Desenvolvimento com Motores de Jogos I

Aula 11 - O Game Controller e a Transição Entre Cenas

Apresentação

Fala, galera! Tudo certo? Chegamos ao último terço de nossa disciplina e já temos um joguinho bem legal desenvolvido. Entretanto, precisamos adicionar ainda algumas funcionalidades para o jogo ficar completinho, assim como dissemos que seria desde o começo!

Já temos um personagem animado navegando por um cenário bem legal, onde há desafios diversos. Esse personagem, no entanto, ainda se perde em alguns buracos que colocamos pelo cenário. Melhor alterar isso, não é? Na aula de hoje, aprenderemos a alternar entre cenas no nosso jogo e poderemos, uma vez que o personagem caia no buraco, reiniciar a cena.

Esse reinício, é claro, custará uma das tentativas, correto? Não podemos deixar o nosso jogo correr para sempre! É necessário, então, conheceremos o componente Game Controller, responsável por armazenar dados gerais do jogo e facilitar a transição entre cenários, fazendo exatamente o que o seu nome diz – controlando o jogo.

Com esse controle de vidas, além de reiniciarmos nosso nível, aprenderemos a passar a uma tela de Game Over, ou mesmo a uma outra fase, caso o personagem consiga chegar até o final inteirinho. Bacana, não? Adicionaremos, também, duas novas animações ao nosso personagem relativas aos dois novos comportamentos que adicionaremos ao jogo: uma celebração para quando finalmente chegarmos ao IMD e um fim não tão feliz para quando cairmos no buraco.

Muitas coisas interessantes, como precisam ser, agora que já temos bastante conhecimento em nosso motor de jogos e podemos avançar satisfatoriamente com o nosso projeto. Animados? Eu estou! Vamos lá!

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Utilizar o SceneManager para navegar entre cenas no Unity;
- Utilizar um Game Controller para controlar o seu jogo;
- Manter objetos vivos entre cenas e reaproveitar objetos já criados.

1. Adicionando Novas Animações ao Personagem

Começaremos a nossa aula com duas novas animações para o nosso personagem! Faremos com que ele role de alegria quando encontrar a graminha do IMD e também adicionaremos um pequeno defeito de fabricação para ocorrer caso ele encontre as profundezas dos buracos colocados em nossa fase. A fim de realizar esses objetivos, faremos um processo igualzinho ao feito em nossa aula sobre animações. Lembram-se bem? Caso não, agora é um bom momento para voltar lá e dar uma olhada nos detalhes! Enquanto você faz isso, já deixa baixando as duas novas Sprite Sheets que utilizaremos em nossa aula. Elas podem ser encontradas aqui!

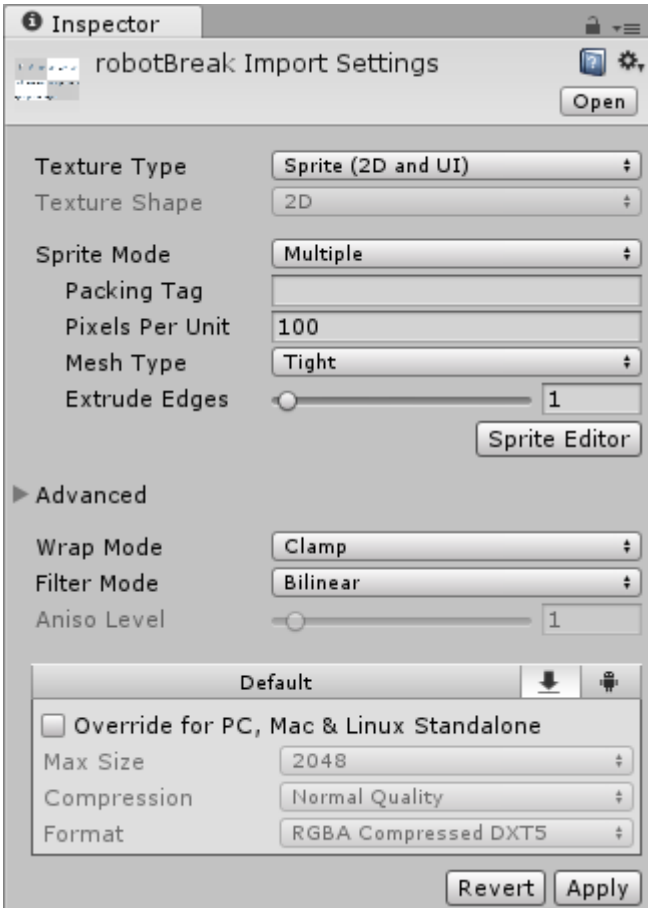
Como já vimos anteriormente os detalhes da criação de animações, passaremos rapidamente pelo processo. Caso fique alguma dúvida, consulte as aulas de animação ou vá até o fórum falar conosco! Ficaremos felizes em ajudar! :D

1.1 Importando as Sprite Sheets no Unity

O primeiro passo será a importação, para o nosso projeto, das duas novas Sprite Sheets que baixamos acima. Para tanto, basta arrastar as novas imagens para a pasta Assets -> Sprites, no Unity ou até mesmo no navegador de arquivos. Feito isso, o Unity processará as imagens e elas estarão disponíveis em nosso projeto. Basta clicar em cada uma para iniciarmos as configurações de importação.

Lembrando rapidamente os passos, precisamos alterar, para cada uma das Sprite Sheets, o Sprite Mode de **Single** para **Multiple** e, então, aplicar as alterações no botão Apply, no canto inferior direito das propriedades. Em seguida, utilizamos o botão Sprite Editor para iniciar o corte de nossos sprites a partir das Sprite Sheets. Todas essas propriedades podem ser vistas na **Figura 1**.

Figura 01 - Import Settings da Sprite Sheet robotBreak.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Após clicar em Sprite Editor, acessaremos o editor de Sprites. Como já fizemos antes, devemos alterar o tipo de corte de Automatic para Grid by Cell Size, utilizando o mesmo tamanho que havíamos usado em outras Sprites Sheets: X = 600 e Y = 800.

Em seguida, precisamos selecionar os cortes vazios criados ao final e, então, deletá-los, a fim de evitar a criação de sprites desnecessários em nossa Sprite Sheet. Utilizamos mais uma vez o botão Apply para aplicar os cortes que fizemos. Após fazer isso para as duas novas imagens adicionadas, teremos os nossos sprites prontos para serem animados em nosso Player.

