Desenvolvimento com Motores de Jogos I

Aula 14 - Criação de Elementos em Tempo de Execução, Dano e Elementos Coletáveis

Apresentação

Fala, pessoal! Estamos quase no final de nossa disciplina e já temos um jogo bem legal desenvolvido! Mas será que ainda falta alguma coisa? Na verdade, faltam várias! E infelizmente nem teremos tempo de ver tudo nesta disciplina. Mas conseguimos adiantar ainda mais algumas coisas que vocês poderão replicar em qualquer um de seus futuros projetos, com base no estudado aqui!

A primeira novidade da aula de hoje será a instanciação de elementos em tempo de execução. Você já deve ter visto objetos, os quais não eram partes de uma cena, surgirem em um jogo à medida que algo acontece, não? E é justamente esse comportamento que aprenderemos a adicionar agora em nosso jogo! Criaremos o Prefab de um míssil e, em seguida, uma bazuca capaz de atirá-lo em um intervalo aleatório, criando, assim, um desafio a mais para o nosso jogador!

Mas só terá graça mesmo se esses objetos causarem algum dano no jogador, utilizando a barra de energia desenvolvida na aula passada, certo? Trabalharemos isso ainda nesta aula! Como podemos adicionar (e mostrar) o dano em nossa cena? É o que também veremos ao longo da aula! Cuide-se, robozinho!

Para finalizar, ajudaremos um pouco o robô. Utilizaremos o conceito de elementos coletáveis, como as moedinhas do Mário ou mesmo os Power Ups do Megaman, a fim de criar um pacote de recuperação de vida para o nosso personagem. Esse pacote, assim como os itens citados, será coletado e sumirá logo após cumprir a sua função! Isso não deve ser tão difícil com o que já aprendemos, não é?

A aula de hoje funcionará como uma grande revisão, pois serão apresentados conceitos de mais alto nível! Caso não compreenda algo, fique à vontade para entrar nos fóruns e falar conosco. Estaremos aguardando! E vamos à aula!

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Criar objetos em tempo de execução no Unity;
- Criar elementos coletáveis em sua cena;
- Utilizar o sistema de mensagens do Unity;
- Trabalhar com o conceito de dano.

1. Criação Dinâmica de Objetos

É muito comum, em jogos digitais, haver alguns objetos que são criados durante a execução do jogo, com base em algum evento ou até mesmo em algum contador de tempo (também conhecido como timer). Na primeira parte da aula de hoje, aprenderemos como criar objetos dinamicamente, em tempo de execução, para os nossos níveis. Adicionaremos à nossa segunda fase uma pequena variação em relação à primeira, transformando uma das caixas em uma bazuca mortal, atirando mísseis em uma direção predefinida. Esses mísseis causarão dano ao personagem se fizerem contato com ele.

A fim de ser possível o usuário saber quanta energia ainda lhe resta mesmo após um impacto com o míssil, precisaremos, ainda, atualizar a barra de energia construída em aulas anteriores. Isso será feito a partir do próprio Player, à medida que ele registrar o dano tomado, seja qual for a fonte. Logo de início, deixamos, novamente, o aqui para o acesso ao projeto desenvolvido até agora. Precisaremos também da Sprite Sheet do míssil, para que ele seja animado adequadamente. Ela está disponível aqui.

Atenção!

Como já estamos chegando ao final de nossa disciplina, retomaremos diversos conceitos que vimos anteriormente e faremos uma grande revisão, sem nos atermos aos detalhes sobre processos já vistos! Caso algo não pareça claro, volte a uma das aulas anteriores e veja o passo a passo do processo. Fique à vontade, sempre, para utilizar o fórum caso a sua dúvida não seja sanada! Estaremos lá para ajudar.

1.1 Criando o Prefab do Míssil

O primeiro passo para a criação de elementos dinamicamente no Unity é criar um Prefab que represente o elemento a ser criado. Definiremos um Prefab para o míssil, com todos os seus componentes, e, então, a partir desse Prefab, criaremos

um segundo componente capaz de instanciá-lo em tempo de execução. A ideia é simples! Vejamos como fazer!

Para começar, importaremos a Sprite Sheet que disponibilizamos do míssil. Essa Sprite Sheet é um pouco diferente das outras, pois as imagens do míssil são quadradas. Com isso, a Sprite Sheet ficou em 4 x 4, com o tamanho de 500 x 500 para cada um dos sprites envolvidos.

Após importar a Sprite Sheet para a pasta sprites de nosso projeto, devemos configurá-la para o modo Multiple e, em seguida, fatiá-la em pedaços de 500 x 500. Isso nos dará um total de dezesseis sprites diferentes que serão criados a partir dela. Utilizaremos os dois primeiros sprites para a animação de movimento e os quatorze últimos para a animação de explosão.

Com a Sprite Sheet fatiada, podemos começar a criação do elemento em si, como parte da cena. Utilizaremos o primeiro sprite da Sprite Sheet como referência para a criação do elemento. Arraste-o para a cena, criando um novo objeto com o nome missile_sprite_sheet_0. Renomeie o objeto para Missile e altere logo a sua Order in Layer para 1. Esse objeto será utilizado como a base para a criação de nosso Prefab. Iremos configurá-lo adequadamente e, em seguida, geraremos o Prefab a partir disso e o deletaremos, para não ficarmos com uma instância órfã na cena.

