Desenvolvimento com Motores de Jogos I

Aula 06 - Mecânicas de Pulo

Apresentação

Olá, desenvolvedores de jogos! Estamos de volta à nossa aula de Desenvolvimento com Motores de Jogos I, agora com a sexta fase do nosso jogo. Como sabemos, são 15 níveis no total, então, já cumprimos o primeiro terço do desafio! Aprendemos bastante coisa, não?

E ainda há muito a aprender. A cada nova fase que encontramos, novos desafios são propostos e resolvidos, de maneira gradual, até chegarmos juntos ao desafio final, vencendo-o por meio do desenvolvimento de um jogo completinho! Bacana, certo?

Na aula de hoje, estudaremos mais um detalhe bem importante para percorrer um nível em um side-scroller 2D: o pulo! Entenderemos um pouco sobre essa mecânica importante em jogos desse tipo e aprenderemos a fazer o nosso amigo robô pular! Agora que ele já tem a física bem definida e a câmera o segue bem, está na hora de levá-lo a outros espaços, concorda? Então vamos nós!

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Compreender as diferentes mecânicas de pulo;
- Compreender a técnica de Line Casting;
- Implementar duas mecânicas de pulo para o nosso amigo robô;
- Utilizar Camadas no Unity.

Introdução

Esta aula, diferentemente do que vínhamos fazendo anteriormente, terá um teor mais prático, por se tratar de uma aula de mecânicas e não ser relacionada, em parte, aos aspectos do motor em si, ou mesmo da teoria de desenvolvimento de jogos.

Logo de início, deixamos um link para o projeto que estamos desenvolvendo e recomendamos a vocês abrirem os seus projetos a fim de acompanhar, pois desenvolveremos muitas coisas ao longo desta aula diretamente em nossos joguinhos.

Você verá que explicaremos algumas variações das técnicas apresentadas, umas diretamente no jogo, outras em projetos novos para não misturarmos tudo, mas, ao fim, manteremos em nosso jogo apenas uma delas, com o propósito de continuarmos o desenvolvimento nas próximas aulas. Então, caso prefira acompanhar exatamente os projetos que estamos desenvolvendo, fique atento a quais das técnicas manteremos em nossos níveis!

Dito isso, vamos começar de fato os nossos estudos nesta aula conhecendo um pouco mais sobre como são implementadas as diferentes técnicas de pulo em jogos de plataforma 2D, como é o nosso! Go, go, go!

Alterando o Nosso Jogo para Conter o Pulo

Em uma de nossas primeiras aulas, estudamos os eixos e o modo como podemos controlar o personagem utilizando os valores obtidos por estes. Isso gerou o script de movimentação e controle do personagem que estamos utilizando até agora como nosso CharacterController. Esse script, no entanto, contém ainda algumas imperfeições e, por ter sido um dos nossos primeiros contatos com a linguagem, foi desenvolvido para ser simples e não eficiente.

Passadas algumas aulas, já temos alguma experiência e algum tempo de contato com a ferramenta, então, podemos nos aventurar em um melhor script de movimento, fazendo o personagem parecer mais um elemento de um jogo de plataforma como usualmente vemos. Para isso, vamos fazer algumas alterações em nosso script e aderir a algumas novas guidelines de programação.

Importante!

Quando trocamos uma técnica utilizada por outra, para abordar novos aspectos, não significa que a técnica anterior é inútil/errada. Nosso personagem, como está no momento, com controles horizontais e verticais, poderia ser um excelente exemplo para um jogo subaquático, por exemplo, no qual é possível se mover igualmente em todos os eixos. Bastaria tirar a gravidade!

Resumindo: Nunca esqueça qualquer uma das técnicas estudadas nas aulas anteriores, mesmo que elas sejam substituídas!

Removendo a Movimentação Livre no Eixo Y

Dito isso, começaremos as nossas alterações de movimentação do personagem removendo sua capacidade de voar. Agora, com o nosso novo script contendo pulos simples, o personagem estará sempre no chão, sujeito à gravidade, até que o pulo seja acionado. Também removeremos sua capacidade de forçar a descida, tirando, assim, completamente o controle sobre o eixo vertical, exceto pelo próprio pulo!

Outra mudança que faremos a partir de agora é começar a se acostumar com algo comum em qualquer ambiente de programação e, também, no mundo do desenvolvimento de jogos: a língua inglesa! A partir das listagens desta aula, utilizaremos, sempre que possível, nomes de funções e variáveis em Inglês, para vocês já se acostumarem e poderem se adaptar bem às situações reais que encontrarão. Viva Inglês Técnico I e II!

São tantas as alterações que é melhor criarmos um novo script, deixando o velho de lado. Para isso, delete o script antigo do seus Assets ou simplesmente o renomeie para algo a não ser mais utilizado. Cuidado para não se confundir após renomear, hein! Criaremos um novo script C# chamado PlayerController, atrelado ao player, onde iniciaremos a criação do nosso novo código de movimentação.

