Desenvolvimento com Motores de Jogos I

Aula 03 - Movimentação - Scripts e Conceitos

Apresentação

Olá, gamers! Estamos de volta para a nossa terceira aula da disciplina de Desenvolvimento com Motores de Jogos I! Temos alguns conhecimentos preliminares no motor de jogo Unity e já começamos a ter, baseado no que foi desenvolvido nas aulas anteriores, algo com cara de jogo... Mas só com cara, né?! Afinal, o que temos nem se move! E isso é o mais básico dos jogos, certo? Desde os primórdios, já tínhamos um controle envolvido pois, se bem lembramos, parte do que define um jogo é a possibilidade que temos de interagir com ele, correto?! Já imaginou se existisse um tipo de jogo onde você está de frente a uma bela história, com bons personagens, um gráfico maravilhoso e nenhum controle do que está acontecendo?! Ah é! Chamam isso de filme, acho. LUL

Para começarmos a solucionar esse problema, vamos, na aula de hoje, começar a estudar uma parte muito importante do desenvolvimento de jogos - o Input. Utilizamos esse nome como maneira de generalizar todo e qualquer tipo de entrada que o nosso jogo poderá receber, seja qual for o lugar de origem. O Unity principalmente trabalha muito bem com essa ideia de ter um Input genérico, de fonte não tão conhecida. Isso é interessante principalmente quando pensamos em desenvolver um jogo multiplataforma. Nesse caso, poder receber entradas de um teclado, mouse, touchscreen ou mesmo um controle regular pode ser a diferença entre conseguir portar o jogo ou não.

E lembre-se também que o PC é uma plataforma capaz de receber diferentes entradas, assim como é o smartphone. É possível utilizar suas interfaces regulares ou conectar algum gamepad especial para assumir os controles do jogo e tornar a experiência como um todo bem mais divertida. Já vamos ficar com isso em mente! Ah! E nessa aula começaremos, finalmente, a escrever algum código! Prontos para a Aula 03?! Então vamos adiante!

It's dangerous to go alone! Take this!

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Utilizar scripts para realizar funções básicas no Unity.
- Receber e tratar inputs do usuário de maneira a tornar o jogo controlável
- Entender os diferentes tipos de input que o Unity é capaz de receber e como a utilização de cada um deles pode ser útil no desenvolvimento de seu jogo.

Movimentação em Jogos - Lidando com o Input

A movimentação das personagens é parte crucial de qualquer jogo. Rabin [3] diz que "um jogo é uma experiência interativa, de modo que uma das tarefas mais importantes é a de reunir os inputs (entradas de dados) do jogador e reagir de acordo com eles." Como já conversamos na apresentação, esses inputs podem vir de fontes diversas, desde as mais clássicas, como mouse, teclado e gamepad, até as mais diferentes formas de interação, como câmeras, áudio, voz e até mesmo ondas cerebrais.

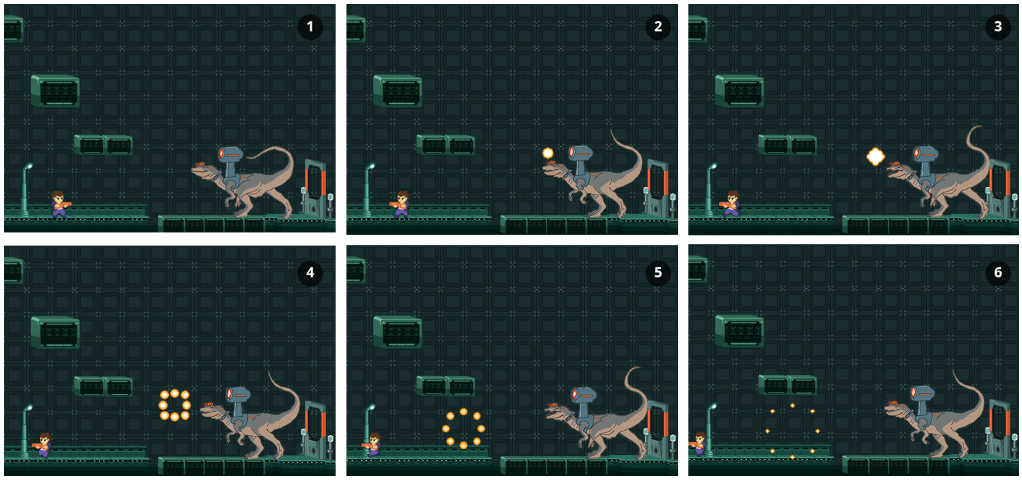
Em comum entre todas essas formas de interação, no entanto, nós temos um aspecto muito importante. O input deve sempre ser **preciso** e **instantâneo**. Sabemos que ser instantâneo é fisicamente impossível para jogos eletrônicos, uma vez que há, no mínimo, o tempo de transmitir o sinal e o processar. Isso, no entanto, não é desculpa para fazermos jogos com entradas terríveis, que frustram o usuário rapidamente. Podemos (e vemos isso em jogos desde os primórdios dos jogos) trabalhar com um processamento rápido desses inputs de modo a reduzir ao máximo o *lag* ou *delay* que temos desde o botão ser apertado até algo acontecer no jogo, buscando tornar isso *visivelmente* instantâneo, ou seja, rápido o bastante para o nosso cérebro não perceber o problema.

Já na questão de precisão, precisamos sempre garantir que nenhuma entrada do usuário será perdida. Se o botão B é o responsável por pular e o usuário apertou o botão B, o personagem deverá pular (se as restrições do jogo permitirem ele pular no momento, claro). Caso o personagem esteja parado, pronto para pular, e isso não aconteça assim que o usuário apertar o botão, isso irá causar uma frustração no jogador e será um fator importante na quebra de imersão do usuário, podendo ser um motivo para que ele perca o interesse no jogo.

O ideal para se resolver esses dois problemas é trabalhar com uma boa **taxa de atualização**. Um conceito muito comum que vemos em jogos diversos e que gera muita discussão é a questão do FPS, sigla de Frames Per Second, ou Quadros por Segundo. Essa medida nos dá quantas vezes a tela está sendo atualizada a cada segundo para garantir que o usuário tenha a sensação de fluidez na imagem. Existem algumas taxas de atualização conhecidas, como o cinema, que trabalha com

24 quadros por segundo, ou os videogames que operam a 30 quadros por segundo. Monitores de 60hz utilizam uma taxa de atualização de 60 FPS. Hoje em dia, já temos os ~~maravilhosos, lindos e perfeitos~~ caríssimos monitores de 144hz, capazes de atualizar a tela 144 vezes a cada segundo, permitindo uma performance de 144 FPS para os jogos. Já a Realidade Virtual, por exemplo, trabalha com 90 FPS.

