07

Construindo a torre e disparando mísseis

Começando a batalha: nossa primeira torre

Até o momento fizemos um cenário, manipulamos a câmera, luz e temos até um inimigo percorrendo um caminho, mas ainda falta o principal: a Torre!

Durante a criação de um jogo, existem vários profissionais envolvidos de diversas áreas: programadores, designers, animadores, desenhistas, engenheiros de som, roteiristas...

O modelo 3D de uma torre geralmente passa pelos desenhistas, animadores e designers para que os programadores possam incorporá-la ao jogo. Será que os programadores tem que esperar todo esse processo terminar para só então começar a trabalhar com a torre?

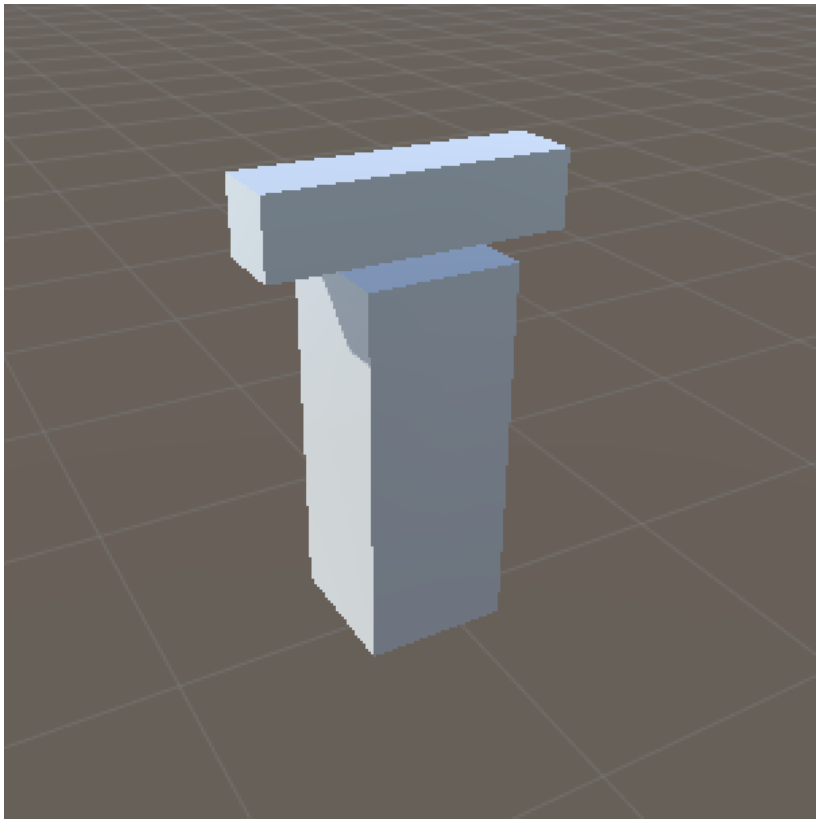
Para evitar essa ociosidade e diminuir o tempo de criação de um jogo, enquanto o modelo da torre é feito, os programadores simulam o modelo com formas 3D mais simples, mas com as mesmas características da torre original!

Dessa forma, quando o modelo 3D da torre estiver pronto, é só trocar a forma simples pelo modelo final. Isso é bastante fácil de fazer no Unity.

