso o segundo elo da corrente está ligado no primeiro elo assim como uma corrente no mundo real veja:



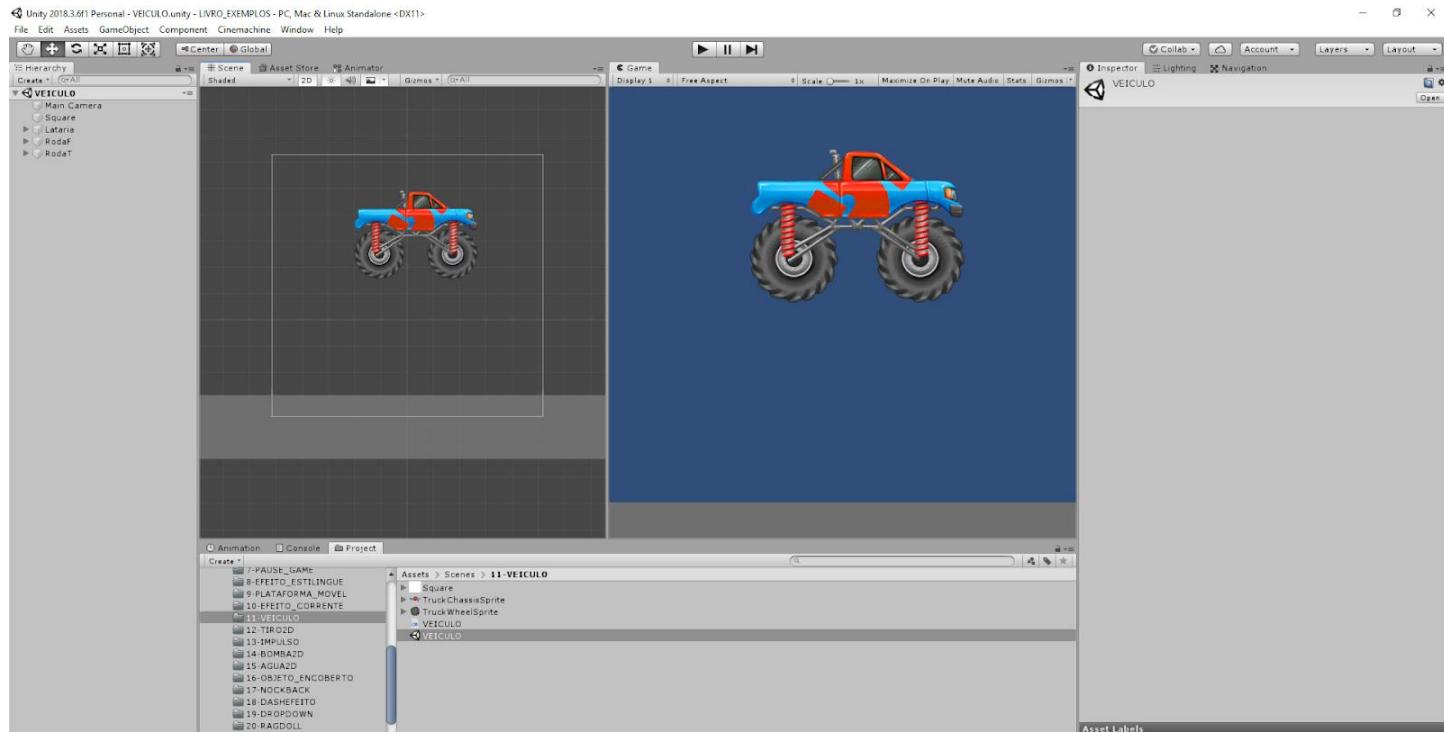
E esse processo deve ser feito em todos os outros elos sempre ligando o elo mais baixo com o seu antecessor, feito isso o exemplo vai funcionar sem nenhum problema.

VEICULO 2D

Quem nunca teve vontade de criar um jogo de corrida, mesmo que em 2D?

Acho que todo mundo jogos que simulam mecânica de carros são muito legais e acredite muito tranquilos de se criar.

Então vamos criar um exemplo, para isso crie uma cena semelhante a da imagem abaixo:



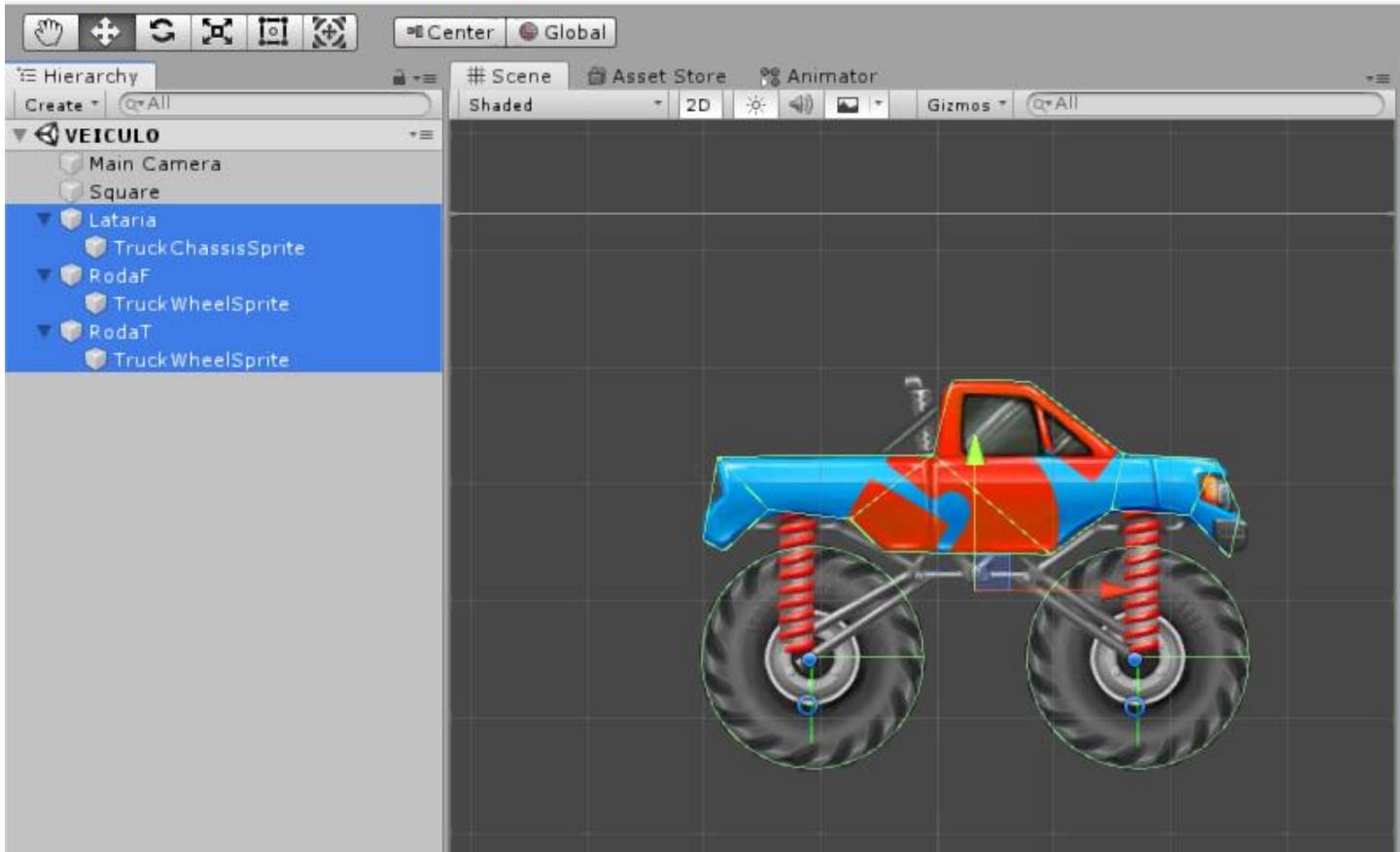
Veja que nela temos uma sprite para o chão e um carro já montadinho pronto para andar, então vamos ver como esse carro foi montado.

Veja na imagem abaixo que temos vários objetos de jogo para compor esse carro, é importante deixar claro que cada objeto desse carro foi feito seguindo a mesma lógica.

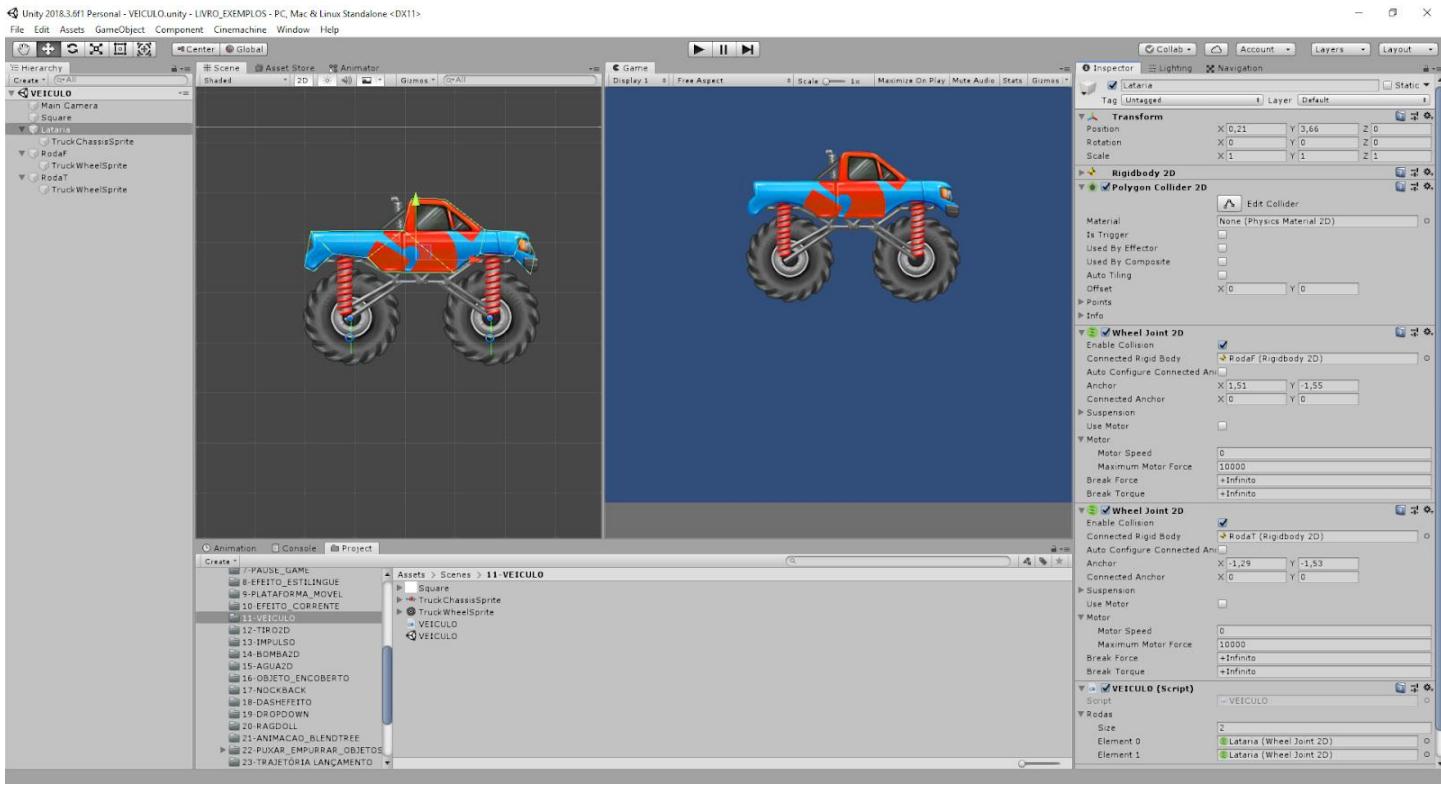
Um Gameobject vazio e dentro dele a sprite que representa uma parte do carro.

No caso o Gameobject vazio com o nome de Lataria tem como filho a sprite da lataria do carro.

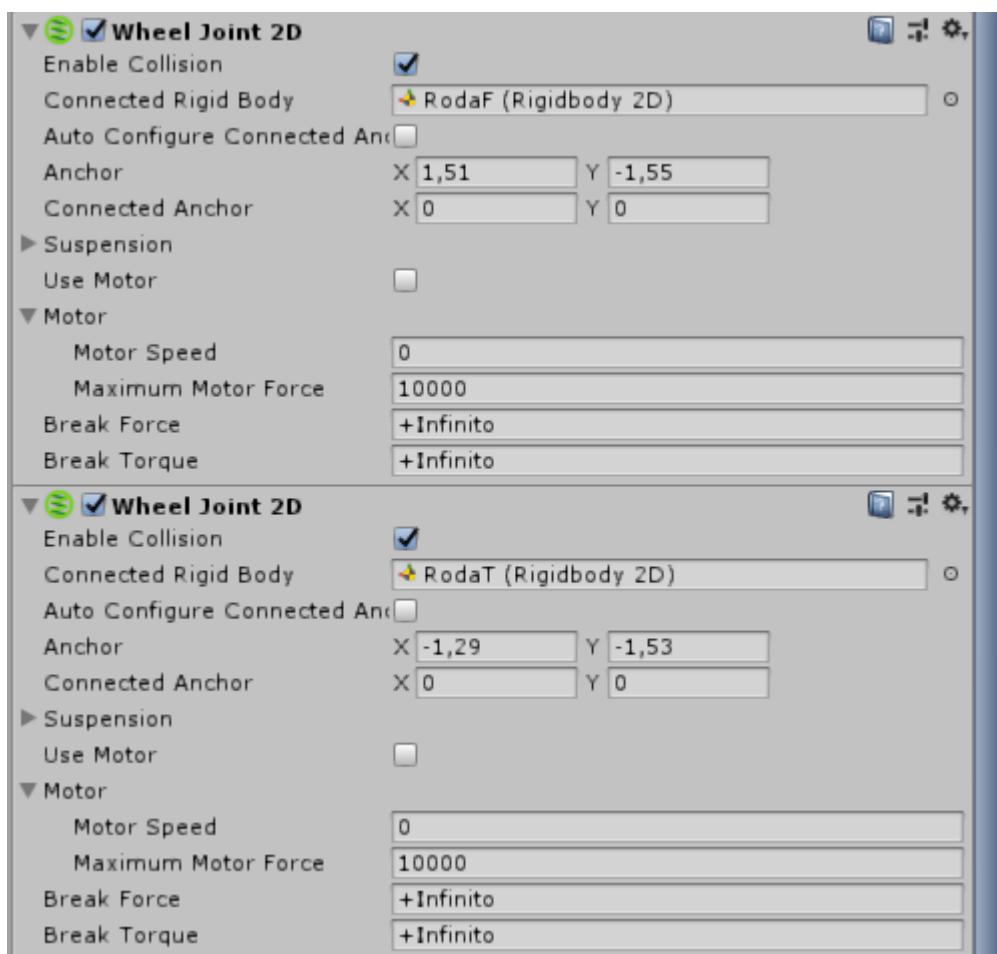
O Gameobject vazio com o nome de RodaF tem como filho a sprite de uma roda e por último o gameobject vazio RodaT também tem uma sprite da roda como objeto filho.



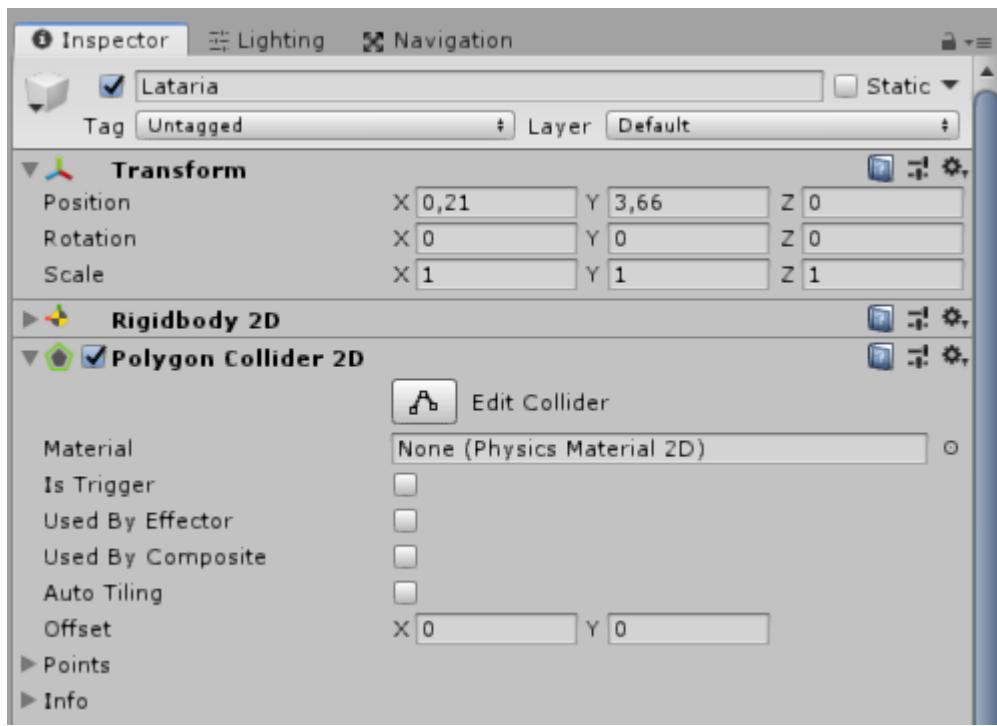
Agora vamos ver as configurações de cada objeto separadamente, vela na imagem abaixo que o Gameobject vazio Lataria tem um Rigidbody2D, dois WheelCollider, um Polygon Collider e um código com o nome de **VEÍCULO**.



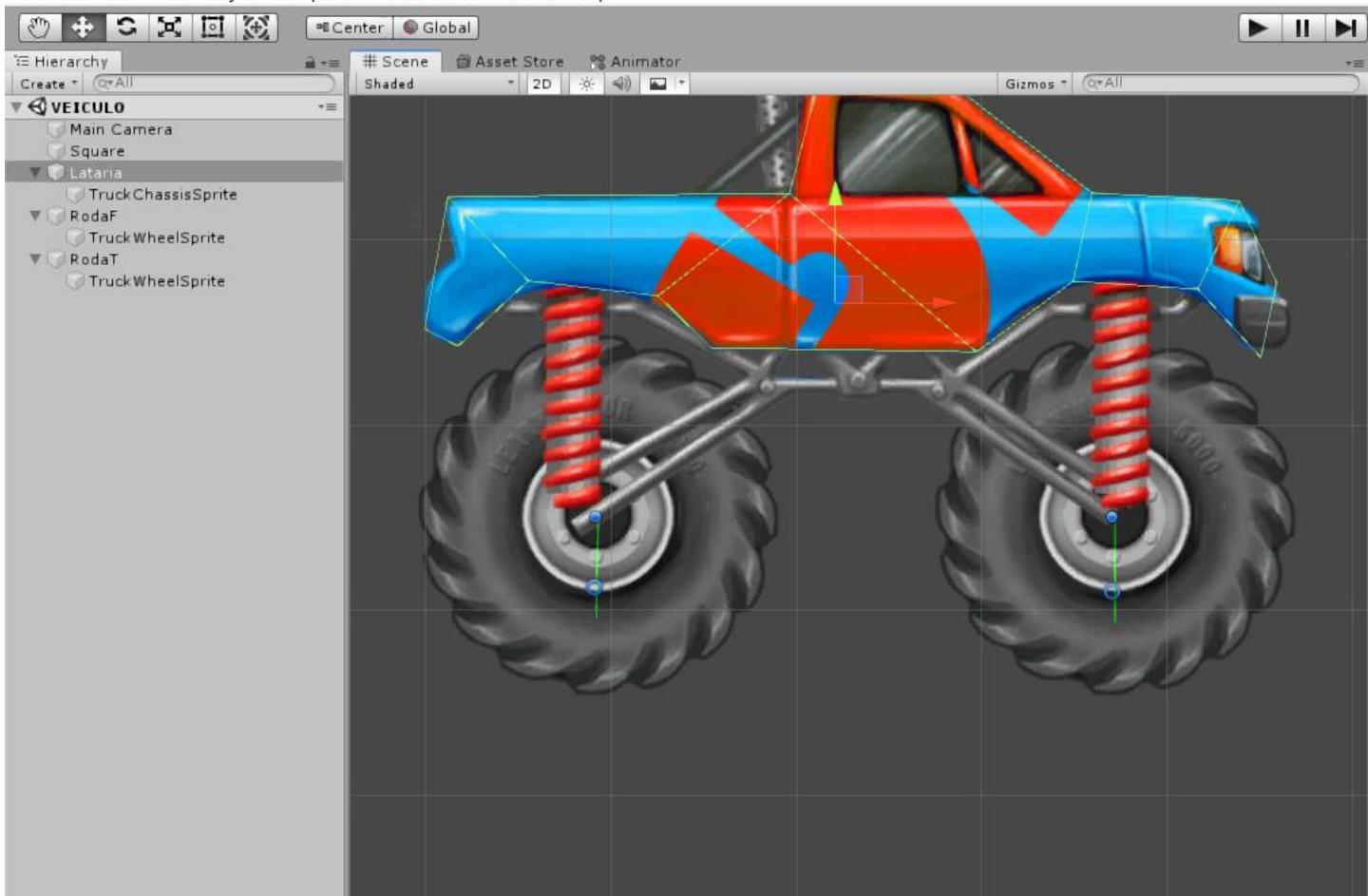
Aproximando um pouco mais da lista de componentes da Lataria podemos ver que cada WheelCollider2D tem duas configurações muito importantes que foram ajustadas logo de inicio. O Enable Collision para garantir a colisão e o Connect Rigidbody que em um WheelCollider eu coloco o RodaF e no outro o RodaT para simbolizar as duas rodas do carro.



Depois temos um corpo colisor que é o Polygon que nos permite desenhar as linhas de colisão no objeto.



E com esses ajustes agora precisamos fazer um ajuste mais manual nos dois WheelCollider veja:



Repare que nas rodas desse carro existem pontos de controle que são usados para definir o quando o carro vai ter um amortecimento das rodas.

Configurando da forma que a imagem acima teremos um carro com uma suspensão leve e muito interessante.

O sprite dentro do objeto Lataria não sofre nenhuma configuração.

Nas rodas temos apenas um collider e um Rigidbody2D o objeto filho das rodas não tem nenhuma configuração adicionada.



Certo agora nosso carro está devidamente configurado então vamos escrever o código que vai no objeto Lataria.

```

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

public class VEICULO : MonoBehaviour
{
    public JointMotor2D motorCarro;
    public WheelJoint2D[] rodas;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float vel = Input.GetAxis("Horizontal");
        motorCarro.motorSpeed = vel * 1000;
        motorCarro.maxMotorTorque = 10000;
        rodas[0].motor = motorCarro;
        rodas[1].motor = motorCarro;
    }
}

```

Veja que nesse código temos duas variáveis uma do tipo JointMotor2D para controlar o motor do carro e outra do tipo WhellCollider2D que serve para trabalhar sobre as rodas do veículo.

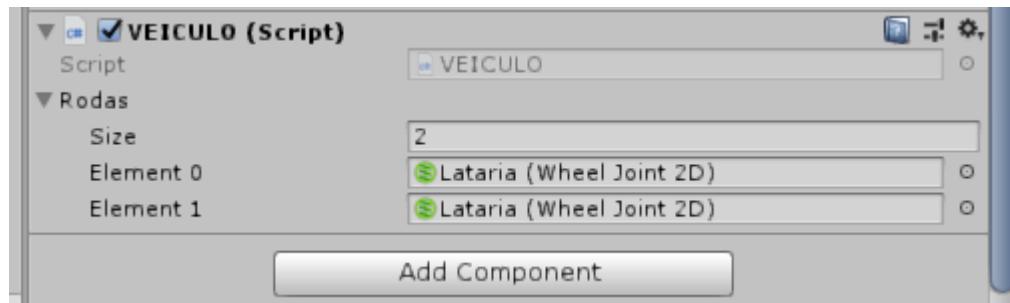
```
public JointMotor2D motorCarro;
public WheelJoint2D[] rodas;
```

Depois dentro do método Update temos uma variável do tipo float com o nome de vel que recebe o valor do input horizontal.

Depois usamos esse valor para ajustar a velocidade do motor assim como seu torque para as duas rodas que são representadas pelo array rodas.

```
void Update()
{
    float vel = Input.GetAxis("Horizontal");
    motorCarro.motorSpeed = vel * 1000;
    motorCarro.maxMotorTorque = 10000;
    rodas[0].motor = motorCarro;
    rodas[1].motor = motorCarro;
}
```

Com isso vamos retornar ao Unity e fazer um último ajuste no código desse veículo.

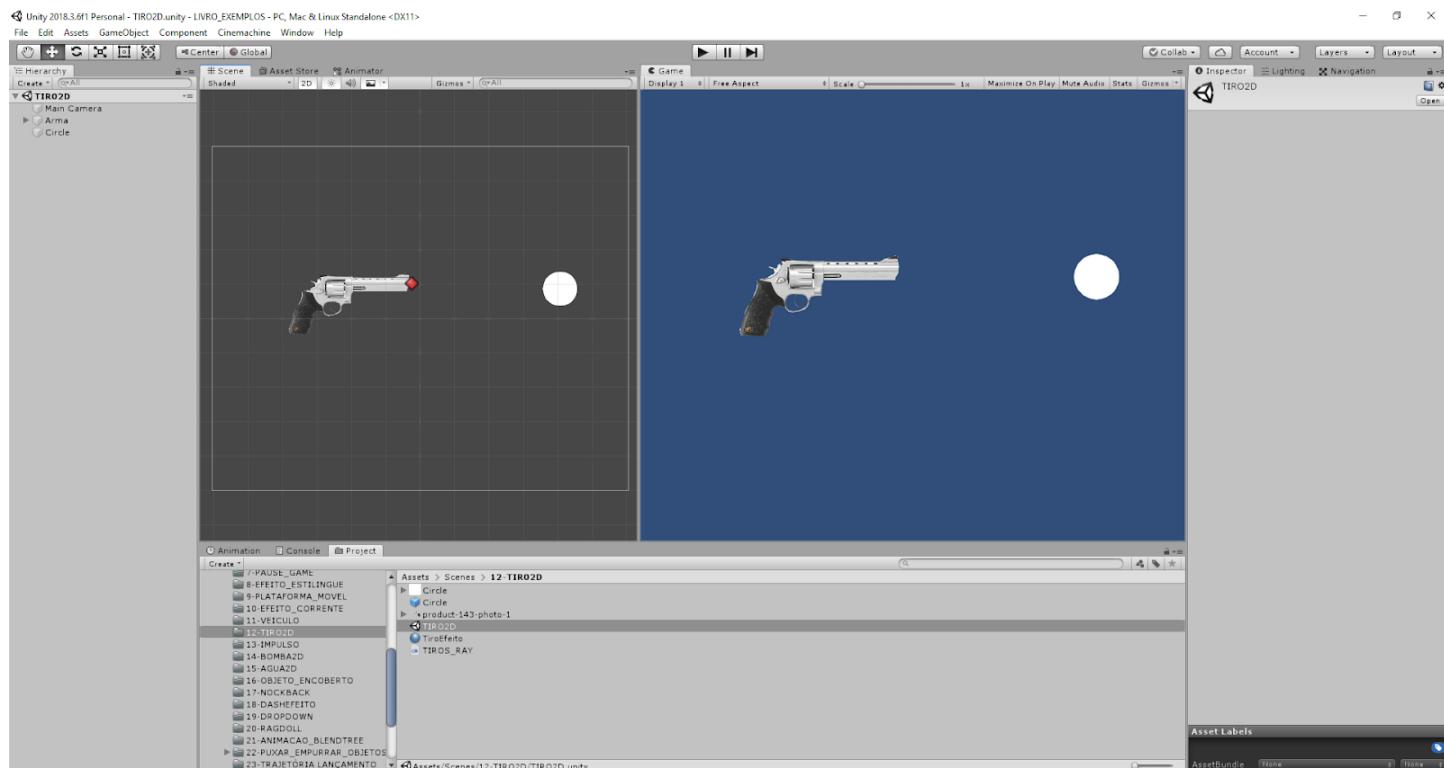


Veja que passamos para o array os dois WheelJoint2D do nosso carro agora sim o exemplo está pronto para funcionar.

TIRO 2D

Outra mecânica muito comum em games é a de tiros e existem várias formas de criar esse tipo de efeito o tradicional é criar objetos na cena que vão se comportar como verdadeiras balas de revólver sendo disparadas na direção desejada.

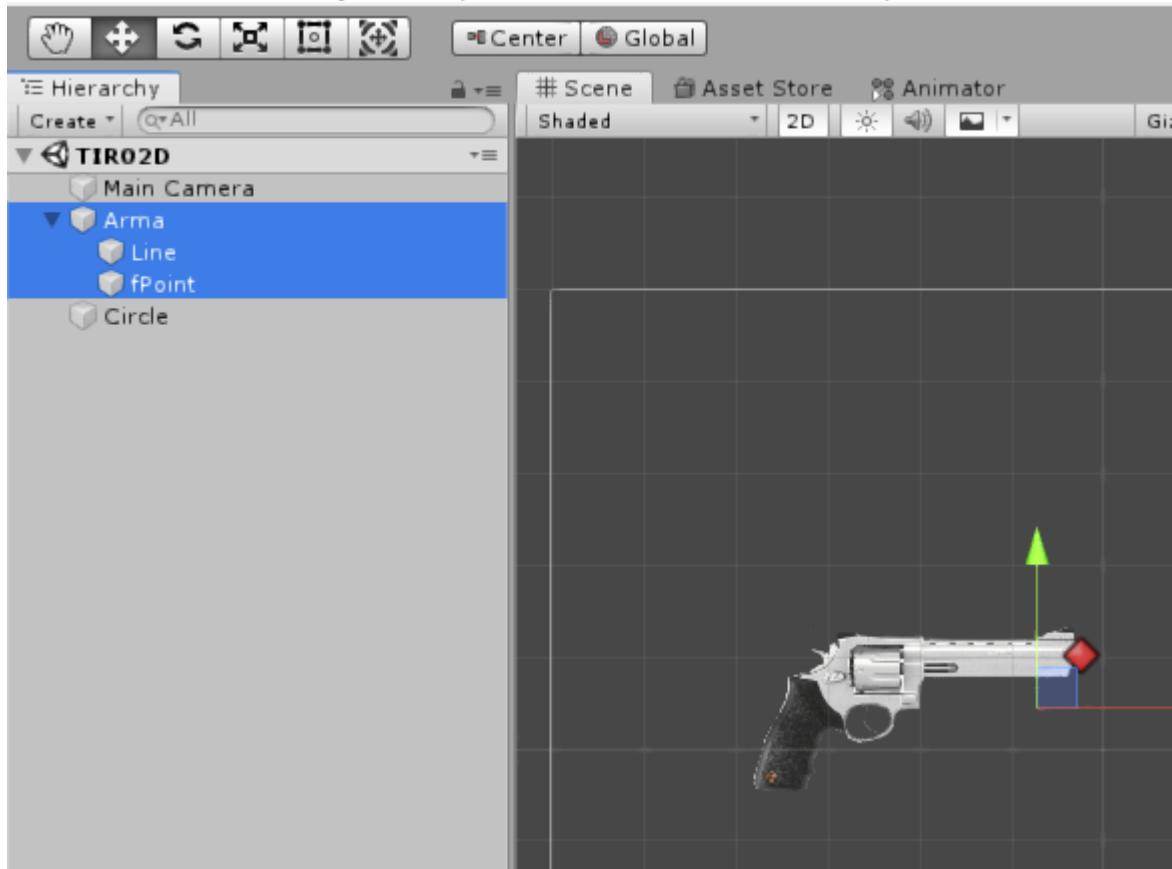
Mas nesse exemplo eu vou mostrar algo diferente, então para esse exemplo crie uma cena semelhante a da imagem abaixo:



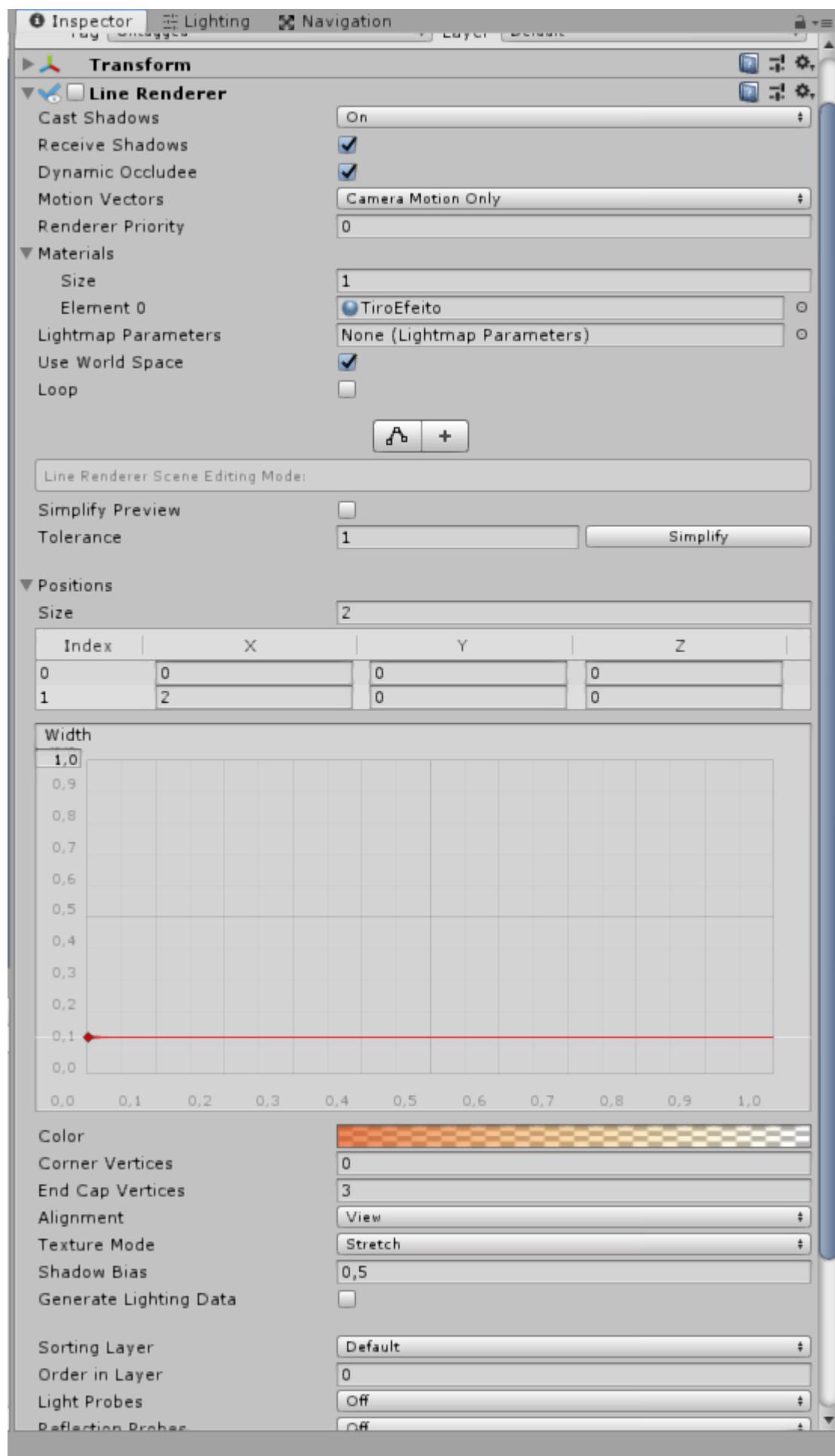
Veja que nessa cena tenho apenas duas sprites uma de um revólver e outra de uma esfera que vai ser atingida pelos tiros.

O revólver é composto por alguns objetos como por exemplo um gameobject vazio chamado fPoint que é de onde o tiro é disparado e um objeto chamado Line que é o LineRender que usamos para esse efeito.

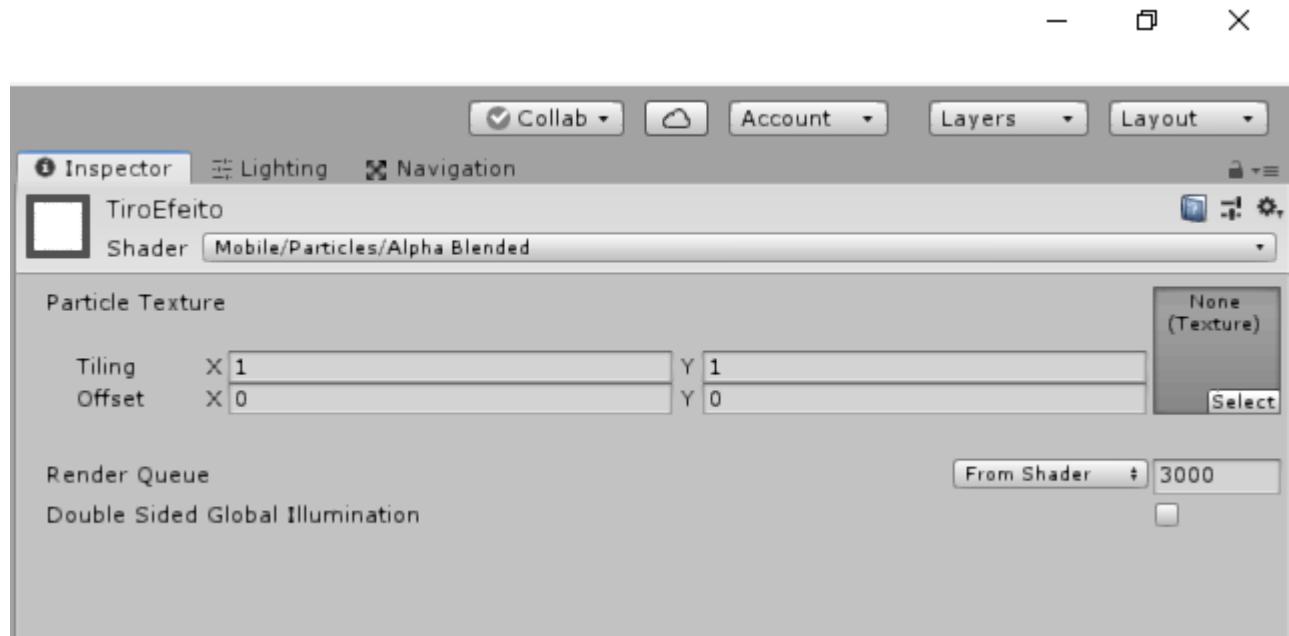
O fPoint é fácil de enxergar ele é representado pelo ícone de diamante vermelho que fica no cano da arma.



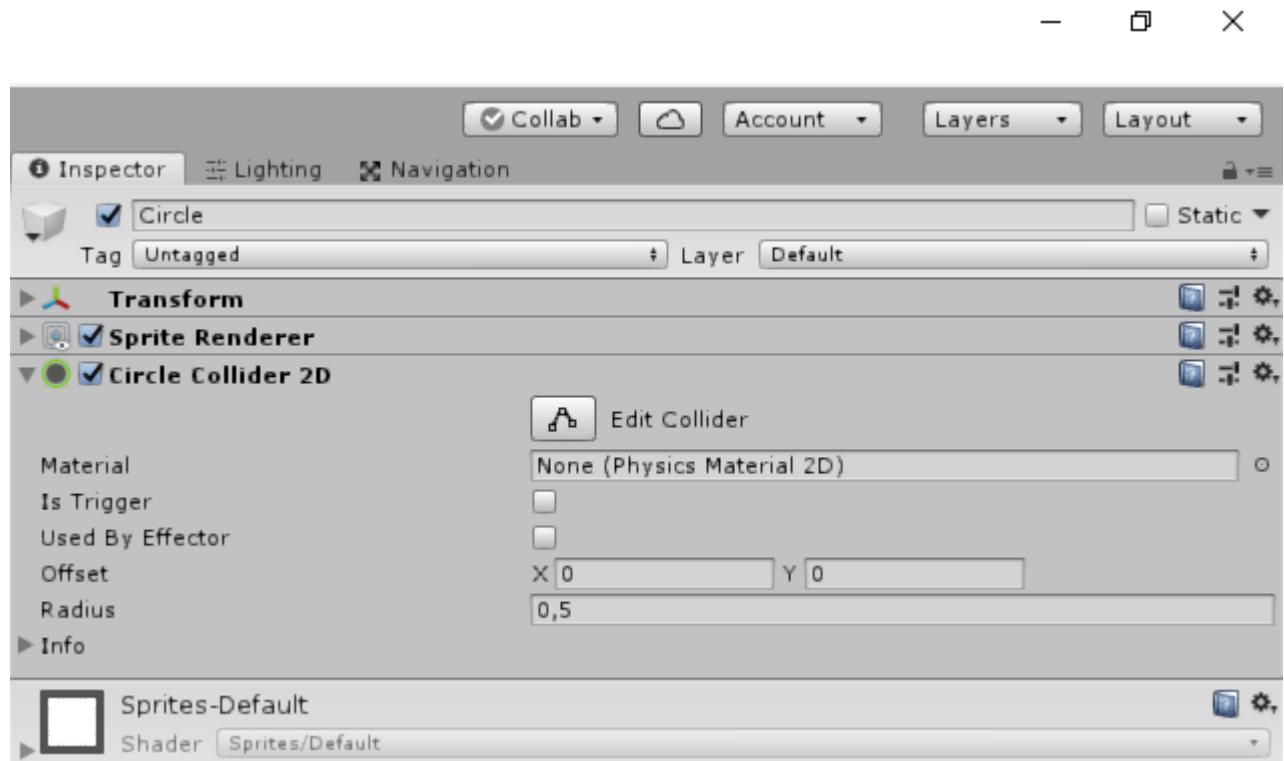
Já o LineRender fica invisível como mostra a imagem abaixo:



E veja que nesse LineRender temos as configurações de visualização do mesmo onde definimos o material usado, a cor, ajuste das posições, enfim. O material usado nesse LineRender é esse aqui:



Veja que não é nada demais apenas um material de partículas sem nenhuma imagem como textura. A esfera que vai ser atingida tem apenas um corpo colisor como você pode ver na imagem abaixo:



E com essas configurações devidamente ajustadas, basta criar um arquivo de código chamado TIROS_RAY adicionar dentro do objeto Arma e escrever o seguinte código:

```

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

public class TIROS_RAY : MonoBehaviour
{
    public LineRenderer lineRenderer;
    public Transform firePoint;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if(Input.GetButtonDown("Fire1"))
        {
            StartCoroutine(Tiro());
        }
    }

    IEnumerator Tiro()
    {
        RaycastHit2D hit =
Physics2D.Raycast(firePoint.position,firePoint.right);
        if(hit)
        {

            lineRenderer.SetPosition(0,firePoint.position);
            lineRenderer.SetPosition(1,hit.point);

        }
        else
        {
            lineRenderer.SetPosition(0,firePoint.position);
            lineRenderer.SetPosition(1,firePoint.position +
firePoint.right * 100);
        }

        lineRenderer.enabled = true;

        yield return 0;

        lineRenderer.enabled = false;
    }
}

```

Veja que a primeira coisa que fizemos aqui foi criar as variáveis que vamos usar no exemplo. temos uma variável do tipo LineRenderer que nos permite manipular a visualização do Linerender. Depois temos outra variável do tipo Transform que nos informa a posição o objeto fPoint.

```
public LineRenderer lineRenderer;
public Transform firePoint;
```

Agora veja que no método Update temos uma estrutura condicional que verifica se apertamos o botão esquerdo do mouse.

Se realmente apertamos esse botão vamos iniciar a corotina chamada Tiro.

```
void Update()
{
    if (Input.GetButtonDown("Fire1"))
    {
        StartCoroutine(Tiro());
    }
}
```

Agora vamos analisar essa corotina.

Veja que aqui usamos um Raycast para disparar nossos tiros da posição firePoint para a direita e na sequencia ajustamos o lineRender para funcionar nas posições definidas por setPosition.