O primeiro aspecto a fazer é adicionar novamente a movimentação horizontal, a qual já estava funcionando de maneira adequada. Para isso, precisaremos, ainda, da referência ao RigidBody2D, da velocidade com a qual o jogador acelerará e, então, poderemos capturar os inputs e traduzi-los em comandos para o motor de física. Vejamos a **Listagem 1** para entender melhor esses primeiros passos.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PlayerController : MonoBehaviour {
6
7      private Rigidbody2D rigidBody;
8
9      public float acceleration = 100f;
10
11     // Use this for initialization
12     void Awake () {
13         rigidBody = GetComponent<Rigidbody2D> ();
14     }
15
16     //Called in fixed time intervals, frame rate independent
17     void FixedUpdate()
18     {
19         float moveH = Input.GetAxis ("Horizontal");
20
21         rigidBody.AddForce (Vector2.right * moveH * acceleration);
22     }
23 }
```

Listagem 1 - Código para o movimento inicial do personagem.

Parou! Parou! Função nova no pedaço! O que é esse tal de Awake, professor?

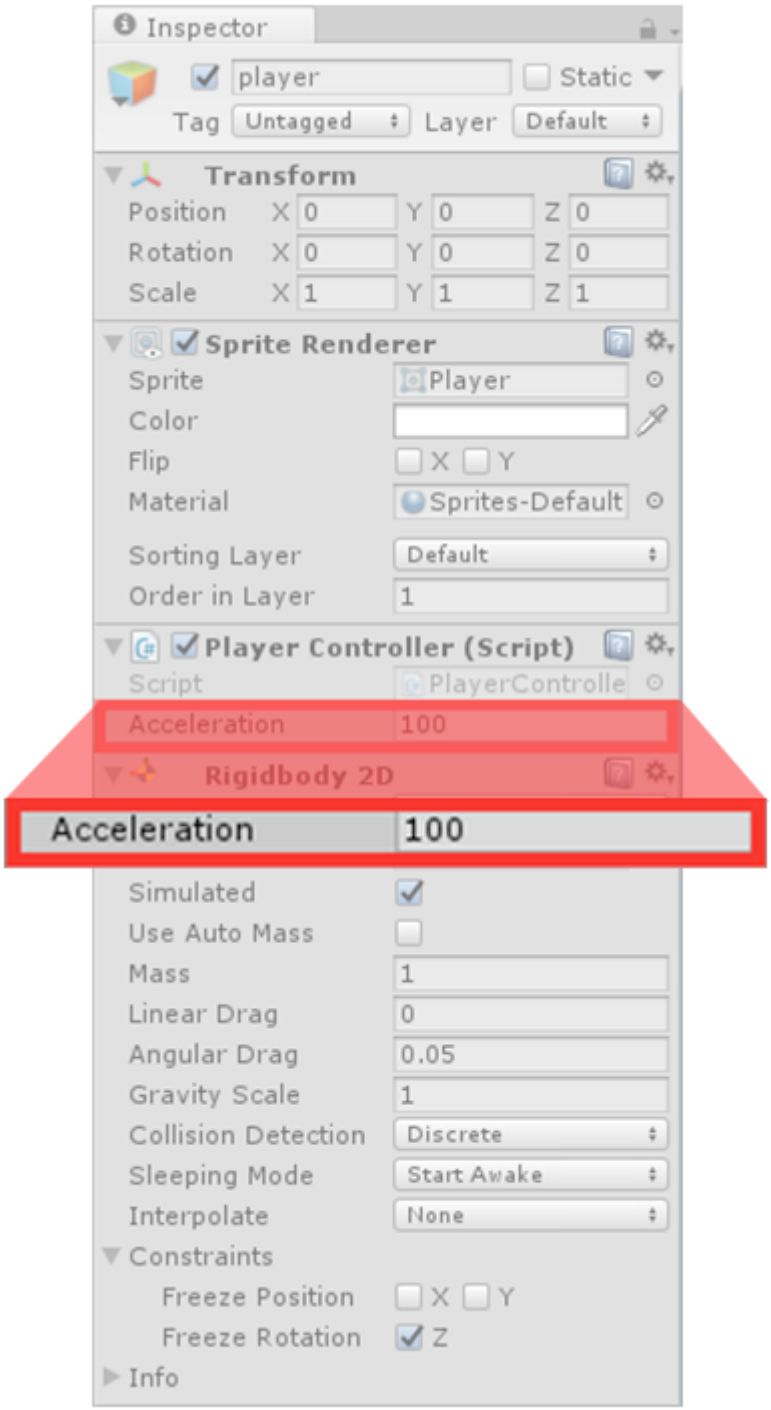
A função Awake, assim como a função Start, é utilizada para a inicialização de valores nos objetos. Ela também é um callback, mas é chamada em um momento diferente da inicialização! A função Awake é invocada antes mesmo de o jogo começar! E, com isso, você pode definir tudo que é necessário "já estar pronto" quando o jogo for iniciado. Ou seja, podemos utilizar a Awake objetivando, de fato, inicializar os nossos objetos, e a Start, vista anteriormente, em um momento seguinte, quando tudo já estiver quase pronto. Isso pode ser útil, por exemplo, para inicializar valores no Awake e, quando necessário utilizar no Start, para eles estarem prontos. Simples! Não criemos pânico.

As outras chamadas não parecerão estranhas a nós, pois são muito similares ao que já estávamos utilizando. Modificamos, apenas, a maneira como adicionamos a força ao nosso RigidBody2D. Lembra de anteriormente nós criarmos um Vector2 com os valores desejados e, então, passarmos esse vetor para o AddForce? Pois é! Agora estamos utilizando uma abordagem diferente. A constante Vector2.right cria um vetor unitário apontando para a direita no sistema de eixos! E aí, se multiplicarmos esse valor pelo valor que o GetAxis nos dá (de 1 a -1, lembra? Aula de input!), chegamos à direção correta na qual gostaríamos de executar o movimento. Finalizando o valor correto a ser passado, adicionamos à conta a aceleração do personagem e, desse modo, o personagem está se movendo no eixo X corretamente!

Clica lá em Play e testa para ver! (Sério. É importante acompanhar!)

Se o seu personagem não se moveu, é porque faltou definir o valor da aceleração, por alguma razão. Definimos essa variável como public justamente para ela poder ser alterada diretamente lá no editor. Então vai lá fazer isso, caso não tenha feito ainda. Gostei de utilizar o valor inicial 100, com a massa do RigidBody sendo 1. Fica excelente! Vejamos na **Figura 1** o campo, só para lembrar.

Figura 01 - Alterando o valor do campo de aceleração do PlayerController.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/

Ok! Então o nosso personagem já voltou a se mover. Excelente! E nem foi difícil. Mas agora começaremos as modificações que ainda não havíamos visto.

Adicionando uma Limitação de Velocidade

A primeira coisa que faremos será relacionada à velocidade máxima do jogador. Deixar o player sem uma velocidade máxima pode ser perigoso, concordam? Logo, logo ele estará mais rápido do que podemos controlar e aí tudo perderá um pouco do sentido. Veremos, então, na **Listagem 2**, como limitar a velocidade dele de uma maneira a não destruir muito o motor de física 2D, como discutimos em aulas anteriores!

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

public class PlayerController : MonoBehaviour {


    private Rigidbody2D rigidBody;

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

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

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

        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;
        }
    }
}
```