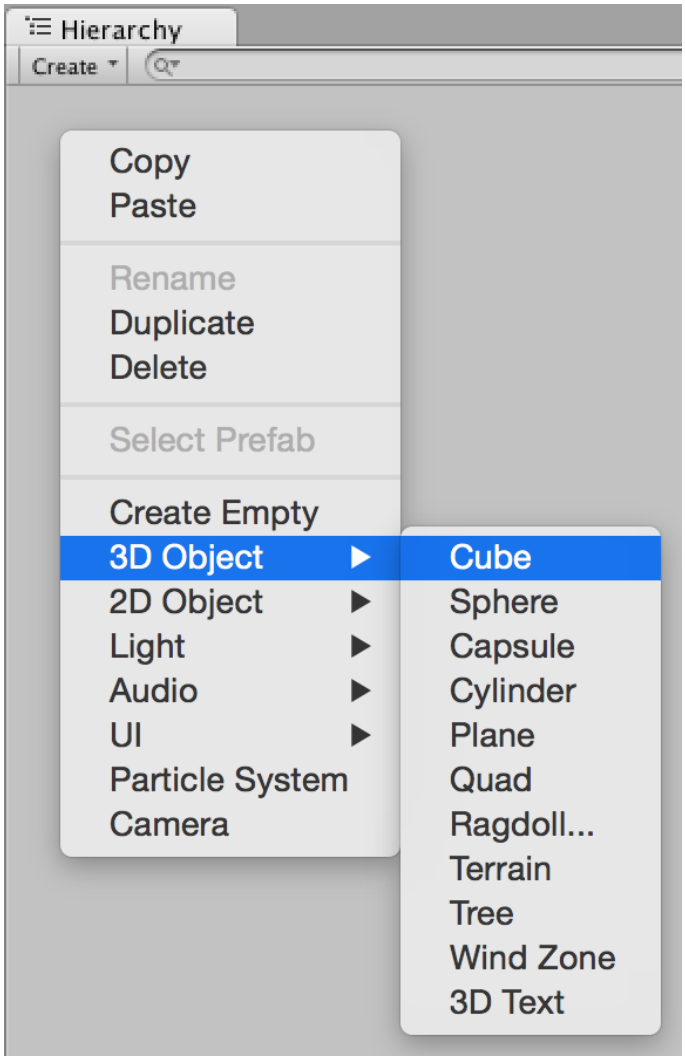
Então, vamos começar a programar nossa torre, mesmo sem ter o modelo 3D pronto neste momento, para vermos como funciona esse fluxo.

Como será essa torre?

Nossa torre será composta por dois "cubos": um será o **corpo da torre** e o outro será o **canhão da torre** arranjados dessa forma:



Como essa torre será um objeto do nosso jogo, lembre-se que ela será também um *Game Object*! Então, podemos criar as partes da nossa torre diretamente na aba `Hierarchy`.

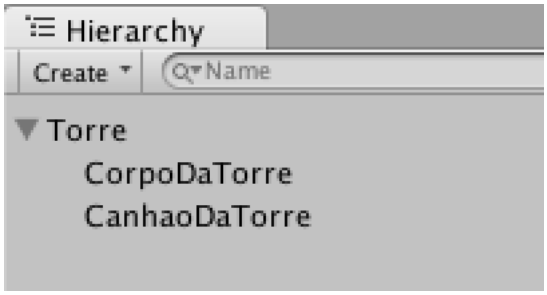


Agora temos um `CorpoDaTorre` e um `CanhaoDaTorre`, mas em nenhum lugar temos um objeto chamado, de fato, `Torre`. Caso queiramos mover todas as partes da torre, precisaríamos selecionar uma a uma, e se nosso objeto for composto por 20 partes diferentes?

Para gerenciarmos melhor nossos objetos e transformá-los num único corpo, podemos criar um outro *Game Object* cuja única função será agrupar as diversas partes do nosso objeto composto. Mas qual *Game Object* serviria para essa função? Podemos criar mais um `Cube`, ou uma `Sphere` ...?

Veja que temos a possibilidade de criar um **Empty** *Game Object*, que pode ser usado justamente para agrupar outros objetos!

Com esse `Empty Game Object` criado, podemos simplesmente colocar as partes da torre em seu interior e chamá-lo de `Torre`.



Dessa forma, podemos arrastar e fazer qualquer configuração à torre como um todo, sem ter que repetir cada configuração para todas as suas partes.

Míssil

Criamos nossa torre, mas até o momento ela não faz nada; nossos inimigos podem andar tranquilamente na sua frente que nada acontecerá. Para mudar essa situação, vamos tornar nossa `Torre` um pouco mais perigosa fazendo-a disparar mísseis!

Vamos trabalhar com um objeto simples que representará nosso míssil: um `Cube` .

A diferença dessa vez está na seguinte pergunta: quem será o responsável por criar esse *Game Object*? O jogador já verá criados todos os mísseis que nossa torre pode disparar?

Como nossa `Torre` será a responsável por disparar o `Missil` , temos que torná-lo um objeto instanciado dinamicamente no nosso jogo: um **prefab**.

Prefab em detalhes

Imagine que tenhamos um *Game Object* criado para cada `Missil` a ser disparado pela nossa `Torre` , mas agora queremos alterar o tamanho desses mísseis. Como os *Game Objects* são independentes dos outros (mesmo que tenham o mesmo nome), se quisermos alterar um atributo de um *Game Object*, teríamos que replicar essa alteração em **todos** os outros *Game Objects* da nossa cena.

