10

GUI: mostrando informações na tela

Contando as vidas do jogador

Até este momento, nosso jogo já tem inimigos, que são gerados infinitamente, e também torres que podem ser construídas pelo jogador para destruir os inimigos. Mas o que acontece se o jogador decidir não construir nenhuma torre? Existe alguma motivação para o jogador destruir os inimigos? Por enquanto não! Talvez isso aconteça pelo fato de nossos inimigos serem inofensivos!

Devemos lembrar que o cenário do nosso jogo representa a última linha de defesa contra os inimigos invasores. Isso significa que se os inimigos não forem destruídos, algo ruim deve acontecer. Uma possibilidade seria encerrar imediatamente o jogo quando um inimigo conseguisse cruzar o caminho mas isso tornaria o jogo difícil demais.

Uma alternativa mais amigável seria permitir que o jogador pudesse falhar um número limitado de vezes. Dessa forma, ele conseguiria perceber que não destruir inimigos é algo ruim e que ele precisa construir as torres para evitar que o jogo acabe. Vamos então fazer nosso jogador começar o jogo com um número limitado de vidas.

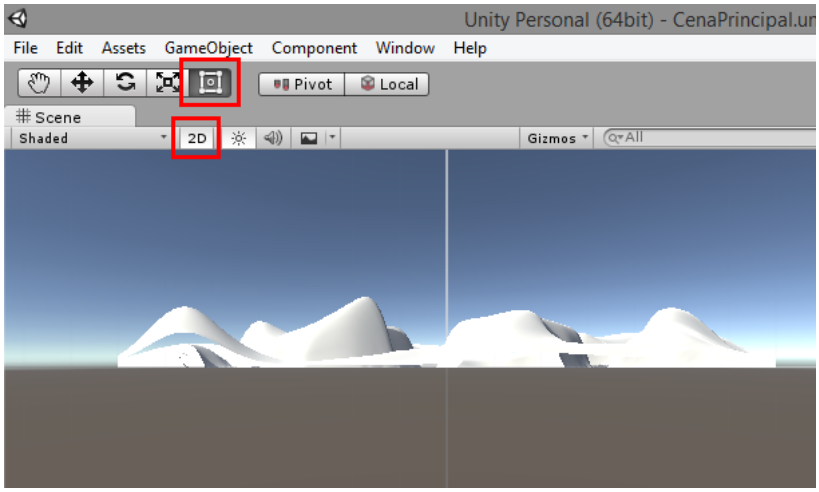
Então, como primeiro passo, podemos começar mostrando essa quantidade de vidas na tela do nosso jogo. Poderíamos criar um *game object* para apresentar esse dado na tela mas o que aconteceria se precisássemos mudar a posição da câmera? Teríamos que reposicionar esse *game object* para que ele continuasse visível para o jogador! E se tivéssemos vários *game objects* com essa função? Também teríamos que atualizar a posição de todos eles!

Como essa tarefa é algo bastante repetitivo e como apresentar dados para o usuário em uma interface gráfica é algo bastante comum, o Unity já possui um conjunto de *game objects* específicos para essa função.

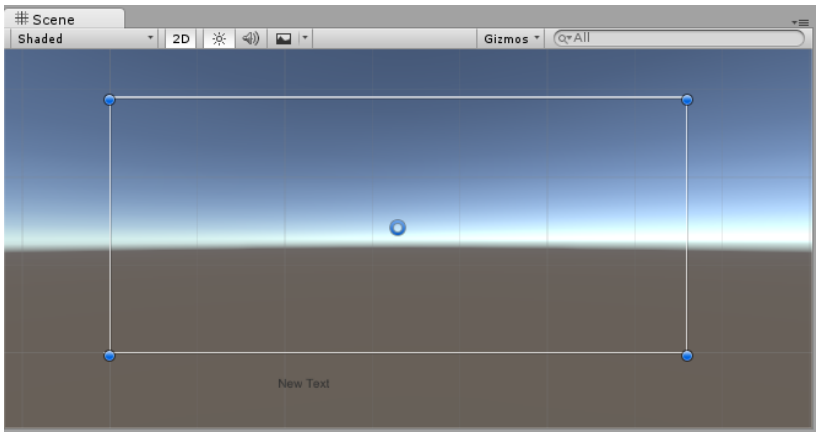
Interface gráfica do usuário

Para exibir um mostrador de vidas na tela do nosso jogo, podemos começar adicionando um *game object* **Text** à nossa cena. Na `Hierarchy`, podemos clicar em `Create -> UI -> Text`. Neste momento, o Unity irá adicionar automaticamente um objeto `EventSystem` e um `Canvas` que tem um `Text` como filho.