Listagem 2 - Código para adicionar uma limitação à velocidade máxima.

Ok. Temos algumas coisas novas e legais para conversar agora. Primeiramente, criamos uma nova variável pública do tipo float chamada maxSpeed. Essa variável serve para nos indicar qual a velocidade máxima que o personagem poderá atingir

(duh!).

Em seguida, já na função FixedUpdate, movemos o nosso AddForce para dentro de um IF. Esse IF é bem claro e faz justamente o que nós queremos neste momento da aula: limitar a velocidade máxima. O teste a ser feito diz respeito a *se a velocidade na direção desejada for menor que a velocidade máxima, adicione a força.* Com isso, já melhoramos o nosso limite de velocidade, mas ainda há um ponto importante que pode acontecer devido a fatores externos, ou qualquer outra coisa: e se a velocidade passar da máxima? Devemos reduzi-la para a máxima.

Justamente para lidar com esse problema, adicionamos o segundo IF visto em nossa listagem. Esse IF traz duas novas funções que não havíamos lidado ainda: Mathf.Abs e Mathf.Sign. Assim como em Java, existem bibliotecas em C# para lidar com funções matemáticas simples. Essas são duas dessas funções.

A primeira delas, Mathf.Abs, nos retorna o **Abs**solute Value de um valor, ou seja, o módulo. Essa função equivale à função matemática módulo.

Já a segunda, a função Mathf.Sign, nos retorna um valor de acordo com o sinal do valor que passamos a ela. Se o valor passado for negativo, retorna-se -1. Caso o valor seja positivo ou zero, retorna-se 1.

E para quê nós utilizamos essas funções nesse IF? A primeira delas, Mathf.Abs, serve para nos retornar o valor absoluto da velocidade em X do personagem. Caso o personagem esteja se movendo na direção negativa do eixo, esse valor será negativo, correto? Para compararmos adequadamente esse valor com a velocidade máxima, utilizamos o módulo! Se a velocidade (independentemente da direção) for maior que a velocidade máxima, temos um problema a tratar!

E aí, dentro do IF, tratamos esse problema. Como? Utilizamos Mathf.Sign para saber a direção na qual o personagem estava se movendo! Então, simplesmente alteramos a velocidade atual dele para ser a máxima, na direção que ele estava indo! E pronto. Com essas alterações, conseguimos limitar a velocidade do player para nunca passar da velocidade máxima!

Em síntese, adicionamos dois IFs ao nosso código. O primeiro só adiciona velocidade se o personagem estiver abaixo da máxima. Já o segundo cuida de, se o personagem estiver acima da velocidade máxima, reduzi-la até esta. Feito! Agora o

personagem já se move adequadamente no eixo X. Para movê-lo no eixo Y, partiremos agora para o universo dos pulos!