Para não precisarmos fazer esse trabalho, o Unity possui um tipo de objeto chamado **Prefab** que permite armazenarmos um *Game Object* juntamente com todos os seus componentes e atributos. Então, quando precisarmos criar um *Game Object* a partir de um *prefab*, basta instanciar esse *prefab* usando o método:

```
Instantiate (meuPrefab);
```

Uma grande vantagem do uso de *prefabs* é que qualquer alteração feita num *prefab* é refletida para todos os objetos na cena instanciados a partir dele!

Disparando mísseis

Precisamos definir o comportamento do responsável em disparar mísseis, mas quem seria esse responsável? A própria `Torre` !

Então, vamos criar um **script C#** chamado `Torre` que conterá o comportamento da nossa torre. Como todo script que define o comportamento de um *Game Object*, esse deverá também ser filho de `MonoBehaviour` :

```
public class Torre : MonoBehaviour
{

}
```

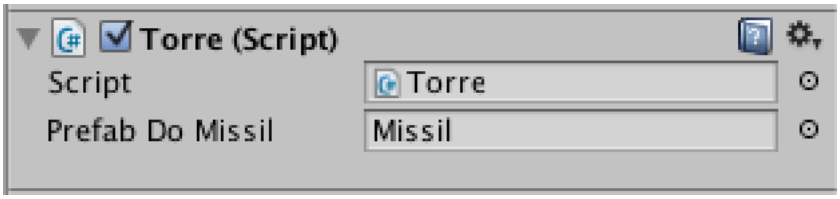
Logo que o objeto `Torre` aparecer na cena, vamos fazê-lo disparar um `Missil` . Então, vamos usar o próprio método `Start` do `MonoBehaviour` para instanciar nosso *prefab*:

```
public GameObject prefabDoMissil;

void Start()
{
    Instantiate (prefabDoMissil);
}
```

Neste momento, como o nosso script `Torre` sabe quem é o *prefab* que representa o `prefabDoMissil` ? Ele simplesmente não sabe! Precisamos, agora, vincular nosso *prefab* a esse atributo.

Usaremos a própria interface do Unity para isso:



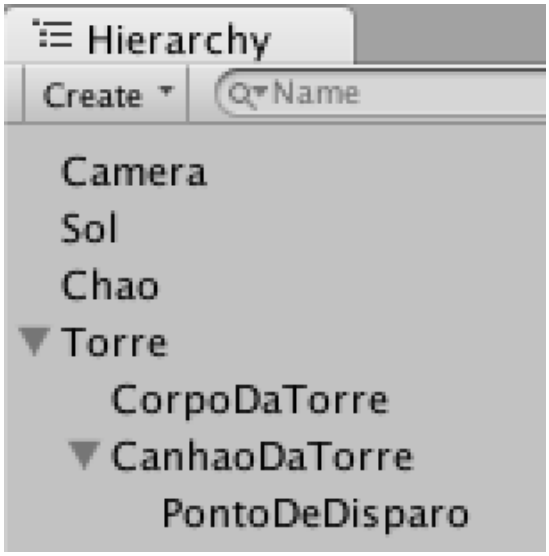
Tornando o ponto de disparo relativo à torre

Ao chamar o método `Instantiate` passando um *prefab*, por padrão o *Game Object* instanciado é posicionado de acordo com o atributo `position` do *prefab*. Ao mover a `Torre` , como não alteramos a posição do nosso *prefab*, o `Missil` continuou sendo disparado da posição anterior.

Mas sempre teremos que lembrar de posicionar o `Missil` perto da `Torre` ? E se tivermos mais de uma `Torre` , onde o *prefab* do `Missil` ficará posicionado?

Para resolvermos isso, vamos criar na nossa `Torre` um **ponto de disparo**, que usaremos como origem dos mísseis instanciados por essa `Torre` . Dessa forma, cada `Missil` disparado por uma `Torre` saberá em qual ponto deverá nascer.

Como o ponto de disparo não é nada além de um ponto, podemos criar um `Empty Game Object` para representá-lo. Além disso, como o ponto de disparo pertence ao `CanhaoDaTorre` , podemos explicitar isso colocando o ponto dentro do canhão na hierarquia:



Como usaremos esse ponto no Instantiate?

