03

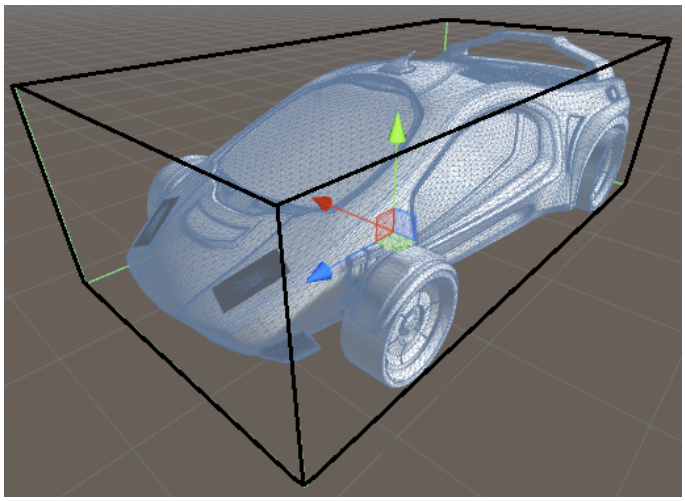
Detectando colisões e destruindo inimigos

Detectando colisões com Colliders

Já temos uma torre capaz de disparar mísseis mas o que deve acontecer quando um míssil atinge um inimigo?

A primeira coisa que precisamos fazer é detectar o momento em que ocorre uma colisão entre o míssil e o inimigo.

Quando trabalhamos com modelos 3D mais detalhados é interessante aproximar a forma do modelo utilizando formas simples como cubos e esferas que permitem um cálculo mais eficiente das colisões entre todos objetos de uma cena. Por exemplo, veja o modelo abaixo:



Note que o modelo é bastante detalhado e poderíamos até utilizá-lo para verificar as colisões mas isso teria um custo muito elevado de processamento no nosso jogo. Mesmo que estivéssemos fazendo uma simulação de corrida ultra-realista, não precisamos de um modelo tão detalhado para checar as colisões.

Poderíamos tratar cada pneu como um cilindro de poucas faces e o corpo do carro como um ou mais cubos. Se o nosso jogo fosse ainda mais simples, como nos antigos jogos de corrida de arcade, poderíamos até mesmo tratar o carro inteiro como um único cubo como na figura! Como o cálculo das colisões é realizado a cada novo quadro de nosso jogo, ganhamos um tempo precioso simplificando essa tarefa.

No Unity, esse modelo simplificado utilizado para o cálculo das colisões é chamado de *collider*.

Para adicionar um *collider* a um objeto existente, só temos que selecionar o objeto e no `Inspector`, clicar em `Add component -> Physics` e escolher um dos vários *colliders* disponíveis. Depois basta configurar os parâmetros do *collider* de forma a ajustá-lo à forma do modelo do seu *game object*. Note que os *game objects* do Unity que representam as formas primitivas já possuem os seus respectivos *colliders* adicionados.

Como estamos interessados em detectar a colisão entre o míssil e o inimigo, e como esses dois objetos foram criados usando formas primitivas então não vamos precisar adicionar os *colliders* manualmente.

Então agora só falta especificar o que devemos fazer quando uma colisão entre esses objetos ocorre. Primeiro vamos dizer para o Unity que estamos interessados em sermos avisados quando uma colisão ocorre.

Fazemos isso selecionando o nosso `Missil`, por exemplo, e no `Inspector -> Box Collider` marcamos a opção `Is Trigger`. Com isso estamos dizendo que a área desse *collider* funcionará como um gatilho que irá disparar um método do script do

`Missil` quando o objeto colidir com outro objeto que possua um *collider*.

Agora precisamos editar o script associado ao `Missil` e implementar o método `OnTriggerEnter` que será chamado quando ocorrer a colisão. Neste método vamos aproveitar e destruir o `Missil` invocando o método `Destroy` passando o próprio *game object* do míssil como parâmetro:

```
public class Missil : MonoBehaviour
{

    // Códigos anteriores

    void OnTriggerEnter (Collider elementoColidido)
    {
        Destroy (this.gameObject);
    }
}
```

Tendo especificado o que fazer em caso de uma colisão do míssil, o que acontece quando testamos o jogo e um míssil atinge um inimigo? Nada! O que será que aconteceu?

Para descobrir o que está acontecendo, precisamos entender um pouco melhor como o Unity lida com colisões.

Simulando física e colisões com Rigidbodies

No Unity, a detecção de colisão entre dois ou mais objetos é responsabilidade do simulador de física. Como nenhum objeto do Unity tem seus movimentos sujeitos às leis da física por padrão, então faz sentido que nossos objetos não sejam considerados na detecção de colisões.

Então precisamos dizer ao Unity para tratar nossos objetos como objetos físicos! Para que isso aconteça é necessário dizer que um *game object* possui o comportamento de um *Rigidbody*. Um *Rigidbody* terá seus movimentos controlados pelo simulador de física do Unity, levando em conta fatores como sua massa, gravidade, entre outros.