A primeira alteração a ser feita no objeto é adicionar a ele um Box Collider 2D, para podermos detectar a sua colisão com outros objetos da cena. Perceba, no entanto, que, ao adicionarmos o Box Collider 2D, ele ocupará toda a área do sprite. Isso geraria colisões terríveis! Editamos, então, o colisor para ele cobrir apenas o míssil em si, ignorando a parte transparente da sprite. Os valores adequados para isso ,encontrados no Box Collider 2D, foram Size X = 2 e Y = 1.25. Isso fará com que a colisão seja um pouco mais precisa. Além disso, é necessário alterar o colisor para Trigger, pois ele apenas deve detectar colisões e não realmente simulá-las fisicamente. Marque a opção Is Trigger.

Precisaremos, ainda, de um RigidBody 2D para realizar as simulações de física a partir dele. Esse corpo, no entanto, não precisa responder à gravidade ou se preocupar com qualquer arrasto. Toda a movimentação dele acontecerá via script, através da alteração de sua velocidade. Por esse motivo, devemos alterar o seu tipo para Kinematic. Nós o referenciaremos no script a ser desenvolvido.

Necessitamos apenas de mais dois componentes para finalizarmos o nosso objeto e podermos criar o Prefab: o script e o Animator. Como o script trará um pouco de código novo e lidará diretamente com o Animator, começaremos adicionando o Animator. Já utilizamos diversos assuntos vistos em nossas aulas anteriores. Vamos a mais um?

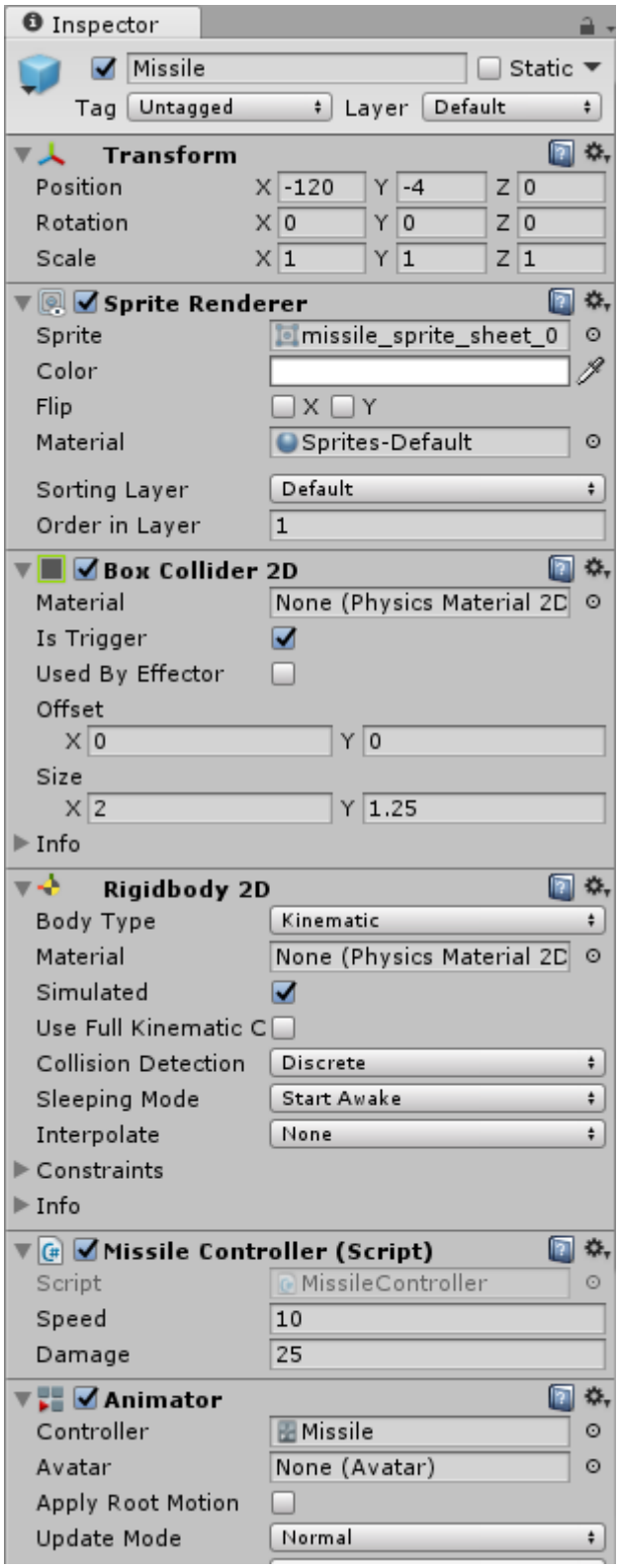
Para começar o processo de animação, abriremos as nossas abas de Animator e Animation. Em seguida, adicionaremos ao nosso objeto um Miscellaneous -> Animator e criamos para ele, dentro da pasta Animations, um novo Animator Controller. Chamaremos esse Animator Controller de MissileController. Atribua esse controlador ao Animator do objeto Missile.

Em seguida, precisamos criar as animações para que o míssil possa se mover e explodir adequadamente. Selecione o componente, vá até a aba Animation e crie uma nova animação, na pasta Animations, chamada Flying. Essa animação representará o míssil voando e será composta pelos dois primeiros sprites de nossa Sprite Sheet já fatiada anteriormente. Clique e arraste esses dois scripts para a timeline a fim de definir a nova animação. Como é uma animação de apenas dois frames, podemos definir uma quantidade de Samples baixa para ela. Sugerimos utilizar o valor 8 para o número de Samples.

