

UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA
COLEGIADO DO CURSO DE ENG. DE
COMPUTAÇÃO



Romualdo Oliveira Santos Filho

Tutorial de Criação de Jogo Digital Com Unity Engine

Orientador: Prof. Dr. Wagner Luiz Alves de Oliveira

**Salvador - BA - Brasil
Fevereiro de 2025**

Romualdo Oliveira Santos Filho

Tutorial de Criação de Jogo Digital Com Unity Engine

Trabalho de Conclusão de Curso
apresentado ao Curso de Graduação
em Engenharia de Computação da
Universidade Federal da Bahia como
parte dos requisitos para a obtenção do
grau de Engenheiro da Computação.

Orientador: Prof. Dr. Wagner Luiz Alves de Oliveira

Salvador - BA - Brasil

Fevereiro de 2025

Romualdo Oliveira santos Filho

Tutorial de Criação de Jogo Digital

Com Unity Engine

Trabalho de Conclusão de Curso
apresentado ao Curso de Graduação
em Engenharia de Computação da
Universidade Federal da Bahia como
parte dos requisitos para a obtenção do
grau de Engenheiro da Computação.

Trabalho aprovado. Salvador - BA – Brasil. Fevereiro de 2025:

Orientador: Prof. Dr. Wagner Luiz Alves de Oliveira

**Salvador - BA - Brasil
Fevereiro de 2025**

*Este trabalho é todo dedicado aos meus pais, pois é graças ao seu
esforço que hoje posso concluir o meu curso.*

Agradecimentos

Em primeiro lugar, a Deus, que fez com que meus objetivos fossem alcançados, durante todos os meus anos de estudos. Ao professor Wagner Luiz Alves de Oliveira, por ter sido meu orientador e ter desempenhado tal função com dedicação e amizade. Ao professor Jes de Jesus Fiais Cerqueira por ter me proporcionado a oportunidade de estagiar no Departamento de Engenharia Elétrica e da Computação. Ao meu amigo Caio Alexandrino Ribeiro, por ter me proporcionado as artes presentes neste tutorial. Ao meu amigo Herbert Barros Coutinho que me ajudou com revisão textual e formatação. Ao meu amigo e colega de curso André Paiva por me ajudar também com a formatação. Aos meus colegas do Coletivo de desenvolvedores de jogos Bahia Indie Devs por me auxiliar a encontrar fontes confiáveis e à minha irmã Carlla Barbosa Lima Fonseca Santos pelo apoio psicológico e emocional durante essa jornada.

“A criatividade não é
questão de talento,
mas sim de atitude.”

(Jenova Chen)

Resumo

A indústria de jogos digitais está em expansão contínua, com um faturamento global de 184,3 bilhões de dólares em 2024 e 251,6 milhões de dólares no Brasil em 2022. As Game Engines são fundamentais nesse setor, pois simplificam e aceleram o desenvolvimento de jogos ao oferecerem sistemas de física e game loop prontos.

Este TCC oferece um guia para iniciantes sobre o uso da Unity, uma engine de jogos multiplataforma lançada em 2005 e comercializada em 2006 pela Unity Technologies. Originalmente utilizada para jogos web e móveis simples, a Unity evoluiu para suportar o desenvolvimento de jogos complexos e de alta qualidade em várias plataformas, incluindo Windows, Linux, MacOS, consoles de jogos, dispositivos móveis, realidade virtual (VR) e realidade aumentada (AR).

O tutorial cobre desde a instalação e configuração do ambiente Unity até a criação de jogos completos, explorando funcionalidades como editor visual, suporte para jogos 2D, simulação física, inteligência artificial e rede multijogador. Com atualizações frequentes, a Unity se tornou uma das game engines mais populares do mundo, com uma comunidade ativa de desenvolvedores, sendo uma boa opção de ferramenta para quem deseja entrar na indústria de desenvolvimento de jogos.

Palavras-chave: Tutorial Unity, Jogos 2D e 3D, Instalação e Configuração, Comunidade de Desenvolvedores, Indústria de Jogos Digitais e Game Engines.

Abstract

The digital games industry is constantly expanding, with global sales of 184 billion dollars in 2023 and 251.6 million dollars in Brazil in 2022. Game Engines are fundamental in this sector, as they simplify and speed up game development by offering ready-made physics and game loop systems.

This TCC offers a beginner's guide to using Unity, a multiplatform game engine launched in 2005 and commercialized in 2006 by Unity Technologies. Originally used for simple web and mobile games, Unity has evolved to support the development of complex, high-quality games on various platforms, including Windows, Linux, MacOS, gaming consoles, mobile devices, virtual reality (VR) and augmented reality (AR).

The tutorial covers everything from installing and configuring the Unity environment to creating complete games, exploring features such as a visual editor, support for 2D games, physics simulation, artificial intelligence and multiplayer networking. With frequent updates, Unity has become one of the most popular game engines in the world, with an active community of developers, making it a good tool for anyone wishing to enter the game development industry.

Keywords: Unity Tutorial, 2D and 3D Games, Installation and Configuration, Developer Community, Digital Games Industry and Game Engines.

Sumário

Glossário	1
Introdução	2
Ambiente Unity	2
Resultados	3
Conclusão	3
Objetivo Geral	3
Objetivo Específico	3
Prefácio	3
1. Baixando e instalando a Unity	6
2. Criando um projeto	11
3. Conhecendo a interface da Unity	13
4. Trabalhando com Game Objects	21
5. Trabalhando com Scripts	30
6. Prefabs	53
7. Motor de Física	59
8. Interface de Usuário	72
9. Animações	89
10. Áudio	95
11. Geração de Executável	105
12. Encerramento	107
Referências	108

GLOSSÁRIO

Game Engine: Software que fornece um conjunto de ferramentas e frameworks para o desenvolvimento de jogos. Ela simplifica a criação de jogos ao fornecer sistemas prontos para gráficos, física, som, inteligência artificial, entrada do usuário e muito mais.

Game Loop: Núcleo do funcionamento de um jogo, sendo responsável por manter o jogo rodando e atualizado. Ele controla a atualização dos elementos do jogo, processa entradas do usuário e renderiza as imagens na tela.

Renderizar: Processo de gerar imagens a partir de dados digitais. No contexto de jogos, significa converter modelos 3D, texturas, luzes e efeitos em quadros visíveis na tela.

Realidade Virtual(VR): Tecnologia que cria um ambiente digital imersivo, permitindo que o usuário interaja com um mundo simulado como se estivesse fisicamente presente nele. Isso é possível por meio de dispositivos como óculos VR, fones de ouvido e controladores de movimento.

Realidade Aumentada(AR): Tecnologia que sobrepõe elementos digitais (imagens, sons, objetos 3D) ao mundo real em tempo real. Diferente da Realidade Virtual (VR), que cria um ambiente completamente virtual, a AR combina o digital com o físico, permitindo interação direta com o ambiente ao redor.

Console de Jogos: Dispositivo eletrônico projetado para rodar jogos digitais, oferecendo uma experiência otimizada para entretenimento interativo. Diferente dos computadores, os consoles são projetados exclusivamente para jogos, com hardware e software otimizados para desempenho e facilidade de uso.

Rede Multijogador: Sistema que permite que jogadores se conectem e interajam em um ambiente virtual compartilhado, seja localmente (LAN) ou pela internet.

Introdução

A indústria de jogos digitais está em constante crescente. Segundo último levantamento da gamesindustry.biz, acessível através do link <https://www.gamesindustry.biz/gamesindustrybiz-presents-the-year-in-numbers-2024>, o mercado de jogos digitais movimentou 184.3 bilhões de dólares em 2024 e segundo relatório da Abragames, acessível através do link https://www.abragames.org/uploads/5/6/8/0/56805537/2023_relat%C3%B3rio_final_v4.3.2_-ptbr.pdf, a indústria brasileira de jogos digitais movimentou cerca de 251.6 milhões de dólares em 2022. Uma parte importante dessa indústria são as chamadas Game Engines, softwares que auxiliam e aceleram o processo de desenvolvimento de jogos por possuírem sistema de física e game loop prontos.

Ambiente Unity

O motor de jogos Unity é um mecanismo de jogo multiplataforma lançado inicialmente em 2005 como uma ferramenta interna para a empresa dinamarquesa Over the edge e lançada oficialmente como uma ferramenta comercial em 2006 pela Unity Technologies. Nos primeiros anos, Unity era usado principalmente para criar jogos para navegadores de internet e jogos simples para dispositivos móveis. No entanto, à medida que o motor evoluiu e novas funcionalidades foram adicionadas, tornou-se cada vez mais popular para o desenvolvimento de jogos complexos e de alta qualidade para uma variedade de plataformas, incluindo Windows, Linux, MacOS, consoles de jogos, dispositivos móveis android e ios, realidade virtual (VR) e realidade aumentada (AR). Ao longo dos anos, Unity passou por diversas atualizações e melhorias importantes, introduzindo novos recursos como editor visual, suporte para jogos 2D, simulação física, inteligência artificial e rede multijogador. Hoje, a Unity é um dos motores de jogos mais populares do mundo, com uma comunidade grande e ativa de desenvolvedores e usuários.

Resultados

Foi produzido um tutorial ensinando a criar o jogo Pong no ambiente Unity. Neste tutorial é ensinado sobre os princípios básicos do ambiente, suas janelas, GameObjects, Prefabs, Script, Motor de física 2D, Interfaces, Animações, Áudio e Deploy.

Conclusão

Utilizando-se de minha experiência prévia, desenvolvi este tutorial e espero que ele seja útil para outras pessoas que queiram começar a desenvolver jogos digitais.

Objetivo Geral

Propagar o conhecimento a cerca do Motor de Jogos Unity para quem possui pouco ou nenhum conhecimento sobre programação ou Unity.

Objetivo Específico

Ensinar para quem possui pouco ou nenhum conhecimento sobre programação ou Unity sobre o processo de criação de um jogo 2D na Unity o que envolve ensinar sobre a instalação da Unity, sobre o processo de criação de um projeto na Unity, sobre a interface da Unity, sobre a criação e escrita de scripts na Unity, sobre o motor de física da Unity, sobre a criação de Interfaces de Usuário na Unity, sobre animação 2D na unity, sobre o sistema de áudio da unity e sobre a criação de um arquivo executável na Unity.

Prefácio

O motor de jogo Unity é uma escolha incrivelmente poderosa e popular para desenvolvedores de jogos profissionais e amadores. Este texto foi escrito com o objetivo de ensinar os conceitos básicos do Unity utilizando como exemplo o

clássico jogo Pong (1972). Este texto é indicado para aqueles que possuem pouco ou nenhum conhecimento sobre o motor de jogos Unity.

Introdução à Unity

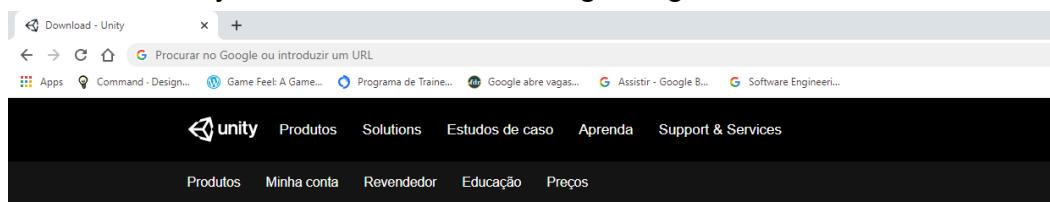
1. Baixando e instalando a Unity

1.1 Instalando a Unity no Windows

1. Acesse o link abaixo para criar um Unity ID

<https://id.unity.com/en/conversations/6b7c777d-0e43-4c5a-a279-ede62f3499d4001f?view=register>

2. Acesse o link abaixo <https://unity3d.com/pt/get-unity/download> e clique em Baixar Unity Hub como mostra a imagem figura 1.1.



Download Unity

Seja bem-vindo! Você está aqui porque deseja descarregar o Unity, a plataforma de desenvolvimento mais popular do mundo, usado para criar jogos multiplataforma e experiências interativas em 2D e 3D.

Antes disso, escolha a versão do Unity certa para você.

[Escolha o seu Unity + download](#) [Baixe Unity Hub](#)

[Mais informações sobre o novo Unity Hub aqui.](#)

Descarregar Unity Beta

Obtenha acesso antecipado aos nossos recursos mais recentes, e ajude-nos a melhorar a qualidade, dando-nos a sua valiosa opinião.

[Descarregar Beta](#)

Figura 1.1 Página de downloads da Unity.

3. Siga os passos do instalador do Unity Hub
4. Abra o Unity Hub, selecione a aba Installs no canto direito e clique em ADD, como mostra a figura 1.2.

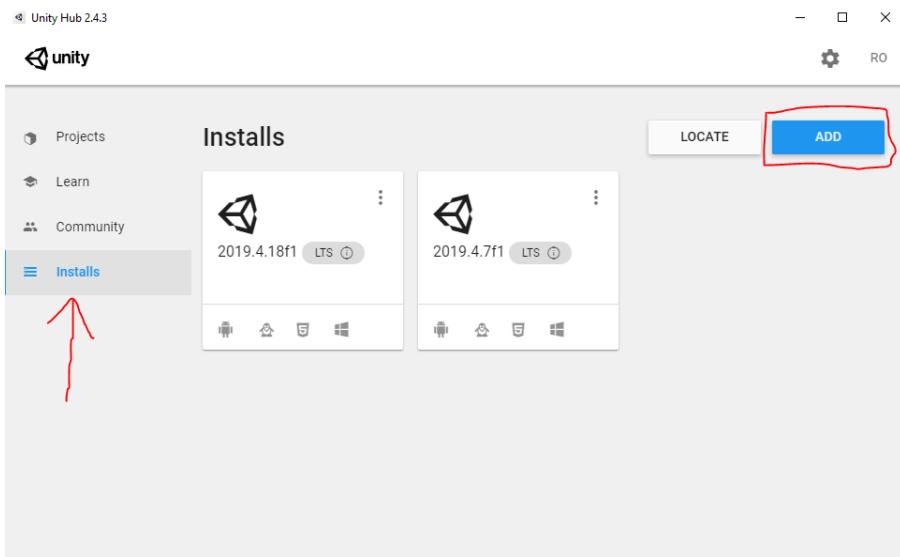


Figura 1.2 Unity Hub.

5. Clique em *instal* do lado da versão mais recente

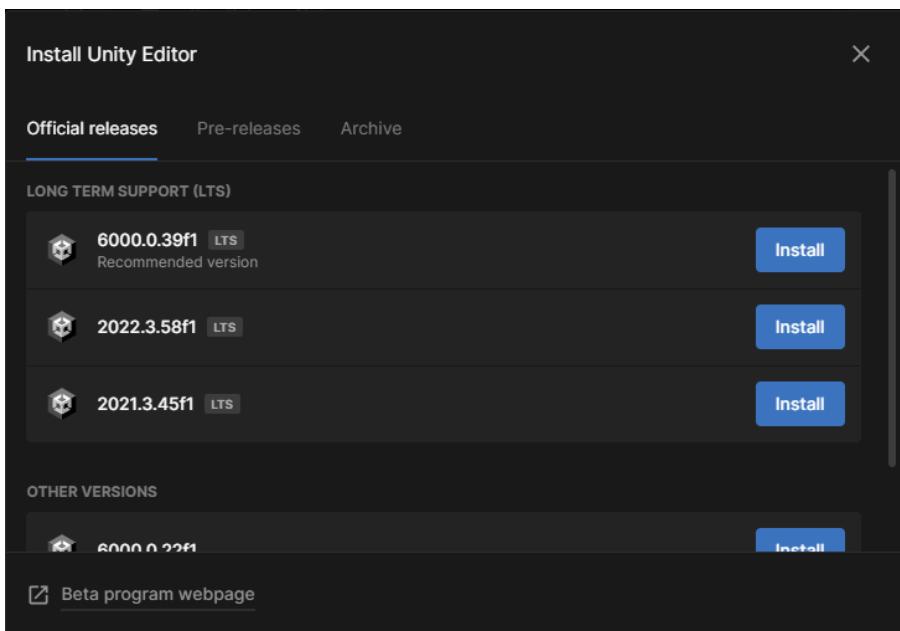


Figura 1.3 Seleção de versão.

6. Na próxima tela selecione microsoft visual studio community e as plataformas de desenvolvimento (para esse tutorial instale o Windows Build Support e Linux Build Support)

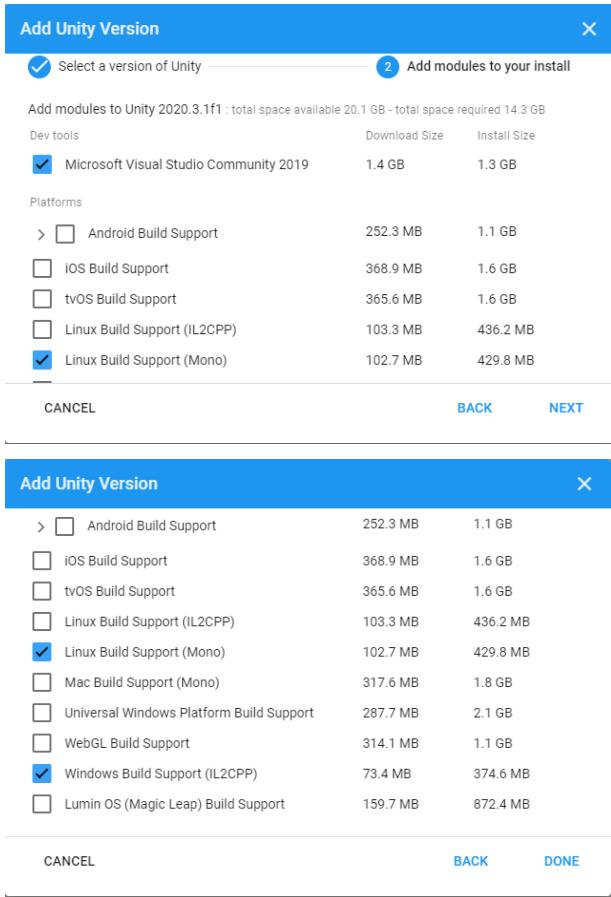


Figura Figura 1.4 Seleção de Módulos.

7. Clique em *Done* e aguarde a instalação

Instalando o Unity Hub no Linux

1. Acesse o link abaixo para criar um Unity ID

<https://id.unity.com/en/conversations/6b7c777d-0e43-4c5a-a279-ede62f3499d4001f?view=register>

2. Acesse <https://docs.unity3d.com/hub/manual/InstallHub.html#install-hub-linux> para acessar as instruções de instalação do Unity Hub no linux

3. Abra um terminal como indicado na figura 1.5:

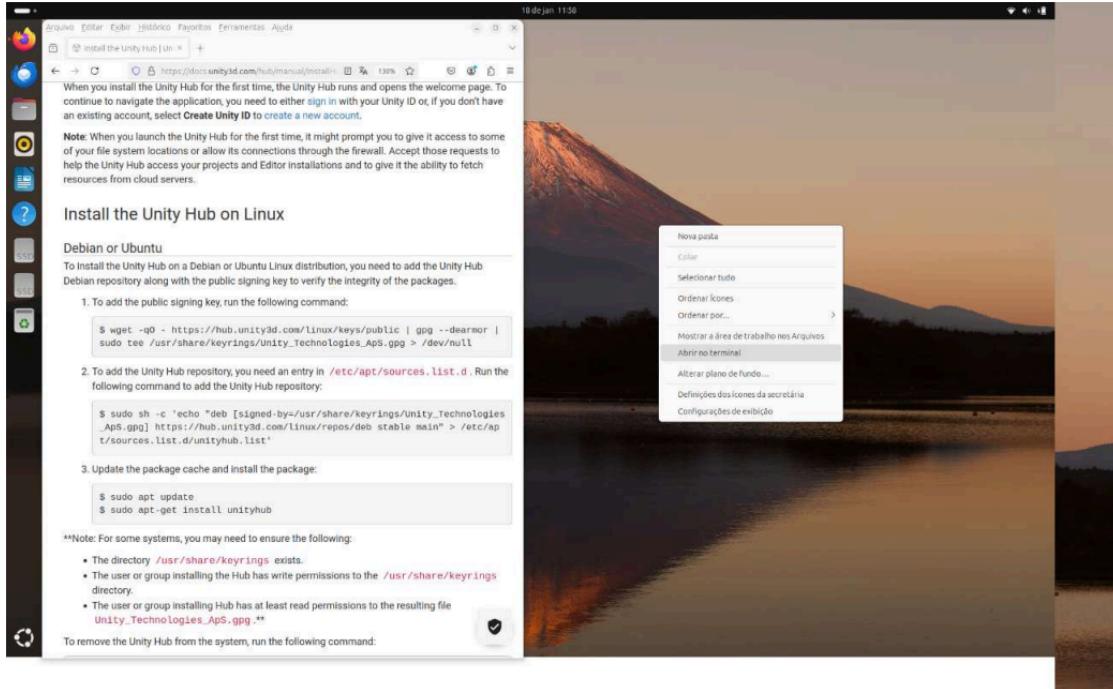


Figura Figura 1.5 Terminal.

4. Copie e cole cada um dos comandos da tela acima, um por vez (não copie o "\$"):

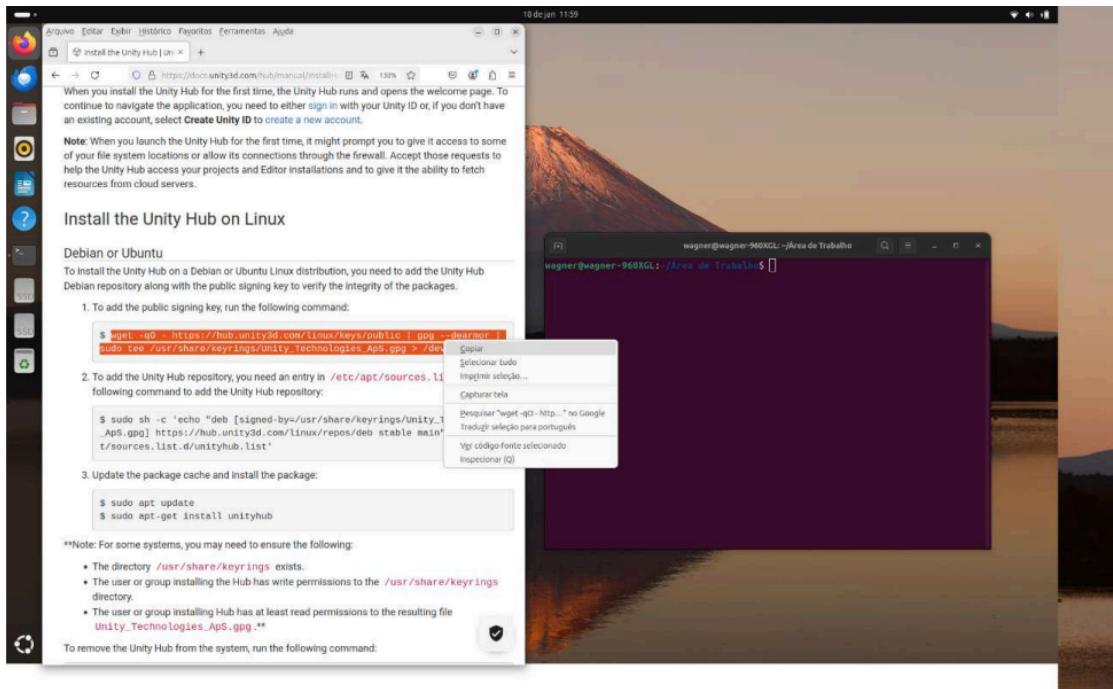


Figura Figura 1.6 Copiando comando.

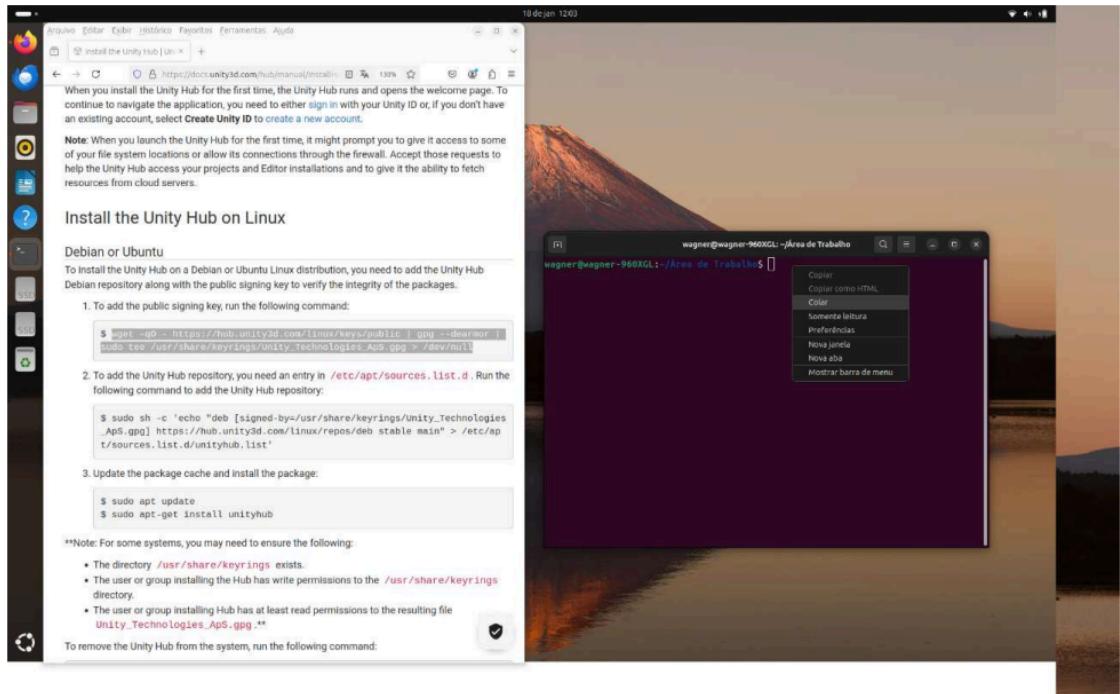


Figura Figura 1.7 Colando Comando.

