Desenvolvimento com Motores de Jogos I

Aula 10 - Desenvolvendo a Primeira Fase - Parte 2

Apresentação

Fala, galera! Tudo certo com a primeira parte de nossa fase? Vimos, na aula passada, a criação de alguns elementos de nosso nível, abordando duas maneiras diferentes de tratar o problema: a duplicação dos elementos na cena e a criação de Prefabs.

Vimos as vantagens e desvantagens de ambas as maneiras e, também, o modo como podemos explorar cada uma destas. Finalizamos a aula adicionando ao nível nossas primeiras plataformas funcionais na forma de Prefabs - as caixas. Agora, veremos novas modificações para o cenário.

Começaremos esta aula adicionando novos tipos de plataforma e criando os Prefabs necessários a partir dos modelos que fizermos, posicionando-os de acordo com o que vemos em nosso esboço, na **Figura 1**.

Figura 01 - Protótipo manual do primeiro nível a ser desenvolvido em nosso projeto.

Fonte: Elaborada pelo autor.

Após todas as plataformas estáticas ficarem em seus lugares, posicionaremos as plataformas móveis. Veremos scripts para a criação de plataformas móveis e de plataformas que caem ao ter o contato do jogador, como vemos em diversos jogos de plataforma. Estudaremos, também, como é possível reproduzir o comportamento dessas plataformas utilizando simplesmente animações.

Animações? É isso mesmo! É possível criar e movimentar elementos com comportamentos variados, já predefinidos, simplesmente a partir de animações! Legal, não? Mas será que esses elementos ficarão iguais aos criados através de scripts ou serão as nossas possibilidades alteradas de acordo com a maneira que utilizarmos para movimentar esses elementos?

Veremos isso ao longo de nossa aula, bem como algumas alterações para o nosso personagem! Teremos muita prática nesta parte do curso! Dispostos? Vamos lá!

Objetivos

Ao final desta aula, você deverá ser capaz de:

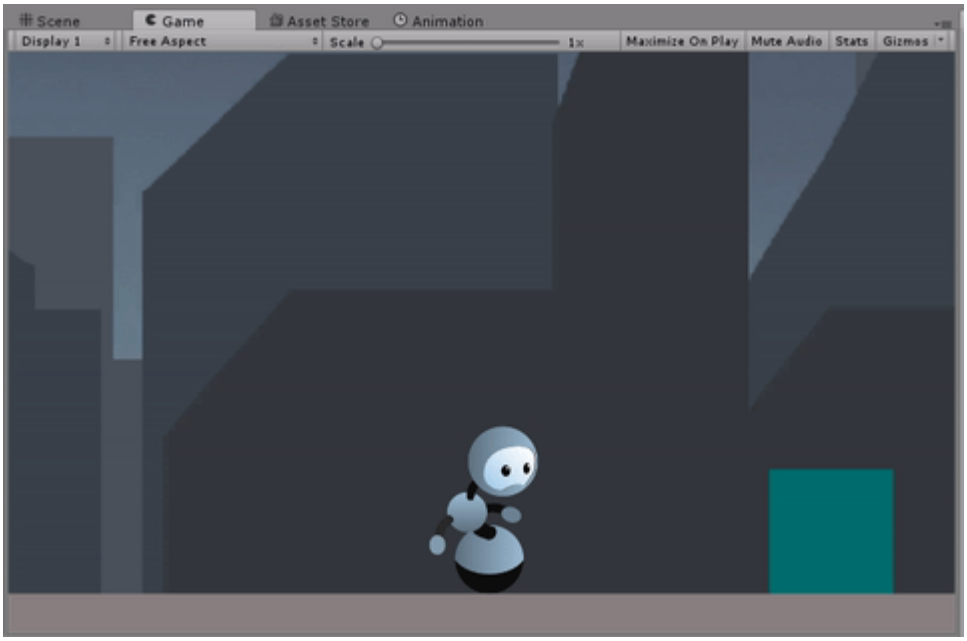
- Adicionar novos comportamentos às plataformas de seu jogo através de scripts.
- Criar Prefabs que possuam scripts e animações.

1. Redefinindo o Personagem

Na aula passada, aumentamos bastante a nossa cena e adicionamos as primeiras plataformas ao nosso jogo, dando ao nosso personagem, pela primeira vez, a real possibilidade de navegar por um cenário, por meio da utilização das mecânicas que já havíamos adicionado de pulo e pulo duplo, além de sua animação.

Alguns problemas, no entanto, surgiram a partir dessa navegação. Devido à maneira que construímos o nosso personagem, alguns pequenos problemas estão acontecendo quando navegamos pelo cenário, como alguns de vocês podem ter notado ao tentar jogar o que criamos na aula passada. A **Figura 2** demonstra o primeiro desses problemas a ser solucionado, o qual é, também, o mais complicado.

Figura 02 - Personagem grudando na plataforma, ao se movimentar na direção dela.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar. de 2017.

Como vemos na **Figura 2**, ao fazermos o personagem pular na direção de uma plataforma e segurarmos o botão de movimentação nessa direção, o personagem simplesmente gruda na plataforma, deixando de cair, subir, ou se movimentar de qualquer maneira.

Não entraremos em detalhes de como isso acontece, mas a razão é puramente física! Como o personagem está exercendo uma força contra a plataforma e ambos possuem uma grande força de atrito, ele simplesmente fica colado na plataforma, devido a esse atrito.

> Esse comportamento é bem estranho e muito diferente do que costumamos ver em jogos de plataforma. Ele pode ser considerado, na verdade, um *bug*. Como podemos corrigi-lo sem precisar alterar muito tudo já feito? Aprenderemos agora como resolver isso!

Uma das maneiras mais simples de resolver esse problema seria alterando o atrito do personagem para zero. Assim, o personagem não teria a possibilidade de ficar grudado na plataforma ou em qualquer outra parte do cenário. Isso, no entanto, traz algumas consequências à jogabilidade. Se o personagem não tem atrito, deslizará por todo o cenário, tornando-se muito mais difícil de controlar, principalmente para acertar plataformas pequenas, como as próprias caixinhas. Por essa razão, precisaremos também alterar um pouco o script de movimentação do personagem, a fim de facilitar o controle tido pelos jogadores.

Antes de começar as alterações, ressaltamos que é disponibilizado aqui o projeto como estava no fim da aula passada. É importante vocês baixarem e acompanharem, se ainda não estiverem fazendo isso! Vai lá, agora! ;)

1.1 Removendo o Atrito do Personagem

O primeiro passo, como dissemos, será a remoção do atrito do personagem, para ele poder, mesmo se movendo contra alguma plataforma, sofrer a ação da gravidade sempre que não estiver no chão.

A fim de executar essa alteração, a primeira ação a ser feita é criar um novo material físico 2D capaz de realizar o que estamos buscando. Vimos um pouco sobre isso na aula a respeito do motor de física, então não deveremos ter qualquer dificuldade. Caso não lembre bem como isso funciona, recomendo dar uma olhada nessa aula!

O primeiro passo, lembrando sempre da importância de organizar os nosso assets, é criar uma nova pasta para podermos guardar os materiais físicos. Alguns já a criaram em oportunidades anteriores. Criaremos uma nova pasta, dentro de nossa pasta Assets, chamada Physics Materials. Dentro dela, clicaremos com o botão direito e selecionaremos a opção Create -> Physics Material 2D. Utilizei o nome Robot para o meu material.

Em seguida, clique no novo material criado e, então, no Inspector, veremos as duas variáveis possíveis de serem alteradas para o material. A primeira delas é justamente a que queremos - Friction. Ao alterar a Friction para 0, o nosso personagem não colará mais em qualquer objeto do cenário... Nem no chão! Basta adicionar o material ao RigidBody do nosso personagem. Aos que não lembram, é necessário somente selecionar o personagem e, no Inspector, no componente RigidBody, selecionar o Material na propriedade de mesmo nome. Também é possível arrastar o Material até a propriedade, para defini-lo como material utilizado. Vejamos na **Figura 3** o resultado da alteração feita.

Figura 03 - Personagem grudando na plataforma, ao se movimentar na direção dela.

Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar. de 2017.

Wow! Passamos direto corretamente! E ainda sobrou um pouco de energia a fim de deslizarmos um pouco mais pelo cenário após cairmos do outro lado da plataforma. Para um cenário de gelo, essa ideia até parece interessante, não? Mas para um cenário regular, como o que temos, não será legal.

Objetivando que o personagem não mais deslize por tanto tempo, alteraremos um pouco o nosso script de controle do personagem. Mais precisamente, alteraremos a parte de movimentação para ele se movimentar exatamente de acordo com a utilização da tecla de movimento, sem a atuação de forças.

1.2 Alterando o Script de Controle

Agora que o nosso personagem está deslizando por todo o cenário, precisamos alterar o script de movimentação, a fim de evitar a utilização de forças. Essa alteração substituirá, por um comando mais simples, toda a parte relacionada à adição de força (rigidBody.AddForce) e controle de velocidade máxima (rigidBody.velocity), sendo responsabilidade desse comando transmitir diretamente a tecla pressionada pelo jogador para a velocidade do nosso personagem. Da mesma maneira, quando não houver tecla pressionada, não haverá movimentação. A **Listagem 1**, a seguir, demonstra essa alteração, no método FixedUpdate do nosso script PlayerController.cs.