"E o que eu ganho com ter mais FPS/taxa de atualização?", você pode se perguntar. Você ganha em fluidez, eu te digo. "Ah, mas o olho humano só enxerga 30 fps!", você pode responder. "CONSOLE PEASANT! PC MASTER RACE!!" (link: https://pt.wikipedia.org/wiki/PC_Master_Race), direi! :P



A verdade é que quanto mais frames por segundo tivermos, mais fluida a imagem irá parecer. O olho humano não é uma máquina digital para ter uma "taxa de atualização". E isso tudo influenciará diretamente no seu Input também. Lembra que falamos que quanto menos o seu input demorar a aparecer, melhor o jogador se sentirá? Pois é! A fluidez com que o seu jogo está rodando e o momento correto da captura desse input é uma maneira importante de garantir que o usuário receberá uma resposta quase instantânea.

O ideal quando estamos trabalhando com Inputs é **manter a taxa de atualização de seus comandos igual à taxa de atualização de quadros do seu jogo**, pois assim poderemos garantir que cada comando dado influenciará um quadro causando uma reação quase que imediata. Se você está trabalhando com uma taxa de 30 quadros por segundo, visando sistemas com processamentos inferiores, você deverá mostrar o seu dispositivo de entrada 30 vezes por segundo

também. Mais precisamente, a primeira coisa que você deverá fazer ao começar um novo quadro será receber o valor do Input para poder processá-lo adequadamente ao longo daquele quadro.

Para evitar qualquer discrepância e também lag desnecessário ao aumentar o processamento do seu jogo, é importante também que o input recebido no começo de um frame seja considerado valido por todo aquele frame, evitando qualquer alteração ou mudança por parte do usuário e também evitando ter que buscar novamente por um valor toda vez que for necessário tomar decisões baseadas no Input.

Perceba que tudo isso que estamos falando é muito importante quando se está desenvolvendo um jogo inteiro "no braço", criando cada uma de suas partes e seu loop principal inteiro. No entanto, esse conceito também é válido para motores de jogos e consequentemente para o Unity, uma vez que o Unity disponibiliza em seus scripts uma função exclusiva para que possamos adicionar etapas ao loop principal que ele mesmo é responsável por gerenciar. E aí usamos esse conhecimento adquirido para incluir em nosso loop um comando de receber os inputs! Como fazer isso?! Vejamos adiante!

Atividade 01

1. Em relação ao jogo que você está desenvolvendo, qual a taxa ideal de atualização para os seus inputs? Como ela se relaciona com as taxas de quadros do seu jogo?

2. Quais os tipos de Input que podemos ter em um jogo? Cite 5 com exemplos de jogos que os utilizam.

Utilizando o Input no Unity

Receber e utilizar corretamente os inputs do usuário é uma funcionalidade imprescindível para qualquer jogo de sucesso desenvolvido visando qualquer plataforma. Para garantir essa estabilidade e funcionalidade o Unity tem uma classe

inteira dedicada a tal feito. A classe Input, parte da classe UnityEngine, foi criada inteiramente para lidar com o input e dar ao desenvolvedor diversas facilidades ao cumprir tal tarefa.

O Unity, através dessa classe, suporta todos os tipos de input convencionais utilizados em jogos, tais como teclado, mouse e controle (gamepad). Além disso, já é nativo ao Unity receber entradas de touchscreen (com suporte a multi-touch) e também acelerômetros dos dispositivos móveis. Essa funcionalidade é uma facilidade tremenda para os desenvolvedores que querem trabalhar com essas plataformas, que são verdadeiras minas de ouro atualmente e que permitem trabalhar com menores equipes e atingir excelentes resultados no desenvolvimento, como vimos nas aulas de Introdução aos Jogos Digitais.

Utilização de Inputs Tradicionais no Unity

O Unity, ao trabalhar com Inputs tradicionais, já traz diversas facilidades ao usuário. É possível, por exemplo, configurar os comandos que o usuário quer utilizar para cada um dos inputs necessários ao controle do jogo ao iniciá-lo no PC. Como veremos adiante, basta definir o que nós queremos dar de liberdade ao usuário em relação ao recebimento de entradas e aí, ao executar o jogo após ser compilado em versão final, o usuário poderá simplesmente escolher as teclas que quer utilizar para cumprir cada uma das funcionalidades. Não quer pular no B? Altere uma configuração e agora você pula na barra de espaço! Sem nenhuma necessidade de código extra por parte do desenvolvedor. Muito bom, não?!

É possível utilizar qualquer um dos dispositivos de entrada tradicional no Unity sem que haja nenhuma necessidade de configuração extra, apenas utilizando a classe Input. Podemos, inclusive, utilizar interfaces diferentes misturadas, da maneira que quisermos, para o nosso jogo. Sente falta do clássico multiplayer em uma máquina só que os consoles de antigamente traziam? Não tem problema! Isso está a alguns comandos de distância no Unity. Reviva essa experiência única! E me mande o link do jogo pois terei interesse em adquirir! :D

Para que esses inputs tradicionais possam funcionar adequadamente e de maneira fácil, o Unity utiliza o conceito de **eixos virtuais**. Esses eixos nada mais são do que nomes predefinidos para que possamos acessar os valores de algumas

teclas clássicas, utilizadas em jogos de computador de diversos estilos. Os eixos padrão que o Unity traz são:

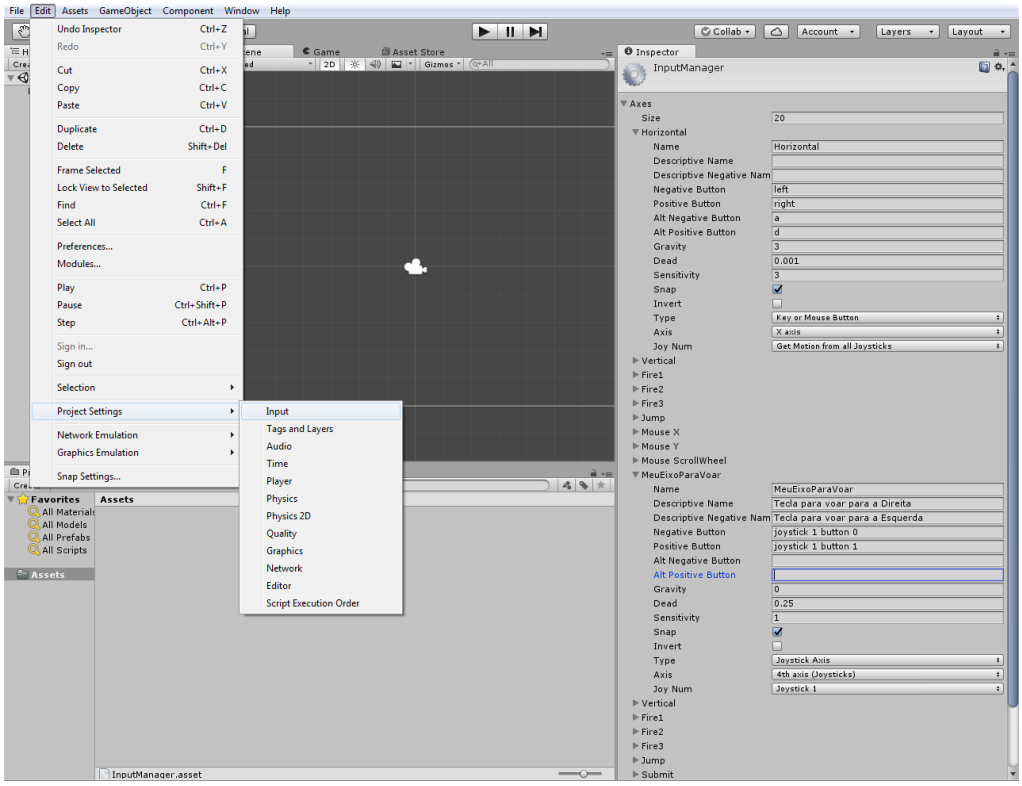
- **Horizontal e Vertical**, que representam a movimentação do usuário e estão por padrão mapeados para as setinhas do teclado ou para WASD.