Técnicas de Pulo

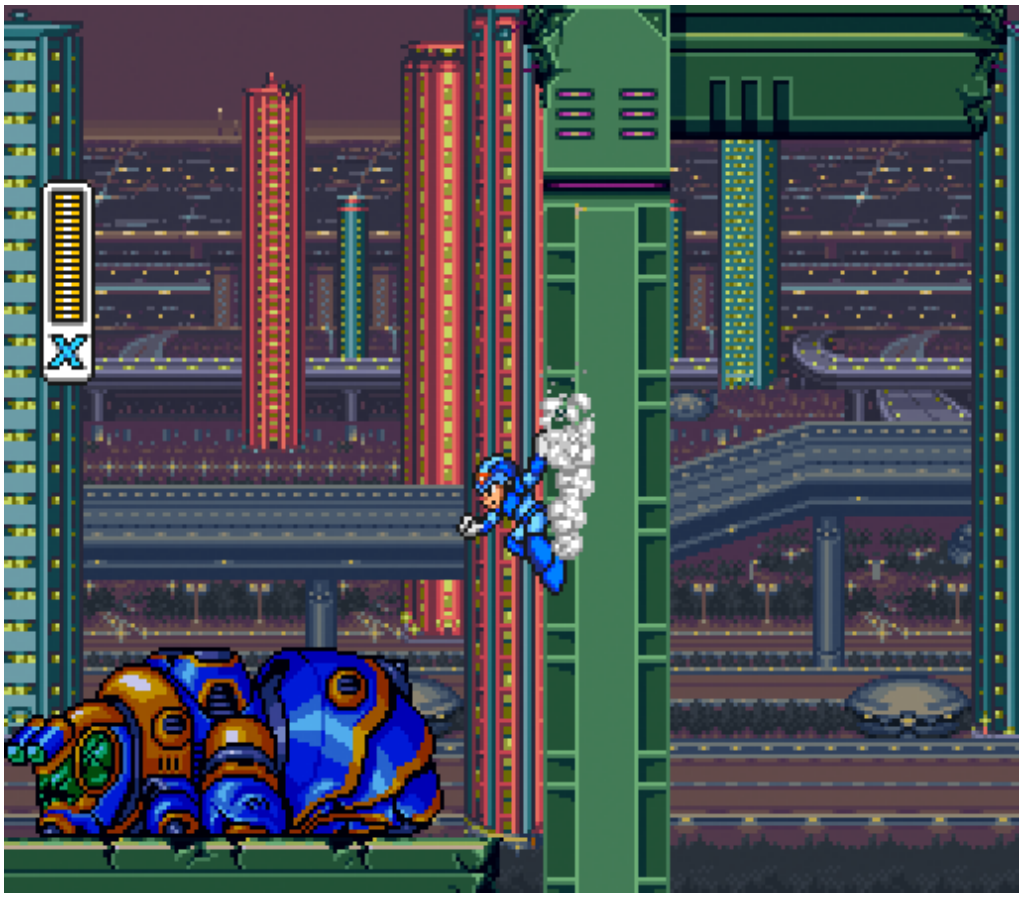
Uma das coisas que define o gênero de jogos de plataforma, desde seu princípio, por ser uma das bases de seu funcionamento, é a capacidade do jogador de pular, para alcançar as diferentes plataformas do cenário e superar os diversos buracos encontrados ao longo dos níveis.

Figura 02 - Donkey Kong, um dos primeiros jogos de plataforma criados, introduzindo o Jumpman (o qual depois ficou conhecido como Mario).

Fonte: Wikipédia. Disponível em : https://en.wikipedia.org/wiki/File:Donkey_Kong_Screen_3.png; Acesso em: 15 fev. 2017

Desde então, diversos outros jogos do gênero surgiram e com eles novas técnicas de pulo foram aparecendo. Não era possível apenas pular, mas também pular uma segunda ou terceira vez no ar, planar na descida, utilizar as paredes como apoio para pulos ou mesmo para deslizar lentamente. Além disso, é comum em diversos jogos de plataforma ter uma maneira de pular em diferentes alturas, de acordo com os comandos dados pelos usuários, seja a duração em que a tecla é mantida pressionada ou mesmo o quanto a apertou antes de sair do chão.

Figura 03 - Megaman utilizando a parede como uma plataforma para deslizar e pular.



Fonte: Jogo Megaman, da Nintendo. Imagem disponível em: .

Todas essas técnicas trouxeram novos aspectos aos jogos de plataforma em geral e também criaram novos desafios para nós, desenvolvedores, tratarmos em nossos jogos. Entraremos em mais detalhes, em nossa aula, sobre três técnicas principais, implementando duas delas em nosso jogo e deixando a terceira para vocês como o desafio da semana! Quem conseguirá?

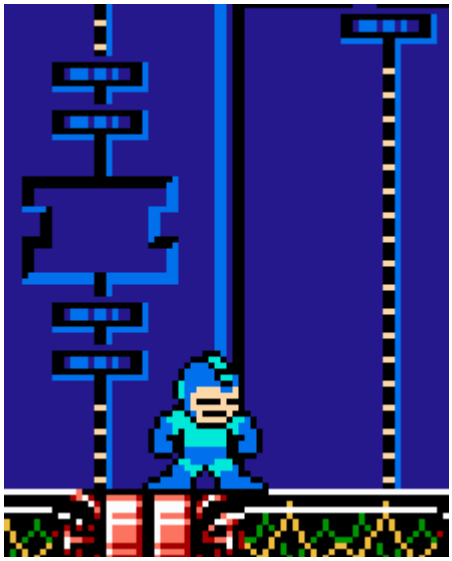
O Pulo Simples

O primeiro tipo de pulo a ser discutido em nossa aula é o pulo simples. Nesse tipo de pulo, o personagem sai do chão ao ter o botão de pulo pressionado e sobe, com uma velocidade determinada, até a gravidade atuar e o fazer parar, começando a descer em seguida. Mesmo esse tipo de pulo, no entanto, pode ter diversas variações de acordo com o que o desenvolvedor do jogo quiser.

> Dê uma olhada neste artigo online e veja como pulos simples podem ser complexos, mesmo estando todos dentro da mesma mecânica:
>
> <https://critpoints.wordpress.com/2015/05/18/5-games-5-jumps/>

Incrível, não? Todos eles utilizam o tempo da tecla pressionada, ou algum aspecto extra, para poder definir a altura que o pulo terá, ou mesmo o quão rápido o personagem retornará ao solo. Esse tipo de modificação é bem simples e já pode trazer diferenças importantes ao seu jogo. Quem já jogou Rockman/Megaman do NES sabe a importância de um pulo na altura certa para matar um inimigo e a dificuldade em dominar isso adequadamente no controle!