Nesse primeiro caso o raio é lançado e para ao colidir com alguma coisa já depois do else deixamos o raio passar direto dando a ilusão que o tiro foi disparo sem atingir ninguém então simplesmente ele some da nossa vista.

```
IEnumerator Tiro()
{
    RaycastHit2D hit =
Physics2D.Raycast(firePoint.position, firePoint.right);
    if (hit)
    {

        lineRenderer.SetPosition(0, firePoint.position);
        lineRenderer.SetPosition(1, hit.point);

    }
    else
    {
        lineRenderer.SetPosition(0, firePoint.position);
        lineRenderer.SetPosition(1, firePoint.position +
firePoint.right * 100);
    }

    lineRenderer.enabled = true;
}
```

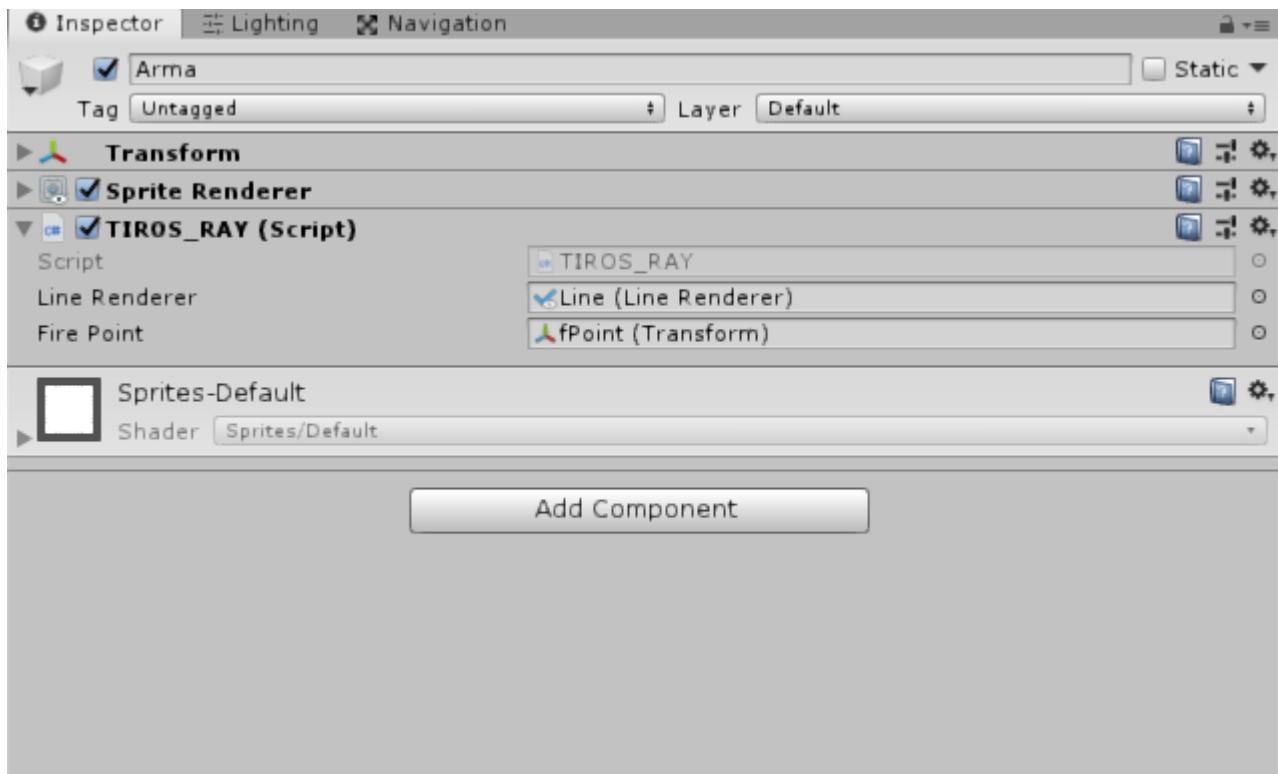
```

        yield return 0;

        lineRenderer.enabled = false;
    }
}

```

O código é apenas isso nada demais, então vamos fazer os últimos ajustes nele porem, lá no Inspector.

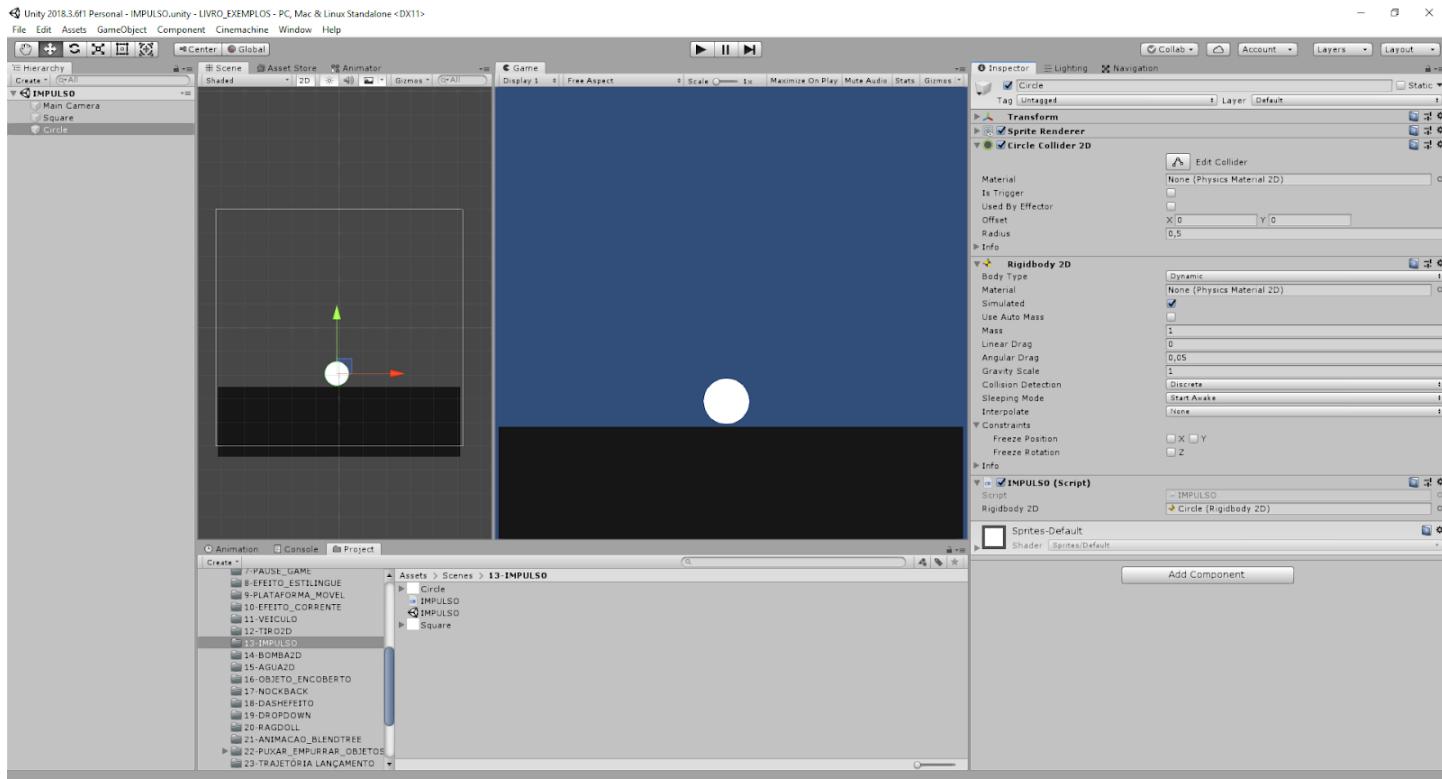


Veja que como mostra a imagem acima apenas passamos para o código quem é o objeto FirePoint e quem é o lineRender.

Feito isso é só executar e ver o efeito final.

IMPULSO 2D

Outro exemplo comum no mundo dos games é a adição de impulso em um determinado objeto, mesmo sendo algo simples muita gente tem problemas em fazer tal efeito quando estão iniciando. Então vamos ver como criar esse efeito, antes de qualquer coisa crie uma cena semelhante a esta:



Veja que nessa cena temos apenas dois objetos um representando o chão e outro uma esfera que vai receber o impulso.

Vale lembrar que esses objetos tem corpo colisor e a bola tem também um rigidbody2D.

Sendo assim para que o efeito funcione só precisamos de um código que vamos criar com o nome de **IMPULSO**.

Dentro desse arquivo de código vamos adicionar as seguintes linhas:

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

public class IMPULSO : MonoBehaviour
{
    public Rigidbody2D rigidbody2D;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
}
```

```

void Update()
{
    if(Input.GetKeyDown(KeyCode.Space) )
    {
        rigidbody2D.AddForce(transform.up * 1000) ;
    }

}

```

Nesse código temos apenas uma variável que é a do Rigidbody2D na esfera. E logo depois em Update temos a verificação da tecla espaço se tivermos apertado essa tecla adicionamos uma força na bola fazendo ela subir.

```

void Update()
{
    if(Input.GetKeyDown(KeyCode.Space) )
    {
        rigidbody2D.AddForce(transform.up * 1000) ;
    }

}

```

Agora basta ir no Unity onde esse código esta adicionado a esfera e passar o Rigidbody para ele assim o código vai funcionar perfeitamente.

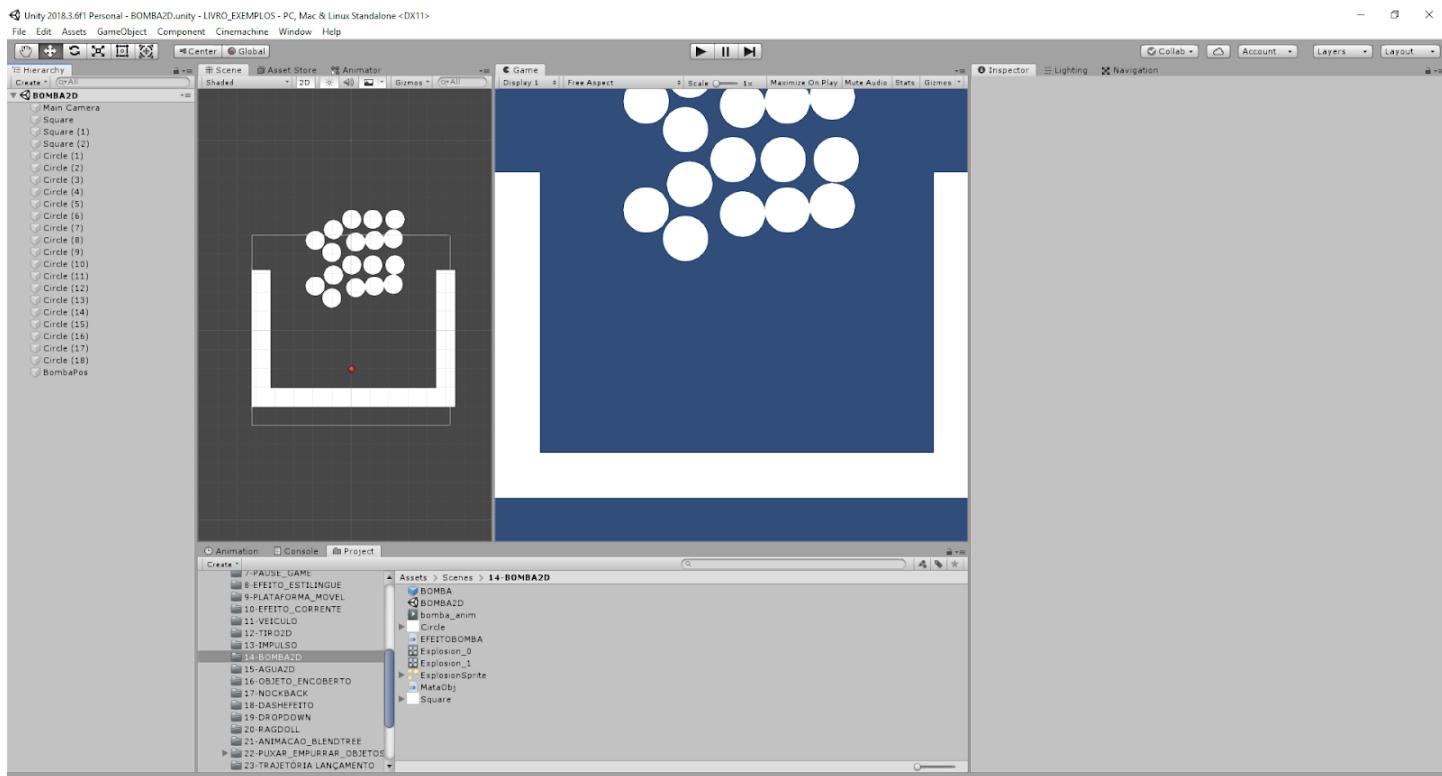
BOMBA 2D

Outro efeito muito bacana para se usar em games é o de bomba ou explosão.

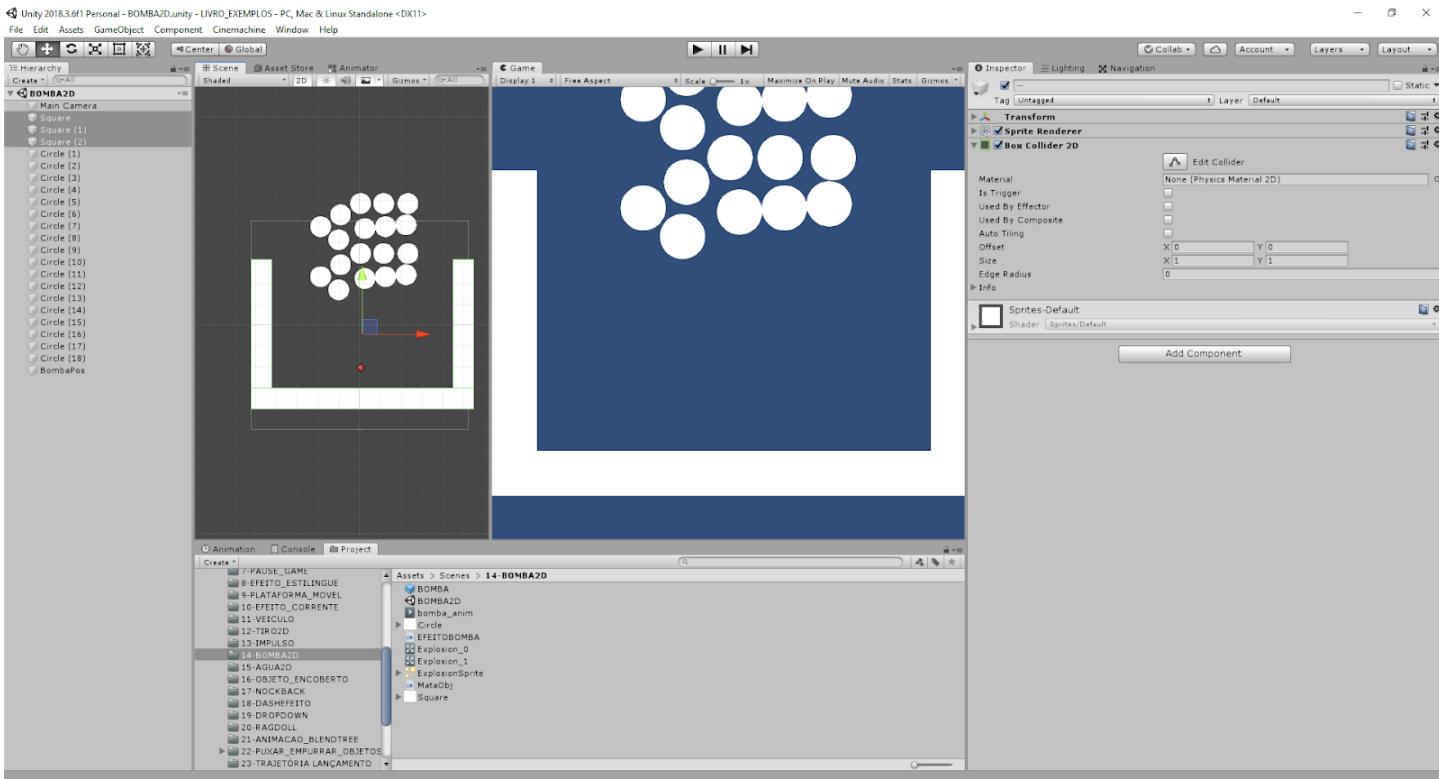
Nada mais belo que ver vários objetos sendo lançados por causa da força de uma explosão.

E é isso o que vamos ver nesse exemplo como criar uma bomba 2D.

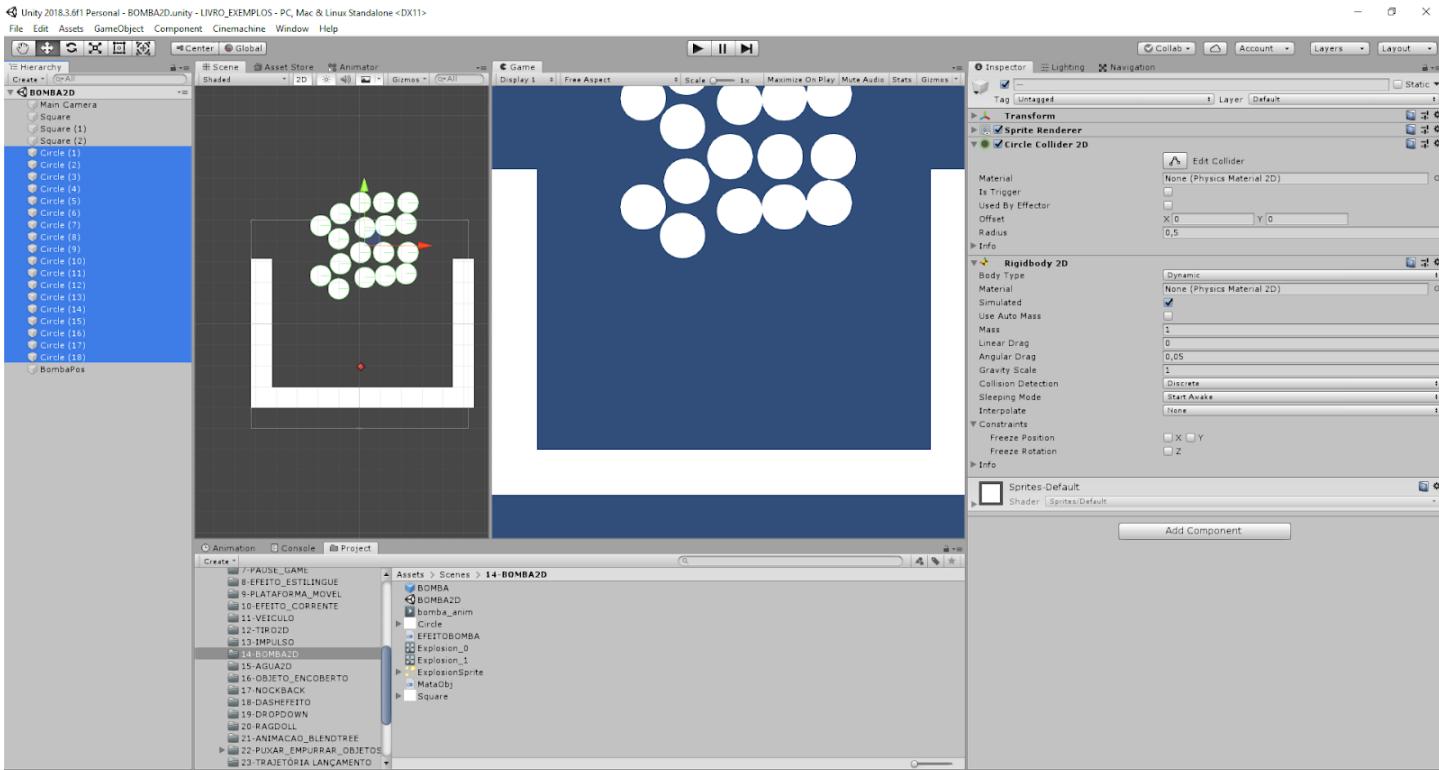
Então já sabe para criar esse efeito precisamos de uma nova cena parecida com essa:



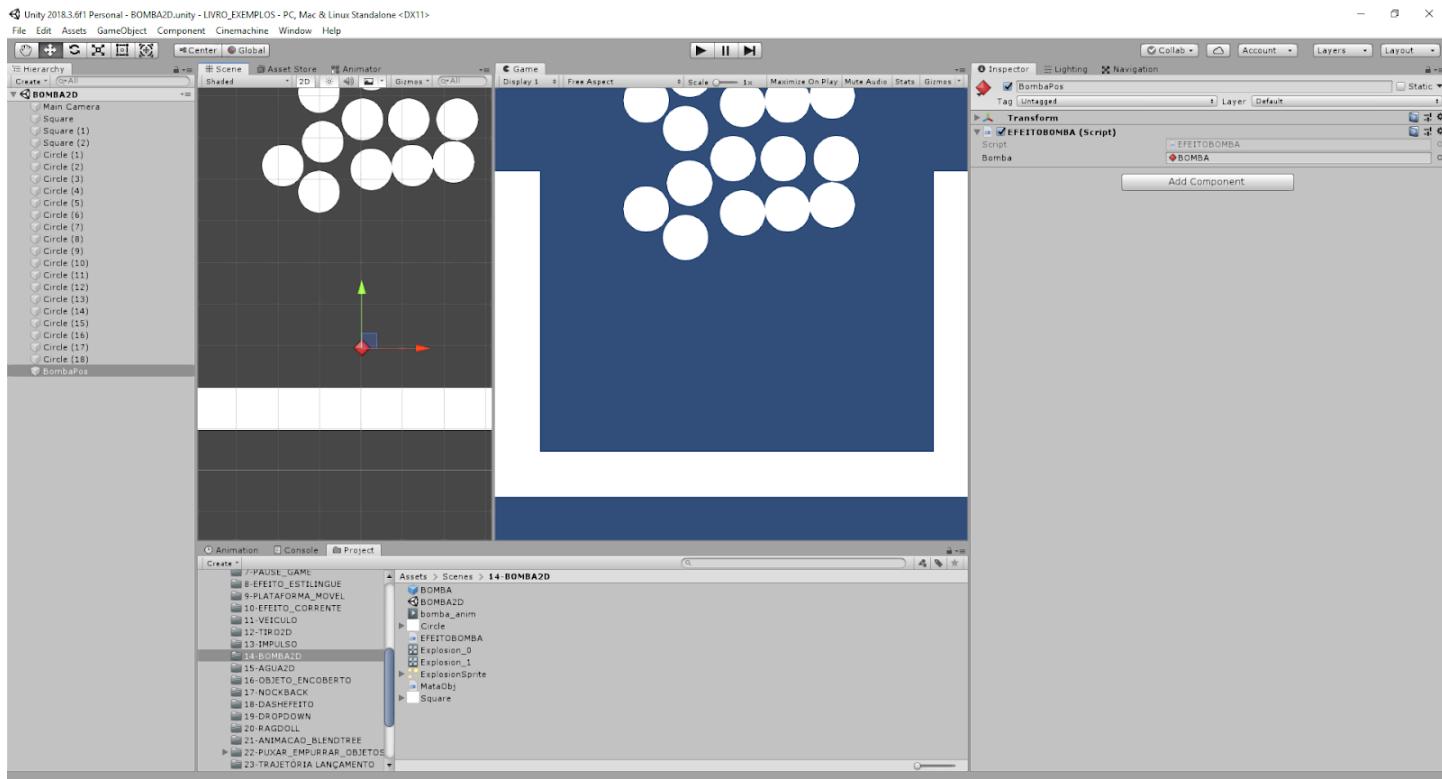
Veja que nessa cena temos retângulos que servem de chão e paredes essas estruturas tem apenas um corpo colisor como mostra a imagem abaixo:



As esferas também têm um corpo colisor e um rigidbody2D para que possam sofrer os efeitos físicos desse exemplo.



Também temos em cena um gameobject vazio que representa a bomba e carrega o código desse efeito que se chama **EFEITOBOOMBA**.



Legal a cena já foi praticamente toda esclarecida mas ainda falta uma coisa a animação de explosão. Essa animação usa a imagem de explosão abaixo e essa animação foi transformada em um prefab para que possamos criar varios objetos com essa animação em cena.



Lembre se que para criar um prefab basta arrastar um objeto da aba Hierarchy para a pasta Project.

Ok agora que temos o arquivo da animação de explosão vamos escrever nosso código.

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

```

public class EFEITOBOOMBA : MonoBehaviour
{
    public GameObject bomba;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Instantiate(bomba, transform.position, Quaternion.identity);
        }
    }
}

```

Nesse código usamos apenas uma variável do tipo GameObject que é a representação da bomba que será criada em cena.

```
public GameObject bomba;
```

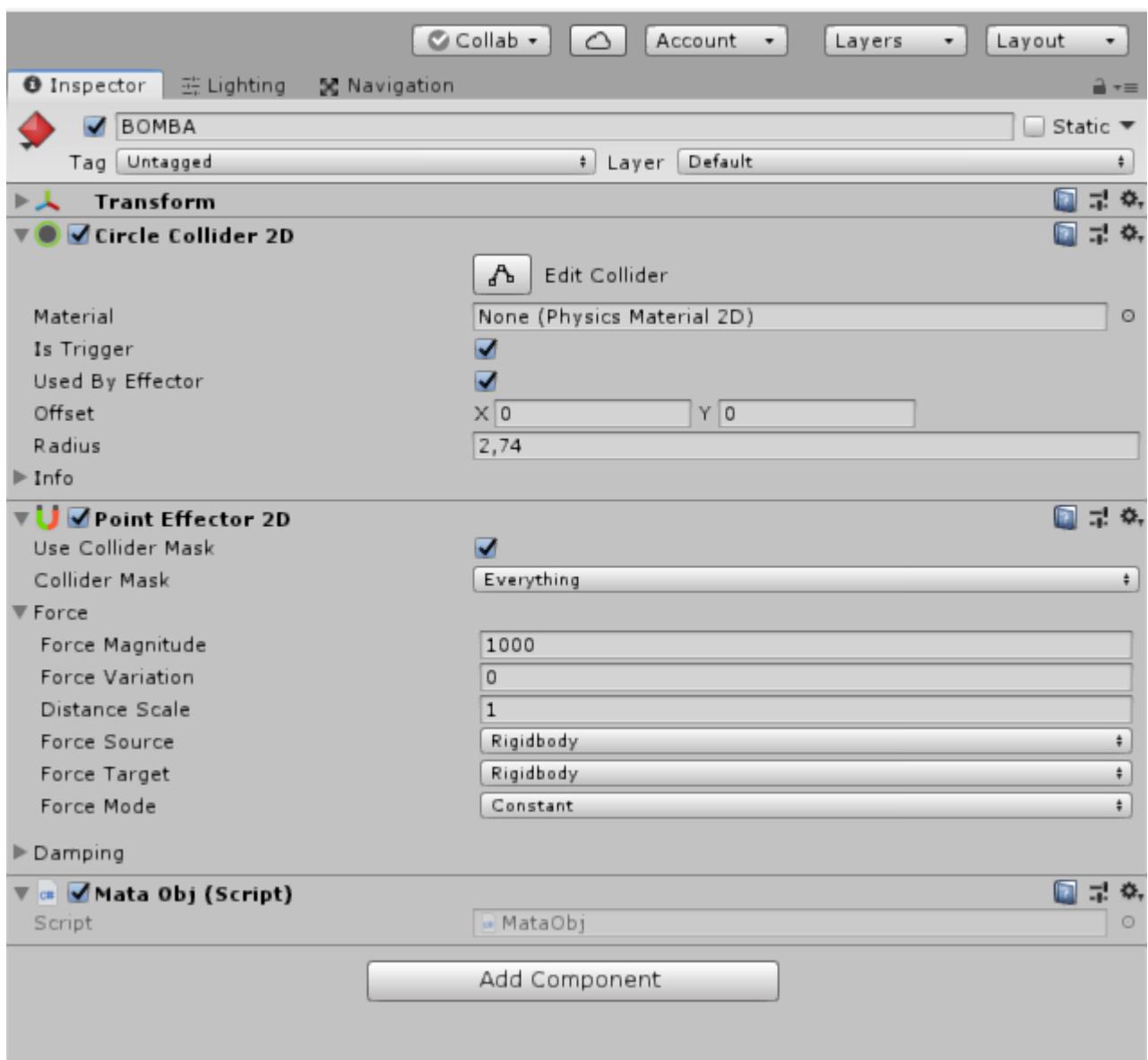
Depois disso no método Update criamos uma estrutura condicional que verifica se apertamos a tecla espaço caso essa condição seja satisfeita criamos uma bomba na cena.

```

void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Instantiate(bomba, transform.position, Quaternion.identity);
    }
}

```

O objeto que é criado na cena tem a seguinte configuração:



Veja que ele tem um corpo colisor e um Point Effector que é quem é responsável pelo efeito de bomba, repare também que nesse objeto existe um código chamado MataObj esse código tem a função de destruir a bomba pouco tempo depois de ser criada.

Veja o código:

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

public class MataObj : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
    }
}
```

```
{  
    Destroy(gameObject, 0.5f);  
}  
}
```

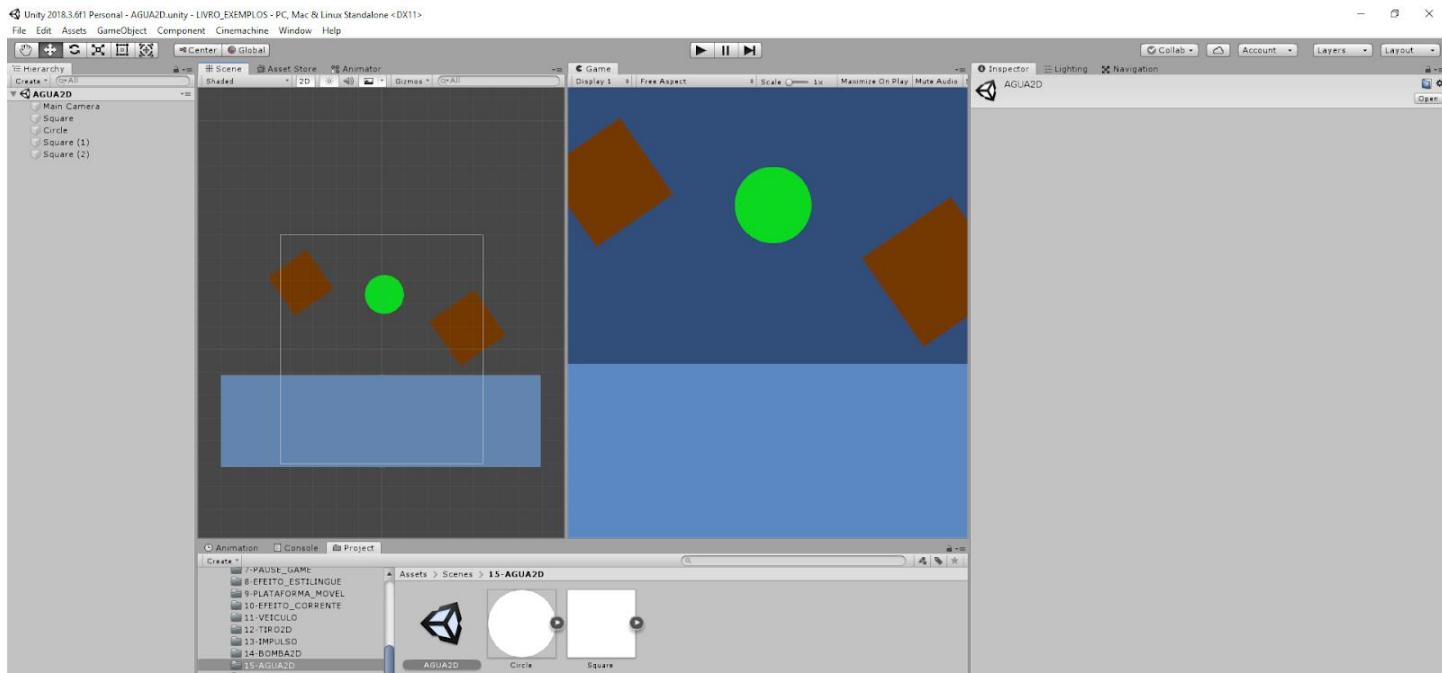
Bem simples não é mesmo depois de criado o objeto bomba já falamos que depois de uma tempo ele precisa ser destruído.

Prontinho agora é só reproduzir o efeito e ver tudo voar pelos ares.

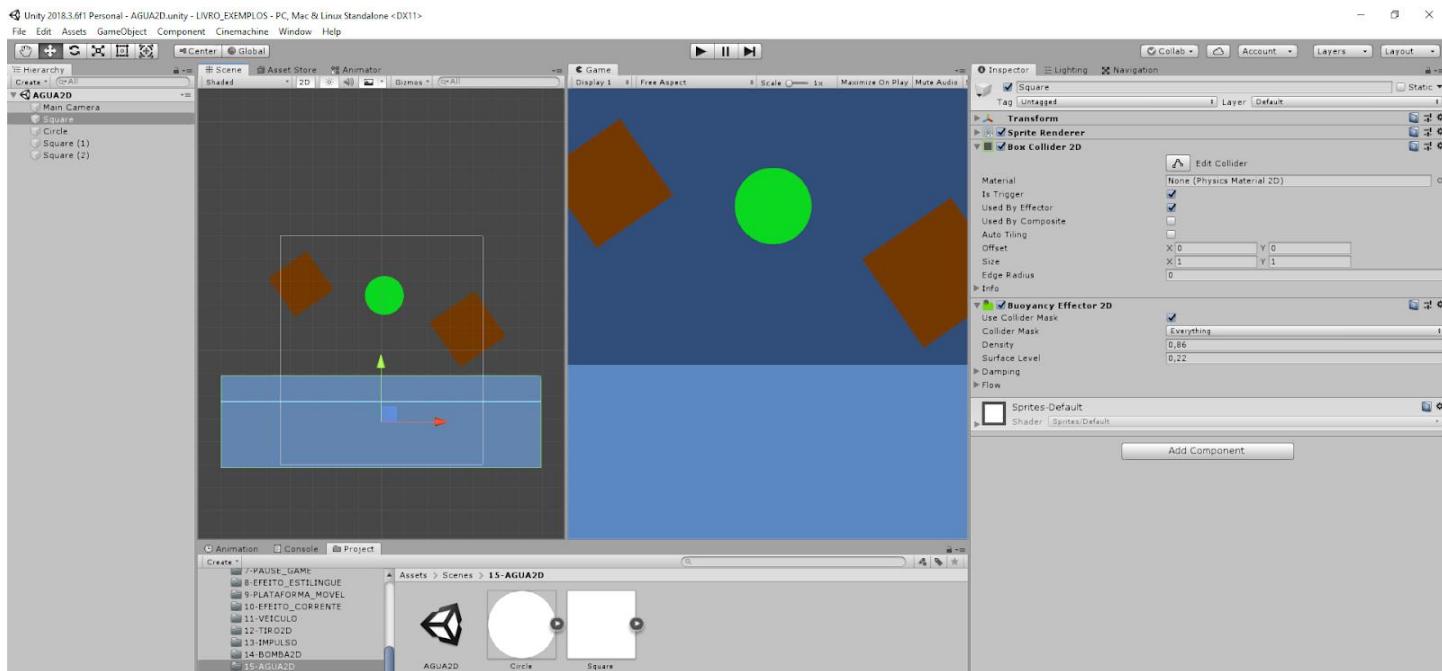
ÁGUA 2D

Mais um efeito muito interessante para usar em games é o efeito de agua, garantir que um objeto pode flutuar na agua e até mesmo ser arrastado por sua correnteza.

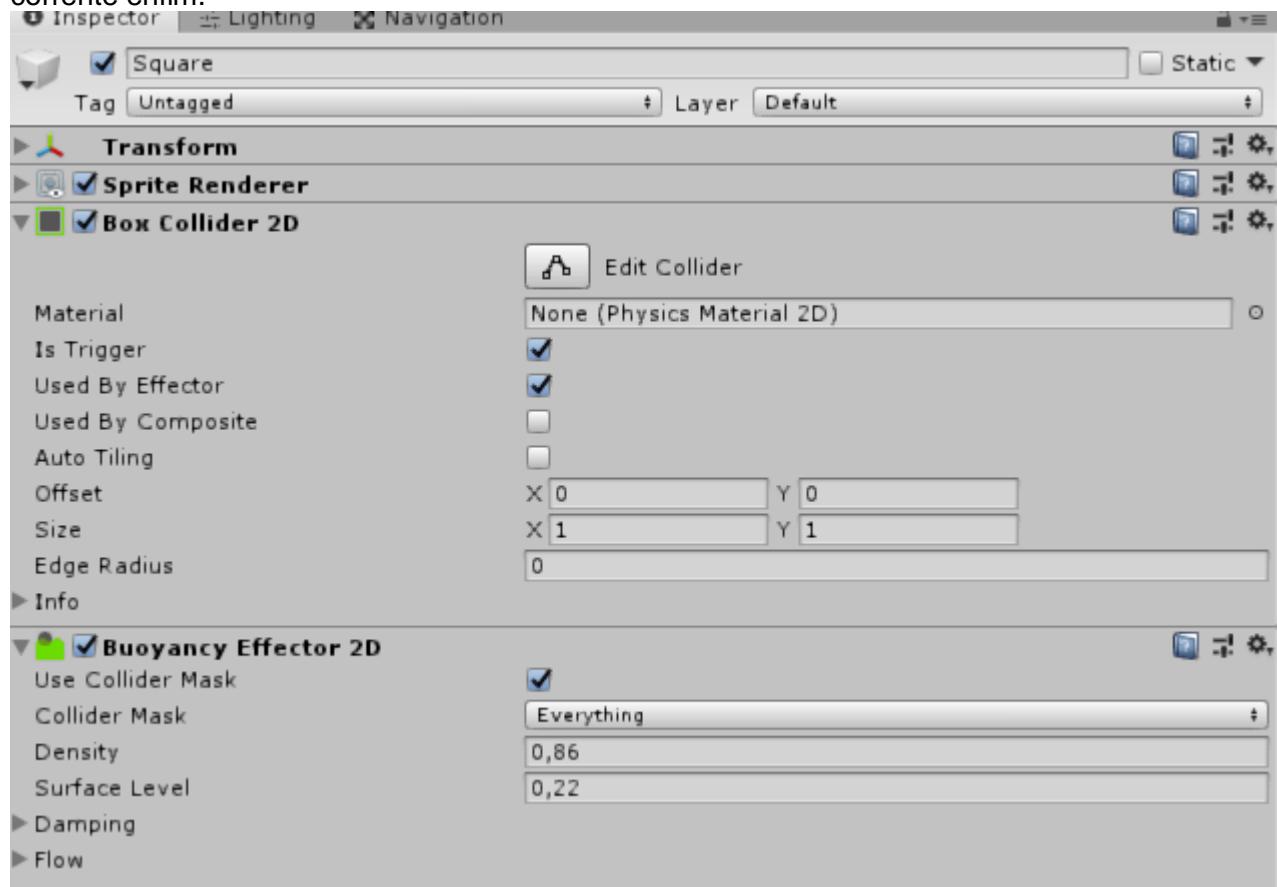
Criar um efeito desse tipo no Unity é muito simples, basta criar uma cena parecida com essa:



Veja que essa cena usa apenas círculos e quadrados, sendo o retângulo azul a representação da agua e os outros objetos representação de possíveis objetos físicos como caixas e bolas. O objeto que representa a agua tem a seguinte configuração:



Veja que ele tem um corpo colisor com a opção de Used By Effector habilitado pois usamos um Effector chamado Buoyancy Effector 2D que controla os efeitos da nossa agua como densidade força corrente enfim.



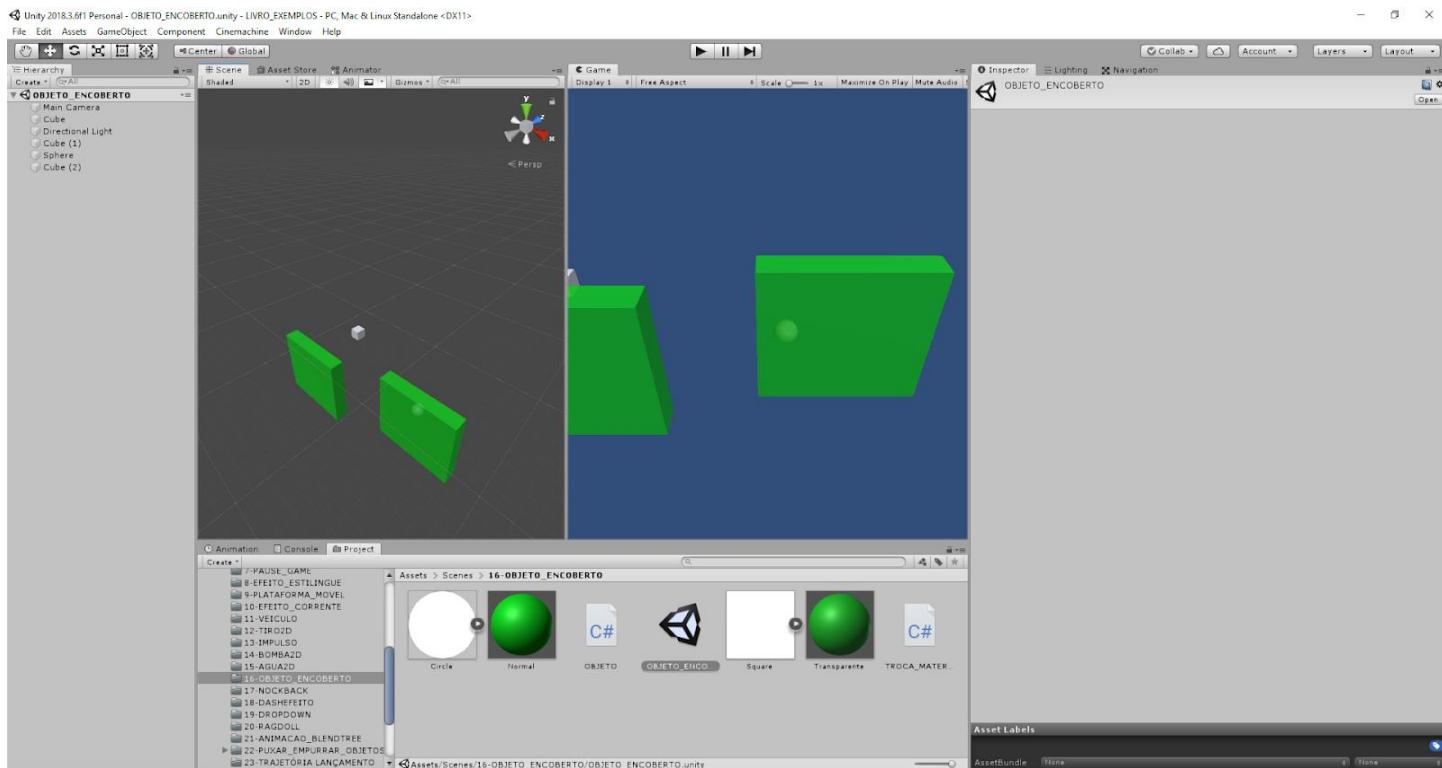
Os outros objetos são mais tranquilos cada um tem apenas um corpo colisor e um rigidbody2d. Pois dessa forma quando iniciarmos o exemplo os objetos vão cair sobre o objeto que representa a agua e vão flutuar. Simples assim.