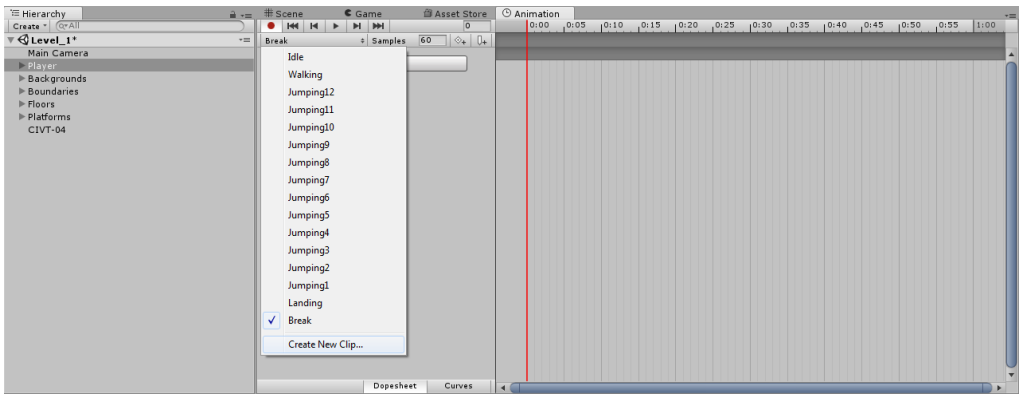
1.2 Criando Novas Animações no Player

Criaremos duas novas animações no Player, como já dissemos. Elas serão um pouco diferentes das que já havíamos criado antes. Não na criação! Serão criadas da mesma maneira e faremos as mesmas configurações vistas na aula sobre esse

assunto. No entanto, elas terão uma diferença no Animator: não haverá transições ligadas a elas! As animações serão tocadas exclusivamente por comandos expressos via script. Vamos ver como fazer isso?

Como dito, a criação das animações seguirá os mesmos passos de antes. Abriremos a aba Animation (Ctrl + 6 ou Window -> Animation) e selecionaremos na Hierarchy o nosso Player. Ao fazer isso, suas animações serão mostradas na aba Animation. Clicaremos no menu que contém as animações, dentro dessa aba, e selecionaremos a última opção – Create New Clip. Dê a esse primeiro clipe de animação o nome Break e o salve na pasta Animations, mantendo sempre os nossos assets organizados! A **Figura 2** mostra a animação já criada, no menu.

Figura 02 - Aba Animation já com a animação de Break criada.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

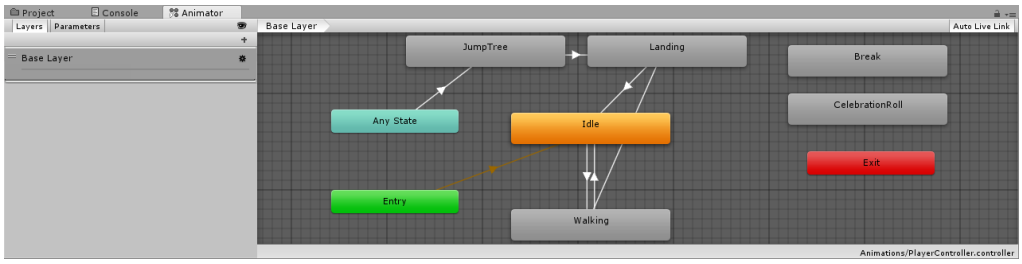
Com a animação criada, o próximo passo é configurar a quantidade de samples de nossa animação. Como estamos trabalhando com dezesseis quadros e essa animação foi feita para durar cerca de um segundo e meio, podemos utilizar doze samples por segundo, a fim de a animação ficar com uma qualidade adequada. Em seguida, devemos arrastar todos os sprites da Sprite Sheet robotBreak para a animação. Isso criará os keyframes correspondentes e deixará a animação já pronta para ser executada.

Após configurar a animação de Break, podemos repetir o processo para a animação de Rolling. Selecionaremos, no menu visto na **Figura 2**, a opção Create New Clip, e utilizaremos dessa vez o nome CelebrationRoll! Essa animação será utilizada quando o nosso robozinho finalmente encontrar a verde grama do IMD.

Criado o arquivo de animação, o próximo passo é configurá-la. Para ela, temos quatorze imagens, totalizando um segundo de execução. Assim, configuraremos as samples dessa animação para quatorze, arrastando, em seguida, todas as imagens para a aba Animation. Com isso, teremos as duas animações criadas e configuradas.

Lembra-se de falarmos que elas ficariam um pouco diferente no Animator Controller? Pois é! Vejamos na **Figura 3** como essas animações aparecem por lá. Ressaltando que elas são adicionadas automaticamente ao controlador quando são criadas pela aba Animation.

Figura 03 - Animator Controller do Player com as duas novas animações.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

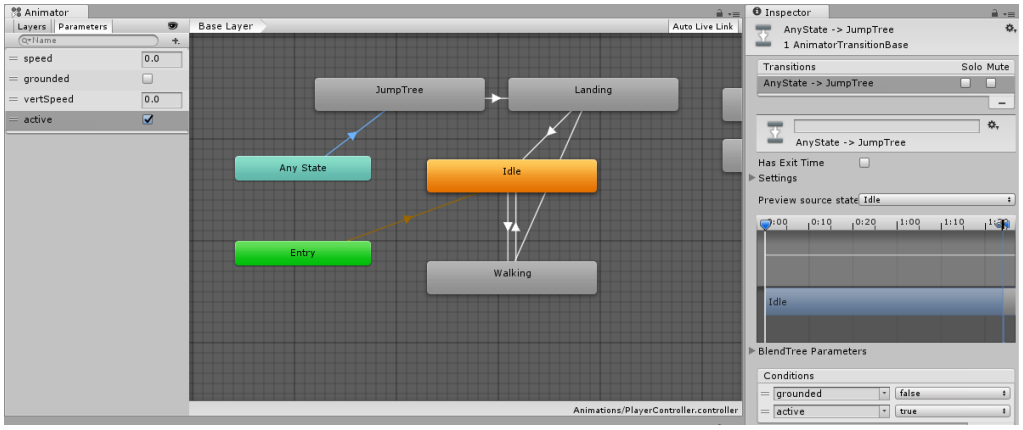
Viu? Elas ficam separadas do resto, sem qualquer transição que as leva ou mesmo faz o controlador sair delas. Tudo bem! Ambas as animações serão utilizadas em fim de fases e, para serem úteis, não precisam retornar o controle de qualquer maneira.

Temos um pequeno problema em nosso plano, no entanto. A animação JumpTree, como vemos na **Figura 3**, é iniciada a partir de Any State, ou seja, ela pode roubar a execução da animação de Break ou de CelebrationRoll, uma vez que há uma transição para ela a partir de qualquer estado. Para evitar isso acontecer, precisamos adicionar um novo parâmetro ao nosso Animation Controller. Esse parâmetro será um booleano, chamado *active*, o qual será, também, uma condição para a transição Any State -> JumpTree. A transição ocorrerá somente se o *grounded* for **false** e se o *active* for **true**. Modifique a transição para que isso aconteça.