Agora, podemos usar uma versão mais "inteligente" do método `Instantiate`, que recebe a `posição` onde o *prefab* será instanciado e uma `rotação` inicial também:

```
Instantiate (prefab, posicao, rotacao);
```

Mas, para saber a `posicao`, precisamos encontrar nosso `PontoDeDisparo`. Para isso, podemos usar o método `Find` passando o **nome** do *Game Object* desejado e, na sequência, obter sua posição:

```
GameObject pontoDeDisparo =
          this.transform.Find ("CanhaoDaTorre/PontoDeDisparo").gameObject;
Vector3 posicao = pontoDeDisparo.transform.position;
```

Por fim, podemos atribuir uma rotação que o objeto instanciado terá. Podemos usar a própria rotação contida no *prefab* chamando `transform.rotation`:

```
GameObject pontoDeDisparo =
          this.transform.Find ("CanhaoDaTorre/PontoDeDisparo").gameObject;
Vector3 posicaoDoPontoDeDisparo = pontoDeDisparo.transform.position;
Instantiate (projetilPrefab, posicaoDoPontoDeDisparo, transform.rotation);
```

Com isso, somos capazes de disparar mísseis sempre na mesma posição **relativa à torre**.

Implementando a movimentação do míssil

Conseguimos disparar o `Missil`, mas ainda ele fica "congelado" na frente do canhão da nossa torre. Agora, vamos fazê-lo se movimentar pelo campo de batalha.

Como faremos para nosso `Missil` se movimentar? Estamos falando de um comportamento que somente nosso `Missil` terá, então, precisaremos de um script associado ao nosso *Game Object*!

Vamos criar um script C# chamado `Missil`, que deverá ser filho de `MonoBehaviour` para garantir que esse comportamento será aplicado ao nosso objeto do jogo:

```
public class Missil : MonoBehaviour
{
    void Start ()
    {

    }

    void Update ()
    {

    }
}
```

Por padrão, o Unity já popula nossa classe com os métodos `Start` e `Update`. Já vimos antes que o `Start` é chamado sempre que nosso objeto for criado, mas e o método `Update`?

O método Update

Todo `MonoBehaviour` possui o método `Update`, que é chamado pelo Unity a **cada *frame*** do jogo. Essa característica do `Update` permite implementarmos atualizações de qualquer tipo em um *Game Object* com a garantia de que será efetuada a cada frame do nosso jogo!

Para mover o `Missil`, precisaremos **reposicionar** nosso objeto a cada frame do jogo, então esse será um uso perfeito para o método `Update`. Mas como será feita essa alteração de posição?

```
void Update ()
{
    Vector3 posicaoAtual = transform.position;
    //Como alteraremos essa posicaoAtual?
}
```

Podemos fazer nosso objeto se mover para a frente com uma velocidade constante de 10 metros. Por convenção, o eixo `z` de qualquer *game object* é tratado como a sua "frente". Basta capturarmos o eixo `z` e seremos capazes de mover nosso objeto para a frente.

Para facilitar, todo `GameObject` já possui uma forma de obtermos especificamente o eixo Z. Basta fazer:

```
Vector3 frente = transform.forward;
```

Agora é só usar essa `frente` e multiplicar pela nossa velocidade para sabermos o quanto temos que mover nosso objeto:

```
private float velocidade = 10;

void Update ()
{
    Vector3 posicaoAtual = transform.position;
    Vector3 frente = transform.forward;
    Vector3 deslocamento = frente * velocidade;
}
```

Agora, podemos adicionar esse `deslocamento` à `posicaoAtual` do nosso `GameObject`:

```
private float velocidade = 10;

void Update ()
{
    Vector3 posicaoAtual = transform.position;
    Vector3 frente = transform.forward;
    Vector3 deslocamento = frente * velocidade;
    transform.position = posicaoAtual + deslocamento;
}
```

Então, a **cada frame** movemos nosso `GameObject` em 10 metros. Será que isso é suficiente para garantir um deslocamento constante?

Frames não são constantes

Um frame nada mais é do que a exibição de um único instante do jogo. Porém, para esse frame ser exibido, muitos cálculos devem ser efetuados pela GPU, e esses cálculos não possuem um tempo fixo para serem encerrados; dessa forma, um frame pode demorar mais do que outro para ser exibido!

No nosso caso, como estamos deslocando o `GameObject` com uma velocidade de 10, na realidade estamos dizendo que queremos alterar nosso objeto com uma velocidade de 10 metros **por frame** em vez de 10 metros **por segundo**.