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

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

    grounded = Physics2D.Linecast(transform.position, groundCheck.position,
    layerMask);

    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 (rigidBody.velocity.x * moveH < maxSpeed) {
       rigidBody.AddForce (Vector2.right * moveH * acceleration);
    }

    if (Mathf.Abs (rigidBody.velocity.x) > maxSpeed) {
       Vector2 vel = new Vector2 (Mathf.Sign (rigidBody.velocity.x) * maxSpeed,
       rigidBody.velocity.y);
       rigidBody.velocity = vel;
    }*/

    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;
    }

 }
```

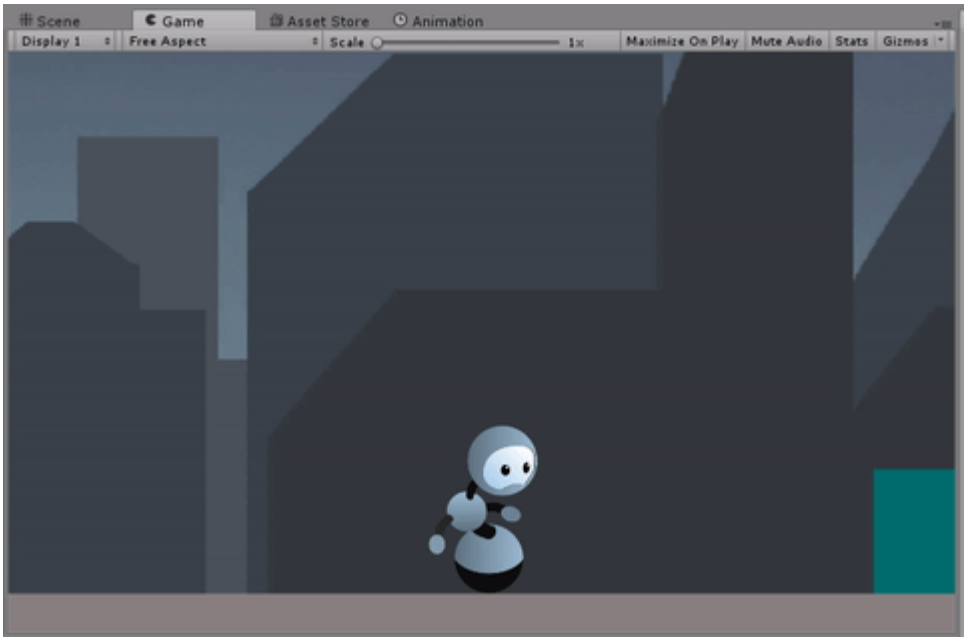
Listagem 1 - Alteração na movimentação do personagem.
Fonte: Elaborada pelo autor.

Pronto! Comentamos toda a parte que era anteriormente relacionada à parte de movimentação, e adicionamos uma nova linha, a destacada em negrito, ao nosso script.

Essa linha altera diretamente a velocidade em X do personagem, através do atributo rigidBody.velocity, para uma multiplicação entre o maxSpeed e o moveH, sendo este último o valor do eixo horizontal do controle do jogador. Com essa alteração, a velocidade do personagem será máxima quando o botão estiver apertado totalmente, e zero quando o botão for solto. Assim, mesmo com o atrito zero que adicionamos ao nosso personagem, quando não houver botão pressionado, ele não se movimentará no eixo X.

Perceba, ainda, que o Vector3 criado a fim de alterar a velocidade utiliza, para o valor de Y, rigidBody.velocity.y. Ou seja, alteramos o valor em X da velocidade para ser diretamente relacionado ao controlador, mas, para o eixo Y, a velocidade se mantém inalterada, tornando-se obrigação da gravidade, dos buracos e do pulo. Separamos bem os eixos e conseguimos uma movimentação um pouco diferente! Vejamos, na **Figura 4**, o resultado dessa nova movimentação, a qual afetará também o personagem no ar.

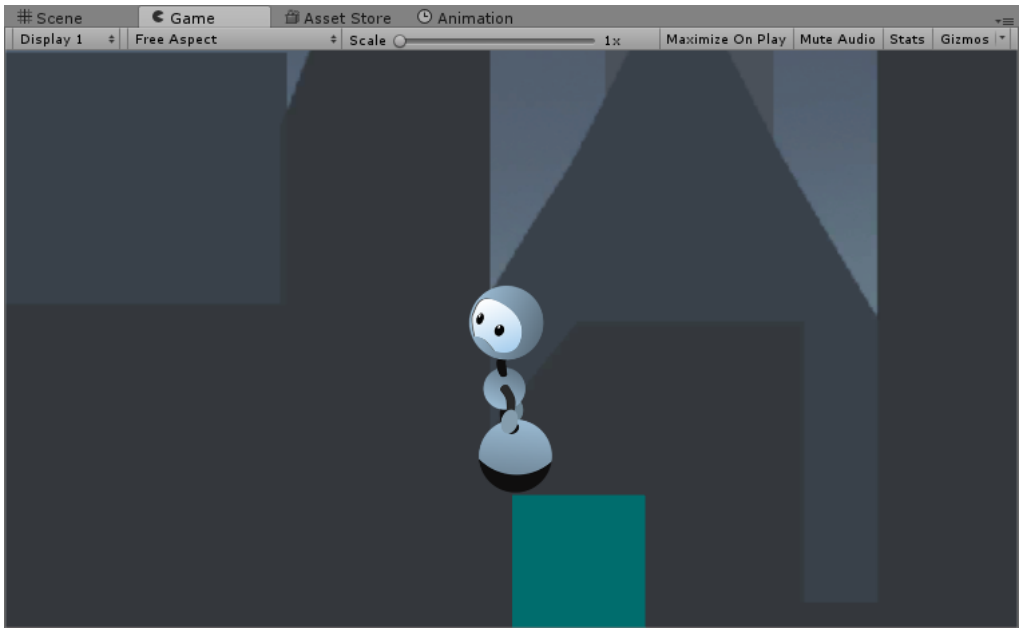
Figura 04 - Personagem se movimentando com o novo controle.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar. de 2017.

Ok! Tudo certo, agora, não é? Nosso personagem parou de grudar em nossas plataformas e, além disso, está se movendo de uma maneira nova e bem adequada ao que queremos. Só tem mais um probleminha, no entanto... Vejamos a **Figura 5**!

Figura 05 - Personagem na borda de uma plataforma.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar. de 2017.

Como assim? O personagem está em cima de uma plataforma! Por que ele está na posição de pulo? O que está acontecendo?

Está acontecendo que o groundCheck do nosso personagem já saiu de cima da plataforma, apesar de o personagem ainda estar em cima dela! Desse modo, o nosso Line Casting simplesmente passa direto pela plataforma, sem contato, fazendo o personagem não achar que está no chão. Isso pode complicar algumas situações, principalmente devido ao fato de o personagem precisar estar no chão para conseguir pular.

“E como podemos corrigir isso?” Bem, podemos alterar a maneira como a detecção de chão ocorre! “E como a alteramos, professor?” Fica tranquilo, trataremos disso a seguir! ;)

1.3 Alterando o GroundCheck

Na aula sobre pulos, vimos uma das maneiras de fazer o groundCheck, lembra? A outra foi deixada ao final como curiosidade, para vocês poderem conhecer as duas. Aos que foram curiosos e deram uma olhada por lá, ótimo! Aos que ainda não viram, discutiremos rapidamente a alteração necessária.

Em nosso script, utilizamos um LineCast do centro do personagem ao GroundCheck, para definir se estamos ou não no chão. Porém, a área coberta para isso é apenas uma linha! Sendo assim, os lados do nosso personagem terminam ficando de fora da equação, como visto na **Figura 5**. A outra maneira de fazermos esse teste, no entanto, consegue cobrir uma área maior.

A biblioteca Physics2D, que possui o método LineCast utilizado em nosso script, também possui métodos para definir áreas de contato, ao invés de um simples ponto. Assim, podemos definir, por exemplo, uma caixa que fique em torno do ponto groundCheck, o qual definimos anteriormente, e buscar, dentro dessa área, o contato com a camada desejada, ao invés de utilizar apenas um ponto. Com isso, podemos cobrir uma maior parte do personagem e facilitar a detecção de chão. O método que utilizaremos para isso é o OverlapBox, visto na **Listagem 2**, o qual altera, mais uma vez, o FixedUpdate do script PlayerController.

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

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

    //grounded = Physics2D.Linecast(transform.position, groundCheck.position, layerMask);

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

    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 (rigidBody.velocity.x * moveH < maxSpeed) {
        rigidBody.AddForce (Vector2.right * moveH * acceleration);
    }

    if (Mathf.Abs (rigidBody.velocity.x) > maxSpeed) {
        Vector2 vel = new Vector2 (Mathf.Sign (rigidBody.velocity.x) * maxSpeed,
                rigidBody.velocity.y);
        rigidBody.velocity = vel;
    }*/

    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;
    }

}
```