Ah! E é muito importante você, ao criar o novo parâmetro, configurar o valor inicial dele para verdadeiro, clicando na caixinha ao lado do nome do parâmetro. Assim, teremos a certeza de que o personagem começará ativo, como deve ser, e se

alterará para inativo apenas quando for explicitamente solicitado. A **Figura 4** mostra a transição alterada e o parâmetro adicionado.

Figura 04 - Parâmetro *active* criado com o valor **true** e adicionado à transição.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Pronto! Animações criadas e adicionadas como parte do controlador! Para finalizar essa parte do nosso projeto, apenas mais uma coisa: removeremos o loop da animação de Break, para que o nosso personagem fique quebrado após sua execução.

A fim de fazer isso, navegue até a pasta contendo as animações de seu projeto e clique sobre a animação Break. Ela abrirá algumas de suas propriedades no Inspector. Há algumas informações interessantes lá, como a duração em segundos e os FPS, mas o que nos interessa no momento é a propriedade Loop Time. Desmarque essa caixa para desativar o loop da animação e estaremos prontos para avançar!

2. Modificando o Jogo Para Quebrar o Personagem

Agora que já temos nossas novas animações, podemos começar a utilizá-las! Modificaremos um pouco o nosso jogo para que o personagem possa quebrar ao cair nos buracos, gastando, assim, uma de suas chances de chegar ao objetivo final.

Para tanto, a primeira movimentação que faremos será na câmera, a fim de ela poder acompanhar a queda do personagem.

2.1 Alterando o Script da Câmera

Agora finalmente passaremos a ver o que acontece com o nosso personagem quando ele vai além dos buracos existentes em nossa fase, portanto, precisamos modificar um pouco a nossa câmera para acompanhá-lo indo para baixo, assim como ela já faz com a movimentação dele para cima.

Precisamos ter cuidado, no entanto, para a câmera não ir abaixo do fim de nosso background, revelando, assim, que aquilo é só uma imagem e não representa o mundo como um todo! Absurdo! :O

Vejamos a **Listagem 1**, contendo as alterações no script da câmera, para, então, discutirmos as mudanças.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class CameraMovementScript : MonoBehaviour {
6
7     public Transform player;
8
9     // Use this for initialization
10    void Start () {
11
12    }
13
14    // Update is called once per frame
15    void Update () {
16       if (player.position.y >= 5) {
17          transform.position = new Vector3 (player.position.x, (player.position.y - 5.0f), transform.posit
18       } else if (player.position.y <= -3) {
19          if ((player.position.y + 3.0f) > -3.4f) {
20             transform.position = new Vector3 (player.position.x, (player.position.y + 3.0f), transform.po
21          } else {
22             transform.position = new Vector3 (player.position.x, -3.4f, transform.position.z);
23          }
24       } else
25          transform.position = new Vector3(player.position.x, transform.position.y, transform.position
26    }
27 }
```

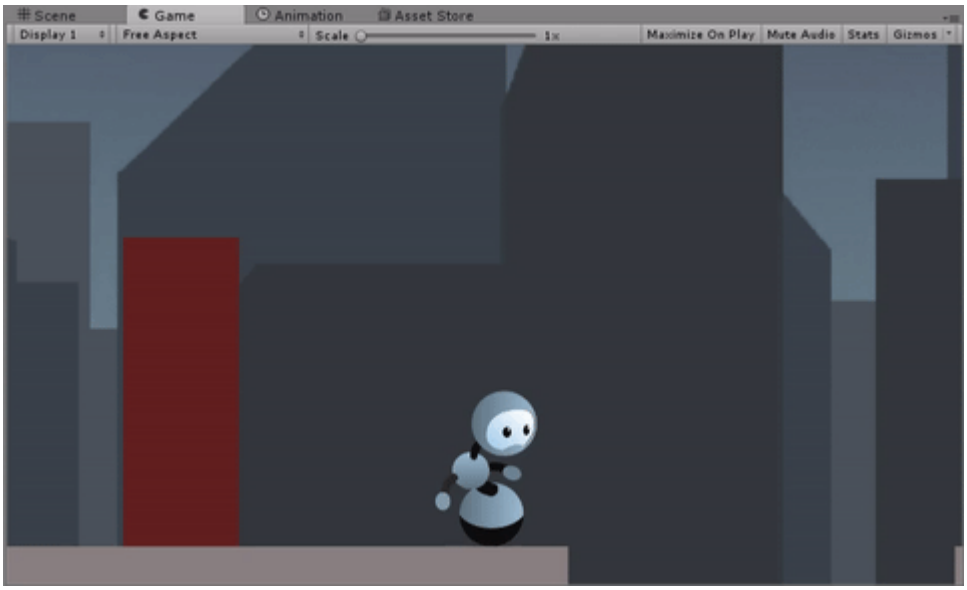
Listagem 1 - Script da câmera modificado para acompanhar também o movimento para baixo do personagem.
Fonte: Elaborada pelo autor.

Modificamos um pouco os nossos scripts, como vocês podem ver, mas mantivemos a mesma lógica utilizada anteriormente. Primeiro, adicionamos uma nova condição, além da posição maior do que cinco. Agora, observamos, também, se a posição do player é menor do que três, o valor da posição dele no chão. Caso seja, o personagem está caindo e a câmera deve acompanhá-lo.