Antes de seguirmos em frente, vamos aprender como configurar a aba `Scene` para trabalhar com a interface gráfica. Na aba `Scene`, clique no botão `2D` e na ferramenta `Rect Handles` como indicado na figura abaixo:



Depois, selecione o `Canvas` no `Hierarchy`. Posicione o cursor do mouse sobre a aba `Scene` e pressione a tecla `F` para centralizar a cena no nosso `Canvas`. Em seguida, ajuste o *zoom* da cena com a rodinha do mouse até que todo o `Canvas` esteja visível. Sua visualização deverá ficar parecida com a da figura abaixo:



Agora que ficou mais fácil de visualizar a nossa interface gráfica, vamos entender o que significam os objetos que o Unity criou na nossa cena.

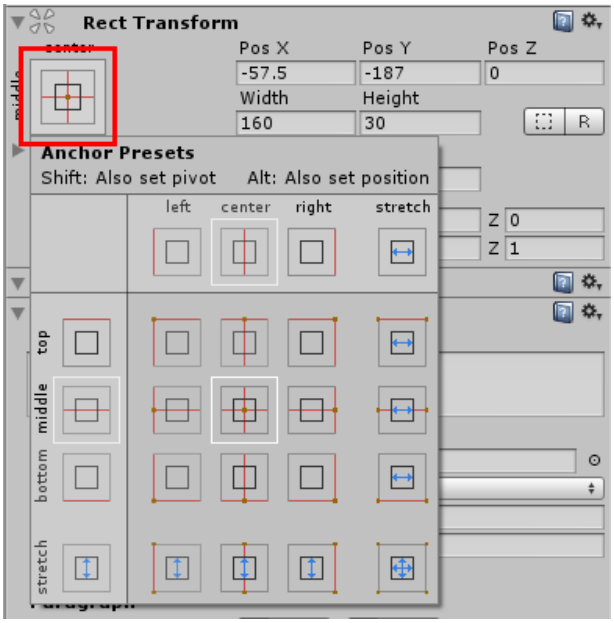
O `Canvas` representa a interface gráfica do nosso jogo e qualquer componente da categoria `UI` deve ser obrigatoriamente colocado dentro do `Canvas` para ser apresentado na tela.

O objeto `Text` representa um texto a ser exibido na interface gráfica. Se selecionarmos esse objeto, veremos que ele difere dos *game objects* que utilizamos até agora porque ele não possui um componente `Transform`. No lugar desse componente, ele possui um componente similar chamado de `Rect Transform`.

Rect Transform

O componente `RectTransform` permite que os objetos da interface gráfica sejam posicionados de forma relativa dentro do `Canvas`. Isso é bem interessante pois assim fazemos com que a posição dos objetos não dependa de um formato de tela ou resolução específica.

Por exemplo, nosso mostrador de vidas pode ser posicionado no canto superior esquerdo do `Canvas`. Para fazer isso, podemos utilizar a ferramenta `Anchor Presets` do `Rect Transform` mostrada abaixo:



Nesta ferramenta, vamos clicar na combinação correspondente a `Top/Left` para indicar que nosso objeto deve ficar ancorado no canto superior esquerdo do `Canvas`. Mas note que, por enquanto, o objeto `Text` continua na mesma posição.

Para alterar a sua posição, vamos editar no `Inspector -> Rect Transform` os campos `Pos X = 0` e `Pos Y = 0`. O que aconteceu ao fazer essas alterações? O centro do nosso objeto foi posicionado na coordenada (0, 0) relativa à âncora que definimos anteriormente!