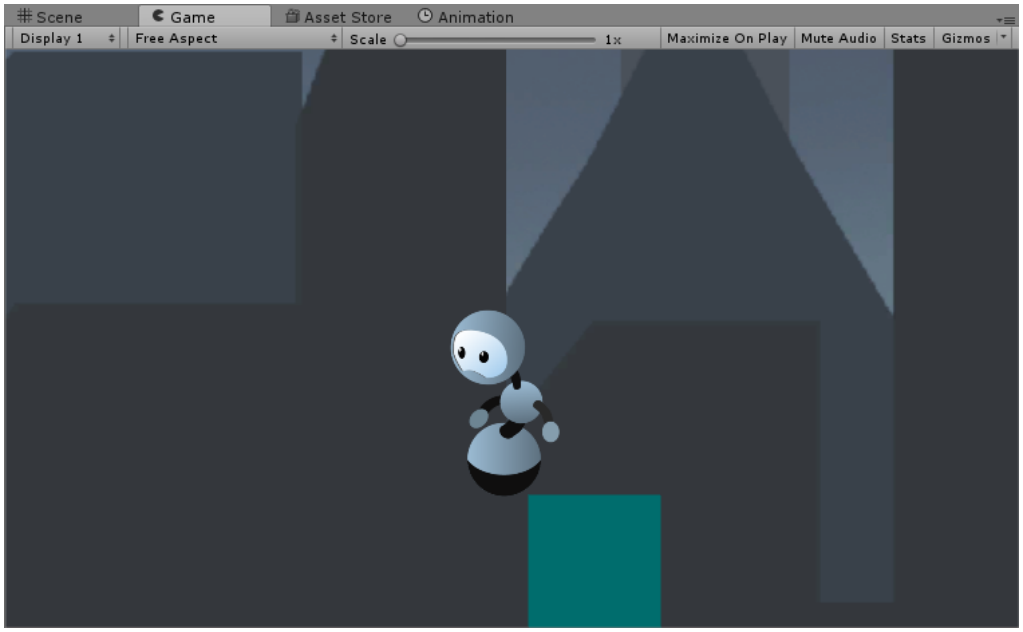
Listagem 2 - Adicionando o OverlapBox ao FixedUpdate do script PlayerController.
Fonte: Elaborada pelo autor.

O método OverlapBox, da biblioteca Physics2D, utiliza quatro parâmetros para definir a área procurada. O primeiro parâmetro indica a posição em que a caixa (box) será definida. Utilizamos a posição do nosso objeto GroundCheck (groundCheck.position). Em seguida, definimos o tamanho da caixa. Isso tem de ser

um pouco experimental. Após alguns testes, chegamos ao tamanho (1.3, 0.2). Esse tamanho é definido através de um Vector2. Depois, temos o ângulo no qual a caixa deve ser criada. Utilizamos zero para esse parâmetro, a fim de a caixa estar sem qualquer rotação. Por fim, definimos a camada de interesse para a qual a caixa deve reportar quando houver colisão. Nesse caso, utilizamos a camada ground, representada pela mesma LayerMask que já utilizávamos anteriormente.

Com isso, o nosso personagem agora busca o chão em uma pequena área quadrada ao redor do GroundCheck, e não mais utilizando apenas uma linha. Esse método é um pouco mais caro, computacionalmente, mas também produz um melhor resultado visual, como podemos notar na **Figura 6**.

Figura 06 - Personagem detectando a plataforma adequadamente.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar. de 2017.

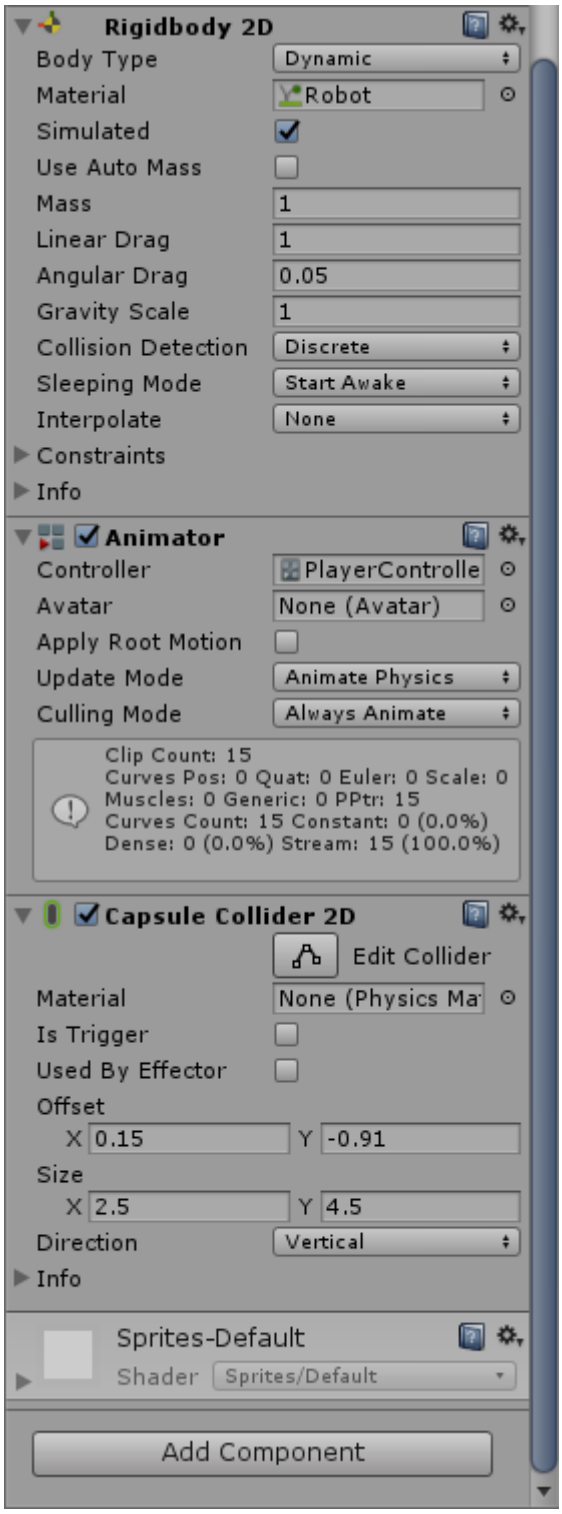
Perceba que há uma pequena diferença entre o colisor e a plataforma. Essa diferença pode ser ajustada alterando o tamanho do colisor, alterando o tamanho da caixa de detecção do chão ou mesmo o tipo de colisor o qual estamos utilizando. Vamos a essa última opção, como mais uma modificação a fazer no personagem.

1.4 Alterando os Colisores do Personagem

Em nossas primeiras aulas, resolvemos utilizar dois colisores para cobrir todo o corpo do personagem - um Box Collider 2D e um Circle Collider 2D. No entanto, esses dois colisores podem ser também representados por um Capsule Collider 2D, que vimos na aula de física, anteriormente. Vamos fazer essa alteração em nosso personagem como último elemento antes de partir para as novas adições do nível!

Primeiramente, selecione o personagem em nossa Hierarchy e remova os dois Colliders já existentes no Inspector. Para isso, basta clicar na engrenagem que há ao lado do nome de cada collider e selecionar a opção RemoveComponent. Em seguida, adicione um novo componente pelo botão Add Component -> Physics 2D -> Capsule Collider 2D. Configure o novo componente para os valores adequados, como visto na **Figura 07**, a seguir:

Figura 07 - Adicionando um Capsule Collider 2D para substituir os dois colisores anteriores.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar. de 2017.

Caso ainda ache que o personagem está se comportando de maneira estranha em relação às colisões, fique à vontade para alterar o colisor e chegar a valores que julgue mais adequados! Se encontrar algum valor mais legal, passa lá no fórum e fala para todo mundo!

1.5 Permitindo o Pulo Duplo durante a Queda

Uma outra alteração que não é necessária, mas talvez seja interessante fazermos em nosso jogo, é em relação à regra do pulo duplo. Do jeito como está o nosso código, permitimos ao personagem executar o pulo duplo sempre após um pulo simples. Entretanto, em uma situação na qual o personagem está somente caindo, não permitimos que ele pule no ar, com um pulo duplo, pois não houve antes um pulo simples.

Podemos alterar o nosso código para passar a permitir, ainda, que o segundo pulo, feito no ar, seja feito também quando o personagem estiver caindo de uma plataforma alta, por exemplo. Para isso, precisamos apenas de uma alteração simples em nosso código. Veja, na **Listagem 3**, uma pequena alteração que podemos fazer no método Update de nosso script PlayerController, a fim de permitir esse comportamento.

```
1  // Update is called once per frame
2  void Update() {
3      if (grounded) {
4          doubleJump = true;
5      }
6
7      if (Input.GetButtonDown("Jump")) {
8          if (grounded) {
9              jumping = true;
10             doubleJump = true;
11         } else if (doubleJump) {
12             doubleJumping = true;
13             doubleJump = false;
14         }
15     }
16 }
```

Listagem 3 - Permitindo que o pulo duplo seja feito após uma queda, alterando o Update do Script PlayerController.
Fonte: Elaborada pelo Autor.

