05

Múltiplas torres para combater infinitos inimigos

Gerando muitos inimigos

Um único inimigo já não é mais suficiente para manter nosso jogo interessante já que ele pode ser destruído facilmente por nossa torre. Vamos melhorar isso gerando mais inimigos!

Para começar, precisaremos de um objeto que ficará responsável por instanciar os novos inimigos. Como esse objeto não precisa ser visível para o jogador, que tipo de objeto iremos criar? Criaremos um *game object* vazio e associaremos a ele um script gerador de inimigos.

Uma forma prática de criar um objeto vazio no `Hierarchy` é utilizar o atalho **CMD+SHIFT+N** (Mac) ou **CTRL+SHIFT+N** (Windows). Depois só precisamos renomeá-lo para algo que faça mais sentido como `PontoGeradorDeInimigos`. Como iremos gerar novos inimigos na posição desse objeto, vamos posicioná-lo no começo do nosso caminho tomando o cuidado de mantê-lo sobre o terreno (coordenada `Y = 0`).

No script que associaremos ao `PontoGeradorDeInimigos`, vamos precisar instanciar novos `Inimigo` mas antes disso, o que precisamos fazer com o `Inimigo` para que isso seja possível? O que o método `Instantiate` recebia como um parâmetro? Um *prefab*!

Então o que precisamos fazer é transformar o `Inimigo` em um *prefab*. Para fazê-lo basta arrastar o `Hierarchy/Inimigo` para `Project/Prefabs` lembrando de remover o `Hierarchy/Inimigo` logo em seguida.

Com o *prefab* preparado, vamos agora adicionar um script `GeradorDeInimigos` ao objeto `PontoGeradorDeInimigos`. Esse script deverá instanciar novos inimigos em intervalos regulares. Onde já fizemos isso antes? Nossa torre também faz algo bastante similar, ela instancia novos mísseis em intervalos regulares! Logo vamos ter um script bem parecido para o nosso gerador de inimigos:

```
public class GeradorDeInimigos : MonoBehaviour
{

    [SerializeField] private GameObject inimigo;
    private float momentoDaUltimaGeracao;

    [Range(0,3)]
    [SerializeField] private float tempoDeCriacao = 2f;

    void Update ()
    {
        GeraInimigo ();
    }

    private void GeraInimigo ()
    {
        float tempoAtual = Time.time;
        if (tempoAtual > momentoDaUltimaGeracao + tempoDeCriacao)
        {
            momentoDaUltimaGeracao = tempoAtual;
            Vector3 posicaoDoGerador = this.transform.position;
            Instantiate (inimigo, posicaoDoGerador, Quaternion.identity);
        }
```

```
    }
}
```

Para que tudo funcione bem, só falta vincular o *prefab* do `Inimigo` ao nosso objeto `PontoGeradorDeInimigos`. Então selecionamos o `PontoGeradorDeInimigos` e arrastamos `Project/Prefabs/Inimigos` para `Inspector -> Gerador De Inimigos -> Inimigo`.

Pronto! Já temos nosso gerador de inimigos em funcionamento!

Escolhendo em quem atirar

Até esse momento, nossa torre não precisava se preocupar em como escolher seu alvo já que tínhamos apenas um inimigo. Agora que temos infinitos inimigos o que fazer? Outro problema é que nossa torre hoje tem alcance infinito! Isso significa que a sua posição não faz diferença alguma no nosso jogo. Precisamos fazer com que mais prá frente, o posicionamento de uma torre se torne uma decisão importante para o jogador.

Um jeito de conseguir isso é fazendo com que a torre tenha um raio de alcance limitado. Dessa forma, ela só poderia atirar nos inimigos que estivessem mais próximos. Podemos então adicionar um atributo `raioDeAlcance` ao script da `Torre`:

```
public class Torre : MonoBehaviour
{
    [SerializeField] private float raioDeAlcance;
}
```

Mas se agora a torre tem um raio de alcance, nem todos os inimigos do cenário são alvos válidos já que eles podem estar longe demais. Para levar isso em conta, vamos começar criando um método `EscolheAlvo` que inicialmente ficará responsável por pegar apenas os inimigos que estão na área de alcance da torre.