OBJETOS ENCOBERTOS

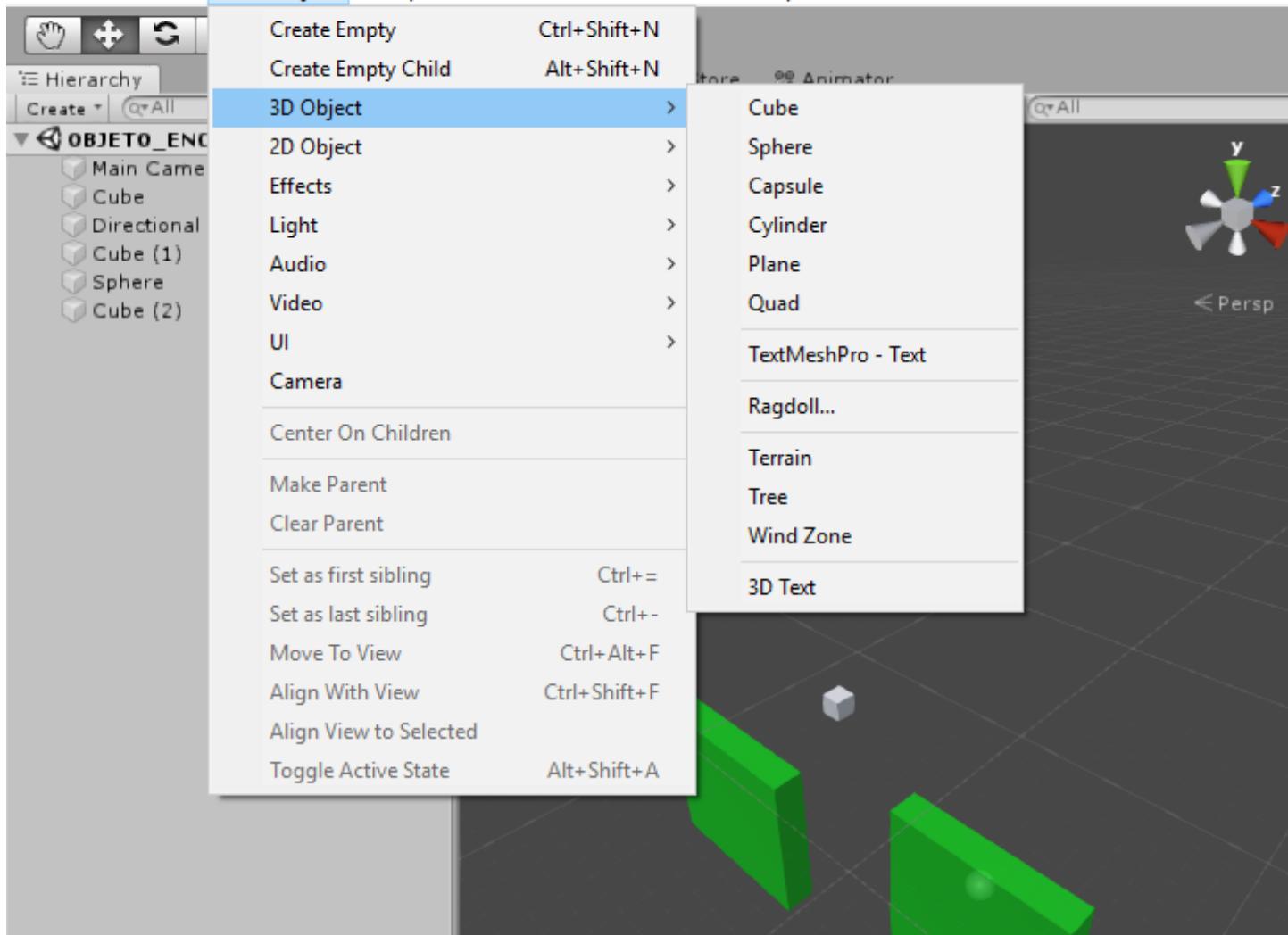
Mais uma mecânica muito legal e muito útil é a de objetos encobertos, afinal dependendo do jogo que você for criar é possível que em um momento uma parede ou coisa do tipo fique entre a câmera e o personagem prejudicando a visão do jogador.

Para resolver esse problema existem varias soluções mas sem duvida a mais comum é a que vou mostrar agora.

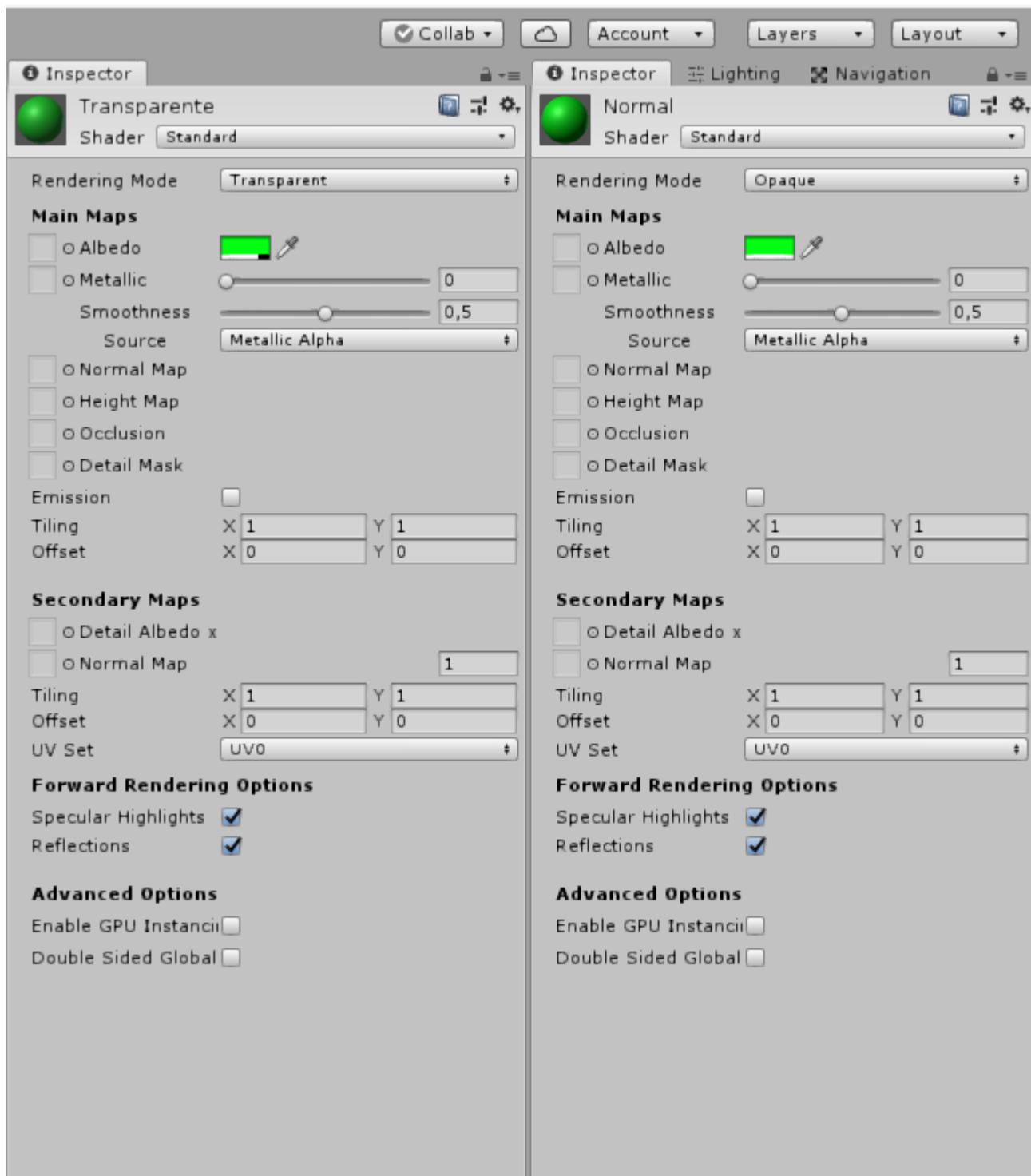
Então crie uma cena para esse novo exemplo e deixe a parecida com a imagem abaixo:



Nessa cena estamos usando objetos 3D que são cubos e esferas, para criar esses objetos em cena é muito simples basta ir em GameObject depois em 3D Object e na lista de objetos que vai aparecer escolher os cubos e esferas.



Feito isso precisamos criar dois materiais para aplicar nos objetos que vão encobrir o personagem.



Veja que os materiais tem a mesma cor à diferença é que um deles está definido como Opaque em Rendering Mode e o outro está definido como Transparent.

Dessa forma podemos deduzir que vamos ter uma troca de material opaco para transparente toda vez que o objeto prejudicar a visão do jogador.

Então tendo isso em mente vamos conhecer os códigos que fazem com que esse efeito seja possível. O primeiro arquivo de código é o **OBJETO** esse arquivo de código é adicionado na nossa câmera.

```

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

public class OBJETO : MonoBehaviour
{
    public Transform alvo;
    public RaycastHit hit = new RaycastHit();
    public TROCA_MATERIAL tMaterial;
    public LayerMask mask;

    // Update is called once per frame
    void Update()
    {
        if(Physics.Linecast(transform.position, alvo.position, out hit, mask))
        {
            Debug.DrawLine(transform.position, alvo.position);
            tMaterial = hit.transform.GetComponent<TROCA_MATERIAL>();
            tMaterial.renderer.material = tMaterial.mat[1];
        }
        else
        {
            if(tMaterial != null)
                tMaterial.RetornaTransp();
        }
    }
}

```

Veja que no código da câmera iniciamos nosso trabalho criando algumas variáveis como por exemplo o alvo que é uma variável do tipo Transform e serve para definir o nosso alvo o objeto que estamos olhando.

Depois temos um RaycastHit para verificar quando nosso raio de visão colide com um obstáculo. Temos uma variável do tipo **TROCA_MATERIAL** que nada mais é que o arquivo de código que vai ser adicionado nas paredes que podem encobrir a visão do player. E por ultimo temos um LayerMask para nos auxiliar na colisão com os obstáculos.

```

public Transform alvo;
public RaycastHit hit = new RaycastHit();
public TROCA_MATERIAL tMaterial;
public LayerMask mask;

```

Dentro do método Update usamos um Linecast para verificar se colidimos com alguma parede e se isso tiver acontecido vamos desenhar uma linha branca da câmera até o alvo e já vamos modificar o material da parede para a opção de material mat[1] que representa o material com transparência, caso contrario chamamos o método RetornaTransp que faz a parede voltar ao seu material de origem.

```
void Update()
```

```

    {
        if(Physics.Linecast(transform.position, alvo.position, out
hit,mask))
        {
            Debug.DrawLine(transform.position,alvo.position);
            tMaterial = hit.transform.GetComponent<TROCA_MATERIAL>();
            tMaterial.renderer.material = tMaterial.mat[1];
        }
        else
        {
            if(tMaterial != null)
                tMaterial.RetornaTransp();
        }
    }
}

```

Agora para deixar isso mais claro vamos ver o código TROCA_MATERIAL.

```

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

public class TROCA_MATERIAL : MonoBehaviour
{
    public Material[] mat;
    public Renderer renderer;
    // Start is called before the first frame update
    void Start()
    {
        RetornaTransp();
    }

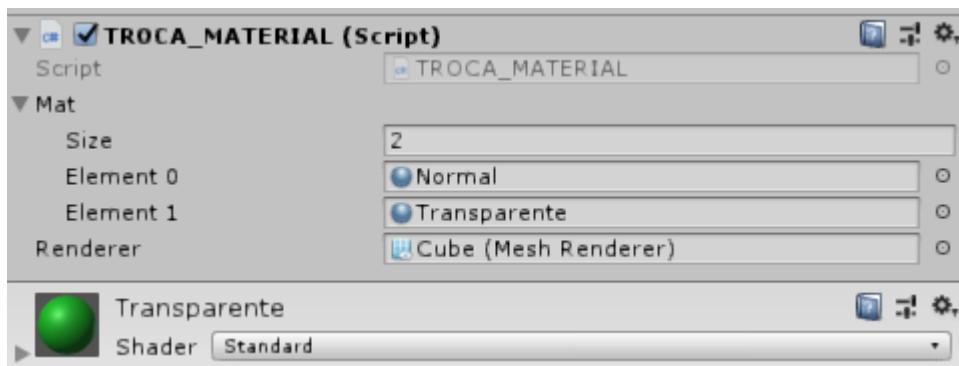
    public void RetornaTransp()
    {
        renderer.material = mat[0];
    }
}

```

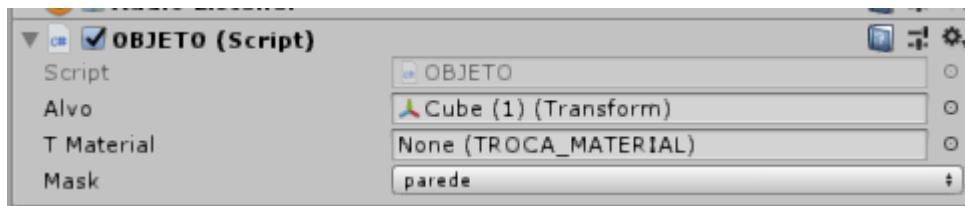
Veja que o código aqui é muito simples apenas criamos duas variáveis uma do tipo Material que é um array de materiais e outra do tipo Renderer que manipula o material de um objeto, depois disso criamos um método que retorna o objeto para seu material padrão sem transparência e aproveitamos para chamar esse mesmo método dentro de Start para que as configurações iniciem dessa forma.

Esse é o código que deve ser adicionado nas paredes que tem a possibilidade de obstruir a visão do jogador.

Veja que depois de adicionar o código nos objetos parede você precisa mostrar os materiais que vão ser usados no array e logo depois o Renderer desse objeto.



Já o código da câmera precisamos passar o objeto que será nosso alvo no caso é o cubo da cena e em Mask informamos o layer que toda parede tem.:



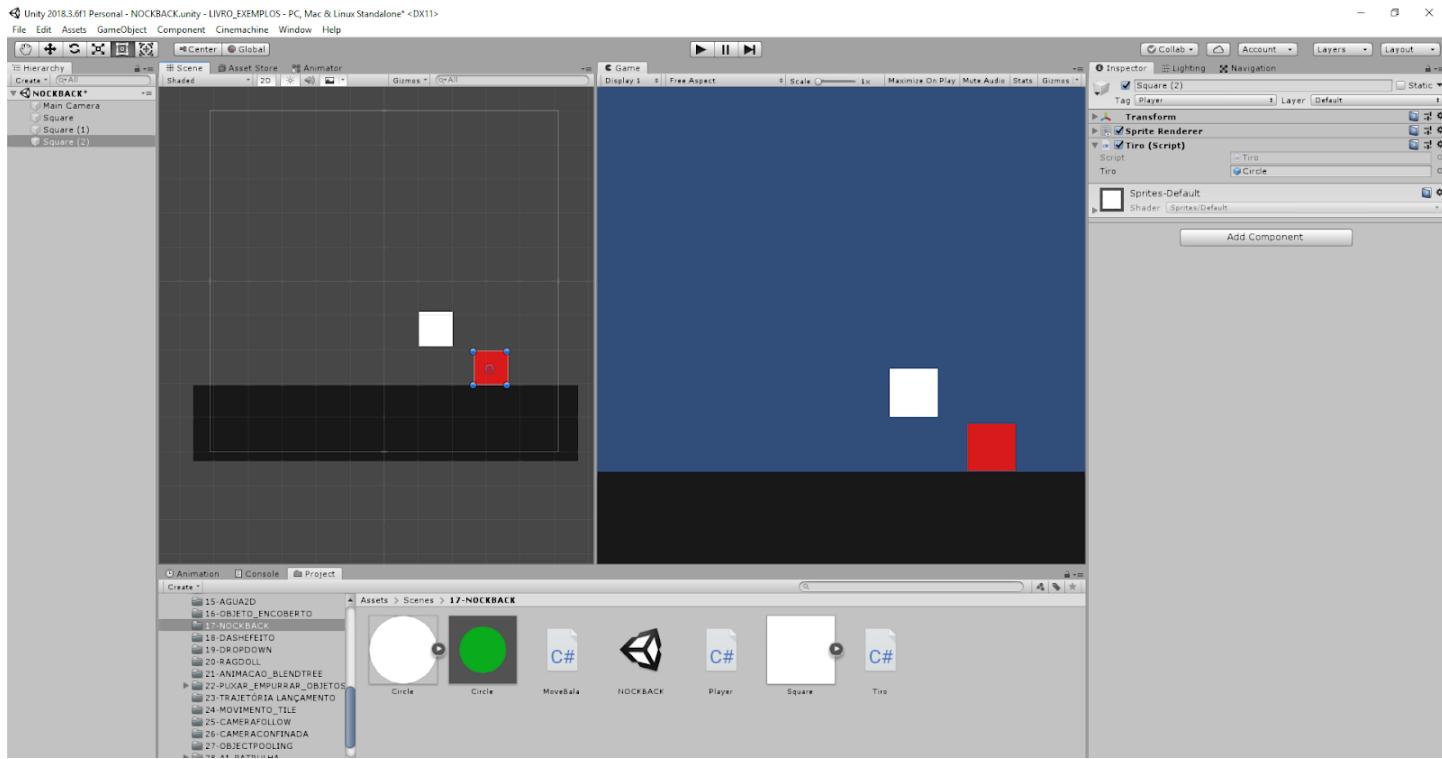
Prontinho com isso o exemplo já vai funcionar.

KNOCK BACK

Mais um efeito que não pode faltar no seu jogo é o Knock Back um efeito que faz com que objetos que sofrem dano sejam impulsionados na direção contrária do ataque.

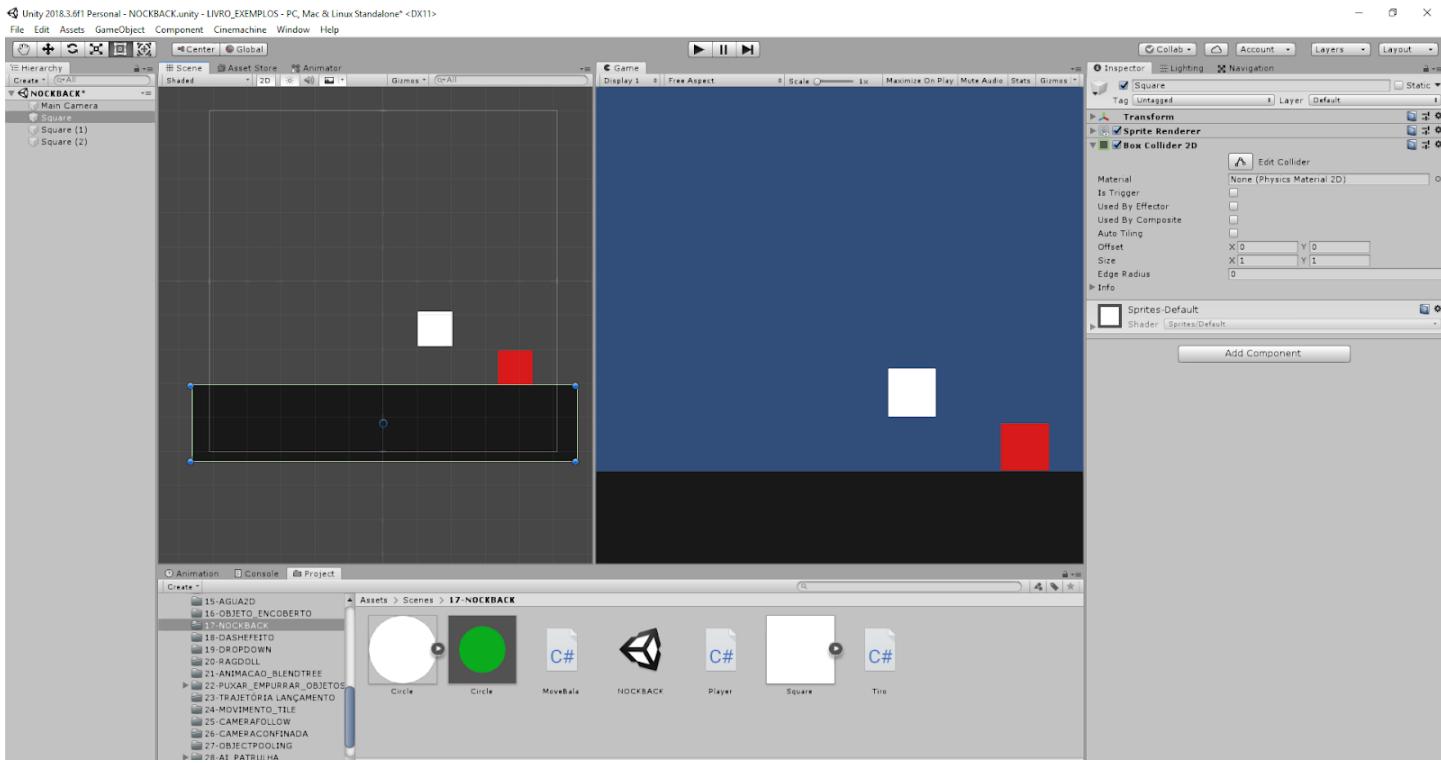
Dessa forma aumentamos o realismo do nosso jogo.

Para reproduzir esse exemplo crie uma cena semelhante a essa:

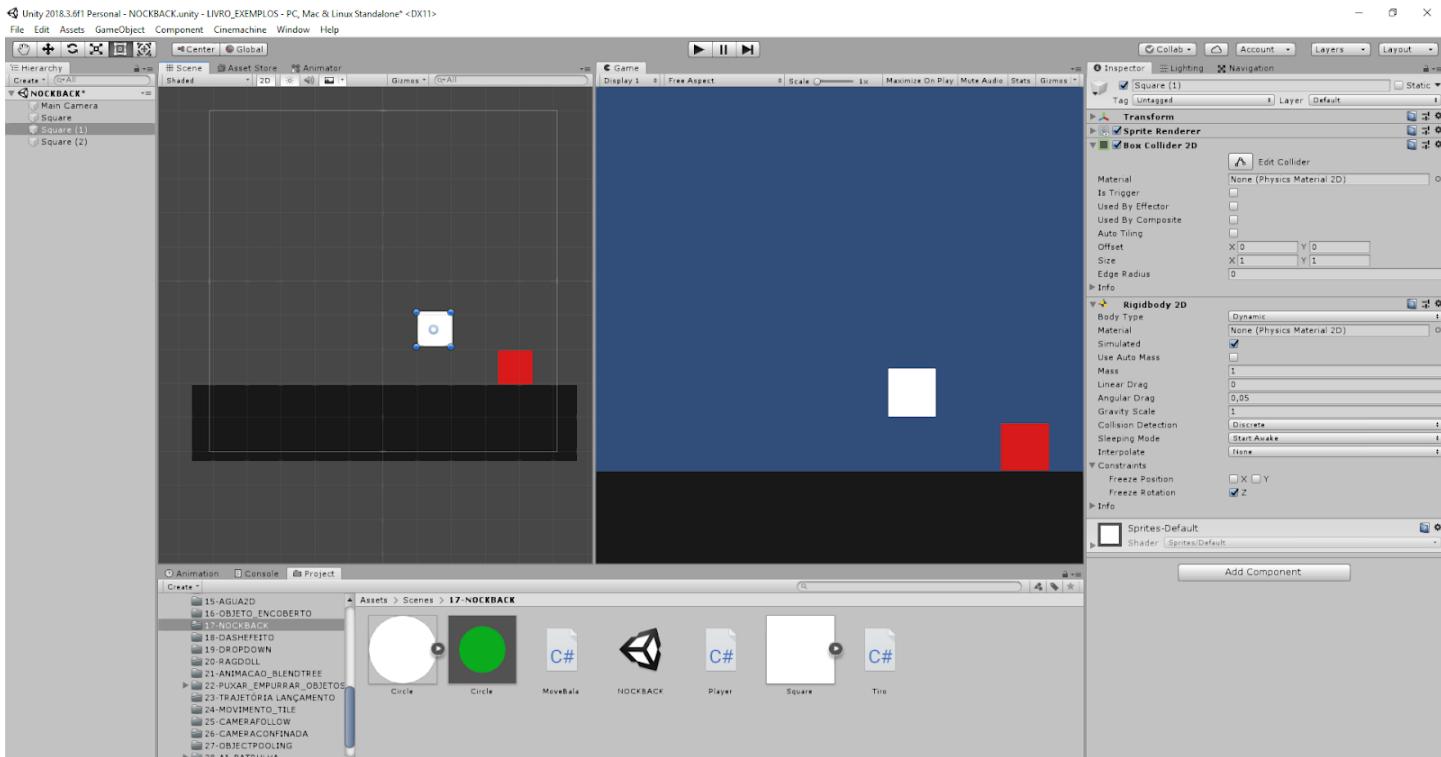


Veja que nessa cena temos apenas sprites quadradas sendo uma retangular para representar o chão e outras duas quadradas para representar os personagens.

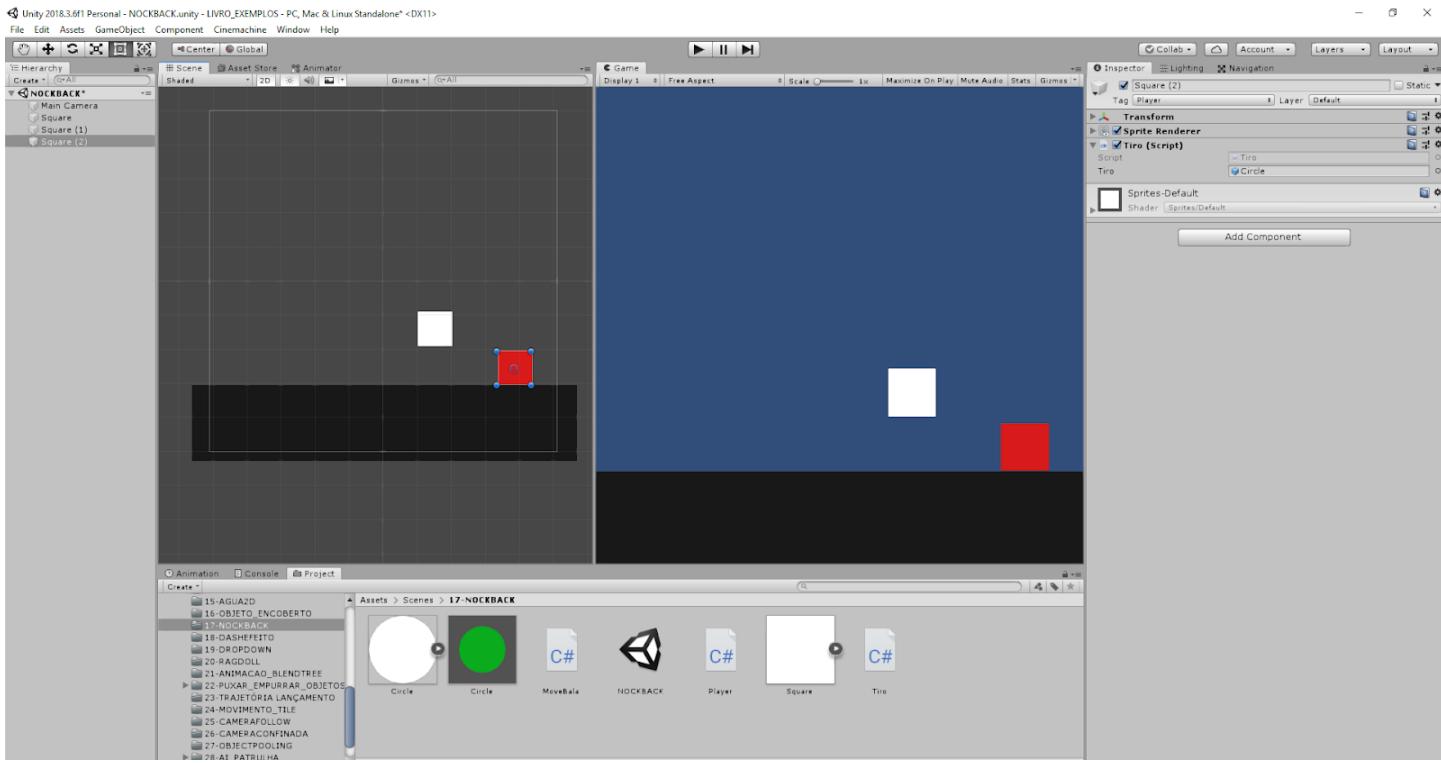
O chão tem apenas um Corpo colisor como você pode ver na imagem abaixo:



Já o quadrado branco tem um corpo colisor e um rigidbody2D.



O quadrado vermelho tem apenas um arquivo de código que é o que vamos usar para reproduzir esse efeito.



Enfim esses são os elementos que formam a nossa cena, sendo assim só nos falta analisar o código que segue logo abaixo:

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

public class Tiro : MonoBehaviour
{
    public GameObject tiro;
    // Start is called before the first frame update
    void Start()
    {
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Instantiate(tiro, transform.position, Quaternion.identity);
        }
    }
}
```

Esse código está inserido dentro do objeto quadrado vermelho e sua única tarefa é criar as sprites de tiro.

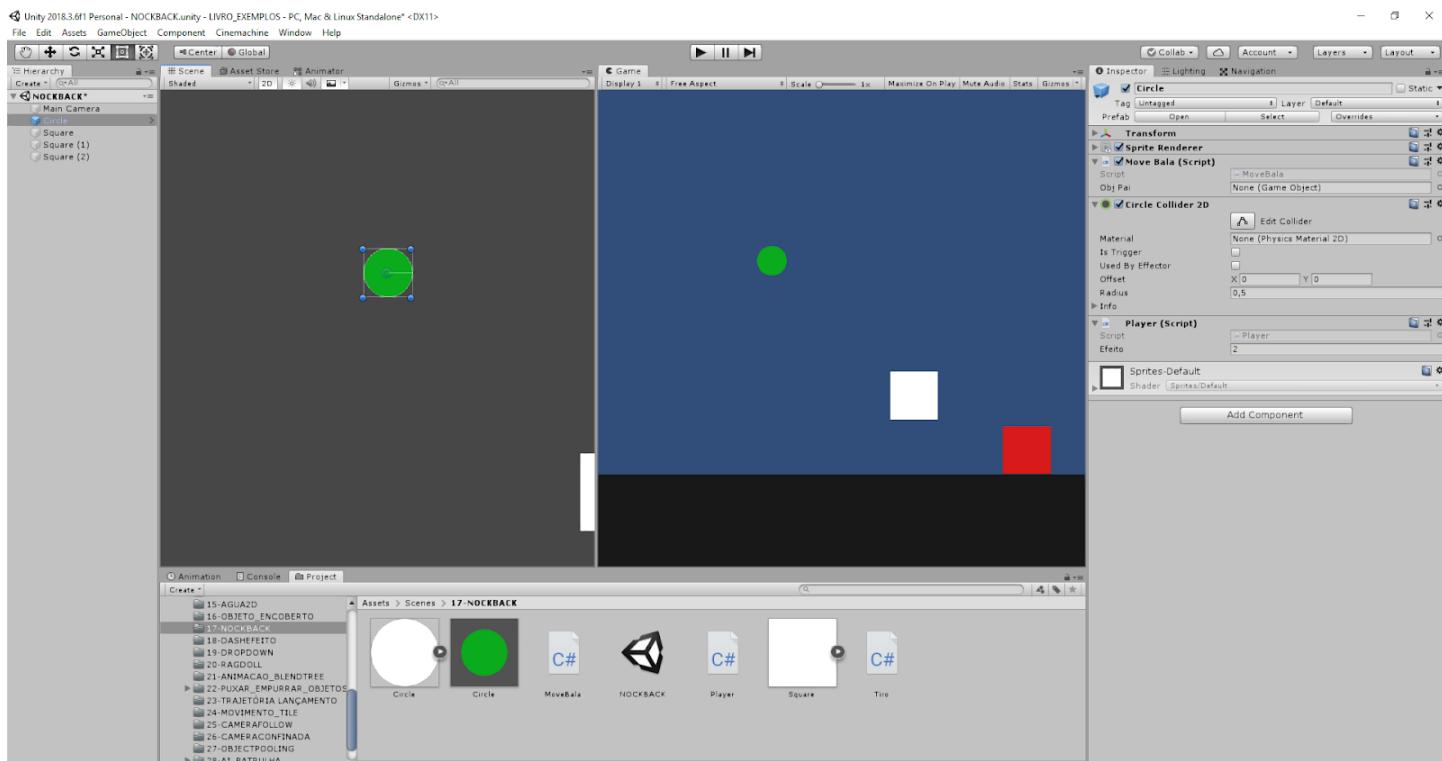
Isso é feito usando uma variável do tipo GameObject.

```
public GameObject tiro;
```

E dentro do método Update verificamos quando foi apertado a tecla espaço pois esse é o gatilho para criar copias da bala dentro da cena.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Instantiate(tiro, transform.position, Quaternion.identity);
    }
}
```

Agora você deve estar se perguntando quem é o objeto que está sendo criado, é simples o objeto é esse prefab aqui:



Veja que esse objeto tem um corpo colisor e dois arquivos de código que fazem com que ele funcione. O primeiro arquivo de código se chama **MoveBala** e tem a função de mover a bala de um ponto até outro.

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

public class MoveBala : MonoBehaviour
{
```

```

public GameObject objPai;
// Start is called before the first frame update
void Start()
{
    objPai = GameObject.FindGameObjectWithTag("Player");
}

// Update is called once per frame
void Update()
{
    transform.Translate(objPai.transform.localScale.x *
Time.deltaTime, 0, 0);
}

void OnCollisionEnter2D(Collision2D collision)
{
    Destroy(gameObject);
}

}

```

Para que isso funcione corretamente é necessário criar uma variável do tipo GameObject que vai buscar o nosso atirador o player da cena.

```
public GameObject objPai;
```

Essa busca acontece dentro do método Start.

```

{
    objPai = GameObject.FindGameObjectWithTag("Player");
}

```

Com isso conseguimos pegar a direção do personagem que vai atirar e dessa forma conseguimos fazer com a bala se movimente sempre para frente do personagem e nunca para as costas dele. Veja que dentro de Update movimentamos a bala levando em consideração a escala local no eixo X do personagem isso faz com que a bala sempre vá na direção correta.

```

void Update()
{
    transform.Translate(objPai.transform.localScale.x *
Time.deltaTime, 0, 0);
}

```

Por último verificamos se existe uma colisão sobre esse objeto existindo destruímos o mesmo.

```

void OnCollisionEnter2D(Collision2D collision)
{
    Destroy(gameObject);
}

```

Esse foi o primeiro código, o segundo é esse aqui:

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

public class Player : MonoBehaviour
{
    public float efeito;

    // Start is called before the first frame update

    void OnCollisionEnter2D(Collision2D collision)
    {
        Rigidbody2D rb =
collision.collider.GetComponent<Rigidbody2D>();

        if(rb != null)
        {
            Vector3 direcao = collision.transform.position -
transform.position;
            direcao.y = 0;

            rb.AddForce(direcao.normalized * efeito,
ForceMode2D.Impulse);
        }
    }

}
```

Nesse código usamos uma variável do tipo float que serve para informar a força do impacto sobre o objeto atingido.

```
public float efeito;
```

Depois em OnCollisionEnter2D ajustamos a direção do efeito e aplicamos uma força ao objeto que recebe a ação para que o mesmo sempre seja arremessado levemente para a direção oposta do ataque.

```
void OnCollisionEnter2D(Collision2D collision)
{
    Rigidbody2D rb =
collision.collider.GetComponent<Rigidbody2D>();

    if(rb != null)
    {
```

```

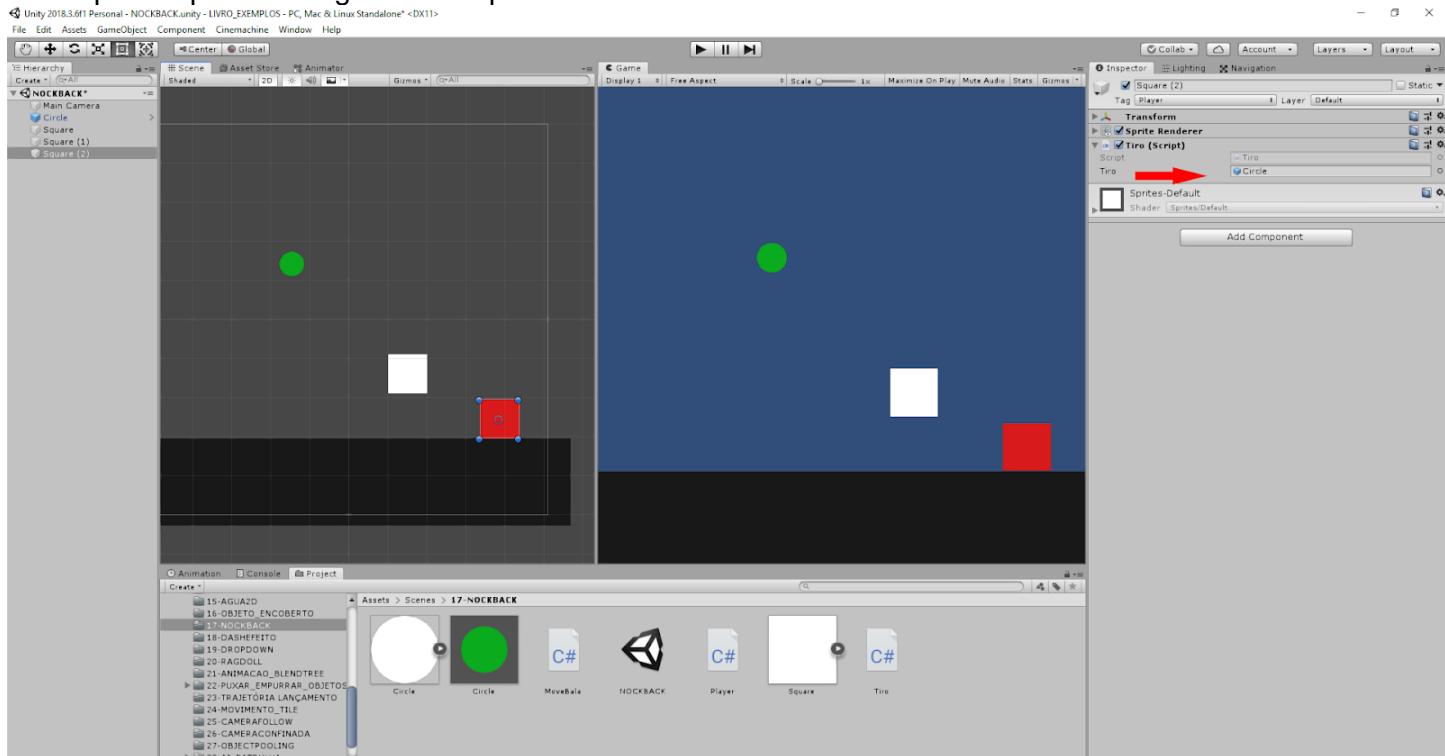
        Vector3 direcao = collision.transform.position -
        transform.position;
        direcao.y = 0;

        rb.AddForce(direcao.normalized * efeito,
        ForceMode2D.Impulse);
    }
}

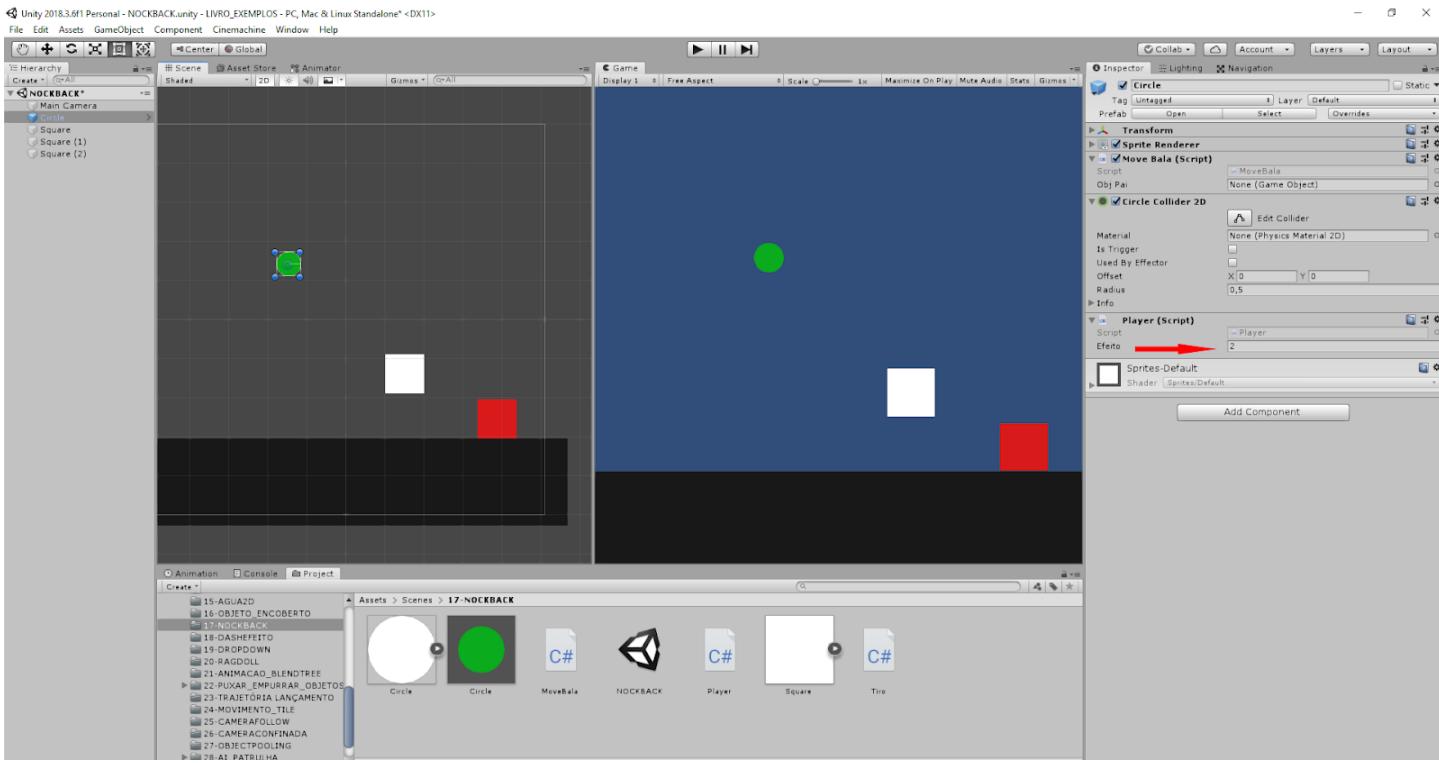
```

É isso com esses códigos e configurações o exemplo já vai funcionar claro que antes de qualquer coisa precisamos fazer mais alguns ajustes dentro da janela Inspector.

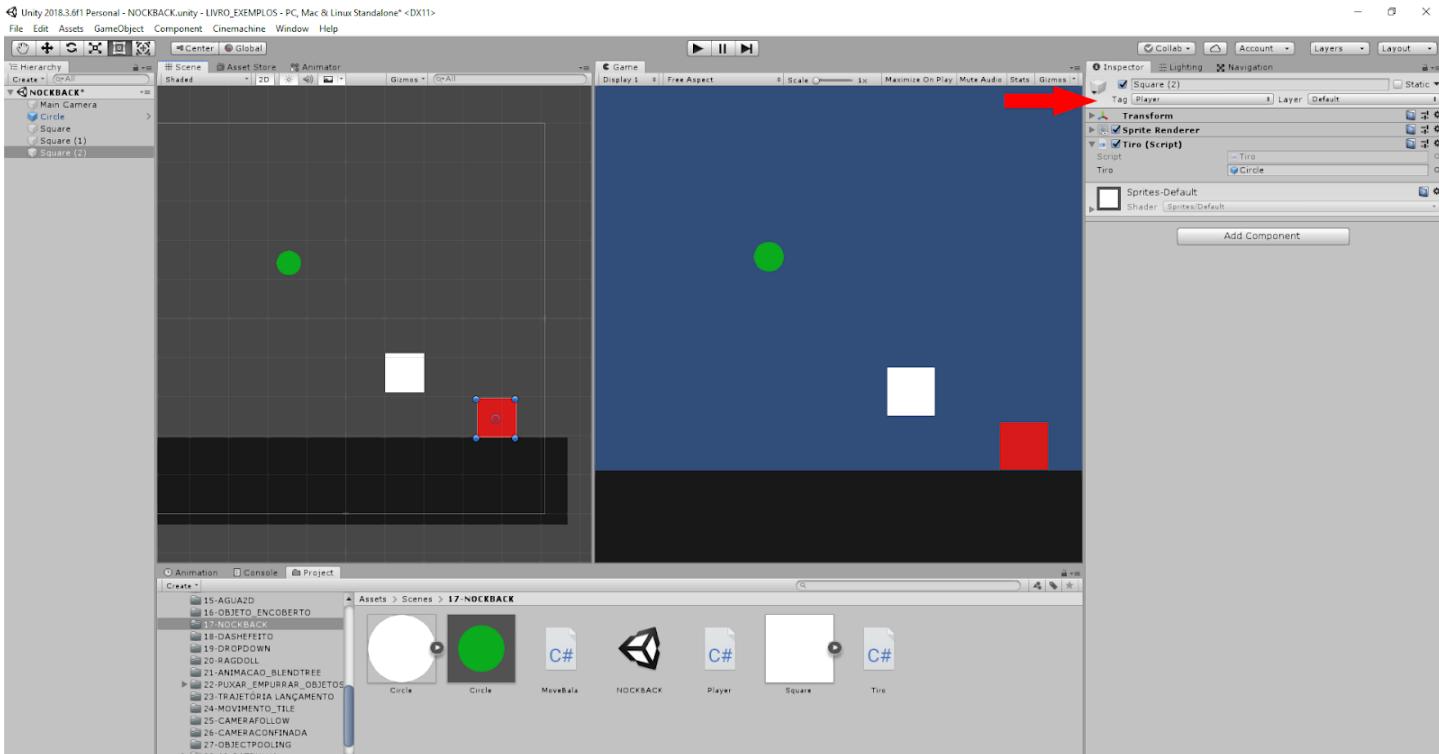
Como passar para o código de tiro o prefab do tiro.