Para corrigir isso, o próprio Unity já oferece uma forma de compensar esse tempo de cálculo de cada frame: a classe `Time` .

Usando Time para compensar o frame

Usando o atributo `deltaTime` da classe `Time` , podemos capturar quanto tempo demorou entre a exibição do frame atual e o anterior. E essa informação pode ser usada para tornar nossa velocidade independente da taxa de frames exibidos:

```
private float velocidade = 10;

void Update ()
{
    Vector3 posicaoAtual = transform.position;
    Vector3 frente = transform.forward;
    Vector3 deslocamento = frente * velocidade * Time.deltaTime;
    transform.position = posicaoAtual + deslocamento;
}
```

Disparando vários mísseis

Em vez de disparar apenas um único `Missil` , podemos fazer nossa `Torre` disparar mísseis em um intervalo fixo. Então, olhando o script da nossa `Torre` , já podemos fazer uma alteração logo de cara: colocar o comportamento de disparo logo no `Update` !

```
public class Torre : MonoBehaviour
{
    void Update ()
    {
        Atira ();
    }
}
```

Mas agora, a cada quanto tempo nossa `Torre` atira? Sempre que o método `Update` for chamado, o que é bastante rápido!

Para podermos parametrizar o tempo do disparo, vamos contar com o `Time.time` , que permite obtermos o tempo desde o início do jogo:

```
private float momentoDoUltimoDisparo;

public float tempoDeRecarga = 1f;

private void Atira ()
```

```
{
    float tempoAtual = Time.time;
    if (tempoAtual > momentoDoUltimoDisparo + tempoDeRecarga) {
        momentoDoUltimoDisparo = tempoAtual;

        // Dispara...
    }
}
```

Então, sempre que estiver passado um intervalo `tempoDeRecarga` entre o `tempoAtual` e o `momentoDoUltimoDisparo`, podemos disparar um míssil!

Mísseis teleguiados

Conseguimos disparar mísseis num intervalo fixo, porém nossos mísseis sempre caminham para a frente. Desse jeito munca seremos capazes de acertar algum inimigo.

Vamos fazer algo melhor: logo após serem disparados, nossos mísseis irão seguir o inimigo!

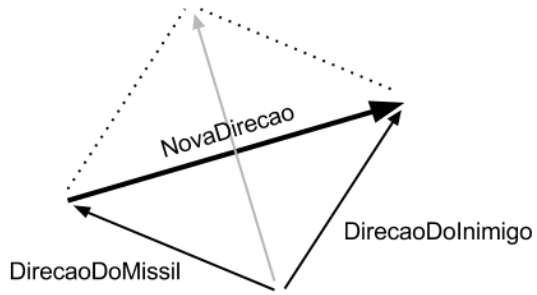
Como nosso `Missil` sempre anda para a frente, podemos a cada frame corrigir a sua rota para sempre apontá-lo para a posição atual do inimigo. Isso será feito num método chamado `AlteraDirecao` no próprio script do `Missil`:

```
void Update ()
{
    Anda ();
    AlteraDirecao ();
}
```

Nesse método `AlteraDirecao` precisamos fazer nosso `Missil` apontar para o alvo. Se ele sempre estiver apontando, ao caminhar para a frente, cada vez estará mais perto do alvo, dando a ideia de ser "teleguiado".

Mas como faremos o `Missil` sempre apontar para o alvo?

Podemos pegar o vetor que representa a direção atual do míssil e corrigí-lo para apontar para a direção do inimigo. Como ambos são vetores, podemos subtraí-los:



NovaDirecao = DirecaoDoInimigo - DirecaoDoMissil

Ao fazer essa subtração, o vetor resultante representa a nova direção do míssil, justamente a "correção" que estávamos procurando!

```
private void AlteraDirecao()
{
    Vector3 direcaoDoMissil = transform.position;
    Vector3 direcaoDoInimigo = alvo.transform.position;

    Vector3 novaDirecao = direcaoDoInimigo - direcaoDoMissil;
}
```

Temos a nova direção. Só precisamos fazer o `Missil` rotacionar para essa `novaDirecao`. Podemos fazer isso usando o método `LookRotation` da classe `Quaternion` atribuindo isso para a `rotation` do próprio `GameObject`:

```
transform.rotation = Quaternion.LookRotation (novaDirecao);
```