Temos de ter um cuidado adicional, no entanto! Precisamos ver se esse acompanhamento, em algum momento, resulta em um valor menor do que 3.4f para a câmera. Caso resulte, paramos de mover a câmera e a fixamos nesse novo ponto, pois esse é o limite dela, a fim de não revelar além do background. É interessante, também, memorizarmos esse ponto para que, quando formos configurar a área de fim do personagem, possamos criá-la baseada nesse ponto, por essa mesma razão!

Com isso, concluímos o novo script da câmera. Jogue um pouco e caia nos buracos para ver o novo comportamento adicionado ao nosso jogo! A **Figura 5** mostra essa nova movimentação de câmera.

Figura 05 - Nova movimentação da câmera, em conformidade com a alteração no script.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Agora que a câmera já consegue acompanhar o nosso personagem enquanto ele cai, o próximo passo é adicionar o local onde ele encontrará o que há abaixo dos buracos do nosso nível. Como em diversos outros jogos de plataforma, podemos

adicionar lava, espinhos, serras, ou qualquer outro objeto desejado. Para o nosso robô, o fim será um chão preto.

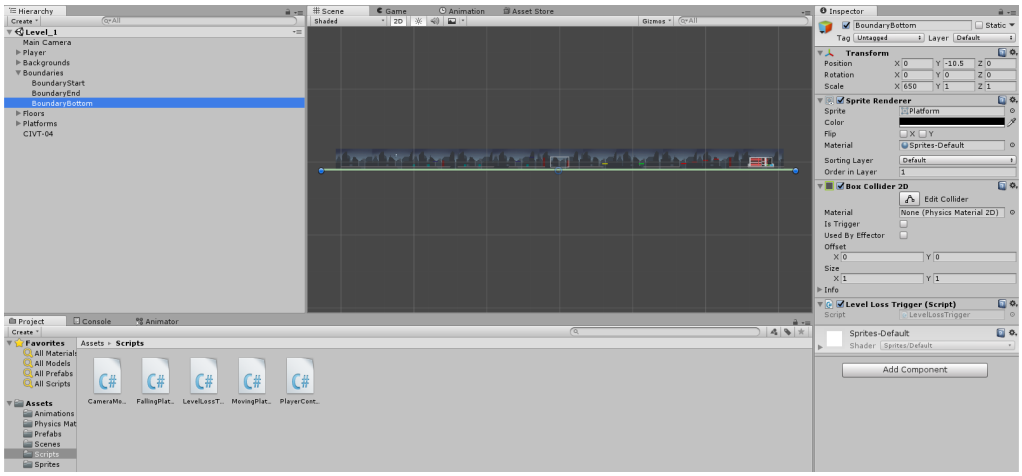
2.2 Adicionando um Objeto para Quebrar o Personagem

Existem algumas abordagens para adicionar objetos os quais afetem toda a fase. Aqui, utilizaremos a mais simples. Adicionaremos um objeto grande, do tamanho da fase, um pouco abaixo do nível (lembram do valor que eu pedi para memorizarem?), com um script para iniciar (também conhecido como triggar) a quebra de nosso personagem.

Esse objeto será criado a partir do mesmo sprite das plataformas, alterado para uma cor preta e posicionado de maneira a ocupar toda a parte de baixo da fase. Como também se trata de uma fronteira da fase, podemos adicionar esse elemento dentro de nossas Boundaries. Comecemos criando o objeto a partir do sprite Platform, arrastando-o para a nossa Hierarchy, em cima do objeto Boundaries, a fim de criá-lo como seu filho. Em seguida, renomeemos o novo objeto para BoundaryBottom.

Em relação aos componentes do objeto, precisaremos adicionar alguns novos componentes e alterar algumas propriedades dos atuais. Modificaremos, inicialmente, o Transform dele para que a posição seja (0, -10.5, 0) e a escala seja (650, 1, 1). Em seguida, no Sprite Renderer, alteramos a cor para preto e a Order in Layer para 1. Por fim, precisaremos adicionar um Box Collider 2D ao nosso objeto, para este poder parar o personagem, além de um script, que iniciará a quebra do personagem. Nomeie o script LevelLossTrigger e mova-o para a pasta Scripts. O resultado de todas essas alterações pode ser visto na **Figura 6**.

Figura 06 - Adição da fronteira de baixo da fase, com todos os seus componentes.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Agora, precisamos apenas alterar o código do script criado, a fim de fazer o que queremos. Mais uma vez, vejamos o código, na **Listagem 2**, para, em seguida, discutir os elementos contidos.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class LevelLossTrigger : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9      }
10
11     void OnCollisionEnter2D (Collision2D col) {
12         if (col.gameObject.CompareTag("Player")) {
13             col.gameObject.GetComponent<PlayerController>().Break();
14     }
15 }
```