E passar para o código Player que esta na bala o valor do efeito.



E para fechar só adicionamos uma tag ao quadrado vermelho identificando o mesmo como sendo o nosso Player.



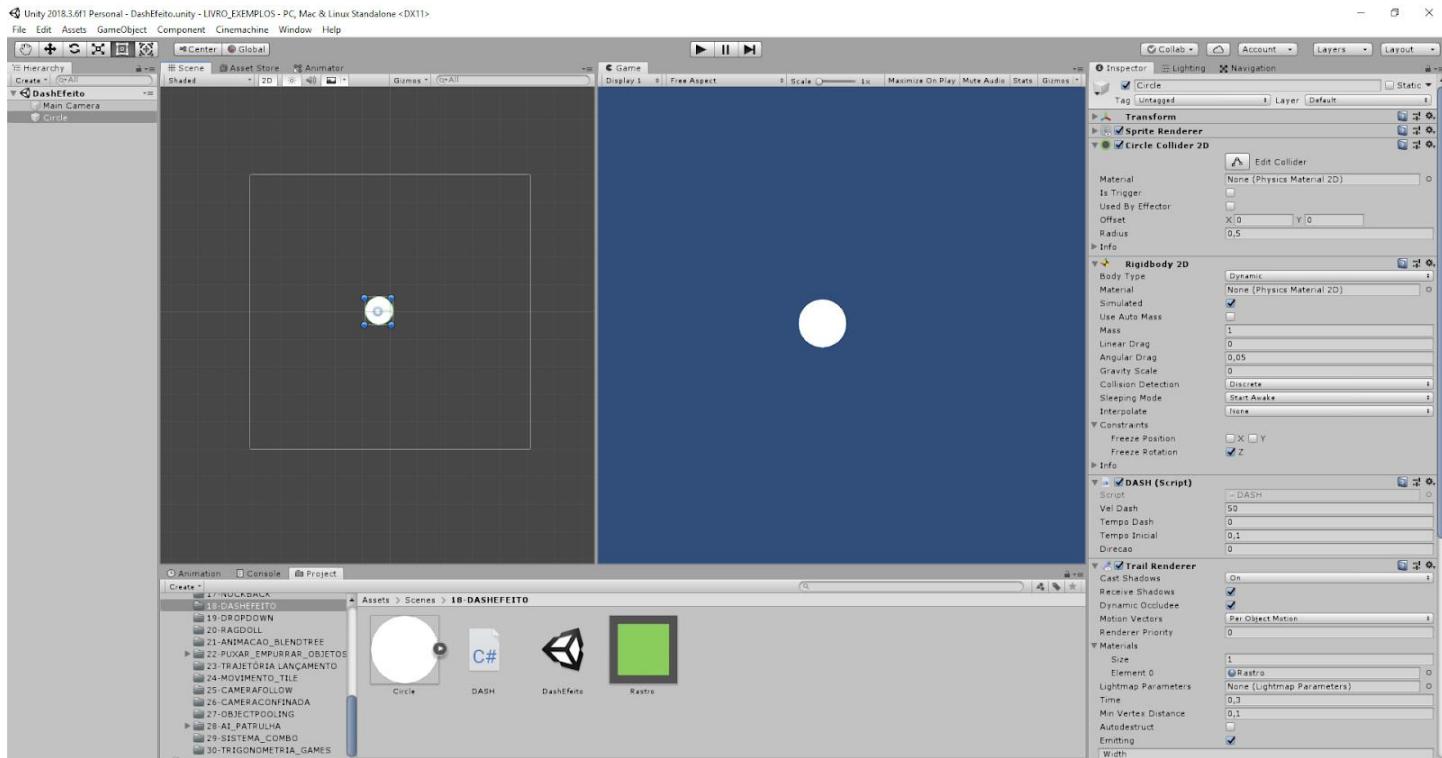
Agora é só executar o exemplo e ver tudo funcionando.

DASH EFEITO

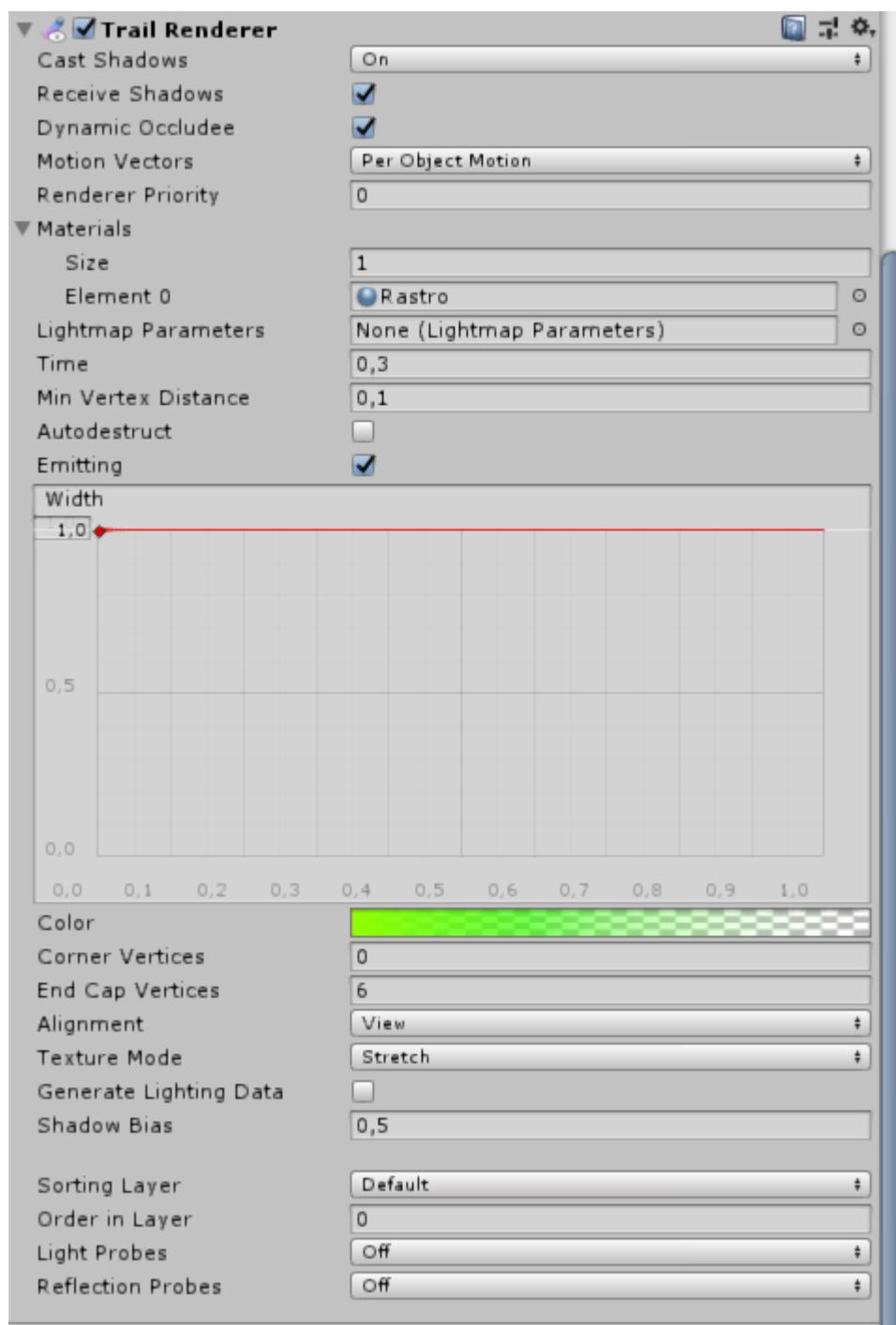
Outro efeito muito legal usado em games é o efeito Dash que nada mais é que um deslocamento rápido do personagem.

Esse tipo de mecânica é usada principalmente em jogos onde o personagem se move muito rápido em determinados momentos deixando até mesmo um rastro do seu movimento.

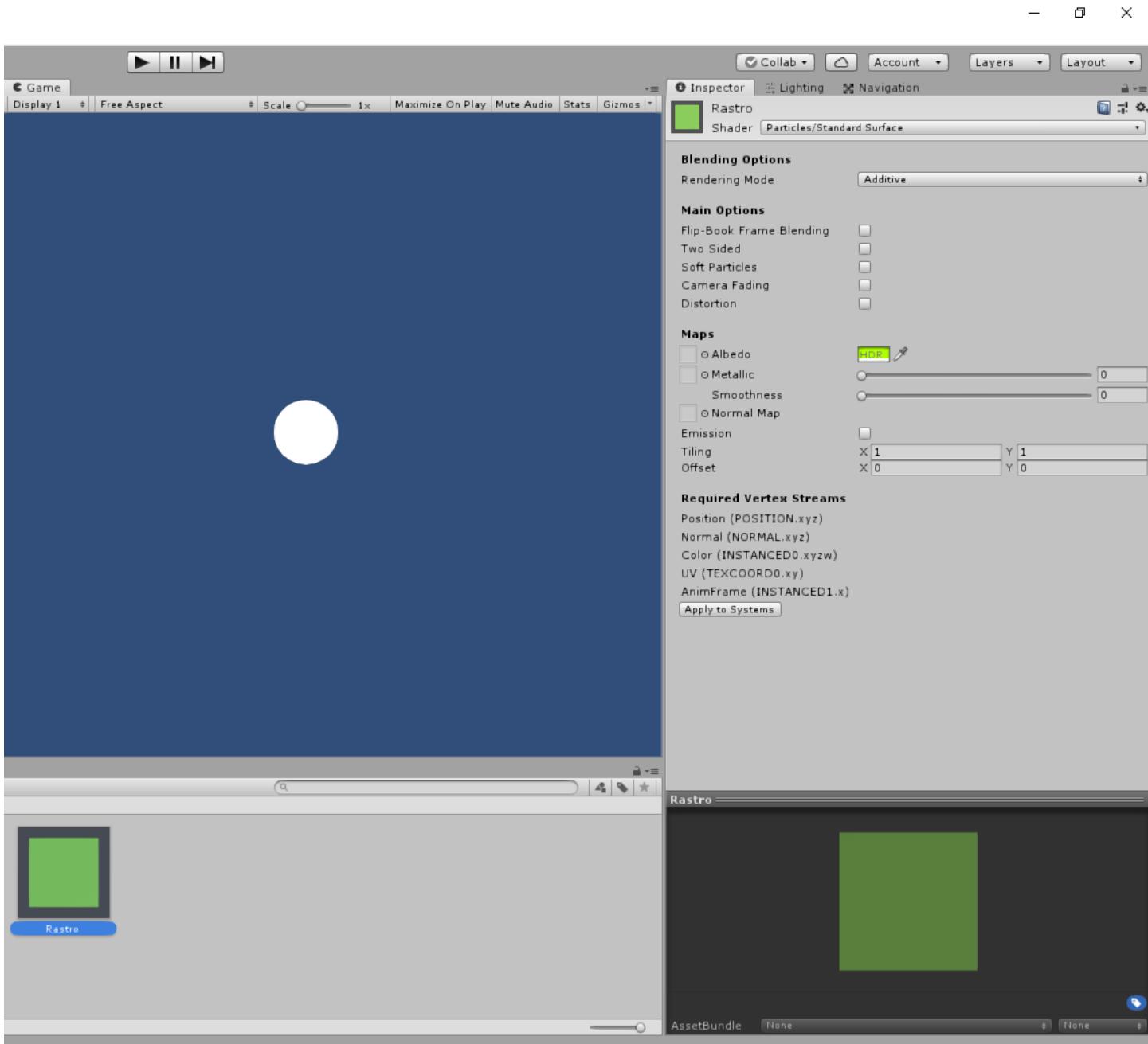
Para criar esse efeito vamos criar uma cena muito simples, veja:



Veja que nessa cena temos apenas uma sprite redonda que contém um corpo colisor, um rigidbody2D um arquivo de código com o nome de DASH e um TrailRenderer para simbolizar o rastro do objeto. As configurações do TrailRenderer são essas aqui:



Veja que quase nada foi alterado nesse componente apenas adicionamos um material com o nome de Rastro.



Ajustamos o valor de Time para que o rastro não fique muito tempo visível em cena, mexemos na largura do efeito usando o gráfico acima da configuração Color que por sua vez também foi alterada para ter uma cor verde.

E por último ajustamos o valor de End Cap Vertices para deixar o rastro mais arredondado.

Com esses ajustes em cena devidamente feitos podemos focar no código que é adicionado a sprite esférica:

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

public class DASH : MonoBehaviour
{
```

```

private Rigidbody2D rb;
public float velDash;
public float tempoDash;
public float tempoInicial;
public int direcao;

// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    tempoDash = tempoInicial;
}

// Update is called once per frame
void Update()
{

    if(direcao == 0)
    {
        if(Input.GetKeyDown(KeyCode.LeftArrow) )
        {
            direcao = 1;
        }
        else if(Input.GetKeyDown(KeyCode.RightArrow) )
        {
            direcao = 2;
        }
    }
    else
    {
        if(tempoDash <= 0)
        {
            direcao = 0;
            tempoDash = tempoInicial;
            rb.velocity = Vector2.zero;
        }
        else
        {
            tempoDash -= Time.deltaTime;

            if(direcao == 1)
            {
                rb.velocity = Vector2.left * velDash;
            }
            else if(direcao == 2)
            {
                rb.velocity = Vector2.right * velDash;
            }
        }
    }
}

```

Veja que para esse efeito começamos criando as variáveis que serão necessárias aqui:

```
private Rigidbody2D rb;  
public float velDash;  
public float tempoDash;  
public float tempoInicial;  
public int direcao;
```

Veja que temos uma variável do tipo `rigidbody2D` que nos permite manipular a sprite que tem esse componente.

Depois temos variaveis do tipo float que servem para ajuste de velocidade do efeito, tempo e tempo inicial.

E por último uma variável do tipo int que nos auxilia na direção do efeito.

Com isso tudo definido passamos via script a informação de rigidbody para a variável rb e logo depois passamos para a variável tempoDash o valor de tempoInicial.

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    tempoDash = tempoInicial;
}
```

Mais adiante dentro do método Update criamos uma estrutura condicional que verifica se a direção é igual a zero se for posso continuar o processo de verificações passando para aproxima que verifica se apertamos a tecla seta para esquerda.

Se realmente apertamos essa tecla passaremos para a variável direção o valor 1 caso contrario se apertamos a seta para direita passamos o valor 2 para a variável

Depois disso temos os cálculos de tempo para ajustar o efeito e fazer com que o mesmo volte para o seu estado atual que é o parado.

```
void Update()
{
    if(direcao == 0)
    {
        if(Input.GetKeyDown(KeyCode.LeftArrow))
        {
            direcao = 1;
        }
        else if(Input.GetKeyDown(KeyCode.RightArrow))
        {
            direcao = 2;
        }
    }
}
```

```

    }
    else
    {
        if(tempoDash <= 0)
        {
            direcao = 0;
            tempoDash = tempoInicial;
            rb.velocity = Vector2.zero;
        }
        else
        {
            tempoDash -= Time.deltaTime;

            if(direcao == 1)
            {
                rb.velocity = Vector2.left * velDash;
            }
            else if(direcao == 2)
            {
                rb.velocity = Vector2.right * velDash;
            }
        }
    }
}
}

```

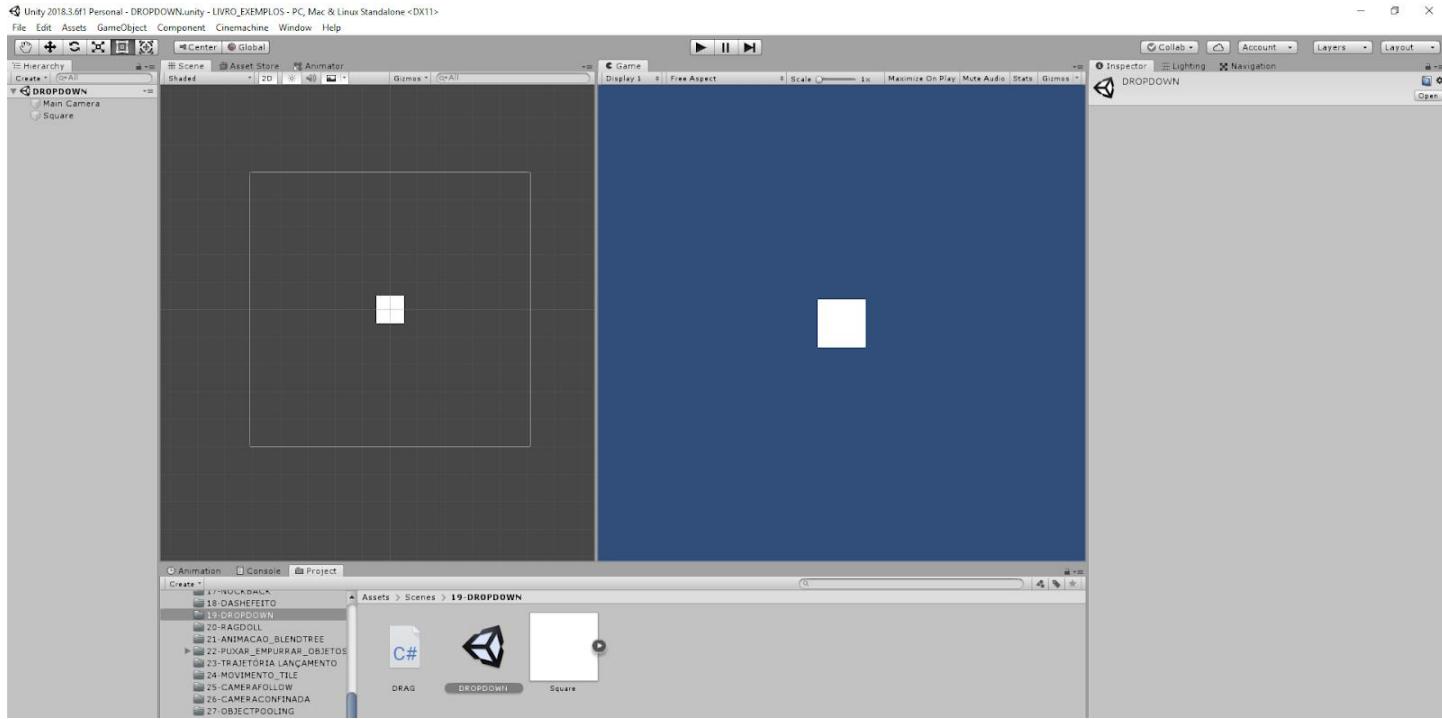
Para fechar o exemplo agora só nos falta fazer os últimos ajustes dentro do Inspector adicionando valores em Vel Dash e Tempo inicial.



Feito isso é só executar o exemplo para ver tudo funcionando.

DRAG AND DROP

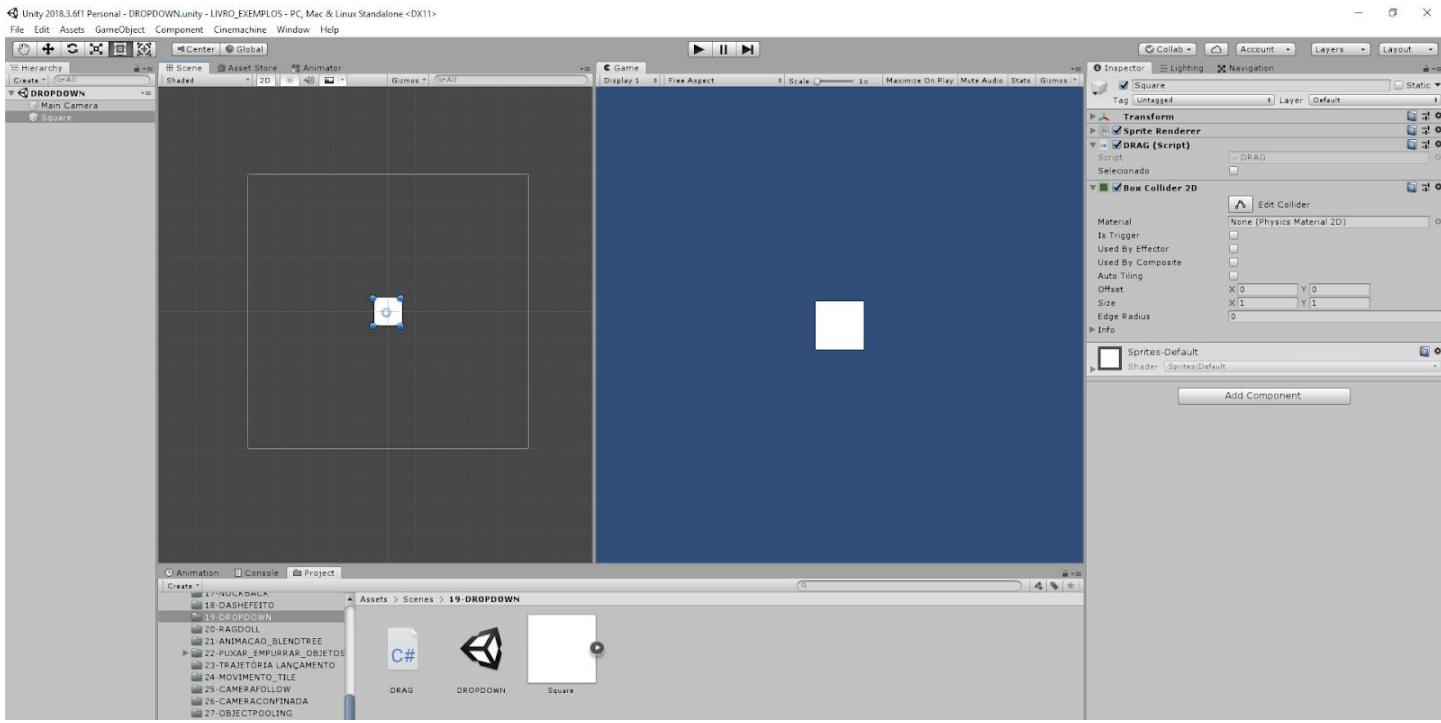
Outro efeito muito útil dentro dos games é o de pegar arrastar e soltar objetos em cena. Para criar um efeito desse tipo é muito simples a primeira coisa que precisamos fazer é criar uma cena conforme a imagem abaixo:



Veja que a cena é mais do que simples a única coisa que temos aqui é uma sprite quadrada que será arrastada pela cena com o auxílio do mouse.

Essa sprite tem como podemos ver na imagem abaixo tem apenas um corpo colisor e um arquivo de código vinculado a ela.

O arquivo de código tem o nome de **DRAG**.



Agora vamos analisar o código:

```

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

public class DRAG : MonoBehaviour
{

    public bool selecionado;

    // Update is called once per frame
    void Update()
    {
        if(selecionado)
        {
            Vector2 pos =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
            transform.position = new Vector2(pos.x,pos.y);

            if(Input.GetMouseButtonUp(0))
            {
                selecionado = false;
            }
        }
    }

    void OnMouseOver()
    {

```

```
        if (Input.GetMouseButtonUp (0))  
        {  
            seleccionado = true;  
        }  
    }  
}
```

Veja que nesse código temos uma variável do tipo booleana que vai nos auxiliar na seleção do objeto que será arrastado e solto em diferentes locais dentro da cena.

```
public bool selecionado;
```

No método `Update` verificamos se a variável selecionada é verdadeira se for passamos para a variável `pos` o valor de `ScreenToWorldPoint` que passa o valor de posição do espaço de tela em espaço de mundo.

Na sequência usando a variável pos para definir a posição do nosso objeto e mais adiante verificamos de deixamos de segurar o botão esquerdo do mouse se isso realmente aconteceu a variável **selecionada** fica falsa.

```
void Update()
{
    if(seleccionado)
    {
        Vector2 pos =
Camera.main.ScreenToWorldPoint(Input.mousePosition);
        transform.position = new Vector2(pos.x, pos.y);

        if(Input.GetMouseButtonUp(0))
        {
            seleccionado = false;
        }
    }
}
```

Agora para fechar esse código temos o método `OnMouseOver` que executa alguma ação se o mouse estiver sobre o objeto em questão.

No caso se o mouse estiver sobre a sprite esférica e apertarmos a botão esquerdo do mouse a variável selecionada fica verdadeira.

```
void OnMouseOver()
{
    if (Input.GetMouseButtonDown (0))
    {
        seleccionado = true;
    }
}
```

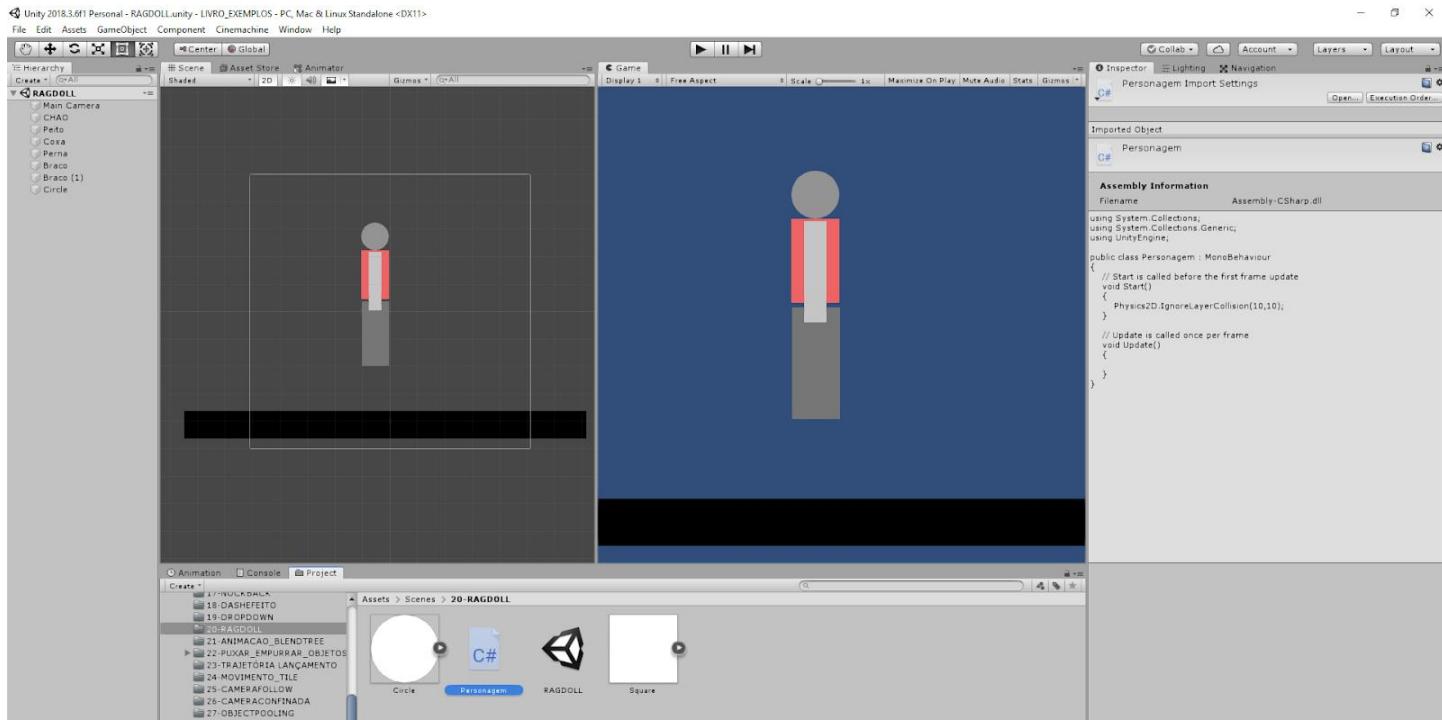
Com isso conseguimos pegar objetos arrasta-los e solta-los em qualquer parte da cena.

RAGDOLL

Agora vamos reproduzir mais uma mecânica muito legal que é a de ragdoll, essa mecânica consiste em simular um corpo realista no momento de sua queda ou arremesso.

Para fazer isso é necessário criar juntas que se comportem como as juntas do corpo humano respeitando rotações de cada articulação.

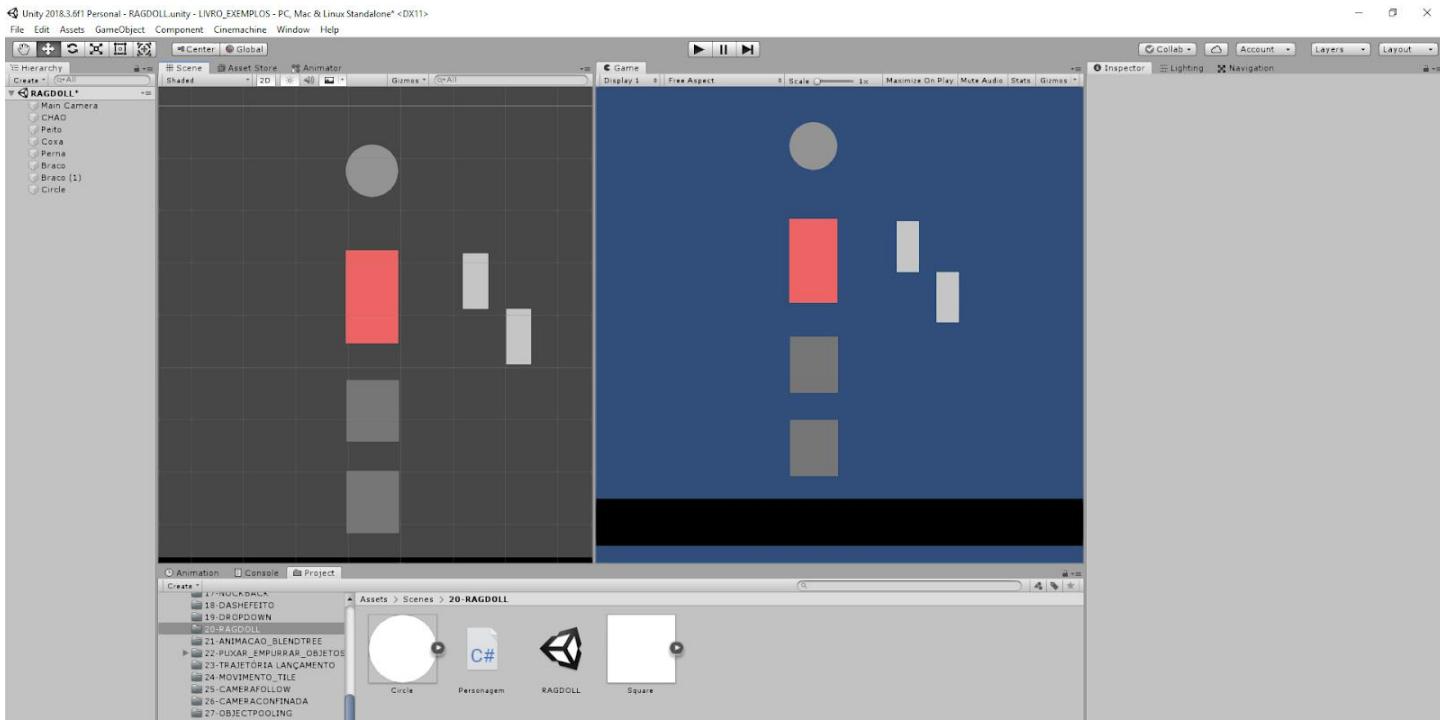
Para criar esse efeito vamos precisar de uma cena parecida com a da imagem abaixo:



Veja que na imagem acima criamos uma estrutura humanoide vista de perfil.

Temos a cabeça, peito, braços e pernas.

Essa estrutura é formada por quadrados e círculo veja:

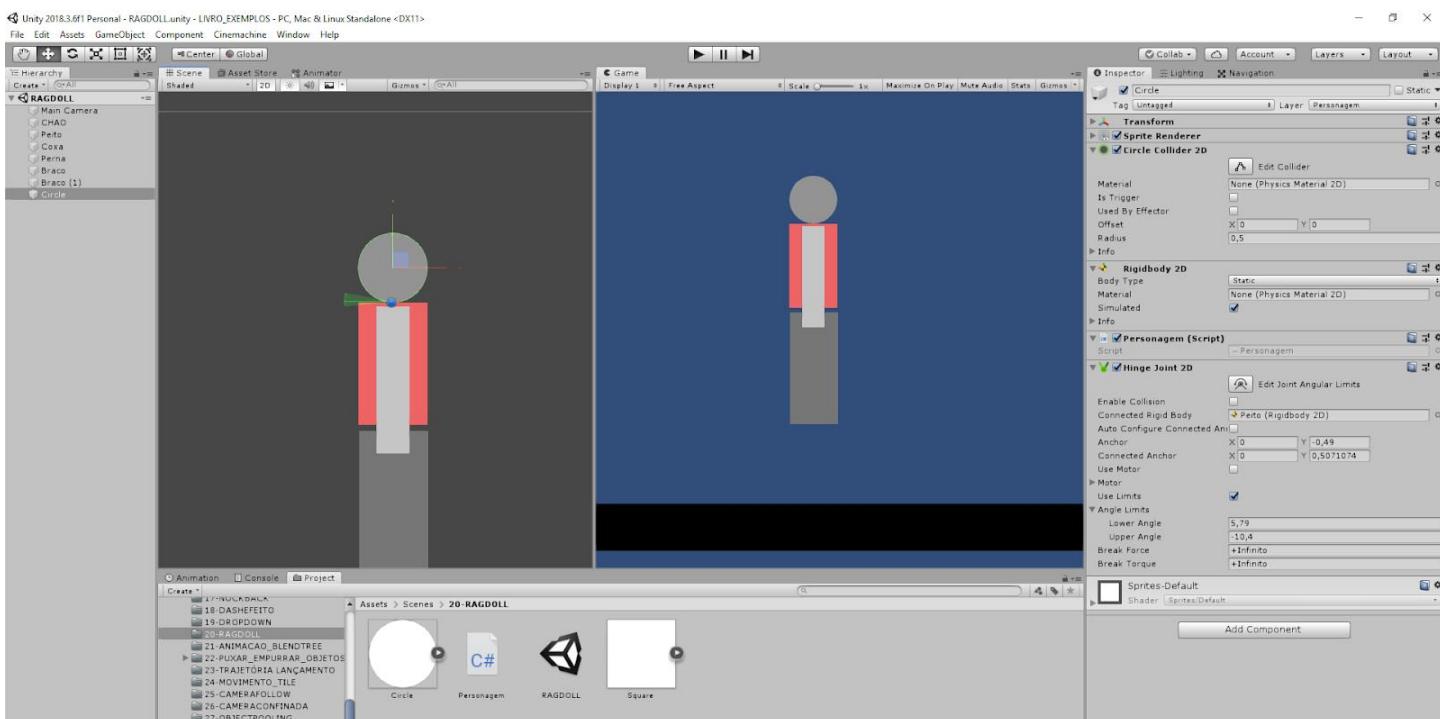


E como é feito para unir cada parte e deixar parecendo com articulações humanas?
 É simples vamos ver a estrutura desses objetos.

Veja na imagem abaixo que a cabeça é uma sprite circular com um corpo colisor, um rigidbody2D, um código chamado **Personagem** e por último um HingeJoint2D.

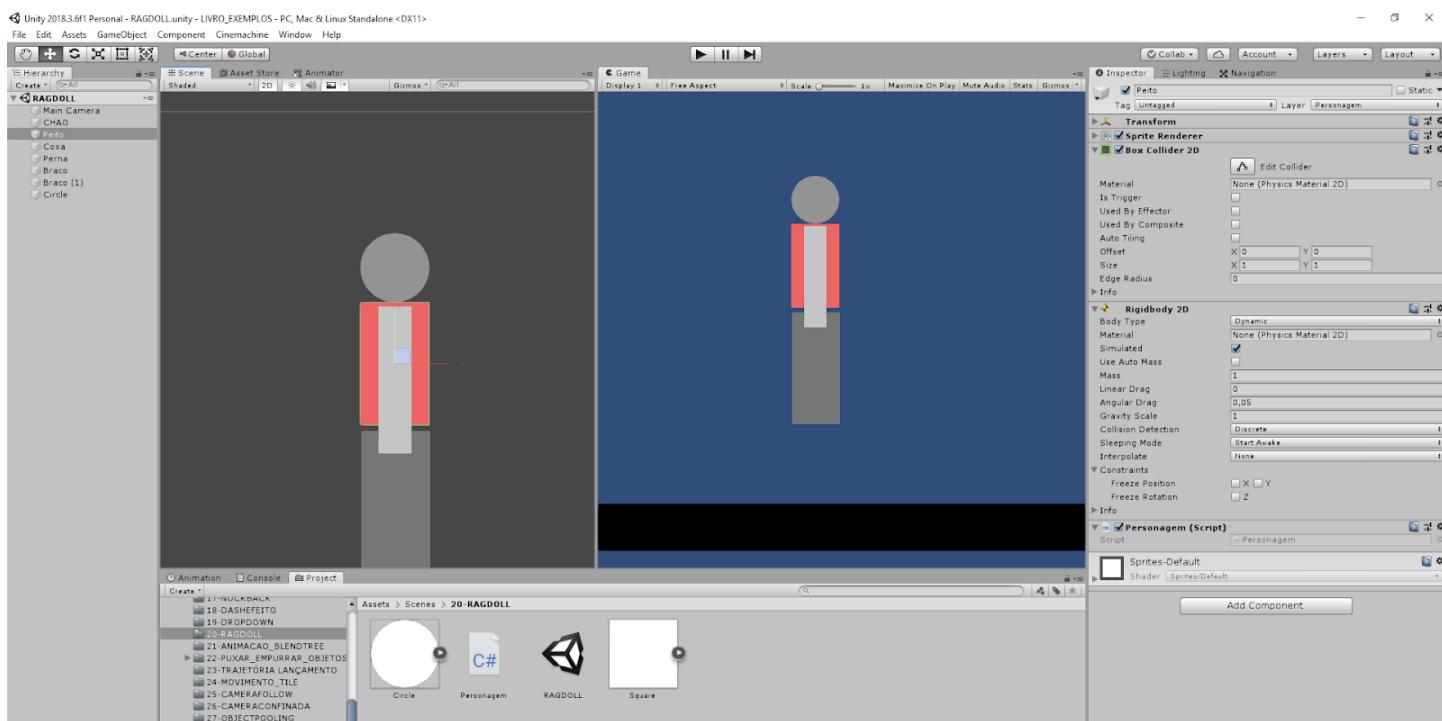
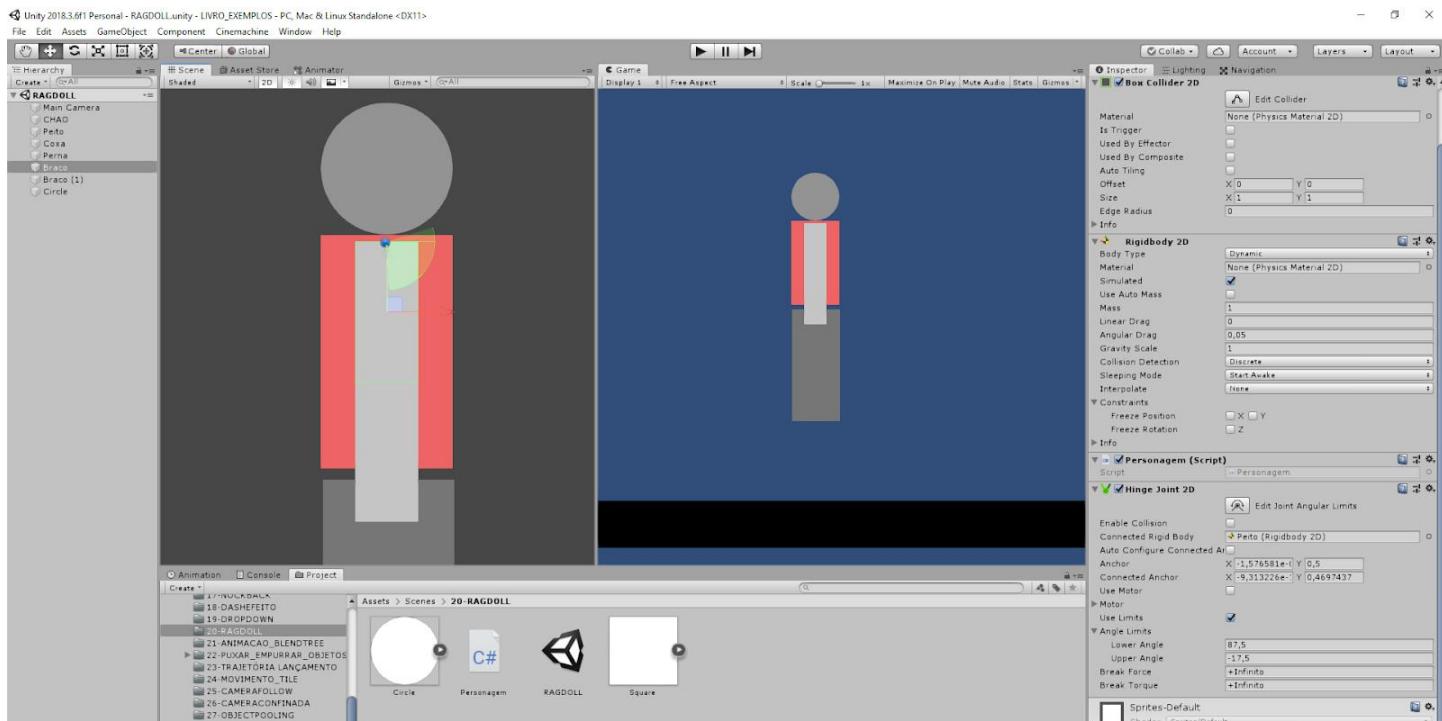
Esse HingeJoint2D é responsável por conectar a cabeça com o peito do personagem, veja que isso esta claro no campo Connected Rigidbody.

Mais abaixo ligamos a opção de usar limites para poder limitar a rotação da cabeça em Angle Limits.