Perceba termos adicionado um novo IF ao nosso método Update. Agora, se estivermos no chão, habilitamos para haver o pulo duplo. Com isso, caso o personagem caia de uma plataforma, ele não poderá pular, uma vez que não está grounded, mas poderá executar algo equivalente ao pulo duplo para subir no ar novamente, uma vez que a variável de pulo duplo estará TRUE devido a ele ter estado no chão.

Caso o personagem execute um pulo duplo, a variável volta a ser FALSE até ele voltar ao chão, evitando, assim, que ele possa executar o pulo duplo indefinidamente. Mais um comportamento opcional, o qual pode, porém, ser interessante para a configuração de nossa fase!

1.6 Finalizando o Personagem

Precisaremos fazer mais duas alterações para que todo o conteúdo a ser visto ao longo da aula funcione adequadamente. A primeira delas diz respeito a precisarmos de uma maneira de identificar o player. Para isso, existe uma tag já predefinida no Unity, uma vez que esse requisito é bem comum. Selecione o nosso Player e, no menu de Tags, atribua a ele a tag "Player".

A última alteração necessária no personagem é em sua camada de renderização. Altere a **Order in Layer** dele para 2, visto que colocaremos o nosso objetivo final na posição 1, a frente do background (0), mas atrás do personagem (agora, 2).

Assim, finalizamos as alterações necessárias! Podemos partir para a adição de novos elementos em nosso cenário. Começaremos por novas plataformas estáticas e, em seguida, adicionaremos as plataformas móveis. Passaremos mais rapidamente pelas plataformas estáticas devido à sua semelhança com as já feitas na aula passada, de modo a atentarmos mais para as plataformas móveis e seus scripts.

Importante!

Todas essas alterações que fizemos em nosso personagem, como já falamos em outras oportunidades, são feitas com o intuito de ensinar a vocês mais de uma maneira de realizar a mesma tarefa, destacando sempre quando cada uma funciona melhor. Não esqueçam o que já aprenderam e pratiquem sempre a maneira mais adequada, seja ela qual for!

2. Adicionando Novas Plataformas Estáticas

Após deixarmos o nosso personagem bem adequado para as interações com as plataformas que já havíamos adicionado, passaremos às novas plataformas de nosso cenário, adicionando mais algumas possibilidades de estratégias e tornando o nosso nível mais desafiador! Olharemos mais uma vez, na **Figura 8**, o esboço de nosso nível, a fim de vermos as plataformas que adicionaremos primeiro.

Figura 08 - Protótipo manual do primeiro nível a ser desenvolvido em nosso projeto.

Fonte: Elaborada pelo autor.

Lembrando, também, o que definimos na aula passada em relação aos tipos de componentes e plataformas (notando quais deles já concluímos):

- Backgrounds ✔
- Chãos espaçados por meio de buracos ✔
- Plataformas
    - Caixa no chão ✔
    - Plataforma fixa
    - Paredes

    - Plataformas móveis
    - Plataforma que cai
- Começo e fim da fase ✔

Precisamos, agora, trabalhar as plataformas fixas, que flutuam sobre o cenário e servem como alvo para pulos mais longos; as paredes, que são bloqueios mais altos que os outros elementos; as plataformas móveis, feitas com animação e com script; e, por fim, a plataforma que cai ao ser tocada pelo usuário. Comecemos pelas plataformas fixas!

2.1 Plataformas Fixas

A plataforma fixa será similar às caixas adicionadas na aula passada, porém, flutuarão pelo cenário, requisitando ao usuário um pulo mais alto, ou até mesmo o apoio de uma caixa, para que consiga subir.

Começaremos a criação de nossa plataforma fixa do mesmo modo como fizemos com a caixa anteriormente. Selecionaremos, na pasta Script, o script que criamos para as plataformas e o arrastaremos para o GameObject Platforms, a fim de criar um novo filho desse objeto com o componente Sprite Renderer já definido para o sprite que estamos utilizando nas plataformas. Renomeie o GameObject criado para Platform, caso já não esteja assim nomeado.

A primeira alteração que faremos nesse GameObject será em relação à sua cor. Utilizaremos a cor RGB = (37, 32, 98) ou, em Hex, #252062FF. Enquanto estivermos no componente Sprite Renderer, aproveitaremos para colocar a plataforma em sua ordem adequada, com o valor 1 para a Order in Layer. Em seguida, adicionaremos um Box Collider 2D à nossa plataforma para que seja possível colidir com ela. Já fizemos isso várias vezes, em Add Componente -> Physics 2D -> Box Collider 2D. Por fim, alteraremos a **escala em X** da plataforma para o valor **7**, pois essas plataformas são mais achatadas e compridas do que as caixas previamente definidas.

Feito isso, temos a nossa plataforma já definida. Precisamos, apenas, criar suas várias instâncias e posicioná-las adequadamente. Como já temos o objeto base concluído, podemos criar o nosso Prefab! Lembra como?

Isso! Basta arrastar o componente configurado para a pasta Prefabs e, então, nessa pasta, teremos o novo asset criado! Agora, com o asset criado, adicione duas novas instâncias da nossa plataforma à cena. Lembrando, basta arrastar o asset até o GameObject **Platforms**.

Como vimos na **Figura 8**, utilizaremos uma dessas plataformas logo no começo, após a segunda caixa, para pular a primeira parede. Em seguida, utilizaremos mais duas para subir na segunda parede do cenário, usando uma caixa para subir, mais uma vez. Com isso, totalizamos as três plataformas que devemos criar.

Como essas plataformas ficam elevadas, diferentemente das caixas, precisaremos alterar, nas três instâncias, tanto a posição em X quanto a posição em Y de cada uma delas. Os valores finais para a posição de cada uma das caixas devem ser (-170.5, 0.8, 0), (-57.3, 0.8, 0) e (-38.2, 6.5, 0). Perceba que tanto o valor X quanto o valor Y varia, fazendo-as terem posições diferentes no cenário, além de alturas também diferentes. Desse modo, o cenário deve ficar como visto na **Figura 9**.

Figura 09 - Cenário com as plataformas fixas e as caixas posicionadas.

Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Agora que posicionamos as três primeiras plataformas, adicionaremos as paredes, para essas plataformas fazerem um pouco mais de sentido em relação à maneira como estão posicionadas.

2.2 Paredes

Adicionadas as plataformas fixas e as caixas, o último componente estático que adicionaremos ao nosso cenário são as paredes. Esses componentes serão mais longos verticalmente e se posicionarão próximos às bordas de alguns dos buracos que a fase possui.

Definiremos as paredes como Prefabs, da mesma maneira como fizemos anteriormente. Utilizaremos, para criar um novo GameObject como filho do Platforms, o sprite de plataformas que criamos previamente. Dessa vez, chamaremos o novo objeto criado de Wall.

As definições que faremos ao objeto são as mesmas feitas anteriormente. Primeiramente, alteraremos a **cor** dele para, em RGB, (98, 32, 32) ou, em Hex, #622020FF. Em seguida, ainda no componente Sprite Renderer, alteraremos a **Order in Layer** para 1. Por fim, adicionaremos o **Box Collider 2D** que tratará as colisões com esse objeto.

Em relação à **posição**, alteraremos tanto o X quanto o Y da posição desses objetos, portanto, não há necessidade de definir esses campos no Prefab. Acerca da **escala**, por sua vez, utilizaremos a mesma largura para todos, com X = 3. O Y da escala também variará de acordo com o posicionamento que for feito para a parede.

Definidos a cor, a Order in Layer, o Collider, o nome e a escala do objeto, precisamos, apenas, definir a camada do objeto e, então, estaremos prontos para criar o nosso asset Prefab a partir desse componente. Escolha a **camada** Ground, para que o personagem possa detectá-lo adequadamente como chão. Clique no componente, arraste-o até a pasta Prefabs, como vimos anteriormente, e pronto! Ele estará criado.

De acordo com o que vimos na **Figura 8**, precisaremos de quatro paredes para definir o nosso cenário. A primeira será posicionada após a primeira plataforma, a segunda será após o segundo grupo de plataformas, a terceira será do outro lado do buraco, e a última será o obstáculo final, após as plataformas que caem. Com isso, precisaremos adicionar mais três instâncias do Prefab, totalizando quatro elementos do tipo Wall. O posicionamento e a escala desses elementos podem ser vistos na **Tabela 1**.

Elemento	**Posição (X,Y,Z)**	**Escala(X,Y,Z)**
Wall	(-160, -2, 0)	(3,8,1)
Wall	(-21.5,0,0)	(3,12,1)
Wall	(11.5, -2, 0)	(3,8,1)
Wall	(234, 0, 0)	(3,12,1)

Tabela 1 - Posicionamento e escala dos Objetos Wall.
Fonte: Elaborada pelo autor.