5. Ao executar o primeiro comando, será solicitada a senha de root:

6. Após a instalação do pacote, digite unityhub na linha de comando para abrir o Unity Hub e siga as mesmas instruções de instalação para o Windows a partir do passo 4

2. Criando um projeto

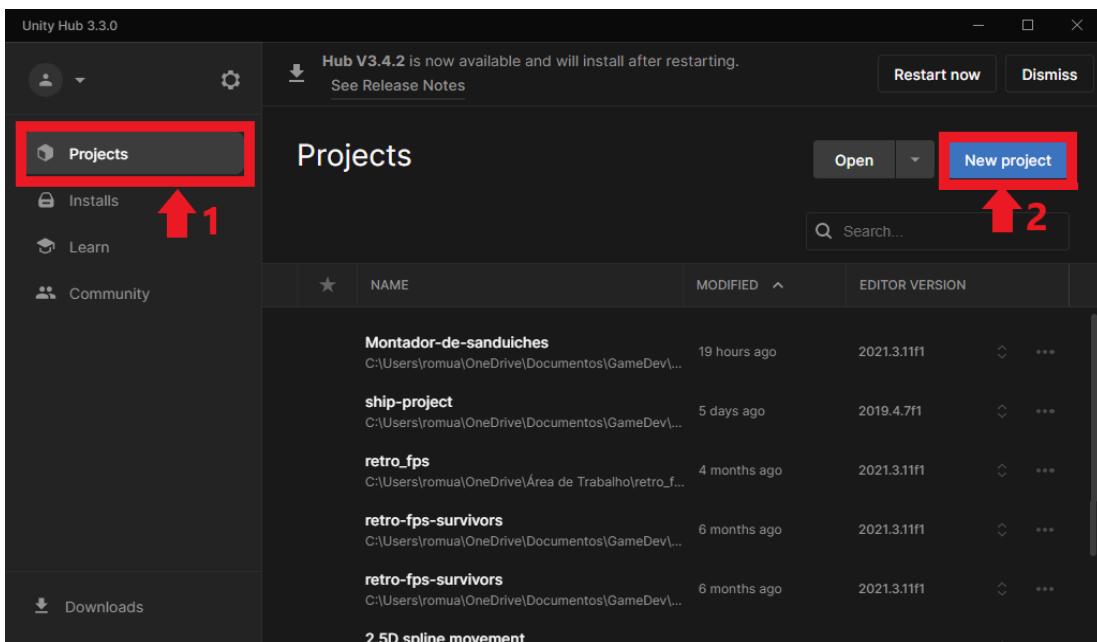


Figura 2.1 Tela de Projetos

Para criar um projeto na Unity abra o Unity Hub. No menu da esquerda clique em *Projects* (1) e depois clique em *New Project* (2).

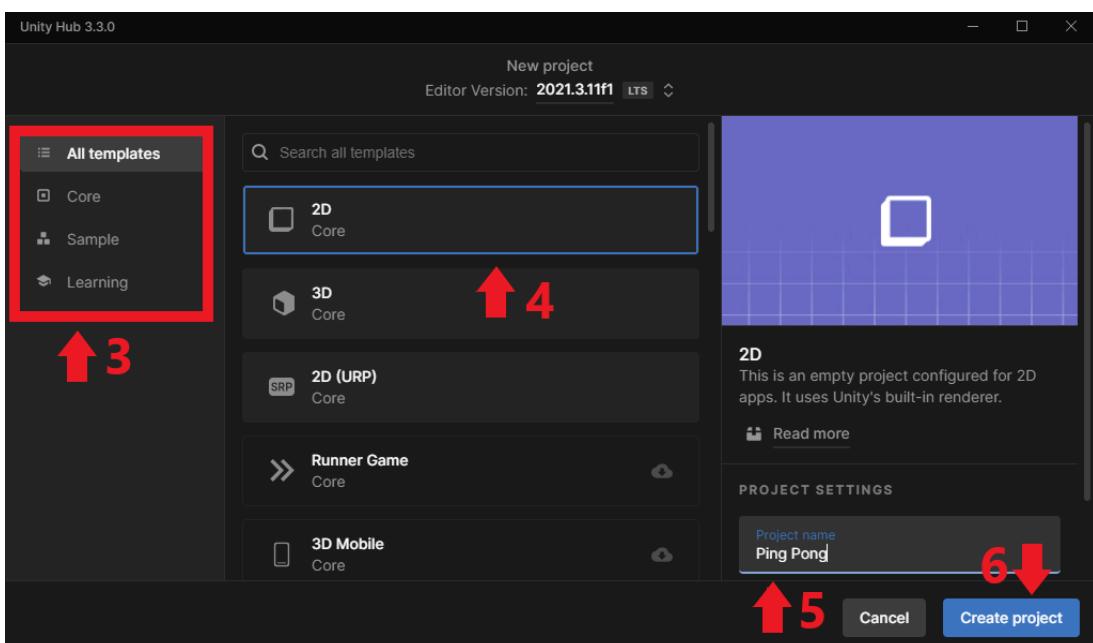


Figura 2.2 Tela de criação de novo projeto.

Na nova janela que abriu verifique que no menu da esquerda (3) esteja selecionada a opção *All Templates* ou *Core*. No menu do meio (4) podemos

selecionar um template para o nosso projeto, para esse projeto utilizaremos o template 2D. No canto inferior direito existe um espaço chamado *Project Name* (5) onde escolhemos o nome para o projeto, eu chamei de Ping Pong, mas fique à vontade para escolher um nome criativo. Após todas essas configurações clique em *Create Project* (6) e aguarde a Unity compilar o seu novo projeto.

3. Conhecendo a interface da Unity

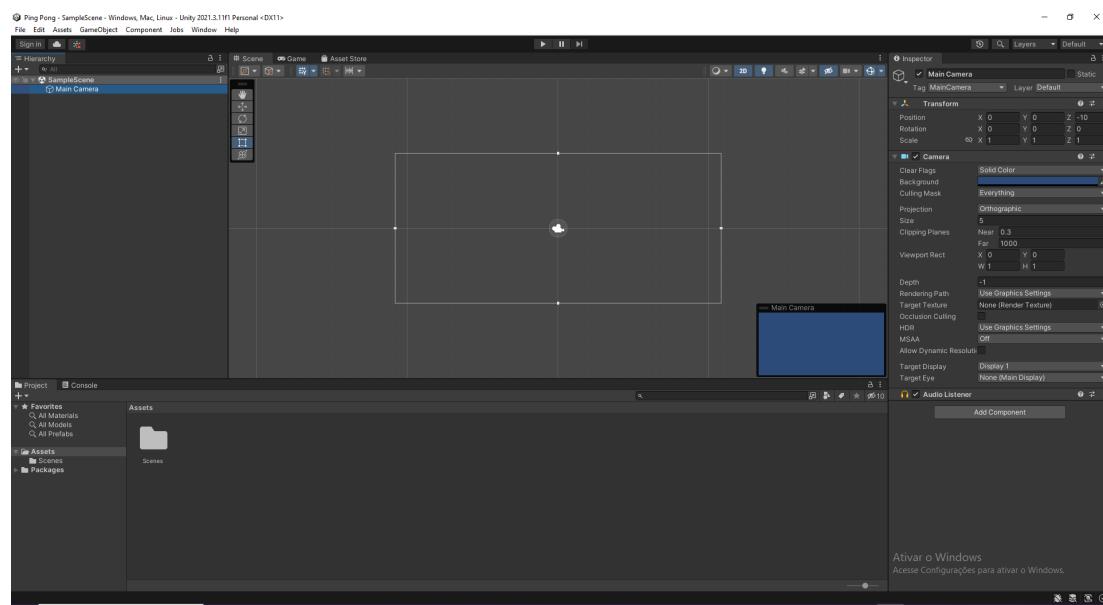


Figura 3.1 Interface padrão da Unity

A interface da Unity é dividida em pequenas janelas conhecidas como *Views* que podem ser arrastadas e realocadas de acordo com sua preferência. Vamos conhecer as principais *Views* da Unity.

3.1 Project (Projeto)

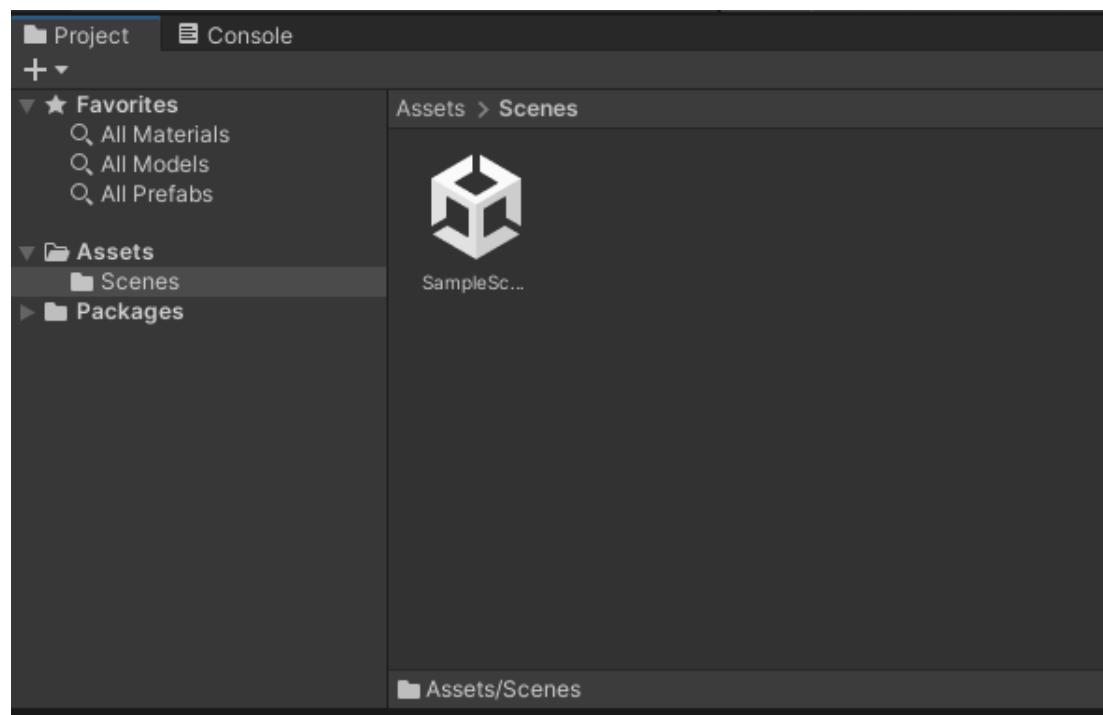


Figura 3.2 Project View.

A janela de Projeto mostra todos os *assets* (recursos como arquivos, scripts, texturas, modelos etc.) presentes no seu projeto. Você pode organizar pastas, importar novos *assets* e gerenciar recursos de maneira geral. Ao criar um projeto, você notará uma única pasta denominada *Assets*. Se você for para a pasta do seu disco rígido onde você salvou o projeto, você também encontrará uma pasta *Assets*. Isso ocorre porque o Unity reflete na janela de Projeto as pastas no disco rígido. Se você criar um arquivo ou pasta no Unity, o correspondente será criado no disco rígido dentro da pasta *Assets* (e vice-versa). Você pode mover itens na *project view* simplesmente arrastando e soltando. Isso permite que você coloque itens dentro de pastas ou reorganize seu projeto em tempo real.

Um *asset* é qualquer item que existe no disco rígido, como um arquivo em sua pasta de *Assets*. Todas as texturas, malhas, arquivos de som, scripts e assim por diante são considerados *assets*. Por outro lado, se você cria um objeto de jogo, mas ele não cria nenhum arquivo correspondente no disco rígido, nenhum *asset* foi criado.

Baixe os *Assets* do projeto no link:

https://github.com/ROMUBOY/AssetsTutorial_pong_unity

ATENÇÃO: A Unity mantém vínculos entre os vários assets associados aos projetos. Como resultado, mover ou excluir itens fora da Unity pode causar problemas. Como regra geral, é uma boa ideia fazer todo o seu gerenciamento de assets dentro da Unity.

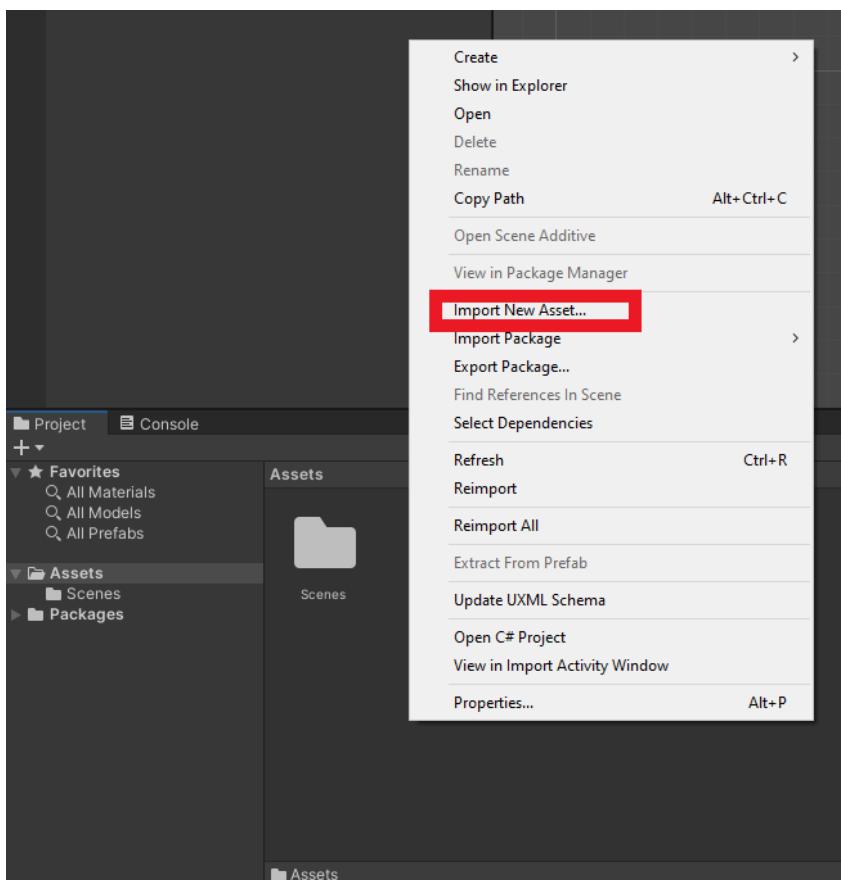


Figura 3.3 Importando novos Assets.

Ao clicar com o botão direito do mouse na região da Janela de Projeto surgirá um menu com várias opções. Clicando em Import New Asset... abrirá um explorador de arquivos que permite importar diversos arquivos para o seu projeto. Vá até a pasta onde você descompactou os arquivos do projeto, selecione todos os arquivos e clique em import.

3.2 *Hierarchy* (Hierarquia)

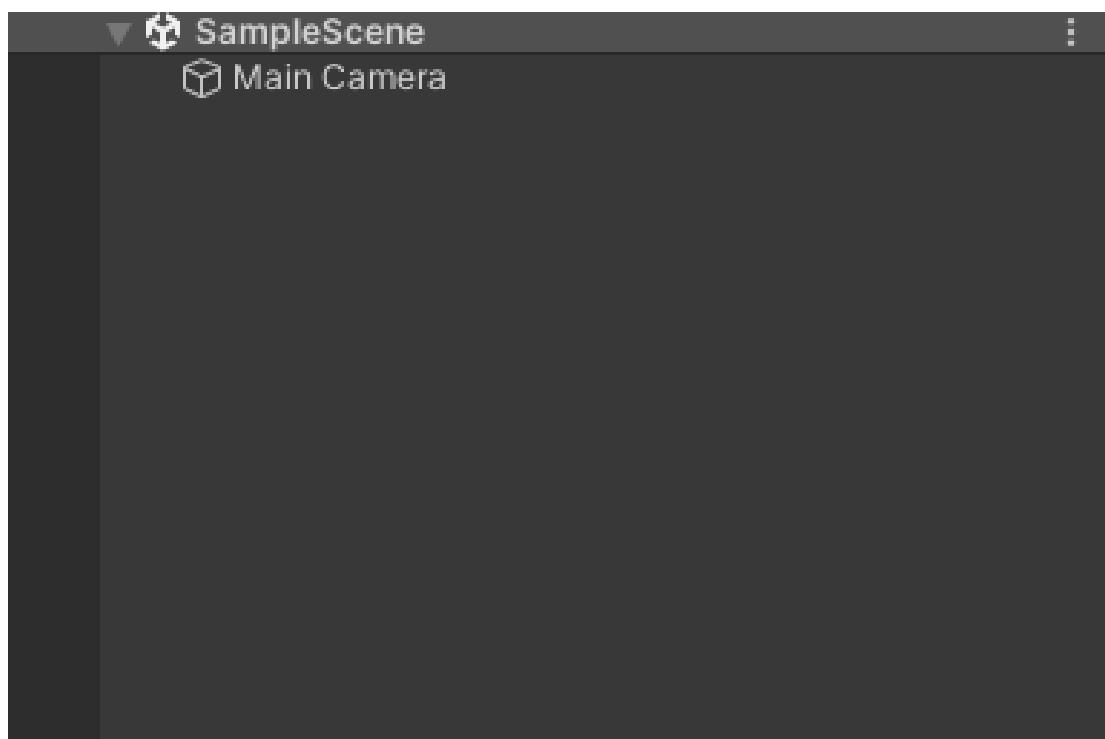


Figura 3.4 Hierarchy View.

A janela da Hierarquia exibe uma lista hierárquica de todos os objetos presentes na cena. É aqui que você pode organizar e acessar facilmente os elementos da sua cena. A Hierarquia é muito parecida com a Janela de Projeto. A diferença é que a Hierarquia mostra todos os itens na cena atual em vez de todo o projeto.

Na Unity, uma cena é um ambiente ou nível onde os elementos do seu jogo são organizados e renderizados. Ela serve como o espaço onde você constrói e combina diferentes elementos, como objetos, personagens, ambientes, luzes e efeitos, para criar a experiência interativa final. Se você estivesse construindo um jogo com uma fase de neve, uma fase de selva e este jogo tivesse um menu inicial, as duas fases seriam cenas separadas e o menu inicial seria uma terceira cena.

Ao criar um projeto 2D pela primeira vez com a Unity, você obtém a cena padrão, que possui apenas um item, a Câmera Principal. Conforme você adiciona itens à sua cena, eles aparecerão na Hierarquia. Assim como na Janela de Projeto, você pode abrir o menu clicando com o botão direito do mouse para adicionar itens para sua cena, como mostra a figura 3.5.

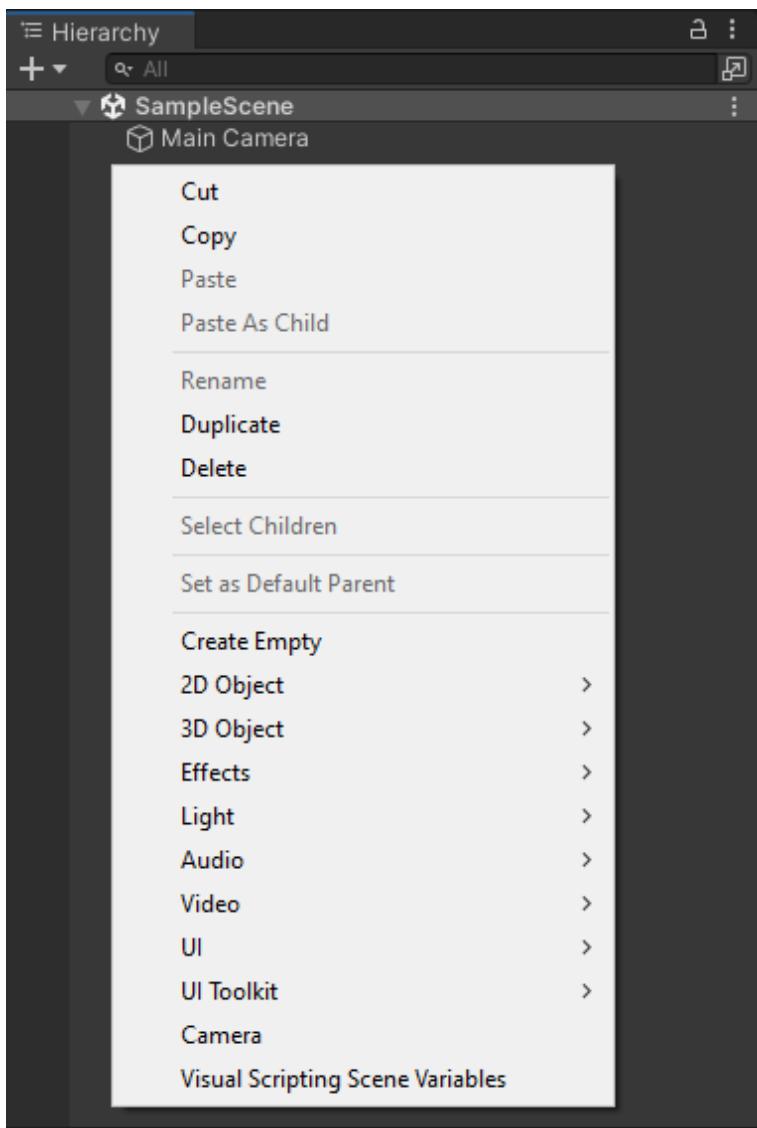


Figura 3.5 Adicionando novos elementos em uma cena.

Também é possível clicar e arrastar os itens para organizar e “aninhá-los”.



Figura 3.6 Itens aninhados.

Aninhamento é o termo para estabelecer uma relação entre dois ou mais itens da hierarquia. Na Hierarquia, clicar e arrastar um item para outro item aninhará o item arrastado sob o outro. Isso é comumente conhecido como relacionamento pai-filho. Nesse caso, o objeto no topo é o pai e quaisquer objetos abaixo dela são filhos. Você saberá quando um objeto está aninhado porque ele estará indentado. Como você verá mais tarde, aninhar objetos na Hierarquia pode afetar como eles se comportam.

3.3 Inspector (Inspetor)

A janela de inspetor permite que você veja todas as propriedades de um item atualmente selecionado. Simplesmente clique em qualquer recurso ou objeto da Janela de Projeto ou da Hierarquia e o Inspetor automaticamente exibe as informações.

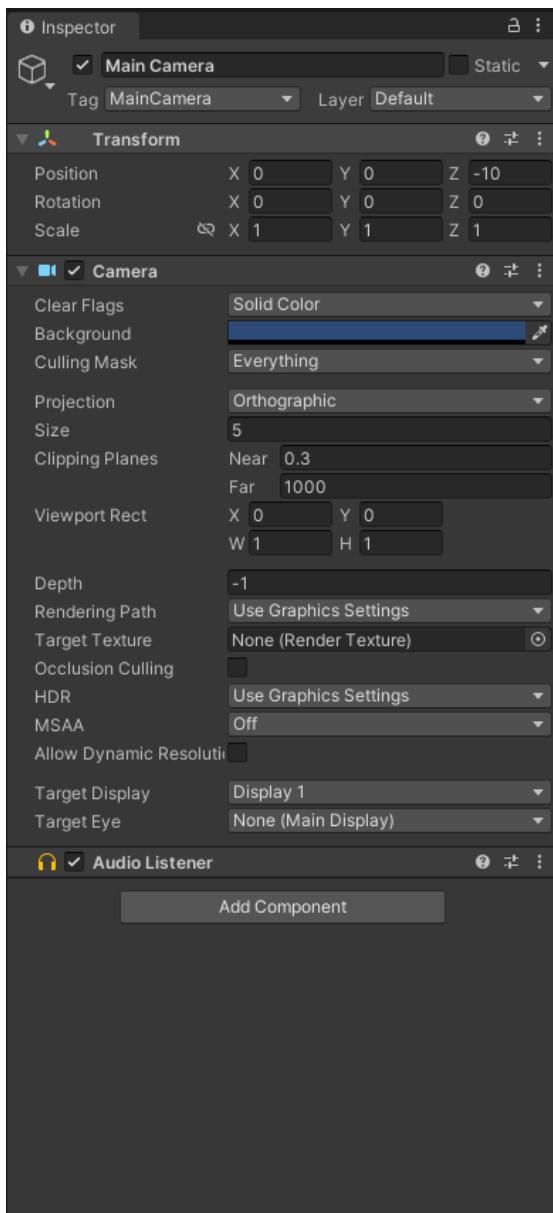


Figura 3.7 Inspector View.

Na figura 3.7 podemos ver o Inspetor depois que o objeto *Main Camera* foi selecionado na hierarquia.

Vamos detalhar algumas dessas funcionalidades:

- Se você clicar na caixa de seleção ao lado do nome do objeto, ele ficará desativado e não aparecerá no projeto.

- *DropDown Lists* (como as listas Layer ou Tag; mais sobre isso depois) são usadas para selecionar um conjunto de opções predefinidas.
- Caixas de texto, menus suspensos e controles deslizantes podem ter seus valores alterados e as alterações serão automaticamente e imediatamente refletidas na cena - mesmo se o jogo estiver em execução!
- Cada objeto do jogo atua como um contêiner para diferentes componentes (como Transform, Camera e Audio Listener na figura acima). Você pode desativar esses componentes desmarcando ou removê-los clicando com o botão direito do mouse e selecionando Remover componente.
- Os componentes podem ser adicionados clicando no botão Adicionar componente.
- *Transform* é o único componente presente em todos os Objetos e não pode ser desativado.