Listagem 2 - Código do LevelLossScript.cs.
Fonte: Elaborada pelo autor.

Perceba que esse script não usa nem o Start, nem o Update. A função importante para esse script é a função OnCollisionEnter2D, a qual nos avisará quando um objeto entrar em contato com o nosso objeto fronteira. Ao haver uma colisão, verificaremos se o objeto que colidiu é o Player (não se esqueça de verificar

se o seu objeto Player está com a tag Player atribuída a ele!). Caso seja, buscaremos o script PlayerController e ativaremos nele o método Break(), responsável por alterar o personagem para o comportamento de quebrar.

O código é bem simples e não é tão complicado de entender, mas, como vemos na própria Listagem 2, o método Break está em vermelho. O que isso significa? Que não temos um método Break em nosso script PlayerController! Precisamos resolver isso, certo?

2.3 Modificando o Script PlayerController para Adicionar um Método de Quebra

Vejamos a **Listagem 3**.

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

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;

    // Use this for initialization
    void Awake () {
        rigidBody = GetComponent<Rigidbody2D> ();
        ani = GetComponent<Animator>();
    }

    // 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");
```

```
52
53          ani.SetFloat("speed", Mathf.Abs(moveH));
54
55          grounded = Physics2D.OverlapBox (groundCheck.position, (new Vector2 (1.3f, 0.2f)), 0f, layer
56
57          ani.SetBool("grounded", grounded);
58          ani.SetFloat("vertSpeed", rigidBody.velocity.y);
59
60          if (moveH < 0 && movingRight) {
61              Flip();
62          } else if (moveH > 0 && !movingRight) {
63              Flip();
64          }
65
66          rigidBody.velocity = new Vector3 (maxSpeed * moveH, rigidBody.velocity.y, 0);
67
68          if (jumping) {
69              rigidBody.AddForce(new Vector2(0f, jumpSpeed));
70              jumping = false;
71          }
72          if (doubleJumping) {
73              rigidBody.velocity = new Vector2 (rigidBody.velocity.x, 0);
74              rigidBody.AddForce(new Vector2(0f, jumpSpeed));
75              doubleJumping = false;
76          }
77      }
78  }
79
80  void Flip() {
81      movingRight = !movingRight;
82      transform.localScale = new Vector3((transform.localScale.x * -1), transform.localScale.y, transfo
83  }
84
85  public bool isGrounded () {
86      return grounded;
87  }
88
89  public void Break () {
90      active = false;
91      rigidBody.bodyType = RigidbodyType2D.Static;
92      ani.SetBool("active", false);
93      ani.Play("Break");
94  }
95 }
```

Listagem 3 - Modificações (em negrito) feitas no *PlayerController.cs* para adicionar o comportamento de quebra ao personagem do jogo.
Fonte: Elaborada pelo autor.

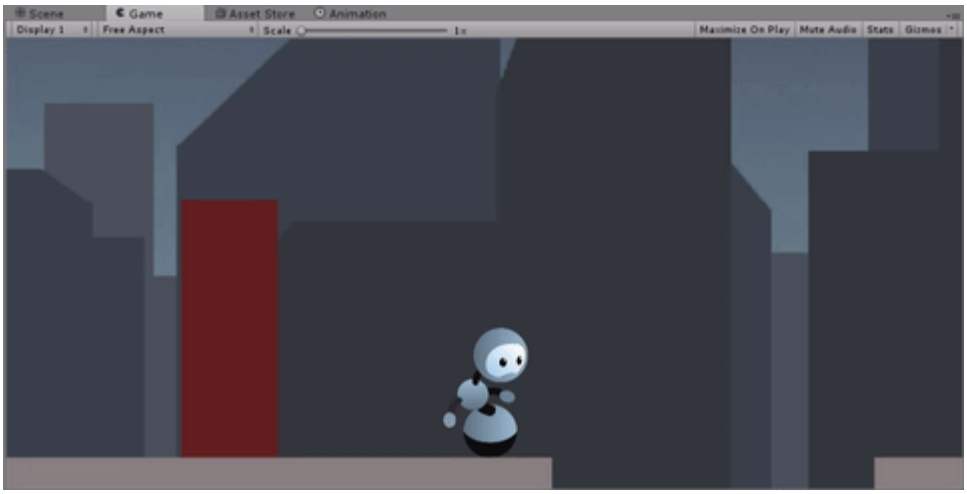
Criamos, como visto na **Listagem 3**, o nosso método Break. A primeira coisa que ele precisa fazer, no entanto, é parar o personagem. Para tanto, precisamos criar uma nova variável booleana **active** que nos mostra se o personagem está ativo ou não. Caso não esteja, ignoramos todo o controle tido pelos métodos FixedUpdate e Update sobre o personagem, tornando o personagem estático. Fazemos isso com um novo **if** no início de cada um desses métodos, checando se estão ativos ou não.

E por falar em estático, uma vez que o nosso personagem não está mais sob o controle do jogador e acaba de se quebrar, podemos também alterar o tipo do corpo rígido para Static, a fim de ter a garantia de ele não estar mais em nenhum dos cálculos físicos feitos pelo motor.

Com isso, precisamos, agora, apenas realizar mais duas etapas. A primeira delas é avisar ao nosso Animator Controller que o personagem não está mais ativo, evitando, assim, de ele cair no estado de pulo/queda, já que o personagem não está mais *grounded* e esse estado parte de **Any State**. A última etapa é tocar a nova animação criada no início de nossa aula. Com o método Play, do nosso Animator, podemos iniciar a execução de uma animação pelo nome. Passamos, então, ao parâmetro “Break”.

Assim, nosso método de quebra estará funcionando e pronto para ser chamado pelo script de LevelLoss. Vamos testar? Vai lá, dá o play no jogo e veja o que acontece!

Figura 07 - Personagem quebrando ao tocar no fim do cenário.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Calma, pessoal! Ele não morreu! Está tudo bem com ele! Um colega de mecatrônica encontrou o robô em pedaços, após o ocorrido, e juntou todas as peças novamente. Está tudo bem! Duvida? Pois vamos adiante, para a próxima seção!

3. O Game Controller

Um dos objetos superimportantes que nós temos no desenvolvimento de jogos é o Game Controller, ou controlador do jogo. Esse objeto, como o nome indica, é responsável por controlar aspectos mais gerais do jogo e gerenciar diversos acontecimentos, tais como os níveis de um jogo, as vidas possuídas por um personagem, a pontuação geral, entre outras coisas.

O nosso jogo ainda não possui um Game Controller, portanto, não temos uma maneira eficiente de gerenciar qualquer um desses parâmetros, inclusive a troca de níveis ou a quantidade de vidas restantes ao personagem. Chegou o momento de mudar isso! Vamos criar o nosso Game Controller!

Apesar de ser tão especial, o Game Controller não tem nada demais atrelado a ele, em relação ao GameObject que o representa. Criaremos o Game Controller a partir de um GameObject vazio, nomeado GameController. Em seguida, alteraremos a sua tag para Game Controller. O Unity já traz essa tag predefinida, por ser, como

dissemos, um elemento importante na criação de jogos diversos. Por último, adicionaremos um script, também chamado GameController, ao nosso GameController. No fim, teremos um objeto chamado GameController, com a tag GameController e um script chamado GameController atrelado. Legal!

3.1 Alterando o Script do GameController para Adicionar Transição Entre Cenas

O script do GameController será o responsável por gerenciar os níveis de nosso jogo, bem como as chances restantes possuídas pelo nosso personagem e o consequente fim de jogo quando o jogador chegar ao final dessas tentativas. Para começar, adicionaremos ao nosso GameController a capacidade de reiniciar o nível quando o personagem se quebrar. Em seguida, controlaremos a quantidade de vidas restantes.

Vejamos, na **Listagem 4**, o código do nosso script GameController.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5
6  public class GameController : MonoBehaviour {
7
8      private string sceneName;
9
10     // Use this for initialization
11     void Start () {
12
13     }
14
15     // Update is called once per frame
16     void Update () {
17
18     }
19
20     public void Break () {
21         Invoke("RestartLevel", 3f);
22     }
23
24     private void RestartLevel () {
25         sceneName = SceneManager.GetActiveScene().name;
26         SceneManager.LoadScene(sceneName);
27     }
28 }
```

Listagem 4 - Script GameController.cs, responsável pelo controle do jogo.
Fonte: Elaborada pelo autor.

Oba! Métodos novos! Que maravilha! O que são?

Bom, vamos aos principais destaques! Para começar, temos um novo import! Acredito ser a primeira vez que precisamos trazer um novo pacote ao nosso script para fazê-lo funcionar. Com a palavra-chave **using**, trazemos ao nosso script o pacote **UnityEngine.SceneManagement**, responsável por realizar o controle de cenas no Unity. É nesse pacote que encontramos os dois outros métodos novos a serem trazidos ao nosso código – o GetActiveScene() e o LoadScene().

O método GetActiveScene, do SceneManager, é responsável por buscar e fornecer informações sobre a cena atual na qual o jogo se encontra. Uma dessas informações é o parâmetro **name**, que salvamos na variável *sceneName* a fim de utilizar como referência para o método seguinte, chamado LoadScene.

O método LoadScene, por sua vez, é responsável por carregar uma nova cena, a qual possua o nome igual a string que lhe é passada na chamada. Também é possível chamar esse método utilizando um número, o qual faz referência ao código da cena na ordem de cenas. Veremos mais sobre isso adiante.

Esse carregamento de cena, no entanto, é imediato! Quando chamamos esse método, ocorre a mudança para a nova cena. Nesse caso, a animação de quebra do robozinho não seria sequer executada! Então, para não termos esse problema, o nosso método Break, na verdade, apenas invoca, com um delay de três segundos, o método RestartLevel, que faz, de fato, a troca de cenas. Já havíamos conhecido o método Invoke anteriormente, então ele não deve ser uma novidade.