Ok! Criado o Prefab, replicadas as instâncias e posicionadas de acordo com a **Tabela 1**, teremos todas as nossas paredes adicionadas ao cenário nas posições corretas. Já adicionamos mais um pouco de dificuldade ao nosso jogo, o que o torna mais interessante de se jogar! Vejamos o resultado na **Figura 10**.

Figura 10 - Cenário com as caixas, plataformas fixas e paredes.

Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Assim, concluímos toda a parte estática de nosso cenário. Vamos, agora, às plataformas móveis! Tudo bem até aqui? Vimos muitos elementos interessantes para a dinâmica de nosso jogo, e ainda temos muito mais a conhecer nesta aula, aguenta firme! ;)

3. Adicionando Plataformas Móveis

As plataformas móveis adicionam um grau de desafio maior ao seu jogo, uma vez que o jogador depende de uma sincronização maior entre os comandos e o cenário, para conseguir alcançar o seu objetivo de atingir uma das plataformas. Adicionaremos, agora, algumas dessas plataformas em nosso jogo, a fim de completar o nosso primeiro cenário.

3.1 Plataforma Móvel com Script

A primeira nova plataforma que adicionaremos ao nosso jogo será a plataforma móvel com script. Afirmamos, na introdução, haver duas maneiras de fazer plataformas móveis - scripts e animações. Começaremos pela plataforma com scripts, a qual nos dá algumas liberdades a mais.

O primeiro passo, novamente, é criar a plataforma do mesmo modo como já fizemos com todas as outras. Selecionaremos o sprite Platform, o adicionaremos à cena como filho do GameObject Platforms, alteraremos a sua **camada** para Ground, alteraremos sua cor para RGB (200, 200, 0) ou Hex #C8C800FF, sua **Order in Layer** para 1, e adicionaremos um **Box Collider 2D**, tudo como feito antes. Também alteraremos o **nome** do componente para MovingPlatformS (S de Script). O **posicionamento** da plataforma será em (62,-2.5,0) com a **escala** de (7,1,1). Como só utilizaremos uma em nosso cenário, podemos adicionar esses valores ao próprio Prefab.

Contudo, dessa vez, a nossa plataforma também precisará de um script, o qual será responsável por movê-la de acordo com o comportamento que definirmos para ela. No caso de nosso jogo, de acordo com o esboço feito, criaremos uma plataforma que se moverá horizontalmente até o meio de um buraco, onde encontrará uma outra plataforma, a qual levará o personagem até o outro lado. Adicionemos à plataforma um novo script C# chamado MovingPlatformController. Mova o novo script criado para a pasta Scripts e, então, crie o Prefab da plataforma na pasta Prefabs, uma vez que todos os componentes já estão configurados.

Agora, discutiremos o script em si. Primeiramente, precisamos pensar como a plataforma se comportará. Ela deverá sair do ponto inicial, o qual diz respeito a onde ela está posicionada, e, então, se mover em direção a um novo ponto final, retornando ao ponto inicial em seguida. Precisaremos, assim, definir um ponto inicial, um ponto final e uma maneira de mover a plataforma entre esses pontos. Os pontos são definidos como Floats e se baseiam na posição inicial do objeto e numa distância a se deslocar. A parte mais interessante é como é feita a movimentação em si. Vejamos a **Listagem 4**, a seguir, antes de discutirmos isso.

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

public class MovingPlatformController : MonoBehaviour {

    private Vector3 startPosition;
    private Vector3 endPosition;

    public float movingDist = 20f;

    public float secs = 5f;

    // Use this for initialization
    void Awake () {
        startPosition = transform.position;
        endPosition = transform.position + (movingDist*Vector3.right);
    }

    // Update is called once per frame
    void FixedUpdate () {
        transform.position = Vector3.Lerp (startPosition, endPosition,
        Mathf.SmoothStep(0f, 1f, Mathf.PingPong(Time.timeSinceLevelLoad/secs, 1f)));
    }

}
```

Listagem 4 - Código para movimentação da plataforma horizontalmente.
Fonte: Elaborada pelo autor.

EITA! Quanto código interessante e quanta informação em uma só linha! Vamos discutir os detalhes.

Primeiramente, temos dois vetores que indicam a posição inicial e a posição final entre as quais a plataforma deve se locomover. Logo no Awake, assim que a plataforma é criada, definimos o ponto inicial como o ponto no qual ela se encontra, e o ponto final como um ponto deslocado movingDist unidades para a direita. Até aí, tudo bem. Já conhecemos a constante Vector3.right e sabemos lidar com posições. O FixedUpdate é que vem cheio de novidades!

No FixedUpdate, a cada atualização, modificamos a posição de nossa plataforma de acordo com vários parâmetros. O método Vector3.Lerp faz uma interpolação linear entre dois pontos. Quando fazemos uma interpolação linear, quebramos uma distância qualquer (no caso, startPosition e endPosition) em passos menores, os

quais podem ocorrer a cada atualização do FixedUpdate. Assim, temos a impressão de a movimentação ser contínua, e não só um teletransporte do ponto inicial ao ponto final.

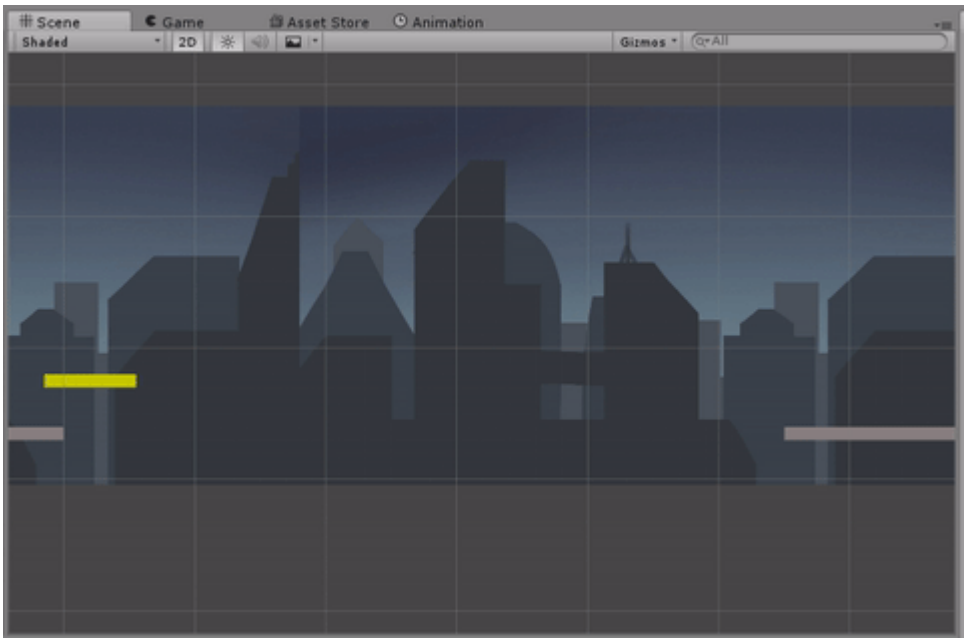
Mas qual é o passo que utilizaremos para quebrar essa distância? Bem, nesse caso, utilizaremos um passo a partir da biblioteca Mathf, chamado PingPong. Esse método varia o primeiro valor passado a ele, de modo a nunca ser maior que o segundo valor passado e nunca ser menor que zero. Ou seja, partiremos de um valor X e chegaremos até o valor 1f, retornando ao valor 0f, em seguida.

E quem é esse valor X? É o resultado da divisão de Time.timeSinceLevelLoad por **secs**, uma variável indicadora do tempo, em segundos, que a animação durará. Já a constante timeSinceLevelLoad é uma constante que mede, em segundos, quanto tempo faz que a nossa cena foi carregada.

Ou seja, imagine que nos moveremos, andando de um ponto A a um ponto B. Precisaremos levar **secs** segundos para fazer esse caminho. Qual o tamanho do passo a ser dado? É exatamente isso que está sendo feito no FixedUpdate. Estamos indicando que a plataforma deve se mover do ponto startPosition até o ponto endPosition com passos do tamanho PingPong(Time.timeSinceLevelLoad/secs, 1f). E não é só isso! Para garantir serem considerados, quando próximo às bordas, a aceleração inicial e o freio final, utilizaremos o método SmoothStep. Com isso, quando estivermos próximos ao início e ao final, os passos serão menores, para dar a impressão de aceleração e freio. Muito interessante, não?

É um pouco complicado também, concordo. Mas leia e releia essa parte algumas vezes, que as coisas farão sentido. E pode usar o fórum, caso ainda fique alguma dúvida! Vejamos o resultado dessa movimentação na **Figura 11**.

Figura 11 - Plataforma móvel se movendo de acordo com o script.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Percebe como há uma impressão de haver uma aceleração inicial e um freio, ao final? Isso se deve ao SmoothStep! Já esse comportamento de ir e voltar, parecendo que há uma movimentação contínua, se deve à variação do método PingPong, utilizado junto ao método Lerp. Super interessante, não? E funcional!

E tem mais um detalhe interessante que talvez possamos adicionar à nossa plataforma móvel. É comum, em muitos jogos, vermos o personagem, ao subir em uma dessas plataformas móveis, ser carregado por ela. Como será que podemos fazer isso com a nossa plataforma?

É bem simples! Basta o personagem tornar-se filho da plataforma quando eles tiverem em contato! E aí a alteração de posição da plataforma também alterará a posição do filho - o personagem. E fazer isso não é nada complicado! Basta utilizarmos os métodos de detecção de colisão. Só tem uma coisa que precisamos fazer antes, no PlayerController. Para a plataforma poder saber quando o personagem está de fato em cima dela, precisamos de algum método que possa informá-lo isso. Adicionaremos ao nosso PlayerController script o código contido na **Listagem 5**.