3.4 Scene View

A *Scene View* é a janela mais importante com a qual você trabalha porque é onde você constroi e edita o ambiente (cena) do seu jogo. Você pode posicionar objetos, ajustar iluminação, câmeras e realizar outras operações de edição.

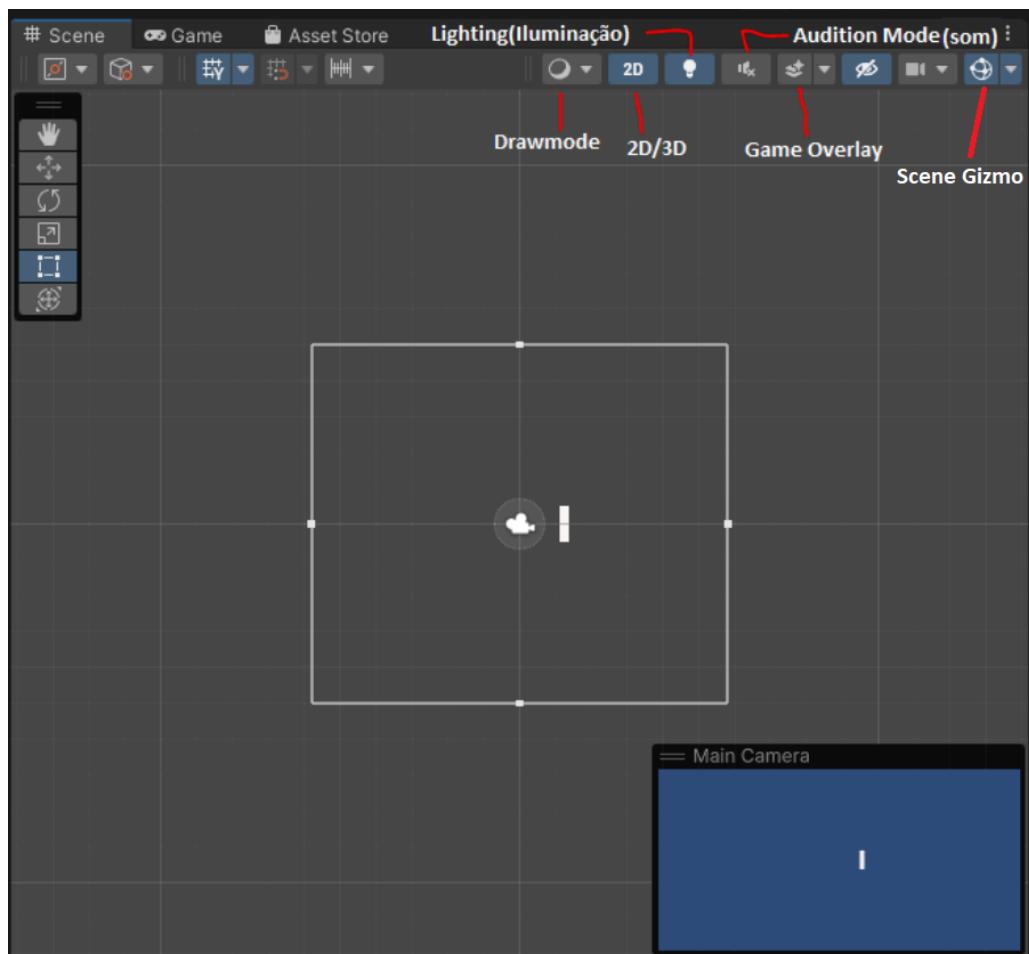


Figura 3.8 Scene View.

A seguir há uma descrição de alguns controles da barra de ferramentas da *Scene View*:

- **Drawmode (Modo de Renderização)**: Alterna entre os modos de renderização da cena (shaded, wireframe, textured etc.).
- **2D/3D Mode (Modo 2D/3D)**: Alterna entre o modo de visualização 3D e o modo de visualização 2D da *Scene View*.
- **Lighting (Iluminação)**: Abre o menu suspenso de configurações de iluminação da cena, onde você pode ajustar as configurações de iluminação em tempo real.
- **Audition mode (Modo de Som)**: Define se uma fonte de áudio funciona na *Scene View* ou não.
- **Game overlay**: determina se itens como skyboxes, neblina e outros efeitos aparecem na *Scene View*.
- Um componente que é bastante mencionado ao estudar sobre Unity é o *Scene Gizmo*. No modo 3D o *Scene Gizmo* mostra qual direção você está enfrentando no momento e para alinhar a vista da cena com um eixo. Como focaremos no desenvolvimento 2D não nos aprofundaremos no assunto.

No modo 2D é bastante simples navegar pela cena. Você pode navegar pela cena mantendo o botão direito do mouse pressionado e arrastando. Você pode alterar o zoom da cena com a roda do mouse. Se quiser focar em um game object específico é só clicar nele duas vezes na Hierarquia que a *Scene View* focará nele.

3.5 Game View (Visão de Jogo)

A *Game View* permite visualizar o jogo em execução, proporcionando uma visão mais próxima de como será a experiência do jogador. Ela exibe o que a câmera veria durante a execução do jogo.

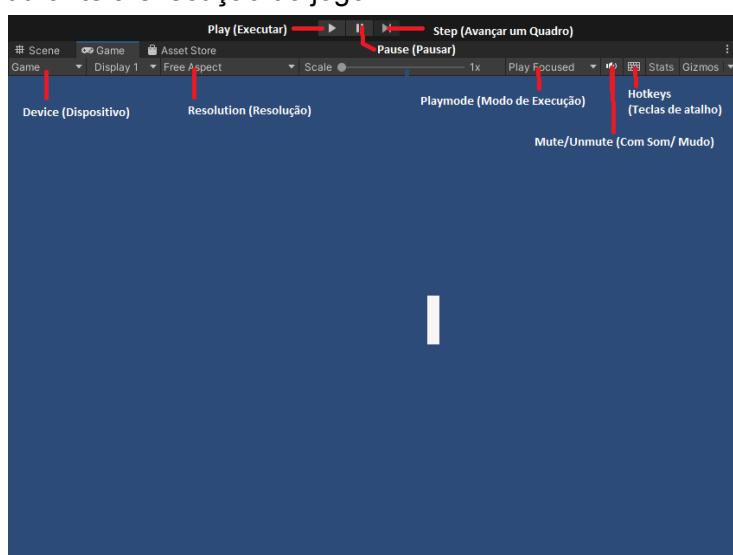


Figura 3.9 Game View.

Na Unity, a barra de ferramentas (*toolbar*) da *Game View* fornece acesso rápido a vários comandos e ferramentas úteis para visualizar e interagir com o jogo em execução. Aqui está uma descrição dos botões e comandos comuns encontrados na toolbar da *Game View*:

- **Play (Executar)**: Inicia a execução do jogo no modo de reprodução. Este botão é usado para testar o jogo e verificar o comportamento em tempo real.
- **Pause (Pausar)**: Pausa a execução do jogo durante o modo de reprodução. Isso permite examinar o estado atual do jogo e depurar problemas.
- **Step (Avançar um Quadro)**: Avança a execução do jogo um único quadro enquanto o jogo está pausado. Útil para depurar comportamentos específicos de quadros individuais.
- **Play Mode (Modo de Execução)**: Permite selecionar o modo de execução, Tela Cheia, Tela Normal e fora de foco.
- **Stats (Estatísticas)**: Exibe estatísticas de desempenho do jogo, como quadros por segundo (FPS), uso de memória e muito mais. Útil para monitorar o desempenho durante a execução do jogo.
- **Device (Dispositivo)**: Abre o menu suspenso para selecionar um dispositivo de simulação, útil para testar jogos em diferentes plataformas e tamanhos de tela.
- **Resolution (Resolução)**: Abre o menu suspenso para selecionar a resolução da janela da *Game View*. Isso pode ser útil para testar o jogo em diferentes resoluções de tela.
- **Gizmos (Gizmos)**: Abre o menu suspenso de configurações de gizmos na *Game View*, permitindo ativar ou desativar gizmos específicos durante a execução do jogo.
- **Mute/Unmute (Com Som/Mudo)**: Botão que ativa e desativa os sons da execução.
- **Hotkeys (Teclas de Atalho)**: Botão que permite desativar as teclas de atalho da Unity durante a execução.

4. Trabalhando com *Game Objects*

Na Unity, *GameObjects* são os blocos de construção fundamentais de um jogo. Eles representam todos os elementos que compõem a cena do jogo, como personagens, objetos, câmeras, luzes, efeitos especiais e muito mais.

4.1 Revisando o Sistema de Coordenadas Cartesianas

Os *GameObjects* não existem de forma aleatória na cena, todos eles são colocados em um sistema de coordenadas cartesianas. Um sistema de coordenadas cartesianas 2D (que é o nosso foco pois estamos desenvolvendo um jogo 2D) é uma estrutura matemática que permite localizar pontos em um plano bidimensional usando dois eixos perpendiculares: o eixo x (horizontal) e o eixo y (vertical).

Cada ponto no plano é identificado por um par ordenado de números (x, y), onde:

- x representa a distância horizontal do ponto até o eixo y (positiva para a direita e negativa para a esquerda).
- y representa a distância vertical do ponto até o eixo x (positiva para cima e negativa para baixo).

O ponto $(0,0)$ onde os eixos x e y se encontram é chamado de origem. É o ponto de referência inicial para medir as coordenadas de outros pontos no plano.

Veja alguns exemplos de coordenadas na figura 4.1.

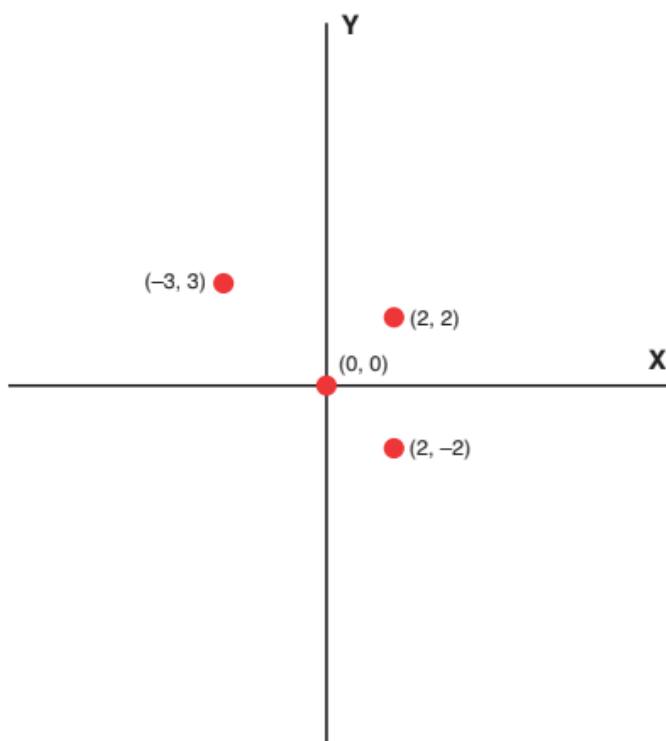


Figura 4.1 Sistema de coordenadas cartesianas.

Observação: a Unity é originalmente um motor 3D, o que significa que mesmo para jogos somente com elementos 2D esses elementos existem em um espaço 3D, portanto possuem 3 coordenadas (x, y, z) onde z é a profundidade. Como estamos focando no desenvolvimento 2D evitaremos alterar a coordenada z dos nossos *GameObjects*.

4.2 Coordenadas Globais e Locais

O que revisamos de coordenadas cartesianas até agora são equivalentes ao sistema de coordenadas globais na Unity, onde sabemos a posição de cada *GameObject* na cena. No entanto existe outro tipo de sistema de coordenadas na Unity, as coordenadas locais, que é único para cada *GameObject*, isolado de outros *GameObjects*.

Coordenadas Globais:

- As coordenadas globais de um objeto são relativas ao mundo ou à cena em que o objeto está localizado. Isso significa que a posição, rotação e escala de um objeto são medidas em relação ao ponto de origem do mundo, geralmente localizado na origem (0,0,0) do sistema de coordenadas global da cena.
- Por exemplo, se um objeto está localizado em (5,0,0) em suas coordenadas globais, isso significa que ele está posicionado 5 unidades à direita do ponto de origem global.

Coordenadas Locais:

- As coordenadas locais de um objeto são relativas ao próprio objeto. Isso significa que a posição, rotação e escala de um objeto são medidas em relação ao seu próprio centro ou origem local.
- Por exemplo, se um objeto está localizado em (0,0,0) em suas coordenadas locais, isso significa que ele está posicionado no centro de seu próprio espaço.

Veja o exemplo da figura 4.2.

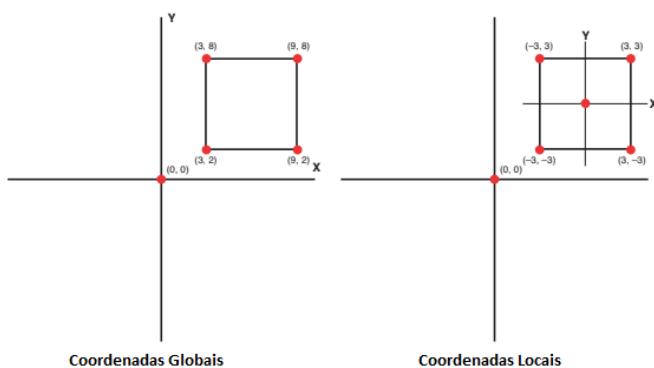


Figura 4.2 Coordenadas Locais e Globais.

4.3 Transformações (*Transforms*)

As transformações na Unity referem-se aos componentes que controlam a posição, rotação e escala de um *GameObject* dentro de uma cena. Cada *GameObject* possui um componente *Transform* associado a ele, que define sua localização e orientação no espaço tridimensional.

Você deve se lembrar que o componente *transform* é o único componente que todo *GameObject* possui. Até mesmo *GameObjects* vazios possuem o componente *transform*. Usando este componente, você pode ver a transformação atual do objeto e alterar (ou transformar) a transformação do objeto. Pode parecer confuso agora, mas é bastante simples. Você vai pegar o jeito. O transform é composto de *position* (posição), *rotation* (rotação) e *scale* (escala), e existem três métodos separados (chamados de transformações) para alterar a transform: *translation* (translação), *rotation* (rotação) e *scaling* (dimensionamento), respectivamente. Essas transformações podem ser alcançadas usando o *Inspector* (figura 4.3) ou o *transform toolbar*, acessível pela *Scene View* (figura 4.4).

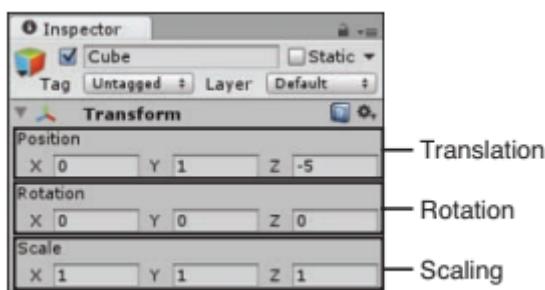


Figura 4.3 Project View.

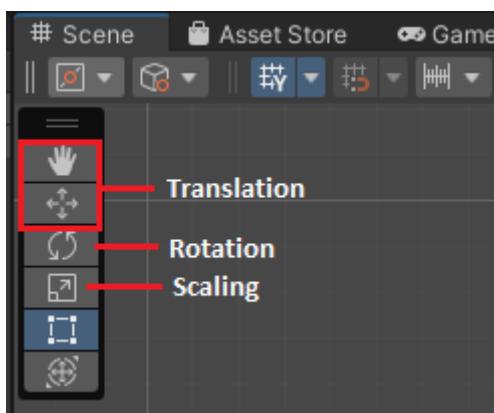


Figura 4.4 Transform Toolbar.

4.4 Position (posição)

A posição de um *GameObject* define sua localização no espaço tridimensional. É representada por um vetor tridimensional (x, y, z) que especifica a distância do *GameObject* em relação ao ponto de origem da cena.

4.5 Translation (translação)

É a ação de alterar a posição de um *GameObject* movendo-o ao longo dos eixos x, y e z. Na figura 4.5 temos um objeto em vermelho e esse mesmo objeto em azul após uma translação.

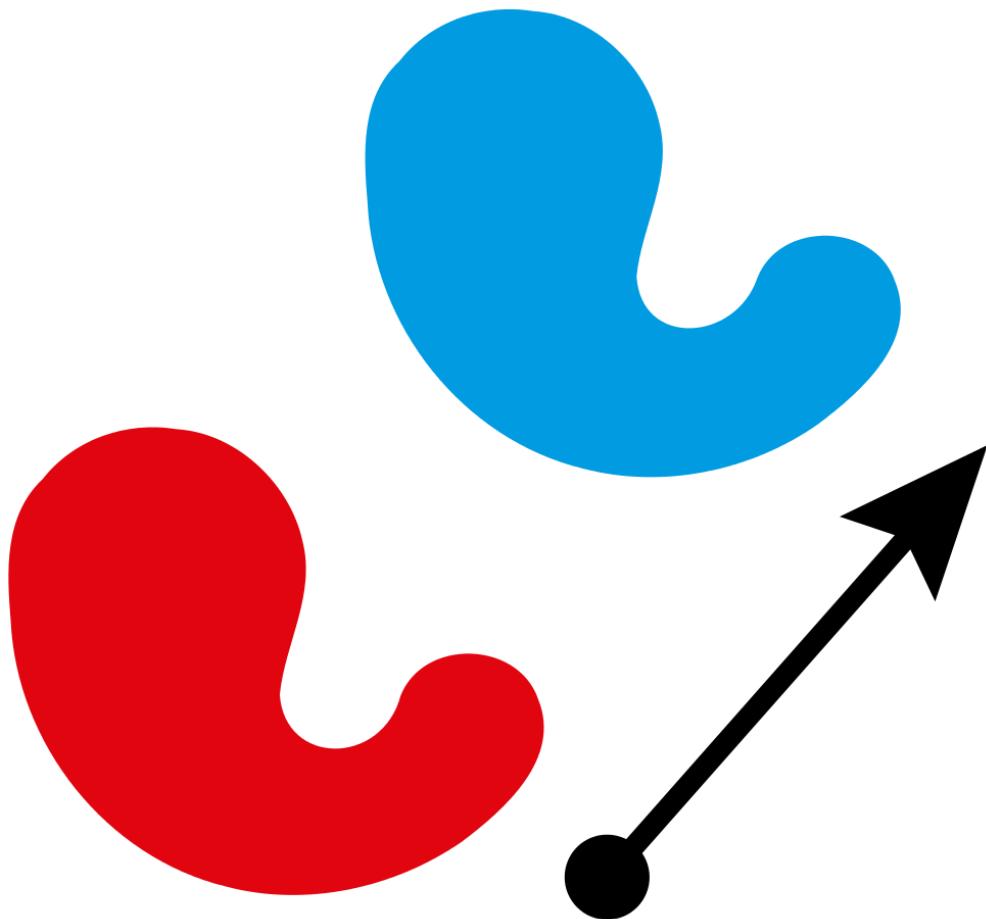


Figura 4.5 Exemplo de translação.

Ao selecionar a ferramenta *Transform* (tecla de atalho: w), você notará que na *Scene View* duas setas aparecerem apontando para longe do centro do objeto ao longo dos dois eixos (ou três se estiver no modo 3D). Estes são os *gizmos* de translação, e eles ajudam você a mover seus objetos na cena. Clicando e segurando qualquer uma destas setas, elas ficam amarelas. Então, se você mover o mouse, o objeto se moverá ao longo desse eixo. A imagem abaixo mostra a aparência dos gizmos de translação. Observe que os *gizmos* aparecem apenas na *Scene View*, se você estiver na *GameView* não os verá.

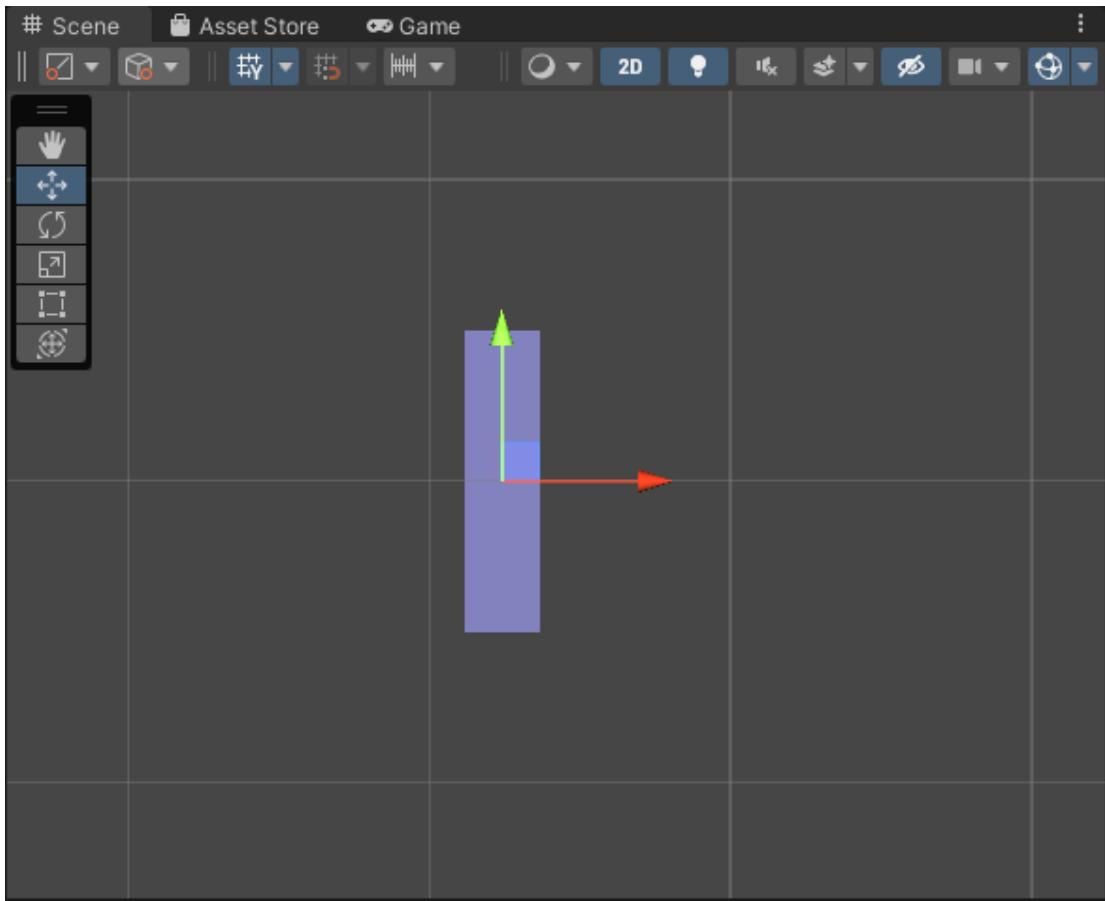


Figura 4.6 Gizmos de translação.

Utilize os *Gizmos* de translação para mover as barras para as laterais da câmera, como no jogo Pong.

4.6 *Rotation* (Rotação)

A rotação de um *GameObject* controla sua orientação no espaço tridimensional. Ao contrário das demais transformações, a rotação não é representada por vetores tridimensionais, é representada por uma estrutura matemática chamada Quatérnion. Quatérnions são um conjunto de quatro números (x, y, z, w) que representam rotações e orientações no espaço tridimensional. Os quatérnions são usados internamente pela Unity para representar rotações, pois permitem cálculos mais eficientes. Apesar de serem precisos e eficientes, os quaternions não são muito intuitivos, por esse motivo eles são apresentado no inspector através de outra representação conhecida como ângulos de Euler (*Euler angles*) que especifica a inclinação em torno dos eixos x, y e z. Euler Angles são mais intuitivos e fáceis de aprender, porém menos eficientes, motivo pelo qual a Unity utiliza os Quaternions internamente. Alterar a rotação de um *GameObject* gira-o em torno dos eixos x, y e z.

4.7 Rotacionar

Ato de alterar a rotação de um *GameObject*, girando-o em torno dos eixos. Quando um objeto gira em torno de um eixo, diz-se que ele está girando sobre esse eixo. Na figura 4.7 as linhas em azul representam um objeto e as linhas em vermelho representam esse mesmo objeto depois de rotacionar em relação ao eixo z.

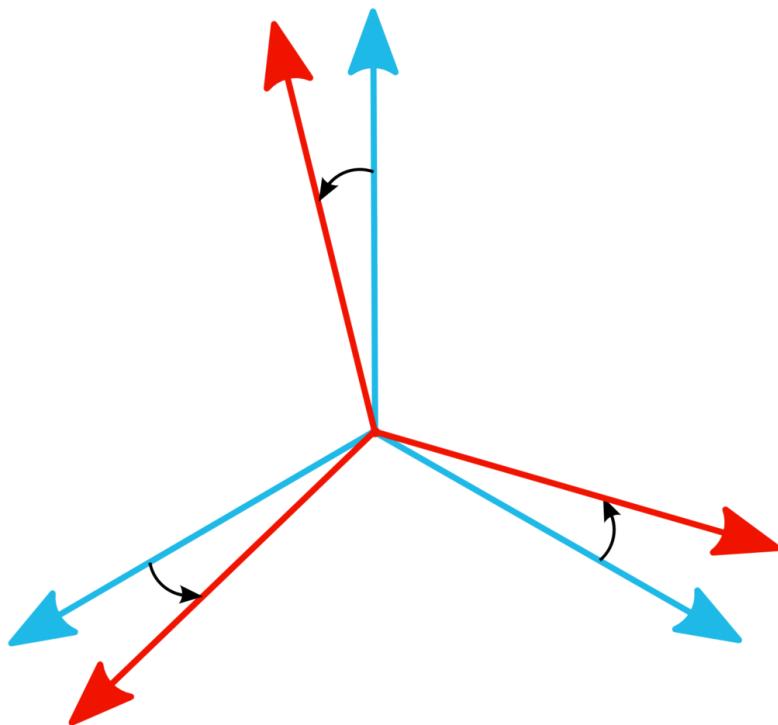


Figura 4.7 Exemplo de rotação.