Mas seria mais interessante se pudéssemos posicionar o canto superior esquerdo do nosso objeto na posição (0, 0) da âncora ao invés do seu centro. O Unity fornece recursos para isso, só precisamos modificar a posição do pivô do nosso `Text`. Isso pode ser feito alterando os valores `Pivot X = 0` e `Pivot Y = 1` no `Rect Transform`.

Agora sim, vamos alterar novamente a posição do `Text`, fazendo `Pos X = 20` e `Pos Y = -10`. Note que agora estamos posicionando o nosso objeto a 20 *pixels* de distância da margem esquerda e 10 *pixels* de distância da margem superior.

O componente Text

Quando selecionamos o objeto `Text`, vemos que ele possui também um componente chamado `Text`. Neste componente podemos alterar o texto exibido pelo objeto, os parâmetros da fonte, alinhamento, entre outros.

Para tornar o nosso mostrador de vidas mais vísivel, em `Inspector -> Text` vamos alterar `Text = "Vida :"`, `Font Size = 20`, `Font Style = Bold` e `Color` para branco. Assim temos uma fonte maior em negrito e na cor branca para destacá-lo em relação ao fundo do nosso jogo.

Se testarmos o jogo agora, veremos que o mostrador de vidas já aparece mas ele ainda não mostra a quantidade de vidas do jogador. Para que isso aconteça vamos ter que definir um novo comportamento para nosso mostrador de vidas.

Antes disso vamos renomear os nossos objetos `Canvas` para `GUI` e `Text` para `MostradorDeVidas`. Agora sim, vamos adicionar um novo script no nosso objeto `MostradorDeVidas` que ficará responsável por atualizar a quantidade de vidas mostradas a cada novo quadro do jogo:

```
using UnityEngine.UI;

public class MostradorDeVida : MonoBehaviour
{
    private Text campoTexto;

    void Start ()
    {
        campoTexto = GetComponent<Text> ();
    }

    void Update ()
    {
        campoTexto.text = //como passaremos a vida do jogador aqui?
    }
}
```

Agora esbarramos em outro problema: onde vamos guardar a quantidade de vidas do jogador? Podemos usar um atributo associado a algum *game object* da nossa cena. Mas em qual dos nossos objetos faz sentido guardar esse atributo? Vamos criar um novo *game object* de nome `DadosDoJogador` para guardar os dados do nosso jogador!

Nesse objeto vamos adicionar um script `Jogador`:

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

    public int GetVida () {
        return vida;
    }
}
```

Como vamos precisar de uma referência do `Jogador` no nosso script `MostradorDeVida`, vamos adicionar um novo atributo nesse script e atualizar o mostrador de vidas com a quantidade de vidas do jogador:

```
using UnityEngine.UI;

public class MostradorDeVida : MonoBehaviour
{
    private Text campoTexto;
    public Jogador jogador;