Uma forma de fazer isso é utilizando o método estático `FindGameObjectsWithTag` do *game object*. Esse método devolve todos os objetos da cena que possuem uma *tag* específica. Para nossa sorte, já temos uma *tag* que distingue os inimigos dos demais objetos! Vamos então fazer essas alterações no script `Torre`:

```
public class Torre : MonoBehaviour
{
    private Inimigo EscolheAlvo()
    {
        GameObject[] inimigos = GameObject.FindGameObjectsWithTag("Inimigo");
        return null;
    }
}
```

Com isso temos todos os inimigos da cena armazenados no *array* `inimigos`. Como fazemos agora para verificar se estes inimigos se encontram ao alcance da torre? Basta calcular a distância de cada um deles até a nossa torre e comparar com o raio de alcance!

Mas vamos devagar porque vamos precisar utilizar mais alguns novos métodos do Unity. Primeiro vamos percorrer o nosso *array* de inimigos utilizando a instrução `foreach` do C#:

```
public class Torre : MonoBehaviour
{
```

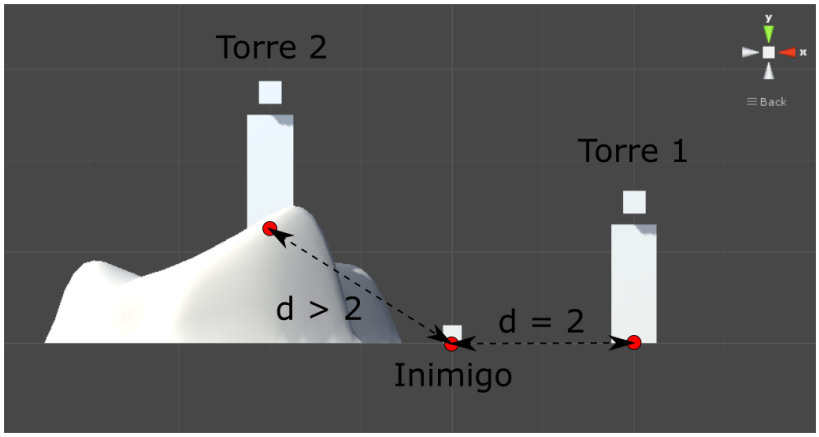
```
{
    private Inimigo EscolheAlvo()
    {
        GameObject[] inimigos = GameObject.FindGameObjectsWithTag("Inimigo");
        foreach (GameObject inimigo in inimigos)
        {
            //Como saberemos em qual alvo atirar?
        }
        return null;
    }
}
```

Agora podemos fazer o cálculo da distância do inimigo até a torre. Para fazer isso podemos utilizar o método estático `Distance` da classe `Vector3`. Esse método recebe dois `Vector3` como parâmetros e devolve a distância entre esses dois vetores.

Então para calcular a distância entre nossa torre e o inimigo podemos utilizar a instrução:

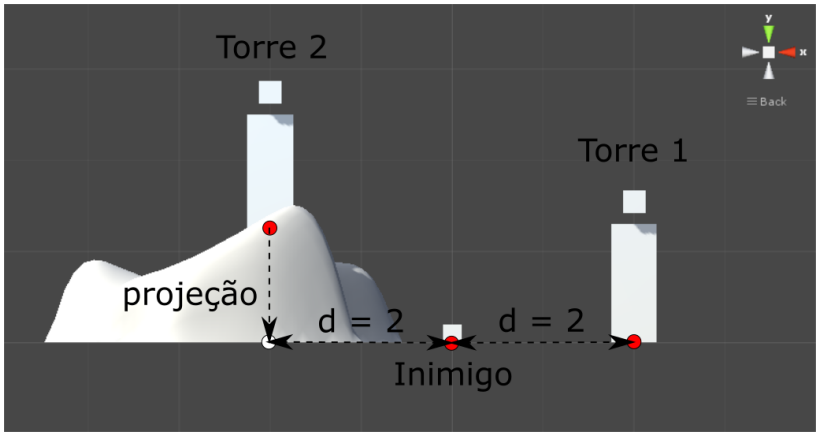
```
float distancia = Vector3.Distance(posicaoDoInimigo, posicaoDaTorre);
```

Mas atenção! A distância será calculada com relação à coordenada (0, 0, 0) de cada objeto! Isso significa que se a nossa torre estiver em algum ponto mais elevado do terreno pode ser que ela não consiga atirar nos inimigos como podemos ver na figura abaixo.



O `Inimigo` se encontra a duas unidades de distância da `Torre 1`. Já em relação a `Torre 2` a distância é maior que `2`. Se as torres tiverem um raio de alcance de `2` então somente a `Torre 1` conseguiria atirar.