O próximo passo é a animação de explosão. Criaremos uma nova animação para esse componente com os quatorze sprites que não foram utilizados para a primeira animação, a da Sprite Sheet do míssil. Chame essa nova animação de Exploding e salve-a na pasta Animations, como já estamos acostumados. Utilize o valor 14 para a quantidade de samples. Por fim, vá até o Animator Controller criado e confirme que as duas animações estão lá, sendo a Flying a animação padrão, marcada de laranja. Com isso, nosso míssil já possui todos os componentes necessários, faltando apenas adicionar o script para gerenciar o seu movimento.

Crie um novo script para o míssil, chamado MissileController, e o salve na pasta Scripts. A partir daí, o nosso objeto já estará pronto para ser gerado como Prefab. As configurações finais que fizemos nessa primeira etapa do desenvolvimento podem ser vistas na **Figura 1**. Atente se está tudo similar!

Figura 01 - Configuração dos componentes do objeto Missile, que será criado dinamicamente em nossa cena.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 20 de mar de 2017

Configurado o componente, basta arrastá-lo para a nossa pasta Prefabs, a fim de criar o asset a partir do GameObject, como já fizemos anteriormente. Feito isso, estamos prontos para seguir adiante.

Perceba que o componente script, demonstrado na **Figura 1**, já dá alguns spoilers do que faremos adiante. Então, antes de eu contar mais, vamos a ele? **Listagem 1**, contendo o script completo do MissileController!

```
using UnityEngine;
using System.Collections;

public class MissileController : MonoBehaviour {

    public float speed;
    public float damage;

    private Animator ani;

    // Use this for initialization
    void Start () {
        ani = gameObject.GetComponent<Animator>();
    }

    // Update is called once per frame
    void Update () {
        transform.GetComponent<Rigidbody2D>().velocity = Vector2.right * speed;
    }

    void OnTriggerEnter2D (Collider2D col) {
        ani.Play("Exploding");
        transform.GetComponent<Rigidbody2D>().simulated = false;

        if (col.CompareTag ("Player")) {
            if (col.gameObject.GetComponent<PlayerController>().HealthChange(-damage) <= 0)
                transform.parent.GetComponent<ShooterScript>().Stop();
        }

        Invoke("Remove", 1f);
    }

    void StopMissile () {
        transform.GetComponent<Rigidbody2D>().simulated = false;
    }

    void Remove () {
        Destroy(gameObject);
    }

}
```

Listagem 1 - Código completo do script MissileController, utilizado no objeto
Fonte: Elaborada pelo autor

Esse código é bem simples, contendo, em geral, apenas coisas já vistas. Mas vamos analisá-lo por partes! Começamos o código criando duas variáveis do tipo ponto flutuante – speed e damage. A variável speed indicará a velocidade que esse objeto se moverá. A variável damage, por sua vez, indicará o dano causado em nosso personagem quando entrar em contato com ele. Criamos esses dois campos

públicos para poderem ser editados diretamente pelo editor ou também pelo objeto que criá-lo, pois pode ser interessante alterar esses campos para cada instância. A última variável criada é a variável ani, uma referência ao Animator do nosso objeto.

O nosso método Start é bem simples e apenas faz uma busca, em nosso objeto, pelo componente Animator, para inicializar a variável ani, criada anteriormente.

O método Update também é simples e tem apenas uma linha alterando a velocidade diretamente de nosso corpo rígido para a Speed * Vector2.Right. Ou seja, o nosso corpo, que é do tipo Kinematic, se moverá na velocidade indicada para a direita, a partir do ponto onde for criado.

O método OnTriggerEnter2D é a variação para Triggers do OnCollisionEnter2D e tem o mesmo objetivo deste. A diferença, no entanto, é que ele recebe um Collider2D e não uma Collision2D. Como esse método indica que houve uma colisão entre o objeto e alguém, já o iniciamos com uma linha para tocar a animação de explosão e desabilitar a simulação do objeto.

É importante desabilitar essa simulação devido ao tempo levado pela animação Exploding para tocar. Não destruiremos o objeto assim que houver o contato, para a animação ter tempo de ser executada. Se não desabilitarmos a simulação imediatamente, é possível que o jogador entre em contato com ele novamente, mesmo ele sendo, no momento, só uma fumaça de explosão. Isso não é o desejado! Por isso, alteraremos o campo **simulated** para **false**.

Em seguida, depois de garantirmos que o objeto terá iniciado o seu processo de autodestruição, verificamos com quem ele colidiu. Faremos isso a partir do compareTag. Se detectarmos que a colisão foi com o Player, ativaremos um novo método do Player. Esse método, criado a seguir, será responsável por alterar os pontos de energia restantes, além de retornar o que sobrou, após alterar o total. Se o total resultante for menor ou igual a zero, o jogador perdeu uma tentativa e o nível deverá ser parado.

Isso será feito pela bazuca e não pelo míssil diretamente. Preste bastante atenção nesse passo!

Quando criarmos o script da bazuca, o qual será responsável por criar os nossos mísseis, teremos o cuidado de colocar todos os mísseis como filhos da bazuca. Isso acontece para a bazuca poder ser responsável por todos eles e se comunicar com cada um caso um evento como esse aconteça. A fim de isso funcionar, o míssil responsável por destruir o jogador deverá avisar à bazuca o ocorrido, para ela avisar a todos os outros mísseis. Esse sistema de comunicação é algo novo, que conheceremos em breve. A primeira parte, no entanto, é o filho avisar ao pai que há uma nova mensagem a ser transmitida. Isso é feito pelo método Stop, criado no componente PAI do míssil, a bazuca.