    void Start ()
    {
        campoTexto = GetComponent<Text> ();
    }

    void Update ()
    {
        campoTexto.text = "Vida: " + jogador.GetVida ();
    }
}
```

Antes de testar o jogo precisamos injetar a referência do `Jogador` no nosso `MostradorDeVida`. Fazemos isso selecionando o `MostradorDeVidas` e arrastando o objeto `DadosDoJogador` para `Inspector -> Mostrador de Vidas -> Jogador`. Além disso, vamos selecionar o `DadosDoJogador` e em `Inspector -> Jogador` alterar `Vida = 10`.

Pronto! Já podemos testar o jogo e verificar que o mostrador de vidas apresenta a quantidade de vidas do jogador. Podemos ainda testar se o mostrador se altera se modificarmos as vidas do jogador. Basta selecionar o `DadosDoJogador` e alterar o valor do atributo `Vida`. Perceba que o mostrador se modifica automaticamente para refletir o novo valor!

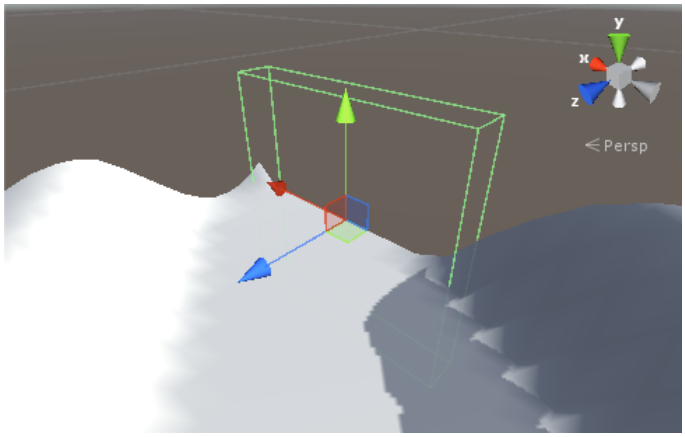
Detectando inimigos no fim do caminho

Com o mostrador de vidas funcionando, agora só precisamos deduzir as vidas do jogador sempre que algum inimigo alcançar o fim do caminho. Mas como faremos para descobrir se um inimigo terminou o caminho? Podemos utilizar um *collider*!

A ideia é colocar uma parede invisível no final do caminho e fazer o jogador perder uma vida sempre que algum inimigo colidir com essa parede. Como já temos um objeto `FimDoCaminho`, só precisamos adicionar um *collider* a esse objeto e um script para tratar a colisão com os inimigos.

Para adicionar o *collider*, basta selecionar o `FimDoCaminho` e clicar em `Inspector -> Add Component -> Physics -> Box Collider`. Em seguida, modificamos os atributos desse *collider* para que todo inimigo colida com ele obrigatoriamente ao

chegar no fim do caminho. Por exemplo, podemos ter um *collider* como na figura abaixo:



Como queremos que o `FimDoCaminho` seja informado quando algo colidir com ele, vamos marcar a opção `Is Trigger` no `Inspector -> Box Collider`.

Antes de implementar de fato o comportamento do `OnTriggerEnter`, vamos pedir para o Unity imprimir uma mensagem avisando toda vez que esse método for invocado. Para fazer isso podemos utilizar o método abaixo:

```
Debug.Log("Minha mensagem de teste!");
```

Durante a execução do jogo, essas mensagens serão apresentas na janela `Console` do Unity. Para abrir essa janela, basta ir no menu `Window -> Console`.

Então agora vamos criar um script `DetectorDeCruzamento` no `FimDoCaminho` e implementar o método `OnTriggerEnter` mostrando uma mensagem avisando que alguém chegou no fim do caminho:

```
public class DetectorDeCruzamento : MonoBehaviour
{
    void OnTriggerEnter (Collider collider)
    {
        Debug.Log ("Chegou no fim do caminho!");
    }
}
```

Se rodássemos nosso jogo e agora aguardássemos algum inimigo chegar ao fim do caminho, deveríamos receber uma mensagem "Chegou no fim do caminho!" na janela do `Console` mas isso ainda não acontece... por que?

Lembre-se que a detecção de colisões no Unity é feita pela simulação da física e que somente objetos considerados nessa simulação terão suas colisões detectadas. Como nosso `Inimigo` não possui um componente `Rigidbody`, então ele não é capaz de gerar eventos de colisão!

Podemos corrigir isso facilmente selecionando o `Project/Prefabs/Inimigo` e clicando em `Inspector -> Add Component -> Physics -> Rigidbody`.

Agora estamos prontos para testar novamente e dessa vez veremos que as mensagens são apresentadas corretamente!

Penalizando o jogador

Já que agora sabemos quando os inimigos chegam ao fim do caminho, podemos penalizar o jogador reduzindo o número de vidas toda vez que isso acontecer.

Uma vez que um inimigo alcança o fim do caminho, ele não tem mais função no nosso jogo, portanto podemos destruí-lo. Para fazer isso, só precisamos destruir o objeto associado ao *collider* recebido pelo método *OnTriggerEnter* no `DetectorDeCruzamento`:

```
void OnTriggerEnter (Collider collider)
{
    Debug.Log ("Chegou no fim do caminho!");
    Destroy (collider.gameObject);
}
```

Depois dessa alteração o que aconteceria se um míssil atingisse o *collider* do `FimDoCaminho`? Ele também seria destruído! Isso porque não verificamos que tipo de objeto colidiu com o `FimDoCaminho`. Vamos corrigir esse detalhe utilizando novamente o recurso das *tags*!

Como nosso `Inimigo` já possui uma *tag* "Inimigo", basta verificar se o objeto associado ao *collider* possui essa *tag*:

```
public class DetectorDeCruzamento : MonoBehaviour
{
    void OnTriggerEnter (Collider collider)
    {
        if (collider.CompareTag ("Inimigo"))
        {
            Debug.Log ("Chegou no fim do caminho!");
            Destroy (collider.gameObject);
        }
    }
}
```

Para completar a implementação desse script, agora só precisamos fazer o jogador perder uma vida. Resolvemos isso adicionando um atributo referenciando o `Jogador` e depois invocamos um método `PerdeVida` no próprio jogador:

```
public class DetectorDeCruzamento : MonoBehaviour
{
    [SerializeField] private Jogador jogador;

    void OnTriggerEnter (Collider collider)
    {
        if (collider.CompareTag ("Inimigo"))
        {
            Debug.Log ("Chegou no fim do caminho!");
            Destroy (collider.gameObject);
            jogador.PerdeVida ();
        }
    }
}
```

Como estamos invocando um método que ainda não existe no `Jogador`, vamos implementá-lo agora:

```
public class Jogador : MonoBehaviour
{
    public void PerdeVida ()
    {
        vida --;
    }
}
```

Não podemos esquecer de injetar a referência para o `Jogador` no `FimDoCaminho`. Vamos selecionar o `FimDoCaminho` e arrastar o `DadosDoJogador` para `Inspector -> Detector De Cruzamento -> Jogador`.

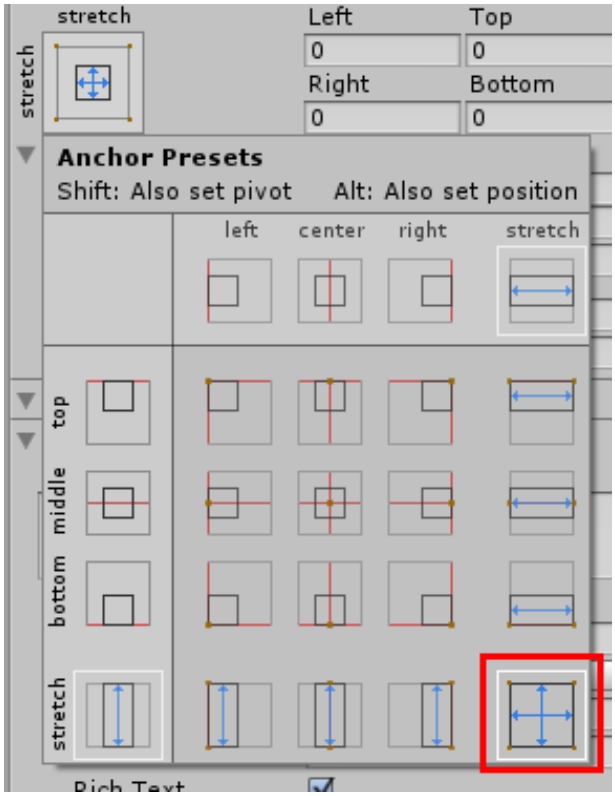
Ao testar o jogo novamente, notamos que agora as vidas são deduzidas como esperávamos!

Fim de jogo

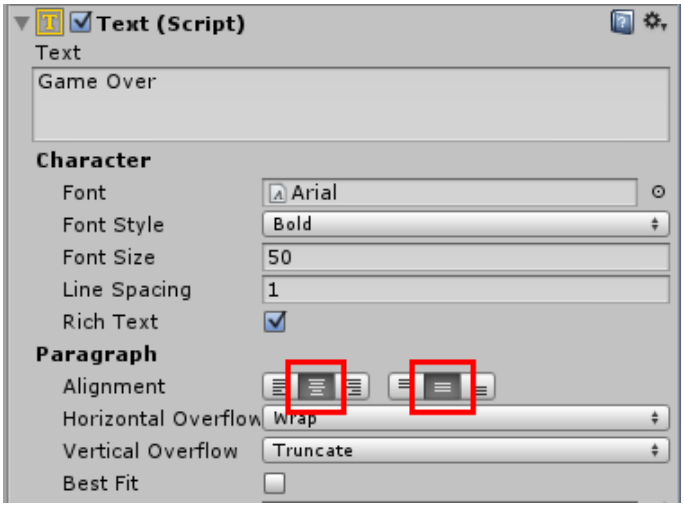
Agora que é possível perder vidas em nosso jogo, o que deveria acontecer quando essas vidas acabassem? Nesse caso, precisamos terminar o jogo e apresentar uma mensagem para o jogador.

Para exibir a mensagem de fim de jogo, vamos adicionar um novo objeto `Text` dentro do nosso objeto `GUI`. Podemos clicar com o botão direito em `GUI` e selecionar `UI -> Text`. Renomeamos esse objeto para `GameOver` para facilitar a identificação desse objeto.

Em seguida, vamos posicionar esse objeto no centro da nossa tela. Para fazer isso, selecionamos o `GameOver` e utilizamos a ferramenta `Anchor Presets`. Nessa ferramenta, seguramos a tecla `ALT` e escolhemos as opções `Stretch Horizontal & Vertical` para que o nosso objeto ocupe a tela inteira. Essa opção está indicada na figura abaixo:



Vamos também mudar alguns parâmetros para tornar nossa mensagem mais visível. Ao selecionar o `GameOver`, podemos aumentar o tamanho da fonte e mudar a sua cor para vermelho, por exemplo. Além disso, vamos alinhar o texto horizontalmente e verticalmente clicando nos botões indicados na figura abaixo:



Dependendo da cor do nosso texto e do nosso cenário pode ser que o texto ainda não esteja bem visível. Para melhor isso um pouco mais, vamos adicionar uma sombra nesse objeto. No `Inspector`, clicamos em `Add Component -> UI -> Effects -> Shadow`. Agora só precisamos ajustar os atributos `X` e `Y` do componente `Shadow` para posicionar a sombra como quisermos.

Mostrando o Game Over na hora certa

Da maneira como criamos a mensagem de "Game Over", ela estará visível desde o início do jogo. Precisamos ocultar essa mensagem e mostrá-la somente quando o jogador não tiver mais vidas.

Quando precisamos mostrar ou ocultar um *game object* no Unity, podemos utilizar o método `SetActive` como no exemplo abaixo:

```
meuObjeto.SetActive(false); // oculta o meuObjeto
meuObjeto.SetActive(true);  // exibe o meuObjeto
```

Então, no início do jogo, podemos ocultar o objeto `GameOver`. Como isso é um comportamento do nosso jogo, vamos implementá-lo no script `Jogo`:

```
public class Jogo : MonoBehaviour
{
    [SerializeField] private GameObject gameOver;

    void Start ()
    {
        gameOver.SetActive(false);
    }

    //Códigos anteriores...
}
```

Vamos aproveitar e já injetar a referência do objeto `GameOver` arrastando-o para o atributo `Game Over` do nosso objeto `Jogo`.

Com isso, já não mostramos mais a mensagem de "Game Over" quando o jogo se inicia. Mas agora que fizemos isso, o que deveria acontecer quando as vidas acabassem? Precisamos tornar vísivel novamente a mensagem de "Game Over"!

Encerrar o jogo quando as vidas acabam é uma das regras do nosso jogo. Por esse motivo, vamos colocar esse comportamento no nosso script `Jogo`!

Primeiro vamos criar um método `JogoAcabou` que ficará responsável por checar as vidas do jogador e informar se o jogo deve ser encerrado ou não:

```
public class Jogo : MonoBehaviour
{
    [SerializeField] private Jogador jogador;

    private bool JogoAcabou ()
    {
        return !jogador.EstaVivo ();
    }
}
```

Note que estamos usando um método do `Jogador` que ainda não existe. Vamos implementá-lo:

```
public class Jogador : MonoBehaviour
{
    public bool EstaVivo ()
    {
        return vida > 0;
    }
}
```

Agora que já temos esse dois métodos prontos, vamos retornar ao script `Jogo` e utilizar o método `JogoAcabou` para decidir entre continuar o jogo normalmente e encerrá-lo mostrando a mensagem de "Game Over":