Figura 04 - Diferentes alturas de pulo possíveis de se obter no Megaman do NES.



Fonte: Artigo Sobre Pulos, disponível em: https://critpoints.wordpress.com/2015/05/18/5-games-5-jumps/.

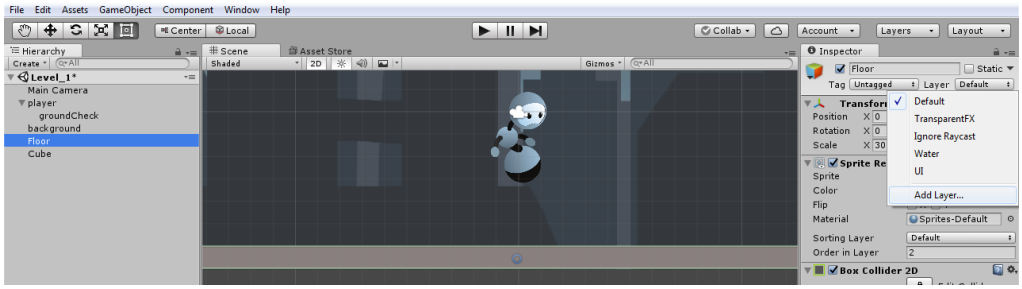
Para o nosso controle, utilizaremos algo bem simples. Ao apertar o botão, o nosso personagem será lançado no ar por uma força de pulo, a qual chamaremos de jumpForce, e obedecerá unicamente à gravidade até retornar ao chão. A fim de o personagem poder pular, no entanto, teremos uma condição: ele deverá estar no chão antes de pular. E como poderemos testar se ele está no chão ou não? O que é o chão?

Primeiramente, precisaremos lidar com esse segundo problema. Precisamos encontrar uma maneira de dizer ao jogo o que é o chão e como lidar com ele. Para isso, voltando ao que vimos em nossas primeiras aulas, utilizaremos uma **Layer**.

Relembrando rapidamente, uma camada, ou Layer, é um grupo criado para adicionar objetos que apresentam comportamentos semelhantes e, com isso, tratá-los de maneira também semelhante. Obviamente não temos apenas UM objeto chão no jogo todo, concorda? Bom, no momento temos, mas nossa fase há de crescer em breve, né? Então, para todos esses objetos poderem ser vistos da mesma maneira, precisamos adicioná-los a uma mesma camada, a qual, nesse caso, chamaremos de *Ground*.

Pretendendo definir essa camada, basta selecionar o objeto que queremos, adicioná-lo à camada e então, no Inspector, no menu de Layers, escolhemos a opção de criar uma nova camada. Esse caminho pode ser visto na **Figura 5**.

Figura 05 - Adicionando uma nova camada à lista do projeto.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/.

Ao clicar nessa opção, uma lista de camadas aparecerá no Inspector. Perceba que algumas camadas se chamam "Builtin Layer" e outras "User Layer". As Builtin Layers são camadas utilizadas pelo próprio sistema e não é permitido alterar os seus nomes. Já as camadas User Layer são aquelas que o desenvolvedor pode redefinir para utilizar em seus projetos. Alteremos, então, a primeira User Layer disponível

para **Ground**. Com isso, ao selecionar novamente o objeto e clicar na lista, como visto na **Figura 5**, a opção Ground vai agora estar disponível para seleção. Definam nosso chão e vamos adiante!

Importante!

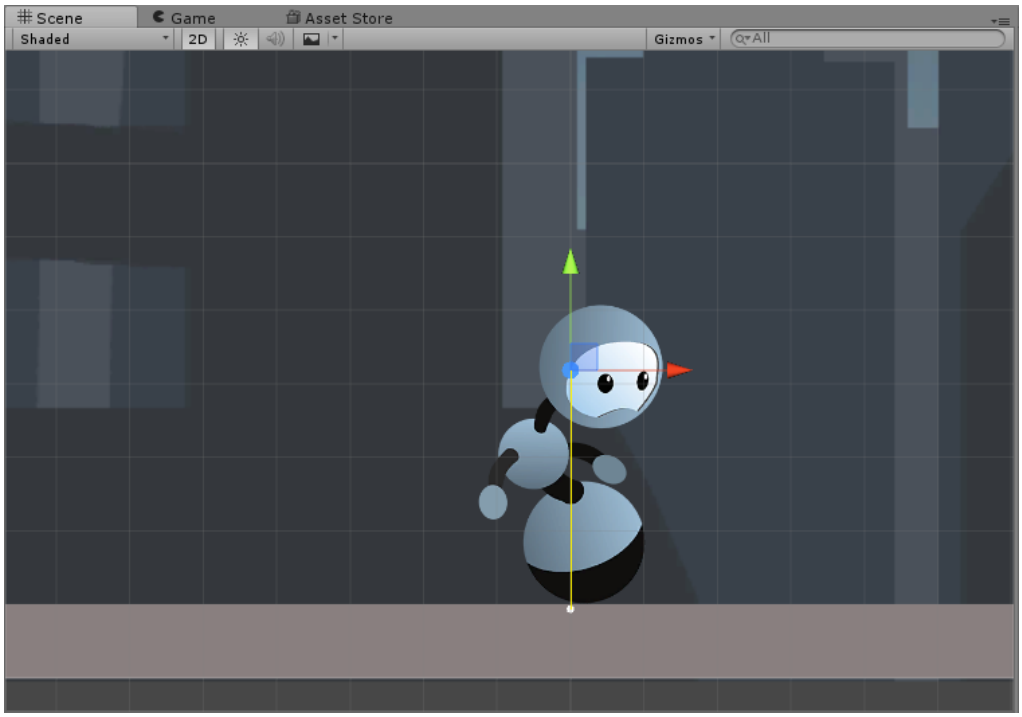
Ao exportar pacotes de Assets (.unitypackage), como estamos fazendo em todas as aulas para compartilhar os nossos projetos, o Unity não carrega, por padrão, as Tags e Layers definidas no projeto! Com isso, a partir de agora, ao importar um Unity Package qualquer que tenhamos disponibilizado, é preciso REDEFINIR AS TAGS E LAYERS para que tudo continue funcionando normalmente. TENHA CUIDADO!