- **Mouse X e Mouse Y**, que indicam a movimentação do mouse em relação a um desses eixos.

- **Fire1, Fire2 e Fire3**, que podem ser utilizados para disparar algum evento como tiro, pulo ou qualquer outra coisa e estão mapeados, na sequência, para Control, Alt e Command.

Com todos esses elementos já é possível desenvolver muitos controladores diferentes para os nossos jogos. Se isso não for o bastante, como usualmente não é, o Unity também permite que você adicione eixos customizados, dentro das opções de seu programa. Isso é importante para garantir ao programador a facilidade de alcançar os controles que necessita para o seu jogo independente do quão diferente dos padrões ele for. Para acessar o menu de adição de novos eixos, utilizamos a opção Edit -> Project Settings -> Input. Isso abrirá uma nova aba em nosso Inspector indicando as possibilidades que temos para modificar os eixos. A **Figura 1** nos mostra como abrir o menu e também a criação de um novo eixo, representando os controles de vôo de um personagem, por exemplo.

Figura 01 - Configurações de Input e Criação de um novo Eixo.



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

Perceba que para a criação de um novo eixo, diversa propriedade para esse eixo tem que ser alteradas. Além disso, no começo das configurações, há uma propriedade chamada Size. Ela indica quantos eixos serão criados para o seu projeto. Caso não haja espaços para novos eixos, basta aumentar essa variável e então novas posições surgirão para que você possa utilizar.

Em seguida, temos as propriedades relacionadas aos eixos em si. São diversas propriedades diferentes e cada uma com uma funcionalidade diferente. Vejamos, a seguir, a função de cada uma dessas propriedades:

- **Name** - Indica o nome que o eixo receberá. Isso é importante para que possamos acessar esse eixo durante a programação.
- **DescriptiveName** - Indica o nome que aparecerá para o jogador durante a etapa de configuração dos comandos. Deve ser um nome que facilite a identificação por parte do jogador para que ele possa fazer a configuração adequadamente.

- **Descriptive Negative Name** - Indica o nome que aparecerá para o jogador configurar o lado negativo do eixo em questão.

- **Negative Button** - Qual o botão que será utilizado por padrão para mover o eixo na direção negativa.

- **Positive Button** - Qual o botão que será utilizado por padrão para mover o eixo na direção positiva.

- **Alt Negative Button** - Botão alternativo para o eixo negativo.

- **Alt Positive Button** - Botão alternativo para o eixo positivo.

- **Gravity** - Velocidade com a qual o eixo retorna ao centro caso nenhum botão esteja sendo pressionado.

- **Dead** - Caso esteja sendo utilizado um joystick analógico, essa propriedade indica o espaço que será considerado "morto", ou seja, que não haverá movimentação do eixo mesmo havendo a movimentação de leve do joystick. Isso é útil para evitar que o personagem se mova loucamente por uma pequena folga no eixo, ou um dedo descansando em cima.

- **Sensitivity** - A sensibilidade indica a velocidade que o eixo vai crescer na direção apontada (positiva ou negativa). Isso funciona apenas em controles digitais, uma vez que os analógicos podem construir a sensibilidade através do próprio controle.

- **Snap** - Essa caixa, ao ser ativada, indica que o eixo deve ser imediatamente zerado quando pressionado nas duas direções simultaneamente.

- **Invert** - Essa opção inverte o eixo. Isso é muito utilizado em jogos de avião, por exemplo, onde pressionar o botão para cima faz o avião ir para baixo.

- **Type** - Tipo de input que será utilizado para controlar o eixo.

- **Axis** - O eixo do dispositivo escolhido sendo tal eixo o utilizado para controlar o eixo criado.

- **Joy Num** - Se haverá algum joystick específico que controlará esse eixo, ele deve ser discriminado nessa opção. Caso não seja, qualquer joystick terá a opção de controlar o eixo.

Note que são diversas propriedades e cada uma tem suas especificidades. Temos a lista completa acima, como referência, mas não é necessário decorar isso tudo. Caso seja necessário, pode voltar à aula e dar uma olhada, ou mesmo utilizar o manual de referência do Unity. Para facilitar, o Unity também tem pequenas dicas ligadas a cada uma das propriedades no próprio editor - basta posicionar o mouse em cima e aguardar alguns instantes até que a dica apareça.

Veja que um eixo é, usualmente, um valor entre -1 e 1, que varia a medida que uma das teclas é pressionada para um lado ou para o outro. Ou seja, utilizamos eixos, usualmente, para fazer a movimentação de nossos personagens, seja ela horizontal, vertical ou como o jogo permitir. Podemos criar outros eixos para representar outros tipos de modificações, como fizemos no exemplo criando um eixo para vôo do personagem, mas não temos nenhuma obrigação de utilizar esses conceitos ou mesmo criar novos eixos. Encare isso como uma facilidade que o Unity nos traz!

Para mover nosso personagem para esquerda e direita, ter uma tecla que indica +1 e outra -1 é bem simples, correto?! Ah, mas e para pular?! E para atirar? E para lançar foguinho da mão depois que eu pegar uma florzinha no meu jogo novo que estou criando e tive a ideia inovadora de adicionar isso? Eu preciso de um eixo para isso? Na verdade, precisa sim! Apesar de não ser obrigatório. Vamos entender melhor.