Após ter causado o dano e repassado qualquer mensagem de destruição, o elemento cumpriu sua missão e está pronto para ser destruído. A destruição deve acontecer com um atraso, no entanto, para garantir que a animação de explosão tenha tempo de tocar. Utilizamos, para isso, nosso conhecido método Invoke, com um atraso de 1 segundo.

Para finalizar o script, temos mais dois métodos. O primeiro deles, StopMissile, desabilita o míssil. Esse método será utilizado como receptor da mensagem da bazuca. Caso ela avise que o jogo chegou ao fim, o míssil cuidará de parar sua simulação. O último método será chamado pelo Invoke para destruir o componente após um tempo.

Com isso, finalizamos o nosso objeto Missile por completo! Ele está funcional e pronto para ser criado por nossa bazuca. Não conseguiremos ainda, no entanto, executar o nosso jogo para ver o componente funcionando, pois não definimos a nova função no player e nem criamos a nossa bazuca, a qual receberá a mensagem de stop que estamos enviando. Porém, caso queira ver o objeto funcionando, comente todo o IF do método OnTriggerEnter2D e aperte o play! Posicione o objeto adequadamente no nível para você poder vê-lo e interagir. A **Figura 2** demonstra o componente em ação, ainda sem o dano, com os métodos comentados.

Figura 02 - Míssil funcionando e acertando o personagem, ainda sem dano.

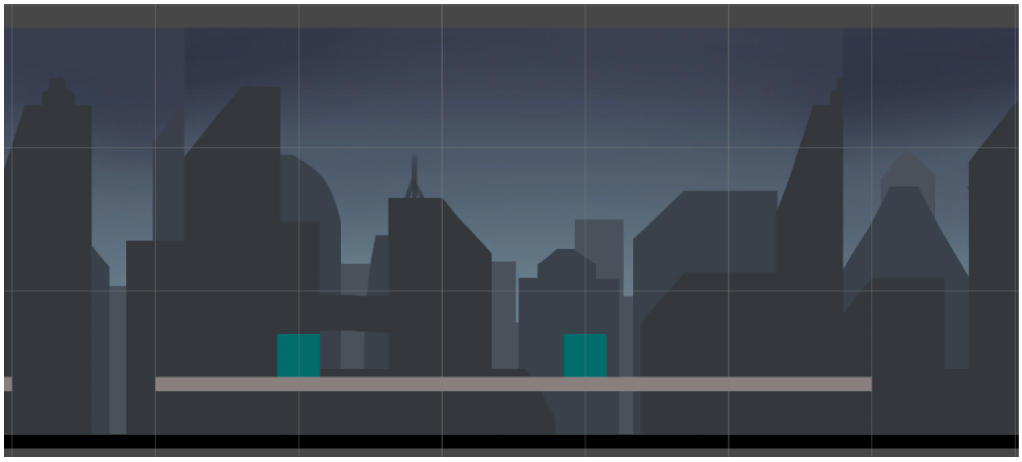
Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 20 de mar de 2017

Agora que já vimos o míssil funcionando, podemos, então, deletar o objeto da cena.

“Mas professor, nós criamos um objeto esse tempo todinho para deletá-lo?”

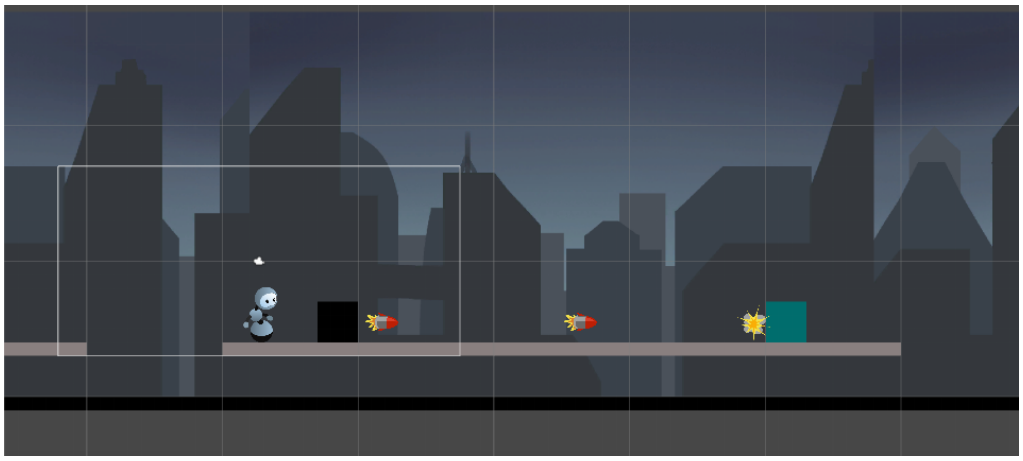
Sim! Estamos deletando apenas da cena! O Prefab criado a partir dele ainda está mantido. E é o Prefab que utilizaremos em nossa bazuca a fim de criar os mísseis na cena e desafiar um pouco mais o jogador a partir disso. O próximo passo, agora, é alterar o nosso Level_2, na parte vista na **Figura 3**, para podermos adicionar a bazuca à nossa cena. Veja, na **Figura 4** o resultado da alteração, que desenvolveremos a seguir.

Figura 03 - Parte do Level_2 a ser alterado para adicionarmos a bazuca.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 20 de mar de 2017

Figura 04 - Level_2 após as alterações realizadas, com a bazuca já adicionada à cena (em preto).