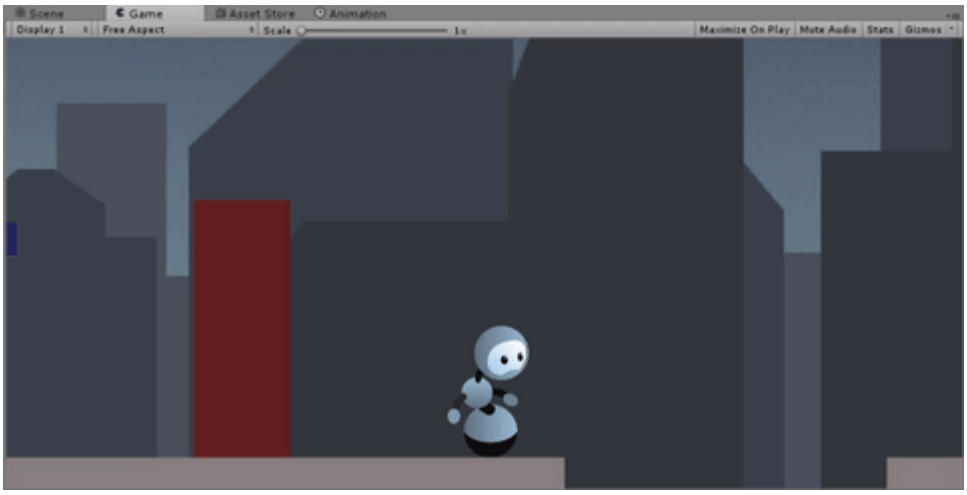
Excelente! Já temos o nosso GameController e ele já faz a troca de cenas. Só precisamos de mais uma coisa para ele funcionar adequadamente: precisamos que a troca de cenas seja solicitada! E, para isso, podemos alterar a nossa função de quebra do personagem, a fim de ela também chamar a quebra do GameController! Para tanto, utilizaremos um método novo chamado FindObjectOfType<GameController>(). Vejamos, na **Listagem 5**, o código alterado do método Break do script PlayerController.

```
public void Break () {
        active = false;
        rigidBody.bodyType = RigidbodyType2D.Static;
        ani.SetBool("active", false);
        ani.Play("Break");
        FindObjectOfType<GameController>().Break();
}
```

Listagem 5 - Método Break do PlayerController chamando o método Break do GameController.
Fonte: Elaborada pelo autor.

O método FindObjectOfType recebe um tipo de objeto e retorna o primeiro objeto daquele tipo que for encontrado. Como só temos um GameController, o FindObjectOfType retornará o nosso GameController. A partir daí, podemos chamar normalmente qualquer um de seus métodos públicos, e é exatamente isto que faremos: chamaremos o método Break. Agora, quando o nosso personagem tocar a animação de quebrar, ele também chamará o método de reiniciar o nível, resultando em algo como visto na **Figura 8**.

Figura 08 - Nível reiniciando após o personagem quebrar.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Viram? Ele está bem! Obrigado, colega de mecatrônica, por ter consertado o nosso robô!

Somente com essa modificação, adicionando o GameController ao nosso jogo, o personagem já é capaz de retornar ao início da fase. Bacana, não? Mas ainda há outras coisas que o GameController pode fazer. Vamos aprender um pouco mais?

3.2 Adicionando a Contagem de Chances Restantes do Jogador

É muito bom ver o nosso amigo robô de volta após sofrer aquele pequeno acidente no buraco! Mas também há limites para tudo. Em algum momento, as chances de passar do nível se esgotarão e o jogo chegará ao seu fim, como todo jogo. E como será que podemos contar as chances restantes?

A resposta é simples: com o GameController!

Podemos adicionar uma nova variável ao script do nosso GameController, a qual conte quantas chances restam ao jogador para concluir o nível e, sempre que houver uma quebra, essas chances serão diminuídas. Quando chegarmos a zero chances restantes, o jogo deverá se encerrar com uma cena de Game Over.

Para isso, criaremos uma nova variável inteira chamada robotsLeft. Iniciaremos o jogo com 3 chances! Além disso, caso as chances se esgotem, alteraremos, para ir à Game Over, o código de carregar a próxima cena. O código alterado pode ser visto na **Listagem 6**.

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

public class GameController : MonoBehaviour {

    private int robotsLeft = 3;
    private string sceneName;
    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }

    public void Break () {
        robotsLeft -= 1;
        Debug.Log(robotsLeft);
        Invoke("RestartLevel", 3f);
    }

    private void RestartLevel () {
        sceneName = SceneManager.GetActiveScene().name;
        if (robotsLeft > 0) {
            SceneManager.LoadScene(sceneName);
        } else {
            SceneManager.LoadScene("GameOver");
        }
    }
}
```

Listagem 6 - Alteração do código do script GameController realizada para obtermos a contagem das chances restantes.
Fonte: Elaborada pelo autor.

Adicionamos, ao nosso jogo, a contagem de robôs restantes e, para o caso de as tentativas se encerrarem, também adicionamos a troca de cenas pela cena de Game Over. Optamos, ainda, por colocar uma chamada ao Log indicando quantas tentativas restam a cada vez que o jogo for reiniciado, para podermos saber quantas chances ainda temos. Vejam o resultado disso ao executarem o jogo! Testem!

Vídeo 01 - Jogo em Execução

Ops... Estranho, não? Como pode o jogador perder uma chance, ficar com duas, perder mais uma e ficar com... duas?

Pois é! Isso acontece devido ao reinício da cena! Quando o robô quebra, nós vamos até o SceneManager e solicitamos que a cena seja carregada novamente, certo? E o que acontece quando a cena é carregada novamente? Os scripts também são carregados novamente! Assim, os valores iniciais são retornados! Como podemos resolver isso?

3.3 Utilizando o Padrão Singleton a fim de Manter Objetos Entre Cenas

O Unity tem um método para lidar com esses problemas gerados ao trabalharmos com objetos entre cenas, o **DontDestroyOnLoad()**. Esse método, como o nome sugere, indica que um objeto não deve ser destruído quando houver o carregamento de uma nova cena. Utilizando-o, podemos manter o nosso GameController vivo entre cenas, ou mesmo após o reinício da mesma cena.

Desse modo, mantendo vivo o nosso objeto controlador, podemos manter valores importantes para o andamento do jogo intactos, mesmo se houver um avanço ou reinício de cena. Podemos, por exemplo, manter a quantidade de chances

restantes atualizada e, assim, chegar até o final, caso elas sejam esgotadas.

No entanto, há um outro problema sutil ocasionado ao utilizarmos essa função. Quando iniciamos uma cena, todos os objetos contidos nela são carregados, incluindo o objeto que nós mandamos não destruir. Se a recarregarmos em sequência, o objeto não é destruído, mas um novo é criado, em razão de ela ter sido recarregada. E se a cena for recarregada de novo, o que acontece? Vê o problema?

Há, porém, uma solução simples para isso! Podemos implementar, para o objeto a ser mantido entre as cenas, um padrão Singleton. Esse padrão nos diz que um objeto é único e garante não haver, a todo momento da existência desse objeto, outro objeto igual ativo em nosso programa. Queremos exatamente isso, não é?

A fim de implementar o padrão Singleton, podemos utilizar o método Start() do script do GameController, uma vez que esse método é chamado sempre na inicialização. E então, dentro dele, testamos se o objeto criado é o primeiro ou não. Caso seja, deixamos o objeto como Singleton e o mantemos através da função DontDestroyOnLoad(). Caso não seja o primeiro, simplesmente destruímos o novo criado, garantindo que só o primeiro passará adiante. Entendeu bem? Não é complicado! Basta reler algumas vezes e a informação será assimilada. Qualquer dúvida, é só passar no fórum! Vejamos esse código alterado para o GameController script na **Listagem 7**.

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

public class GameController : MonoBehaviour {

    private int robotsLeft = 3;
    private string sceneName;
    public static GameController instance = null;

    // Use this for initialization
    void Start () {
        if (instance == null) {
            instance = this;
        } else if (instance != this) {
            Destroy(gameObject);
        }
        DontDestroyOnLoad(gameObject);
    }

    // Update is called once per frame
    void Update () {

    }

    public void Break () {
        robotsLeft -= 1;
        Debug.Log(robotsLeft);
        Invoke("RestartLevel", 3f);
    }

    private void RestartLevel () {
        sceneName = SceneManager.GetActiveScene().name;
        if (robotsLeft > 0) {
            SceneManager.LoadScene(sceneName);
        } else {
            SceneManager.LoadScene("GameOver");
        }
    }
}
```