Sabendo disso, podemos fazer com que nosso `Missil` se comporte como um `Rigidbody` selecionando-o e clicando em `Inspector -> Add Component -> Physics -> Rigidbody .`

Mas se testarmos nosso jogo agora veremos que nossos mísseis caem imediatamente após terem sido disparados! Isso acontece porque agora ele está sofrendo a ação da gravidade!

Vamos então corrigir isso definindo que queremos que ele seja tratado como um objeto físico mas não queremos que ele fique sujeito a ação da gravidade. Podemos definir isso indo no `Inspector -> Rigidbody` e desmarcando a opção `Use Gravity .`

Agora sim nossos mísseis voltaram a ter o comportamento de antes!

Testando o jogo agora, veremos que a colisão é detectada e o míssil é destruído como esperávamos mas o que acontece se um míssil atingir sem querer o cenário?

Para chegarmos numa resposta, vamos selecionar o nosso objeto `Chao` e verificar os componentes desse objeto no `Inspector` . Temos os componentes `Transform` , `Terrain` e `Terrain Collider` ! Como esse objeto também tem um *collider*, o míssil também será destruído se colidir com o terreno!

Esse tipo de comportamento poderia tornar o nosso jogo mais realista mas dependendo da irregularidade do terreno, isso pode tornar o jogo difícil demais para o jogador. Nesse caso, seria interessante que o míssil ignorasse as colisões com o terreno... mas como poderíamos fazer isso?

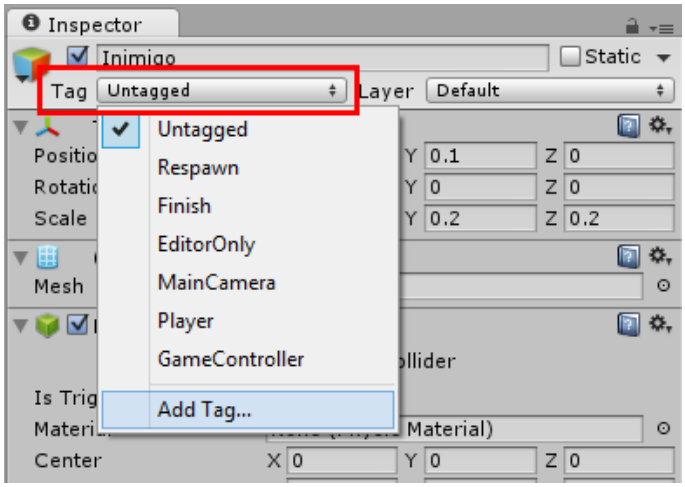
Categorizando nossos Game Objects com Tags

Se prestarmos atenção no nosso método `OnTriggerEnter` do script `Inimigo`, veremos que esse método recebe como parâmetro um `Collider`. Esse `Collider` representa o *collider* do outro objeto que colidiu com o nosso. Então, nessa hora poderíamos perguntar para esse `Collider` quem é o objeto associado a ele!

Então basta verificar se o nome do objeto com o qual colidimos é "Inimigo". Mas se implementarmos a verificação dessa forma, o que acontece se resolvermos alterar o nome do nosso inimigo? Teríamos que verificar todos os nossos scripts em buscas de referências ao nome antigo e substituí-lo pelo novo nome.

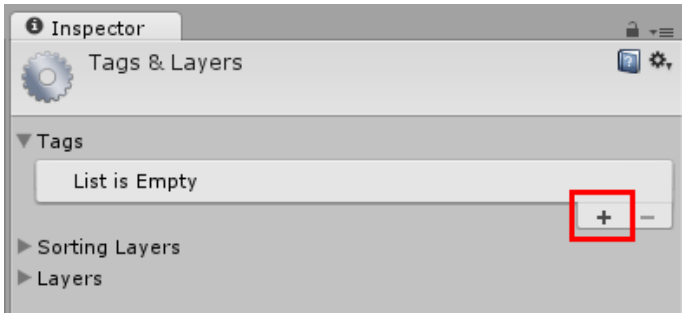
No Unity existe uma maneira melhor de fazer isso. Ao invés de comparar os objetos utilizando seus nomes, podemos associar um objeto com uma `Tag`. Uma `Tag` é como um rótulo que podemos colocar em nossos objetos para agrupá-los. Assim podemos verificar se um objeto possui ou não uma determinada `Tag` para realizar alguma ação.

Para criar uma nova `Tag`, selecionamos um objeto qualquer e no `Inspector` clicamos na caixa de seleção ao lado da palavra `Tag` indicada na figura abaixo:



Depois selecionamos a opção `Add Tag...`. Com isso, abriremos uma nova tela no `Inspector` com opções referentes aos recursos de `Tags` e `Layers`.

Em `Inspector -> Tags`, temos a lista de todas as *tags* que foram criadas no nosso projeto. Inicialmente, esta lista estará vazia e podemos criar uma nova *tag* clicando no botão `+` localizado no canto inferior esquerdo da lista como na figura abaixo:



Com isso adicionamos uma nova entrada `Tag 0` à lista de *tags*. Podemos alterar o nome dessa *tag* para algo que faça mais sentido no nosso caso, por exemplo, "Inimigo".