Quando queremos disparar um evento baseado apenas no fato de uma tecla ter sido pressionada ou não, não precisamosnos preocupar com eixos, ou com valores +1, 0 e -1. Basta que nós detectemos que essa tecla foi pressionada e aí então podemos agir de acordo, iniciando a ação que gostaríamos de iniciar! Nesse caso podemos configurar um eixo como visto anteriormente, porém, apenas com teclas para as modificações positivas, com uma gravidade e uma sensitivity muito alta e nada de deadzone. Com isso, transformamos o nosso eixo em apenas um click! E aí

sim conseguimos trabalhar as vantagens de tornar as teclas configuráveis. Entende?! Vejamos um exemplo da diferença de configurar um eixo mesmo quando não se é necessário:

- Na primeira opção, criamos um novo eixo, chamamos ele de Jump e dizemos que quando a tecla positiva do Jump for apertada, que configuramos para barra de espaço, o personagem irá disparar um evento de pulo.
- Na segunda opção, não criamos qualquer eixo e vamos direto para o código. Nesse caso, dizemos diretamente Input.GetKey ("space") e isso faz com que seja disparado o nosso evento de pulo.

Os dois casos podem parecer funcionar da mesma maneira, correto? Mas... eSeOEspaçoDoMeuTecladoQuebrar?NãoConseguireiMaisJogarOSeuJogo,Né?:(Pois é! Essa é a principal importância de se utilizar os eixos. Caso tivesse utilizado a primeira opção, poderia simplesmente reconfigurar o menu de entrada para que o pulo fosse no "leftctrl" e aí conseguiria prosseguir sem problemas, até conseguir um teclado novo! Então, vamos tentar, sempre que possível, trabalhar com eixos, ok?!

Só devemos ressaltar que há uma diferença **importante!** Quando formos trabalhar com movimentações e eventos que o eixo realmente importa, vamos utilizar a função Input.GetAxis("EIXO"). Já quando formos utilizar um eixo apenas pelas facilidades que ele nos trás, representando um botão qualquer, utilizaremos a função Input.GetButtonDown("EIXO"). Estamos combinados?! :)

Pause! Ufa! Quanta informação! Vamos tomar um copo de água e dar uma relembrada em tudo que passamos até agora para que possamos seguir para a nossa próxima parte e ver os últimos detalhes antes de retomar o nosso projeto do jogo!

Conceitos importantes:

- Input deve ser preciso e instantâneo, tendo como uma boa referência a taxa de frames. Capturar a cada frame o valor e utilizar esse valor como verdadeiro pelo resto da iteração.

- O Unity trabalha com o conceito de eixos para facilitar a configuração e a programação para recebimento de Inputs, nomeando teclas para mascarar quais realmente são.

- Devemos utilizar eixos sempre que possível, com valores positivos e negativos para movimentação, e somente positivos para triggers.

    Ex: Eixo Horizontal: -1 no A e +1 no D
  Eixo de Pulo: +1 no Espaço

- Para capturar o eixo de movimentação com seus dois valores, utilizamos Input.GetAxis("EIXO").

- Para capturar o eixo de eventos, com apenas o seu valor de pressionado ou não, utilizamos o Input.GetButtonDown("EIXO") em nosso script.

Antes de começarmos a nossa seção de código propriamente dita, vamos discutir mais dois pontos importantes em relação aos eixos no Unity. O primeiro deles é que **é possível ter mais de um eixo com o mesmo nome**. Isso é importante, pois podemos configurar mais de um tipo de input para fazer a mesma função. Pense, por exemplo, que você está desenvolvendo um jogo 2D de plataforma para PC e quer que o seu personagem se mova no WASD ou nas setas do teclado. Isso é tranquilo e já vem até como padrão para o eixo *Horizontal*.

Mas e se eu quiser também que seja possível fazer uma movimentação através de um joystick que tenha sido conectado ao PC? Simples! Podemos criar outro eixo também chamado *Horizontal* e então fazer com que esse eixo se comporte da mesma maneira, porém recebendo inputs do joystick.Nesse caso, o Unity vai sempre pegar o que tiver o valor mais representativo. Ou seja, se o seu teclado estiver em 0 (parado) e o seu joystick indicar uma movimentação -1, o Unity trará o -1 para o seu código, indicando que houve movimento e permitindo que você posicione de acordo. Super simples, não?!

O segundo ponto importante que precisamos ver antes de avançar é em relação aos nomes das teclas. Cada tecla tem um marcador específico que indica ao código que ela é a tecla que está sendo utilizada. Esses marcadores podem ser vistos em

detalhes na documentação oficial do Unity. Pare um pouco e vá lá dar uma olhada rápida. Eles são bem intuitivos, mas é bem importante lembrar deles na hora de criarmos os nossos eixos para a nossa programação! Segue o link:

https://docs.unity3d.com/Manual/ConventionalGameInput.html

É interessante notar que há teclas para botões diversos no mouse, já que temos aqueles mouses gamers com 20 botões. Também temos marcadores específicos para pegar de qualquer Joystick e de apenas um joystick que seja numerado como primeiro, segundo, etc... Também temos acesso às teclas especiais, de modificação e de funções, como F1, F2, etc... Apesar desses diversos nomes, todos são apenas strings que funcionam da mesma maneira em seu código e te permitem trabalhar com todas as teclas igualmente. Mais uma facilidade da utilização de um motor de jogos!

Agora que já discutimos bastante todas as funcionalidades de Input e como o Unity lida com tudo isso, podemos seguir para o nosso projeto que começamos aula passada, mais uma vez. Vamos fazer o nosso amigo robô se mover?!