Observação: ao contrário das demais transformações, a rotação em projetos 2D geralmente não é executada nos eixos x e y mas é executada no eixo z.

Selecionando a ferramenta rotation (tecla de atalho e) e selecionar um *GameObject* na hierarquia fará surgir círculos ao redor desse *gameObject* na Scene View. Esses círculos são os Gizmos de Rotação. Clicando no círculo azul e arrastando após ele ficar amarelo você rotaciona esse *GameObject* no eixo z.

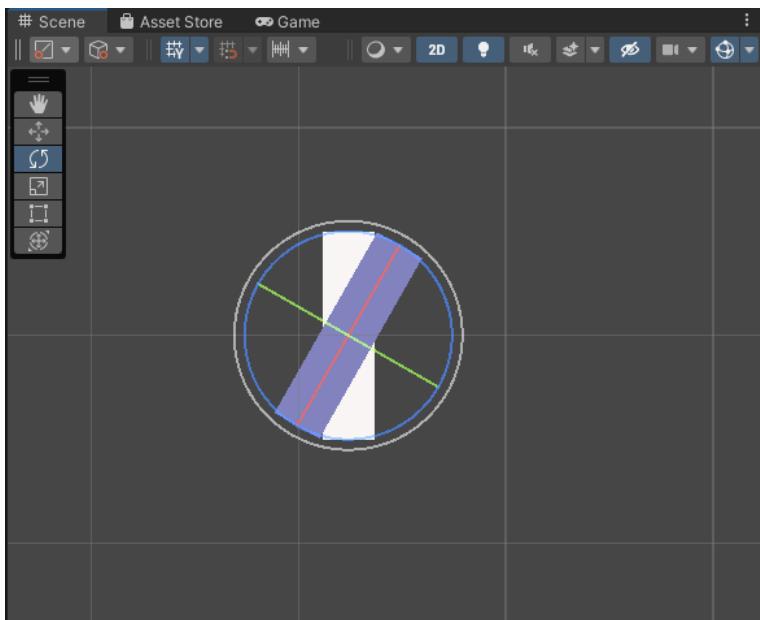


Figura 4.8 Gizmos de rotação.

Use os gizmos de rotação para deixarem as barras laterais em pé, como no jogo pong.

4.8 Scale (Escala)

A escala de um GameObject determina seu tamanho relativo no espaço tridimensional. É representada por um vetor tridimensional (x, y, z) que especifica a multiplicação da dimensão do GameObject ao longo dos eixos x, y e z.

4.9 Scaling (Dimensionamento)

Ação de alterar a escala de um GameObject aumentando ou diminuindo seu tamanho. Dimensionar um objeto em qualquer eixo faz com que seu tamanho mude nesse eixo. Você ativa os gizmos de Dimensionamento selecionando a ferramenta de scaling (tecla de atalho r) como demonstrado na figura 4.9. Segurando e arrastando um dos gizmos de dimensionamento altera o tamanho do gameObject.

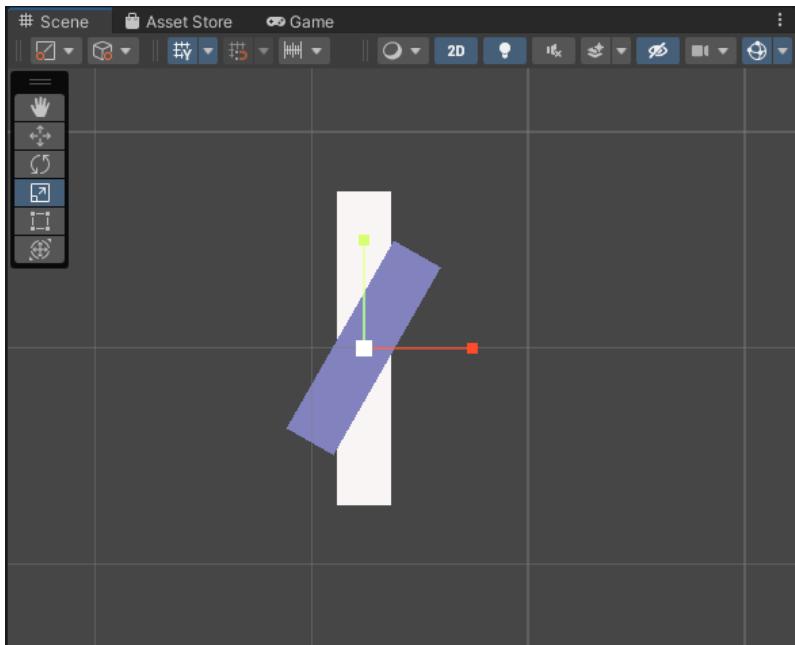


Figura 4.9 Gizmos de Dimensionamento.

4.10 Cuidados ao realizar transformações

Conforme mencionado anteriormente, as transformações usam o sistema de coordenadas local. Portanto, as mudanças que são feitas impactam transformações futuras. Observe a figura 4.10. Note como as mesmas transformações, quando aplicadas na ordem inversa, têm efeitos muito diferentes.

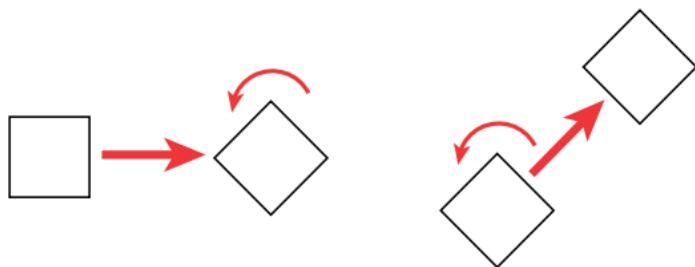


Figura 4.10 Transformações em ordens diferentes.

Como você pode ver, não prestar atenção à ordem de transformação pode ter consequências inesperadas. Felizmente, as transformações têm efeitos constantes que podem ser planejados:

Translação: A translação é uma transformação bastante inerte. Isso significa que quaisquer alterações aplicadas depois geralmente não serão afetadas.

Rotação: A rotação altera a orientação dos eixos do sistema de coordenadas locais. Quaisquer translações aplicadas após uma rotação fazem com que o objeto se move ao longo dos novos eixos. Se você girar um objeto 180 graus em torno do eixo z, por exemplo, e depois movê-lo na direção y positiva, o objeto irá se mover para baixo em vez de para cima.

Dimensionamento: o dimensionamento altera efetivamente o tamanho da grade de coordenadas locais. Basicamente, quando você dimensiona um objeto para ser maior, você está na verdade dimensionando o sistema de coordenadas local para ser maior. Isso faz com que o objeto pareça crescer. Essa mudança é multiplicativa. Por exemplo, se um objeto for dimensionado para 1 (seu tamanho natural padrão) e depois movido 5 unidades ao longo do x eixo, o objeto parece mover 5 unidades para a direita. Se o mesmo objeto fosse dimensionado para 2, entretanto, então transladar 5 unidades no eixo x resultaria no objeto parecendo mover 10 unidades para a direita. Isto ocorre porque o sistema de coordenadas locais é agora o dobro do tamanho e 5 vezes 2 é igual a 10. Inversamente, se o objeto fosse dimensionado para 0,5 e depois movido, ele pareceria mover apenas 2,5 unidades ($0,5 \times 5 = 2,5$).

Cascateamento de Transformações

No último capítulo, você aprendeu como aninhar GameObjects na Hierarquia (arraste um objeto para outro). Lembre-se de que quando você tem um objeto aninhado dentro de outro, o objeto de nível superior é o pai e o outro objeto é o filho.

Cada GameObject na hierarquia possui seu próprio componente Transform, que controla sua posição, rotação e escala. Quando um GameObject é filho de outro, suas transformações são definidas em relação ao espaço do GameObject pai.

As transformações locais de um GameObject filho são relativas ao seu GameObject pai. Por exemplo, mover um GameObject filho em (1, 0, 0) altera sua posição em relação ao seu pai, não ao mundo.

As transformações globais de um GameObject filho são relativas ao mundo ou à cena. Isso significa que as transformações do GameObject pai são aplicadas às transformações do GameObject filho.

Quando você aplica uma transformação (como translação, rotação ou escala) a um GameObject pai, essa transformação é "herdada" por todos os GameObjects filhos na hierarquia. Por exemplo, se você rotacionar um GameObject pai em 90 graus em torno do eixo y, todos os GameObjects filhos serão rotacionados em relação ao eixo y. É importante observar que as transformações globais de um GameObject filho são calculadas com base nas transformações locais do GameObject pai. Isso significa que as transformações de um GameObject pai afetam as transformações de seus filhos, mas as transformações dos filhos ainda são relativas ao pai.

5. Trabalhando com Scripts

Na Unity, os scripts são usados para adicionar funcionalidades e comportamentos aos *GameObjects* em sua cena. Eles são escritos na linguagem programação C# (C Sharp).

Antes de criar scripts, é melhor criar uma pasta Scripts na pasta Assets na janela de projeto. Assim que tiver uma pasta para conter todos os seus scripts, basta clicar com o botão direito na pasta e selecionar Criar > Script C#. Depois de criado, você precisa dar um nome ao seu script antes de continuar.

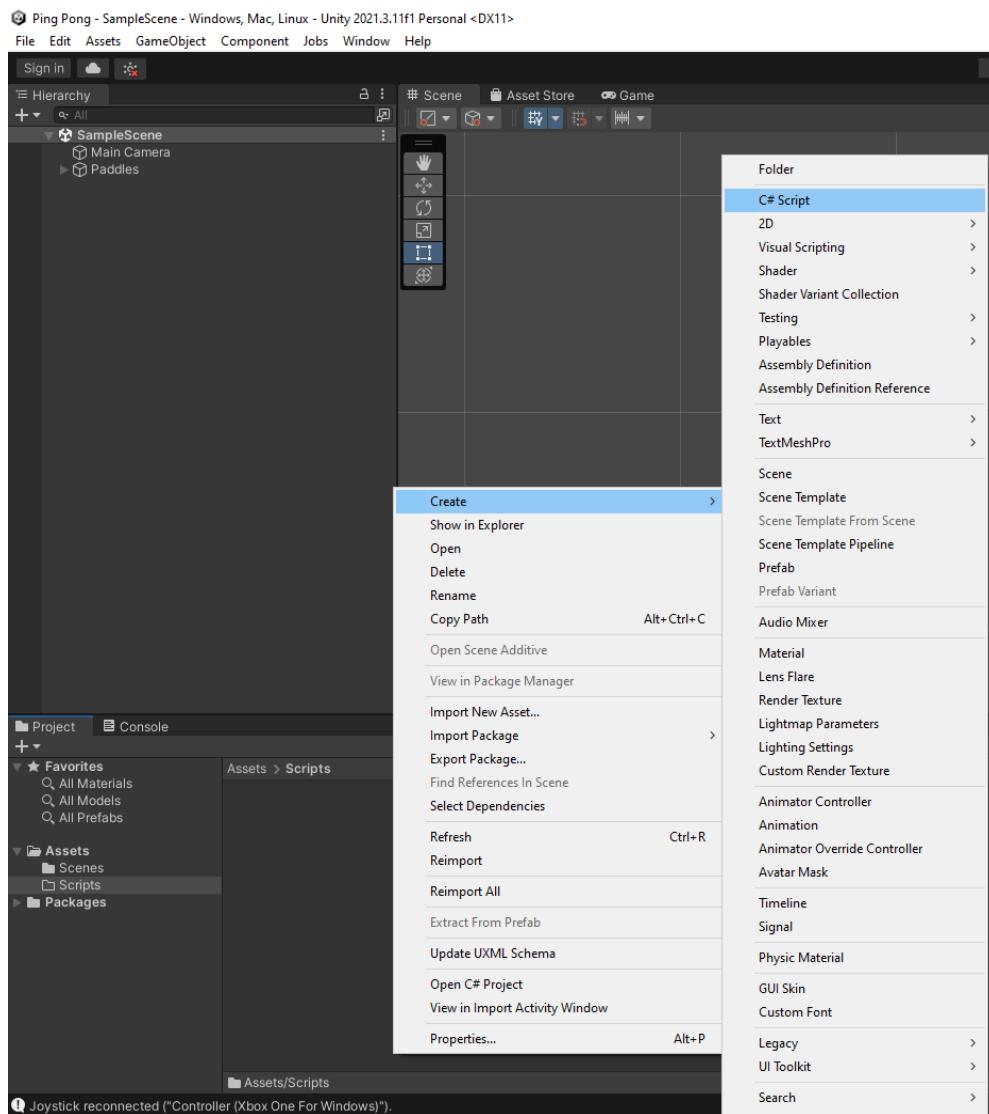


Figura 5.1 Criando Script Através da Project View.

Os scripts são associados aos *GameObjects* na forma de Componentes. Você pode criar novos scripts ou adicionar scripts existentes a um *GameObject* arrastando e soltando o script no Inspector ou em cima do *GameObject* desejado na Hierarquia. Existe a opção de clicar na hierarquia em Add Component > New Script > Create and Add.

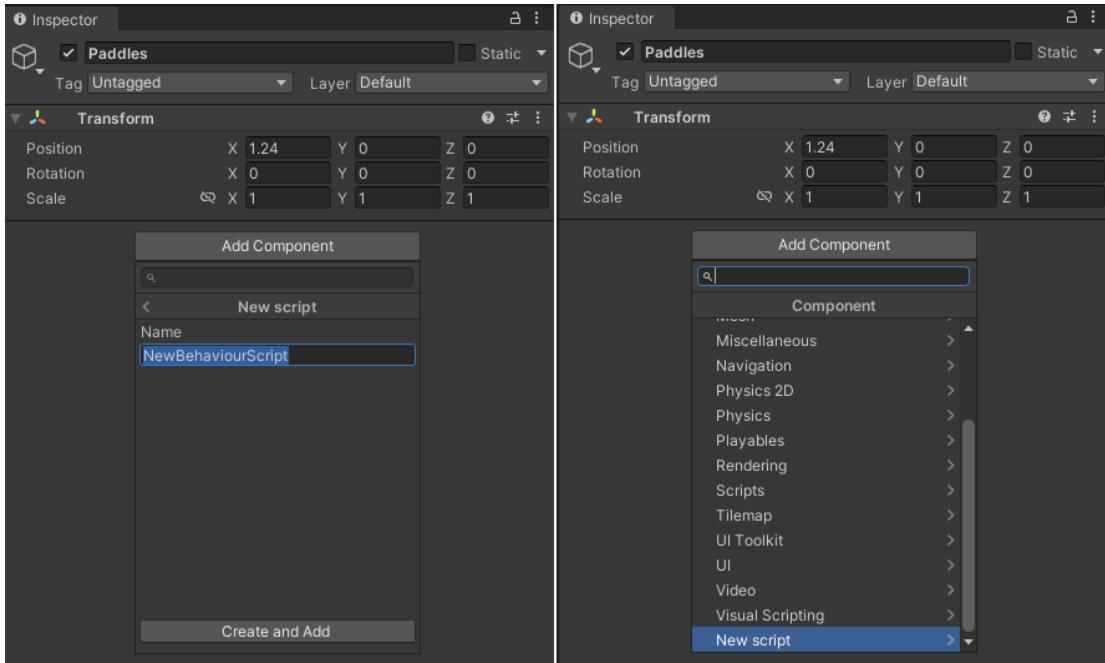


Figura 5.2 Criando Script Através do Inspector.

Crie um Script, dê a ele o nome de *PlayerMovement* e adicione ao *gameObject* da barra da direita.

5.1 Selecionando a IDE

Na Unity, a escolha da IDE (*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado) é uma decisão importante para os desenvolvedores, pois é onde eles escrevem e gerenciam seus scripts C#. Por padrão a Unity utiliza o Microsoft Visual Studio, porém muitos desenvolvedores preferem usar uma IDE como o *Visual Studio Code*. A escolha da IDE fica a seu critério, caso queira trocar a IDE para uma diferente do Microsoft Visual Studio clique em *Edit>Preferences* como na imagem abaixo.

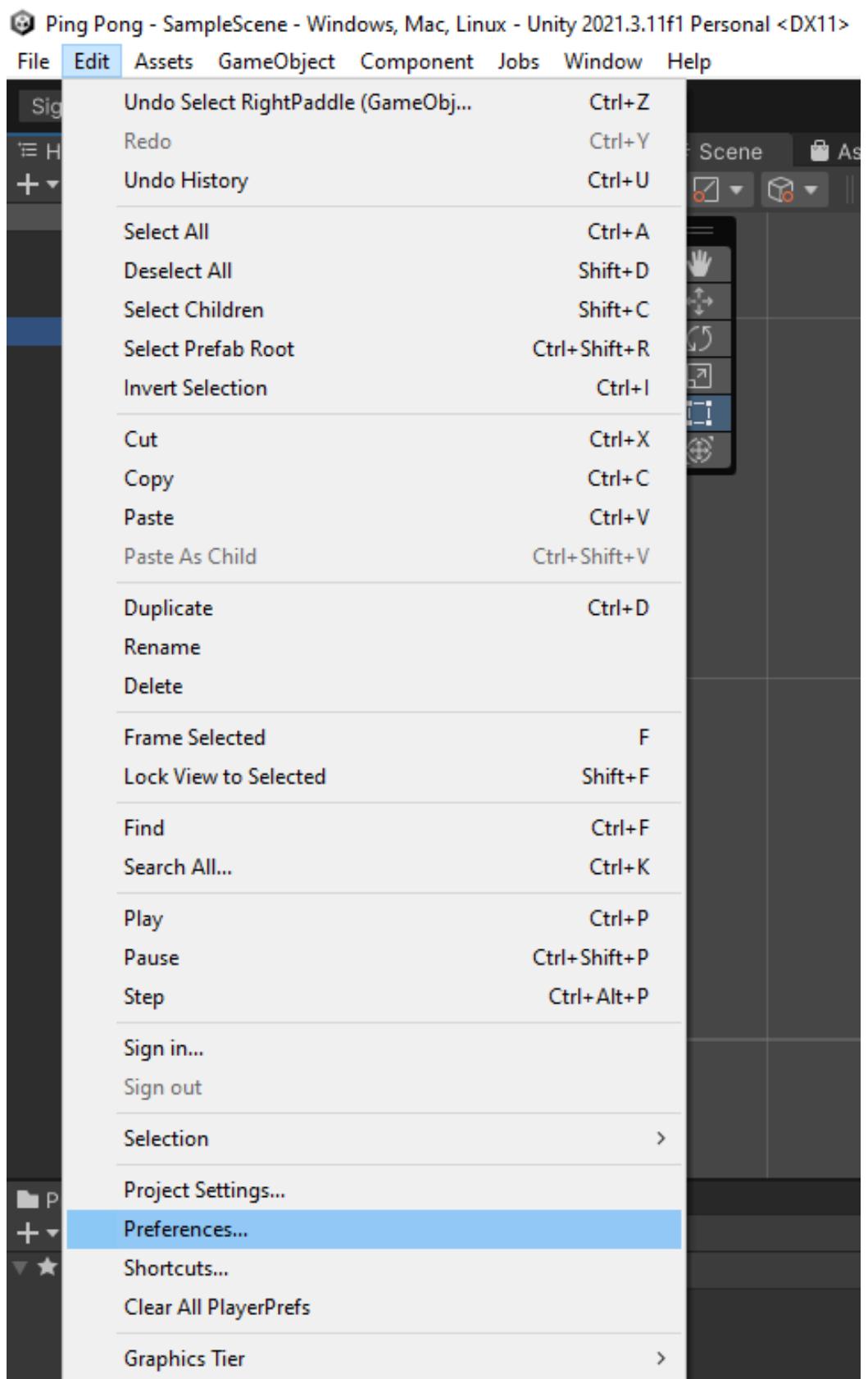


Figura 5.3 Abrindo a janela Preferences.

Na janela que abriu, selecione *External Tools* e selecione a IDE desejada no menu flutuante *External Scripts Editor* como indicado na figura 5.4.

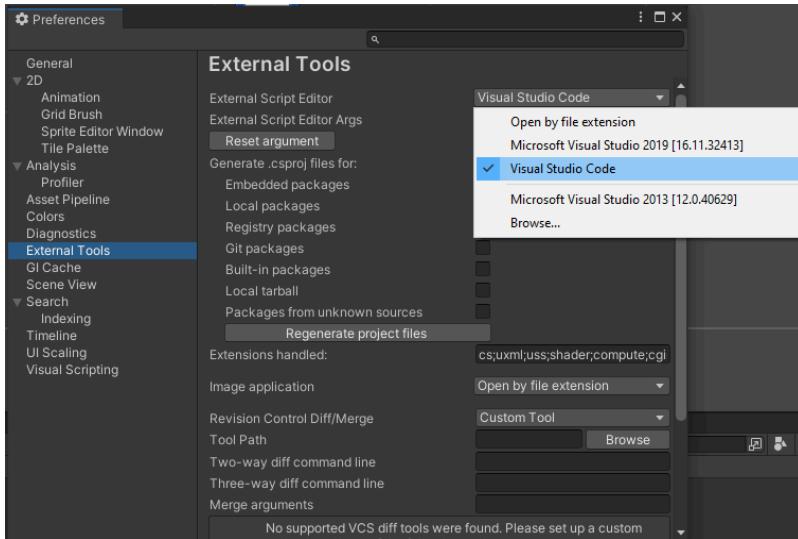


Figura 5.4 Menu flutuante de IDE.

Após isso feche essa janela e clique duas vezes no script *PlayerMovement.cs*.

5.2 Estrutura Básica de um Script Unity

Ao clicar no Script *PlayerMovement.cs* você se deparou com o código abaixo, que é criado automaticamente pela Unity:

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

public class PlayerMovement : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

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

Esse código pode ser dividido em 3 seções. Veja sobre cada seção a seguir.

5.3 Seção *using*

A primeira parte lista as bibliotecas que este script utilizará. Se parece com isso:

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

A linha **using UnityEngine** importa o *namespace* *UnityEngine*, que contém todas as classes e funções necessárias para interagir com o motor da Unity. Isso permite que você use os tipos e funcionalidades fornecidos pela Unity em seu script. De modo geral, você não irá alterar esta seção e deverá deixá-la de lado por enquanto.

5.4 Seção de declaração de Classe

A próxima parte é chamada de declaração de classe. Cada script contém uma classe com o nome do script. Se parece com seguinte:

```
public class PlayerMovement : MonoBehaviour {}
```

Todo o código entre a chave de abertura { e a chave de fechamento } fará parte desta classe e, portanto, parte do script. Todo o seu código deve ficar entre esses colchetes. Você deve ter notado o nome *MonoBehaviour*. Classes são um conceito do Paradigma de Programação Orientada a Objetos, paradigma utilizado pelo C#. Um conceito importante deste paradigma é a herança, em que uma classe pode herdar as características de outra classe, e é exatamente o que está acontecendo com o *MonoBehaviour*. *MonoBehaviour* é uma classe própria da Unity, ela permite que as classes que criamos utilizem os recursos próprios da Unity, por isso todo script criado deve herdar as características de *MonoBehaviour*. Mais uma vez, como acima, você raramente altera isso e deve deixá-lo como está por enquanto.

5.5 O conteúdo da Classe

Tudo que está entre os colchetes de abertura e fechamento da classe é considerado “dentro” da classe. Todo o seu código irá aqui. Por padrão, um script criado pela Unity contém dois métodos dentro da classe, *Start* e *Update*.

```
// Start is called before the first frame update
void Start()
{
}
```

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

Esses métodos são chamados de *Callbacks*. Os métodos de callback são métodos especiais que são chamados automaticamente pela Unity em resposta a eventos específicos durante a vida útil de um *GameObject*.

- **Start:** é chamado uma vez quando o script é inicializado e o *GameObject* ao qual está anexado se torna ativo na cena. Ele é executado apenas uma vez durante a vida útil do *GameObject*.
- **Update:** é chamado a cada quadro de renderização enquanto o *GameObject* estiver ativo na cena. Isso significa que ele é chamado várias vezes por segundo, dependendo da taxa de quadros (frame rate) do jogo. O Update é usado para realizar ações que precisam ser atualizadas continuamente, como movimento de objetos, detecção de entrada do jogador, etc. A taxa de quadros é influenciada pela capacidade de processamento do hardware e pelo tempo que cada quadro leva para ser renderizado.

Importante notar que o Método *Start* sempre é chamado antes da primeira chamada do Método *Update*.

Vamos entender na prática como esses métodos funcionam. Para isso utilizaremos o método *print*, que é utilizado para enviar mensagens para a janela de console da Unity.

Dentro do método *Start* digite o seguinte código:

```
void Start()
{
    print("Start roda antes do Update");
}
```

Salve seu código, espere a Unity recarregar o código e clique em *play* com o modo *play focused* ativo na *Game View*. Note como a mensagem do método *print* aparece uma única vez no console.