Retornamos à programação normal. Agora, entendendo bem o que é o chão, podemos passar ao nosso próximo passo: dar ao nosso personagem capacidade de identificar o chão. Definimos o comportamento do pulo - pulo simples, pula uma vez apenas quando estiver no chão, e também vimos a definição do chão para que isso possa funcionar. Como faremos agora para o personagem detectar que de fato está no chão?

Bom... Existem várias técnicas que nos permitem fazer isso! Utilizaremos aqui uma muito comum e bem fácil de implementar, conhecida como *Ray Casting*. Na verdade, como estamos em 2D e também pela maneira que utilizaremos a técnica, a chamaremos de *Line Casting*.

A técnica de ***Line Casting*** consiste em definir um ponto de origem e um ponto de destino entre os quais traçaremos uma linha, a fim de detectar qualquer objeto que essa linha cruze. Caso ela passe por um objeto qualquer, significa que esse objeto está definido entre os pontos. E como utilizaremos isso para que o personagem detecte o chão? Simples! Criaremos um ponto o qual ficará um pouco abaixo do personagem e traçaremos essa linha entre o centro do personagem e esse ponto um pouco abaixo. Caso encontremos chão no caminho, o personagem está sobre o chão! Vejamos na Figura 6 como exatamente acontecerá.

Figura 06 - Linha amarela representando o Line Casting entre o centro do personagem e o círculo branco que está logo abaixo dele.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/

Obviamente não precisamos que o ponto branco seja visível, mas, para efeitos didáticos, o adicionamos na imagem. Esse ponto branco vai representar o elemento chamado de "groundCheck". Esse elemento serve como ponto final para a linha e também define o limite no qual o personagem será considerado em contato com o chão. Uma vez definido esse ponto, podemos simplesmente traçar essa linha amarela (que na verdade não existe, está apenas ilustrada para que vocês entendam melhor) para saber se, em algum ponto dela, há contato com o chão. Caso sim, o personagem está no chão e apto a pular. Conseguiu entender?

Resumindo: define-se um ponto abaixo do personagem (ponto branco), lança-se uma linha entre esse ponto e o centro do personagem (linha amarela) e verifica se em algum momento essa linha toca um objeto que está na camada "Ground". Se sim, o personagem está no chão e apto a pular. Ok? Entenda bem isso antes de passarmos adiante para o código e adição desse ponto ao nosso jogo!

Vejamos, na **Listagem 3**, o código necessário para que o nosso pulo funcione como descrevemos anteriormente.

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

public class PlayerController : MonoBehaviour {

    private bool jumping;
    private bool grounded = false;

    private Rigidbody2D rigidBody;
    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> ();
    }

    // Update is called once per frame
    void Update() {
        grounded = Physics2D.Linecast(transform.position, groundCheck.position,
        layerMask);

        if (Input.GetButtonDown("Jump") && grounded)
        {
            jumping = true;
        }
    }