Adicionando o Primeiro Script ao Projeto

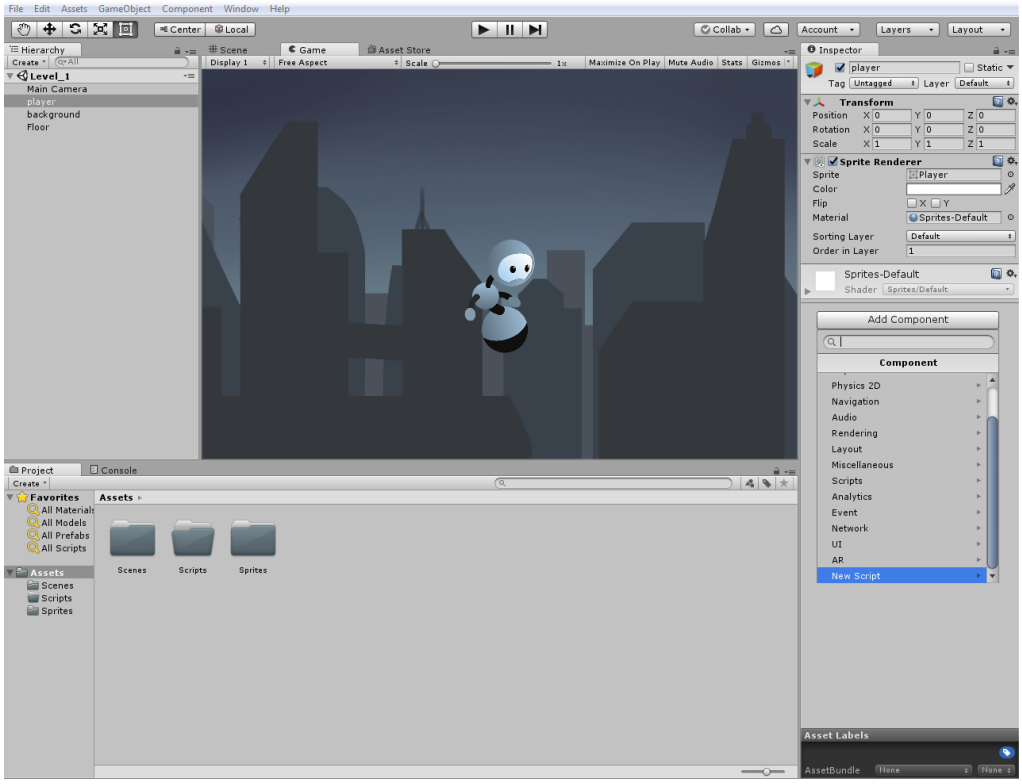
Continuando o que fizemos na aula passada, o próximo passo para o nosso projeto agora é adicionar uma movimentação ao nosso robô! Vamos começar abrindo o projeto que chamamos de Projeto DMJ I. Caso você não tenha salvo o projeto da aula passada, é possível baixar o package contendo os assets até onde chegamos na última aula nesse link. Uma vez baixado o arquivo, basta criar um novo projeto com esse nome e então importar o pacote clicando no arquivo baixado ou utilizando o menu Assets -> ImportPackage -> CustomPackage. E agora lembre-se sempre de salvar o seu projeto para que possamos continuar evoluindo ele ao longo das próximas aulas! Vamos sempre tentar trabalhar em cima do mesmo jogo para que, ao final, tenhamos desenvolvido juntos um jogo bem completo, como dito que faríamos na Aula 01!

Aberto o projeto, iremos começar a adição de nosso primeiro script em C#! Para isso, o primeiro passo é o mesmo dos outros assets que começamos a criar em outras aulas - criaremos uma nova pasta com o nome Scripts dentro da pasta de

assets para mantê-la sempre organizada, como já conversamos em outras oportunidades! Clica na pasta Assets com o botão direito -> Create -> Folder e então põe o nome Scripts.

Feito isso, vamos criar um novo script em C# para a movimentação do nosso player. Para criar scripts, temos algumas maneiras diferentes. Iremos pela maneira que já adiciona o script como um componente ao objeto que o utilizará. Primeiro, selecione o nosso player na aba de Hierarchy. Feito isso, o Inspector indicará os componentes que estão presentes em nosso objeto. Adicionaremos um novo através do botão Add Componente -> New Script. A **Figura 2** mostra esse procedimento.

Figura 02 - Criando um novo script através do botão AddComponent.



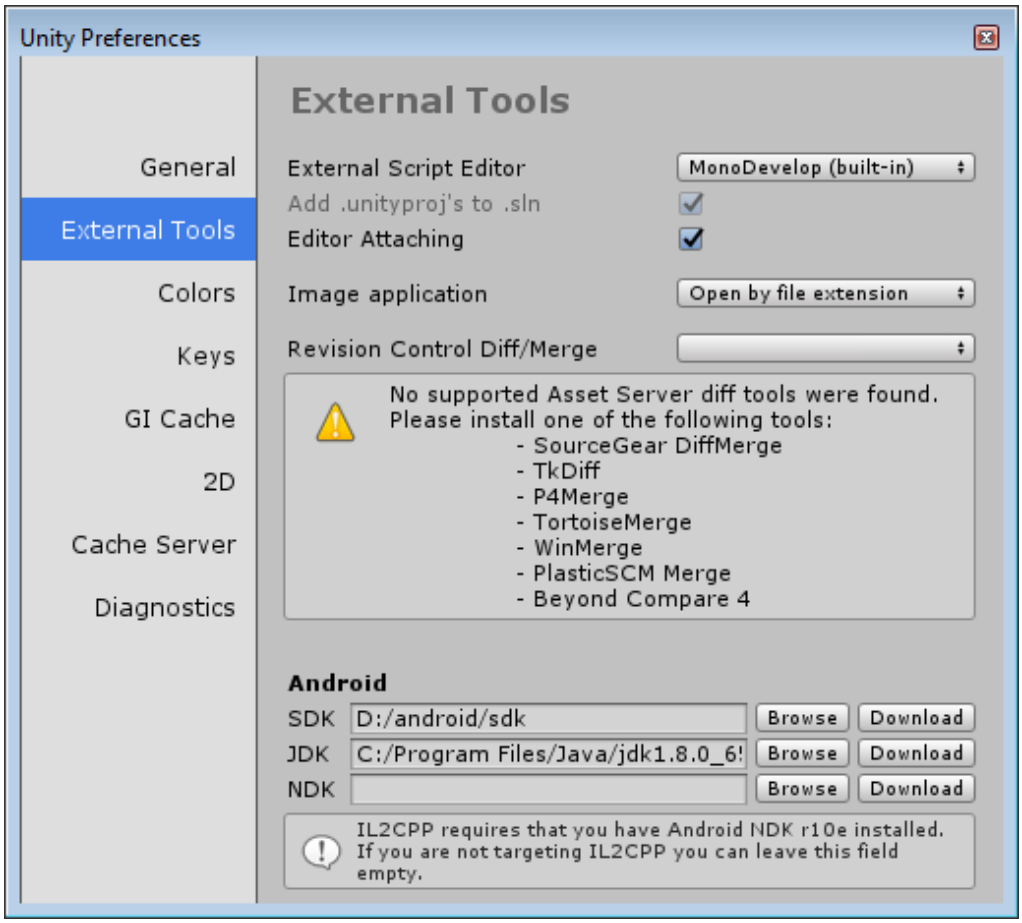
Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/

Ao clicar em New Script, uma nova seção aparecerá solicitando o nome que daremos ao script e também a linguagem que será utilizada para ele. Como já dito, utilizaremos a linguagem **C Sharp** para as nossas aulas. Já em relação ao nome do script, trabalharemos com PlayerController. É importante lembrar bem desse nome pois caso em outras oportunidades se faça necessário referenciar esse script, precisaremos copiar o nome exatamente como é para não haver erros. Isso também levanta outro ponto importante, relacionado ao acompanhamento de vocês.

É fundamental que vocês se mantenham motivados e acompanhem efetivamente os projetos que desenvolveremos aqui, juntamente às aulas. Caso estejam fazendo isso, é importante que vocês prestem muita atenção nos nomes utilizados nos scripts, nas layers e tags e em todos as configurações em geral que vamos fazer, uma vez que essas serão utilizadas em scripts ao longo das aulas. Caso vocês utilizem nomes diferentes para os elementos que vocês fizerem, é importante lembrar de substituir todos quando forem acompanhar os scripts que vamos desenvolver ao longo das aulas. Ok?! Lembrem-se disso!