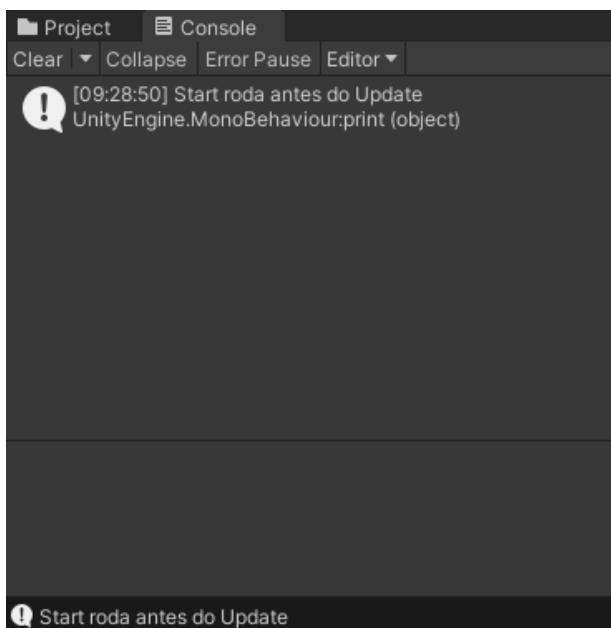


Figura 5.5 Log do console para método *Start*.

Agora, dentro do método *Update* digite o seguinte código:

```
void Update()
{
    print("Update roda repetidamente depois do Start");
}
```

Novamente salve seu código, espere a Unity recarregar o código e clique em *play* com o modo *play focused* ativo na *Game View*. Note como a mensagem do método *print* aparece diversas vezes no console. Note também que o método *Start* rodou antes de todos os métodos *Updates* (para ver isso você precisará voltar para o topo do console com a barra de rolagem).

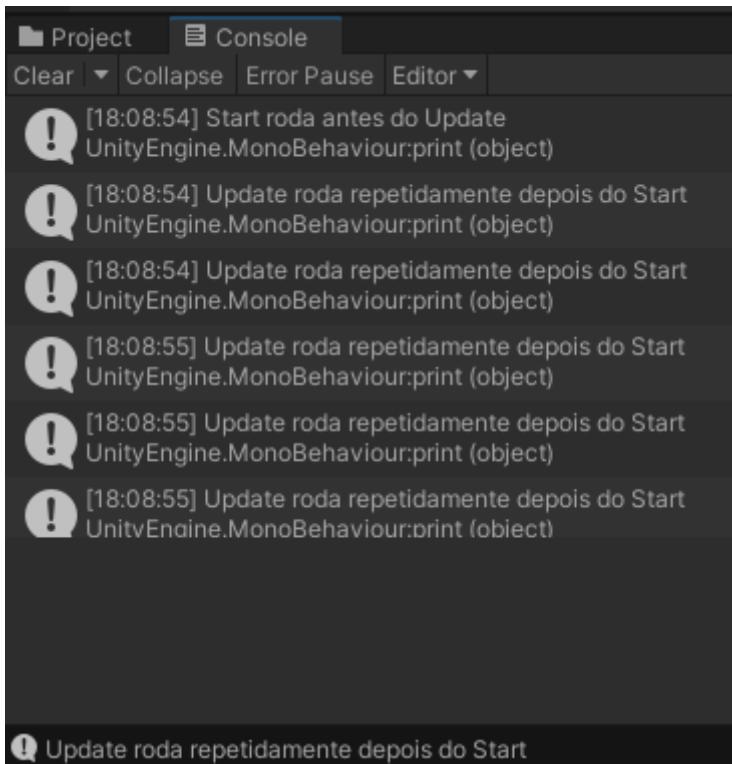


Figura 5.6 Log do console para método Start e Update.

Agora apague o conteúdo do método *Update* para que possamos realizar testes futuros.

5.6 Variáveis

O C# é uma linguagem fortemente tipada, ou seja, cada variável deve ser atribuída um tipo, além do nome. Você cria uma variável com a seguinte sintaxe:

<tipo de variável> <nome>;

Portanto, para criar um número inteiro chamado num1, digite o seguinte:

```
int num1;
```

A seguir temos uma tabela com os tipos de variáveis mais comuns em C#. Cada tipo de variável possui um tamanho específico em bytes na memória e uma faixa de valores que pode armazenar.

Tipo	Descrição	Tamanho	Faixa de Valores
int	Números inteiros	4 bytes	-2,147,483,648 a 2,147,483,647
float	Números de ponto flutuante	4 bytes	-3.402823e38 a 3.402823e38
double	Números de ponto flutuante	8 bytes	-1.79769313486232e308 a 1.79769313486232e308
bool	Valores booleanos (verdadeiro ou falso)	2 byte	true ou false
char	Caractere Unicode	2	Qualquer caractere Unicode
string	Sequência de caracteres	Variável	Qualquer sequência de caracteres

Vector2 e Vector3

Esses são tipos de variáveis da Unity, eles carregam respectivamente dois e 3 valores numéricos dentro de si e são usados para indicar os valores de vários componentes importantes da Unity. Veja abaixo como criar essas variáveis:

```
Vector2 vector2 = new Vector2(0, 0);
Vector3 vector3 = new Vector3(0, 0, 0);
```

5.7 Escopo de variável

O escopo de uma variável em C# refere-se à região do código onde essa variável é acessível e pode ser usada. Em outras palavras, o escopo determina onde uma variável é válida e pode ser referenciada. A razão pela qual isso é importante é que as variáveis só podem ser usadas nos blocos em que são criadas. Portanto, se uma variável for criada dentro do método Start de um script, ela não estará disponível no método Update.

Escopo Local: Uma variável declarada dentro de um bloco de código, como dentro de um método, uma instrução condicional (if) ou um loop (for, while), tem escopo local. Ela só pode ser acessada dentro desse bloco de código e não é visível fora dele. Exemplo:

```
public class PlayerMovement : MonoBehaviour
{
    void Start()
    {
        int x = 10; // Variável escopo local
        print(x); // Acesso válido
    }
}
```

Escopo Global: Variáveis globais, que são declaradas fora de qualquer método ou bloco de código, têm escopo global e podem ser acessadas em todo o código da classe.

```
public class PlayerMovement : MonoBehaviour
{
    int variavelGlobal = 100;
    // Variável com escopo global

    void Start()
    {
        print(variavelGlobal);
        //Acesso válido dentro do método
    }
}
```

5.8 Visibilidade de variável

Na Unity, a visibilidade de uma variável se refere à sua acessibilidade dentro do editor da Unity e em scripts C#. A visibilidade determina se a variável pode ser acessada por outros scripts, se é exibida no Inspector da Unity e se pode ser modificada diretamente no Editor. Aqui estão os principais níveis de visibilidade de variáveis na Unity:

Pública (*public*): Variáveis públicas são visíveis para outros scripts e podem ser acessadas e modificadas no Inspector da Unity. Elas são declaradas usando o modificador *public*.

```
public int minhaVariavelPublica;
```

Privada (*private*): Variáveis privadas não são visíveis para outros scripts e não podem ser acessadas ou modificadas no Inspector da Unity. Elas são declaradas usando o modificador *private*.

```
private int minhaVariavelPrivada;
```

Vamos ver isso na prática. Digite o seguinte código dentro da classe PlayerMovement:

```
public class PlayerMovement : MonoBehaviour
{
    public int minhaVariavelPublica;

    private int minhaVariavelPrivada;

    // Start is called before the first frame update
    void Start()
    {
        print("minha variável pública");
        print(minhaVariavelPublica);
        print("minha variável privada");
        print(minhaVariavelPrivada);
    }
}
```

Salve seu código, espere a Unity recarregar e selecione o gameObject da barra da esquerda na hierarquia. Note que no inspector aparece a variável minhaVariavelPublica mas não aparece a variável minhaVariavelPrivada.

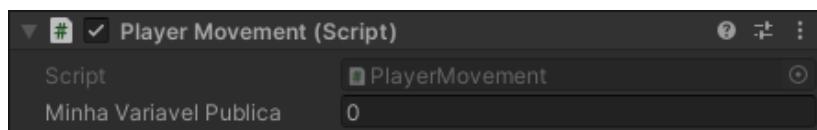


Figura 5.7 Variável pública no Inspector.

Altere o valor do campo Minha Variavel Publica para um valor diferente de 0 e clique em play com o modo play focused ativo na Game View. Repare como o valor que você digitou aparece como variável pública. Como não definimos um valor para a variável privada ela aparece como 0, que é o valor padrão para variáveis do tipo int.

5.9 Operadores

Em C#, os operadores são símbolos especiais que realizam operações em variáveis e valores. Aqui está uma lista dos principais operadores em C#:

- **Aritméticos:**
 - + (adição)
 - - (subtração)
 - * (multiplicação)
 - / (divisão)
 - % (módulo, retorna o resto da divisão)
 - ++ (incremento)
 - -- (decremento)
- **Atribuição:**
 - = (atribuição)
 - += (adição e atribuição)
 - -= (subtração e atribuição)
 - *= (multiplicação e atribuição)
 - /= (divisão e atribuição)
 - %= (módulo e atribuição)
- **Comparação:**
 - == (igual a)
 - != (diferente de)
 - > (maior que)
 - < (menor que)
 - >= (maior ou igual a)
 - <= (menor ou igual a)
- **Lógicos:**
 - && (e lógico)
 - || (ou lógico)
 - ! (negação lógica)

Esses são alguns dos operadores mais comuns em C#. Eles são usados para realizar uma variedade de operações, desde operações matemáticas básicas até operações de comparação, lógica e manipulação de *bits*. Entender e usar os operadores corretamente é fundamental para escrever código eficiente e funcional em C#.

5.10 Estruturas de decisão e repetição

Em C#, as estruturas de decisão e repetição são usadas para controlar o fluxo de execução do programa com base em condições e iterações.

if-else: A estrutura *if-else* permite executar um bloco de código se uma condição for verdadeira e outro bloco de código se a condição for falsa.

```
if (condicao)
{
    // Bloco de código se a condição
    // for verdadeira
}
else
{
    // Bloco de código se a condição
    // for falsa
}
```

Às vezes você deseja que seu código se bifurque em um dos vários caminhos. Você pode querer que o usuário possa escolher entre uma seleção de opções (como um menu, por exemplo). O *if /else if* é estruturado da mesma maneira que a estrutura anterior, exceto que possui múltiplas condições:

```
if (condicao1)
{
    // Bloco de código se a condição 1
    // for verdadeira
}
else if (condicao2)
{
    // Bloco de código se a condição 1
    // for falsa e a condição 2 for
    // verdadeira
}
else
{
    // Bloco de código se todas as
    // condições forem falsas
}
```

switch-case: A estrutura *switch-case* permite avaliar uma expressão e executar diferentes blocos de código com base nos valores dessa expressão.

```
switch (expressao)
{
    case valor1:
        // Bloco de código se a expressão for
        // igual a valor1
}
```

```

        break;
case valor2:
    // Bloco de código se a expressão for
    // igual a valor2
    break;
default:
    // Bloco de código se a expressão não
    // corresponder a nenhum valor
    break;
}

```

for: A estrutura *for* é usada para executar um bloco de código um número específico de vezes.

```

for (int i = 0; i < limite; i++)
{
    // Bloco de código a ser repetido
}

```

while: A estrutura *while* é usada para repetir um bloco de código enquanto uma condição for verdadeira.

```

while (condicao)
{
    // Bloco de código a ser repetido
}

```

5.11 Métodos

Em C#, métodos são blocos de código que executam uma tarefa específica e podem ser chamados (ou invocados) de outros lugares no código. Eles são usados para agrupar e organizar funcionalidades relacionadas, o que facilita a manutenção e reutilização do código.

5.12 Declaração do Método

Um método é definido usando a seguinte sintaxe:

```

[modificador] tipoRetorno NomeDoMetodo(parâmetros)
{
    // Corpo do método
}

```

- modificador: opcional e pode ser *public*, *private*, *protected* ou *static*, entre outros. Define a visibilidade e a acessibilidade do método.
 - *public*: O método pode ser acessado de qualquer lugar no código, dentro ou fora da classe.
 - *private*: O método só pode ser acessado dentro da própria classe em que foi definido. Ele não é visível para outras classes ou subclasses.
 - *protected*: O método pode ser acessado na própria classe e em classes derivadas (herança).
 - *static*: Métodos estáticos pertencem à classe e não a uma instância específica. Para chamá-los, usa-se o nome da classe.
- tipoRetorno: especifica o tipo de dados que o método retorna. Pode ser qualquer tipo de dados válido em C#, incluindo tipos primitivos, classes, estruturas ou até mesmo void se o método não retornar nenhum valor.
- NomeDoMetodo: o nome dado ao método.
- parâmetros: valores que podem ser passados para o método para que ele os utilize durante sua execução. Os parâmetros são opcionais e podem ser de qualquer tipo de dados.

Veja um exemplo de método simples:

```
public int Somar(int a, int b)
{
    return a + b;
}
```

Neste exemplo, o método Somar recebe dois parâmetros do tipo int e retorna a soma deles.

Os métodos são chamados em outras partes do código da seguinte maneira:

```
int resultado = Somar(3, 5);
```

Um método pode retornar um valor usando a palavra-chave *return*. Se o método não retornar nenhum valor, seu tipo de retorno deve ser *void*.

```
public void PrintNúmero(int a)
{
    print(a);
}
```

Vamos escrever o método de *PlayerMovement.cs* que será responsável por fazer as barras laterais subirem e descerem.

```
// Update is called once per frame
void Update()
{
    MovePlayer();
```

```
}

private void MovePlayer()
{
    print("Player está se movendo");
}
```

O nosso código ainda não faz as barras se moverem, mas cuidaremos disso mais para frente.

5.13 Inputs

Sem o *input* do jogador, os videogames seriam apenas vídeos. O *input* do jogador pode vir em muitas variedades diferentes. As entradas podem ser físicas, como *gamepads*, *joysticks*, teclados e mouses. Existem controles capacitivos, como as telas sensíveis ao toque encontradas em dispositivos móveis modernos. Existem também dispositivos de movimento como o *Wii Remote*, o *PlayStation Move* e o Microsoft *Kinect*. Mais rara é a entrada de áudio que usa microfones e a voz do jogador para controlar um jogo. Nesta seção, você aprenderá como escrever código para permitir que o jogador interaja com seu jogo por meio de dispositivos físicos.

Aqui estão algumas maneiras de identificar inputs na Unity usando C#:

Input.GetAxis(string chave):

Este método retorna o valor atual de um eixo virtual (valor entre 1 e -1), que é uma representação abstrata de entradas analógicas, como as teclas de seta ou um *joystick*. Isso é útil para movimentos suave, como mover um personagem. Para isso, você deve fornecer uma chave como parâmetro para definir qual eixo está sendo usado. Você pode encontrar, criar e definir os eixos clicando em *Edit > Project Settings* e na janela que abrir selecionar *Input Manager*.

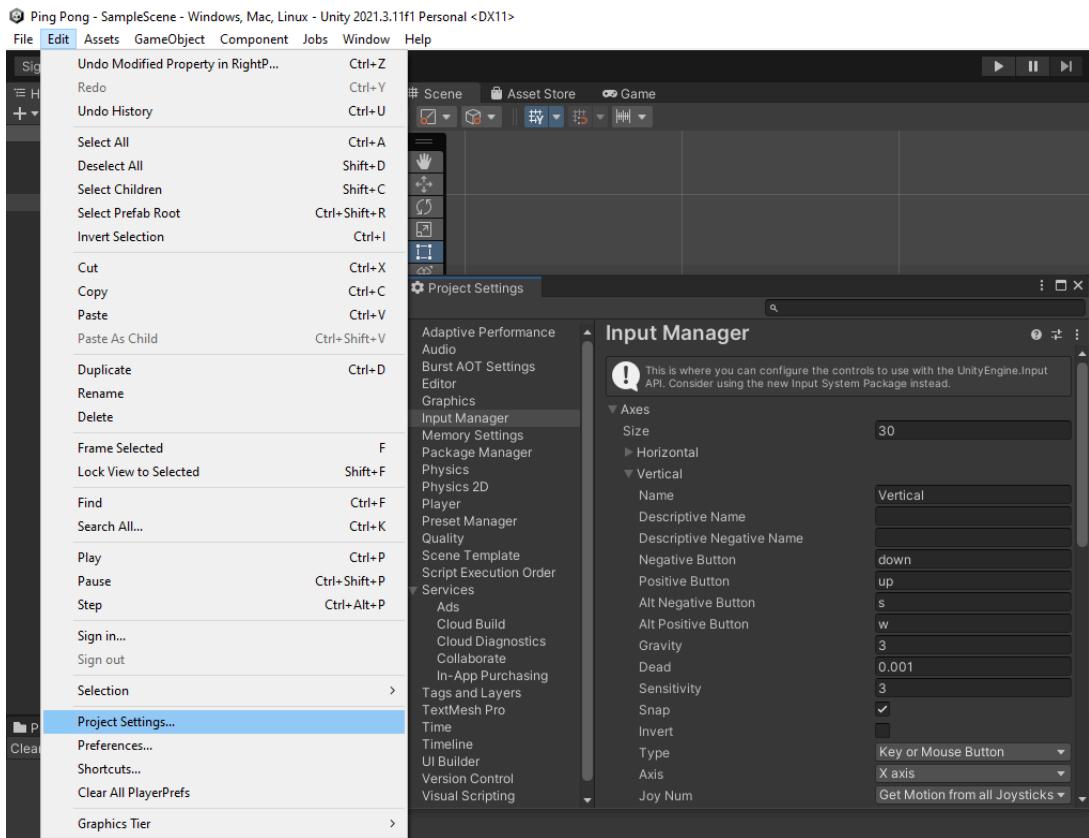


Figura 5.8 Abrindo Project Settings.

Analisemos o eixo vertical na figura 5.9.

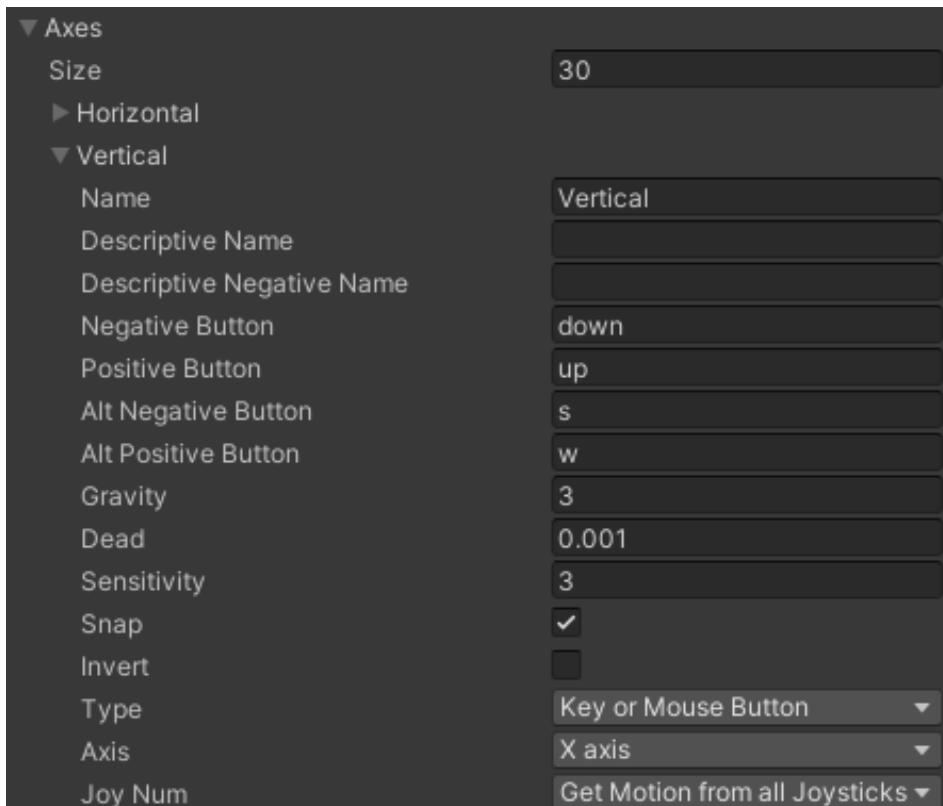


Figura 5.9 Eixo Vertical.

Os campos *Negative Button* e *Alt Negative Button* indicam os inputs que retornam -1. *Positive Button* e *Alt Positive Button* indicam os inputs que retornam 1. Isso significa que a chamada `Input.GetAxis("Vertical")` retorna -1 caso o jogador mantenha pressionada a tecla da seta para baixo ou a tecla s, retorna 1 quando o jogador mantenha pressionada a tecla *seta para cima* ou a letra w. Caso nenhuma dessas teclas seja pressionada o método retorna 0. Vamos testar isso no método `MovePlayer()`:

```
private void MovePlayer()
{
    if(Input.GetAxis("Vertical") > 0)
    {
        print("Player está se movendo para cima");
    }
    else if(Input.GetAxis("Vertical") < 0)
    {
        print("Player está se movendo para baixo");
    }
    else
    {
        print("Player está parado");
    }
}
```

Salve seu código, espere a Unity recarregar o código e clique em play com o modo *play focused* ativo na *Game View*. Note que enquanto você não pressiona as teclas do eixo a mensagem "Player está parado" é escrita várias vezes no console. Ao manter pressionado w ou tecla para cima a mensagem "Player está se movendo para cima" aparece no console e ao manter pressionado o s ou tecla para baixo a mensagem "Player está se movendo para baixo" aparece no console.

Input.GetKey(KeyCode keyCode):

Este método retorna verdadeiro em todos os frames em que o jogador estiver pressionando a tecla especificada como parâmetro e retorno falso caso contrário. Ele recebe como parâmetro um *KeyCode*.

KeyCode é um enum, uma estrutura de dados que permite definir um tipo de valor com um conjunto de constantes nomeadas. *KeyCode* possui constantes que representam cada tecla do teclado.

```
if (Input.GetKey(KeyCode.W))
{
    // Ação enquanto a tecla W está sendo
```

```
// mantida pressionada  
}
```

Realize o mesmo teste que realizamos para `Input.GetAxis(String)` mas com o código abaixo:

```
private void MovePlayer()  
{  
    if(Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.UpArrow))  
    {  
        print("Player está se movendo para cima");  
    }  
    else if(Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.DownArrow))  
    {  
        print("Player está se movendo para baixo");  
    }  
    else  
    {  
        print("Player está parado");  
    }  
}
```

Inputs diferentes para cada jogador

Até agora definimos *inputs* para subir e descer a barra, porém cada jogador precisará fazer isso usando teclas diferentes do teclado, Jogador 1 usando W e S e jogador dois usando as setas para cima e para baixo. Poderíamos criar um script chamado *Player2Movement*, mas essa solução não é ideal pois a cada mudança que fizéssemos em um script teríamos que lembrar de realizar no segundo script também. Existem duas possíveis soluções para este problema, mas antes vamos criar uma variável que indique qual jogador está controlando a barra.

Essa variável precisa ter dois valores possíveis específicos representando qual jogador está controlando. Poderíamos usar uma string ou um inteiro, porém como esses tipos podem assumir qualquer valor não são tipos seguros, portanto para isso vamos utilizar um tipo de dados que permite definir um conjunto de constantes nomeadas, um *enum*.

Já trabalhamos com um enum próprio da Unity antes, o enum *KeyCode* que contém constantes para os possíveis inputs do jogador, porém podemos criar nossos próprios *enums* da seguinte maneira.

```
public enum PlayerNumber  
{  
    Player1 = 1,  
    Player2 = 2
```

```
}
```

```
public PlayerNumber playerNumber;
```

Player1 e *Player2* são nossas constantes. As atribuições numéricas são opcionais, por padrão elas vão de 0 em diante. Adicionamos as atribuições numéricas para manter o padrão com o número do jogador.

Note que no *inspector* nosso *enum* aparece como um menu que nos permite selecionar qual constante queremos usar. Selecione um *player* diferente para cada barra.

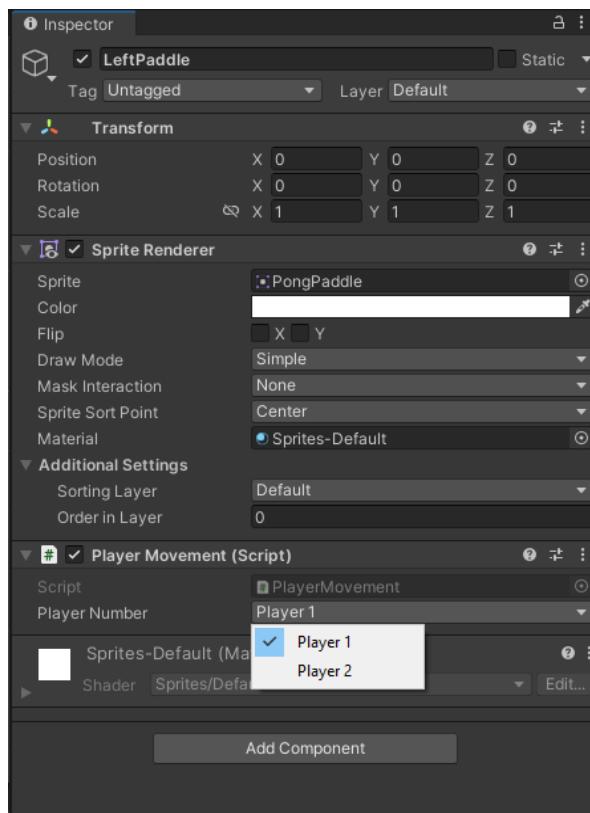


Figura 5.10 Variável Player Number no Inspector.