```
public bool isGrounded () {
    return grounded;
}
```

Listagem 5 - Método para consultar a variável grounded.
Fonte: Elaborada pelo autor.

Simples, não? Mas tem um detalhe importante! O método deve ser **public**, para que a plataforma tenha acesso. Feito isso, vamos ao código final da plataforma em si, visto na **Listagem 6**.

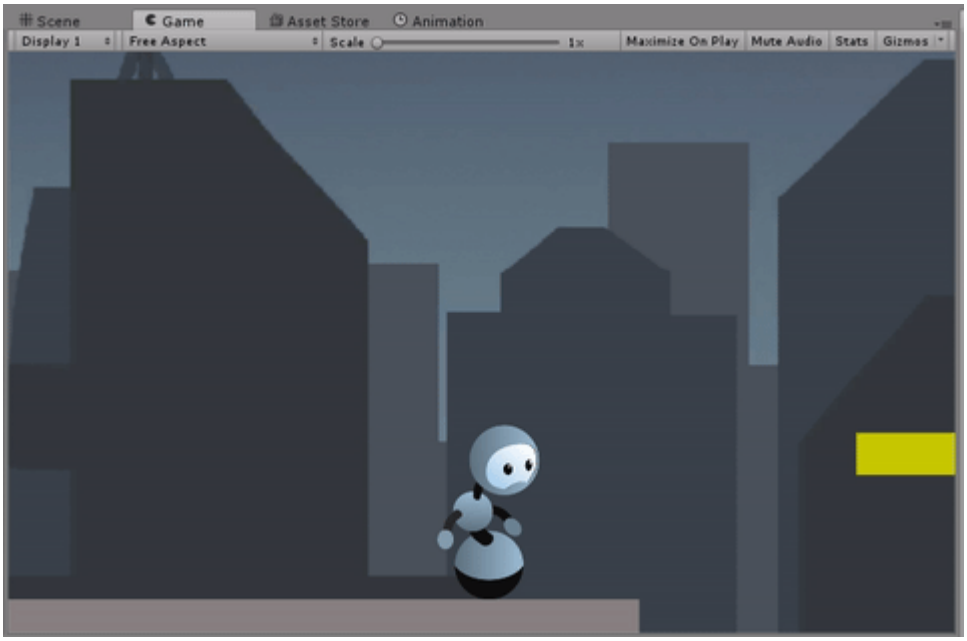
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class MovingPlatformController : MonoBehaviour {

   private Vector3 startPosition;
   private Vector3 endPosition;

   public float movingDist = 20f;

   public float secs = 5f;

   // Use this for initialization
   void Awake () {
      startPosition = transform.position;
      endPosition = transform.position + (movingDist*Vector3.right);
   }

   // Update is called once per frame
   void FixedUpdate () {
      transform.position = Vector3.Lerp (startPosition, endPosition,
      Mathf.SmoothStep(0f, 1f, Mathf.PingPong(Time.timeSinceLevelLoad/secs, 1f)));
   }

   void OnCollisionEnter2D (Collision2D col) {
      if (col.gameObject.CompareTag("Player")) {
         if (col.gameObject.GetComponent<PlayerController>().isGrounded())
            col.gameObject.transform.parent = transform;
      }
   }

   void OnCollisionExit2D (Collision2D col) {
      if (col.gameObject.CompareTag("Player")) {
         col.gameObject.transform.parent = null;
      }
   }
}
```

Listagem 6 - Código da plataforma móvel.
Fonte: Elaborada pelo autor.

Utilizamos dois dos métodos de colisão para detectar a entrada e a saída do player e, então, agir em conformidade. No método OnCollisionEnter2D, verificamos se a colisão ocorrida foi com um objeto "Player", de acordo com a Tag que definimos no início da aula. Se foi, verificamos se o Player está na plataforma, através do novo método public definido, o isGrounded. Caso esteja, dizemos que o parent do nosso player agora é a plataforma.

Quando o player sai do contato com a plataforma, detectamos isso no método OnCollisionExit2D e, então, tiramos a relação de hierarquia que havíamos criado anteriormente, deixando o player se mover livremente novamente. Vejamos o resultado desse script na **Figura 12**.

Figura 12 - Player movendo-se juntamente à plataforma devido à hierarquia criada.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Pronto! Com isso, temos uma plataforma móvel, na qual o player se move junto e as coisas funcionam de acordo com o código desenvolvido no script. Mas como podemos fazer isso utilizando animações?

3.2 Plataforma Móvel com Animação

Uma outra maneira de fazer uma plataforma se mover pode ser através da utilização de animação. O problema desse método, no entanto, é ele nos dar menos liberdade do que ao utilizar um script, pois não podemos, por exemplo, fazer o player se conectar à plataforma após o contato, devido ao fato de não haver um script no qual programamos isso. Para comportamentos simples, contudo, podemos utilizar esse método sem problemas. Vejamos como!

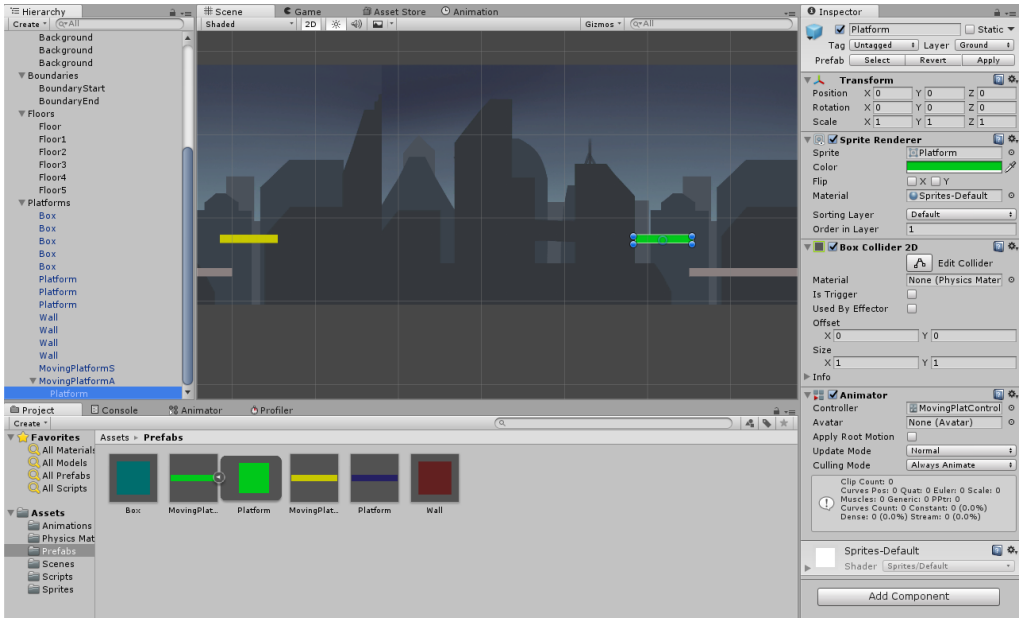
Primeiramente, precisamos criar uma nova plataforma da mesma maneira como criamos as outras. Passarei os valores! **Nome**, Platform. **Layer**, Ground. Color RGB (0, 200, 26) ou Hex #00C81AFF. **Order in Layer 1**. Adicionar **Box Collider 2D**. **Posicionamento** (0, 0, 0) e **escala** (1,1,1). Por fim, adicionaremos um Animator, como já fizemos com o nosso player, à nossa plataforma. Add Component -> Miscellaneous -> Animator.

Para esse sistema funcionar, no entanto, precisamos definir um componente vazio como pai de nossa plataforma. Isso acontece devido ao fato de a movimentação da animação ocorrer em valores absolutos, então, se movermos 100 unidades em nossa plataforma, ela se moverá do ponto 0 ao ponto 100. Utilizando um GameObject vazio como pai, ela se moverá de 0 a 100 em relação ao pai. Então, posicionamos o **pai** na posição que quisermos e ela se moverá corretamente. Deu para entender? Se não deu, está tranquilo, poste sua dúvida no fórum ou aproveite para esclarecê-la no encontro presencial! Certamente, alguém lá poderá te ajudar com isso! ;)

Sendo assim, criemos um objeto vazio, chamado MovingPlatformA (A de animação), como filho de Platforms. Nesse objeto, alteraremos a **posição** para (112, -2.5, 0) e a **escala** para (7,1,1). Em seguida, colocaremos o objeto Platform, criado anteriormente, como **filho** dele.