Seguindo adiante com o nosso script, que chama-se PlayerController, vamos primeiramente cuidar da organização de nossa pasta Assets. É comum que um novo script criado dessa maneira seja posicionado na raiz da pasta. Caso esse seja o caso, basta clicar nele e arrastar até a pasta Scripts que já criamos anteriormente para fazer a manutenção de nossa organização! Feito isso, vamos observar uma configuração relacionada à IDE que irá nos ajudar com nossos scripts. Acesse o menu Edit ->Preferences. Na tela que abrirá, no menu à esquerda, temos a opção External Tools. Clicando nessa opção, veremos a tela como indicado na **Figura 3**.

Figura 03 - Menu de Preferences ->External Tools.



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

Nesse menu conseguimos modificar algumas interações que o Unity faz com componentes externos. Caso tenhamos o Android instalado como Build Target podemos escolher as localizações de SDK e JDK, por exemplo, que servem para fazer a compilação para essa plataforma. Podemos escolher também o que fazer com as imagens. Mas o que nos interessa, no momento, é a primeira opção: External Script Editor. Essa opção diz ao Unity quem será o editor de scripts padrão que será utilizado pelo Unity para abrir os seus scripts e escrever o código que será incorporado em seus jogos. Você pode alterar essa configuração para utilizar o editor de sua preferência, seja ele qual for, através da opção Browse. Nessa aula e no curso como um todo, no entanto, utilizaremos sempre o MonoDevelop como interface padrão para o desenvolvimento. Caso queira acompanhar exatamente como está na aula, fique à vontade! Caso prefira alterar para um de sua preferência, é só saber mexer nele! :D

Após configurar adequadamente, basta fechar a janela e podemos partir para a edição do nosso script de fato. Para iniciar a edição do script, basta clicar duas vezes nele dentro da nossa aba de Project e então o MonoDevelop será iniciado. Perceba que o script, ao abrir já conterá algum código.Esse código inicial pode ser visto na **Listagem 01**.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PlayerController : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9
10     }
11
12     // Update is called once per frame
13     void Update () {
14
15     }
16 }
```

Listagem 1 - Código inicial de um script chamado PlayerController.

Opaaaa!! C#! Massa, né?! Eu acho. E vocês vão achar também, assim que começarem a entender melhor a linguagem. E vejam que não tem nada demais. Vamos entender beeeeem direitinho esse código inicial? É bem importante começar bem, sem dúvidas, para conseguir avançar sem problemas a medida que os scripts ficarem mais complicados. Não se encabulem de perguntar fórum a fora, Moodle a fora, mundo a fora, caso não entendam algo. O feio é levar a dúvida para casa! A não ser que vocês forem pesquisar em casa, né... Aí ok... Mas pesquisem, viu?! A listagem! Vamos deixar de besteirol e vamos a ela!

Nas linhas 1, 2 e 3 da listagem vemos a utilização da palavra chave **Using**. No C#, essa palavra tem uma função parecida com a **Import**, que vimos no Java. Ela indica quais as outras classes que estaremos utilizando em nosso arquivo. Nesse caso, são três e as três estão listadas lá. Tranquilo, né?!

Em seguida, na linha 5, temos a criação da classe em si. Assim como no Java, no C# trabalhamos com o conceito de classes, métodos, objetos, etc... é uma linguagem orientada a objetos, da mesma maneira! Não é estranho a qualquer um de vocês as palavras **public** e **class**, certo?! Em seguida, temos o nome da classe em si, que

sempre tem que ter o mesmo nome do arquivo! PlayerController o script? PlayerController a classe! E aí em seguida temos o :. Esse : quer dizer que a classe deriva de outra. No Unity, em geral, as classes vão derivar da classe **MonoBehaviour**, como a nossa classe já está! Ok?! Garanta que você entendeu cada linha antes de ir para a próxima! E se tá achando easy, parabéns. Nem sempre é! :-P

Nas linhas 8 e 13 temos duas funções ou métodos que são também já incluídos por padrão nos scripts criados. Veja que a maneira como os métodos funcionam no C# são bem similares ao Java também:

Tipo_de_Retorno Nome_da_Função (Parâmetros) {}

Esses métodos já incluídos, no entanto, são métodos especiais, conhecidos como **callbacks**! Esses métodos possuem uma utilização bem específica e interessante que permitem a você interagir com diferentes etapas do seu jogo automaticamente, sem precisar alterar nenhum código do motor diretamente, por exemplo.

A primeira delas, a função Start(), é chamada assim que o objeto é criado. Quando sua inicialização é finalizada, o callback Start() é chamado. Ou seja, não é necessário a você fazer nenhuma chamada ao método Start()! O Unity fará essa chamada automaticamente assim que seu objeto terminar de ser criado. Daí que vem o nome de callback!

Já o segundo método é um mais repetitivo. O método Update é chamado uma vez a cada frame do seu jogo! Isso é excelente para algumas coisas... alguém aí consegue imaginar alguma? Dica: discutimos no começo da aula... Input! Lembra que conversamos mais cedo que o Input deve ser, idealmente, verificado uma vez a cada frame para responder ao usuário o mais rápido possível e dar aquela sensação gostosa de fluidez? Pois é! Se utilizarmos o método Update(), toda novo frame que for desenhado, o Unity chamará automaticamente essa função para que o objeto que possui o script que estamos escrevendo possa tomar alguma decisão para aquele frame!

Aí você pensa... "Poxa! Mas toooodo frame?! Isso não vai ficar pesado?!". Sim. Vai sim. "Mas e aí, professor?! Como faz?!" Bom, aí você não faz nada que seja pesado aqui, né?! Um jogo, hoje em dia, deve rodar a peeelo meeenos 30 FPS. Se você faz

uma operação que demora mais de 30ms para concluir só em um objeto nesse método, já era. Então muito cuidado com o que vai ser feito no Update!!!

E com esses dois métodos a gente conclui nosso entendimento de um script básico, só com o esqueleto, em C#, no Unity. Simples, não?! É bem similar ao Java! Não esqueçam de fechar chaves, de colocar ; no final das linhas de código e tudo mais que vocês já aprenderam do Java! Viram que não tem mistério nenhum?! Vai dar tudo certo. Relax!

Vamos aproveitar agora e fazer uma atividade para ver se vocês já estão craques em scripts e C#?! Para executar um script, é só clicar no botão de play que já vínhamos utilizando anteriormente. Qualquer erro aparecerá na aba Console, que já vimos em aulas anteriores. Vamos lá?!

Atividade 02

---

1. Sabendo que a função **Debug.Log("MENSAGEM");** escreve no console uma MENSAGEM, faça com que o seu player, ao ser iniciado, imprima a mensagem Hello World.

Movimentando o Personagem

Agora que já conseguimos adicionar um script ao nosso personagem e já vimos como é simples toda a interface para lidar com scripts, podemos finalmente escrever o nosso primeiro código! O primeiro passo é definir a maneira que lidaremos com a movimentação do nosso personagem. Existem milhares de maneiras que podemos inventar para fazer um objeto se mover. Vejamos duas das maneiras mais conhecidas:

A primeira maneira é deslocar o personagem no eixo em que o movimento está acontecendo diretamente. Nesse caso, definimos uma **velocidade** em unidades e cada vez que o botão de movimento for pressionado, movemos o personagem naquela direção em **velocidade** unidades. Isso pode, no entanto, gerar uma

movimentação nada fluida ao personagem e pode criar vários problemas com relação à posição do personagem na tela, fazendo com que ele entre em objetos, ou atravesse paredes. Claramente essa não é a melhor maneira, né?!

A segunda maneira é utilizando física! Podemos transformar o nosso objeto em um corpo rígido e então o empurrar por aí, baseado nos Inputs do usuário, fazendo com que o Unity e o seu motor de física façam todos os cálculos de para onde ele deve ir, onde ele deve parar e tudo mais. Parece mais seguro, concorda?! Então vamos à essa abordagem!

Para que possamos trabalhar com a física no Unity precisaremos definir que o nosso personagem é um corpo rígido. Já fizemos isso anteriormente, em nossa primeira aula, para que a bolinha pudesse cair. Naquele caso, no entanto, trabalhamos com física 3D e agora estamos em 2D. Isso é importante pois o Unity trata as duas físicas de maneira diferente e possui componentes para lidar com cada uma delas. Entraremos em mais detalhes sobre isso na próxima aula. Por agora, vamos adicionar um novo componente ao nosso player!

Como vimos na **Figura 2**, podemos adicionar um novo componente diretamente ao objeto no qual queremos ligar o componente através do botão AddComponent. Vamos utilizar esse botão mais uma vez em nosso player para isso. Selecione o player, cliquem em AddComponent e em seguida em **Physics 2D (2D!!!)** e aí em **Rigidbody 2D**, a primeira opção desse menu. Com isso, o nosso player agora terá um **Rigidbody 2D** atrelado a ele e será capaz de responder à física do mundo. Nice, bro!

Como não queremos ainda que ele caia loucamente mundo a fora vamos, por enquanto, remover a gravidade do nosso player. O nosso amigo robozinho, por hora, é capaz de flutuar! Bom para ele. Vamos alterar a propriedade **GravityScale** que o **Rigidbody 2D** possui para 0. Com isso, ele simplesmente não responderá mais à gravidade! Agora que temos como empurrar o nosso robozinho para lá e para cá, já que ele é um corpo rígido, vamos programar o nosso script para fazer justo isso!

Alterando o Script para Empurrar o Robô Mundo a Fora

Voltando ao nosso script, vamos agora começar a mover de fato o nosso robô. Só recapitulando um pouco como a movimentação vai funcionar: o usuário pressionará uma tecla dentro de um eixo - vertical ou horizontal - e então nós empurraremos o robô (utilizando uma força física no corpo rígido) naquela direção que o usuário indicou. Para entender bem, imagine que o nosso robô é uma bolinha em uma mesa de sinuca e o usuário tem o taco utilizando os controles de eixo. Ao apertar, uma força será exercida na bola, empurrando ela por aí, por cima da mesa.

Qual o primeiro passo, então, para alterar no nosso script? Bom... Como precisaremos empurrar o corpo rígido, precisaremos, inicialmente, acessar o corpo rígido em nosso script! Para fazer isso, precisamos buscar o componente que acabamos de adicionar e salvar ele em uma variável. Para fazer isso, criamos inicialmente a variável e, em seguida, no método de inicialização, buscamos o componente em si e guardamos ele em nossa variável. O código fica como está na **Listagem 2**!