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 17 de mar de 2017

1.2 Criando o Spawner (ou Gerador de Objetos)

Agora que já temos o Prefab do elemento a ser criado dinamicamente, podemos criar aquele o qual será responsável por gerar os objetos e os lançar na cena, conhecido como Spawner, que significa algo como "gerador de objetos". Para iniciar o processo, criaremos um novo Sprite para representar a bazuca. Mais uma vez, faremos apenas com um elemento vazio de *placeholder*. Crie um novo sprite quadrado, na pasta Sprites, utilizando o comando Assets -> Create -> Sprites -> Square.

Da mesma maneira que fizemos com os outros objetos, vamos adicionar um GameObject vazio para armazenar as bazucas contidas em nossa fase. Para isso, utilize o menu GameObject -> Create Empty e então renomeie o objeto para Bazookas. Não esqueça de alterar sua posição e rotação para (0,0,0) com a escala (1,1,1).

Feito isso, adicione o novo sprite criado como filho desse novo objeto, formando, assim, a nossa primeira bazuca. Altere a sua Order in Layer para 1. Para posicionar os elementos como vimos na **Figura 4**, precisamos alterar a posição da bazuca para (-131.5, -4.5, 0) com a escala (3,3,0). Já o Prefab Box que aproveitaremos da cena como estava moveremos para (-98.5, -4.5, 0). Com isso, chegaremos à configuração vista na **Figura 4**. Agora falta apenas configurarmos a bazuca para funcionar!

Comecemos então pelos seus componentes. A bazuca precisará de um Box Collider 2D para o Player colidir com ela normalmente, uma vez que ela vai ser um componente do cenário como qualquer outro. As configurações padrões do colisor são o bastante para o que queremos. Outro detalhe que precisamos lembrar é de alterar a camada a qual ela pertence para Ground, assim o personagem poderá subir nela adequadamente.

Feito isso, o próximo passo é a criação do script da bazuca. Vejamos o script completo na **Listagem 2** para então discutirmos o que foi feito.