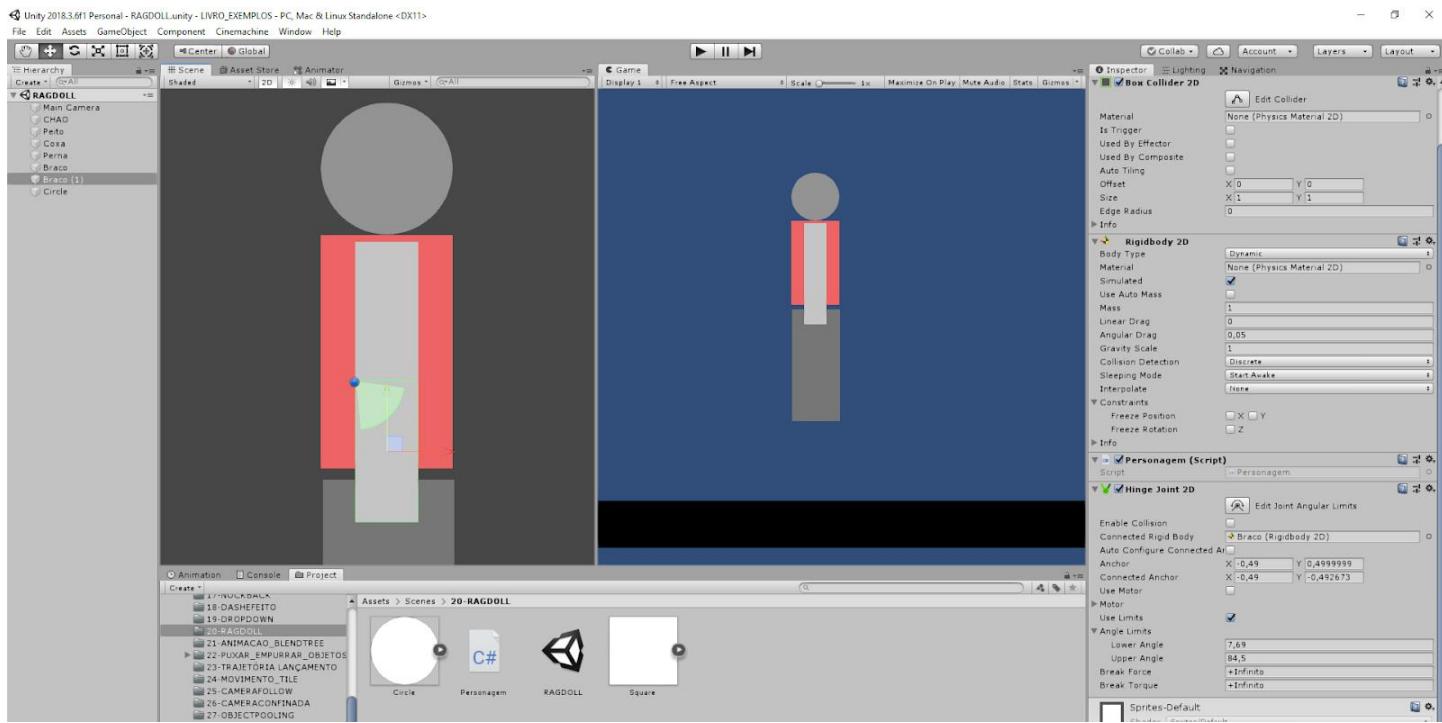


Depois disso selecionamos o peito do personagem para analisar sua estrutura de componentes e veja que nesse caso temos apenas um corpo colisor, um rigidbody2D e o mesmo código Personagem. Nesse caso não foi necessário um HingeJoint2D pois essa parte do corpo não se liga em ninguém, pelo contrário as outras partes que se ligam ao peito.

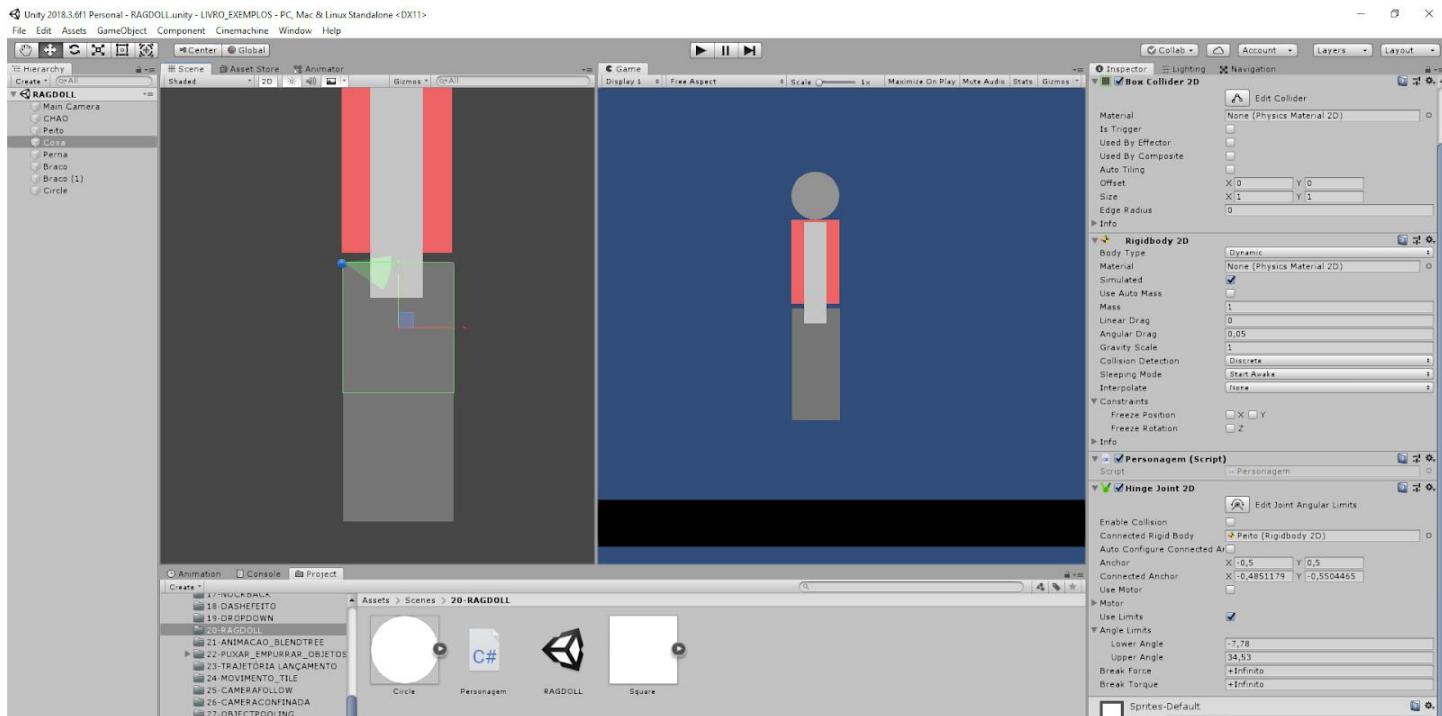
Então continuando temos o braço que como mostra a imagem tem os mesmos componentes, o corpo colisor, o rigidbody, o código e o hingeJoint2D, que no caso está ligado ao peito do personagem porem com configurações de limite diferentes já que aqui temos a rotação de um braço.



Depois do braço temos o antebraço que tem as mesmas configurações que o braço com uma pequena diferença esse objeto não se liga ao peito do personagem mas sim ao braço, veja:

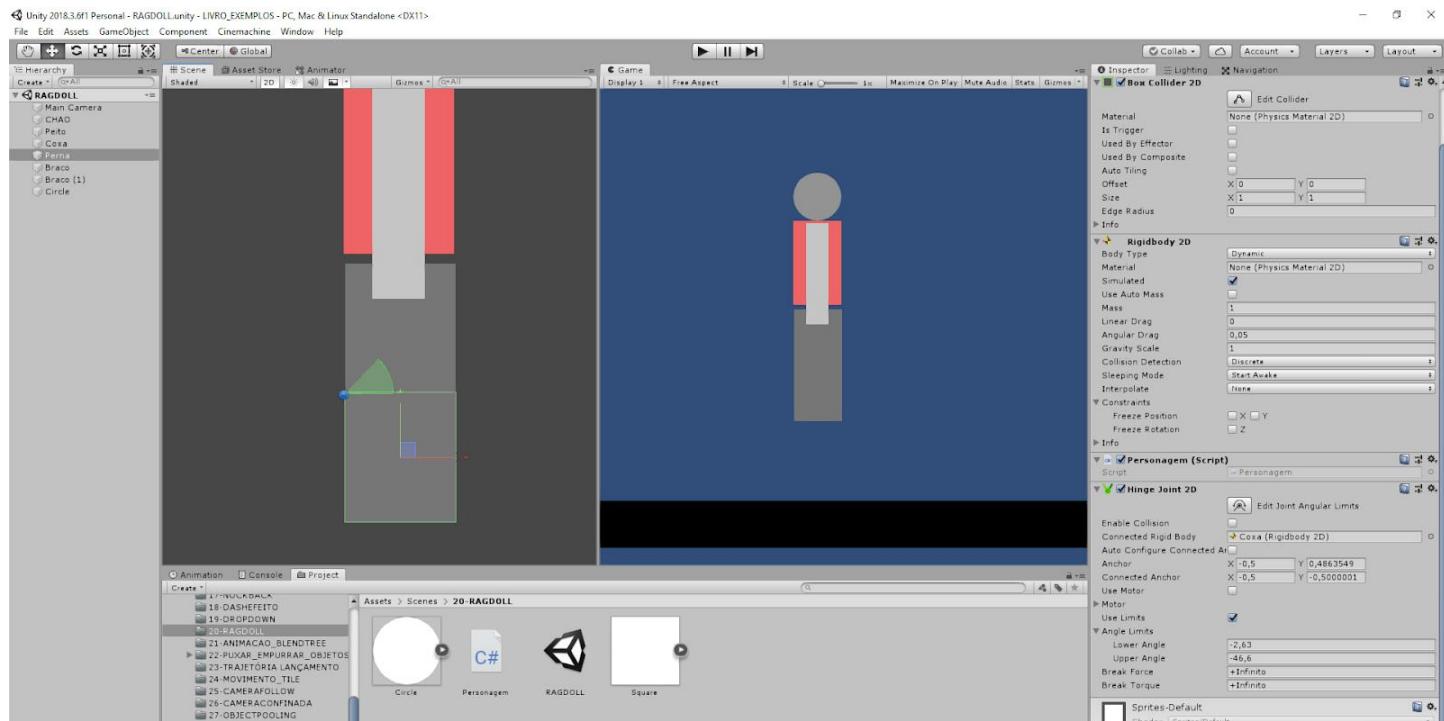


Agora descendo mais na nossa estrutura humanoide temos a coxa do personagem que como você pode ver na imagem abaixo tem os mesmos componentes do braço e até está ligado ao mesmo rigidbody que é o peito.

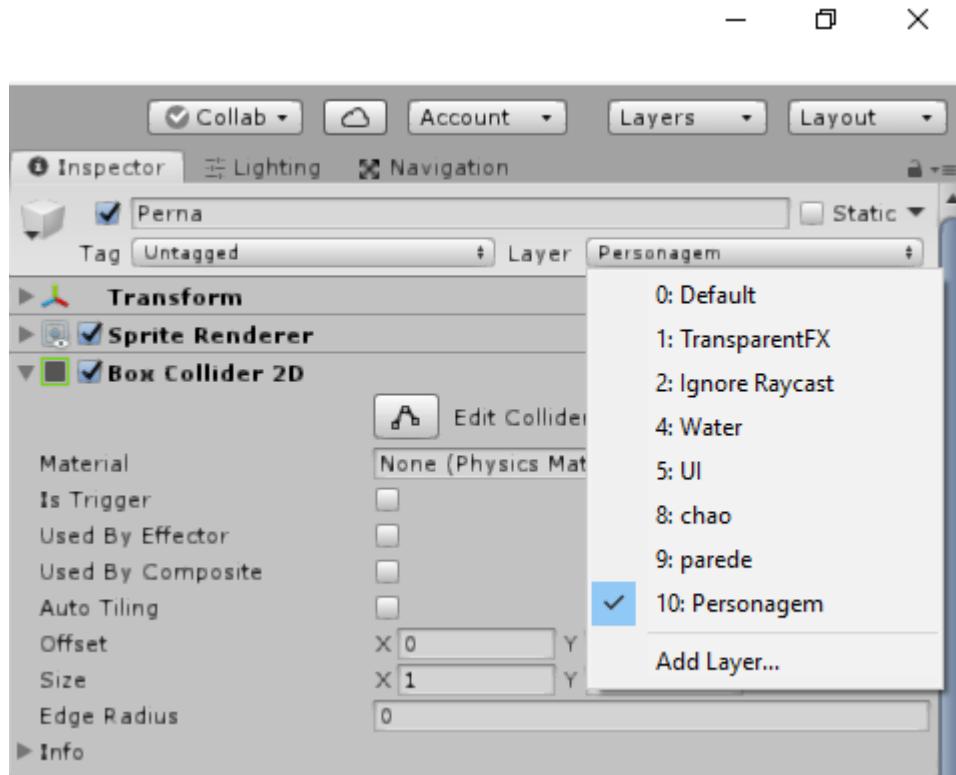


A única diferença são os limites dessa parte do corpo.

E para finalizar temos a perna que tem os mesmos componentes da coxa mas com limites diferentes e essa parte do corpo não é ligada ao peito mas sim a coxa.



Prontinho nossa estrutura esta montada agora só falta mostrar alguns pequenos ajustes como por exemplo a adição de um layer chamado Personagem em cada parte do corpo.



E depois o código que é usado em cada parte do corpo que é esse aqui:

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

public class Personagem : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        Physics2D.IgnoreLayerCollision(10, 10);
    }

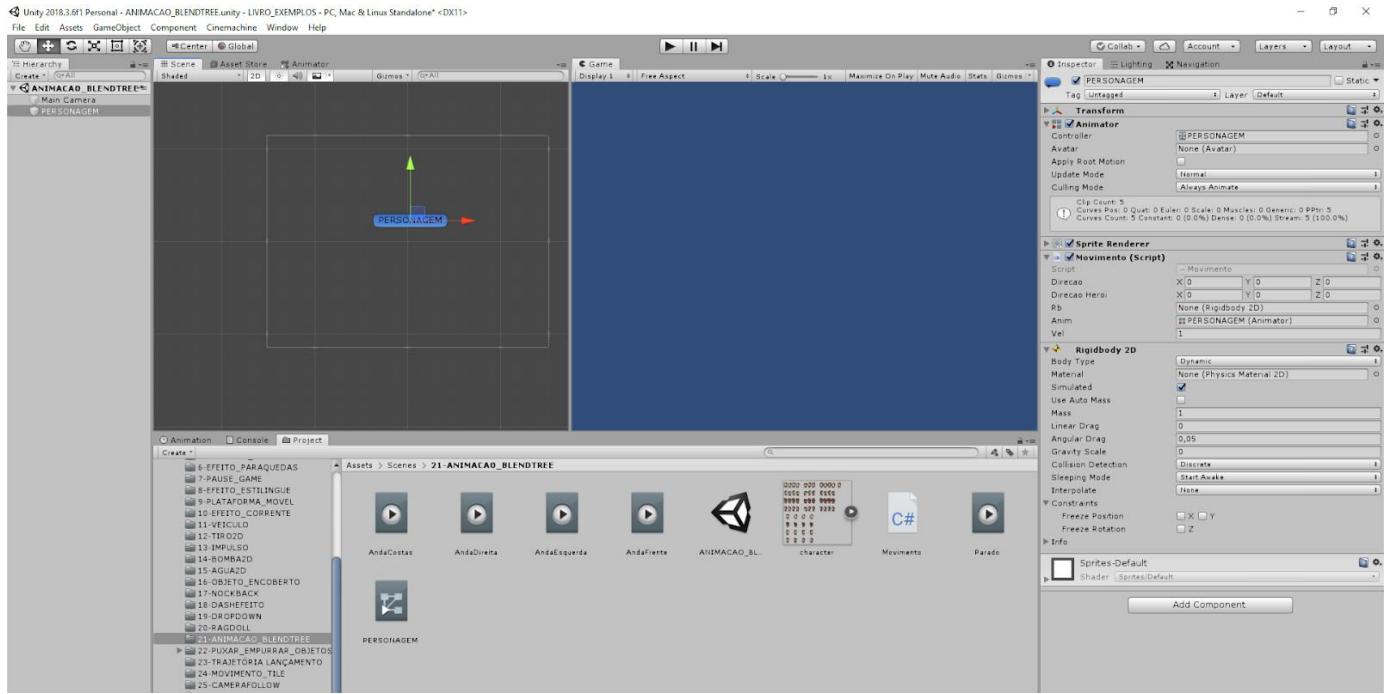
    // Update is called once per frame
    void Update()
    {
    }
}
```

Veja que esse código é muito simples a única coisa que foi feita aqui é dizer para o Unity que objetos com o layer 10 ou seja Personagem devem se ignorar.

Dessa forma teremos o movimento livre sem nenhuma colisão travando as articulações.

ANIMAÇÃO COM BLENDTREE

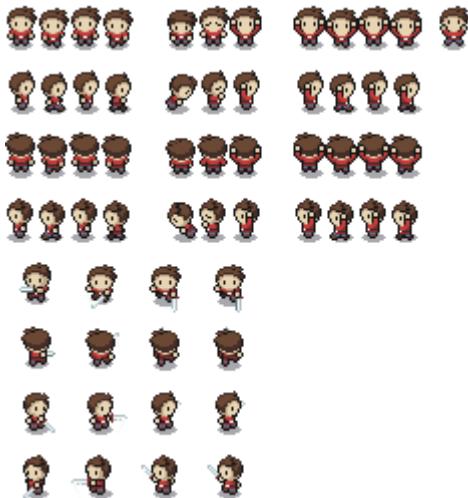
Em qualquer jogo é preciso pensar nas animações dos personagens, na melhor forma de cria-las levando em consideração que podemos querer usar essa lógica para criar qualquer outra animação. Sendo assim vamos ver nesse exemplo como criar uma animação com BlendTree. Para isso crie uma cena semelhante a da imagem abaixo:



Veja que essa cena é muito simples a única coisa que temos ai é um GameObject vazio que representa o personagem animado.

Veja também que esse personagem tem um Animator, um Rigidbody2D e um código chamado Movimento, nesse caso não adicionei nenhum corpo físico pois não é necessário para o exemplo mas, para um game a adição do corpo é necessária.

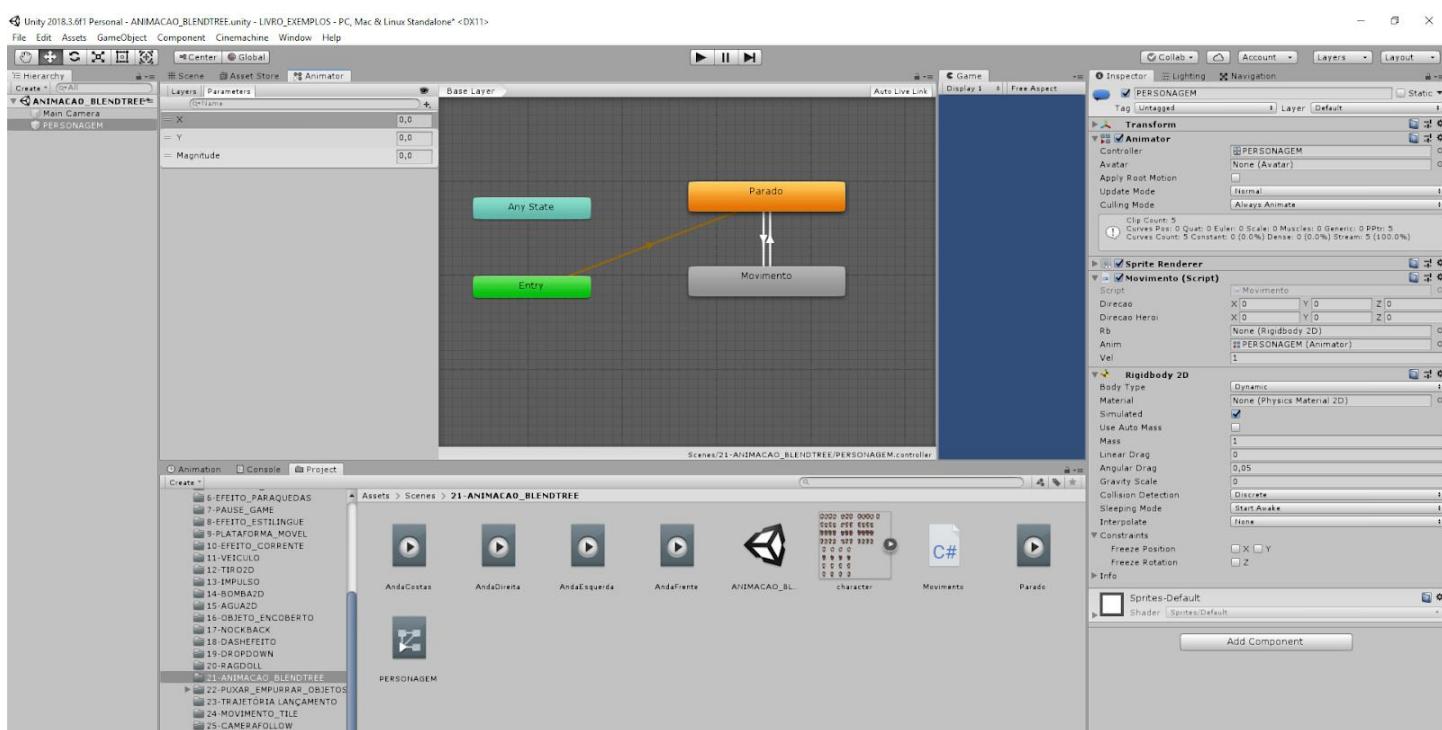
Certo o objeto que representa o personagem tem todos esses componentes e uma sequência de animações que foi extraída dessa imagem:



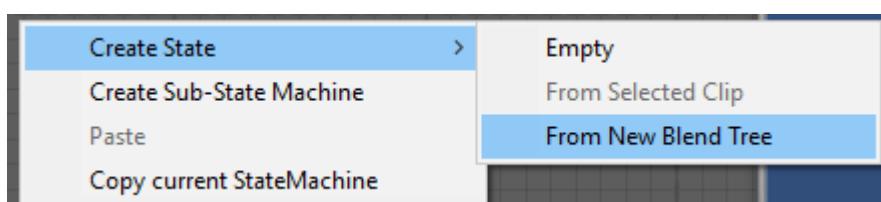
Depois usamos as animações que foram extraídas dentro do nosso Animator.

Sendo que a primeira parte da configuração do Animator necessita de três parâmetros float um para X, outro para Y e um para Magnitude.

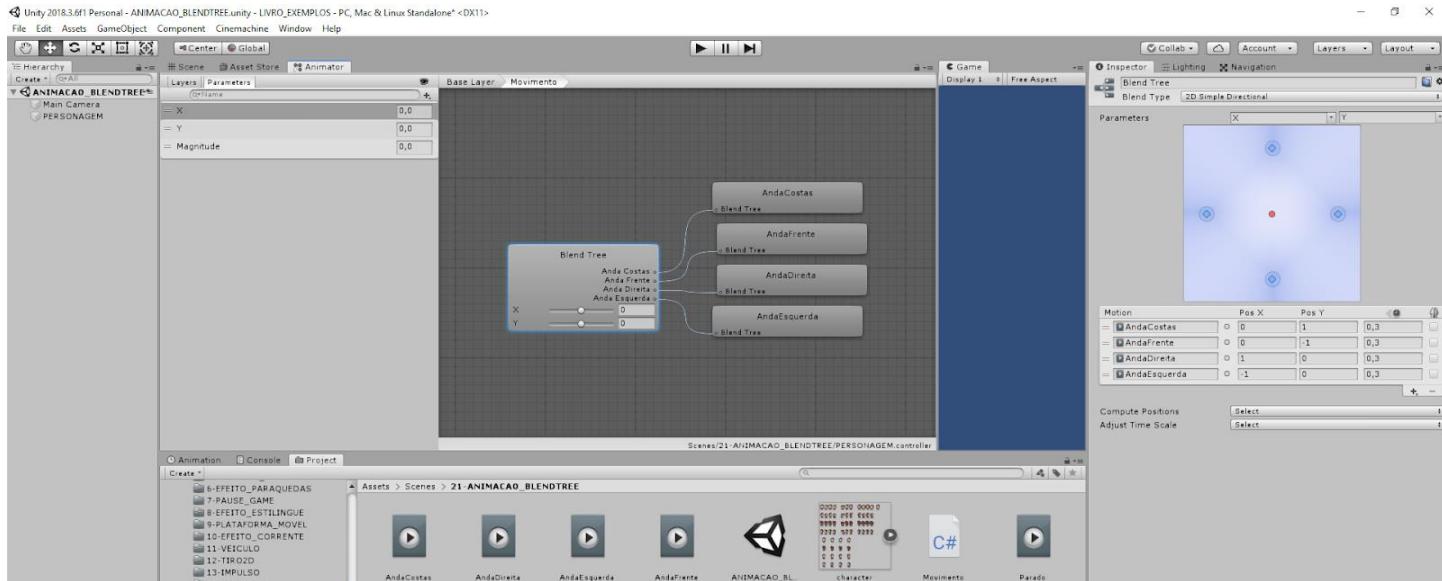
Logo depois disso ao lado temos uma transição de animações que vai da animação Parado para o BlendTree de movimento.



Lembre-se que para criar um BlendTree é necessário clicar na área de animações com o botão direito do mouse e escolher a opção Create State seguida de From New Blend Tree.



Com isso teremos o blendTree que deve ser configurado conforme mostra a imagem abaixo:

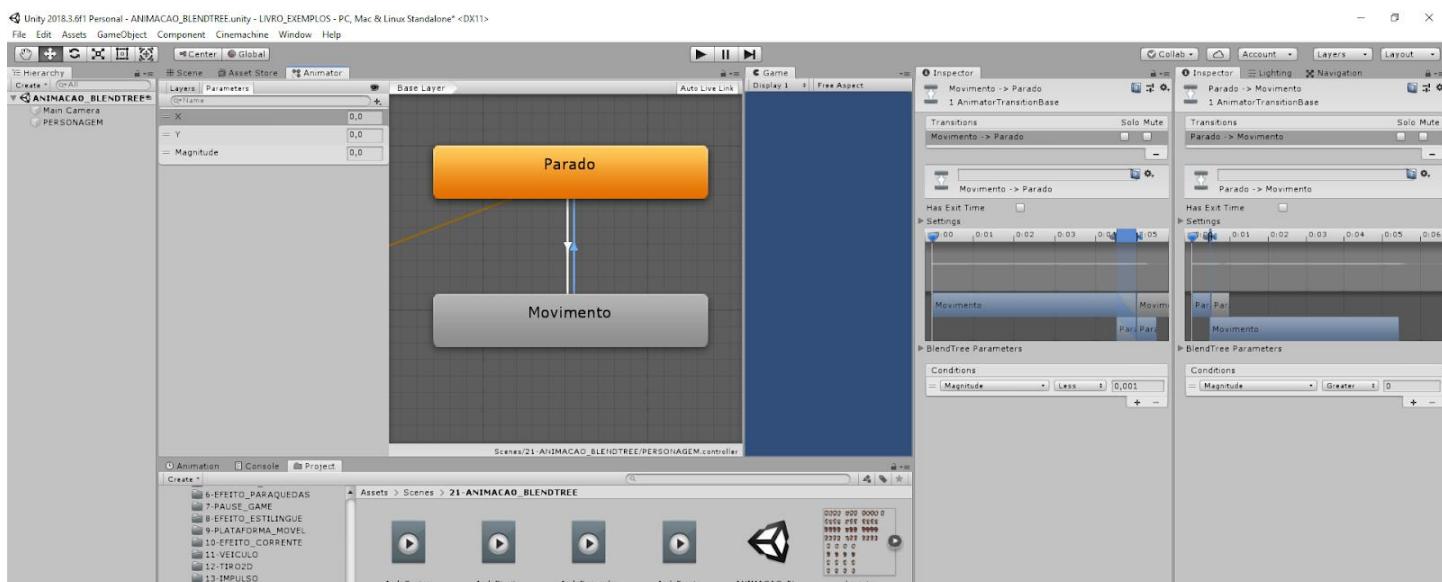


Veja que adicionamos as animações dentro do BlendTree e já sinalizamos que essas animações vão ser controladas por dois parâmetros o X e o Y.

Tudo isso levando em conta os valores que esses parâmetros vão receber e comparar com os valores que adicionamos em Motion.

Para fechar essa configuração precisamos olhar novamente à transição entre a animação de parado e o blend de movimento.

Veja que as transições envolvidas no processo trabalham com o parâmetro Magnitude se esse parâmetro tem seu valor menor que 0,001 passamos do blend de movimento para a animação parado e caso esse parâmetro tenha seu valor maior que 0 passamos da animação parado para o blendTree.



Com isso configurado vamos analisar o código usado nesse exemplo.

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

```

[RequireComponent(typeof(Rigidbody2D))]
public class Movimento : MonoBehaviour
{
    public Vector3 direcao;
    public Vector3 direcaoHeroi;
    public Rigidbody2D rb;
    public Animator anim;
    public float vel;
    // Start is called before the first frame update
    void Start()
    {
        direcao = Vector3.zero;
        rb = GetComponent<Rigidbody2D>();
    }

    // Update is called once per frame
    void Update()
    {
        InputPersonagem();

        if (direcao.x != 0 || direcao.y != 0)
        {
            anim.SetFloat("X", direcao.x);
            anim.SetFloat("Y", direcao.y);

        }

        anim.SetFloat("Magnitude", direcao.magnitude);

        print(direcao.magnitude);
    }

    void InputPersonagem()
    {
        direcao = Vector3.zero;

        if (Input.GetKey(KeyCode.UpArrow))
        {
            direcao += new Vector3(0, 1, 0); //Vector2.up
            direcaoHeroi = direcao;
        }
        if (Input.GetKey(KeyCode.DownArrow))
        {
            direcao += new Vector3(0, -1, 0); //Vector2.down
            direcaoHeroi = direcao;
        }
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            direcao += Vector3.left; //Vector2.left
        }
    }
}

```

```

        direcaoHeroi = direcao;
    }
    if (Input.GetKey(KeyCode.RightArrow))
    {
        direcao += Vector3.right; //Vector2.right
        direcaoHeroi = direcao;
    }
}

void FixedUpdate()
{
    Mover();
}

void Mover()
{
    rb.MovePosition(rb.transform.position + direcao * vel * Time.deltaTime); //ajustado
}
}

```

Veja que nesse exemplo temos algumas variáveis para que possamos trabalhar, uma do tipo Vector3 que é a direção, depois outra variável do mesmo tipo mas para direção do herói, uma variável para o nosso rigidbody outra para o nosso Animator e por último uma para a velocidade.

```

public Vector3 direcao;
public Vector3 direcaoHeroi;
public Rigidbody2D rb;
public Animator anim;
public float vel;

```

Antes de continuar passando para o método Start preste a atenção nessa linha:

```
[RequireComponent(typeof(Rigidbody2D))]
```

Ela foi adicionada acima da nossa classe para dizer que o objeto que tem esse código precisa ter um

componente do tipo Rigidbody2D.

só isso mesmo nada de complexo, continuando, agora sim vamos ver o método Start.

```

void Start()
{
    direcao = Vector3.zero;
    rb = GetComponent<Rigidbody2D>();
}

```

Nele estamos passando o valor de Vector3.zero para a nossa direção e logo em seguida estamos pegando o rigidbody do objeto que tem esse código.

```
void Update()
```

```

{
    InputPersonagem();

    if (direcao.x != 0 || direcao.y != 0)
    {
        anim.SetFloat("X", direcao.x);
        anim.SetFloat("Y", direcao.y);

    }

    anim.SetFloat("Magnitude", direcao.magnitude);

    print(direcao.magnitude);
}

```

No método Update a primeira coisa que fizemos foi chamar o método InputPersonagem que vamos ver mais adiante.

Em seguida temos uma estrutura condicional que verifica se a direção X ou Y é diferente de zero, se for passamos os valores de direcao.x e direcao.y para os parâmetros X e Y do nosso Animator. Isso vai fazer com que nosso BlendTree funcione.

Depois fora dessa estrutura condicional passamos para o parâmetro Magnitude do nosso animator o valor de direcao.magnitude para garantir que a transição de parado e andando possa acontecer.

No fim apenas usamos um print para acompanhar a magnitude mudando de valor.

Agora vamos dar uma olhadinha no método de Input do nosso personagem.

```

void InputPersonagem()
{
    direcao = Vector3.zero;

    if (Input.GetKey(KeyCode.UpArrow))
    {
        direcao += new Vector3(0, 1, 0); //Vector2.up
        direcaoHeroi = direcao;
    }
    if (Input.GetKey(KeyCode.DownArrow))
    {
        direcao += new Vector3(0, -1, 0); //Vector2.down
        direcaoHeroi = direcao;
    }
    if (Input.GetKey(KeyCode.LeftArrow))
    {
        direcao += Vector3.left; //Vector2.left
        direcaoHeroi = direcao;
    }
    if (Input.GetKey(KeyCode.RightArrow))
    {
        direcao += Vector3.right; //Vector2.right
        direcaoHeroi = direcao;
    }
}

```

Veja que esse método já inicia passando o valor de vector3.zero para a nossa direção e só depois faz as verificações de teclas sendo apertadas.

Para cada tecla apertada temos um valor determinado para ser passado para a nossa direção.

Isso faz com que o personagem possa se mover em qualquer direção.

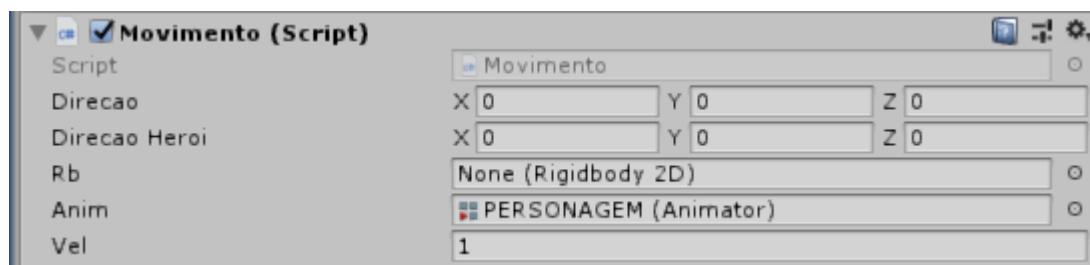
Por último temos dois métodos um é o FixedUpdate que é onde chamamos o método Mover que como o próprio nome diz é responsável por mover o personagem.

E depois temos o método Mover em si que apenas trabalha sobre o rigidbody do personagem definindo seu movimento de acordo com a multiplicação da posição pela direção e velocidade.

```
void FixedUpdate()
{
    Mover();
}

void Mover()
{
    rb.MovePosition(rb.transform.position + direcao * vel *
Time.deltaTime); //ajustado
}
```

Com isso o exemplo está pronto para ser executado só precisamos passar para esse código quem é o animator e qual o valor de velocidade tudo isso lá no Inspector veja:

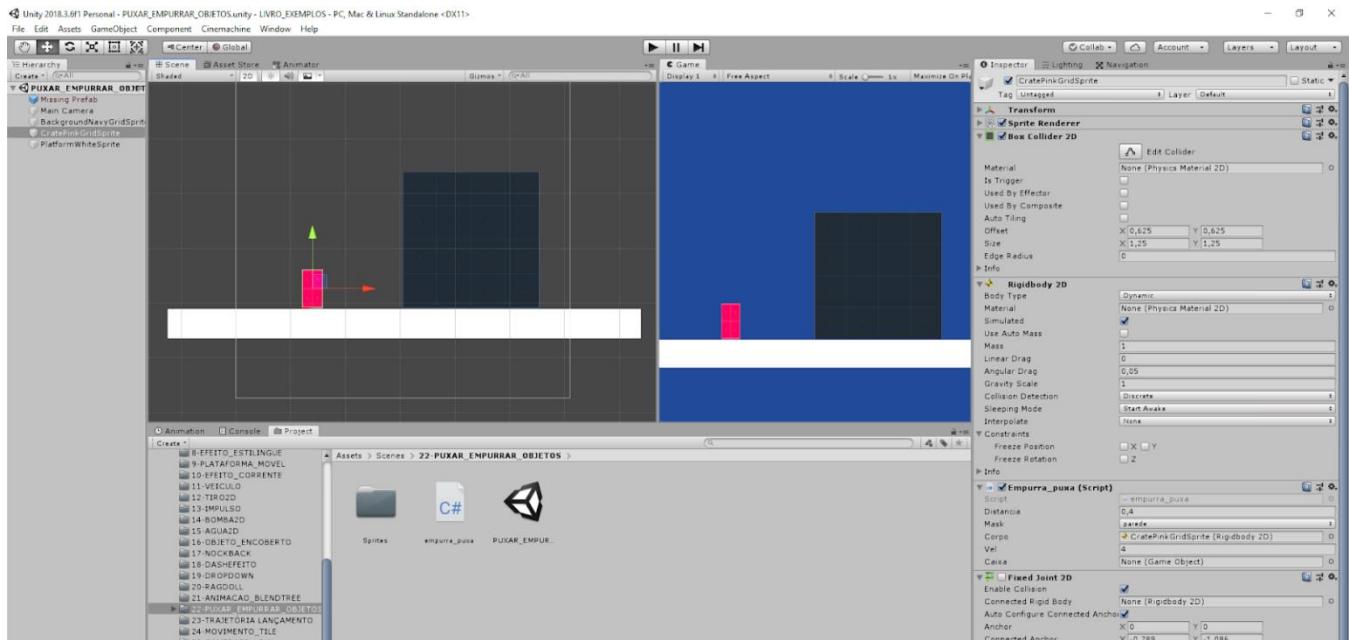


E pronto agora é só executar o nosso exemplo.

PUXAR E EMPURRAR OBJETOS

Agora vamos falar de mais uma mecânica muito legal e muito usada no mundo dos games que é de empurrar e puxar objetos de cena.

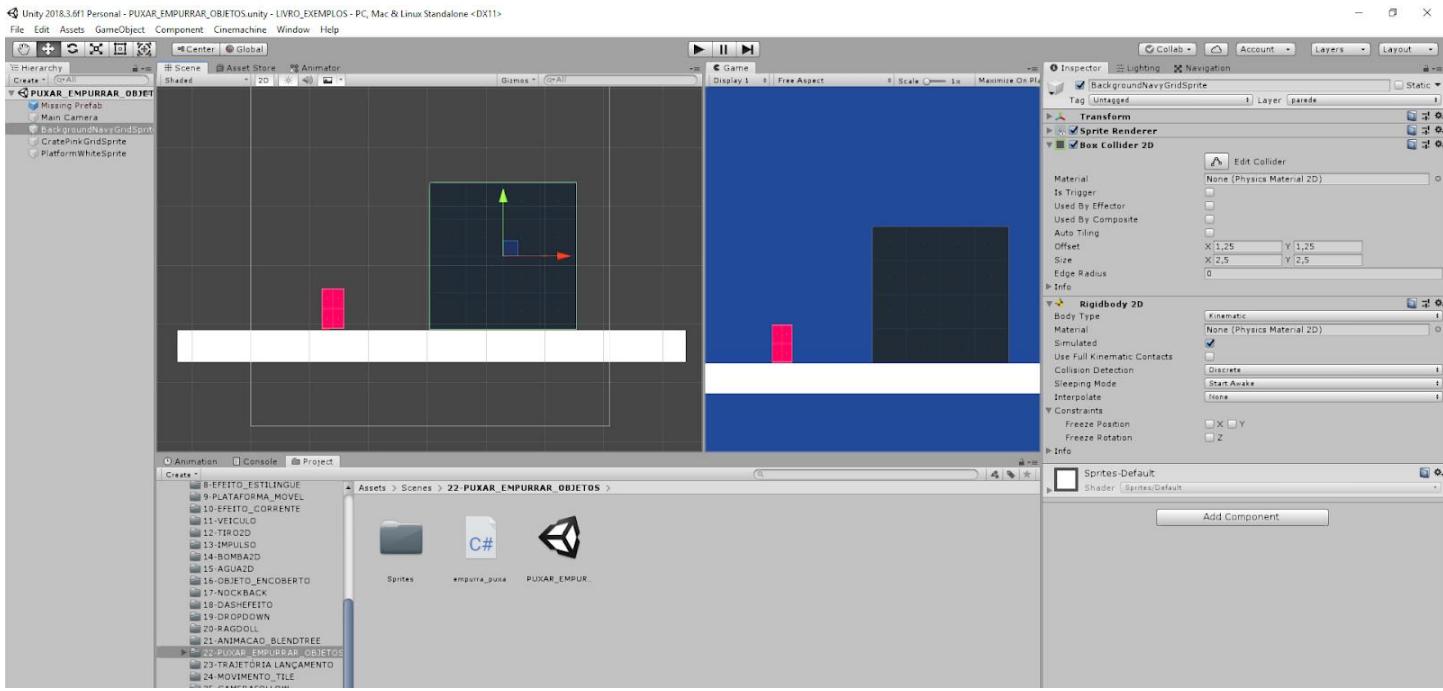
Para criar esse efeito precisamos de uma cena semelhante a da imagem abaixo:



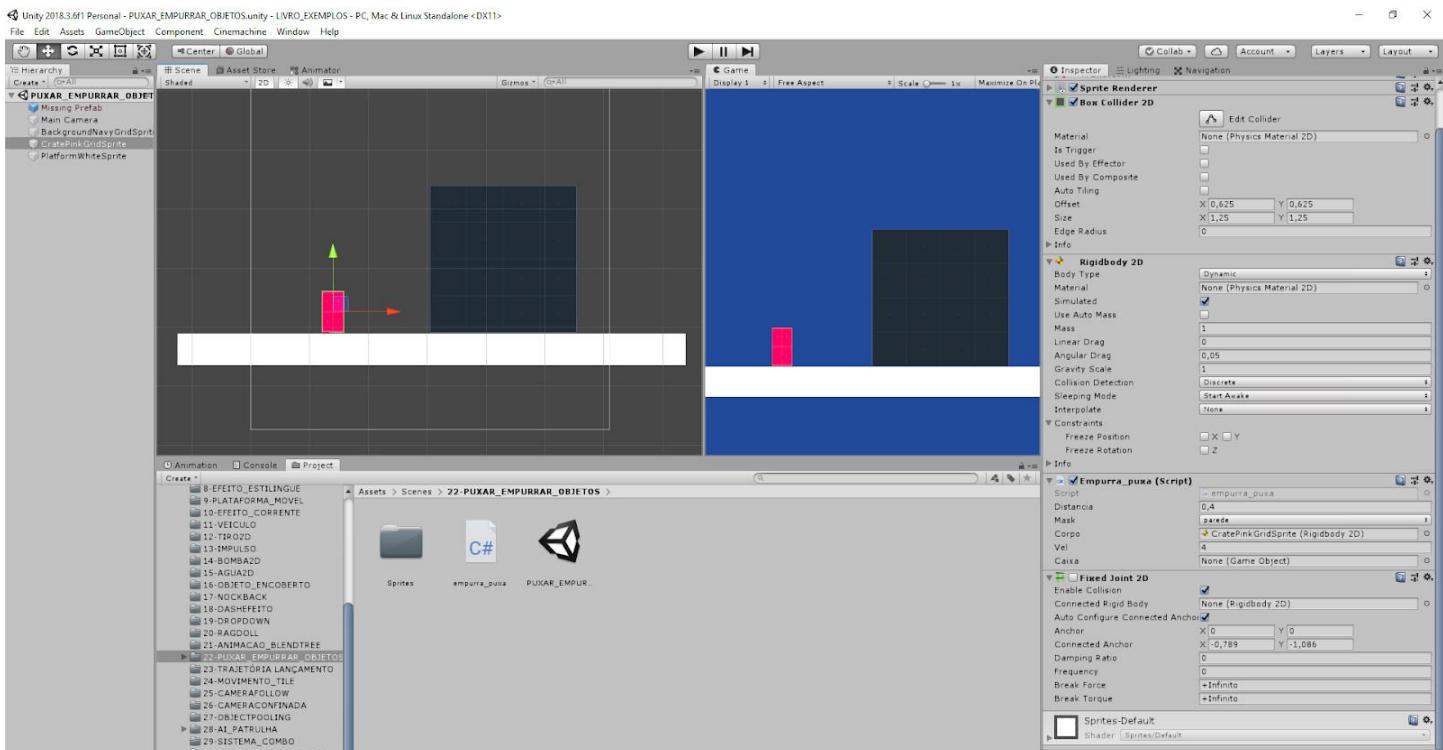
Veja que nessa cena temos só sprites quadradas que foram ajustadas em sua escala para representar chão, personagem e objeto que deve ser arrastado.

O chão é o mais básico dessa estrutura ele tem apenas um corpo colisor.

Já o bloco cinza escuro que vai ser arrastado tem os seguintes componentes, um corpo colisor e um rigidbody2D.



O personagem representado pelo quadrado rosa tem um corpo colisor, um Rigidbody2D, um código chamado Empurra_puxa e um FixedJoint2D que é quem vai linkar o personagem ao bloco que deve ser arrastado pela cena.



Agora vamos analisar o código que faz esse exemplo funcionar.

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

```

using UnityEngine;

public class empurra_puxa : MonoBehaviour {

    public float distancia;
    public LayerMask mask;
    public Rigidbody2D corpo;
    public float vel;
    public GameObject caixa;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void FixedUpdate () {

        Physics2D.queriesStartInColliders = false;
        RaycastHit2D hit = Physics2D.Raycast
(transform.position,Vector2.right *
transform.localScale.x,distancia,mask);

        if (hit.collider != null) {
            caixa = hit.collider.gameObject;

            if (Input.GetKey (KeyCode.Z)) {

                caixa.GetComponent<Rigidbody2D> ().isKinematic =
false;
                this.GetComponent<FixedJoint2D> ().enabled = true;
                this.GetComponent<FixedJoint2D> ().connectedBody =
caixa.GetComponent<Rigidbody2D> ();

            } else {

                this.GetComponent<FixedJoint2D> ().connectedBody =
null;
                this.GetComponent<FixedJoint2D> ().enabled = false;

                if (caixa.GetComponent<Rigidbody2D> ().velocity.sqrMagn
itude < 0.01f)
                {
                    caixa.GetComponent<Rigidbody2D> ().Sleep ();
                    caixa.GetComponent<Rigidbody2D> ().isKinematic =
true;
                }
            }
        } else {
    }
}

```

```

        caixa = null;
    }

    if (Input.GetKey(KeyCode.RightArrow))
    {
        corpo.velocity = new Vector2 (vel,corpo.velocity.y);
    }
    if (Input.GetKey(KeyCode.LeftArrow))
    {
        corpo.velocity = new Vector2 (-vel,corpo.velocity.y);
    }
}

void OnDrawGizmos()
{
    Gizmos.color = Color.red;
    Gizmos.DrawLine (transform.position,
    (Vector2)transform.position + Vector2.right * transform.localScale.x
    * distancia);
}
}

```

Para esse exemplo começamos criando algumas variáveis como um float que controla a distância do personagem e o objeto que vai ser manipulado, um layrmask que identifica qual objeto posso arrastar. Um rigidbody2D que é o rigidbody do próprio personagem, um float para velocidade e por último uma variável do tipo GameObject que representa o objeto arrastado em cena.