Primeiro Método

O primeiro método envolve selecionar *KeyCode*s diferentes para cada jogador. Vamos criar duas variáveis privadas para representar os botões para mover para cima e para baixo e verificar no método *Start* qual o jogador.

```
private KeyCode upMovementButton = KeyCode.UpArrow;
private KeyCode downMovementButton = KeyCode.DownArrow;

// Start is called before the first frame update
void Start()
{
    if(playerNumber == PlayerNumber.Player2)
```

```

    {
        upMovementButton = KeyCode.W;
        downMovementButton = KeyCode.S;
    }
}

```

Agora iremos adaptar o nosso método *MovePlayer*:

```

private void MovePlayer()
{
    if(Input.GetKey(upMovementButton))
    {
        print($"Player{(int)playerNumber} está se movendo para cima");
    }
    else if(Input.GetKey(downMovementButton))
    {
        print($"Player{(int)playerNumber} está se movendo para baixo");
    }
    else
    {
        print($"Player{(int)playerNumber} está parado");
    }
}

```

Você deve ter notado as alterações que fizemos em nossos prints. O operador \$ antes de uma string nos permite realizar uma operação conhecida como interpolação, que nos permite inserir um valor em uma string entre chaves {}. No entanto, o mais interessante desse código é o que estamos interpolando em nossas strings. Quando colocamos um tipo primitivo entre parênteses estamos realizando um casting. Casting é a ação de transformar um tipo em outro, em nosso caso estamos transformando nosso playerNumber em um inteiro, para ser mais exato os inteiros que atribuímos suas constantes: 1 para *Player1* e 2 para *Player2*.

Aperte o play e observe o console. Note que ao apertar W e S ou Seta para Cima e Seta para baixo as mensagens são atualizadas para os players corretos.

Segundo Método

O segundo método consiste em criar Input Axis separados para cada player chamados Vertical1 e Vertical2.

Abra a tela de Input Manager em *Project Settings*, clique com o botão direito em Vertical e Clique em *Duplicate Array Element*.

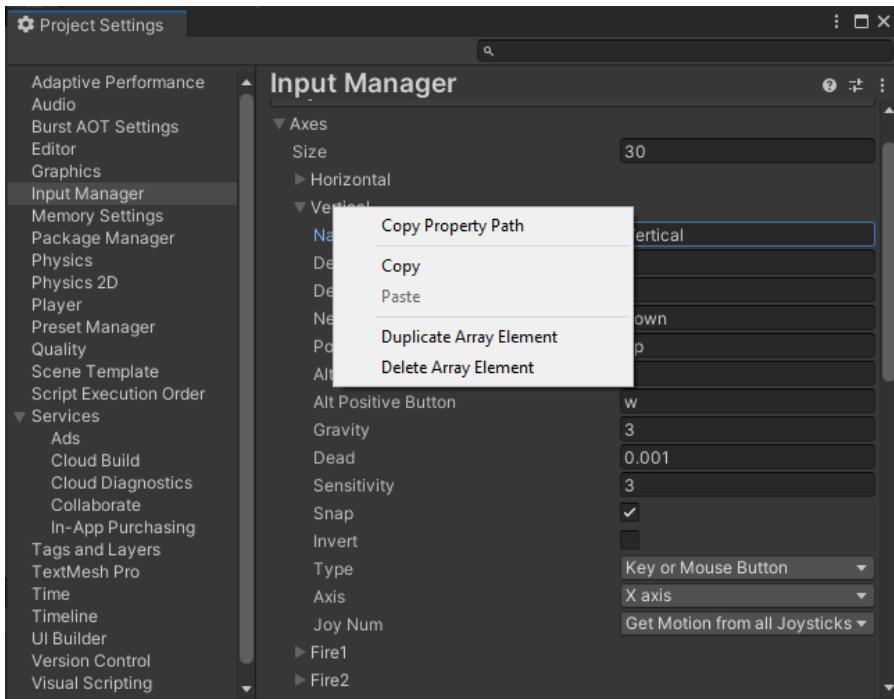


Figura 5.11 Duplicando Input Axis.

Note que agora existem dois espaços chamados Vertical altere seus campos *Name*, *Negative Button* e *Positive Button* como na figura 5.12.

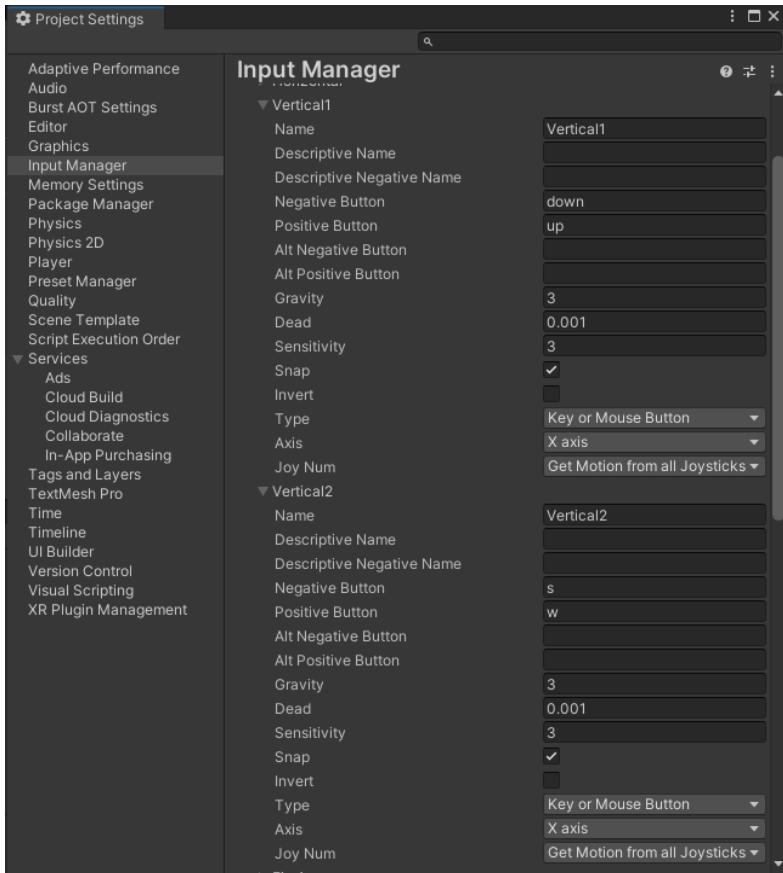


Figura 5.12 Input Axis Vertical1 e Vertical2.

Agora criaremos uma variável do tipo string chamada *playerAxis* e no método *Start* nós atribuímos o valor correspondente ao player.

```
private string playerAxis;

// Start is called before the first frame update
void Start()
{
    playerAxis = $"Vertical{(int)playerNumber}";
}
```

Agora iremos alterar o método *MovePlayer*.

```
private void MovePlayer()
{
    if(Input.GetAxis(playerAxis) > 0)
    {
        print($"Player{(int)playerNumber} está se movendo para cima");
    }
    else if(Input.GetAxis(playerAxis) < 0)
    {
        print($"Player{(int)playerNumber} está se movendo para baixo");
    }
    else
    {
        print($"Player{(int)playerNumber} está parado");
    }
}
```

Aperte o play e observe o console. Note que ao apertar W e S ou Setas para Cima e Setas para Baixo as mensagens são atualizadas para os players corretos.

Estes são os métodos de separação dos jogadores, sintam-se à vontade para escolher qual deles você prefere.

6. Prefabs

Até o momento estivemos alterando os gameobjects de forma separada, porém, quando se trata de jogos eletrônicos muitas vezes existem entidades que possuem características em comum. Podem ser itens, inimigos e até jogadores, e muitos deles podem existir em centenas e até milhares de cópias, o que tornaria bastante custoso fazer uma simples alteração no comportamento deles. Felizmente a Unity possui um poderoso recurso para nos ajudar, os prefabs. Um "prefab" é um tipo especial de *GameObject* que serve como um modelo reutilizável para criar instâncias de *GameObjects* dentro do seu jogo. Um prefab contém uma configuração completa de um *GameObject*, incluindo seus componentes, transformações, scripts e outros atributos.

6.1 Terminologia

Alguns termos são importantes ao trabalhar com prefabs. Se você está familiarizado com os conceitos de programação orientada a objetos, vai notar algumas semelhanças:

- **Prefab:** O prefab é o *GameObject* base. Ele existe apenas na Project View. Pense nele como o modelo.
- **Instância:** Um *GameObject* real do prefab em uma cena. Se o prefab é um modelo para um carro, uma instância é um carro real. Se um *GameObject* na cena é referido como um prefab, significa que é uma instância de um prefab. A frase "instância de um prefab" é sinônimo de "GameObject de um prefab" ou até mesmo "clone de um prefab".
- **Instanciar:** O processo de criar uma instância de um prefab. É um verbo e é usado assim: "Eu preciso instanciar uma instância deste prefab."
- **Herança:** Isso não significa a mesma coisa que herança de programação padrão. Neste caso, o termo herança se refere à natureza pela qual todas as instâncias de um prefab estão vinculadas ao próprio prefab. Isso será abordado com mais detalhes futuramente.

6.2 Trabalhando com Prefabs

Criar, editar e instanciar um prefab são ações bastante simples como veremos a seguir.

Criando um prefab

Existem dois métodos simples para criar um Prefab. O primeiro consiste Como todos os outros assets, você deseja começar criando uma pasta de Prefabs na Project View para armazená-los. Em seguida, clique com o botão direito na pasta recém-criada e selecione *Create > Prefab* como na figura 6.1. Dê o nome de *Ball* para este Prefab. No futuro ele será a bola de nosso jogo.

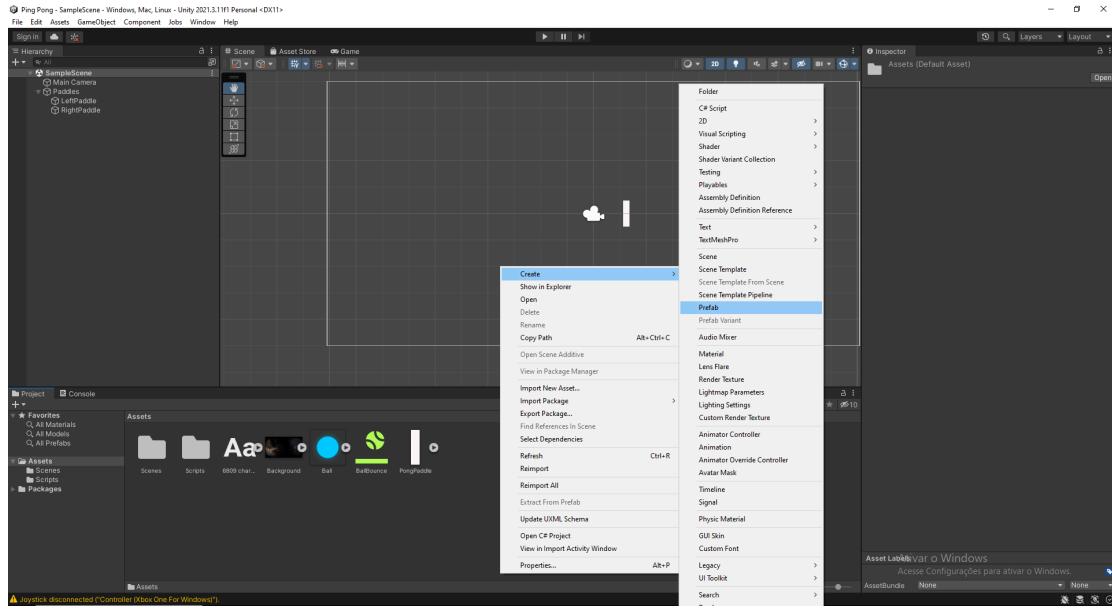


Figura 6.1 Criando um Prefab.

O outro método é mais simples e consiste em criar um prefab a partir de um *GameObject* já existente. Basta arrastar qualquer *GameObject* da Hierarquia para a *Project View*. Fazendo isso, você criará e preencherá o prefab ao mesmo tempo.

Você já deve ter notado que a maioria das alterações que são feitas para a barra de um dos jogadores deve ser aplicada para ambas as barras, por isso vamos criar um prefab para a barra. Clique e arraste o *GameObject* de uma das barras para a pasta *Prefabs* na *Project View*. Essa ação cria um prefab baseado nas barras. Altere o nome do prefab para “paddle” apenas. Note que na Hierarquia o nome da barra que serviu para a criação do prefab da barra agora está azul, é assim que identificamos os *GameObjects* que são prefabs na Hierarquia.

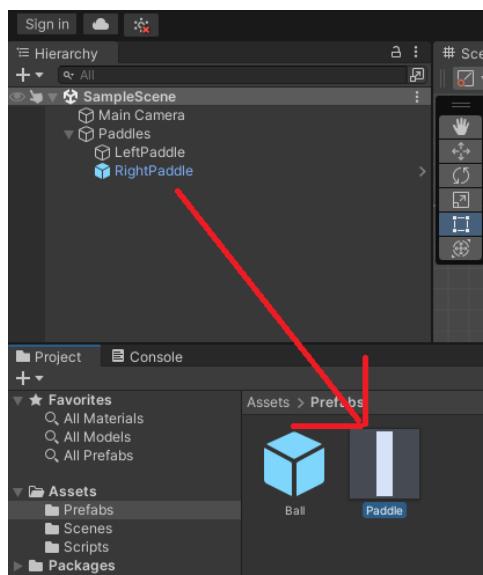


Figura 6.2 Prefab Paddle.

Adicionando uma instância de um Prefab à Cena

Uma vez que um prefab é criado, ele pode ser adicionado quantas vezes você quiser a uma cena ou a qualquer número de cenas em um projeto. Para adicionar uma instância de um prefab a uma cena, tudo que você precisa fazer é clicar e arrastar o prefab da *Project View* para o local desejado na view da Cena, ou na Hierarquia.

Se você arrastar para a Cena, ele será instanciado onde você arrastar. Se você arrastar para uma parte vazia da Hierarquia, sua posição inicial será o que estiver definido no prefab. Se você arrastar sobre outro objeto na Hierarquia, o prefab se tornará um filho desse objeto.

Arraste o Prefab da bola para o centro da cena. Vamos adicionar uma imagem ao prefab da bola para ajudar na visualização. No inspetor clique em *Add Component > Rendering > Sprite Renderer*.

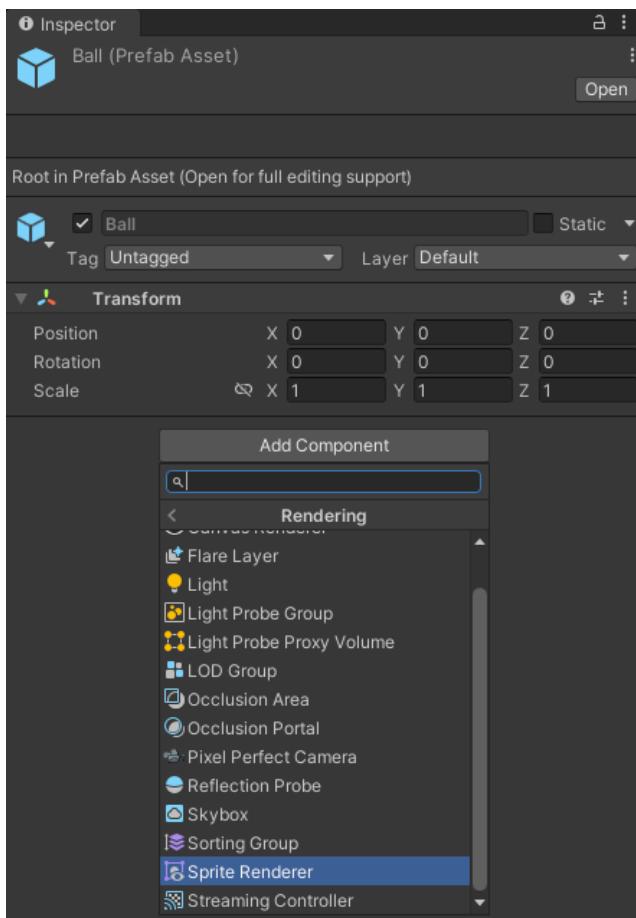


Figura 6.3 Sprite Renderer.

Clique no link à esquerda do campo *Sprite* e selecione a Imagem *Ball*, como mostra a figura 6.4.

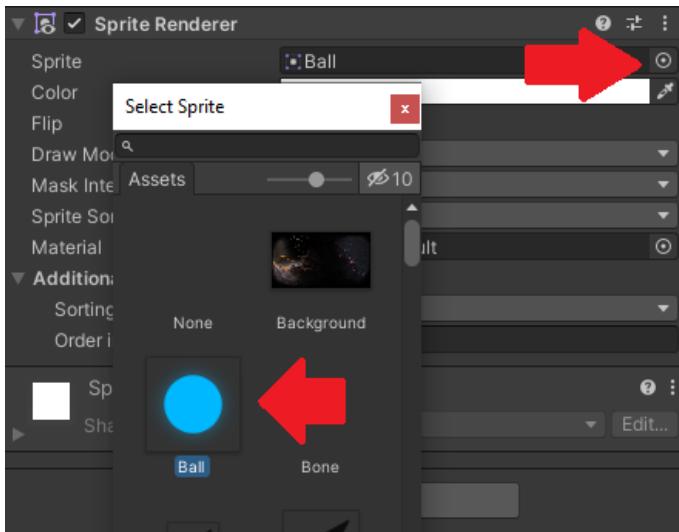


Figura 6.4 Adicionando uma imagem.

Atualmente, somente uma das barras está herdando do prefab paddle. Na Hierarquia, clique no nome da barra que não é o prefab, ou seja, o nome que não está azul, e pressione a tecla delete do teclado. Após isso arraste o prefab da barra para dentro do *GameObject Paddles* e renomeie a nova barra com o nome correspondente a barra que está faltando e mova a barra na cena para seu lado. A Hierarquia deve estar como indica a figura 6.5.

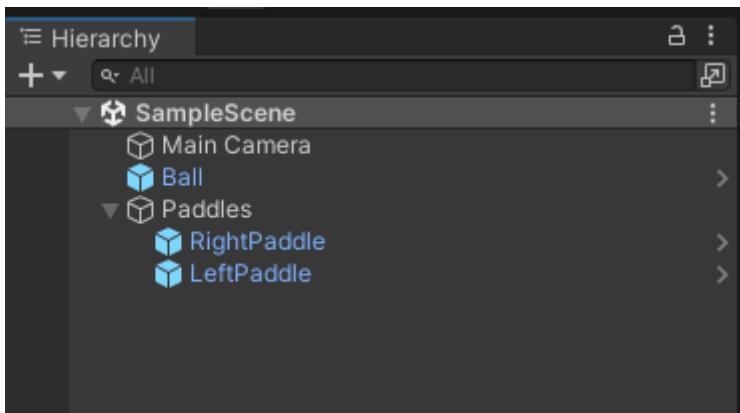


Figura 6.5 Hierarquia com os Prefabs das barras.

Vamos fazer um experimento para ver como os prefabs são poderosos. Clique uma vez no prefab das barras na *Project View* e olhe para o *Inspector*. Repare que ele mostra transformações e uma lista de componentes tal qual um *GameObject*. No componente *Sprite Renderer* existe um campo *Color* com uma barra branca, ao clicar nessa barra branca vai surgir uma janela para selecionar uma cor, com essa ferramenta é possível alterar a cor de uma barra. Altere a cor do prefab e note que na cena, ou na *Game View*, ambas as barras mudam de cor. Esse é o poder da herança dos prefabs. Retorne para a cor branca.

Instâncias de Prefabs com componentes diferentes

Muitas vezes precisaremos que instâncias de um mesmo prefab possuam atributos de componentes diferentes. Já temos um exemplo em nosso jogo, pois as barras precisam ter o atributo *Player Number* diferentes. Felizmente esse problema é fácil de resolver pois as alterações feitas nas instâncias não são refletidas no Prefab. Experimente alterar o *Player Number* de uma das barras. Repare que o atributo *Player Number* da instância que foi alterada está em negrito, é assim que a Unity indica que um atributo está diferente do prefab.

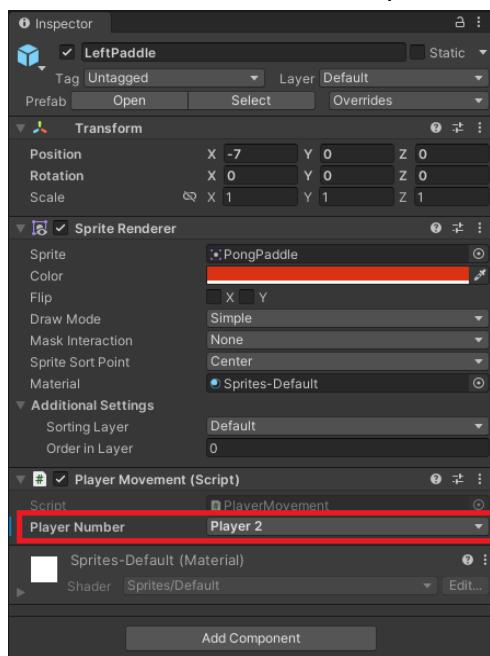


Figura 6.6 Atributo diferente do prefab.

Instanciando Prefabs por meio de Código

Colocar instâncias de Prefab em uma cena é uma ótima maneira de construir um nível consistente e planejado. Às vezes, no entanto, você deseja criar instâncias em tempo de execução. Talvez você queira que os inimigos reapareçam, ou que sejam colocados aleatoriamente. Também é possível que você precise de tantas instâncias que colocá-las manualmente não seja viável. Seja qual for o motivo, instanciar prefabs por meio de código é uma boa solução.

Existem duas maneiras de instanciar GameObjects de prefab em uma cena e ambas usam o método `Instantiate()`. A primeira maneira é usar `Instantiate()` assim:

```
Instantiate(GameObject prefab);
```

Como você pode ver, este método simplesmente recebe uma variável de `GameObject` e cria um objeto com base nela. A localização, rotação e escala do novo objeto são as mesmas do prefab na visualização do Projeto. A segunda maneira de usar o método `Instantiate()` é assim:

```
Instantiate(GameObject prefab, Vector3 position, Quaternion rotation);
```

Este método requer três parâmetros. O primeiro ainda é o *GameObject* para fazer uma cópia. Os segundo e terceiro parâmetros são a posição e rotação desejadas do novo objeto. Note o parâmetro do tipo *Quaternion*. Como foi ensinado no capítulo 3, a rotação é armazenada em algo chamado *Quaternion*, estrutura matemática que a Unity armazena informações de rotação. A verdadeira aplicação do *Quaternion* está além do escopo deste material. O método *Instantiate()* vai ser bastante útil quando futuramente quando precisarmos recolocar no centro da tela.

8. Motor de Física

Na Unity, a física 2D é uma parte essencial do desenvolvimento de jogos 2D, permitindo que os objetos do jogo interajam de forma realista dentro de um ambiente bidimensional. Neste capítulo começamos aprendendo o que são os *rigidbody2D* e o que eles podem fazer por você. Você também aprenderá sobre colisões e os usos mais sutis da colisão com *triggers*. Por fim falaremos sobre Materiais físicos.

7.1 Rigidbody2D (Corpo rígido 2D)

Um *Rigidbody2D* é um componente que adiciona física 2D aos *GameObjects*. Ele permite que esses objetos se comportem de acordo com as leis da física 2D, incluindo movimento, colisões e interações com forças externas. Vamos adicionar um componente do tipo *Rigidbody2D* ao prefab das barras selecionando-o, clicando em *Add Component > Physics 2D > Rigidbody 2D*.

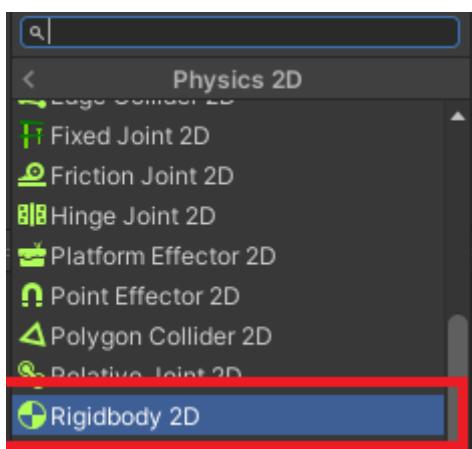


Figura 7.1 Adicionando Rigidbody2D.

Aperte o *Play* e veja a diferença. As barras caem. Isso acontece devido a gravidade que o sistema de física impõe aos *Rigidbody2D*. Como não queremos esse efeito iremos anular a gravidade das barras, para isso altere o valor do campo *Gravity Scale* do *Rigidbody 2D* para 0.

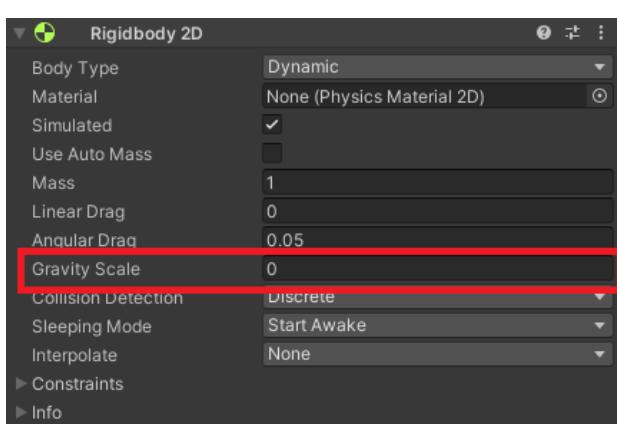


Figura 7.2 Alterando Gravity Scale para 0.

Agora as coisas vão ficar mais interessantes pois com o *Rigidbody2D* podemos fazer os *GameObjects* se moverem, e jogos digitais costumam possuir objetos em movimento.

Para isso utilizaremos um método chamado *GetComponent()* que nos permite manipular os componentes dos *GameObjects* através de código. Primeiro iremos criar a variável que irá armazenar nosso *Rigidbody2D*:

```
private Rigidbody2D rb;
```

Após isso acrescentaremos a seguinte linha de código no nosso método Start:

```
rb = GetComponent<Rigidbody2D>();
```

Quando utilizamos o método *GetComponent* precisamos especificar entre < e > o tipo de componente, no nosso caso um *Rigidbody2D*.