Para finalizar a criação do objeto, deveremos criar, na pasta Animations de nossos assets, um novo Animator Controller que será responsável pela animação de nossa plataforma. Clica com o botão direito -> Create -> Animator Controller. O nome que utilizaremos é MovingPlatController. Agora, é só clicar e arrastar o controller para nossa plataforma e ela estará concluída, apenas aguardando a animação. Criaremos o Prefab dela e tudo ficará como podemos ver na **Figura 13**.

Figura 13 - Plataforma móvel animada criada e adicionada como Prefab.



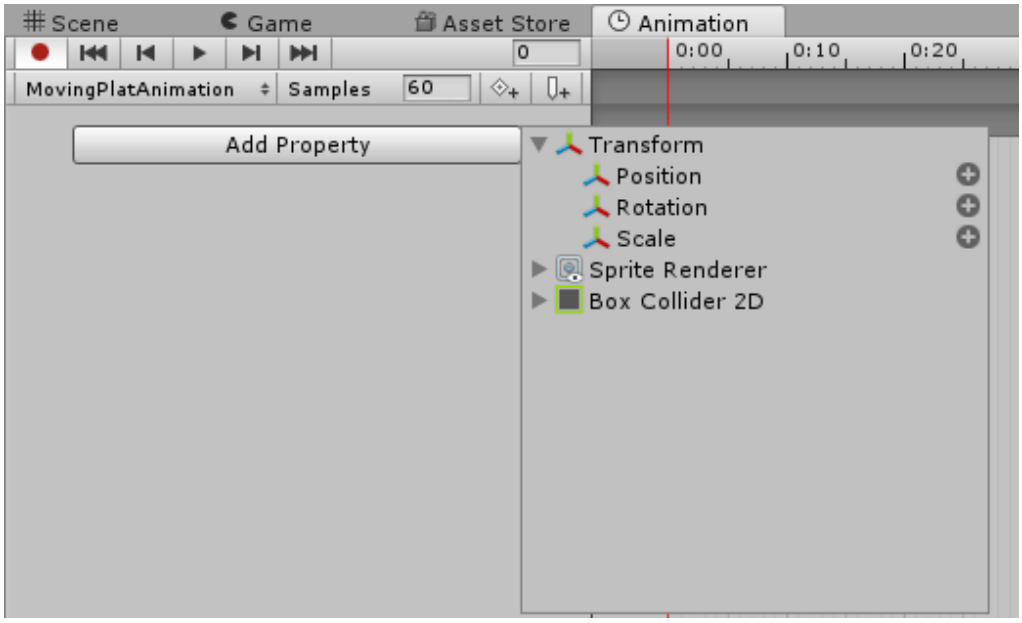
Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Veja agora se tudo está de acordo com o que vemos na **Figura 13**, para evitar surgir qualquer problema adiante!

Agora que já criamos a nossa plataforma e o nosso Animator Controller, basta adicionarmos a ela uma animação. Para isso, como vimos anteriormente, basta clicar na aba Animation, com a plataforma selecionada, e, então, clicar no botão Create. Selecione a pasta de Animations e utilize o nome MovingPlatAnimation. Isso criará um novo clipe de animação.

Note haver do lado esquerdo, na aba Animation, um botão chamado Add Property. Esse botão é responsável por adicionar propriedades à animação. Clicando nele, você verá um menu contendo os valores que podem ser animados. Selecionando a opção transform, você verá as propriedades do componente que podem ser alteradas. Clicando no + ao lado de position, a posição passará a ser um elemento da animação. O menu pode ser visto na **Figura 14**.

Figura 14 - Seleção de position como uma das opções da animação.



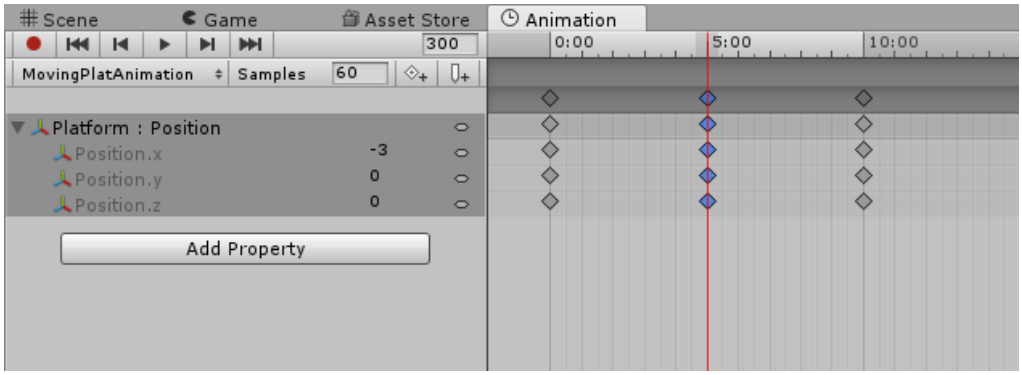
Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Após clicar no botão + e adicionar position como uma das propriedades da animação, precisamos apenas definir a animação de movimentação. Definimos, no outro script, **secs** como 5, indicando que a plataforma demoraria 5 segundos para sair do início até o final. Como queremos manter as plataformas sincronizadas, deveremos definir que a animação durará 10 segundos. Para isso, adicionamos um novo quadro-chave com 5 segundos, representando a metade do movimento, e outro com 10 segundos, finalizando.

Para facilitar, podemos simplesmente, visto que estamos utilizando 60 samples, como vemos na **Figura 14**, navegar para o quadro 300 da animação e adicionar um Keypoint e, então, navegar para o quadro 600 e adicionar outro. Podemos navegar pelos quadros utilizando o valor que está abaixo da aba Asset Store, com um 0, na **Figura 14**. Ao clicar no quadro 300, adicione um keypoint com X = -3, já no quadro 600, retorne o X para 0. Com isso, a animação levará 5 segundos para movimentar a

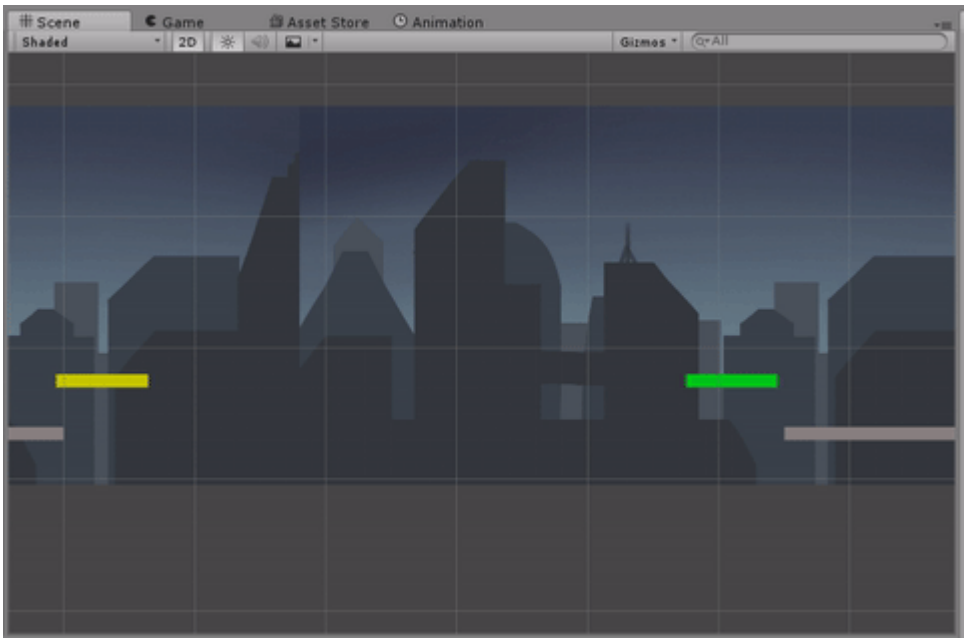
plataforma até o ponto -3 e, então mais 5 segundos para retornar ao 0. A animação completa pode ser vista na **Figura 15**. O resultado, por sua vez, pode ser visto na **Figura 16**.

Figura 15 - Animação para a movimentação da plataforma.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Figura 16 - Plataformas sincronizadas, uma com script e outra com animação.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Viu como as plataformas terminam ficando bem similares? A diferença, no entanto, é que a plataforma feita através de animação não carrega o jogador junto a ela quando há o contato, uma vez que não há script para programarmos isso. As animações são muito versáteis, não é?

3.3 Plataforma que Cai

A última plataforma da qual precisamos, para finalizar o nosso cenário, é a plataforma que cai ao fazer contato com o usuário. Ela é bem comum em jogos diversos do estilo, e a utilizaremos como desafio final para a nossa fase, uma vez que ela será o último obstáculo entre o jogador e o objetivo final.