```

public float distancia;
public LayerMask mask;
public Rigidbody2D corpo;
public float vel;
public GameObject caixa;

```

Depois disso dentro do método FixedUpdate a primeira coisa que precisamos fazer é definir que o raycast que inicia dentro do colisor não deve detectar esse colisor.

Depois disso passamos para o RaycastHit2D sua origem, direção distancia e layermask.

Dessa forma conseguiremos verificar se colidimos com a caixa e se isso realmente aconteceu passamos essa informação para a variável caixa.

Em seguida criamos uma estrutura condicional que verifica se apertamos a tecla Z que define que caixa não é isKinematic, habilitamos o FixedJoint2D e já conectamos com a caixa para que seja possível arrasta-la ou empurra-la.

Caso contrario se deixarmos de apertar a tecla Z deixamos o objeto parado sem sofrer mais nenhuma alteração de acordo com o nosso movimento que é definido na estrutura condicional lá no final que move o corpo para a direita ou para a esquerda.

```

void FixedUpdate () {

    Physics2D.queriesStartInColliders = false;
    RaycastHit2D hit = Physics2D.Raycast
(transform.position,Vector2.right *
transform.localScale.x,distancia,mask);

    if (hit.collider != null) {
        caixa = hit.collider.gameObject;

        if (Input.GetKey (KeyCode.Z)) {

            caixa.GetComponent<Rigidbody2D> ().isKinematic =
false;
            this.GetComponent<FixedJoint2D> ().enabled = true;
            this.GetComponent<FixedJoint2D> ().connectedBody =
caixa.GetComponent<Rigidbody2D> ();

        } else {

            this.GetComponent<FixedJoint2D> ().connectedBody =
null;
            this.GetComponent<FixedJoint2D> ().enabled = false;

            if (caixa.GetComponent<Rigidbody2D> ().velocity.sqrMagnitude < 0.01f)
            {
                caixa.GetComponent<Rigidbody2D> ().Sleep ();
                caixa.GetComponent<Rigidbody2D> ().isKinematic =
true;
            }
        }
    } else {

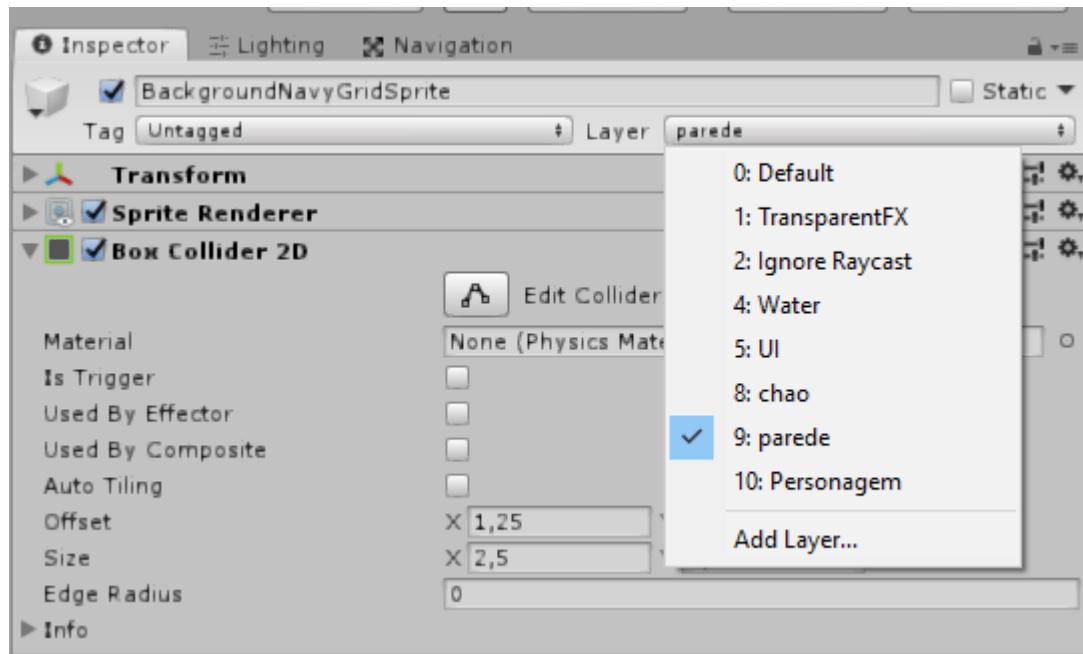
        caixa = null;
    }

    if (Input.GetKey (KeyCode.RightArrow) )
    {
        corpo.velocity = new Vector2 (vel,corpo.velocity.y);
    }
    if (Input.GetKey (KeyCode.LeftArrow) )
    {
        corpo.velocity = new Vector2 (-vel,corpo.velocity.y);
    }
}

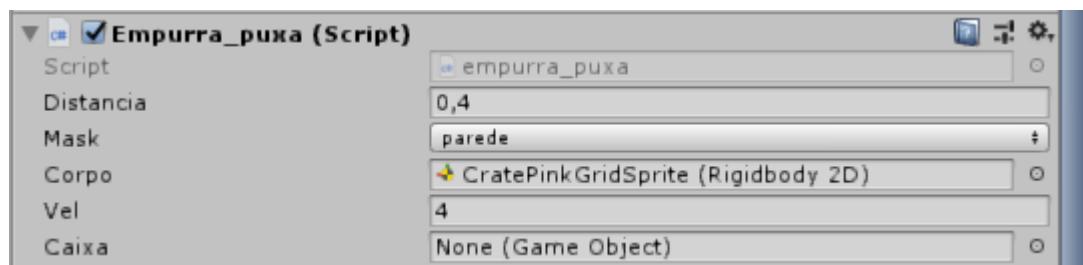
```

E agora para fechar o código temos um método que simplesmente nos mostra a linha do nosso raio para que possamos verificar a distância que é usada para pegar o objeto.

Feito isso só precisamos voltar para o Unity e adicionar um layer na caixa que deve ser empurrada veja:



E depois ajustamos o código do personagem para trabalhar com um valor de distância, com o layer parede, o rigidbody do próprio personagem, uma velocidade média e pronto.



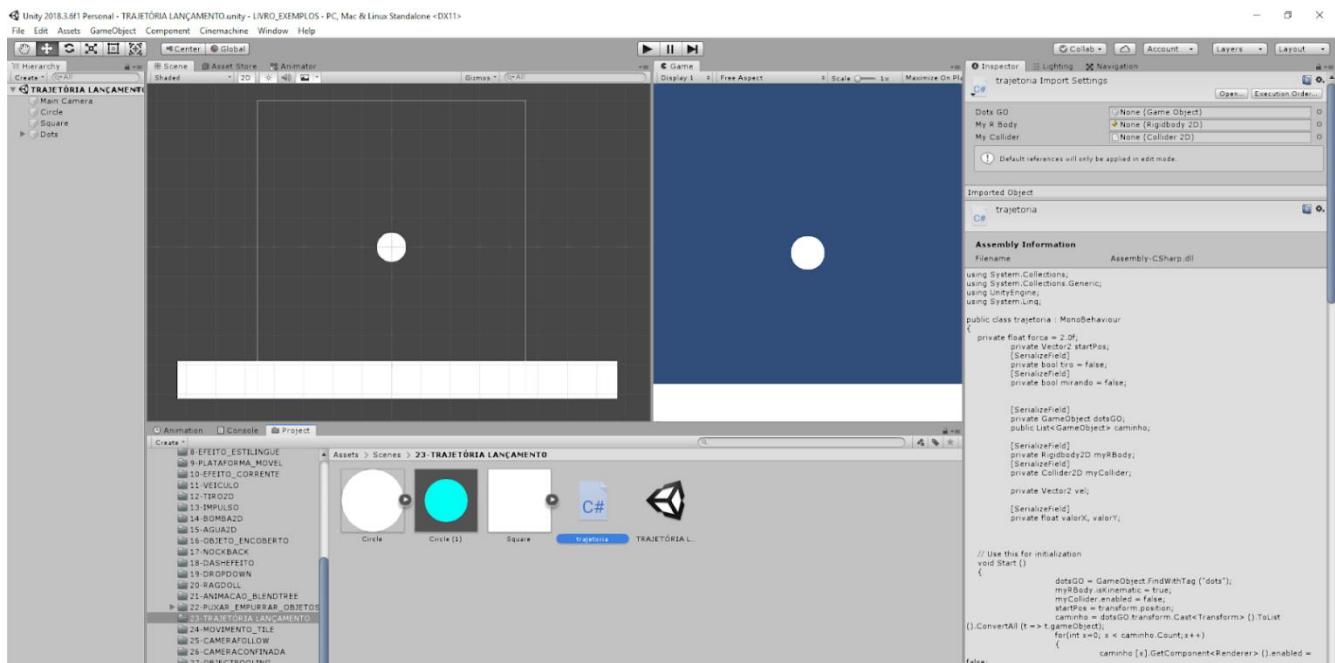
Agora é só testar o exemplo.

TRAJETÓRIA DE LANÇAMENTO

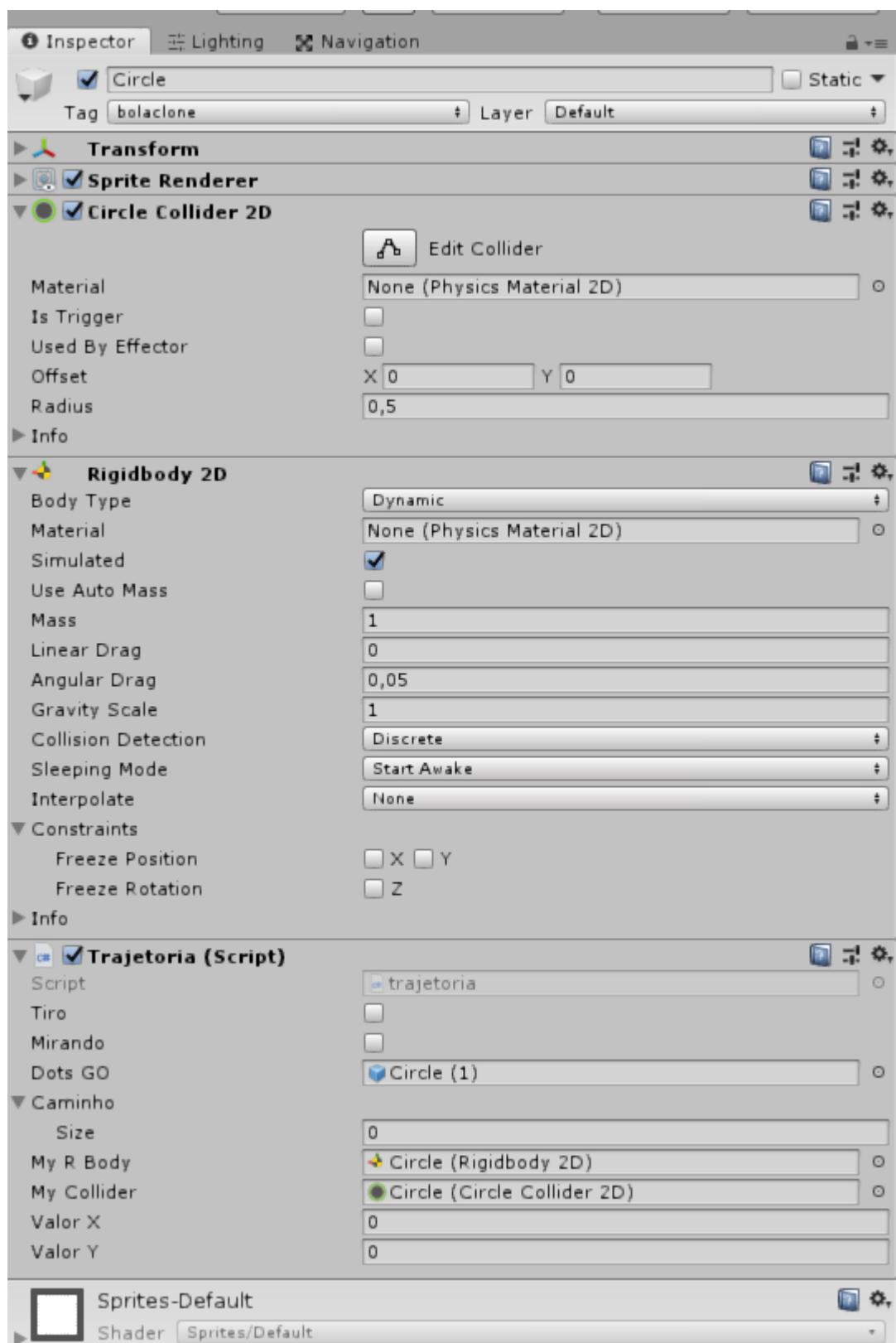
Uma mecânica muito interessante que pode ser usada em vários tipos de jogos é a trajetória de lançamento.

É comum ver jogos de basquete que usam essa mecânica.

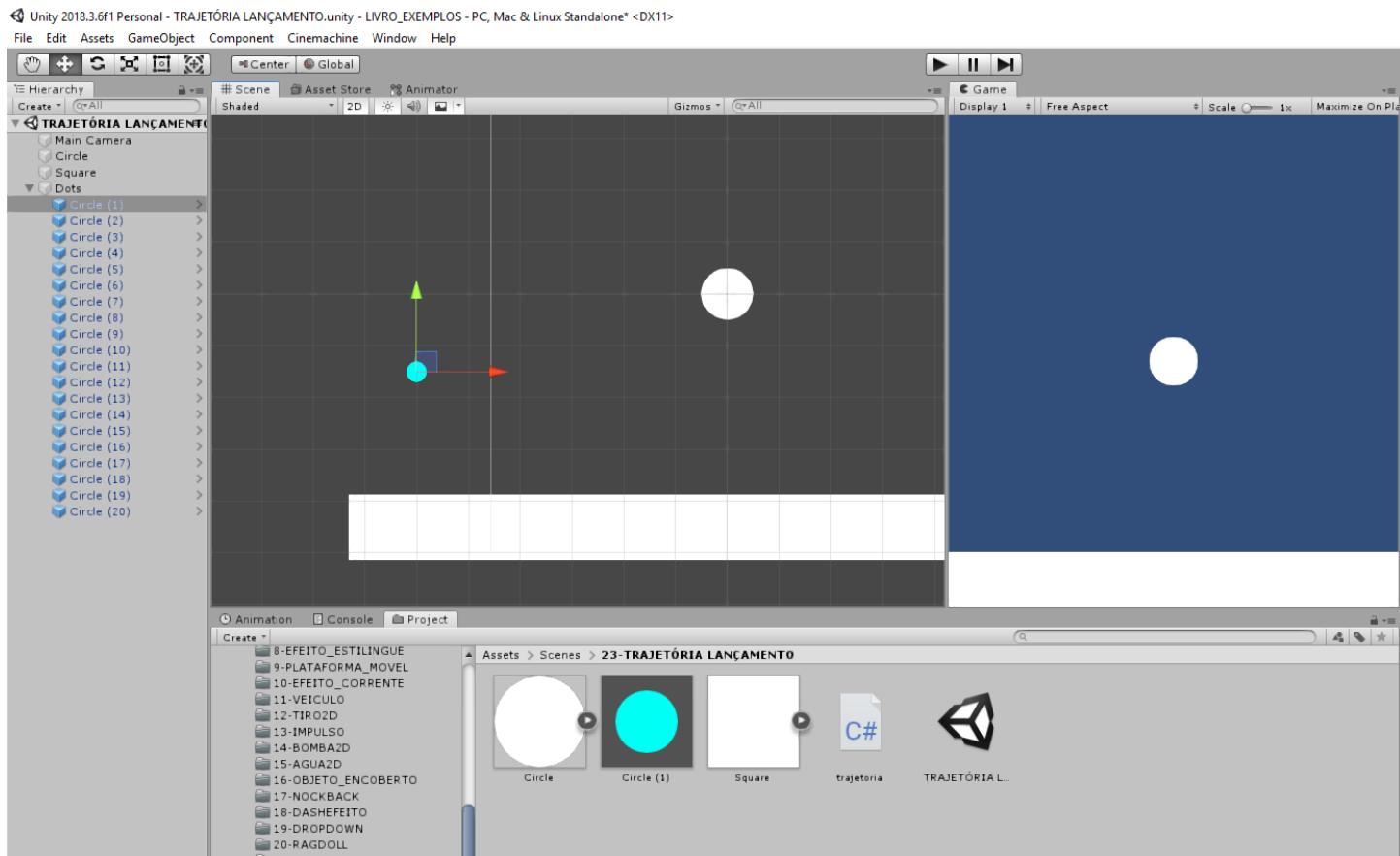
Então vamos ver como fazer algo desse tipo, mas para isso vamos criar uma cena parecida com a da imagem abaixo:



Veja que essa cena é muito simples temos apenas uma esfera e um retângulo aqui, sendo que o retângulo que representa o chão tem apenas um corpo colisor enquanto que a esfera tem mais alguns componentes.



Além disso temos também um conjunto de objetos azul claro que estão definidos como filhos do GameObject. Dots, esses caras vão ser usados para desenhar a trajetória de lançamento.



Entre os componentes existentes na bola vamos falar sobre o código que é onde esta toda a magia dessa mecânica.

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

public class trajetoria : MonoBehaviour
{
    private float forca = 2.0f;
    private Vector2 startPos;
    [SerializeField]
    private bool tiro = false;
    [SerializeField]
    private bool mirando = false;

    [SerializeField]
    private GameObject dotsGO;
    public List<GameObject> caminho;

    [SerializeField]
    private Rigidbody2D myRBody;
    [SerializeField]
```

```

private Collider2D myCollider;

private Vector2 vel;

[SerializeField]
private float valorX, valorY;

// Use this for initialization
void Start ()
{
    dotsGO = GameObject.FindWithTag ("dots");
    myRBody.isKinematic = true;
    myCollider.enabled = false;
    startPos = transform.position;
    caminho = dotsGO.transform.Cast<Transform> () .ToList
() .ConvertAll (t => t.gameObject);
    for(int x=0; x < caminho.Count;x++)
    {
        caminho [x].GetComponent<Renderer> () .enabled = false;
    }
}

void FixedUpdate ()
{
    Vector2 wp = Camera.main.ScreenToWorldPoint
(Input.mousePosition);
    RaycastHit2D hit = Physics2D.Raycast (wp, Vector2.zero);

    if(hit.collider == null)
    {
        if (myRBody.gameObject.CompareTag ("bolaclone"))
        {
            Mirando ();
        }
    }
}

void Update ()
{
}

//Metodos

```

```

void MostraCaminho()
{
    for(int x=0; x < caminho.Count;x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = true;
    }
}

void EscondeCaminho()
{
    for(int x=0; x < caminho.Count;x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = false;
    }
}

Vector2 PegaForca(Vector3 mouse)
{
    return (new Vector2 (startPos.x + valorX ,startPos.y + valorY) - new Vector2 (mouse.x, mouse.y)) * forca;
}

Vector2 CaminhoPonto(Vector2 posInicial,Vector2 velInicial, float tempo)
{
    return posInicial + velInicial * tempo + 0.5f * Physics2D.gravity * tempo * tempo;
}

void CalculoCaminho()
{
    vel = PegaForca (Input.mousePosition) * Time.fixedDeltaTime / myRBody.mass;

    for(int x=0; x < caminho.Count; x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = true;
        float t = x / 20f;
        Vector3 point = CaminhoPonto (transform.position, vel,
t);
        point.z = 1.0f;
        caminho [x].transform.position = point;
    }
}

```

```

    }

void Mirando()
{
    if (tiro == true)
        return;

    if (Input.GetMouseButton(0) /*&&
VERIFICA_AREA_RESTRITA.restrita == false*/)
    {

        if (mirando == false) {
            mirando = true;
            startPos = Input.mousePosition;
            CalculoCaminho ();
            MostraCaminho ();
        } else {

            CalculoCaminho ();
        }

    }else if(mirando == true && tiro == false){

        myRBody.isKinematic = false;
        myCollider.enabled = true;
        tiro = true;
        mirando = false;
        myRBody.AddForce (PegaForca (Input.mousePosition) );
        EscondeCaminho ();
    }
}
}
}

```

Veja que esse código é maior e mais complexo, então vamos por partes aqui primeiro precisamos adicionar a seguinte linha de código logo no inicio.

```
using System.Linq;
```

Depois temos as variáveis que foram criadas para que o efeito funcione.

Temos a força que será usada, a posição inicial, a booleana que define o tiro que lança a bola.

A booleana que diz se estamos mirando para arremessar a bola, a variável que busca o objeto que tem as esferas azuis como suas filhas dotsGo.

Temos a lista que simboliza nosso caminho o rigidbody da bola o seu colisor assim como velocidade e valores de X e Y.

```

private float forca = 2.0f;
private Vector2 startPos;
[SerializeField]
private bool tiro = false;
[SerializeField]
private bool mirando = false;

[SerializeField]
private GameObject dotsGO;
public List<GameObject> caminho;

[SerializeField]
private Rigidbody2D myRBody;
[SerializeField]
private Collider2D myCollider;

private Vector2 vel;

[SerializeField]
private float valorX, valorY;

```

Depois disso temos os ajustes do método Start onde iniciamos buscando o objeto na cena que tem a tag dots pois é nesse objeto que temos as varias esferas azuis claro que serão usadas como trajetória. Definimos o myBody como isKinematic e desabilitamos o collider. Na startPos simplesmente passamos a posição do objeto que contem o código e depois usamos Linq para passar convertido os gameobjects do caminho. E por último temos um laço para controlar o renderer do nosso caminho.

```

void Start ()
{
    dotsGO = GameObject.FindGameObjectWithTag ("dots");
    myRBody.isKinematic = true;
    myCollider.enabled = false;
    startPos = transform.position;
    caminho = dotsGO.transform.Cast<Transform> ().ToList
    () .ConvertAll (t => t.gameObject);
    for(int x=0; x < caminho.Count;x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = false;
    }
}

```

Agora em FixedUpdate passamos para a variável wp a posição do espaço de tela no espaço de mundo seguido pelo nosso raycasthit que nos permite saber se tocamos na bola que será arremessada e que contém a tag bolaclone.

Se estivermos tocando na bola chamamos o método mirando.

```
void FixedUpdate () {
```

```

        Vector2 wp = Camera.main.ScreenToWorldPoint
(Input.mousePosition);
        RaycastHit2D hit = Physics2D.Raycast (wp, Vector2.zero);

        if(hit.collider == null)
        {
            if (myRBody.gameObject.CompareTag("bolacleone"))
            {
                Mirando();
            }
        }
    }
}

```

Continuando temos o método MostraCaminho que simplesmente habilita a nossa visualização dos dots ou esferas azul claro que vão desenhar o caminho no ar.

```

void MostraCaminho()
{
    for(int x=0; x < caminho.Count;x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = true;
    }
}

```

Agora da mesma forma que temos um método que mostra os dots do caminho precisamos de um método que esconda esses dots.

```

void EscondeCaminho()
{
    for(int x=0; x < caminho.Count;x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = false;
    }
}

```

Depois temos outro método que calcula e retorna a força que é usada mais adiante no calculo do caminho.

```

Vector2 PegaForca(Vector3 mouse)
{
    return (new Vector2 (startPos.x + valorX ,startPos.y +
valorY) - new Vector2 (mouse.x, mouse.y)) * forca;
}

```

Continuando temos o método que faz o calculo do caminho que será usado no próximo método que veremos aqui.

```
Vector2 CaminhoPonto(Vector2 posInicial, Vector2 velInicial, float tempo)
{
    return posInicial + velInicial * tempo + 0.5f *
Physics2D.gravity * tempo * tempo;
}
```

Agora esse método é quem calcula o nosso caminho nele passamos o valor do método PegaForca para a variável vel e na sequencia temos um laço do tipo for que habilita o renderer do caminho e passa para a variável point o resultado de CaminhoPonto que é passado para a posição dos elementos do caminho.

```
void CalculoCaminho()
{
    vel = PegaForca (Input.mousePosition) * Time.fixedDeltaTime /
myRBody.mass;

    for(int x=0; x < caminho.Count; x++)
    {
        caminho [x].GetComponent<Renderer> ().enabled = true;
        float t = x / 20f;
        Vector3 point = CaminhoPonto (transform.position, vel,
t);
        point.z = 1.0f;
        caminho [x].transform.position = point;
    }
}
```

Com isso só precisamos de mais um método para usar o método anterior que no caso vai ser o método Mirando.

Veja que ele inicia verificando se a variável tiro é verdadeira se for não faço nada.

Agora na sequencia já temos uma estrutura condicional que verifica se estamos apertando o botão esquerdo do mouse se estiver e se mirando for falso mudamos o valor de mirando para verdadeiro passamos para a variável startPos a posição do mouse e chamamos os métodos de CalculoCaminho e MostraCaminho.

Agora se soltamos o botão esquerdo do mouse aplicamos uma foça na bola para movimenta-la na trajetória definida e fazemos o caminho desaparecer.

```
void Mirando()
{
    if (tiro == true)
        return;

    if (Input.GetMouseButton(0) /*&&
VERIFICA_AREA_RESTRITA.restrita == false*/)
    {
```

```

if (mirando == false) {
    mirando = true;
    startPos = Input.mousePosition;
    CalculoCaminho ();
    MostraCaminho ();
} else {

    CalculoCaminho ();
}

}else if(mirando == true && tiro == false){

    myRBody.isKinematic = false;
    myCollider.enabled = true;
    tiro = true;
    mirando = false;
    myRBody.AddForce(PegaForca(Input.mousePosition));
    EscondeCaminho();
}
}

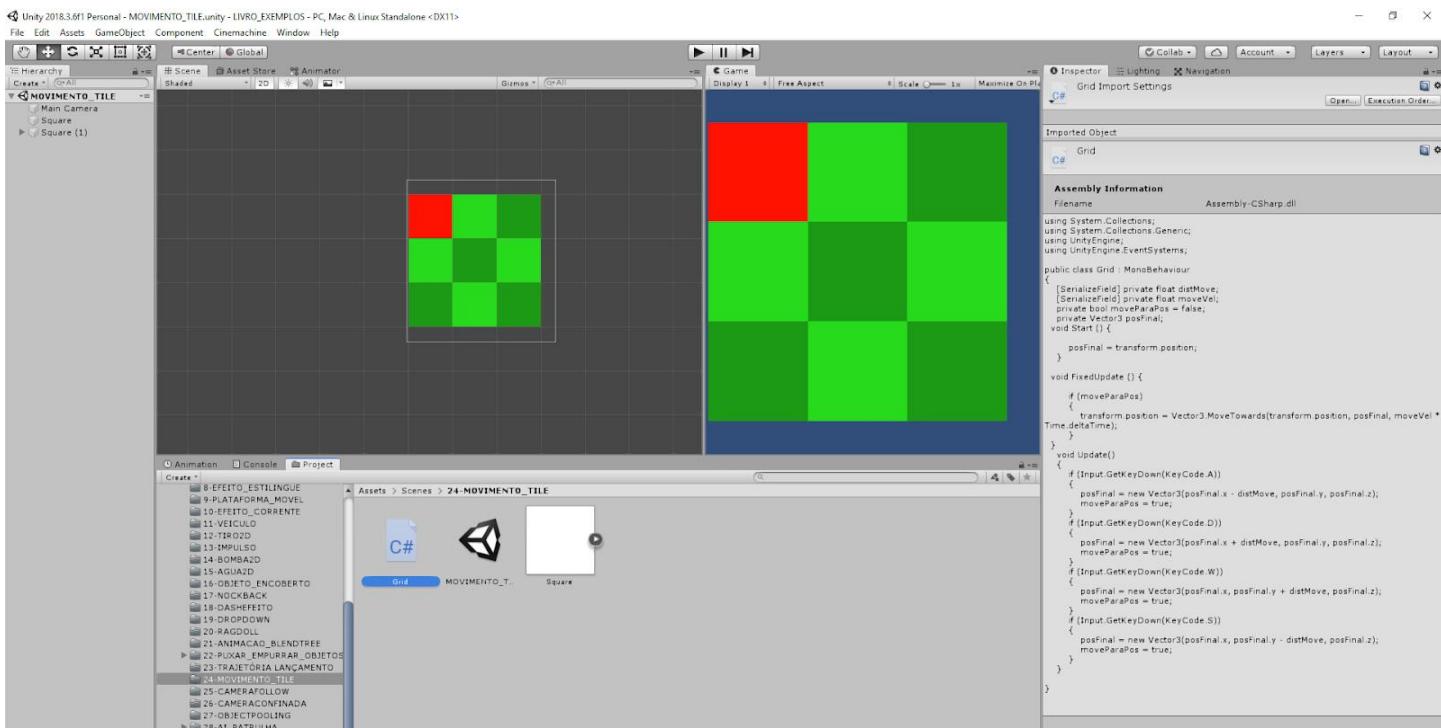
```

Com tudo isso teremos nosso exemplo de trajetória de lançamento.

MOVIMENTO TILE POR TILE

Agora vamos ver mais uma mecânica de movimento que faz um sucesso danado em jogos mobile que é o movimento tile por tile.

Para criar uma mecânica desse tipo é muito simples antes de qualquer coisa precisamos ter uma cena semelhante a da imagem abaixo:



Veja que usamos apenas quadrados nessa cena sendo que nove desses quadrados simbolizam um chão com definição de tiles claros e escuros e um quadrado vermelho que simboliza o personagem

Dentro do quadrado vermelho temos o código **Grid**, veja:

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

public class Grid : MonoBehaviour
{
    [SerializeField] private float distMove;
    [SerializeField] private float moveVel;
    private bool moveParaPos = false;
    private Vector3 posFinal;

    void Start () {
        posFinal = transform.position;
    }

    void Update () {
        if (Input.GetKeyDown(KeyCode.A))
            posFinal = new Vector3(posFinal.x - distMove, posFinal.y, posFinal.z);
        moveParaPos = true;
    }
    if (Input.GetKeyDown(KeyCode.D))
        posFinal = new Vector3(posFinal.x + distMove, posFinal.y, posFinal.z);
        moveParaPos = true;
    }
    if (Input.GetKeyDown(KeyCode.W))
        posFinal = new Vector3(posFinal.x, posFinal.y + distMove, posFinal.z);
        moveParaPos = true;
    }
    if (Input.GetKeyDown(KeyCode.S))
        posFinal = new Vector3(posFinal.x, posFinal.y - distMove, posFinal.z);
        moveParaPos = true;
    }
}
```

```

        }

    void FixedUpdate () {

        if (moveParaPos)
        {
            transform.position =
Vector3.MoveTowards(transform.position, posFinal, moveVel *
Time.deltaTime);
        }
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.A))
        {
            posFinal = new Vector3(posFinal.x - distMove, posFinal.y,
posFinal.z);
            moveParaPos = true;
        }
        if (Input.GetKeyDown(KeyCode.D))
        {
            posFinal = new Vector3(posFinal.x + distMove, posFinal.y,
posFinal.z);
            moveParaPos = true;
        }
        if (Input.GetKeyDown(KeyCode.W))
        {
            posFinal = new Vector3(posFinal.x, posFinal.y + distMove,
posFinal.z);
            moveParaPos = true;
        }
        if (Input.GetKeyDown(KeyCode.S))
        {
            posFinal = new Vector3(posFinal.x, posFinal.y - distMove,
posFinal.z);
            moveParaPos = true;
        }
    }
}

```

Aqui iniciamos criando algumas variáveis uma para distância de movimento, outra para a velocidade do movimento, uma booleana para sinalizar o movimento e por último uma variável com a posição final.

```

[SerializeField] private float distMove;
[SerializeField] private float moveVel;
private bool moveParaPos = false;
private Vector3 posFinal;

```

Depois disso dentro do método Start temos a atribuição do valor da posição do objeto que contem esse código para a variável posFinal.

```
void Start () {  
    posFinal = transform.position;  
}
```

Seguindo mais adiante temos o método FixedUpdate que verifica se a variável moveParaPos é verdadeiro se for ajustamos a posição do objeto usando a Vector3.MoveTowards que necessita de uma posição inicial, um alvo e um delta para mover o personagem.

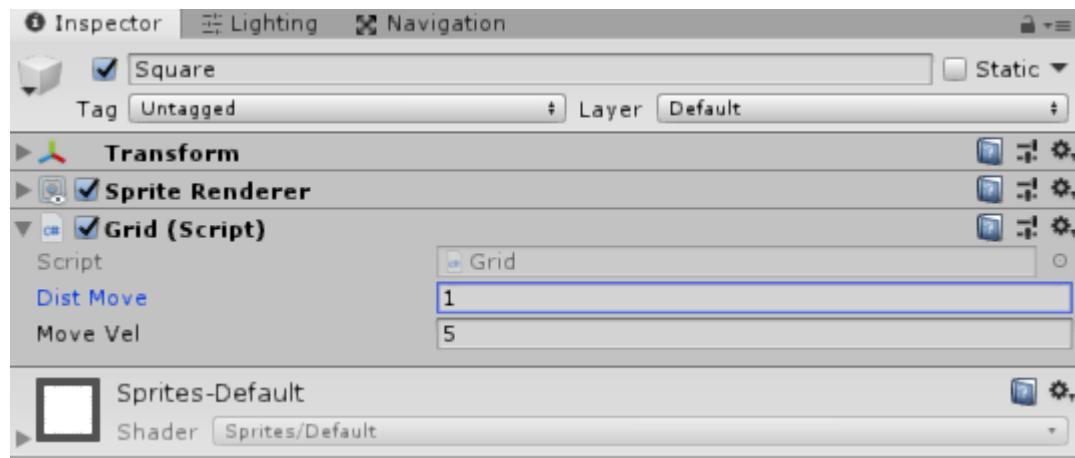
```
void FixedUpdate () {  
  
    if (moveParaPos)  
    {  
        transform.position =  
Vector3.MoveTowards(transform.position, posFinal, moveVel *  
Time.deltaTime);  
    }  
}
```

Por último temos o método Update que tem a estrutura condicional que verifica quais teclas estamos apertando e de acordo com cada tecla ajusta a posFinal para um novo cálculo de movimento.

```
void Update()  
{  
    if (Input.GetKeyDown(KeyCode.A))  
    {  
        posFinal = new Vector3(posFinal.x - distMove, posFinal.y,  
posFinal.z);  
        moveParaPos = true;  
    }  
    if (Input.GetKeyDown(KeyCode.D))  
    {  
        posFinal = new Vector3(posFinal.x + distMove, posFinal.y,  
posFinal.z);  
        moveParaPos = true;  
    }  
    if (Input.GetKeyDown(KeyCode.W))  
    {  
        posFinal = new Vector3(posFinal.x, posFinal.y + distMove,  
posFinal.z);  
        moveParaPos = true;  
    }  
    if (Input.GetKeyDown(KeyCode.S))  
    {  
        posFinal = new Vector3(posFinal.x, posFinal.y - distMove,  
posFinal.z);  
        moveParaPos = true;  
    }  
}
```