```
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      // Use this for initialization
10     void Start () {
11         rigidBody = GetComponent<Rigidbody2D> ();
12     }
13
14     // Update is called once per frame
15     void Update () {
16
17     }
18 }
```

Listagem 2 - Código para salvar o componente RigidBody2D em nosso script.

Perceba que, assim como em Java, é possível criar uma variável como sendo privada ou pública. Como queremos apenas ter acesso a ela internamente, no script, escolhemos para o RigidBody2D uma variável privada chamada rigidBody.

Em seguida, na função Start(), que é chamada automaticamente pelo Unity assim que o objeto é inicializado na cena, atribuímos o valor a essa variável. Como o Unity vai chamar a função após o objeto ser criado, ele já vai possuir esse componente e então, quando chamarmos o método GetComponent<Rigidbody2D>(), teremos o componente retornado e salvo em nossa variável.

Note que esse método de buscar um componente serve para qualquer tipo de componente. Isso é exatamente o que está contido dentro dos <>. Se você, em outra oportunidade, quiser buscar um componente de outro tipo, basta alterar o tipo que está entre <>. Cuidado, inclusive, para não ter colocado <Rigidbody> no seu código, uma vez que esse tipo representa o Rigidbody 3D!!! Ah... e aos curiosos: se houver mais de um componente do mesmo tipo, esse método retorna o primeiro que encontrar e se não houver nenhum, retorna null!

Ok! Conseguimos recuperar o nosso Rigidbody... agora falta empurrar! E para empurrar a gente conversou que utilizaríamos a entrada do teclado como método para tal. Os eixos *Vertical* e *Horizontal* para ser mais preciso. E aí, na sequência, precisamos apenas adicionar a física... mas aí tem um detalhe!

Anteriormente falamos sobre com o método Update é bom para receber inputs, correto? Mas para lidar com física, nem sempre ele é o melhor caminho. Não vamos entrar em muitos detalhes agora, uma vez que essa aula já está bem densa, mas fique sabendo que há um irmão muito próximo dele chamado de FixedUpdate que consegue lidar muito bem com os dois! O FixedUpdate não é chamado exatamente a cada frame, mas é chamado em um intervalo fixo que permite que você trabalhe ainda com fluidez respeitando bem a parte física. Então vamos pegar os nossos comandos nesse método! O código modificado fica como visto na **Listagem 3**.

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

public class PlayerController : MonoBehaviour {

    private Rigidbody2D rigidBody;

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

    //Chamado em um intervalo fixo, independente do FrameRate
    void FixedUpdate()
    {
        float moverV = Input.GetAxis ("Vertical");
        float moverH = Input.GetAxis ("Horizontal");
    }
}
```

Listagem 3 - Código para receber os valores dos eixos vertical e horizontal.