Este comportamento pode confundir o jogador então seria interessante calcular essa distância considerando apenas o plano formado pelos eixos X e Z. Podemos fazer isso projetando a posição das nossas torres e do inimigo no plano XZ como ilustrado na figura abaixo:



Novamente podemos utilizar um método da classe `Vector3` para fazer isso por nós. Utilizaremos o método estático `ProjectOnPlane` para projetar a posição de um objeto no plano especificado.

Para especificar a posição do objeto, utilizaremos o vetor que representa sua posição. Mas como fazemos para especificar o plano onde queremos projeetar essa posição? Utilizaremos também um vetor: o vetor normal! O vetor normal representa a direção perperdicular a uma superfície. Para simplificar, o vetor normal terá a mesma direção de uma caneta colocada de pé sobre o seu plano.

Como vamos querer que os nosso objetos sejam projetados no plano XZ, uma caneta colocada de pé sobre esse plano vai apontar na mesma direção do eixo Y. O Unity já possui tal vetor pré-definido como uma constante `Vector3.up` .

Agora sim podemos retornar então ao nosso script e utilizar todas essas novas informações para concluir nosso método `EscolheAlvo` . Para organizar o nosso código, vamos criar um método `EstaNoRaioDeAlcance` que vai ficar responsável por fazer o cálculo da distância e verificação do alcance:

```
public class Torre : MonoBehaviour
{

    private bool EstaNoRaioDeAlcance(GameObject inimigo)
    {
        Vector3 posicaoDoInimigoNoPlano =
            Vector3.ProjectOnPlane(inimigo.transform.position, Vector3.up);
        Vector3 posicaoDaTorreNoPlano =
            Vector3.ProjectOnPlane(this.transform.position, Vector3.up);

        float distanciaParaInimigo =
            Vector3.Distance (posicaoDaTorreNoPlano, posicaoDoInimigoNoPlano);

        return distanciaParaInimigo <= raioDeAlcance;
    }

    private Inimigo EscolheAlvo()
    {
        GameObject[] inimigos = GameObject.FindGameObjectsWithTag("Inimigo");
        foreach (GameObject inimigo in inimigos)
        {
            if (EstaNoRaioDeAlcance(inimigo))
            {
                return inimigo.GetComponent<Inimigo>();
            }
        }
        return null;
    }
```

```
    }
}
```

Antes de testar o jogo, vamos selecionar a `Torre` e no `Inspector` alterar o valor `Raio de Alcance = 2`.

Agora podemos testar o jogo mas nesse momento veremos que nossa torre ainda não funciona como o esperado. Isso acontece porque ainda não invocamos o método `EscolheAlvo` em nenhum ponto do script `Torre`!

Informando o alvo para o míssil

Inicialmente precisamos fazer com que nossa torre escolha um alvo para atirar. Como os inimigos estão em movimento, nossa torre precisa a todo momento verificar se existem inimigos ao alcance.

Nesse caso, qual seria o método mais adequado para pedir para a torre escolher um alvo? O método `Update`! Como vimos anteriormente ele é chamado a cada novo quadro do nosso jogo.

Então agora teremos o seguinte código no método `Update` do script `Torre`:

```
public class Torre : MonoBehaviour
{
    void Update ()
    {
        Inimigo alvo = EscolheAlvo ();
        if (alvo != null)
        {
            //É só passar o alvo para o método atira...
            Atira (alvo);
        }
    }
}
```

Como agora o método `Atira` recebe um alvo como parâmetro, vamos ter que alterá-lo também. Só precisamos lembrar de um detalhe: nosso míssil até agora sempre tinha como alvo um objeto de nome "Inimigo". Como agora cada míssil pode ter um alvo diferente escolhido pela torre, vamos precisar também alterar o funcionamento do nosso míssil. Então, começamos pelo método `Atira` da `Torre`:

```
public class Torre : MonoBehaviour
{
  private void Atira (Inimigo inimigo)
  {
    float tempoAtual = Time.time;
    if (tempoAtual > momentoDoUltimoDisparo + tempoDeRecarga)
    {
      momentoDoUltimoDisparo = tempoAtual;
      GameObject pontoDeDisparo =
        this.transform.Find ("CanhaoDaTorre/PontoDeDisparo").gameObject;
      Vector3 posicaoDoPontoDeDisparo =
        pontoDeDisparo.transform.position;
      GameObject projetilObject =
        (GameObject) Instantiate (projetilPrefab,
                                  posicaoDoPontoDeDisparo,
                                  Quaternion.identity);
      Missil missil = projetilObject.GetComponent<Missil>();
```

```
            missil.DefineAlvo(inimigo);
        }
    }
}
```