```
    }  
}
```

Feito isso basta adicionar esse código no quadrado vermelho e depois ajustar os valores de Dist Move e Move vel como mostra a imagem abaixo para definir se o objeto anda de um em um quadrado e em qual velocidade.

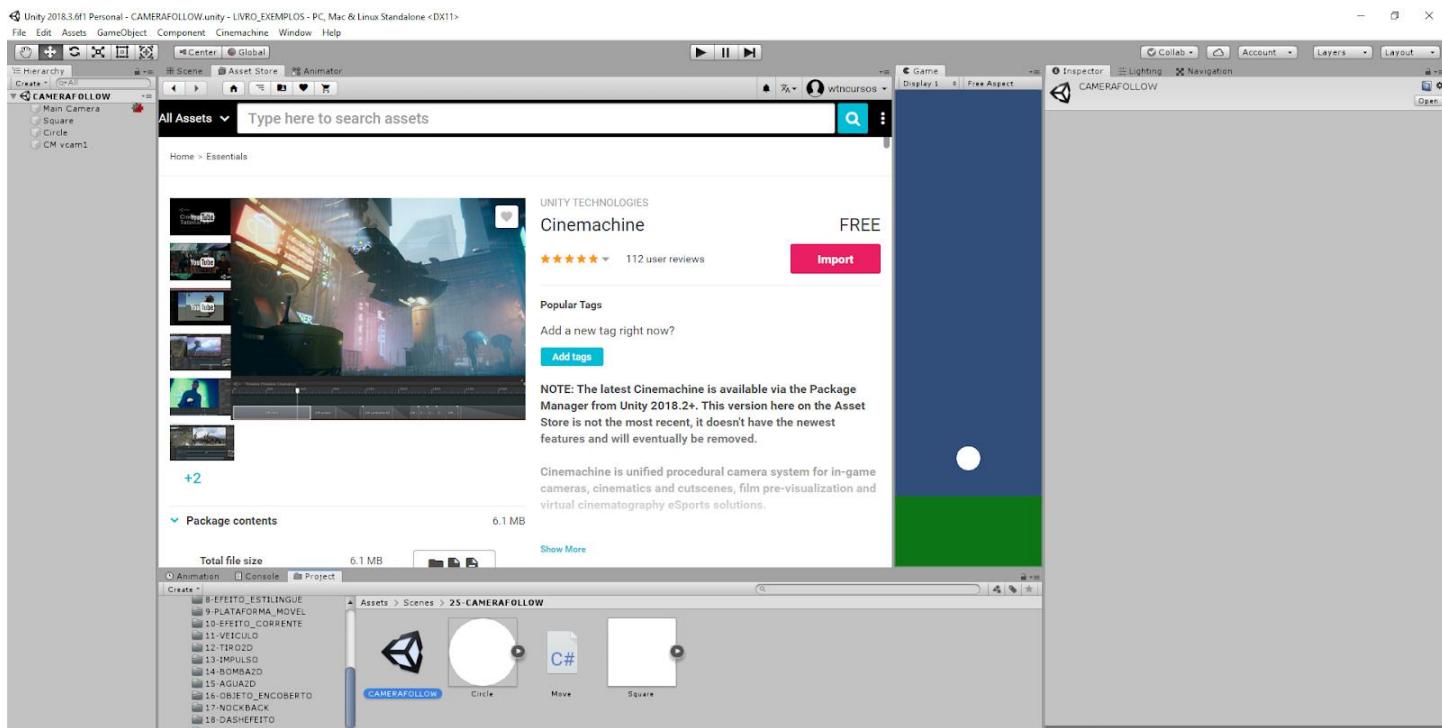


CAMERA FOLLOW

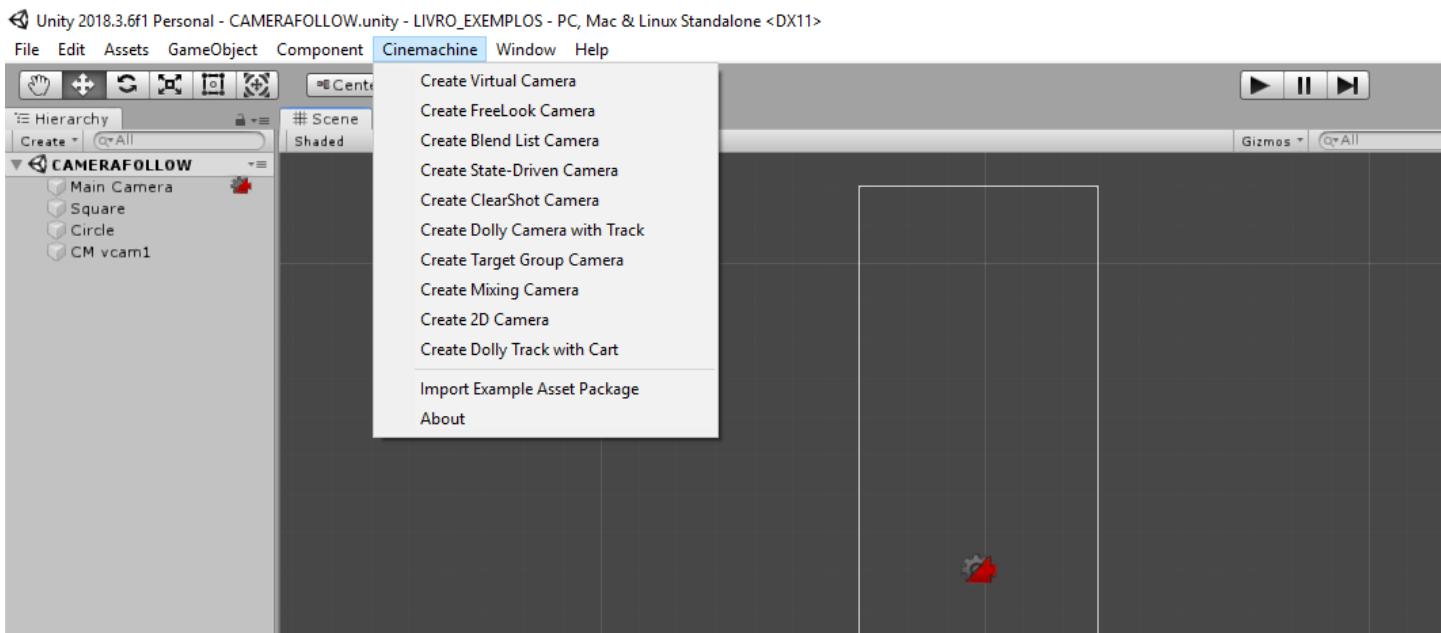
Agora vamos ver uma mecânica que é mais do que conhecida no mundo dos games, mas que nem sempre é valorizada.

Estou falando da câmera perseguidora, essa câmera precisa ser muito bem feita para proporcionar a melhor experiência possível para o jogador então o ideal é usar um cara chamado Cinemachine veja.

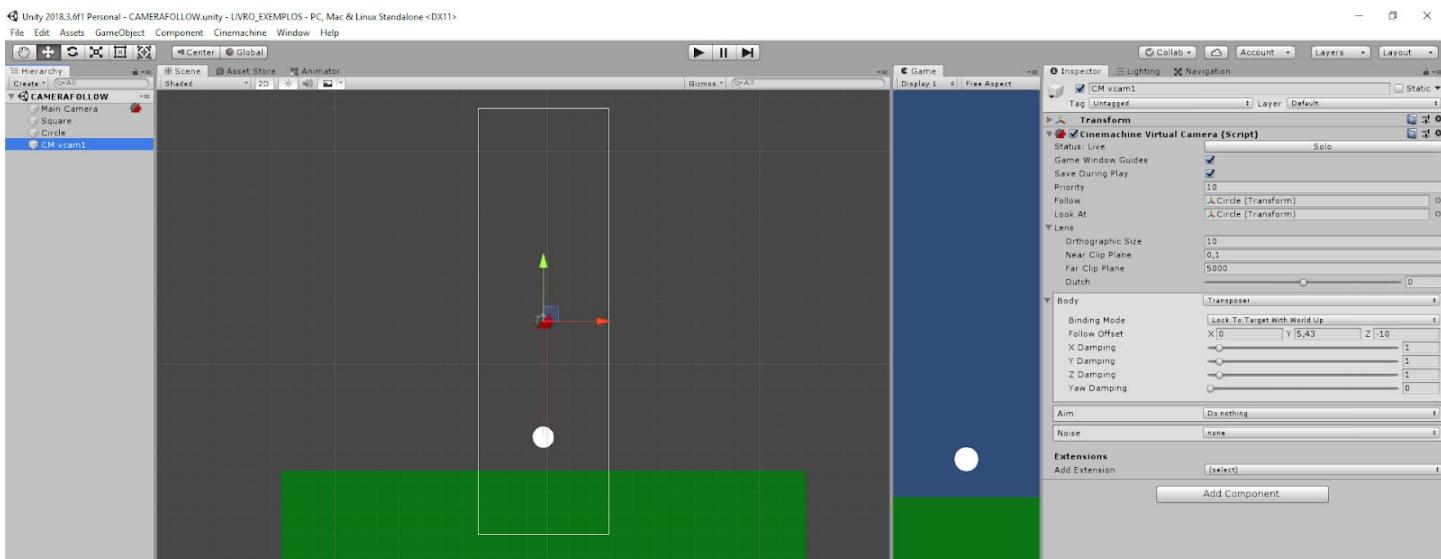
Você pode baixar o cinemachine de graça na Asset Store veja:



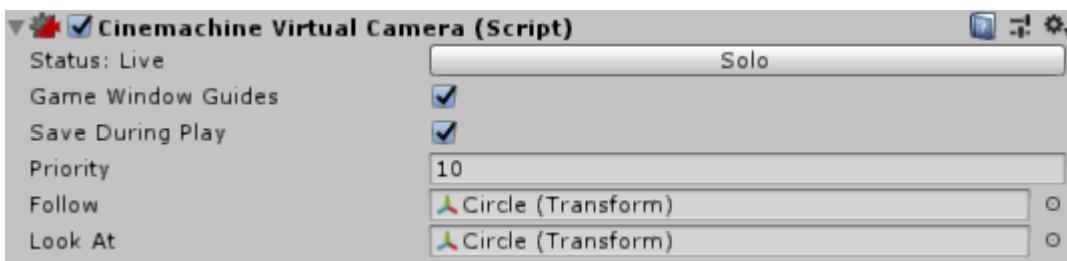
E depois de baixar o cinemachine você vai ter a opção de criar varios tipos de câmera veja:



No exemplo que vou mostrar eu escolhi a opção Create 2D Câmera o que me proporcionou o seguinte:



Veja que na imagem acima tenho varias configurações para essa câmera mas certamente a mais importante é a que me permite fazer a câmera olhar e seguir qual objeto de cena eu quiser.



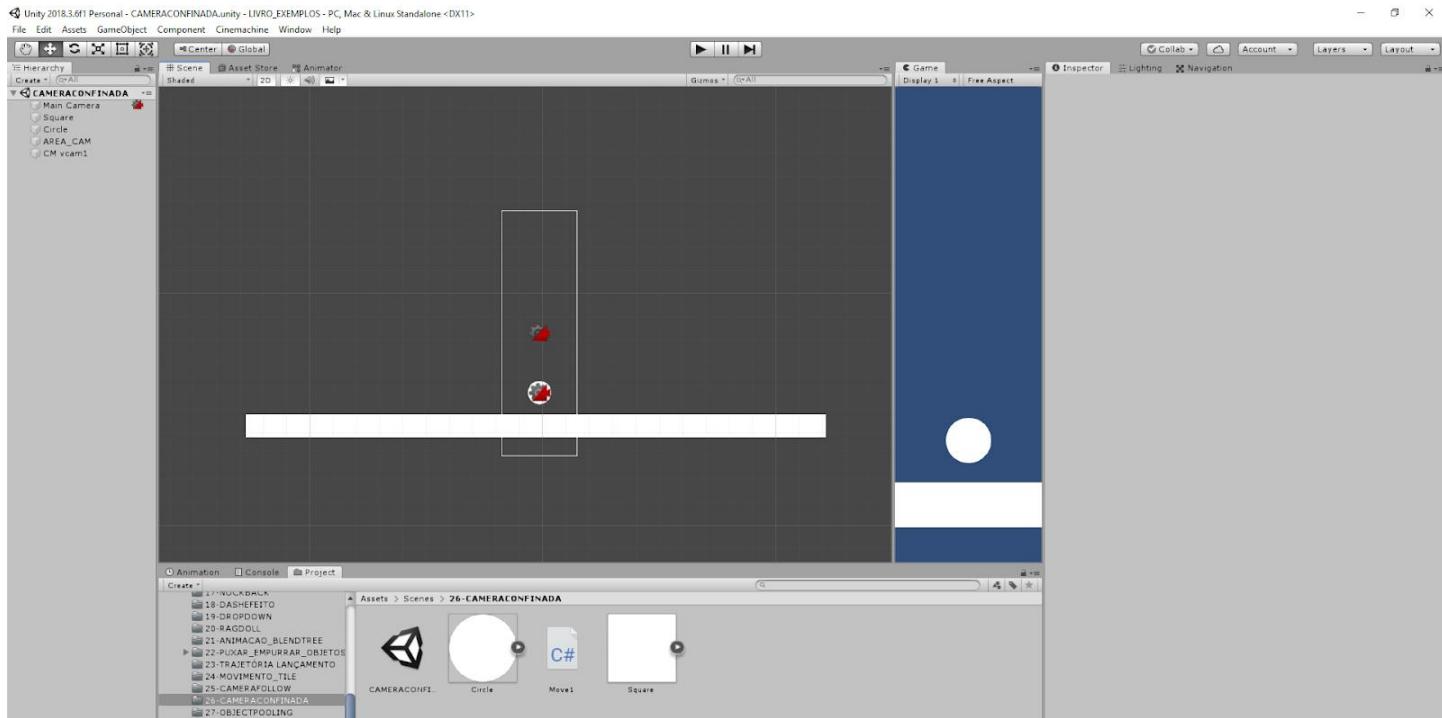
Isso sem dúvida é uma baita mão na roda.

Feito isso basta colocar um personagem que se movimente na cena e você vai ver que a câmera vai perseguir esse objeto de forma suave.

CAMERA CONFINADA

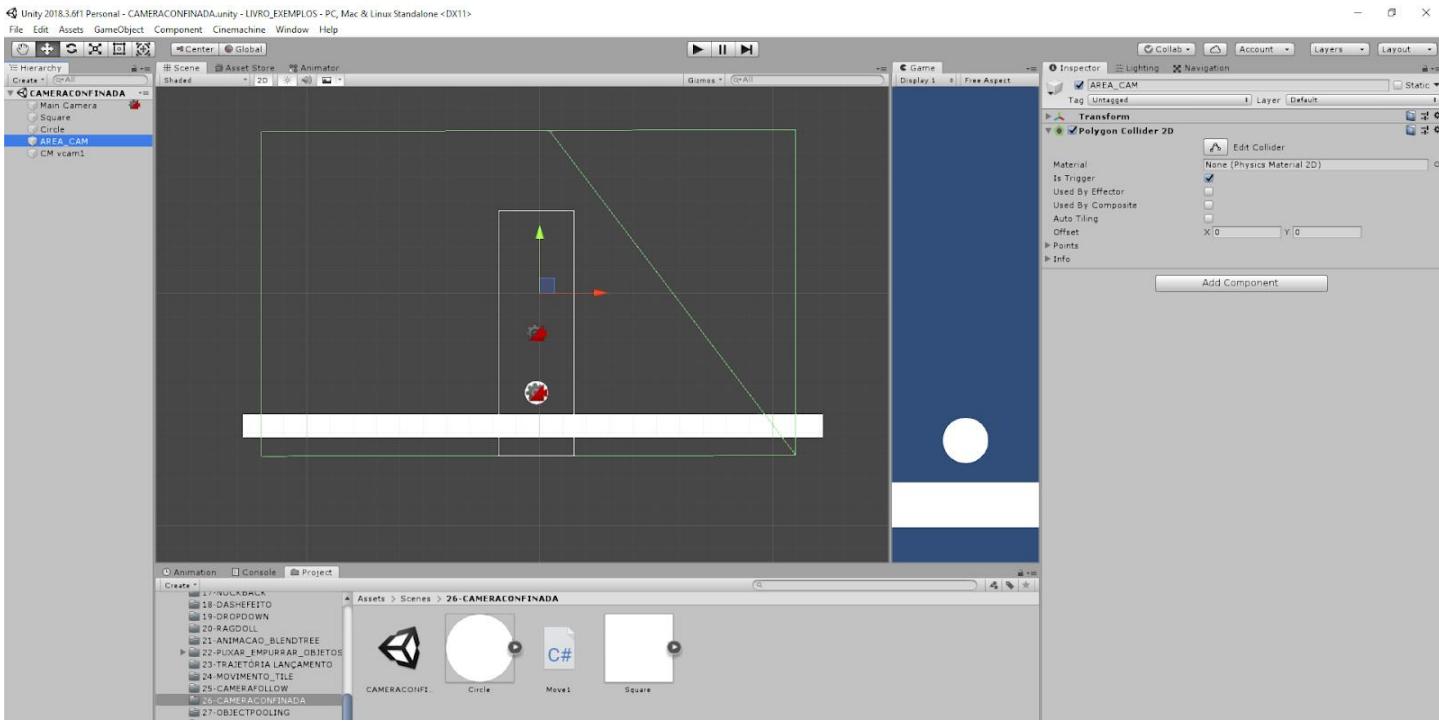
Agora que já vimos como fazer uma câmera perseguidora de forma mais do que fácil, vamos aprender a confinar essa câmera dentro de uma área restrita evitando que a mesma mostre o que não deve ser mostrado para ninguém.

Para esse exemplo eu tenho uma cena bem simples veja:



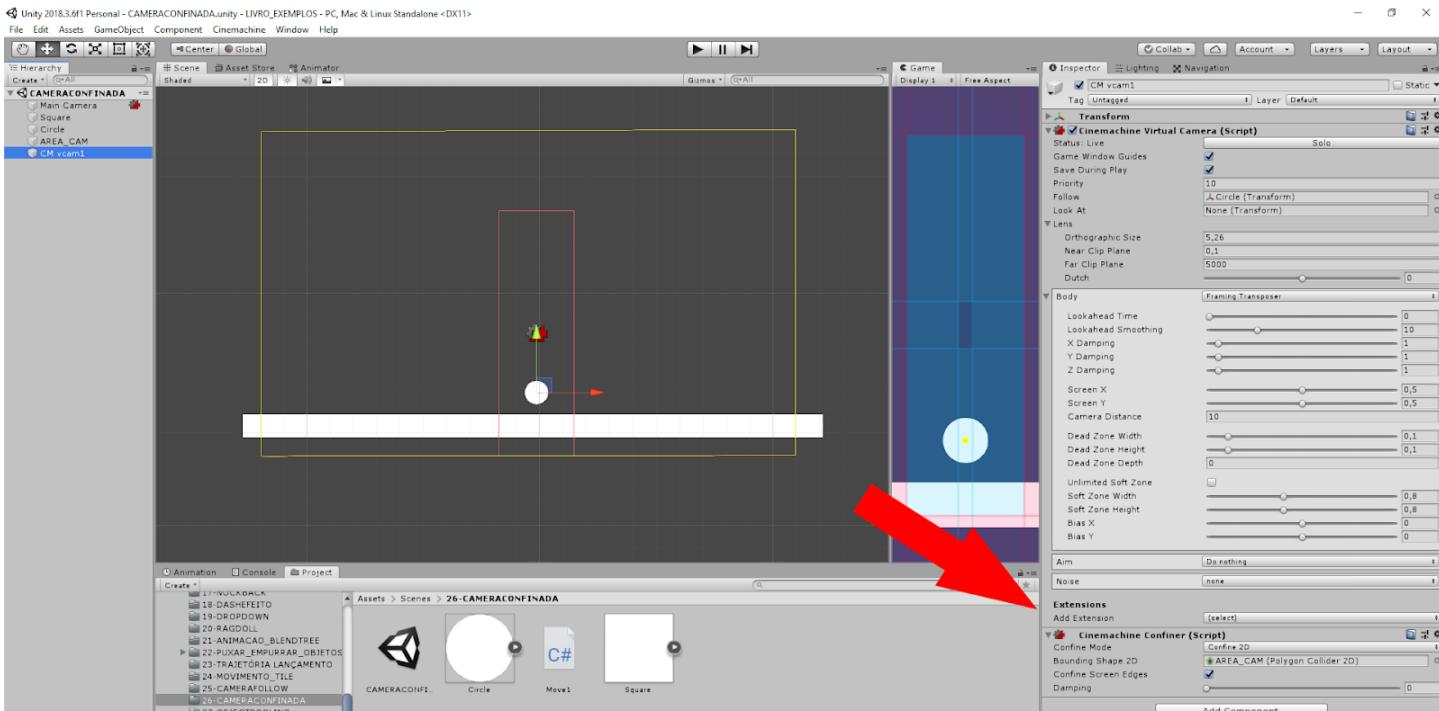
Nem preciso falar dessa cena já que aqui não existe nada que não tenhamos visto anteriormente. Temos um chão um objeto que se move de um lado para o outro uma câmera perseguidora e agora precisamos confinar essa câmera.

Para isso vamos criar um GameObject vazio e adicionar dentro dele um Polygon collider dessa forma:



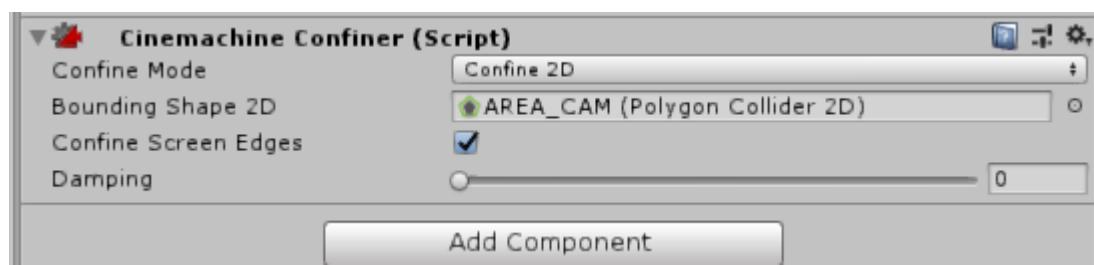
Veja que temos o corpo do polygon collider definido com as linhas verdes isso significa que a câmera não pode sair de dentro desse espaço.

Mas para que a câmera entenda isso precisamos selecionar o objeto com o nome de VCM vcam1 e fazer um pequeno ajuste em suas configurações.



Veja que adicionei a ela uma extensão com o nome Cinemachine Confiner e passei para ela o collider que representa a área que a câmera precisa respeitar.

Feito isso posso mover o personagem que a câmera não vai furar o bloqueio que foi criado.

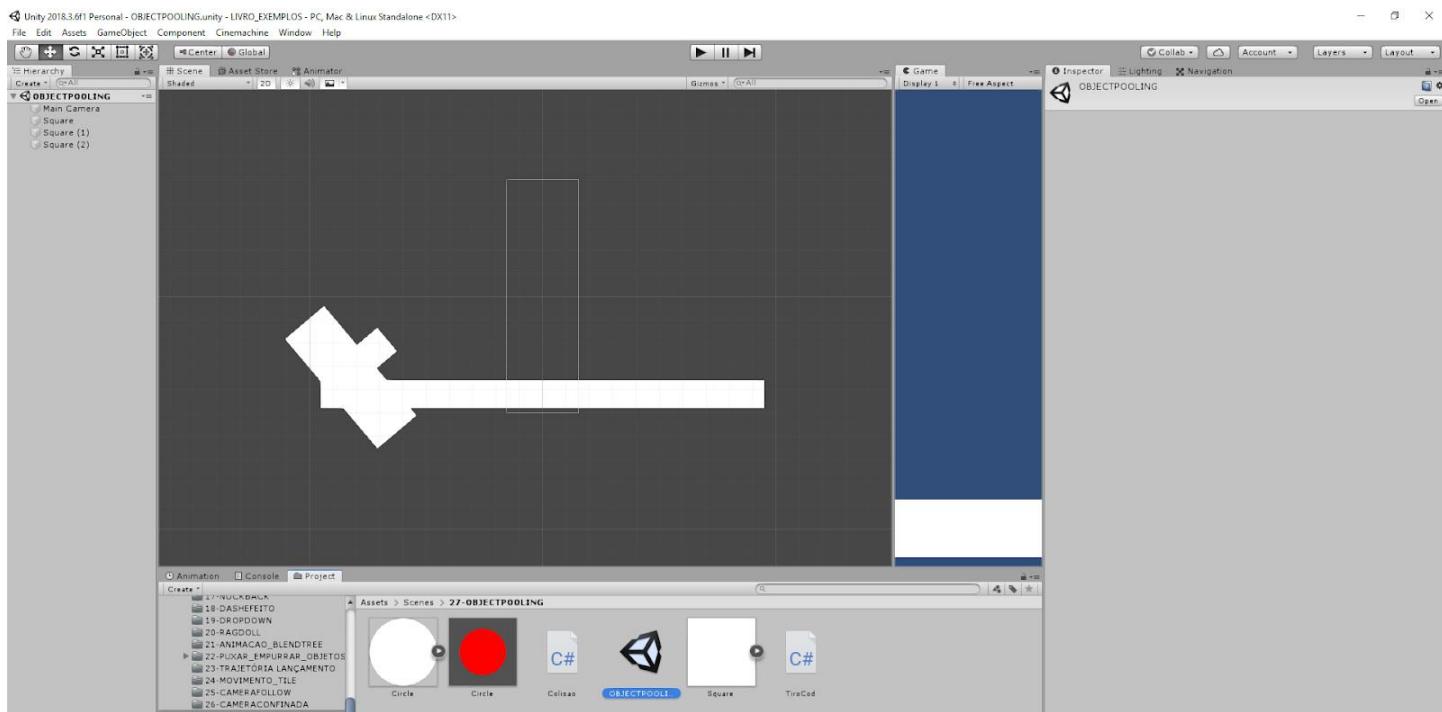


OBJECT POOLING

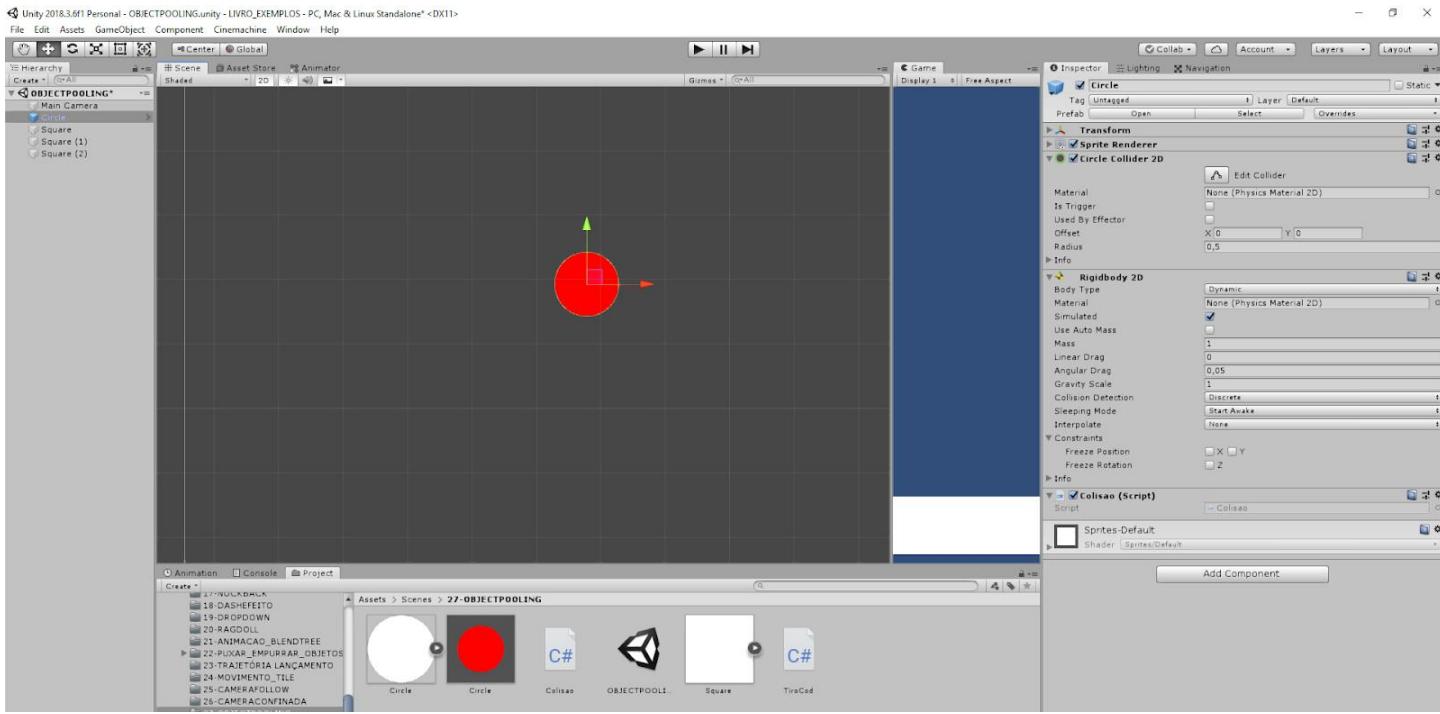
Agora vamos falar de um assunto muito importante quando se fala de múltiplos objetos em cena, no caso estou falando de object pooling.

Com esse camarada conseguimos adicionar e retirar de cena vários objetos sem sobrecarregar demais o nosso jogo.

Para mostrar esse exemplo é necessário uma cena parecida com a da imagem abaixo:



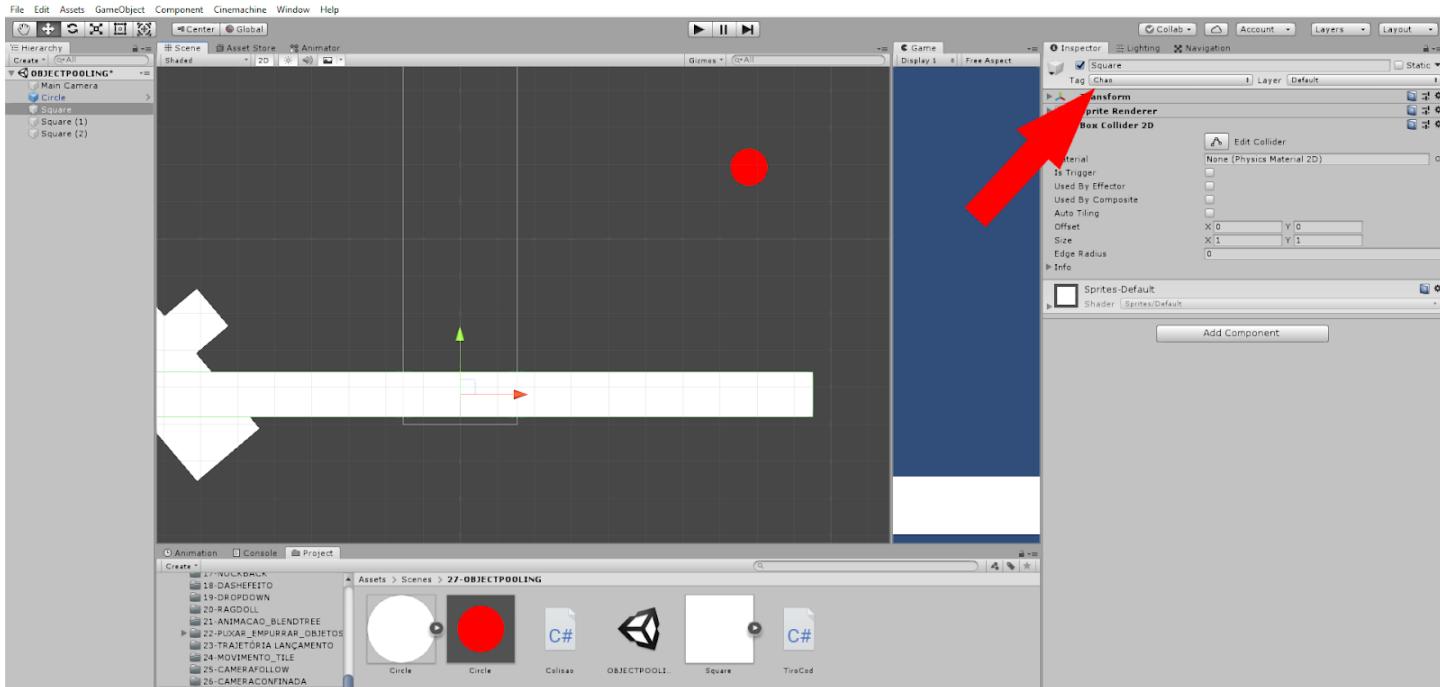
Depois disso precisamos criar um objeto esférico que vai servir de bala que será disparada sempre que eu determinar.



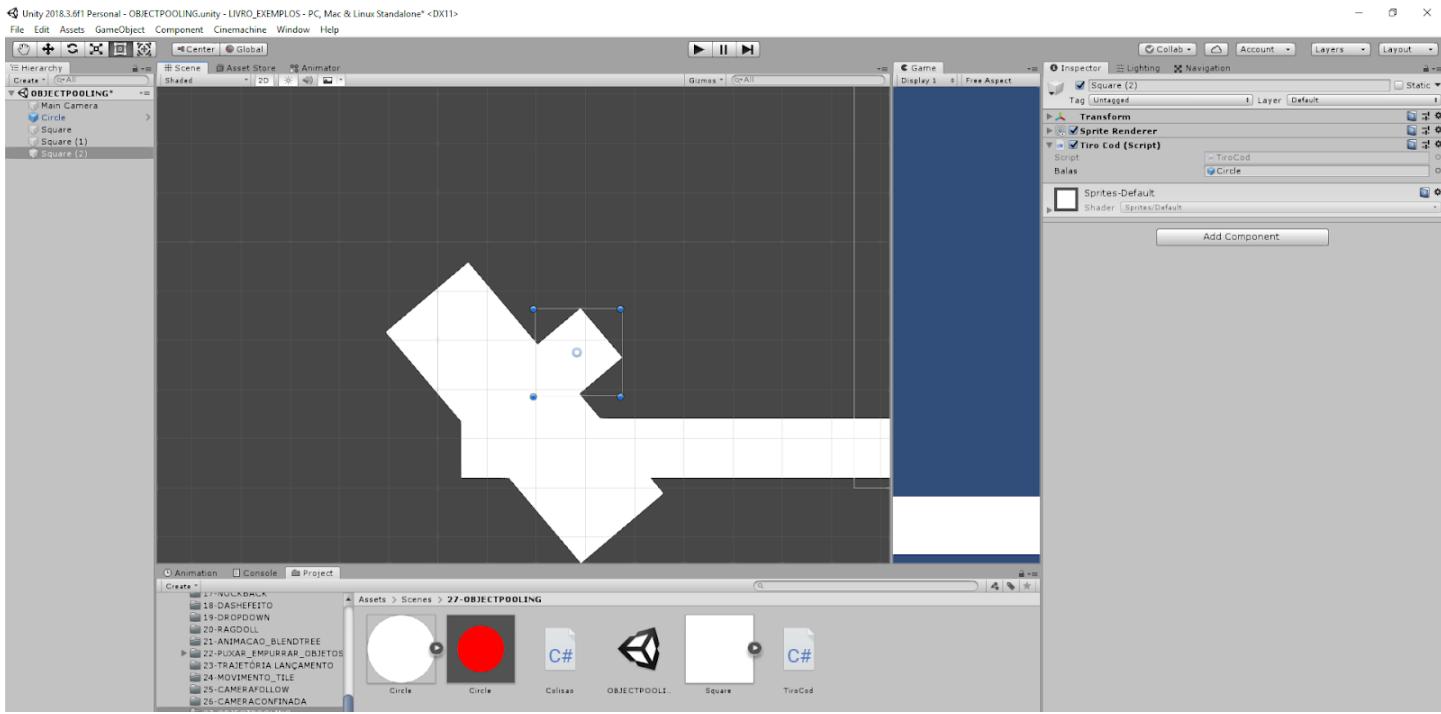
Veja que esse objeto tem um corpo colisor um rigidbody2D e um código com o nome de Colisao. Mas não vamos pensar nesse código agora.

continuando repare mais uma coisa o nosso chão tem uma tag que é a tag Chao isso é importante.

Unity 2018.3.6f1 Personal - OBJECTPOOLING.unity - LIVRO_EXEMPLOS - PC, Mac & Linux Standalone* <DX11>



Veja na imagem abaixo que o objeto quadrado com a aparência de um cano de canhão tem um código chamado de Tiro e é por esse código que vamos começar.



Vamos analisar o código de Tiro.

```

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

public class TiroCod : MonoBehaviour
{

    private float vel = 500;
    public GameObject balas;
    List<GameObject> listaBalas;

    void Start()
    {
        listaBalas = new List<GameObject>();
        for (int i = 0; i < 10; i++)
        {
            GameObject objBala = (GameObject)Instantiate(balas);
            objBala.SetActive(false);
            listaBalas.Add(objBala);
        }
    }

    void Fire()
    {

        for (int i = 0; i < listaBalas.Count; i++)
    }
}

```

```

        if (!listaBalas[i].activeInHierarchy)
        {
            listaBalas[i].transform.position = transform.position;
            listaBalas[i].transform.rotation = transform.rotation;
            listaBalas[i].SetActive(true);
            Rigidbody2D tempRigidBodyBullet =
listaBalas[i].GetComponent<Rigidbody2D>();
            tempRigidBodyBullet.AddForce (tempRigidBodyBullet.trans
form.up * vel);
            break;
        }
    }
}

// Update is called once per frame
void Update()
{
    if (Input.GetMouseButtonDown (0))
    {
        Fire();
    }
}
}

```

Para esse código começamos criando algumas variáveis como uma float de velocidade, uma do tipo GameObject que representa as balas e uma lista das balas.

```

private float vel = 500;
public GameObject balas;
List<GameObject> listaBalas;

```

Depois disso dentro do método Start vamos preencher nossa lista com os game objects da bala dessa forma teremos uma lista de objetos que podemos ativar e desativar em cena.

```

void Start()
{
    listaBalas = new List<GameObject>();
    for (int i = 0; i < 10; i++)
    {
        GameObject objBala = (GameObject) Instantiate(balas);
        objBala.SetActive(false);
        listaBalas.Add(objBala);
    }
}

```

Mais adiante no método Fire temos a magica acontecendo, veja que aqui temos um laço que percorre nossa lista de balas e dentro dele verificamos se um elemento x não esta ativo na cena.

se essa condição for satisfeita ajustamos a posição do elemento da lista para ter a posição do objeto que tem o código, depois ajustamos sua rotação para a rotação do objeto que contem esse código e ativamos o elemento.

Depois disso pegamos o rigidbody2d do elemento corrente e adicionamos uma força a ele.

```
void Fire()
{
    for (int i = 0; i < listaBalas.Count; i++)
    {
        if (!listaBalas[i].activeInHierarchy)
        {
            listaBalas[i].transform.position = transform.position;
            listaBalas[i].transform.rotation = transform.rotation;
            listaBalas[i].SetActive(true);
            Rigidbody2D tempRigidBodyBullet =
listaBalas[i].GetComponent<Rigidbody2D>();
            tempRigidBodyBullet.AddForce(tempRigidBodyBullet.trans
form.up * vel);
            break;
        }
    }
}
```

Agora só precisamos definir um gatilho para fazer com que as balas sejam ativadas em cena e esse gatilho esta no método Update.

```
void Update()
{
    if (Input.GetMouseButtonDown(0))
    {
        Fire();
    }
}
```

Veja que nesse método apenas verificamos se apertamos o botão esquerdo do mouse se tivermos apertado chamamos o método Fire.

Prontinho agora só precisamos entender o código de colisão que é esse abaixo:

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

public class Colisao : MonoBehaviour
{
    private void OnEnable()
    {
        transform.GetComponent<Rigidbody2D>().WakeUp();
        Invoke("hideBullet", 2.0f);
    }
}
```

```
}

void hideBullet()
{
    gameObject.SetActive(false);
}

private void OnDisable()
{
    transform.GetComponent<Rigidbody2D>().Sleep();
    CancelInvoke();
}

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.transform.tag == "Chao")
    {
        gameObject.SetActive(false);
    }
}
```

Nesse código a primeira coisa que fizemos foi definir que quando a bala que contem esse código estiver ativa na cena forçamos que ela acorde e invocamos o método `hideBullet` para garantir que depois de um tempo a bala vai ser desativada.

```
private void OnEnable()
{
    transform.GetComponent<Rigidbody2D>().WakeUp();
    Invoke("hideBullet", 2.0f);
}
```

E a ação de desativar a bala é feita aqui:

```
void hideBullet()
{
    gameObject.SetActive(false);
}
```

Bem simples apenas definimos SetActive como false.

Continuando vamos para o próximo método que vai agir quando o objeto estiver desativado nesse caso definimos o rigidbody como sleep e cancelamos o invoke.

```
private void OnDisable()
{
    transform.GetComponent<Rigidbody2D>().Sleep();
    CancelInvoke();
}
```

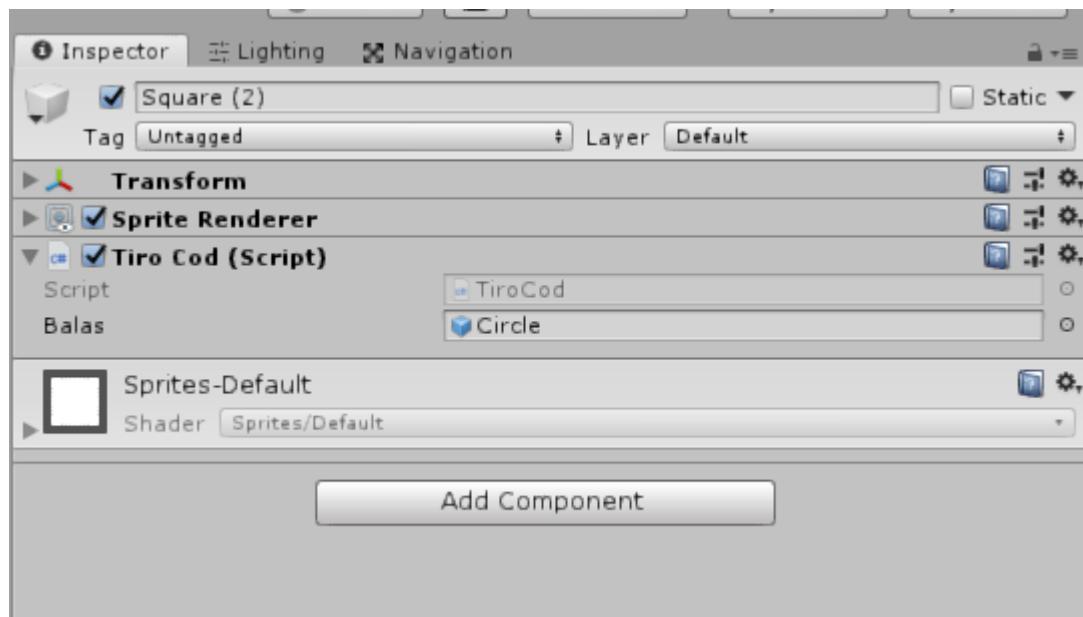
E para finalizar temos o método de verificação de colisão que checa se a bala colidiu com um objeto com a tag Chao, se essa colisão acontecer desativamos a bala.

```

private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.transform.tag == "Chao")
    {
        gameObject.SetActive(false);
    }
}

```

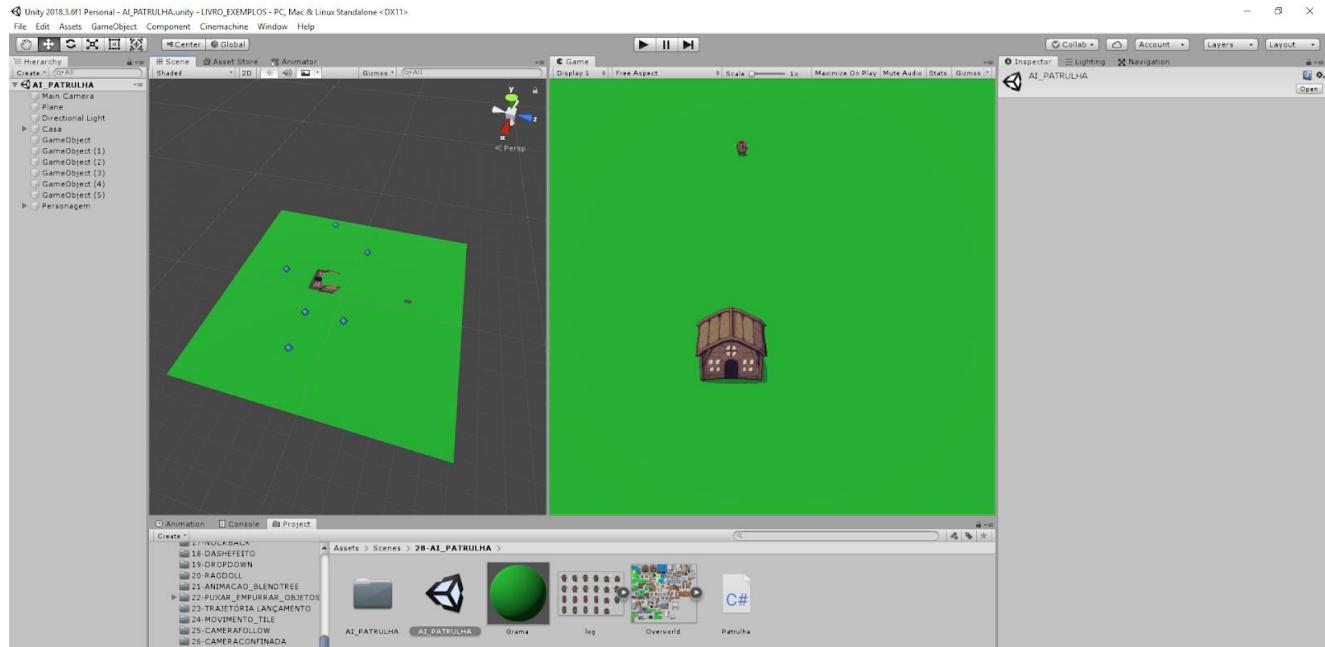
Prontinho agora só precisamos fazer um pequeno ajuste no código do de tiro para que o mesmo saiba quem é o prefab da bala, veja:



Feito isso é só executar e ver tudo funcionando.

AI PATRULHA

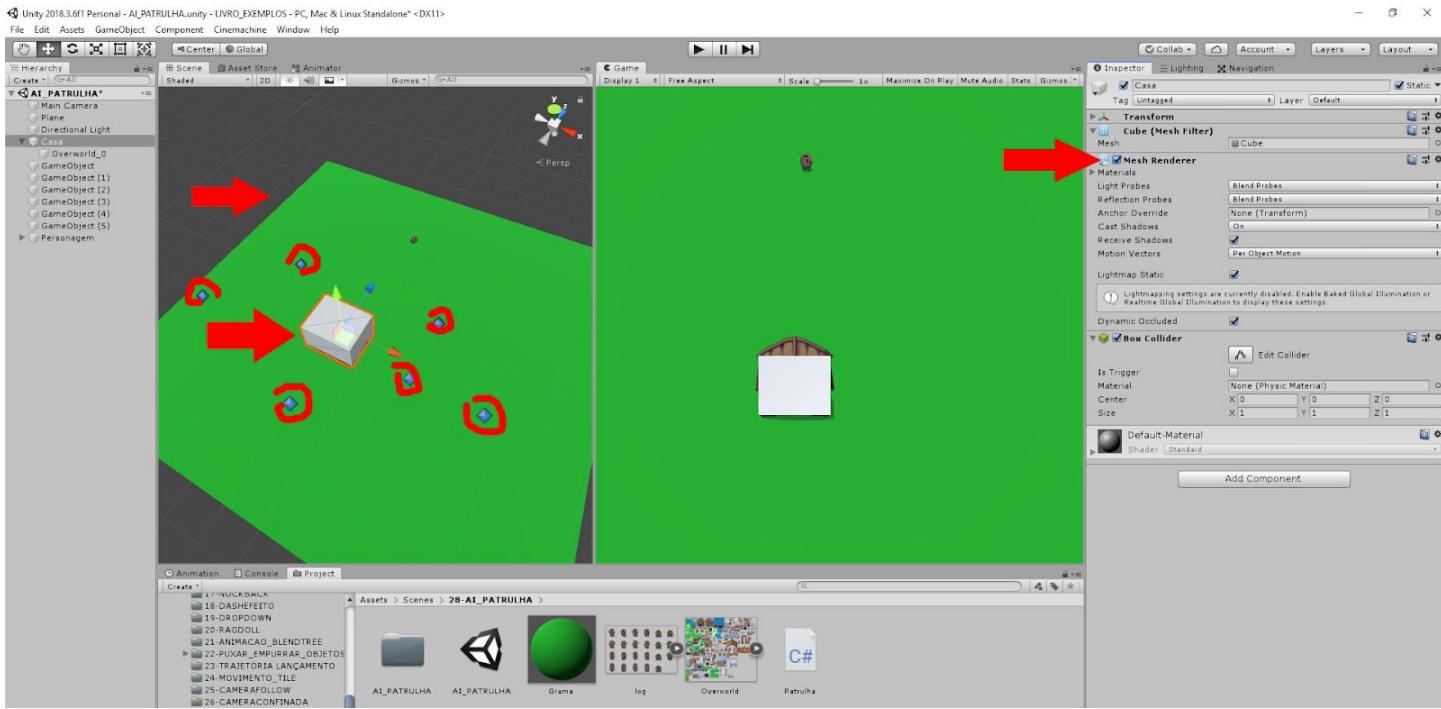
Outra mecânica muito importante dentro de jogos é a patrulha, criar um caminho e fazer com que um personagem percorra esse caminho de forma inteligente sem bater em nenhum obstáculo. Para fazer isso vamos criar uma cena semelhante a que mostra a imagem abaixo:



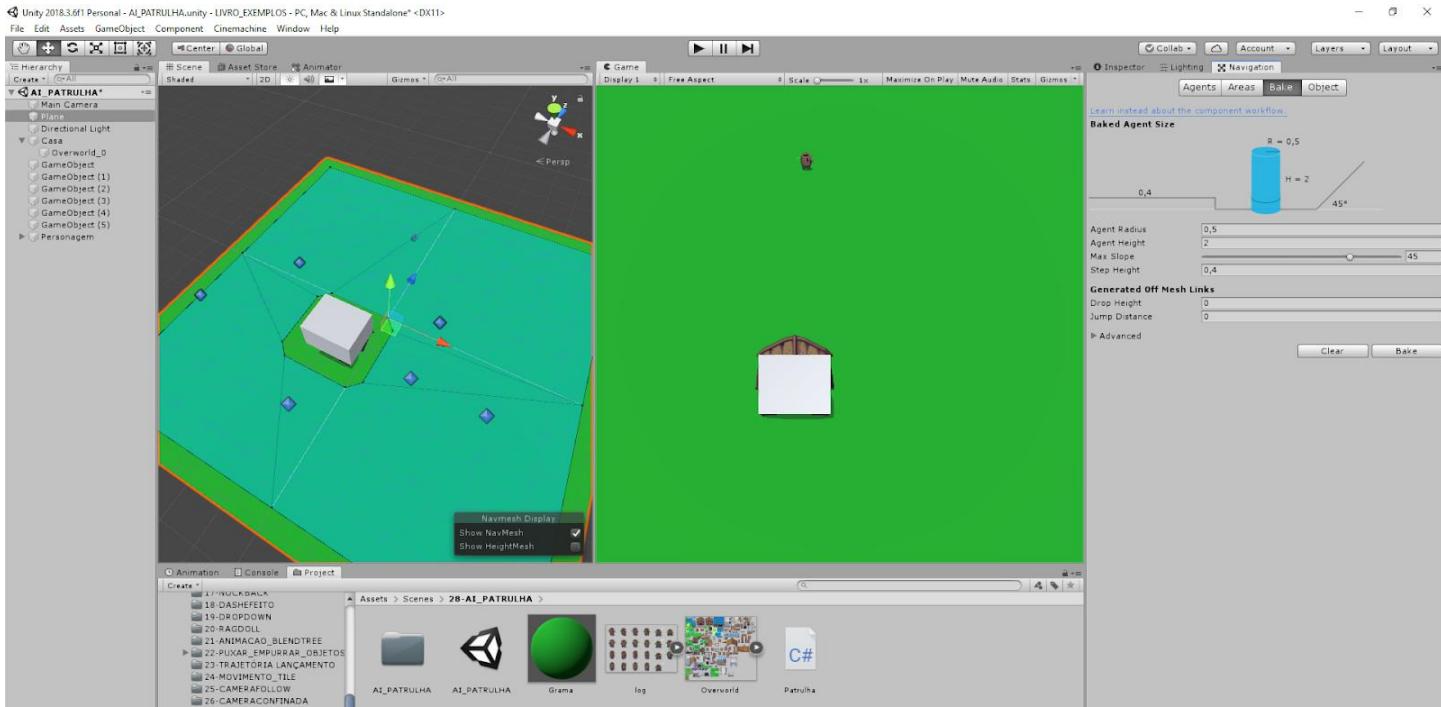
Veja que nessa cena temos dois objetos 3D que são um cubo e uma plane, veja que temos flechas apontando para eles na janela scene.