Criaremos a plataforma da mesma maneira como criamos as outras, como filha de Platforms. Passarei os valores novamente! **Nome**, FallingPlat. **Layer**, Ground. **Color** RGB (200, 0, 0) ou Hex #C80000FF. **Order in Layer** 1. Adicionar **Box Collider 2D. Escala** (7,1,1).

Além disso, adicionaremos a ela um Rigidbody 2D do tipo **Static**! É muito importante fazer essa alteração, pois é através dessa propriedade que faremos a plataforma cair e voltar. Por fim, como você deve imaginar, adicionaremos a ela um script C#. Chamaremos o script de FallingPlatController. Vamos diretamente ao código desse script, podendo tal código ser visto na **Listagem 7**. Discutiremos na sequência!

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

public class FallingPlatController : MonoBehaviour {

    public float fallTime = 1f;
    public float respawnTime = 8f;

    private Rigidbody2D rb;
    private Vector3 pos;

    // Use this for initialization
    void Start () {
        rb = GetComponent<Rigidbody2D> ();
        pos = transform.position;
    }

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

    }

    void OnCollisionEnter2D (Collision2D col) {
        if (col.gameObject.CompareTag("Player")) {
            Invoke ("PlatFall", fallTime);
        }
    }

    void PlatFall() {
        rb.bodyType = RigidbodyType2D.Dynamic;
        Invoke ("Respawn", respawnTime);
    }

    void Respawn() {
        rb.bodyType = RigidbodyType2D.Static;
        transform.position = pos;
    }
}
```

Listagem 7 - Código do script FallingPlatController, adicionado ao objeto FallingPlat.
Fonte: Elaborado pelo autor.

O aspecto mais importante desse script é o método Invoke. Esse método serve para chamar, por meio de um atraso definido, um outro método. O Invoke é utilizado duas vezes nesse script - a primeira, para invocar a queda da plataforma, através do método PlatFall, usando um atraso de fallTime. A segunda, por sua vez, para retornar a plataforma à posição inicial após respawnTime segundos, utilizando o método Respawn.

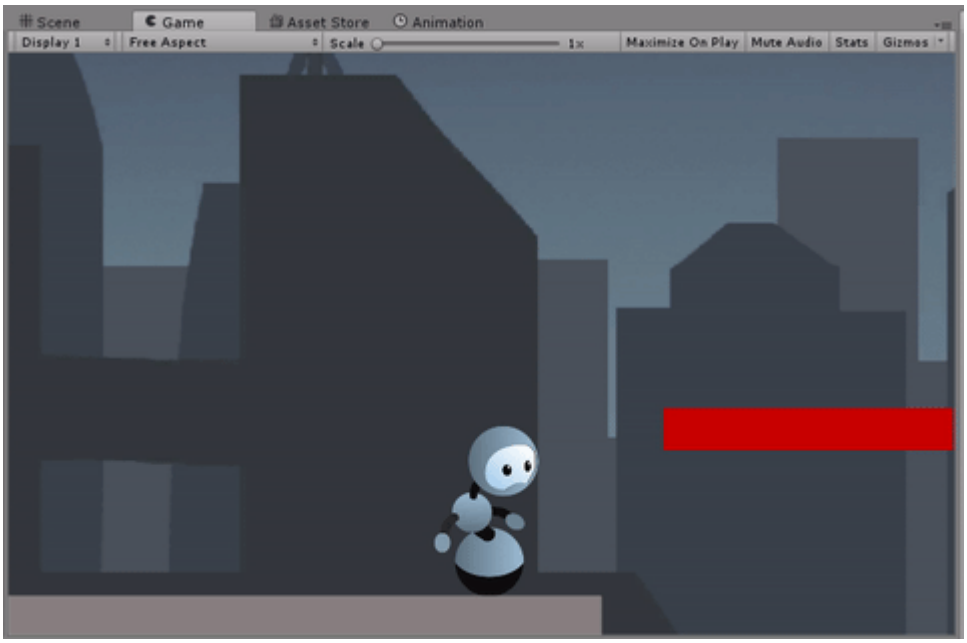
O método PlatFall altera o tipo do RigidBody2D para Dynamic, fazendo-o sofrer a ação da gravidade e cair! Ao retornar a plataforma à sua posição original, utilizando o método Respawn, retornamos o RigidBody2D ao tipo Static, de modo que ele não sofra qualquer tipo de força. Simples, não? Mas bem eficaz! Adicionado esse script e alteradas as propriedades, podemos criar o Prefab e replicá-lo para a nossa cena. As posições podem ser vistas na **Tabela 2**.

Elemento	Posição
FallingPlat	(170, -2, 0)
FallingPlat	(185, 0, 0)
FallingPlat	(200, 2, 0)
FallingPlat	(215, 4, 0)

Tabela 2 - Posicionamento das Instâncias do FallingPlat.
Fonte: Elaborada pelo autor.

Vejamos o resultado do posicionamento e da configuração das plataformas na **Figura 17**.

Figura 17 - Plataformas caindo de acordo com o contato do usuário.



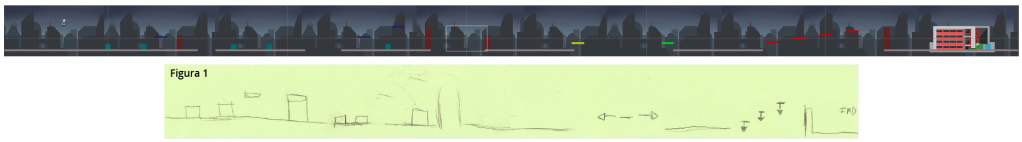
Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

E, com isso, concluímos a criação de nosso cenário. Falta apenas o objetivo final! Qual será? Clique aqui para baixar e descobrir! Vamos, agora, posicioná-lo em nossa cena.

4. Adicionando o Objetivo Final ao Cenário

Importaremos o sprite baixado no link disponibilizado na seção anterior e, então, o adicionaremos à nossa cena, como já fizemos outras vezes. O sprite deverá ser configurado com posição (276, -0.9, 0) e escala (2, 2, 1). Desse modo, finalizamos o nosso cenário! Ficou bem bacana, não? Vamos dar uma olhada na fase completa, na **Figura 18**, e, em seguida, jogar uma vez para ver como ficou!

Figura 18 - Level completo, de acordo com o que foi prototipado na Figura 1.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 04 de mar de 2017

Jogo em ação

E, assim, finalizamos a nossa aula! Vimos muita coisa legal, mas, com todo esse esforço, chegamos ao final de um nível bem completo e divertido! Existem diversas outras coisas que podemos adicionar, ou outros tipos de plataformas que podemos também colocar (tem um effector do tipo Platform, lembra?), entretanto, não temos tempo para mais nada, por agora.

Espero que todos tenham gostado da aula, pois foi muito bom desenvolvê-la! Principalmente por poder jogar o joguinho completo, com uma fase, já! Mas ainda faltam diversas coisas, não é? Não temos uma interface, ainda. Cair no buraco faz o

personagem se perder, mas o jogo não se encerra. Não temos som em nosso jogo! Há muito ainda o que ver! Nos encontraremos na próxima aula para continuar a nossa aventura! Até \o/

Leitura Complementar

Manual Oficial do Unity - A função Invoke https://unity3d.com/pt/learn/tutorials/topics/scripting/invoke

Resumo

Na aula de hoje, finalizamos a criação de nosso primeiro nível completo no Unity. Utilizamos Prefabs, mais uma vez, para criar elementos estáticos e replicá-los. Em seguida, criamos elementos dinâmicos, capazes de se mover via script e, também, via animações.

Utilizamos funções de colisão, como OnCollisionEnter2D e OnCollisionExit2D, para definir, em tempo de execução, a hierarquia de objetos e criar uma plataforma que se move e é capaz de mover o player junto a ela.

Aprendemos que animações podem alterar propriedades de todos os componentes presentes no objeto animado. Também vimos como utilizar alguns keyframes bem posicionados, para criar um efeito interessante de animação.

Conhecemos a função Invoke, a qual faz uma chamada atrasada de uma função de acordo com um tempo, e a utilizamos para criar plataformas que caem após o contato com o usuário e, logo em seguida, voltam à posição inicial.

Além disso, no início da aula, fizemos diversas modificações em nosso player, a fim de torná-lo mais adaptado ao estilo de nível que estávamos desenvolvendo. Trocamos o collider que estávamos utilizando, mudamos o método de pulo duplo, alteramos a movimentação do personagem, entre outras coisas.

Foi uma aula bem extensa e cheia de conteúdos interessantíssimos! O projeto final pode ser baixado neste link, no qual estão disponíveis todas as modificações que fizemos ao longo desta aula, concluindo o nosso primeiro nível e chegando ao IMD! Excelente!

Autoavaliação

1. Em sua opinião, qual o melhor colisor para se utilizar em nosso personagem?
2. Qual a principal diferença entre utilizar AddForce e alterar diretamente a Velocity de um personagem?
3. O que faz o método Invoke?
4. Como funcionam os métodos OnCollisionEnter2D e OnCollisionExit2D?
5. É possível alterar propriedades por meio da utilização de animações? Como?

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.