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class ShooterScript : MonoBehaviour {
5
6     public GameObject missilePrefab;
7
8     private float speed = 12f;
9     private Vector3 spawningPos;
10    private bool active = true;
11
12    // Use this for initialization
13    void Start () {
14       spawningPos = new Vector3(transform.position.x+transform.localScale.x, -4.5f,
15                transform.position.z);
16       Spawn ();
17    }
18
19    void Spawn() {
20       if (active) {
21          GameObject missile = GameObject.Instantiate (missilePrefab) as GameObject;
22
23          missile.transform.position = spawningPos;
24          missile.transform.parent = transform;
25
26          missile.gameObject.GetComponent<MissileController> ().speed = speed;
27
28          Invoke ("Spawn", Random.Range (1f, 1.5f));
29       }
30    }
31
32    public void Stop() {
33       BroadcastMessage("StopMissile");
34       active = false;
35    }
36 }
```

Listagem 2 - Código do script Shooter Script, adicionado ao objeto Bazooka.
Fonte: Elaborada pelo autor.

A primeira linha do código já é o motivo de toda a seção anterior ter existido! Antes de tudo, precisamos em nosso Spawner justamente de uma referência ao objeto que criaremos ao longo da execução do jogo. Como todo objeto herda de GameObject, podemos criar uma variável do tipo GameObject, garantindo que o Prefab poderá ser utilizado nessa variável. Vamos nomeá-la de missilePrefab para facilitar o entendimento.

Em seguida criamos as variáveis de configuração do Prefab em si – speed e spawningPos. A variável speed indicará a velocidade de movimentação do míssil horizontalmente e a spawningPos indicará a localização que o míssil será criado.

A última variável que criamos é a active, ela será responsável por interromper o ciclo de criação caso a cena deixe de estar ativa. Essa variável começará com o valor TRUE e só terá o valor alterado para FALSE caso chegue a mensagem de Stop() que criamos lá em nossos mísseis, lembra?

Em seguida, no método Start, atribuímos o valor da posição de criação do objeto. Essa posição é igual à posição do criador + a escala em X dele, para X, -4.5f para Y e também igual ao criador para Z. Isso indica que o objeto deverá ser criado numa altura fixa, mas em uma posição em X igual à do objeto que está criando o míssil, mais um *offset*. Adicionamos esse *offset* para que o objeto não seja criado em cima do seu criador, colidindo imediatamente e encerrando o seu ciclo de vida.

Após configurar a posição, chamamos o método Spawn pela primeira vez, ainda no método Start(). Com isso, iniciamos o nosso ciclo de criação de objetos, uma vez que o método Spawn, ao fim do seu ciclo, é chamado novamente, com um atraso definido aleatoriamente, entre 1 e 1.5f, através do método Random.Range.

Falando do método Spawn, note que ele está totalmente sob uma condição – o objeto estar ativo. Essa é a condição que quebrará o ciclo de chamadas próprias do método Spawn. Em seguida, caso o objeto esteja ativo, criaremos um novo GameObject, chamado missile, que é um GameObject.Instantiate de nossa variável missilePrefab como um GameObject. O que vemos na linha 20 é exatamente isso! Instanciamos uma nova cópia do Prefab, convertendo-a para um GameObject (pois sabemos que ela será um, com certeza) e salvando-a na nova variável criada.

Perceba que não é necessário criar uma nova variável para armazenar o objeto criado! Fazemos isso para que, em seguida, possamos alterar suas propriedades. Alteramos sua posição para a spawningPos que havíamos definido e, em seguida, alteramos o seu pai para o próprio objeto criado. Isso facilitará o gerenciamento de todos esses objetos filhos, como havíamos comentado anteriormente.

Por fim, acessamos o script do gameObject e alteramos o valor da variável speed para que seja igual à variável speed que criamos nesse script. Isso também pode ser útil caso queira variar o valor de velocidade entre objetos criados, utilizando, por exemplo, o método Random.Range, usado na última linha. Esse método serve justamente para escolher um valor aleatório dentro da escala definida entre os seus dois parâmetros, inclusive contando-os.

O último método em nosso script é justamente aquele que havíamos citado anteriormente, utilizado pelo míssil para avisar que o Player foi destruído. Esse método é responsável por parar a bazuca e avisar também a todos os mísseis já criados que eles podem parar. Podemos fazer isso utilizando o serviço de mensagens do Unity.

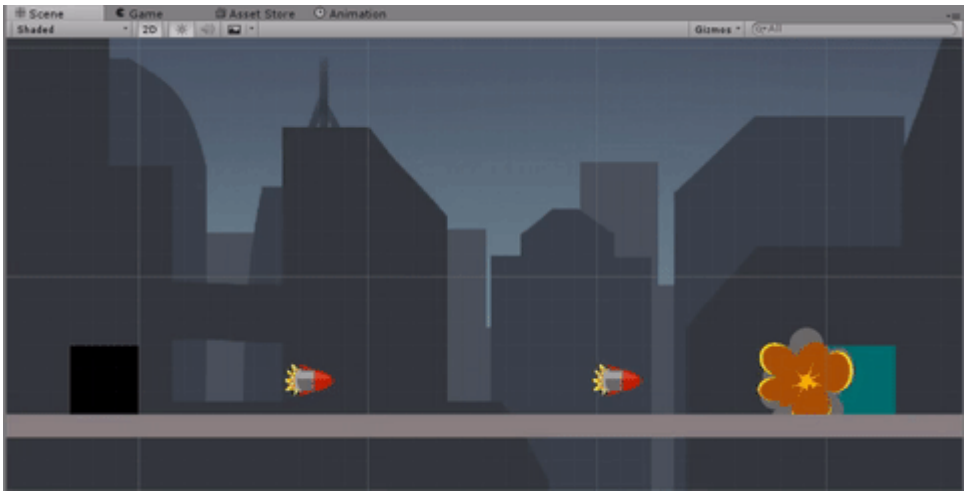
É possível, em diversas situações, trocar mensagens entre componentes os quais possuem alguma relação hierárquica. Para fazer isso, utilizamos os métodos de mensagem do Unity como SendMessage, SendMessageUpwards e BroadcastMessage. O primeiro envia uma mensagem a todos os componentes de um mesmo objeto. O segundo envia a mensagem para cima, ou seja, para o seu pai. Já o último, o BroadcastMessage, passa a mensagem a todos os filhos de um objeto.

Como estamos criando todos os mísseis como filhos da bazuca, ao utilizar o método de mensagens BroadcastMessage, podemos atingir todos os mísseis já criados. Esse método recebe um parâmetro que é justamente o nome do método a ser chamado em cada um desses recebedores. No caso, chamaremos o método StopMissile, o qual vimos na **Listagem 1**, sendo o responsável por desativar um míssil para que não haja mais interação dele com o ambiente.

Adicionado esse script ao nosso objeto Bazooka, na cena, precisamos apenas adicionar o Prefab Missile, criado anteriormente, como Prefab da variável pública a qual criamos no script. Como já vimos, basta arrastar o Prefab até o campo, no editor, para que façamos essa definição.

Com tudo feito, podemos apertar o Play e ver a nossa bazuca atirando diversos mísseis (lembre-se de comentar as linhas do IF da **Listagem 1**, uma vez que ainda não alteramos o Player para tomar dano, adicionando os métodos lá utilizados). O resultado pode ser visto na **Figura 5**.

Figura 05 - Bazuca atirando diversos mísseis com o intervalo de tempo aleatório entre eles.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 20 de mar de 2017

Com isso, concluímos nossa parte de criação dinâmica de elementos em nossa fase. Perceba que o processo é bem simples, como dissemos antes! Criamos um Prefab, criamos um Spawner, adicionamos ao Spawner um script que referencie o Prefab e então, em tempo de execução, seja capaz de o instanciar pelo método GameObject.Instantiate. Assim, novos objetos aparecem a partir daquele Prefab, da maneira que for configurado no Spawner! Massa \o/

2. Adicionando dano ao Jogo

Outro elemento importante do jogo é o dano. Em nosso cenário, temos dois casos diferentes. Caso o jogador caia em algum buraco, ele perderá toda a sua vida. Caso entre em contato com um míssil, tomará o dano igual ao configurado no script do míssil.

Para que o nosso personagem possa tomar esse dano, precisamos alterar um pouco o nosso PlayerController script, adicionando a ele variáveis de energia restante e também de energia total, pois precisamos evitar que ele se cure mais do

que pode. Essa variáveis podem ser vistas na **Listagem 3**.