Vamos criar também uma variável para podermos definir no inspetor a velocidade das barras.

```
public float speed = 5f;
```

Agora podemos fazer a barra se mover utilizando o atributo *velocity* da classe *Rigidbody2D*. Este atributo é do tipo *Vector2* em que seus valores representam respectivamente a velocidade horizontal e vertical do *GameObject* que possui o *Rigidbody2D*.

Caso você tenha optado por utilizar o *Input.GetAxis*, você pode mover a barra para cima e para baixo reescrevendo o método *MovePlayer* da seguinte forma:

```
private void MovePlayer()
{
    float direction = Input.GetAxis(playerAxis);
    rb.velocity = Vector2.up * direction * speed;
}
```

Caso tenha optado por usar o *Input.GetKeys*, você pode mover a barra para cima e para baixo reescrevendo o método *MovePlayer* da seguinte forma:

```
private void MovePlayer()
{
    if(Input.GetKey(upMovementButton))
    {
        rb.velocity = Vector2.up * speed;
    }
}
```

```

else if(Input.GetKey(downMovementButton))
{
    rb.velocity = Vector2.down * speed;
}
else
{
    rb.velocity = Vector2.zero;
}
}

```

Agora vamos usar o *Rigidbody2D* para Mover a Bola. Adicione um *Rigidbody2D* ao Prefab *Ball* e altere o valor do campo *Gravity Scale* para 0.

Crie um Script chamado *BallMovement* ao prefab ball e abra-o no IDE que você estiver utilizando. Apague o método *Update*, não precisaremos dele. Crie um método do tipo void chamado *LaunchBall*. Queremos que a bola ande em uma direção aleatória, para isso utilizaremos o método *Random.Range*. Ele aceita dois argumentos numéricos do mesmo tipo e retorna um valor aleatório maior ou igual ao primeiro e menor que o segundo.

```

public float speed = 5f;
private Rigidbody2D rb;

// Start is called before the first frame update
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    LaunchBall();
}

void LaunchBall()
{
    float randomDirection = Random.Range(0, 2) * 2 - 1; // -1 ou 1
    rb.velocity = new Vector2(randomDirection, Random.Range(-1f, 1f)) * speed;
}

```

A primeira linha do nosso método *LaunchBall* utiliza o método *Random.Range* com dois como argumentos, retornando o valor 0 ou 1. Multiplicado por 2 e subtraído de 1, a variável *randomDirection* pode ser 1 ou -1. Essa variável é utilizada para definir a direção horizontal da velocidade da bola na segunda linha, sendo a velocidade vertical definida com a função *Random.Range* retornando um número real maior ou igual a -1 e menor que 1. Após isso multiplicamos esses valores pela variável *speed*.

Dê play repetidas vezes e veja que a bola sempre anda em uma direção aleatória.

7.2 Colisões

No momento, tanto a bola quanto as barras conseguem sair para fora da tela. Além disso, a bola atravessa as barras. Ambos acontecem pois ainda não definimos colisores para eles. Colisão, simplificando, é saber quando a borda de um objeto entrou em contato com outro objeto. Colisores são componentes que definem as superfícies de contato para as colisões e existem em vários formatos. No nosso caso precisamos adicionar colisores retangulares para as barras e um colisor circular para a bola.

No prefab das barras clique em *Add Component > Physics 2D > Box Collider 2D*.

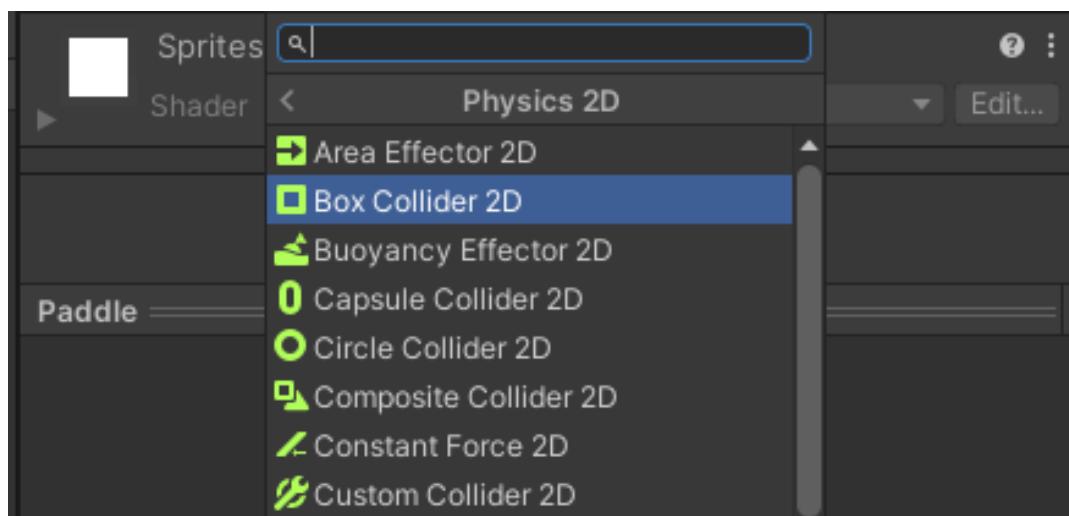


Figura 7.3 Adicionando Box Collider 2D.

Se você prestar atenção notará que ao selecionar uma das barras na hierarquia uma caixa verde aparecerá ao seu redor. Essa caixa é o box collider 2D.

No prefab da bola clique em *Add Component > Physics 2D > Circle Collider 2D*.

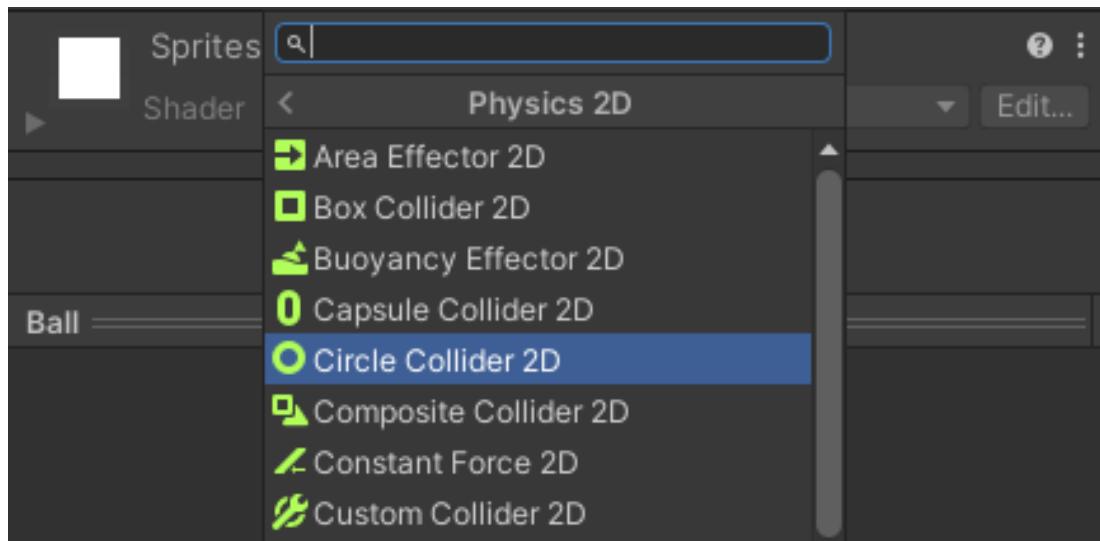


Figura 7.4 Adicionando Circle Collider 2D.

Se você prestar atenção notará que ao selecionar bola na hierarquia um círculo verde aparecerá ao seu redor. Esse círculo é o *circle collider* 2D. O colisor está maior que a bola, precisaremos alterar o seu raio, para isso altere o valor do campo *Radius* do *Circle Collider* 2D no inspetor, como indica a figura 7.5.

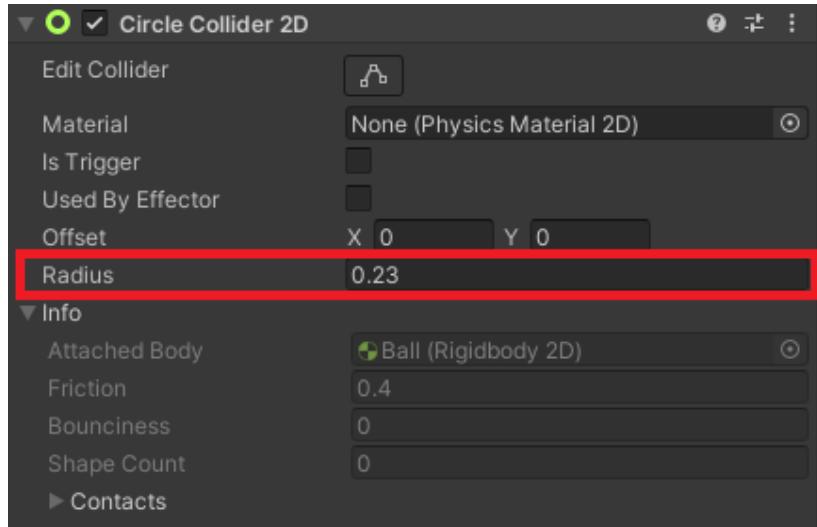


Figura 7.5 Alterando o raio do Circle Collider 2D.

Aperte o *play* e tente fazer a bola colidir com uma das barras. Você deve ter notado algo estranho, as barras começam a rodar quando a bola colide com elas. Isso se deve ao *Rigidbody2D* que possui uma propriedade de torque que permite os corpos rígidos girarem. Para corrigir esse problema vamos fixar sua rotação. Selecione o prefab das barras e procure a seção *constraints* e expanda ela. Marque a opção *Freeze Rotation Z* para impedir que as barras girem. Marque também a opção *Freeze Position X* para garantir que as barras não se movam na horizontal.

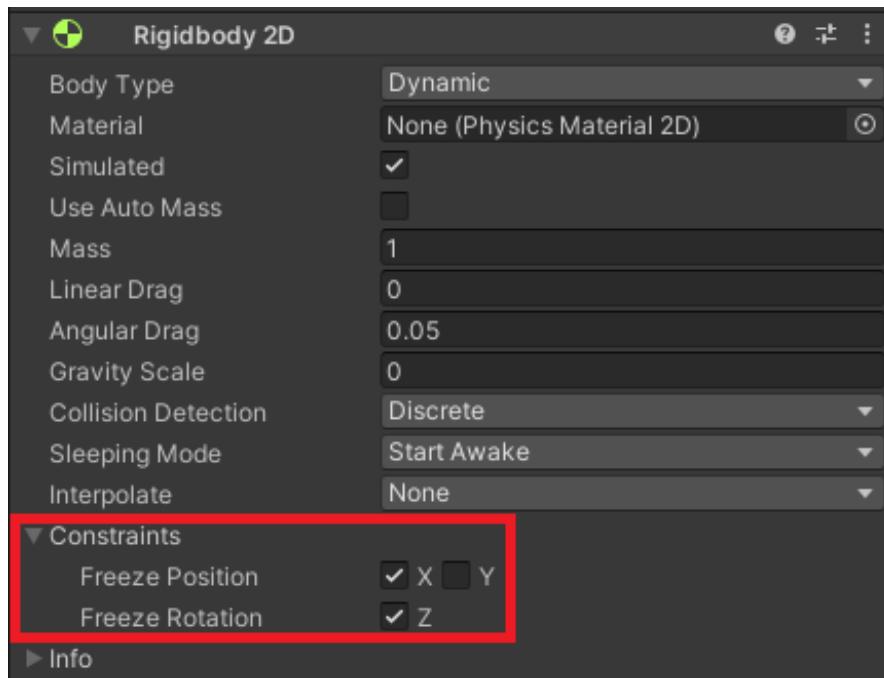


Figura 7.6 Fixando posição X e rotação das barras.

A bola ainda não está se comportando da maneira como queremos, cuidaremos disso mais adiante neste capítulo. Note que a barra não gira mais ao colidir com a bola. Agora faremos com que as barras e a bola não saiam da tela por cima ou por baixo. Crie dois *GameObjects* vazios clicando com o botão direito na hierarquia e clicando em *Create Empty*.

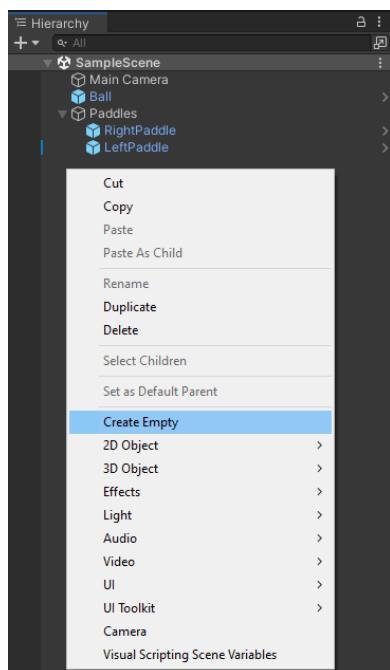


Figura 7.7 Adicionando GameObject vazio.

Em cada um adicione um *Box Collider 2D*. Posicione um acima e um abaixo da câmera, como indica a figura 7.8.

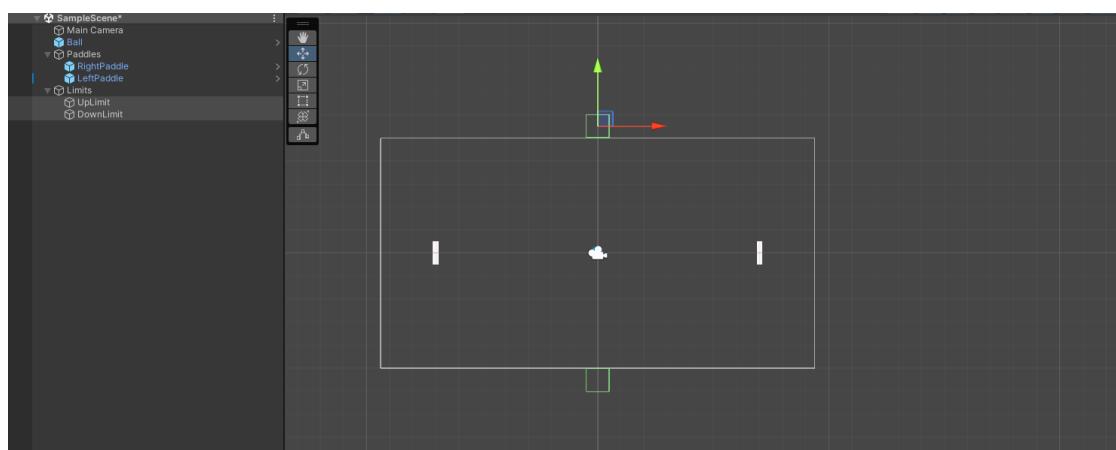


Figura 7.8 Posicionando GameObjects com colisores.

Os colisores não cobrem todo o comprimento da câmera, para resolver isso vamos alterar o tamanho dele, no inspector altere o campo *Size X* do *Box Collider 2D* para cobrir todo o comprimento da câmera, como indica a figura 7.9.

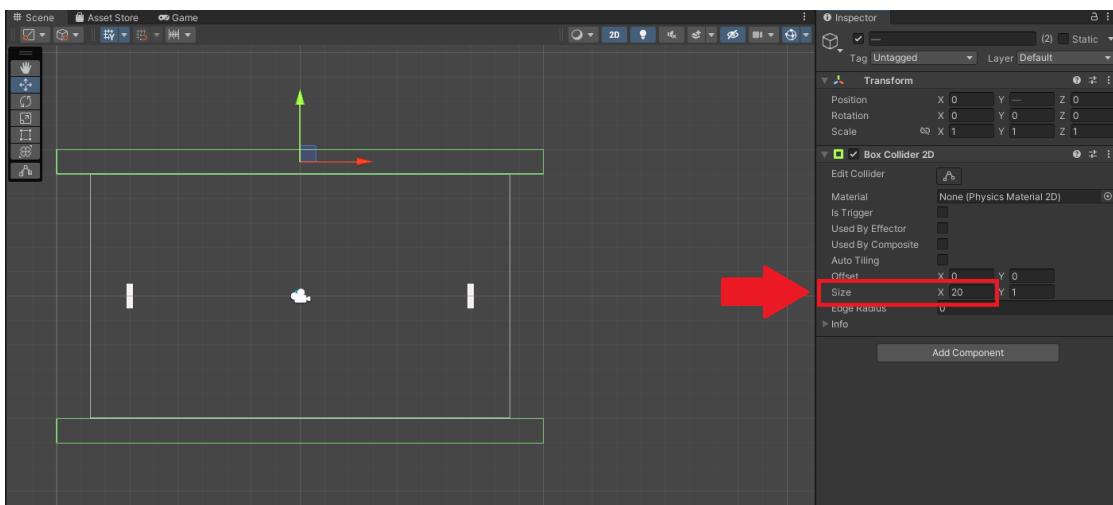


Figura 7.9 Redimensionando colisores.

Agora nem as barras e nem a bola devem conseguir sair para fora da tela por cima ou por baixo.

7.3 Trigger

Muitas vezes necessitamos verificar quando um *GameObject* colide ou entre em uma determinada área, mas sem a resposta física de uma colisão normal. Quando este é o caso, utilizamos os *Triggers*. Um *trigger* é um tipo de colisor que é configurado para detectar quando outro objeto entra ou sai de sua área de detecção, mas sem causar uma resposta física. Em vez disso, os *triggers* chamam três métodos específicos que permitem ao programador determinar o que a colisão significa:

```
void OnTriggerEnter2D(Collider2D other)
void OnTriggerStay2D(Collider2D other)
void OnTriggerExit2D(Collider2D other)
```

Usando esses métodos, podemos definir o que acontece sempre que um objeto entra, permanece ou sai do colisor. Por exemplo, se quisermos escrever uma mensagem no console sempre que um objeto entrar no perímetro de um retângulo, poderíamos adicionar um trigger ao retângulo. Em seguida, adiciona um script ao retângulo com o seguinte código:

```
void OnTriggerEnter2D(Collider2D other)
{
    print("Objeto entrou no colisor");
}
```

Você deve estar se perguntando o que é o parâmetro *other* do tipo *Collider2D*. Ele representa o objeto que entrou no perímetro do trigger. Poderíamos por exemplo exibir o nome desse objeto no console usando o seguinte código:

```

void OnTriggerEnter2D(Collider2D other)
{
    print(other.gameObject.name + " entrou no colisor");
}

```

Podemos ser ainda mais ousados e utilizar o método *Destroy* para eliminar o objeto da cena com o seguinte código:

```

void OnTriggerEnter2D(Collider2D other)
{
    Destroy(other.gameObject);
}

```

Destroy é um método que elimina da cena aquilo que lhe é dado como parâmetro, geralmente utilizado para eliminar *GameObjects*, como no nosso exemplo.

Para nosso jogo, quando a bola toca em alguma das laterais da câmera ela deve ser destruída, o placar do jogo deve ser atualizado e uma nova bola deve surgir no centro da tela.

Adicione dois *GameObjects* vazios e coloque um na direita da câmera e o outro na esquerda da câmera. Adicione em ambos um *Box Collider 2D* e altere o Campo *Size Y* para que cubra as laterais da câmera, como indica a figura 7.10:

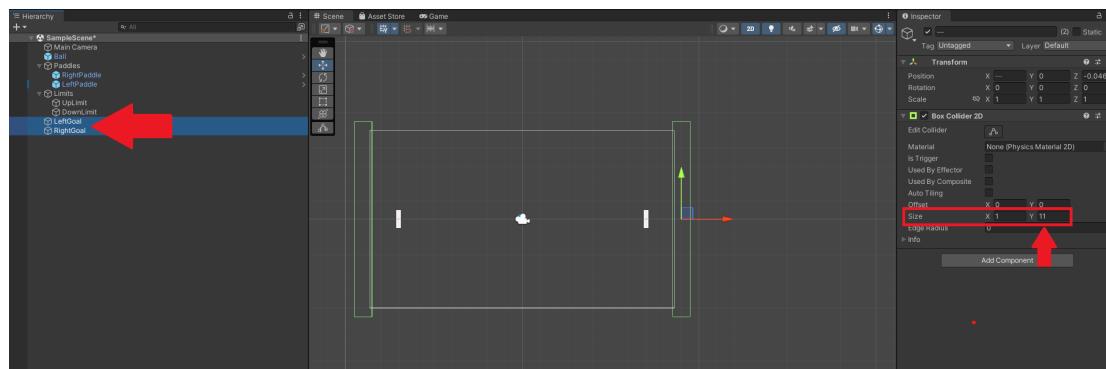


Figura 7.10 Adicionando Colisores nas laterais.

Para transformar os *Box Collider 2D* em *triggers* basta marcar o campo *Is Trigger*, como indicado na figura 7.11.

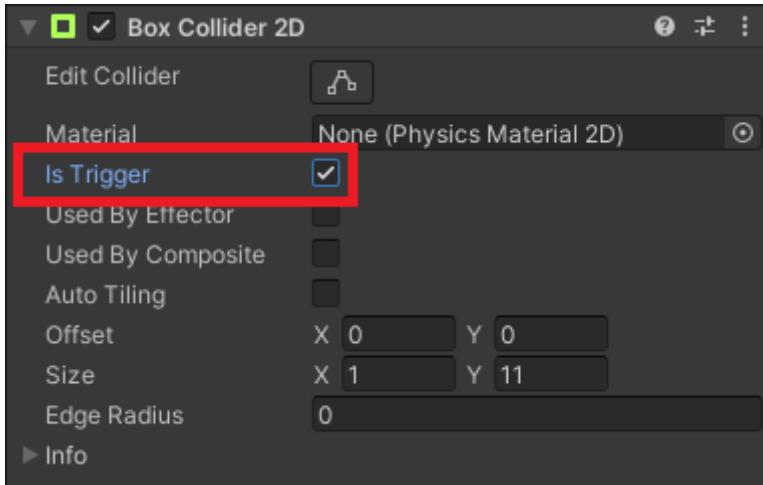


Figura 7.11 Campo Is Trigger.

Adicione aos *GameObjects* dos *triggers* um script chamado *Goal*. Apague os métodos *Update* e *Start*, não precisaremos deles. Vamos criar um enum chamado *PlayerPoint* com os valores Esquerdo e Direito para indicar qual jogador está pontuando em cada lado. Também faremos algo novo, criar uma variável do tipo *GameObject* para guardar o prefab da bola que vamos instanciar quando um dos lados pontuar. Também criaremos uma variável inteira chamada *score* para guardar quantos pontos cada jogador tem.

```
public enum PlayerPoint
{
    Esquerdo,
    Direito
}

public PlayerPoint playerPoint;

public GameObject ball;

private int score = 0;
```

Agora utilizaremos o método *OnTriggerEnter2D* para verificar quando a bola entra na área do *trigger*. Ao entrar na área do *trigger* nós queremos tirar a bola da cena, atualizar a variável *score* e verificar se o *score* é igual a 5, se for igual a 5 vamos mostrar no console uma mensagem dizendo que o jogador venceu, caso contrário mostraremos no console a pontuação atualizada do jogador e instanciaremos uma nova bola. Veja o código abaixo:

```
void OnTriggerEnter2D(Collider2D other)
{
    Destroy(other.gameObject);
```

```

score++;

if(score < 5)
{
    print(score + " pontos para o lado " + playerPoint.ToString());
    Instantiate(ball, Vector3.zero, Quaternion.identity);
}
else
{
    print("Vitória do lado " + playerPoint.ToString());
}

}

```

Você deve ter duas dúvidas quanto a esse código. Primeiro é o método *ToString*. Este é um método presente em vários tipos de dados no C# e como o nome sugere ele transforma o conteúdo em *string*. Outra dúvida deve ser quanto ao *Quaternion.identity*. Usamos esse comando para indicar uma rotação nula, e usamos no nosso código pois queremos instanciar nossa bola sem rotação.

Para cada *GameObject* selecione o *Player Point* correto no *Inspector* e selecione o prefab da bola no campo *ball* como mostra a figura 7.12:

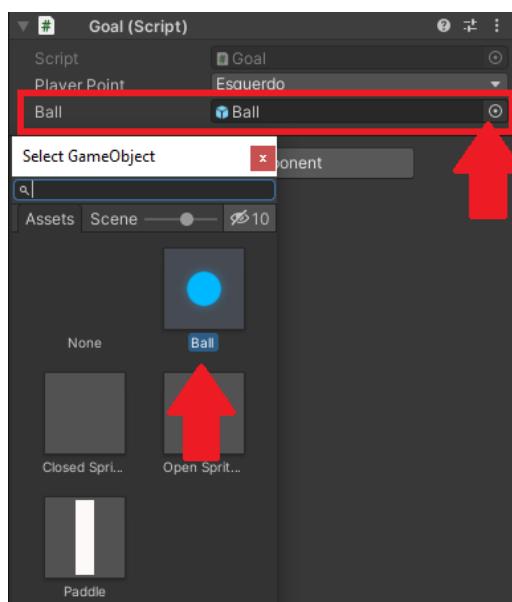


Figura 7.12 Adicionando Prefab Ball ao campo Ball do script Goal.

Agora só resta apertar o *play* e ver se tudo está funcionando como deveria.

Importante: para funcionarem corretamente, os *triggers* necessitam que ao menos um dos *GameObjects* participantes da colisão possua um *RigidBody2D*.

7.4 Physics Material 2D

Materiais físicos, ou *Physics Materials*, podem ser aplicados a colisores para conferir propriedades físicas variadas aos *GameObjects*. Por exemplo, você pode usar o material de borracha para tornar um objeto elástico ou um material de gelo para torná-lo escorregadio. Você até mesmo pode criar o seu próprio para simular um material específico de sua escolha.