Listagem 7 - Código alterado do script do GameController, incluindo o padrão Singleton e o método DontDestroyOnLoad utilizado para manter o objeto vivo entre diferentes níveis do jogo.
Fonte: Elaborada pelo autor.

Vejam a utilização, na definição do objeto, de algumas palavras-chave que acompanham vocês desde as aulas de Programação Estruturada e Programação Orientada a Objetos – o **static** para manter o campo entre objetos, e o **this** para se referir ao próprio objeto.

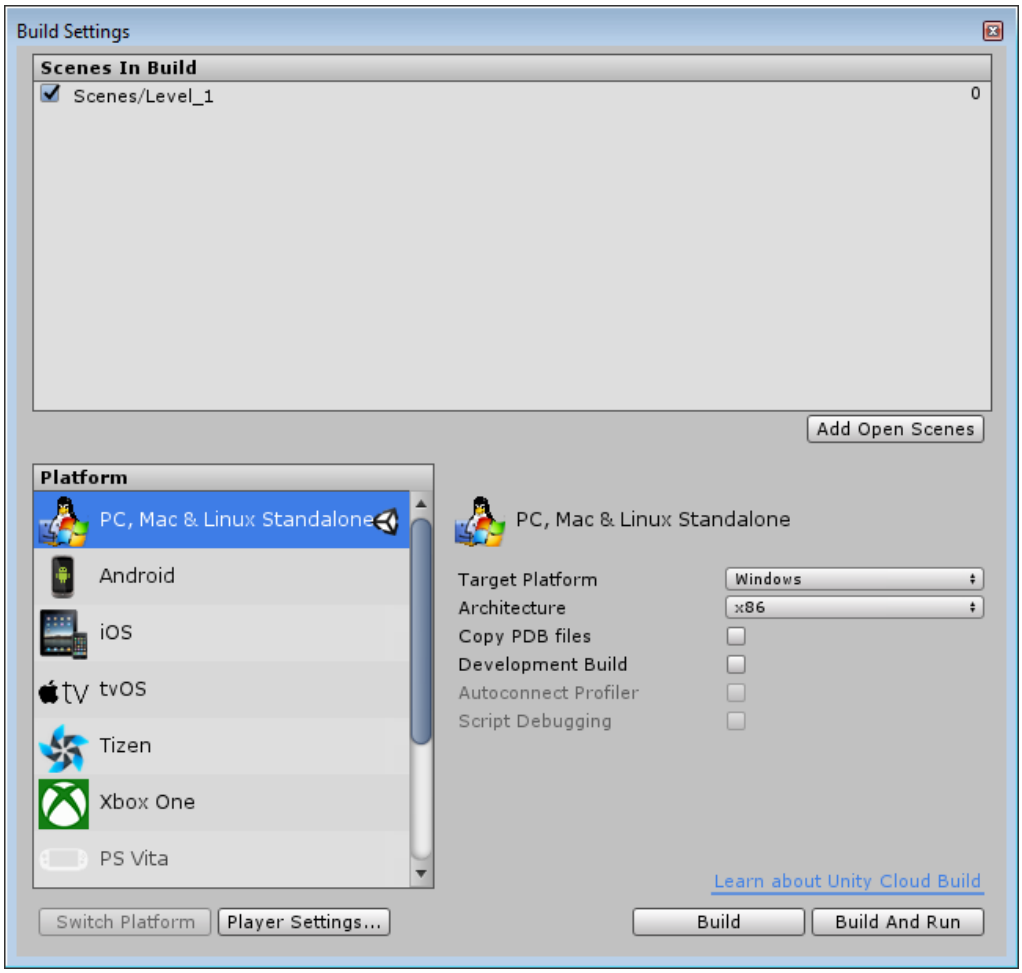
Pronto! Com esse código, já podemos limitar a quantidade de tentativas que o jogador terá antes de ser apresentado à tela de Game Over. Para isso funcionar, falta apenas criarmos a cena de Game Over.

3.4 Criando a Cena de Game Over e Adicionando-a ao Build

A criação de novas cenas não é nenhuma novidade para nós. Já fazemos isso desde a Aula 01 e não tem nenhum mistério. É necessário atentarmos, no entanto, para um outro detalhe importante relativo ao momento no qual passamos a lidar com múltiplas cenas em um mesmo projeto: precisamos que todas as cenas ativas sejam adicionadas ao **Build** para serem compiladas com sucesso e poderem ser acessadas quando forem chamadas!

A fim de fazer isso, basta clicar em File -> Build Settings, ou utilizar o atalho (Ctrl + Shift + B). Desse modo, uma tela com algumas configurações de compilação será exibida. A compilação de jogos e os diferentes Build Targets serão assuntos de nossa Aula 15, então não os detalharemos agora. O que precisamos atentar, nessa tela, é em relação às cenas presentes no build. Inicialmente, essa lista deve estar vazia. Clicando no botão Add Open Scenes, a cena atual deve ser colocada na lista, com o índex 0, indicando que é a primeira cena a ser carregada quando o jogo for iniciado. Veja na **Figura 9**.

Figura 09 - Build Settings com a cena atual adicionada.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 14 de mar. de 2017.

Agora que já temos a cena inicial na lista, podemos fazer o mesmo processo para a cena de Game Over. Para tanto, antes, temos de criá-la! Salve as alterações feita em sua cena e vá até o menu File -> New Scene, a fim de criar uma nova cena. Após criá-la, salve-a, com o nome GameOver, em nossa pasta de Scenes, dentro dos Assets. Feito isso, abra novamente a tela da **Figura 9** e adicione a cena ao Scenes in Build.

Perceba que há mais de uma maneira de fazer isso. Também é possível navegar nos assets até a pasta de Scenes e, então, clicar e arrastar os arquivos de cena para a janela mostrada na **Figura 9**, de modo a adicionar, também, as cenas ao build.