```
public class Jogo : MonoBehaviour
{
    void Update ()
    {
        if (JogoAcabou ())
        {
            gameOver.SetActive(true);
        } else
        {
            if (ClicouComBotaoPrimario ())
            {
                ConstroiTorre ();
            }
        }
    }
}
```

Reiniciando o jogo com Button

Nosso jogo agora tem fim mas o jogador não consegue reiniciá-lo facilmente. Para fornecer essa opção para o jogador, vamos adicionar um botão na interface gráfica.

Como esse botão deve aparecer sempre acompanhado da mensagem de "Game Over", vamos criá-lo como um filho do `GameOver`. Dessa forma, quando exibirmos ou ocultarmos o `GameOver`, estaremos fazendo o mesmo com o botão.

Vamos então clicar com o botão direito em `Hierarchy/GUI/GameOver` e selecionar `UI -> Button`. Em seguida, renomeamos esse botão para `JogarNovamente`.

Para ajustar os parâmetros desse botão, vamos selecioná-lo. No `Inspector -> Text` alteramos `Text = "Jogar Novamente"`. Além disso, vamos também posicionar o botão logo abaixo da mensagem de "Game Over".

Se testarmos o jogo agora, veremos que a mensagem de "Game Over" é apresentada juntamente com o botão de "Jogar Novamente" quando as vidas acabam.

Recarregando a cena

Agora temos um botão para reiniciar o jogo mas ainda não fazemos nada quando clicamos nele. O esperado seria resetar o jogo para seu estado inicial mas como poderíamos fazer isso? Uma possibilidade seria remover todos os inimigos instanciados, todas as torres, aumentar a quantidade de vidas do jogador e ocultar a mensagem de "Game Over". Para o nosso jogo, isso não seria muito trabalhoso mas e se estivéssemos trabalhando num jogo mais complexo?

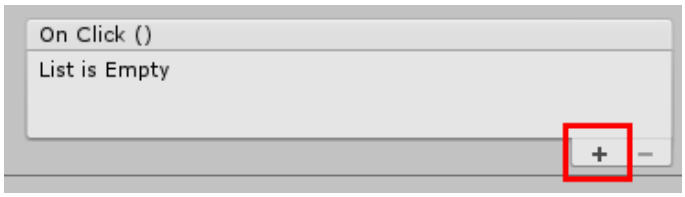
O que gostaríamos de fazer é simplesmente retornar a nossa cena para seu estado inicial. O Unity possui um método `Application.LoadLevel` que pode ser utilizado para carregar uma cena qualquer do nosso jogo. Com esse método, podemos carregar a mesma cena em que estamos restaurando assim o estado inicial da cena.

Vamos utilizar essa ideia para implementar um método que reinicia o nosso jogo. No script `Jogo` vamos adicionar:

```
public class Jogo : MonoBehaviour
{
    public void RecomecaJogo ()
    {
        Application.LoadLevel (Application.loadedLevel);
    }
}
```

Repare que nesse script invocamos o `Application.LoadLevel` passando como parâmetro `Application.loadedLevel` que representa a cena atual do jogo.

Agora só precisamos invocar esse método quando clicarmos no nosso botão `JogarNovamente`. Para fazer isso, basta selecionar o botão e no `Inspector -> Button`, clicar no botão `+` indicado na figura abaixo para adicionar um novo evento de clique.



Depois vamos associar o nosso método `RecomecaJogo` com esse evento arrastando o objeto `Jogo` para o campo `None` e em seguida alterar o campo `No Function` para `Jogo -> RecomecaJogo`.

Então agora nosso jogo já possui fim e permite que o jogador inicie uma nova partida clicando no botão de "Jogar Novamente"!

Rodando o jogo em diversas plataformas

Até o momento, nosso jogo está rodando apenas no próprio ambiente do Unity, que é ótimo para usarmos durante a fase de desenvolvimento. Porém, conforme vamos evoluindo nossa criação, precisaremos testá-lo em outras plataformas.

Se quisermos nosso jogo rodando na web, no Android e no iOS, por exemplo, será que teremos que fazer um código novo para cada universo?

É neste momento que o Unity mostra uma das suas grandes vantagens: todo projeto no Unity pode ser exportado para as principais plataformas do mercado sem grandes impactos no nosso código! Basta adicionarmos os módulos de exportação.

Dentre as opções de exportação, temos Web, *standalone*, Android, iOS, Windows Phone 8, XBox 360 e PS4. Alguns desses módulos possuem licensa paga.

Vamos ver como isso funciona exportando nosso jogo para a plataforma Android, que pode ser usado de forma gratuita.

Exportando para Android

Para sermos capazes de exportar nosso jogo para o Android, precisamos baixar o seu kit de desenvolvimento, que é o mesmo usado para criar aplicativos nativos para Android.

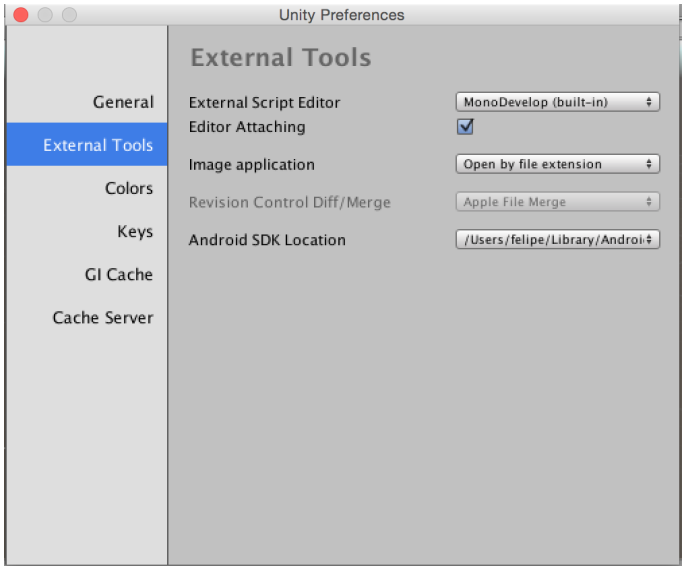
Preparando o ambiente em casa

Caso queira montar o ambiente na sua casa, basta baixar as ferramentas de desenvolvimento do Android (http://developer.android.com/sdk/installing/index.html?pkg=tools).

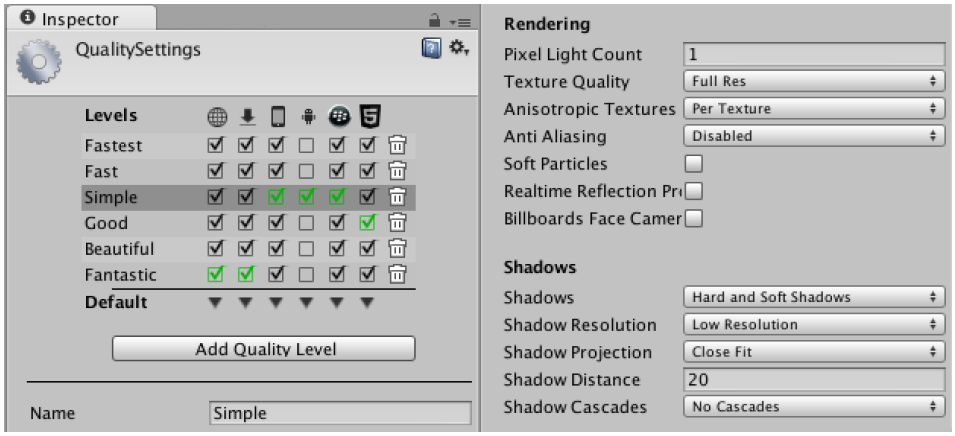
Na sequência, precisaremos baixar o SDK mais recente, de acordo com as instruções da própria documentação do Android (http://developer.android.com/sdk/installing/adding-packages.html).

Caso você já tenha o Android Studio na sua máquina, você já possui o ambiente necessário para o Unity!

Com o SDK baixado, só precisamos apontá-lo no Unity no menu `Edit -> Preferences... -> External Tools`:



Quando exportamos para um aparelho móvel, temos que tomar cuidado com os requisitos mínimos para que nosso jogo rode corretamente e sem lentidão. Podemos adequar essa qualidade gráfica do nosso projeto em `Edit -> Project Settings -> Quality`:



Com essas configurações, podemos ir em `File -> Build Settings -> Build And Run` :