Veja que nesse passo alteramos a nossa função Update para FixedUpdate e também adicionamos duas novas variáveis moverV e moverH que guardam os valores dos eixos *Vertical* e *Horizontal* que são obtidos através da função que já havíamos visto Input.GetAxis("EIXO"). Como esses eixos são padrão do Unity, nem precisamos nos preocupar em fazer as configurações de project settings, como havíamos discutido anteriormente.

Agora que temos os valores da entrada do usuário, precisamos apenas de mais um passo! Vamos empurrar o nosso amigo robô para que ele se mova por aí! Para fazer isso, adicionaremos uma força, na direção em que o robô deve se mover, diretamente ao corpo rígido. Para que essa força seja adicionada, utilizaremos um Vector2. Esse tipo representa um vetor de duas posições, X e Y, ideal para representar coordenadas. Veremos objetos desse tipo muitas vezes ao longo de nossas aulas, já que estamos trabalhando em 2D e estamos sempre lidando com objetos que necessitam ter os seus valores em eixos alterados. Vamos então criar o nosso vetor de movimento e adicioná-lo como força ao nosso robô, como visto na **Listagem 4**!

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

public class PlayerController : MonoBehaviour {

    private Rigidbody2D rigidBody;

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

    //Chamado em um intervalo fixo, independente do FrameRate
    void FixedUpdate()
    {
        float moverV = Input.GetAxis ("Vertical");
        float moverH = Input.GetAxis ("Horizontal");

        Vector2 movimento = new Vector2 (moverH, moverV);

        rigidBody.AddForce (movimento);
    }
}
```

Listagem 4 - Adicionando a força ao nosso robô para fazê-lo mover na direção do Input do usuário.

E pronto! Com isso, finalmente o nosso robô está pronto para começar a se mover de acordo com o Input do usuário! Confere o teu script, vê se tá tudo certo, olha se o RigidBody 2D tá adicionado corretamente, verifica se a gravidade está em 0 e aperta o play!

ESTÁ VIVOOOO!!! Ele se move!! \o/

Mas está bem lento, né?! Beeeem. Leeeento.

Que tal corrigirmos esse detalhe?! Eu prefiro que possamos deixá-lo mais rápido em seus movimentos e que possamos fazer isso sem precisar alterar muito o nosso código... e o Unity tem uma facilidade muito legal para isso! É possível, no Unity, criar propriedades para o seu script que aparecem da mesma maneira na interface como as propriedades de outros componentes. Sério! Não acredita?! Vamos ver a **Listagem 5**, então!

```
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 velocidade;
10
11    // Use this for initialization
12    void Start () {
13       rigidBody = GetComponent<Rigidbody2D> ();
14    }
15
16    //Chamado em um intervalo fixo, independente do FrameRate
17    void FixedUpdate()
18    {
19       float moverV = Input.GetAxis ("Vertical");
20       float moverH = Input.GetAxis ("Horizontal");
21
22       Vector2 movimento = new Vector2 (moverH, moverV);
23
24       rigidBody.AddForce (movimento * velocidade);
25    }
26 }
```

Listagem 5 - Criando uma variável public para editar através da interface gráfica.

Adicionamos agora uma variável velocidade, que faz com que o nosso personagem possa se mover mais rápido, de acordo com a direção do movimento (multiplicamos o movimento pela velocidade na hora de adicionar a força, na linha 24). Essa variável é, diferentemente do Rigidbody 2D, pública! E com isso, e apenas isso, ela agora faz parte da interface e pode ser editada diretamente no editor, até mesmo durante o Play Mode! Lembre-se que quando editamos uma variável no Play Mode ela volta ao valor inicial quando saímos dele. Cuidado para não perder nenhum trabalho nisso!

Outra coisa interessante a se perceber é que, como estamos lidando diretamente com o Rigidbody e a física envolvida, alterar a massa do objeto altera, diretamente, a velocidade com a qual o objeto irá acelerar com a força adicionada. Quanto menor a massa do objeto, mantendo a força constante, maior a aceleração, correto? Teve um cara aí que já falou sobre isso! Procurem! :-P

E com isso concluímos a nossa terceira aula! O nosso personagem que não se movia agora se move pra tudo que é canto! Ele ainda não bate em nada, no entanto... Toda a física que tem nele está funcionando, mas ainda não está 100%. E é justo isso que veremos na aula que vem! Vamos aprender a trabalhar com colisores, entender um pouco mais sobre toda a física envolvida no Unity e melhorar mais um pouquinho o nosso projeto! Devagar e sempre, chegaremos lá! :D

Até! \o

Leitura Complementar

Manual do Unity sobre Input: https://docs.unity3d.com/Manual/Input.html

Manual do Unity sobre Scripting: https://docs.unity3d.com/Manual/ScriptingSection.html

Referência para a classe Input do Unity: https://docs.unity3d.com/ScriptReference/Input.html

Vídeo interessante sobre o GetAxis e a definição de eixos no Unity: https://www.youtube.com/watch?v=XZAxgH1dqXw

Resumo

Na aula de hoje vimos diferentes aspectos relacionados à entrada de dados do usuário e como isso acontece em jogos desenvolvidos no Unity. Começamos a aula entendendo melhor a importância do **Input** e como devemos torná-lo sempre **preciso** e **instantâneo** para garantir uma melhor interação do usuário com o jogo.

Na sequência vimos com o Unity trabalha com o conceito de eixos para lidar com Inputs de eixos e também de ações simples, disfarçando essas como eixos para que seja possível **configurar** as teclas responsáveis pelas ações do usuário. Em seguida, vimos as funções **GetButtonDown()** e **GetAxis()** que são utilizadas para receber e lidar com os valores que vêm dos eixos citados.

Adiante, criamos o nosso primeiro script em C# e entendemos bem como funciona cada uma de suas partes, principalmente com as funções **Start, Update** e **FixedUpdate**. Entendemos também como podemos criar variáveis e utilizar métodos predefinidos para buscar informações de **Input** e com elas **alterar o valor de propriedades de componentes físicos, como o Rigidbody 2D**. No final da aula ainda vimos como é possível utilizar variáveis do tipo Public para que consigamos alterar valores do script diretamente do editor do Unity.

Foi uma aula bem interessante e recheda de assuntos importantes! Só o fato de termos lidado com o nosso primeiro código em C#, com o nosso primeiro script, já é uma coisa bem interessante! Espero que vocês tenham gostado tanto quanto eu! Foi massa :D

E para quem quer dar uma conferida no projeto concluído da aula 3, pode baixar nesse link aqui!

Autoavaliação

1. Defina o que é e qual a importância do Input.
2. É possível ter um jogo sem que haja Input?
3. Qual a taxa de atualização interessante para o Input?
4. Como o Unity permite que o programador utilize nomes predefinidos para o Input de maneira a independer das configurações escolhidas pelo usuário?

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.