Com isso, já conseguimos, ao perder três chances em nossa cena inicial, ir até a cena de Game Over. Essa cena, por enquanto, é apenas uma cena vazia. Veremos como mudar isso na próxima aula, na qual estudaremos as interfaces gráficas no

Unity.

3.5 Adicionando uma Condição de Vitória

Mas chega de derrotas, não é? Falaremos agora de vitórias! Precisamos, ao chegar no fim do nível adequadamente, vencer! Avançar de fase! E celebrar! Faremos isso agora.

Por sorte, já temos um objeto responsável por representar o fim da fase. É o nosso BoundaryEnd. Adicionaremos, a esse objeto, um script capaz de avisar ao nosso GameController e ao nosso Player quando a condição de vitória for atingida. Nomeie esse script EndBoundaryTrigger. O código dele, similar ao já desenvolvido, pode ser visto na **Listagem 8**.

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

public class EndBoundaryTrigger : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    void OnCollisionStay2D (Collision2D col) {
        if (col.gameObject.CompareTag("Player")) {
            if (col.gameObject.GetComponent<PlayerController>().isGrounded()) {
                col.gameObject.GetComponent<PlayerController>().LevelEnd();
            }
        }
    }
}
```

Listagem 8 - Código do script EndBoundaryTrigger.
Fonte: Elaborada pelo autor.

Perceba que o script tem duas diferenças em relação ao da fronteira inferior. Nesse caso, estamos usando o método OnCollisionStay2D. Esse método é chamado enquanto uma colisão estiver ocorrendo. O motivo disso é que queremos iniciar a animação de vitória apenas quando o personagem estiver no chão, então,

detectamos a colisão e só disparamos o evento quando o personagem estiver lá. Isso é testado através do método já utilizado anteriormente, o isGrounded(), do próprio Player.

Mais uma vez, precisamos adicionar ao Player e ao nosso GameController os métodos relacionados ao fim do level. Esses métodos podem ser vistos nas Listagens 9 e 10, respectivamente.

```
1  public void LevelEnd () {
2      active = false;
3      rigidBody.bodyType = RigidbodyType2D.Static;
4      ani.SetBool("active", false);
5      ani.Play("CelebrationRoll");
6      FindObjectOfType<GameController>().LevelEnd();
7  }
```

Listagem 9 - Método LevelEnd adicionado ao script PlayerController.
Fonte: Elaborada pelo autor.

```
1  public void LevelEnd () {
2      robotsLeft += 1;
3      Debug.Log(robotsLeft);
4      Invoke("NextLevel", 3f);
5  }
6
7  private void NextLevel () {
8      int sceneIndex = SceneManager.GetActiveScene().buildIndex;
9      SceneManager.LoadScene(sceneIndex+1);
10 }
```

Listagem 10 - Métodos de LevelEnd adicionados ao script do GameController.
Fonte: Elaborada pelo autor.

Percebam que o método do PlayerController é bem similar ao já desenvolvido antes, mudando apenas a animação que será invocada do Animation Controller. Já o método do GameController varia um pouco.

No caso do GameController, estamos utilizando uma nova abordagem para avançar a fase. Buscamos a cena ativa, da mesma maneira como fazíamos no outro método, mas agora pegamos o índex dessa cena, e não o seu nome. Em seguida, carregamos a cena com a outra versão do LoadScene, o qual recebe um número como parâmetro, e então passamos o índex da cena ativa acrescido em uma unidade. Ou seja, carregamos a próxima cena de acordo com o Build Settings.

Como só temos em nosso Build Settings a cena principal do jogo, como a cena 0 e a cena de Game Over em seguida, como a cena 1, esse método também nos levará à cena de Game Over. Mas isso pode ser mudado, concorda?

3.6 Criando uma Nova Cena para o Jogo

Para finalizar a aula de hoje, adicionaremos uma nova cena ao nosso jogo, que será executada logo após o personagem chegar ao final da primeira cena. A fim de fazer isso, abra mais uma vez a cena inicial, chamada Level_1.

Feito isso, vá no menu File -> Save Scene As... e, então, salve a cena como Level_2, na pasta de Scenes, no nosso diretório Assets. Depois, adicione a nova cena ao Build Settings, logo após a cena Level_1. Para mover a ordem no Build Settings, basta clicar e arrastar.

E fim! Agora, quando chegarmos ao final do nível, celebraremos um pouco e, em seguida, seremos transferidos para a segunda fase de nosso jogo! Muito legal, não? Ah! E ainda ganharemos mais uma chance ao passar de fase, aumentando o nosso contador.

Pode ser que alguns digam que a segunda fase está um pouco parecida com a primeira. Não ligue para eles! É só intriga da oposição! As duas fases são bem diferentes uma da outra e ambas são muito divertidas!

Assim, finalizamos a nossa aula de hoje! Até uma próxima oportunidade, em que desenvolveremos interfaces para o nosso jogo, afinal, não é legal ter uma tela de Game Over na qual nada existe, e também não é bom contar a quantidade de tentativas restantes utilizando um Log do Console, não é?

Até lá, meu povo! o/

Leitura Complementar

Documentação Oficial Sobre o SceneManager - https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html

Desenvolvimento do GameController em um Jogo do Unity - https://unity3d.com/pt/learn/tutorials/topics/2d-game-creation/adding-game-controller

Resumo

Na aula de hoje, adicionamos duas novas animações ao nosso personagem. Essas animações foram utilizadas em situações de encerramento da fase e, por isso, foram adicionadas ao controlador sem qualquer transição entre elas e os outros elementos.

Em seguida, criamos um objeto responsável por fazer a parte inferior da fase e adicionamos a ele um script capaz de detectar a queda do personagem e, então, ativar uma quebra, que inclui animação, perda de vida e reinício do nível.

O reinício do nível, a contagem de vidas e a passada de fases ao chegar ao final do nível foram todos comportamentos também adicionados na aula de hoje através da criação de um objeto bem especial, chamado GameController. Vimos, também, a importância de esse objeto ser um Singleton, para o caso do nosso jogo, e aprendemos como implementar esse padrão no Unity.

Por fim, criamos duas novas cenas e navegamos entre elas utilizando o SceneManager, componente do Unity responsável por lidar com troca de cenas e de informações sobre a cena atual. A cena de GameOver, no entanto, ficou vazia e será populada na próxima aula, quando estudarmos as interfaces gráficas no Unity. Até lá!

Ah! O projeto desenvolvido pode ser obtido aqui!

Autoavaliação

1. O que é o GameController e como ele deve ser utilizado?
2. O que é o SceneManager e qual a sua principal função?
3. Qual a diferença entre os métodos LoadScene(string) e LoadScene(int)?
4. Como podemos manter um objeto vivo entre cenas?
5. Como podemos reiniciar uma cena sem haver a recriação de um objeto que não foi destruído de uma cena para outra?

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.