O cubo precisa ter seu Mesh Renderer desligado para não apareça na cena.

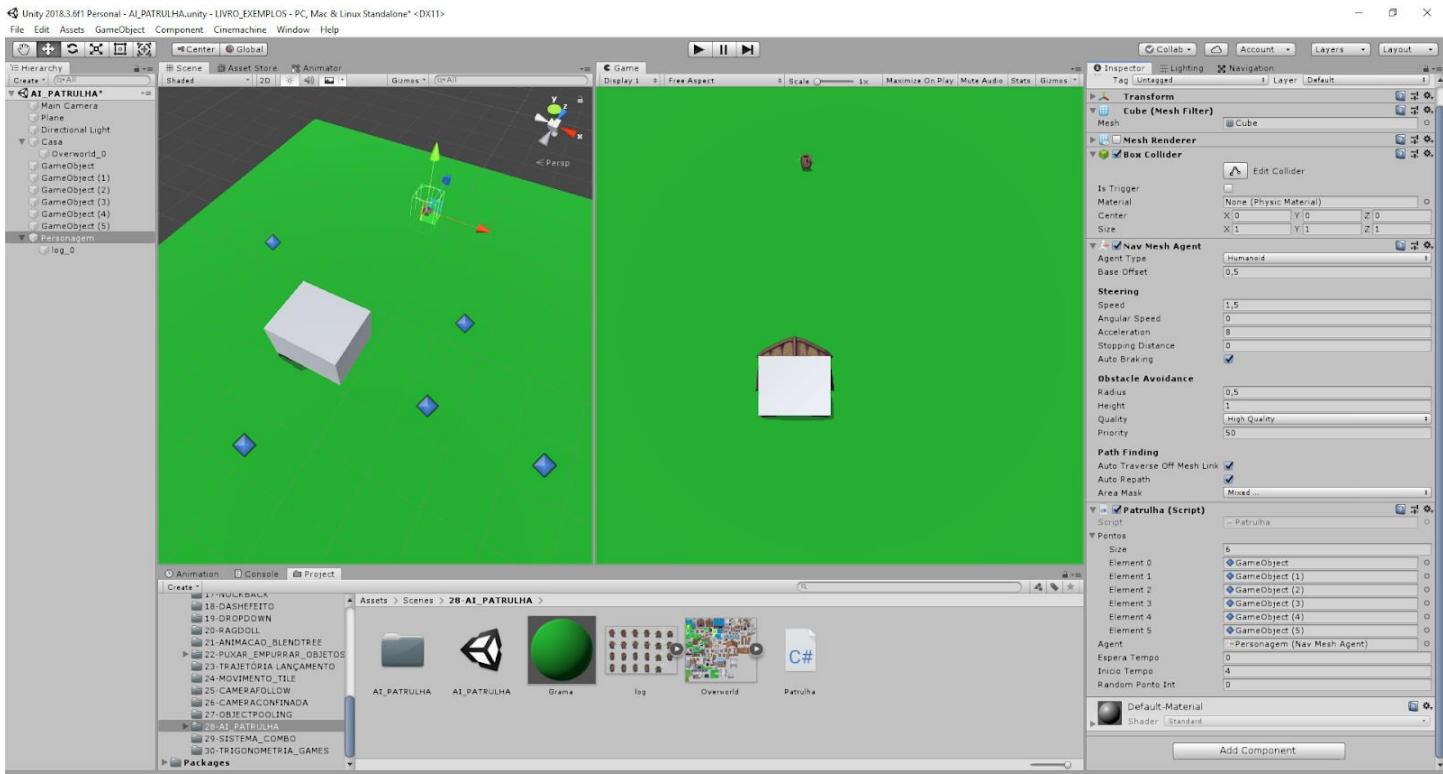
Repare também que temos GameObjects vazios que estão circulados na cena esses caras representam a rota de patrulha do personagem.



Para deixar essa rota mais inteligente definimos o chão e o cubo que representa a casa como sendo estáticos e depois em Navigation demos um Bake para criar a rota inteligente de movimento.



Agora adicionamos um NavMeshAgent ao personagem que vai patrulhar a cena porem, veja que ele também tem outros componentes como corpo físico e um arquivo de código chamado **Patrulha**.



Feito isso vamos ver o código desse exemplo.

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

public class Patrulha : MonoBehaviour
{

    public GameObject[] pontos;
    public NavMeshAgent agent;
    public float esperaTempo;
    public float inicioTempo;
    public int randomPontoInt
    ;

    // Start is called before the first frame update
    void Start()
    {
        esperaTempo = inicioTempo;
        randomPontoInt = Random.Range(0, pontos.Length);
    }

    // Update is called once per frame
    void Update()
    {
```

Veja que iniciamos esse código criando algumas variáveis, a primeira é um array que contém os objetos que representam a rota que deve ser seguida.

Depois temos o NavMeshAgent que para ser usado necessida da seguinte linha de código logo no inicio de tudo “`using UnityEngine.AI;`”

Depois temos duas variáveis float uma que é o tempo de espera e a outra o inicioTempo. Por último temos uma variável do tipo int que é o random entre os pontos de patrulha.

```
public GameObject[] pontos;  
public NavMeshAgent agent;  
public float esperaTempo;  
public float inicioTempo;  
public int randomPontoInt;
```

Agora dentro do metodo Start temos a atribuição dada a variavel esperaTempo que no caso é inicioTempo, depois criamos um random entre 0 e o valor do array de pontos de patrulha.

```
void Start()
```

```

    }

    esperaTempo = inicioTempo;
    randomPontoInt = Random.Range(0, pontos.Length);

}

}

```

Dentro do método Update temos apenas a chamada do método CheckDestinationReached que veremos mais adiante.

```

void Update()
{
    CheckDestinationReached();
}

```

E agora temos o método que faz a patrulha acontecer nele passamos o destino do personagem baseado no random que foi feito dessa forma o personagem anda pela cena e em cada ponto para por alguns instantes antes de continuar .

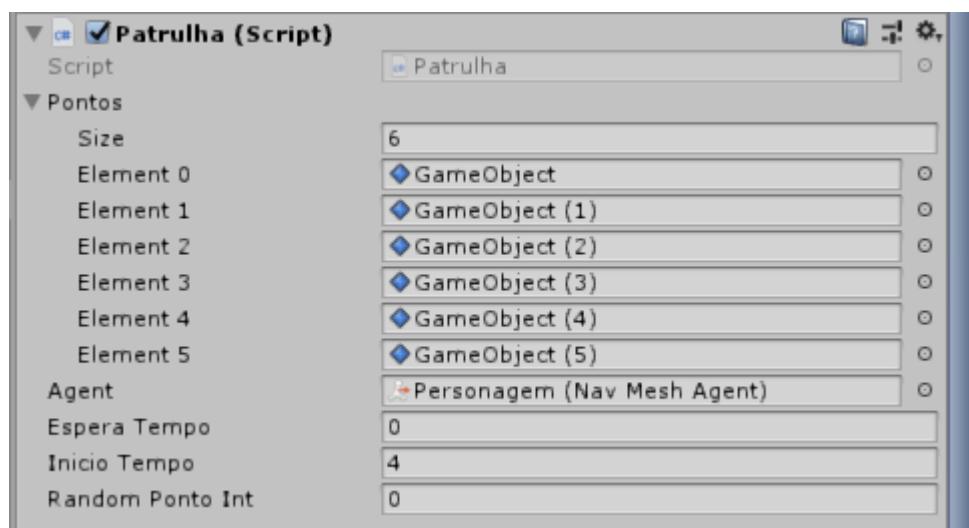
```

void CheckDestinationReached()
{

    agent.SetDestination(pontos[randomPontoInt].transform.position);
    float distanceToTarget =
Vector3.Distance(transform.position,
pontos[randomPontoInt].transform.position);
    if(distanceToTarget < 0.5f )
    {
        if(esperaTempo <= 0)
        {
            randomPontoInt = Random.Range(0, pontos.Length);
            esperaTempo = inicioTempo;
        }
        else
        {
            esperaTempo -= Time.deltaTime;
        }
    }
}

```

Agora para finalizar esse exemplo ajuste as informações desse código no Inspector conforme a imagem abaixo:

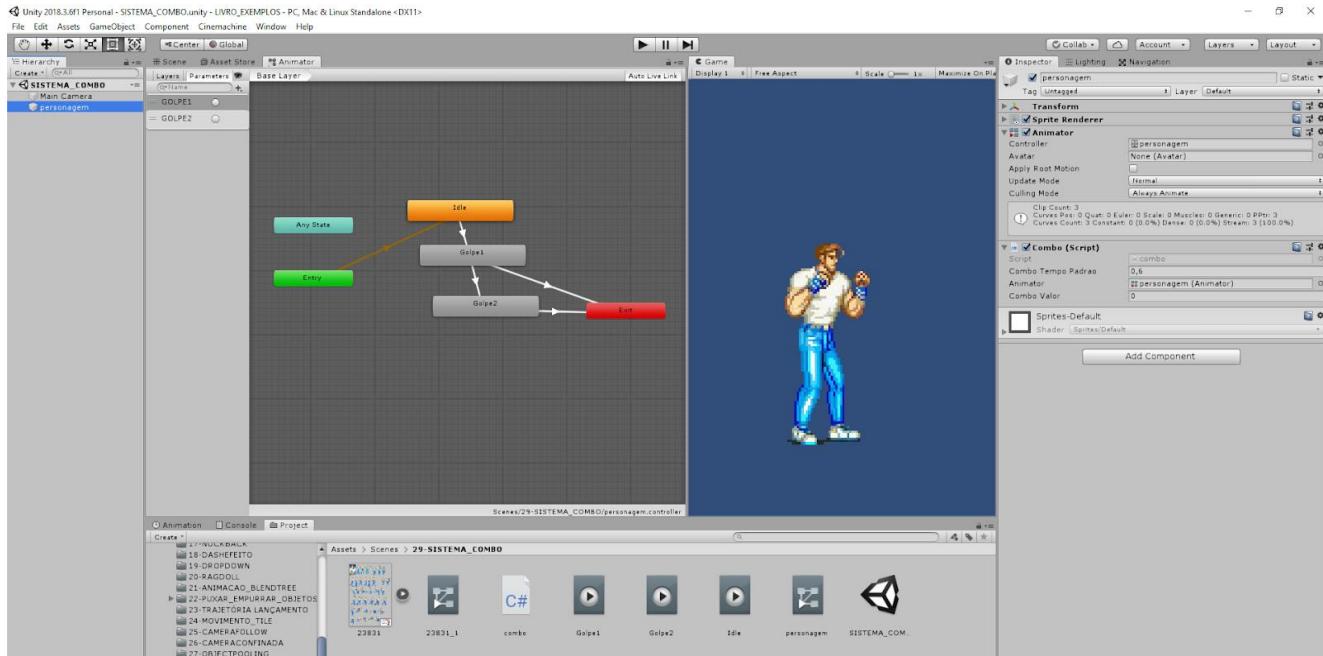


Prontinho agora é só executar o exemplo e ver tudo funcionando.

COMBO SIMPLES

Agora vamos entender como criar um sistema simples de combo onde de acordo com o numero de vezes que apertamos a tecla de um golpe acabamos desencadeado uma sequencia de golpes e não apenas uma animação repetida.

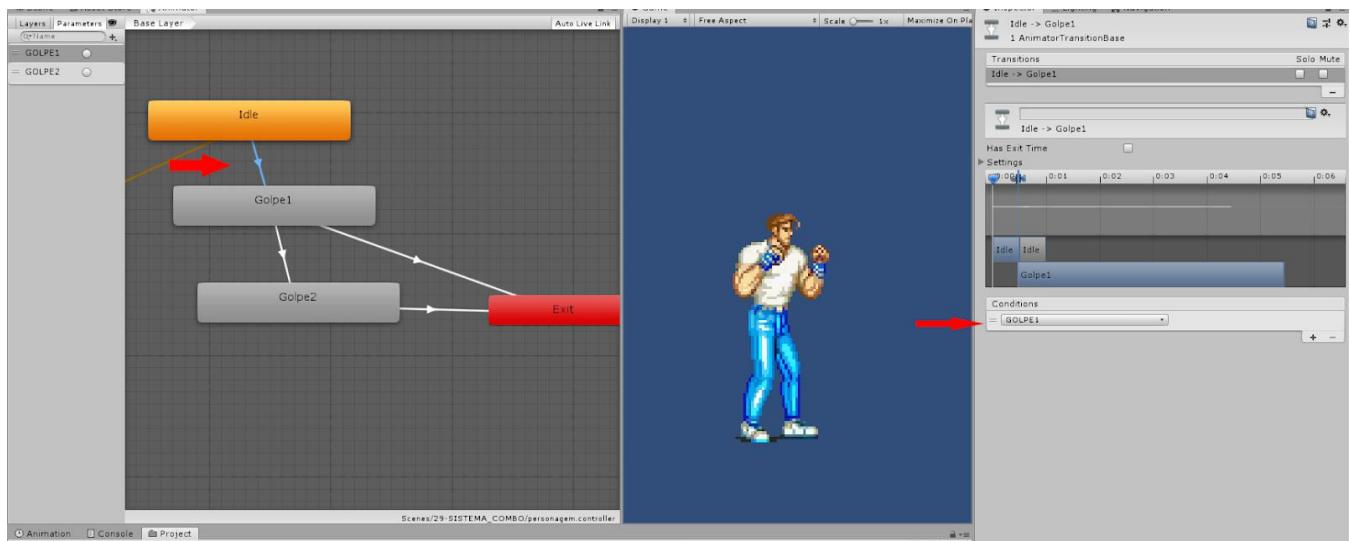
Para isso é necessário uma cena semelhante a da imagem abaixo:



Veja que tenho a imagem de um lutador e ao lado dele existe dentro do seu Animator duas animações que se ligam e terminam em exit, além disso temos dois parâmetros do tipo Trigger que servem para disparar os golpes.

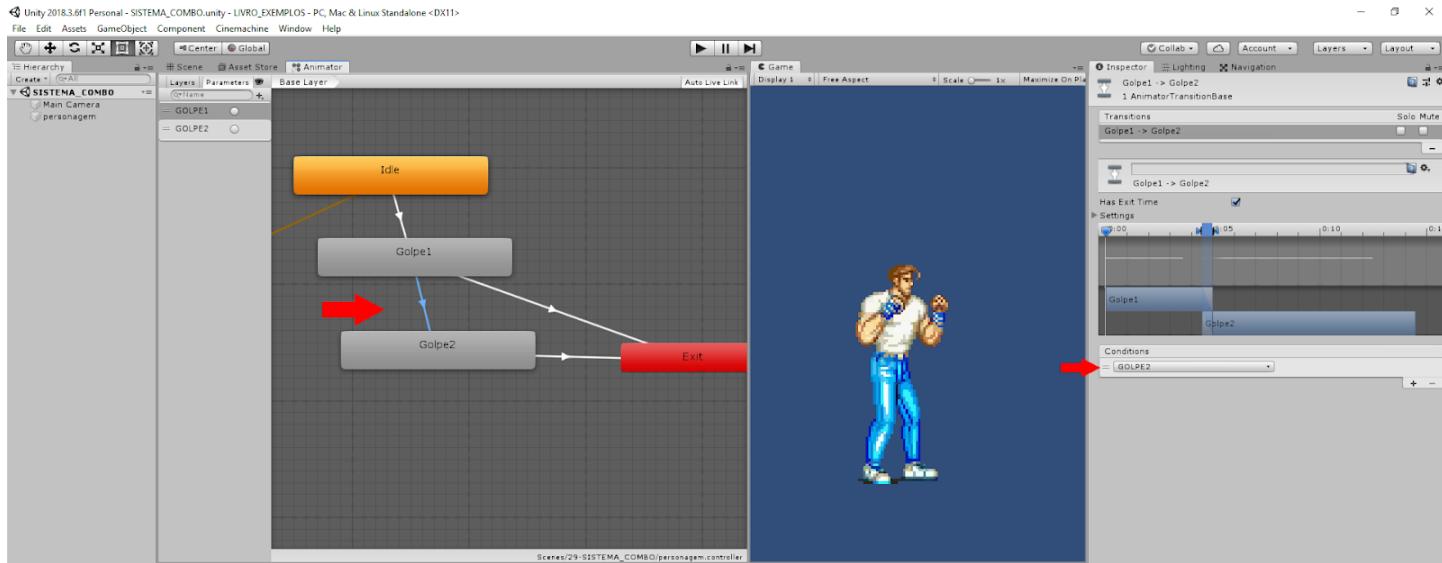
O personagem tem um código que controla essas animação o código se chama Combo, mas ainda não vamos ver esse código.

Primeiro precisamos dar uma olhadinha nas transições do nosso animator.



Veja que na imagem acima temos setas apontando para a transição que nos interessa nesse momento.

No caso a primeira transição vai para a animação de Golpe1 porém ela só vai se o trigger Golpe1 for acionado como mostra a condição do lado direito da imagem.



Depois temos a opção de ir para o Golpe 2 isso acontece se o trigger Golpe2 for acionado, caso contrario do Golpe 1 a animação vai para Exit e quando a animação chega no Golpe 2 o único caminho para ela é o exit também finalizando o combo.

O código que usamos para fazer isso acontecer é esse aqui:

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

public class combo : MonoBehaviour
{
    private bool ativaTempoReset;
    public float comboTempoPadrao = 0.4f;
    public Animator animator;
    public int comboValor;

    // Start is called before the first frame update
    void Start()
    {
        comboValor = 1;
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.P))

```

```

    {

        if(comboValor <= 3)
        {
            animator.SetTrigger("GOLPE1");
            comboValor++;
        }
        else if(comboValor > 3)
        {
            animator.SetTrigger("GOLPE2");
            comboValor = 1;
        }

    }

    if(Input.GetKeyUp(KeyCode.P))
    {
        ativaTempoReset = true;
    }

    ResetCombo();
}

void ResetCombo()
{
    if(ativaTempoReset)
    {
        comboTempoPadrao -= Time.deltaTime;
    }

    if(comboTempoPadrao <= 0)
    {
        comboTempoPadrao = 0.6f;
        comboValor = 1;
        ativaTempoReset = false;
    }
}
}

```

Nesse código iniciamos criando algumas variáveis como você pode ver abaixo:

```

private bool ativaTempoReset;
public float comboTempoPadrao = 0.4f;
public Animator animator;
public int comboValor;

```

Temos uma variável booleana para o ativar o tempo de reset, logo em seguida temos o float que contem o tempo padrão do combo.

Na sequencia temos o nosso animator e um int para o valor do combo.

No método Start atribuimos o valor 1 para a variável comboValor para que dessa forma tenhamos um valor inicial.

```
void Start()
{
    comboValor = 1;
}
```

Agora dentro do nosso método Update temos uma estrutura condicional que verifica se apertamos a tecla P, se realmente tivermos apertado essa tecla já verificamos o valor da variável comboValor se ela for menor que três desencadeamos o trigger do golpe1 e já incrementamos a variável comboValor para que seja possível passar para o segundo golpe.

Lembrando que sempre que apertamos a tecla P definimos a variável ativaTempoReset como verdadeira.

No fim de tudo chamamos o método ResetCombo que veremos em seguida.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.P))
    {

        if (comboValor <= 3)
        {
            animator.SetTrigger("GOLPE1");
            comboValor++;
        }
        else if (comboValor > 3)
        {
            animator.SetTrigger("GOLPE2");
            comboValor = 1;
        }
    }

    if (Input.GetKeyUp(KeyCode.P))
    {
        ativaTempoReset = true;
    }

    ResetCombo();
}
```

Veja que o método ResetCombo tem uma estrutura condicional que verifica se a variável ativaTempoReset é verdadeira se for subtraímos o valor de comboTempoPadrao por Time.deltaTime.

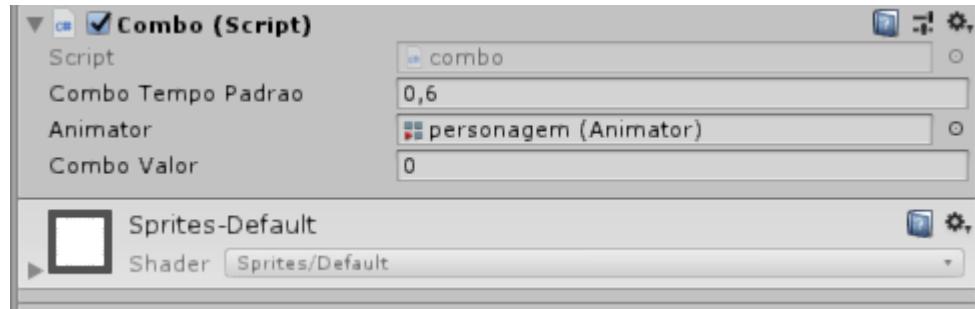
Quando o tempo chega em zero ou menor que isso ajustamos as variáveis de comboTempoPadrão e comboValor para reiniciarem.

E claro por último deixamos a variável ativaTempoReset como falsa.

```
void ResetCombo()
{
    if(ativaTempoReset)
    {
        comboTempoPadrao -= Time.deltaTime;
    }

    if(comboTempoPadrao <= 0)
    {
        comboTempoPadrao = 0.6f;
        comboValor = 1;
        ativaTempoReset = false;
    }
}
```

Com isso basta fazer os últimos ajustes conforme a imagem abaixo e colocar o exemplo para funcionar.

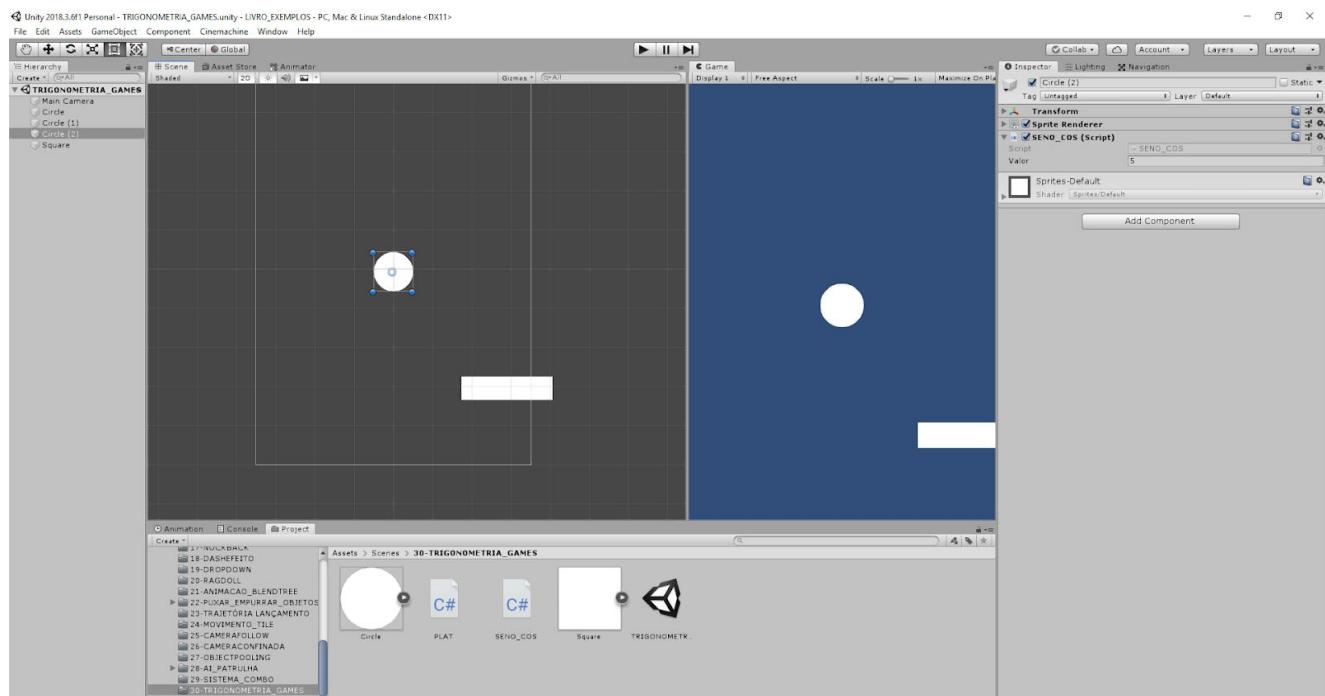


TRIGONOMETRIA SIMPLES PARA GAMES

Matemática pode ser um pesoado para muita gente, mas para outras pessoas é um verdadeiro sonho e quando se fala de trigonometria a coisa fica ainda mais bela.

Nesse exemplo vamos ver o básico do uso de trigonometria dentro do mundo dos games.

Para isso precisamos de uma cena semelhante a que aparece logo abaixo:



Veja que nessa cena temos apenas círculos e um retângulo, digo círculos pois temos três círculos sobrepostos ali todos eles com o mesmo código porém com valores diferentes.

Já o retângulo ali tem um código só dele que vamos analisar logo mais.

Então vamos começar vendo o código que é adicionado nas esferas.

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

public class SENO_COS : MonoBehaviour
{
    private Vector2 posInicial;
    public int valor;

    // Start is called before the first frame update
    void Start()
    {
        posInicial = transform.position;
    }
}
```

```

// Update is called once per frame
void Update()
{
    float y = valor * Mathf.Sin(Time.timeSinceLevelLoad) +
posInicial.y;
    float x = valor * Mathf.Cos(Time.timeSinceLevelLoad) +
posInicial.x;

    transform.position = new Vector2(x,y);

}
}

```

Nesse código iniciamos criando algumas variáveis

```

private Vector2 posInicial;
public int valor;

```

A primeira é a posição inicial do objeto que contem esse código e a segunda é o valor que sera multiplicado pelo efeito que sera realizado nesse exemplo.

Depois disso dentro do metodo Start temos o ajuste da variavel posInicial onde adicionamos a ela o valor da posição atual do objeto.

```

void Start()
{
    posInicial = transform.position;
}

```

No final temos no método Update dois cálculos um do seno do valor em segundos desde que o nível foi iniciado + a posição inicial do eixo y e depois calculamos o cosseno do valor em segundos desde que o nível foi iniciado + a posição inicial do eixo x.

E no final passamos os valores calculados para o posicionamento do objeto.

```

void Update()
{
    float y = valor * Mathf.Sin(Time.timeSinceLevelLoad) +
posInicial.y;
    float x = valor * Mathf.Cos(Time.timeSinceLevelLoad) +
posInicial.x;

    transform.position = new Vector2(x,y);

}

```

Com esse exemplo teremos uma rotação perfeita.

Agora vamos analisar o código do objeto retangular que é esse aqui:

```
using System.Collections;
```

```

using System.Collections.Generic;
using UnityEngine;

public class PLAT : MonoBehaviour
{
    private Vector2 posInicial;
    public int valor;

    // Start is called before the first frame update
    void Start()
    {
        posInicial = transform.position;
    }

    // Update is called once per frame
    void Update()
    {
        float y = valor * Mathf.Sin(Time.timeSinceLevelLoad) +
        posInicial.y;

        transform.position = new Vector2(transform.position.x,y);
    }
}

```

Nesse código temos basicamente a mesma coisa que no código anterior mas aqui ajustamos o movimento em Y para que a plataforma suba e desça sem parar.

PARTE 2

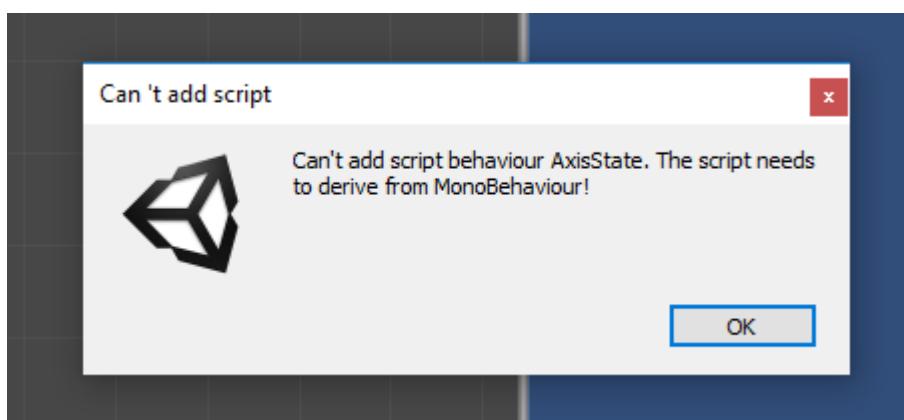
PROBLEMAS COMUNS DE DESENVOLVIMENTO DE JOGOS

PROBLEMA COM NOME DE CLASSES

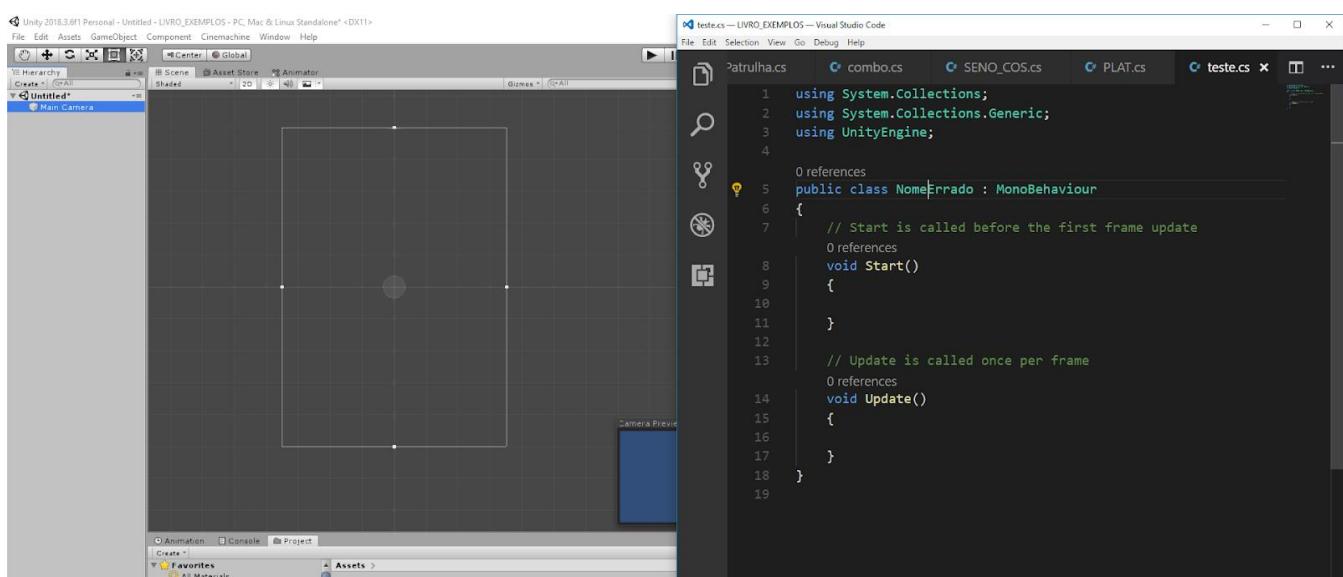
Um problema comum para desenvolvedores iniciantes é o erro do nome de classe, esse erro acontece quando criamos um arquivo de código e por alguma razão não adicionamos um nome ao arquivo.

Deixamos o nome padrão e depois quando percebemos editamos o nome do arquivo ainda dentro da janela Project.

Feito isso acreditamos que esta tudo certo e nenhum erro vai aparecer mas não é bem isso o que acontece quando escrevemos o código e tentamos usa-lo percebemos que o Unity acusa um erro que o erro.



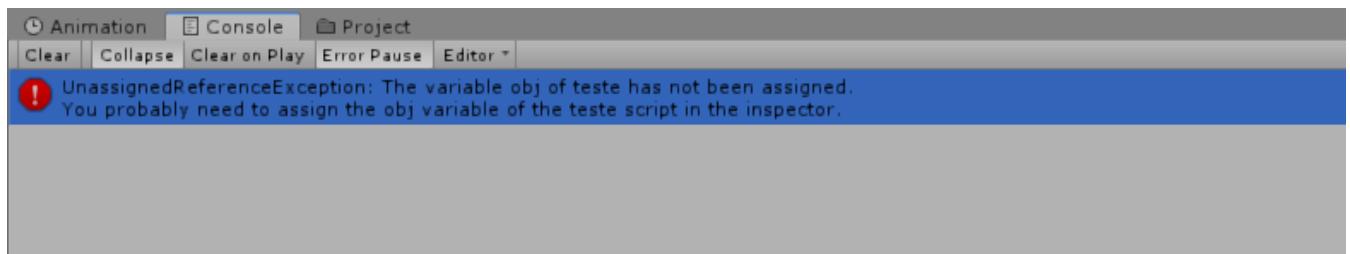
Se esse erro aparecer tenha certeza que o nome do seu arquivo e o nome da sua classe estão diferentes veja:



Para resolver isso basta deixar o arquivo e a classe com o mesmo nome.

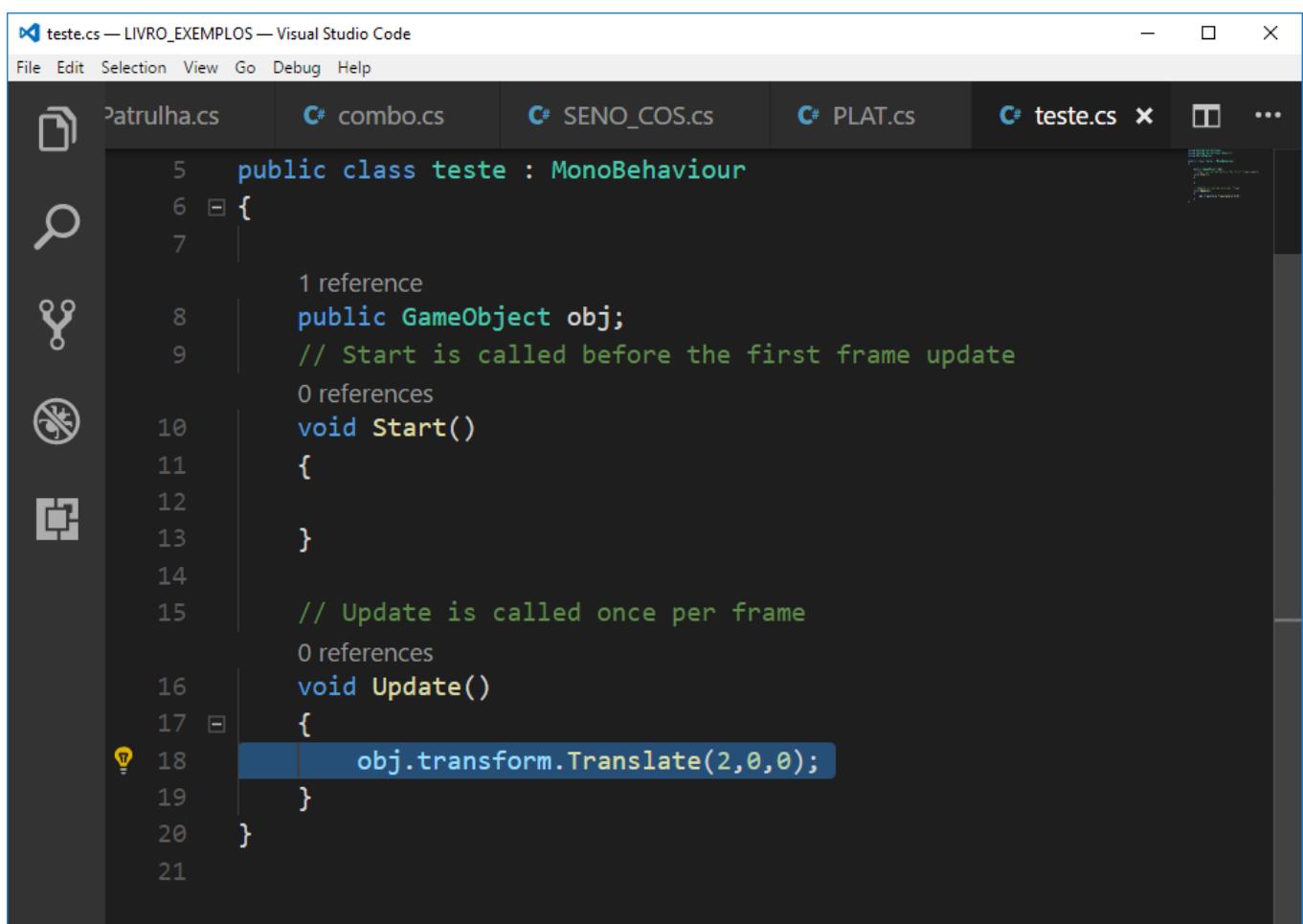
VARIÁVEIS NÃO ENCONTRADAS

Outro problema comum no desenvolvimento de jogos são as variáveis não encontradas. Imagine só que você escreveu seu código criou todas as variáveis que vai precisar e na hora de colocar o código para rodar aparece um erro desses:

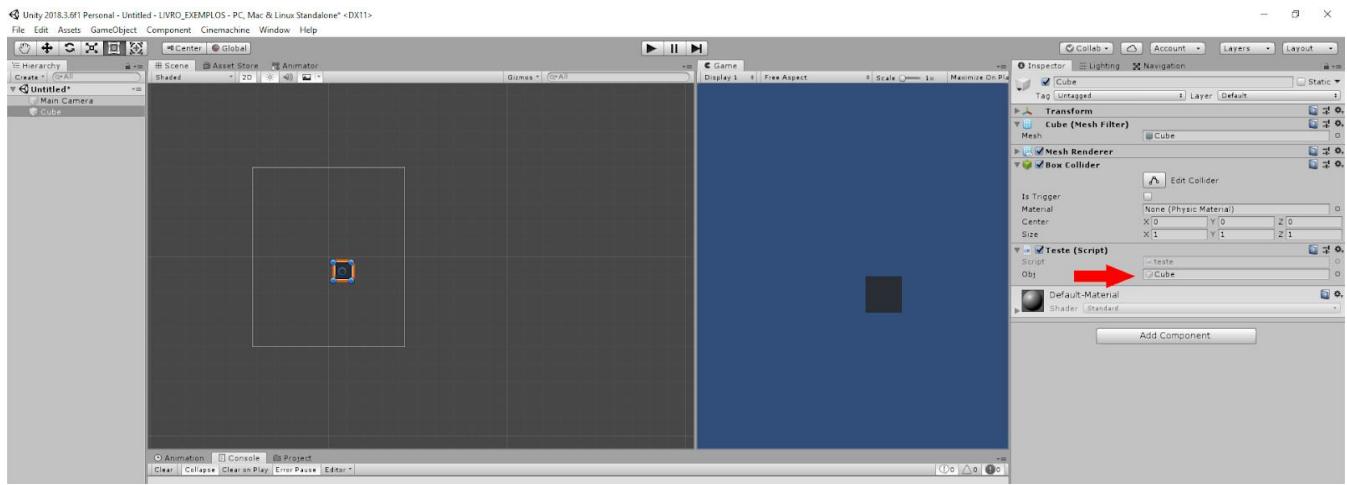


Esse erro está aparecendo por que você criou uma variável esta tentando usa-la mas esqueceu de dizer quem a variável é.

Então clique duas vezes sobre essa mensagem de erro para ser direcionado para a linha do erro dentro do código.



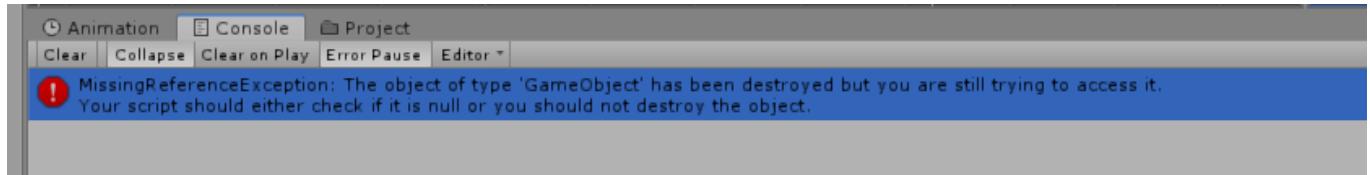
Veja que ele já te aponta onde está o problema e nesse caso o problema é que estamos tentando mover um objeto que o Unity não sabe quem é indicar no Inspector quem é o objeto que a variável se refere.



MATANDO QUEM DEVERIA ESTA VIVO

Outro problema comum de acontecer é matar um objeto dentro da cena e depois tentar usar esse objeto novamente.

É certeza que um erro vai aparecer.



Esse erro acontece por uma razão você destruiu um objeto e ainda está agindo como se ele existisse na cena, para evitar esse erro é necessário trabalhar no código veja:

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

public class teste : MonoBehaviour
{

    public GameObject obj;
    // Start is called before the first frame update
    void Start()
    {
        Destroy(obj);

    }

    // Update is called once per frame
    void Update()
    {
        if(obj != null)
        {
            obj.transform.Translate(2, 0, 0);
        }
    }
}
```

Veja que para evitar o erro antes de manipular o objeto verificamos se ele existe na cena se existir manipulamos o objeto caso contrario não faremos nada.

CHAMANDO MAIS PESSOAS DO QUE A FESTA SUPORTA

Um problema que geralmente tira o sono dos programadores é a lentidão do jogo por excesso de entidades em cena.

Resolver esse tipo de problema nunca é muito fácil você precisa analisar com calma quantos objetos podem estar em cena sem prejudicar seu jogo.

E controlar esse valor criando em cena um objeto somente se outro objeto for deletado. geralmente uma estrutura condicional resolve o problema.

```
void Update()
{
    if(numeroPersonagens < 10)
    {
        CriarPersonagem();
    }
}
```

Veja que no código acima só criamos um personagem na cena que o numero de personagens permitido for menor que 10 caso contrário nada feito.

AO INFINITO NEM SEMPRE VAI AO ALÉM...

Quando se trata de desenvolvimento de jogos precisamos ficar muito espertos para não cair em loops infinitos isso é terrível e honestamente sem serventia.

Então evite coisas do tipo:

```
for ( ; ; )
{
    print("oi");
}
```

Ou

```
while (true)
{
    print("oi");
}
```

Enfim evite esse tipo de coisa.

EVITANDO LIXO

Uma forma elegante de evitar lixo quando você trabalha com corotinas é definir o tempo fora da corotina dessa forma.

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

public class teste : MonoBehaviour
{
    public WaitForSeconds tempo = new WaitForSeconds(1);

    void Start()
    {

    }

    IEnumerator Teste()
    {
        yield return tempo;
        //Faz alguma coisa
    }
}
```

CONSIDERAÇÕES FINAIS

Então é isso pessoal! Tivemos aqui alguns exemplos de efeitos que deixam os games ainda mais divertidos, dinâmicos e completos, nesse volume o foco foi 2D, com uma pequena “palhinha” de 3D em alguns momentos, mas em breve teremos também uma edição voltada apenas ao 3D recheada de efeitos incríveis para deixar seu game perfeito e pronto para o mercado.