Agora vamos fazer as alterações necessárias no `Missil`. Primeiro vamos implementar o método `DefineAlvo`:

```
public class Missil : MonoBehaviour
{

    public void DefineAlvo(Inimigo inimigo)
    {
        alvo = inimigo;
    }
}
```

Mas cuidado porque o atributo `alvo` até o momento se referia a um `GameObject` e agora queremos guardar um `Inimigo` nele. Vamos alterar seu tipo:

```
public class Missil : MonoBehaviour
{
    private Inimigo alvo;
}
```

Agora só falta remover o comportamento anterior que escolhia um alvo de nome "Inimigo":

```
public class Missil : MonoBehaviour
{
    void Start ()
    {
        // Remover a linha abaixo
        // alvo = GameObject.Find("Inimigo");

        AutoDestroiDepoisDeSegundos (10);
    }
}
```

Agora sim! Podemos testar o nosso jogo novamente e ver que a torre só atira nos inimigos que se aproximam dela.

Construindo torres

Neste momento, estamos sendo atacados por vários inimigos e nossa única torre agora está com um alcance limitado e precisa disparar vários mísseis para destruir um alvo! Para aumentar nossas defesas, vamos permitir ao jogador clicar em algum lugar estratégico no nosso mapa e construir múltiplas torres neste local escolhido.

Vamos construir uma `Torre` somente quando o jogador clicar no nosso mapa, mas como sabemos um clique foi feito?

A classe Input

Para detectarmos qual tecla (ou botão do mouse) foi pressionada, podemos usar a classe `Input`, que nos oferece métodos para trabalhar com teclados, mouses e até mesmo joysticks!

Podemos usar o método `GetMouseButtonDown` para sabermos se determinado botão do mouse foi pressionado no **frame atual**:

```
public bool Clicou ()
{
    return Input.GetMouseButtonDown(/* Qual botão do mouse? */);
}
```

Esse método recebe como argumento o botão do mouse que se deseja monitorar no seguinte padrão:

- `0`: botão esquerdo do mouse.
- `1`: botão direito do mouse.
- `2`: botão central do mouse.

Então, para sabermos se o botão esquerdo do mouse foi clicado **no frame atual**, basta fazermos:

```
public bool Clicou ()
{
    return Input.GetMouseButtonDown(0);
}
```

Onde chamaremos esse método para sabermos se, no frame atual, o jogador fez um clique? Existe alguma classe no Unity que possui uma estrutura a ser chamada em **todo frame**?

Podemos criar uma classe filha de `MonoBehaviour` e usar seu método `Update`!

```
public class Container : MonoBehaviour
{
    void Update ()
    {
        if (Clicou())
        {
            // Vamos construir uma nova torre em qual posição?
        }
    }
}
```

Para saber mais...

Além de capturarmos o clique do mouse, podemos obter informações das teclas pressionadas num teclado, da aceleração registrada pelo acelerômetro e até mesmo dos eixos acionados em um joystick!

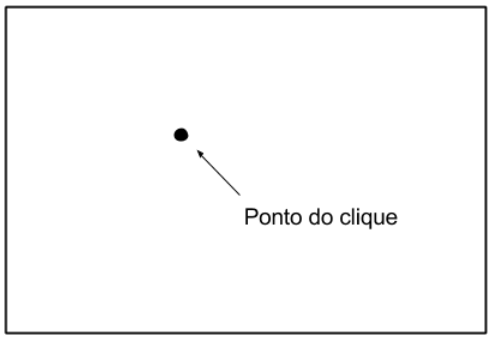
Veja mais detalhes na [documentação do Unity (http://docs.unity3d.com/ScriptReference/Input.html)](http://docs.unity3d.com/ScriptReference/Input.html).