```
1  private float startingHealth = 100f;
2  private float currentHealth;
3  private Slider healthBar;
```

Listagem 3 - Novas variáveis criadas no PlayerController.
Fonte: Elaborada pelo autor

A primeira variável diz respeito à vida total, para evitarmos passar do limite em caso de uma cura encontrada, ou mesmo de um reinício. Já a segunda, indica quanto de energia restante o personagem possui naquele momento. Utilizaremos isso para saber, caso diminua de zero, que uma tentativa foi perdida. Também utilizaremos esse valor para configurar o posicionamento da healthBar, uma referência ao Slider que adicionamos na aula passada.

Com essas três variáveis podemos implementar o novo código necessário para o Player responder a qualquer dano ou cura – o método HealthChange. Esse método irá receber uma variável e alterar a vida de acordo com o valor recebido, respeitando sempre o limite máximo e finalizando a tentativa atual caso o limite mínimo seja ultrapassado. O código pode ser visto na **Listagem 4**.

```
1  public float HealthChange (float value) {
2     if (currentHealth + value < startingHealth) {
3        currentHealth += value;
4     } else {
5        currentHealth = startingHealth;
6     }
7
8     healthBar.value = currentHealth;
9
10    if (currentHealth <= 0) {
11       Break();
12    }
13
14    return currentHealth;
15 }
```

Listagem 4 - Método HealthChange, adicionado ao final do script PlayerController.
Fonte: Elaborada pelo autor.

O método é bem simples! Caso a alteração resulte em um valor maior que o máximo, alteramos para o máximo. Caso contrário, alteramos para o novo valor, após o dano ou cura. Em seguida, utilizamos a propriedade **value** de nosso Slider para indicar o novo valor que ele deve exibir. Isso é feito com o comando

healthBar.value = currentHealth. Por fim, fazemos um teste simples. Se a quantidade atual de vida for menor que 0, ativamos o método Break para finalizar a tentativa. Retornamos, ao final do método, o valor atual da vida para que o componente que causou a alteração possa agir.

Por fim, precisamos de uma pequena alteração no método Break, uma vez que o chão dos buracos atua diretamente com esse método. Caso o personagem vá quebrar, a barra de vida deve ser completamente zerada, independentemente de seu valor anterior. Fazemos isso alterando o valor da propriedade Color do FillRect, componente do Slider que vimos em aulas anteriores. O código completo do novo PlayerController, incluindo as alterações no método Break, podem ser vistos na **Listagem 5**.

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

public class PlayerController : MonoBehaviour {

    private bool jumping = false;
    private bool grounded = false;
    private bool doubleJump = false;
    private bool doubleJumping = false;
    private bool movingRight = true;
    private bool active = true;

    private Rigidbody2D rigidBody;
    private Animator ani;
    public Transform groundCheck;
    public LayerMask layerMask;

    public float acceleration = 100f;
    public float maxSpeed = 10f;
    public float jumpSpeed = 500f;

    private float startingHealth = 100f;
    private float currentHealth;
    private Slider healthBar;

    // Use this for initialization
    void Awake () {
        rigidBody = GetComponent<Rigidbody2D> ();
        ani = GetComponent<Animator>();
        healthBar = FindObjectOfType<Slider>();
        currentHealth = startingHealth;
        healthBar.value = currentHealth;

    }

    // Update is called once per frame
    void Update() {
        if (active) {
            if (grounded) {
                doubleJump = true;
            }

            if (Input.GetButtonDown("Jump")) {
                if (grounded) {
                    jumping = true;
                    doubleJump = true;
                } else if (doubleJump) {
                    doubleJumping = true;
                    doubleJump = false;
```

```
        }
      }
    }
}

//Called in fixed time intervals, frame rate independent
void FixedUpdate() {
    if (active) {
        float moveH = Input.GetAxis ("Horizontal");

        ani.SetFloat("speed", Mathf.Abs(moveH));

        grounded = Physics2D.OverlapBox (groundCheck.position, (new Vector2 (1.3f, 0.2f)), 0f, layer

        ani.SetBool("grounded", grounded);
        ani.SetFloat("vertSpeed", rigidBody.velocity.y);

        if (moveH < 0 && movingRight) {
            Flip();
        } else if (moveH > 0 && !movingRight) {
            Flip();
        }

        rigidBody.velocity = new Vector3 (maxSpeed * moveH, rigidBody.velocity.y, 0);

        if (jumping) {
            rigidBody.AddForce(new Vector2(0f, jumpSpeed));
            jumping = false;
        }
        if (doubleJumping) {
            rigidBody.velocity = new Vector2 (rigidBody.velocity.x, 0);
            rigidBody.AddForce(new Vector2(0f, jumpSpeed));
            doubleJumping = false;
        }
    }
}

void Flip() {
    movingRight = !movingRight;
    transform.localScale = new Vector3((transform.localScale.x * -1), transform.localScale.y, transf
}

public bool isGrounded () {
    return grounded;
}

public void Break () {
    active = false;
    //rigidBody.bodyType = RigidbodyType2D.Static;
    ani.SetBool("active", false);
    ani.Play("Break");
```

```
103         FindObjectOfType<GameController>().Break();
104         healthBar.fillRect.GetComponentInChildren<Image>().color = new Color(0,0,0,0);
105     }
106 
107     public void LevelEnd () {
108         active = false;
109         rigidBody.bodyType = RigidbodyType2D.Static;
110         ani.SetBool("active", false);
111         ani.Play("CelebrationRoll");
112         FindObjectOfType<GameController>().LevelEnd();
113     }
114 
115     public float HealthChange (float value) {
116         if (currentHealth + value < startingHealth) {
117             currentHealth += value;
118         } else {
119             currentHealth = startingHealth;
120         }
121 
122         healthBar.value = currentHealth;
123 
124         if (currentHealth <= 0) {
125             Break();
126         }
127 
128         return currentHealth;
129     }
130 }
131 
```

Listagem 5 - Código alterado do script PlayerController, com o novo método HealthChange, as novas variáveis e as alterações realizadas no método Break.
Fonte: Elaborada pelo autor

3. Adicionando Objetos Coletáveis

Para finalizar a nossa aula, adicionaremos um elemento coletável em nossa cena. Esses elementos são muito comuns em diversos jogos, como os Power Ups do Megaman ou as moedinhas do Mario. Em nossa fase, utilizaremos o mesmo sprite da UI para adicionar um pack de recuperação de energia à nossa cena. Veja na **Figura 6!**

Figura 06 - Objeto coletável adicionado à cena.

Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 20 de mar de 2017

Para começar, criemos um novo objeto vazio chamado EnergyPacks. Esse objeto armazenará os packs de energia. Em seguida, adicione o sprite à cena como filho desse objeto. Ele estará gigante! Altere sua posição para (-92, -4.75, 0) e a sua escala para (0.2, 0.2, 0). Com isso, ele ficará em um tamanho ideal, logo após os nossos mísseis. Renomeie o objeto para EnergyPack e altere sua Order in Layer para 1.

Em seguida, adicione ao EnergyPack um Box Collider 2D para que possa detectar as colisões e recuperar a vida do Player quando oportuno. Para que não haja uma colisão física, como é o comportamento padrão com esses objetos, ative a opção isTrigger. Não é mais necessário qualquer componente físico para que ele funcione!

O último componente que precisaremos é o script que vai executar a cura, de fato. Crie um novo script, na pasta Scripts, chamado Energy Pack e o adicione ao objeto EnergyPack. Esse script terá o código bem simples, como visto na **Listagem 6**.

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

public class EnergyPack : MonoBehaviour {

    public float healthRecovered = 25f;

    void OnTriggerEnter2D (Collider2D col) {
        if (col.CompareTag ("Player")) {
            col.gameObject.GetComponent<PlayerController>().HealthChange(healthRecovered);
            Destroy(gameObject);
        }
    }

}
```

Listagem 6 - Código do script EnergyPack, adicionado ao objeto EnergyPack
Fonte: Elaborada pelo autor

Com isso, quando houver uma colisão, o script irá detectar se foi com o Player. Caso tenha sido, a função de alteração de vida será chamada, passando, dessa vez, um valor positivo. Em seguida, o objeto será destruído para que não seja utilizado novamente. Assim, criamos, com sucesso, uma réplica do comportamento de objetos coletáveis. Se quiser, pode-se criar um Prefab desse elemento e então espalhá-lo pela cena, à medida que o usuário possa necessitar! As oportunidades de alteração ficam ao seu critério.

E com isso, concluímos a nossa penúltima aula! Está quase acabando! E já temos um jogo bem completo, não é? Na próxima aula finalizaremos os últimos detalhes e então veremos como podemos distribuir esse nosso jogo! Até lá, meus amigos! o/

Leitura Complementar

Referência do método Instantiate - Disponível em: https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

Texto, em inglês, sobre a instanciação de Prefabs - Disponível em: https://docs.unity3d.com/Manual/InstantiatingPrefabs.html

Sistema de mensagens do Unity - Disponível em: https://docs.unity3d.com/Manual/MessagingSystem.html

Resumo

Na aula de hoje conhecemos três novas técnicas interessantes. Aprendemos, no começo da aula, como criar um novo Prefab e utilizá-lo, em conjunto com um Spawner, para criar elementos em tempo de execução. Para isso, conhecemos o método Instantiate, que é responsável por tal comportamento. Também aprendemos a editar algumas propriedades do Prefab após criá-lo.

Em seguida, vimos como podemos interagir com o Player a partir dos nossos Prefabs e como eles podem receber feedback do Player. Vimos também como essas alterações no Player podem refletir diretamente na UI, a partir de propriedades específicos dos elementos do HUD.

Por fim, aprendemos a adicionar objetos coletáveis em nossa cena, representando, nesse caso, um objeto capaz de recuperar a energia do nosso robô! Esse componente faz a sua função e é, em seguida, deletado rapidamente para evitar duplicações.

Também conhecemos, na aula de hoje, o sistema de mensagens do Unity e como podemos utilizá-lo para realizar a comunicação entre elementos em pontos diversos da hierarquia. O utilizamos para enviar mensagens de fim de jogo a todos os objetos criados pelo Spawner.

O código final do projeto, como estava ao fim da aula de hoje, pode ser encontrado aqui! Qualquer dúvida que tenham, fiquem à vontade para nos procurar nos fóruns!

Até a próxima aula! Nela finalizaremos a disciplina com toda a parte de distribuição de jogos! o/

Autoavaliação

1. Como podemos instanciar objetos em tempo de execução?
2. Quais são os objetos necessários para se criar objetos em tempo de execução?
3. Quais os componentes necessários para criarmos um elemento coletável?
4. Como foi possível alterar a energia do personagem na UI?
5. Como funciona o sistema de mensagens do Unity?

Referências

Documentação oficial do Unity - Disponível em: https://docs.unity3d.com/Manual/index.html.

Tutoriais oficiais do Unity - Disponível em: https://unity3d.com/pt/learn/tutorials.

RABIN, Steve. **Introdução ao Desenvolvimento de Games**, Vol 2. CENGAGE.