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

        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;
        }
    }
```

```
52 }
```

Listagem 3 - Adicionando o código de pulo ao nosso personagem.

Código novo, funções novas, objetos novos, tudo novo de novo! Vamos então passar por cada novidade com calma e explicando direitinho.

Primeiramente, observemos que agora há duas variáveis booleanas, uma chamada *jumping* e a outra chamada *grounded*. A *jumping* será responsável por dizer se o personagem está tentando iniciar um pulo ou não, enquanto a *grounded* vai dizer se ele está no chão ou não. Como vemos no método Update(), o personagem só será capaz de iniciar um pulo (*jumping = true*) quando o botão de pulo estiver pressionado e ele estiver no chão.

A próxima alteração foi adicionar uma referência ao *groundCheck*, como um transform, uma vez que estamos apenas interessados em sua posição. Definimos também uma variável pública chamada *layerMask*. Essa variável armazenará qual a camada que será procurada pela nossa linha, quando a traçarmos. **Não esqueça de definir o valor dessas variáveis no editor, uma vez que elas são públicas!**

Por fim, definimos a variável jumpSpeed, a qual armazena, em ponto flutuante, o valor a ser utilizado como força para o pulo do nosso personagem.

Definidas as variáveis, adicionamos o código do pulo em si, tanto utilizando o Update quanto o FixedUpdate. Mais uma vez, as alterações no motor de física foram para o Fixed!

No método Update adicionamos um teste, executado a cada frame, relacionado à posição do personagem em relação ao chão. A variável booleana *grounded* que havíamos criado vai receber o resultado de um **Linecast** partindo da posição do próprio transform até a posição do groundCheck, buscando algum objeto da camada layerMask. Caso encontre, será retornado *True*, indicando que o objeto está presente e, consequentemente, estamos no chão. Caso contrário, estaremos no ar.

Definido se estamos no chão, verificaremos se o jogador pressionou o botão de pulo. Caso estejamos no chão **e** o botão de pulo esteja pressionado, pularemos. Perceba que utilizamos, para o botão de pulo, um eixo chamado "Jump", como havíamos estudado na Aula 03. Assim mantemos todas as facilidades que a utilização de um eixo nos traz!

Para finalizar, precisamos apenas adicionar o pulo em si! E aí, como estamos trabalhando com uma força e o motor de física, movemos isso para o método FixedUpdate(). Caso estejamos, de fato, pulando (as duas condições foram atendidas), adicionamos uma força vertical ao nosso RigidBody no valor de *jumpSpeed* e então dizemos que não estamos mais pulando (*jumping = false*).

Com isso, finalizamos o nosso script de pulo! Para que o personagem pule, falta apenas um detalhe: o *groundCheck*! Para que possamos utilizar a técnica descrita com o script demonstrado, precisamos criar o objeto groundCheck e defini-lo no script para que o Linecast funcione adequadamente.

Para criar o groundCheck, devemos selecionar o nosso player, clicar com o botão direito nele e escolher a opção Create Empty. Isso vai criar um novo objeto, filho do nosso objeto Player e que não contém nada além de sua posição. Renomeie esse objeto para *groundCheck*. Criado esse objeto, devemos modificar o seu valor de Y para que fique abaixo do jogador. Estou usando Y = -3.25, mas fique à vontade para adequar o valor ao que for necessário para o seu personagem, caso ele seja diferente em tamanho ou posicionamento. O importante é que o objeto fique abaixo dele, assim como mostrado com o círculo branco na **Figura 6**!

Feito isso, basta selecionar o seu personagem, ir até o script modificado há pouco e selecionar o nosso objeto groundCheck como sendo o groundCheck do script. Com isso, tudo o que precisamos para o nosso personagem pular estará definido e seremos capazes de, ao apertar Play em nosso jogo, utilizar a barra de espaço para ver o nosso personagem pulando.

Coisas a verificar caso o personagem não esteja pulando:

- O groundCheck está posicionado adequadamente e selecionado no script?
- O chão está marcado com a Layer Ground?
- A tecla do eixo Jump está definido? (caso não, reveja a Aula 03)
- O jumpSpeed está alto o bastante para tirar o personagem do solo?
- A layerMask está definida adequadamente de acordo com a layer do chão?

Se tudo isso estiver ok e seu personagem ainda não estiver pulando, fique à vontade para utilizar os fóruns e tirar todas as suas dúvidas! Estaremos lá para trocarmos ideias! ;)

Pulo Duplo

O segundo tipo de pulo que veremos em nossa aula é o pulo duplo, mecânica muito comum em jogos de plataforma. Essa mecânica permite ao personagem, após o seu primeiro pulo, mesmo que não esteja no chão, executar um segundo pulo, voltando a ganhar aceleração vertical.

Figura 07 - Personagem com o P2 na cabeça executando um pulo duplo.

Fonte: Jogo Smash Bros by Nintendo

Como existem muitas maneiras de utilização desse pulo, o assunto torna-se interessante de ser abordado, visto que apenas adicionando essa mecânica ao seu jogo já é possível adicionar alguma variação a mais a ser considerada pelo jogador em suas tomadas de decisões.

Sabe-se que há várias maneiras de se implementar o pulo duplo, entretanto aqui abordaremos uma delas, mas não estranhem se encontrar outras em suas aventuras pelas *interwebs*. Manteremos a lógica que utilizamos para o pulo simples, porém, adicionando a capacidade ao personagem de executar um segundo pulo enquanto está no ar.

Com isso, notemos que existirão três situações com as quais iremos lidar: o caso em que o jogador está no chão e ainda não pulou; ele já pulou e pode executar um segundo pulo; e o jogador já executou o pulo duplo. Cada caso desse será contemplado por uma condicional diferente e gerará uma reação diferente do jogo.

Vejamos o código da **Listagem 4**.

```
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 Rigidbody2D rigidBody;
    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> ();
    }

    // Update is called once per frame
    void Update() {
        grounded = Physics2D.Linecast(transform.position, groundCheck.position,
        layerMask);

        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()
    {
        float moveH = Input.GetAxis ("Horizontal");