Para capturar a posição do clique do jogador na tela, podemos acessar uma variável estática da própria classe `Input`:

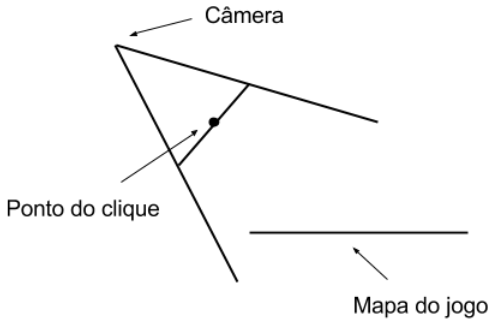
```
if (Clicou())
{
    Vector3 pontoDoClique = Input.mousePosition;
}
```

Ray cast

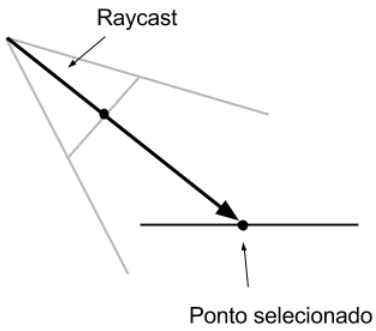
O que obtemos com `Input.mousePosition` representa um ponto do clique **na tela** do jogo:



Mas o que vemos na tela do jogo é simplesmente a informação gerada pela nossa câmera que posicionamos. Então, ao clicar na tela, precisamos converter esse ponto da tela da câmera em um ponto **pertencente ao nosso mapa do jogo**, ao nosso terreno:



Para fazer isso, uma técnica bastante usada é traçar um raio entre a nossa câmera e o ponto do clique até alcançar nosso mapa. O ponto onde esse raio cruzar nosso mapa é o ponto que devemos construir nossa torre:



Essa técnica é conhecida como *ray cast*!

Para saber mais... *ray cast* e *ray tracing*

Enquanto no *ray cast* disparamos um raio que não se separa em raios secundários, na técnica conhecida por *ray tracing* fazemos um processo recursivo assim que o raio atinge algum objeto, disparando vários outros raios (e repetindo o processo).

Apesar de ser mais custoso do que o *ray cast*, usamos *ray tracing* para fazer efeitos de reflexão e refração em objetos.

Usando a classe Raycast do Unity

Apesar da teoria complicada, o Unity já possui uma método na classe `Physics` para disparar um *ray cast*!

```
Physics.Raycast (raio, out raycast, comprimentoMaximo);
```

Só precisamos passar um raio saindo da câmera, uma variável que vai guardar as informações retornadas pelo método e o comprimento máximo desse raio.

A palavra-chave out

No C#, existe uma palavra-chave chamada `out` que, quando usada no argumento de um método, força esse argumento a ser passado por referência:

```
class MinhaClasse
{
    static void UmMetodo(out int x)
    {
        x = 44;
    }

    static void Main()
    {
        int numero;
        UmMetodo(out numero);

        // aqui, numero vale 44.
```

```
    }
}
```

Nosso raio deve sair da nossa câmera, passando pelo ponto do clique na tela, então, podemos usar o método `ScreenPointToRay` para nos devolver o que queremos:

```
if (Clicou())
{
    Vector3 pontoDoClique = Input.mousePosition;
    Ray raioDaCamera = Camera.main.ScreenPointToRay (ponto);
    Physics.Raycast (raioDaCamera, out ??, ??);
}
```

Agora, vamos criar uma variável para armazenar as informações do raio gerado pelo método:

```
if (Clicou())
{
    Vector3 pontoDoClique = Input.mousePosition;
    Ray raioDaCamera = Camera.main.ScreenPointToRay (ponto);
    RaycastHit infoDoRaio;
    Physics.Raycast (raioDaCamera, out infoDoRaio, ??);
}
```

Por fim, só precisamos falar qual o comprimento máximo do *ray cast*:

```
if (Clicou())
{
    Vector3 pontoDoClique = Input.mousePosition;
    Ray raioDaCamera = Camera.main.ScreenPointToRay (ponto);

    RaycastHit infoDoRaio;
    float comprimentoMaximo = 100.0f;
    Physics.Raycast (raioDaCamera, out infoDoRaio, comprimentoMaximo);
}
```

Agora, só precisamos ver se esse raio do *ray cast* colidiu com nosso terreno e, caso tenha colidido, pegar seu ponto. Nesse ponto, podemos instanciar nossa nova `Torre`:

```
if (Clicou())
{
    RaycastHit infoDoRaio;
    //...

    if(infoDoRaio.collider != null) {
        Vector3 posicaoDoElemento = infoDoRaio.point;
        Instantiate(torrePrefab, posicaoDoElemento, Quaternion.identity);
    }
}
```