Agora já temos uma nova *tag* definida mas como associamos essa *tag* com o nosso inimigo? Para isso, vamos selecionar novamente o `Inimigo` e no `Inspector` selecionar `Tag = Inimigo`. Pronto! Nosso `Inimigo` agora pertence à categoria "Inimigo".

De volta ao script `Inimigo`, precisamos verificar se o objeto associado ao *collider* possui a *tag* "Inimigo". Para isso, utilizamos o método `CompareTag` do *game object* passando como parâmetro o nome da *tag* que queremos verificar:

```
public class Missil : MonoBehaviour
{

    // Códigos anteriores

    void OnTriggerEnter (Collider elementoColidido)
    {
        if (elementoColidido.CompareTag ("Inimigo"))
        {
            Destroy (this.gameObject);
        }
    }
}
```

Se testarmos o jogo agora, veremos que o `Missil` só é destruído quando colide com o `Inimigo`. O único problema é que por enquanto nosso inimigo está invencível!

Usando atributos para balancear o jogo

Podemos fazer com que o inimigo seja destruído assim que for atingido por um míssil mas isso tornaria o jogo fácil demais. Ao invés disso, vamos adicionar um atributo no nosso inimigo para marcar a quantidade de pontos de vida que ele possui. Em seguida, vamos fazer com que nossos mísseis causem uma certa quantidade de pontos de dano ao atingir o inimigo. Quando o inimigo não tiver mais pontos de vida, então podemos destruí-lo.

Começamos modificando o script `Inimigo` para adicionar um novo atributo. Para que possamos alterá-lo no editor do Unity, vamos tornar esse atributo público:

```
public class Inimigo : MonoBehaviour
{
    public int vida;
}
```

Isso funciona mas acabamos permitindo que qualquer objeto do nosso jogo também tenha acesso ao atributo `vida` do `Inimigo`. Podemos alterar o modificador de acesso do nosso atributo para `private` mas o que acontece se fizermos isso? O editor do Unity também não terá mais acesso a esse atributo!

Para manter o nosso atributo privado e ainda assim permitir que o editor tenha acesso a ele, podemos utilizar a anotação `[SerializeField]`. Dessa forma, podemos escrever nosso código assim:

```
public class Inimigo : MonoBehaviour
{
    [SerializeField] private int vida;
}
```

Utilizando essa mesma ideia, podemos alterar o script do `Missil` para indicar a quantidade de pontos de dano que ele causa:

```
public class Missil : MonoBehaviour
{
    [SerializeField] private int pontosDeDano;
}
```

Agora precisamos fazer com que o míssil reduza os pontos de vida do inimigo ao atingí-lo. Qual dos nossos scripts é responsável por agir quando uma colisão acontece? O script `Inimigo` ! Vamos alterar então o nosso método `OnTriggerEnter` para adicionar esse novo comportamento:

```
public class Missil : MonoBehaviour
{
    void OnTriggerEnter (Collider elementoColidido) {
        if (elementoColidido.CompareTag ("Inimigo"))
        {
            Destroy (this.gameObject);
            Inimigo inimigo = elementoColidido.GetComponent<Inimigo>();
            // aqui pedimos para o inimigo reduzir seus pontos de vida
        }
    }
}
```

No script acima, pedimos uma referência para o componente `Inimigo` do `elementoColidido` . Mas agora precisamos de alguma forma de alterar a quantidade de pontos de vida do inimigo. Como o atributo `vida` é privado, vamos ter que fornecer um método para deduzir uma certa quantidade de pontos de vida do inimigo:

```
public class Inimigo : MonoBehaviour
{
    public void RecebeDano(int pontosDeDano)
    {
        vida -= pontosDeDano;
        if (vida <= 0)
        {
            Destroy(this.gameObject);
        }
    }
}
```

Note que nesse script já aproveitamos e destruímos o inimigo caso seus pontos de vida cheguem a 0. Agora que temos esse script, basta invocá-lo no script do `Missil` :

```
public class Missil : MonoBehaviour
{
    void OnTriggerEnter (Collider elementoColidido) {
        if (elementoColidido.CompareTag ("Inimigo"))
        {
            Destroy (this.gameObject);
            Inimigo inimigo = elementoColidido.GetComponent<Inimigo>();
            inimigo.RecebeDano(pontosDeDano);
        }
    }
```

```
    }
```

Antes de testar o jogo, não podemos esquecer de definir os valores iniciais para os atributos `vida` e `pontos de dano`. Fazemos isso selecionando o objeto `Inimigo` e modificando o valor `Inspector -> Inimigo -> Vida` para `10`. A mesma coisa para o `Missil` alterando o valor de `Inspector -> Missil -> Pontos de Dano` para `2`.

Para certificar que tudo funciona como o esperado, podemos testar o jogo novamente verificando se agora os inimigos resistem ao impacto de alguns mísseis antes de serem destruídos.