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

        if (Mathf.Abs (rigidBody.velocity.x) > maxSpeed) {
```

```
52          Vector2 vel = new Vector2 (Mathf.Sign (rigidBody.velocity.x) * maxSpeed,
53          rigidBody.velocity.y);
54          rigidBody.velocity = vel;
55      }
56
57      if (jumping) {
58          rigidBody.AddForce(new Vector2(0f, jumpSpeed));
59          jumping = false;
60      }
61      if (doubleJumping) {
62          rigidBody.velocity = new Vector2 (rigidBody.velocity.x, 0);
63          rigidBody.AddForce(new Vector2(0f, jumpSpeed));
64          doubleJumping = false;
65      }
66
67  }
68 }
```

Listagem 4 - Adicionando o pulo duplo ao personagem.

Conforme observado, as modificações não são muitas, dessa vez. Criamos duas novas variáveis booleanas para lidar com o pulo duplo. São elas: *doubleJump* e *doubleJumping*. A primeira, indica se o personagem pode executar o segundo pulo ou não. A segunda indica se ele tentou executar esse pulo.

A lógica utilizada, como dito anteriormente, permanece a mesma. No update, vemos se o personagem está no chão e se o botão de pulo foi pressionado. Caso sim, avisamos o pulo ao FixedUpdate com a variável *jumping* e também ativamos a habilidade de executar um segundo pulo (*doubleJump = true*). Caso a tecla tenha sido pressionada quando o personagem não está no chão e pode dar um pulo duplo, avisamos ao FixedUpdate que ocorrerá um pulo duplo.

Já no FixedUpdate, verificamos duas alternativas e não mais uma apenas. Se o caso for de Jumping, adicionamos uma velocidade vertical ao nosso personagem e retiramos o pulo. Caso seja de DoubleJumping, além de adicionar uma velocidade ao nosso personagem e tirar o pulo duplo, zeramos a velocidade dele em Y antes de adicionar essa força. Fazemos isso justamente para que o personagem possa subir, caso já esteja caindo, uma vez que, se a velocidade não fosse zerada antes do pulo ser ativado, ela apenas agiria contra a gravidade por um instante, diminuindo a velocidade de queda evitando jogar o personagem para cima novamente.

Às vezes esse pode ser o comportamento desejado, por exemplo se tentarmos fazer um jogo no qual o personagem fica flutuando com pulos (Flappy Bird). Nesses casos, podemos retirar essa linha onde a velocidade é zerada e simplesmente adicionar a força normalmente.

Bem legal, não acha? Espero que todos tenham entendido e possam adicionar isso ao jogo sem maiores problemas. Qualquer dúvida, convido-os mais uma vez a discutir no fórum!

Wall Jumping

O último comportamento o qual abordaremos em nossa aula sobre pulos é bem peculiar, uma vez que não envolve diretamente pular do chão e sim pular das paredes! O Wall Jumping adiciona uma nova dimensão aos seus jogos, permitindo que os jogadores interajam com as paredes do cenário, seja utilizando-as para deslizar mais lentamente ou mesmo para apoio ao receber uma nova carga de pulo.

Figura 08 - Zero pulando nas paredes para avançar verticalmente em um cenário.

Fonte: Jogo Megaman by Nintendo.

Já aprendemos, ao longo da aula, diversas técnicas relacionadas ao pulo, como a criação de camadas, a adição de elementos vazios para utilizar como marcadores, o teste de interseções, etc. Todas essas técnicas também deverão ser utilizadas para a implementação desse comportamento no jogo! Só precisamos lidar com isso de maneira diferente, concorda?

Necessitaremos de uma Layer que seja capaz de representar as paredes. Precisaremos também de uma nova maneira de detectar se houve ou não uma colisão com a parede e se o personagem pode pular a partir da parede ou não. Isso o colocará voando! Permitiremos um pulo duplo? Não permitiremos? Todas essas são decisões de design que influenciarão na maneira que você implementa o Wall Jumping. Tenha em mente, no entanto, que os conceitos utilizados serão os mesmos!

> Há também uma maneira diferente de desenvolver o pulo utilizando outro tipo de técnica para detectar o chão. Veja mais informações neste link:
>
> <http://www.fabricadejogos.net/posts/tutorial-jogo-de-plataforma-no-unity-5-parte-6-pulo/>

E com isso, finalizamos a nossa aula de hoje, relacionada ao pulo do personagem em um jogo de plataforma! Em nossa próxima aula, voltaremos a lidar com a imagem de nosso jogo e aprenderemos como, em meio a tudo o que já fizemos, adicionamos animação a ele! Vamos fazer o robozinho se mexer enquanto se movimenta? :P

Até lá! \o

Leitura Complementar

Tipos Diferentes de Pulo Simples

<https://critpoints.wordpress.com/2015/05/18/5-games-5-jumps>

Tutorial: Jogo de Plataforma no Unity 5 – Parte 6 – Pulo

<http://www.fabricadejogos.net/posts/tutorial-jogo-de-plataforma-no-unity-5-parte-6-pulo/>

Resumo

Na aula de hoje, abordamos uma mecânica muito importante para o desenvolvimento de jogos de plataforma: o pulo. Aprendemos sobre sua importância e seus diferentes tipos.

Para acomodar bem o pulo em nosso jogo, fizemos algumas alterações no projeto, adicionando uma velocidade máxima e removendo os controles verticais do eixo Vertical. Em seguida, estudamos o pulo simples e o pulo duplo e como estes podem ser implementados em nosso jogo.

Por fim, falamos sobre Wall Jumping e como isso pode adicionar variações e possibilidades ao seu jogo, lembrando que as técnicas de implementação são as mesmas.

Em relação ao Unity, aprendemos nessa aula sobre Layers e Tags e como elas **não são exportadas junto aos Assets**. Também estudamos mais alguns métodos importantes no código em C# e a utilização de objetos vazios apenas para marcar posições, além de vermos como funciona a técnica de Line Casting.

O projeto, como sempre, estará disponível aqui, mas agora é necessário que vocês adicionem manualmente as layers e tags, sempre que existirem, uma vez que essas configurações não são exportadas.

Autoavaliação

1. Quais são as principais técnicas de pulo utilizadas em jogos de plataforma 2D?
2. Quais os três casos que podemos encontrar ao utilizarmos um pulo duplo?
3. O que é Line Casting? Como isso é implementado?

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.

Chris Wagar. 5 Games, 5 Jumps. Disponível em: https://critpoints.wordpress.com/2015/05/18/5-games-5-jumps/.