Queremos que a bola queique nas barras e nas bordas, então vamos acrescentar ao *Circle Collider 2D* da bola um *Physics Material 2D* com a propriedade *bounce*. Vamos criar uma pasta na *Project View* chamada *Materials*. Dentro da pasta clique com o botão direito e no menu clique em *Create > 2D > Physics Material 2D* e dê o nome de *Ball Bounce*.

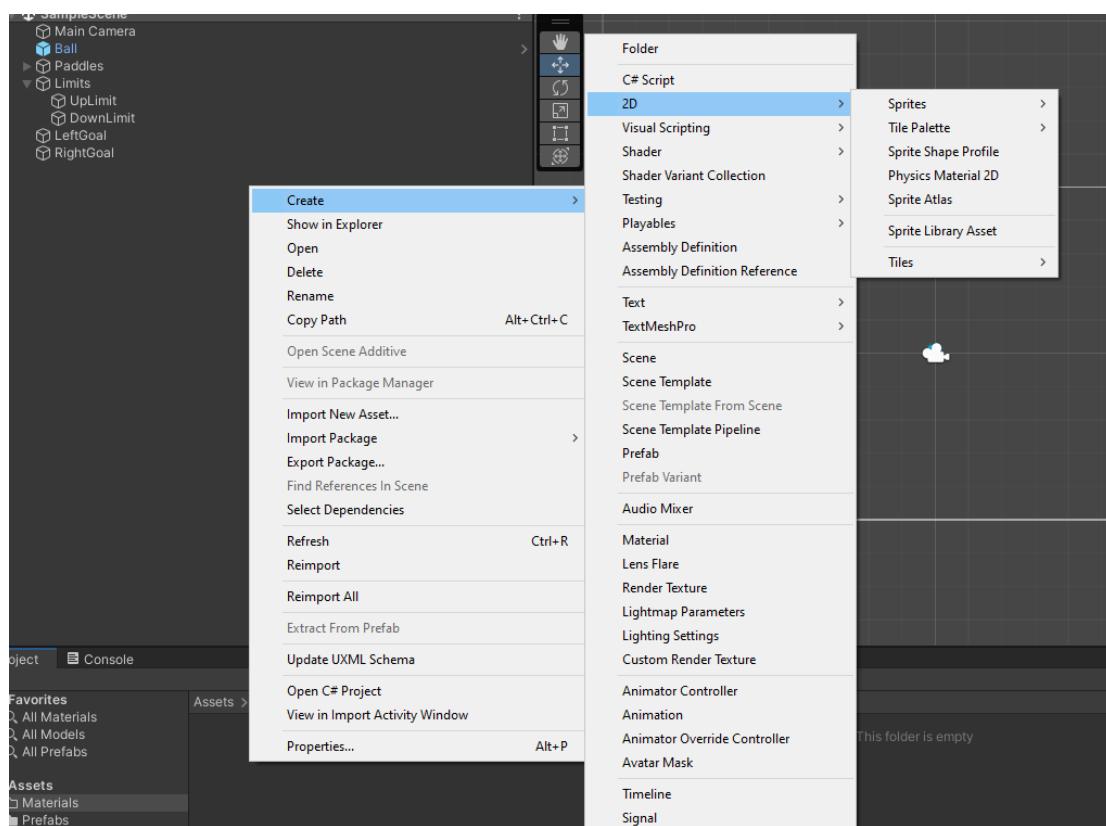


Figura 7.12 Criando um Physics Material 2D.

Olhe para o inspector ao selecionar *Ball Bounce*. Você verá duas propriedades, *friction* e *bounciness*. *Friction* define quanto de atrito o colisor aplica a outros colisores. *Bounciness* define o quão outros colisores serão repelidos ao colidir com seu colisor, exatamente o que precisamos.

Altere o valor de *Bounciness* para 1, isso fará com que a bola altere sua trajetória mantendo o módulo de sua velocidade ao colidir com as barras ou as bordas, e altere o valor de *friction* para 0, isso fará com que a bola não perca velocidade ao colidir com as barras ou as bordas.

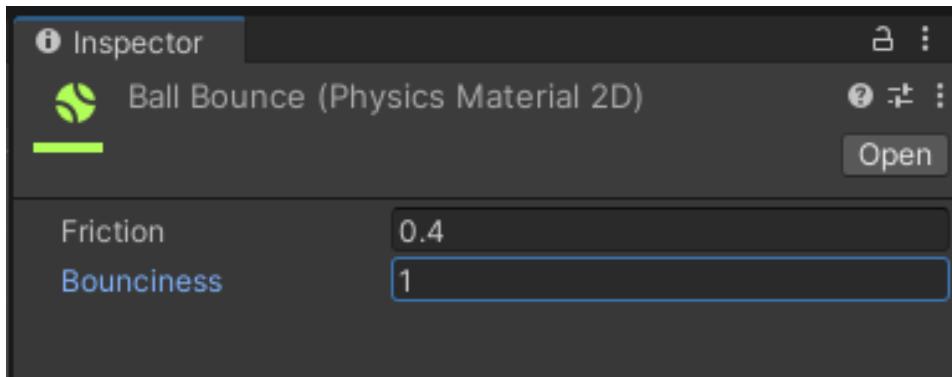


Figura 7.13 Physics Material.

Selecione o prefab da bola, altere o campo *Material* do *Circle Collider 2D* e selecione *Ball Bounce* como indica a figura 7.14.

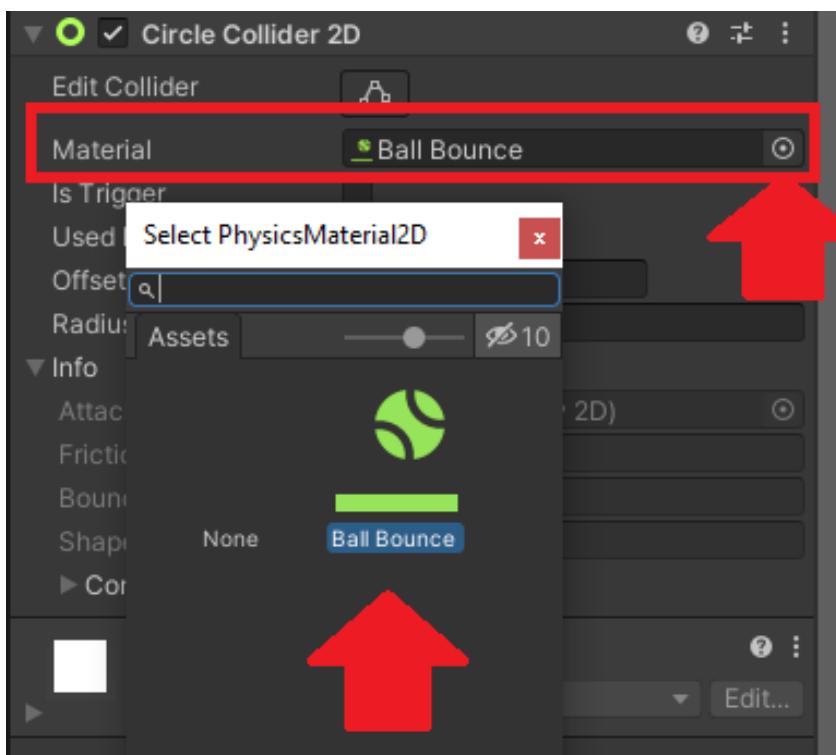


Figura 7.14 Alterando o Physics Material 2D de um Collider 2D.

Aperte o *play* e teste o resultado.

7.5 Adiando a Chamada de uma Função

O que vamos cobrir nessa parte não envolve física, mas é o momento ideal para falarmos sobre. No momento sempre que a bola aparece ela é automaticamente lançada, porém o ideal seria que ela esperasse alguns segundos para ser lançada. Para resolver isso vamos usar o método *Invoke* que chama uma função uma quantidade de segundos depois que ele é chamado.

Altere o método *Start* de *BallMovement* para o código abaixo:

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    Invoke("LaunchBall", 3f);
}
```

Este código chama a função *LaunchBall* depois de 3 segundos. Aperte o *play* e teste seu jogo.

9. Interface de Usuário

Interfaces de usuário, comumente referidas como UIs (do inglês *User Interface*), são elementos especiais de um jogo que existem para fornecer informações ao usuário e aceitar entradas simples do usuário. Essas informações e entradas muitas vezes chamadas na indústria de HUD (*Heads Up Display*) são desenhadas sobre o seu jogo.

8.1 Canvas

Na Unity, todos os *GameObjects* referentes a interface de usuário devem ser filhos de um *GameObject* especial chamado *Canvas*. O *canvas* representa a tela do jogador, todos seus filhos não costumam aparecer no mundo do jogo, eles ficam “grudados” na tela do jogador.

Adicionar um *canvas* a uma cena é muito fácil. Você pode adicionar um simplesmente clicando com o botão direito do mouse na hierarquia e depois em *GameObject* > *UI* > *Canvas*. Uma vez adicionado à cena, você está pronto para começar a construir o restante da UI.

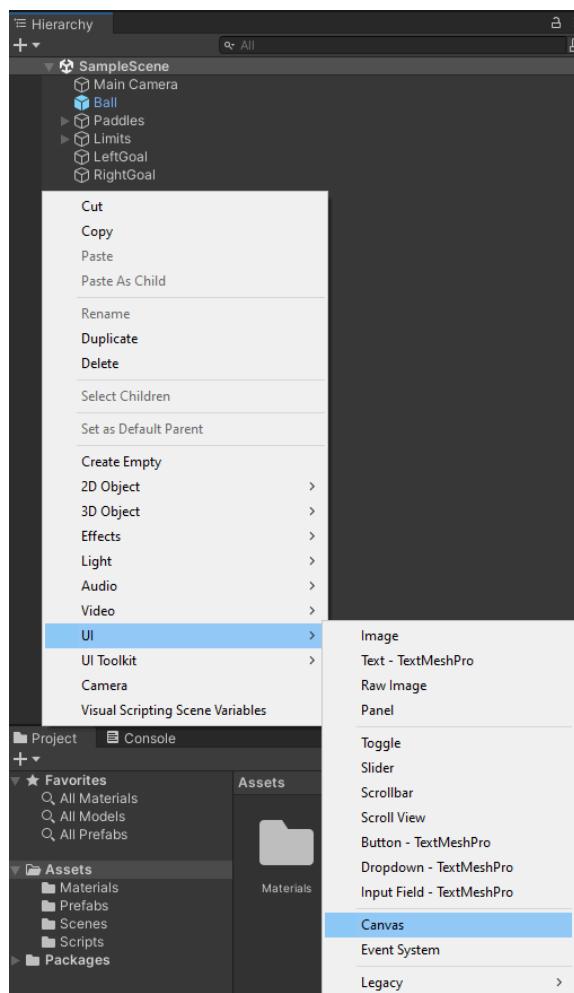


Figura 8.1 Adicionando um Canvas à Cena.

EventSystem: você deve ter notado que ao adicionar um *canvas*, a Unity adicionou automaticamente um *GameObject* chamado *EventSystem*. Esse *GameObject* controla o *canvas* e faz os elementos de interface funcionarem corretamente, portanto não o delete.

8.2 Rect Transform

Você notará que o *canvas* (e todos os elementos de UI) possuem este *Rect Transform* em vez do *Transform* normal com o qual você está familiarizado.

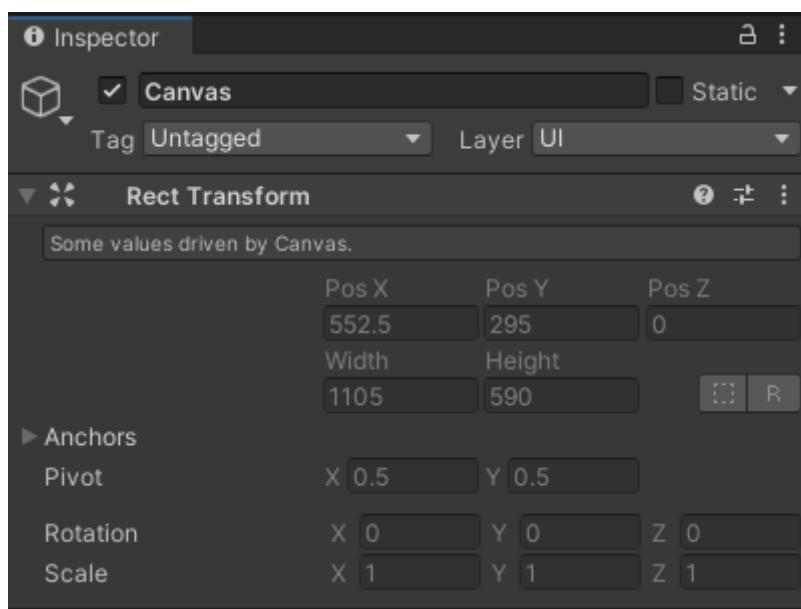


Figura 8.2 Rect Transform.

O *Rect Transform* é um componente crucial na Unity quando se trata de trabalhar com interfaces de usuário. Ele é uma extensão do componente *Transform* utilizado para objetos 2D e UI, fornecendo funcionalidades adicionais específicas para *layout* de UI.

No caso do *canvas* que você criou anteriormente, você notará que o *Rect Transform* está completamente acinzentado. Isso ocorre porque, em sua forma atual, o *canvas* deriva seus valores inteiramente da *GameView* (e, por extensão, da resolução e proporção de qualquer dispositivo em que seu jogo seja executado). Isso significa que a *canvas* sempre ocupará toda a tela. Uma boa prática é garantir que a primeira coisa a fazer ao construir uma UI seja selecionar uma proporção de *aspect ratio* alvo para trabalhar.

8.3 TextMesh Pro

É um sistema de textos para interfaces de usuário utilizado em versões mais recentes da Unity. Ele oferece maior controle, qualidade e desempenho em comparação com o sistema de texto padrão da Unity. Ele é amplamente utilizado para criar textos nítidos e

estilizados em interfaces de usuário (UI), HUDs, e até mesmo dentro do ambiente de jogo.

Para adicionar um *Texto Mesh Pro* clique com o botão direito do mouse no canvas e clique em UI > *Text - TextMeshPro* como indicado na figura 8.3.

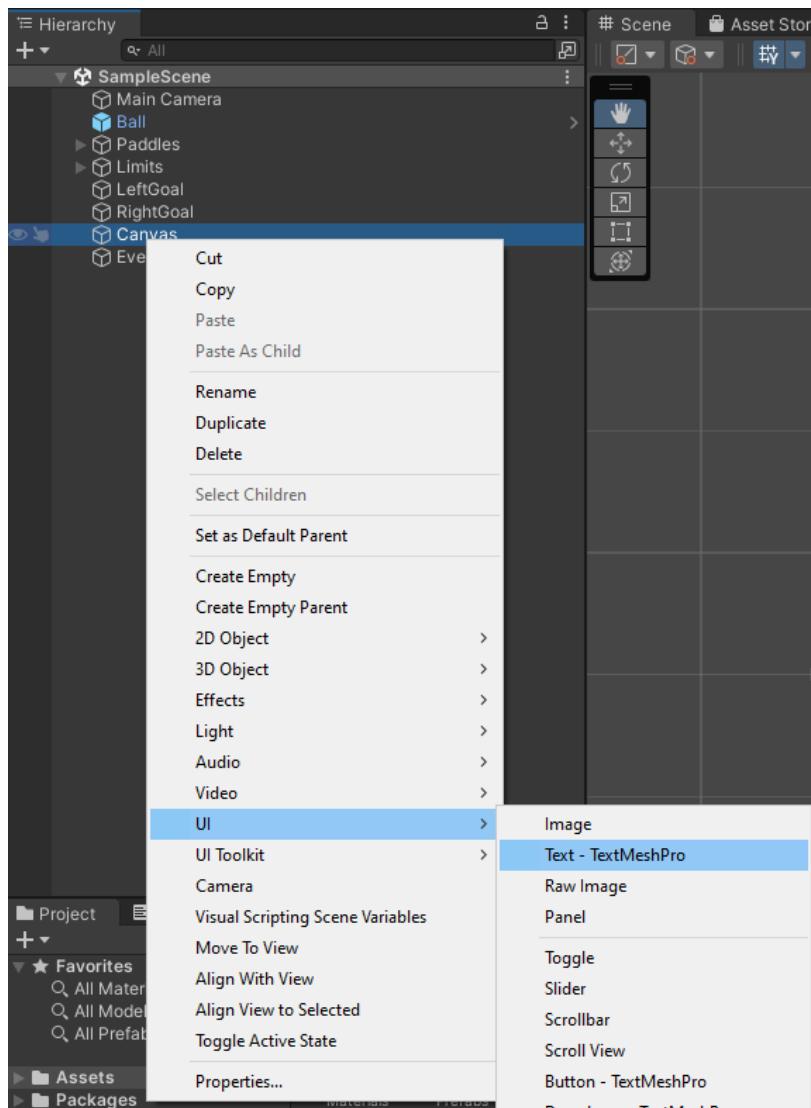


Figura 8.3 Adicionando um TextMeshPro.

Ao fazer isso a janela da figura 8.4 irá aparecer, clique em Import TMP Essentials e aguarde a importação.

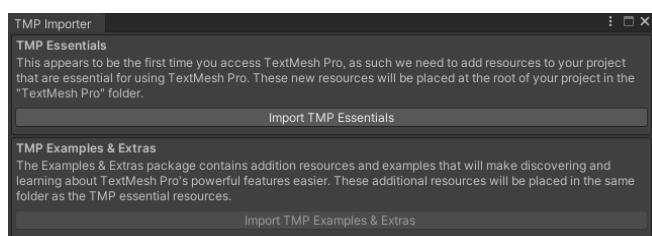


Figura 8.4 Importando Text Mesh Pro.

Na scene View você deverá ver o texto “New Text” no centro, como na Figura 8.5.

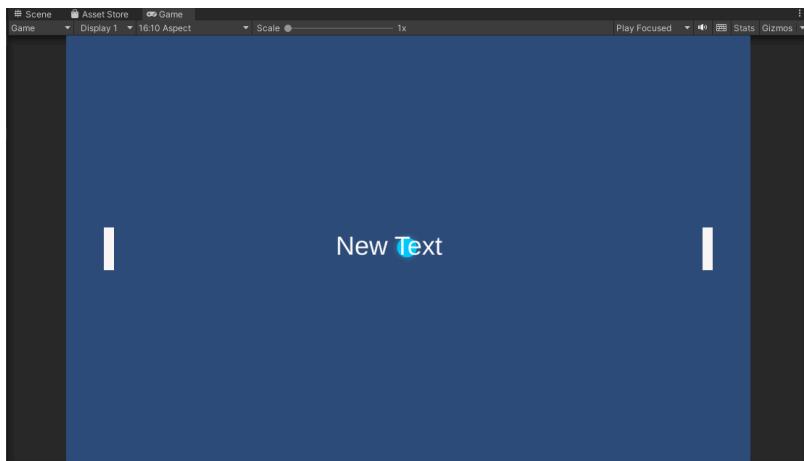


Figura 8.5 Adicionando um Canvas à Cena.

Ao selecionar o texto na hierarquia o componente da figura 8.6 surge no *inspector*.

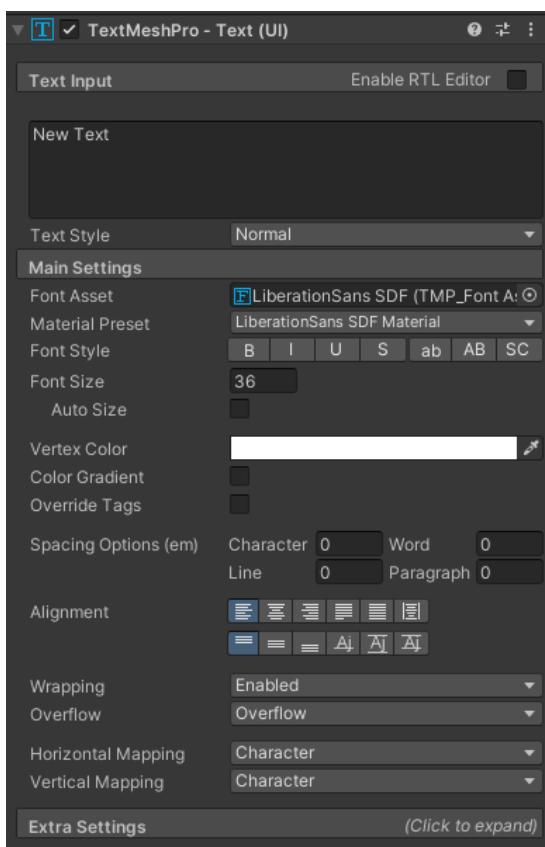


Figura 8.6 TextMeshPro.

Principais campos:

Text: Campo onde você insere o texto que deseja exibir. Suporta *rich text* com *tags*.

Font Asset: Permite selecionar a fonte que será usada para renderizar o texto. Fontes podem ser geradas e configuradas no *Font Asset Creator*.

Font Style: Configurações de estilo como Normal, Negrito, Itálico, Negrito-itálico, etc.

Font Size: Tamanho da fonte para o texto.

Auto Size: Permite que o tamanho da fonte seja ajustado automaticamente com base no espaço disponível.

Vertex Color: Define a cor do texto.

Alignment: Alinha o texto dentro de seu container (esquerda, centro, direita, justificado etc.). Também define as margens ao redor do texto dentro de seu container.

Space Options:

Line: Ajusta o espaçamento entre as linhas de texto.

Character: Ajusta o espaçamento entre os caracteres.

Word: Ajusta o espaçamento entre as palavras.

Paragraph: Ajusta o espaçamento entre parágrafos.

Utilizaremos esse texto para indicar a pontuação da barra da esquerda.

Digite “0” no campo Text e ajuste um tamanho agradável em *Font Size*. No Rect Transform utilize as propriedades Pos X e Pos Y para ajustar a posição do texto acima da barra da esquerda.

Crie um *Texto Mesh Pro* e repita o processo, porém desta vez coloque o texto acima da barra da esquerda.

Deixe a aparência da cena o mais próximo da figura 8.7:

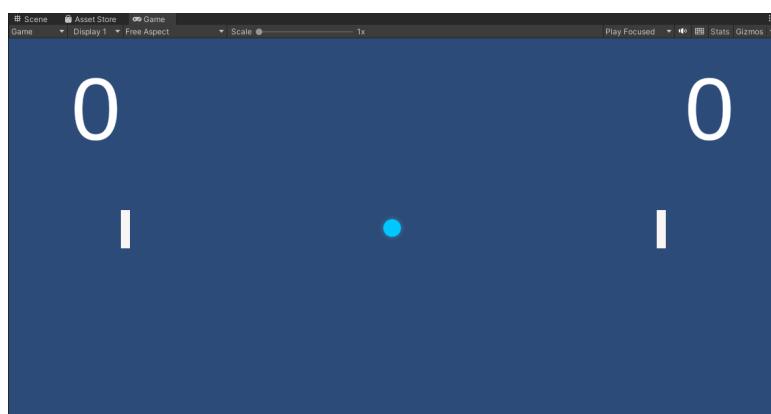


Figura 8.7 Cena com Textos.

É possível alterar o texto de um *Text Mesh Pro* de forma dinâmica utilizando código, veja um exemplo abaixo.

```
using TMPro;
using UnityEngine;

public class Example : MonoBehaviour
{
    public TextMeshProUGUI textMeshPro;

    void Start()
    {
        // Configurações básicas
        textMeshPro.text = "Hello, World!";
    }
}
```

Note que para utilizar o *Text Mesh Pro* em código precisamos adicionar a linha:

```
using TMPro;
```

Para manipular o *Text Mesh Pro* utilizamos a classe *TextMeshProUGUI* e utilizamos sua propriedade *text* para alterar seu texto.

Vamos alterar o código da classe *Goal*. Adicione o *using TMPro;* no início do código e crie uma variável pública do tipo *TextMeshProUGUI* chamada *scoreText*. Altere o método *OnTriggerEnter2D* como indicado no seguinte código:

```
void OnTriggerEnter2D(Collider2D other)
{
    Destroy(other.gameObject);
    score++;
    scoreText.text = score.ToString();

    if(score < 5)
    {
        Instantiate(ball, Vector3.zero, Quaternion.identity);
    }
    else
    {
        print("Vitória do lado " + playerPoint.ToString());
    }
}
```

Note que deixamos de utilizar o método *print* e agora alteramos o texto do *Text Mesh Pro* com o valor da pontuação do jogador.

Retorne ao editor da Unity. No *GameObject* com o script *Goal* do lado esquerdo adicione o *Texto Mesh Pro* do texto do lado direito ao campo *score text* e faça o contrário com o script *Goal* do lado direito. Aperte o *play* e teste. Note que agora as pontuações estão sendo exibidas nos textos.

Agora faremos o mesmo para exibir o jogador que venceu a partida. Adicione outro *Text Mesh Pro* ao centro da cena e apague o texto em seu campo *Text*.

Crie mais uma variável pública do tipo *TextMeshProUGUI* e chame-a de *winText*. Altere o método *OnTriggerEnter2D* novamente como indicado no seguinte código:

```
void OnTriggerEnter2D(Collider2D other)
{
    Destroy(other.gameObject);

    score++;
    scoreText.text = score.ToString();

    if(score < 5)
    {
        Instantiate(ball, Vector3.zero, Quaternion.identity);
    }
    else
    {
        winText.text = "Vitória do lado " + playerPoint.ToString();
    }
}
```

Note que agora informamos o vencedor no *Text Mesh Pro* do campo *winText*.

Retorne ao editor da Unity. Em ambos os *GameObjects* com o script *Goal* adicione o *Texto Mesh Pro* do texto de vitória ao campo *win text*. Aperte o *play* e teste. Note que agora o vencedor está sendo exibido no texto central.



Figura 8.8 Texto com quebra de linha.

Há uma quebra de linha no texto, isso ocorre devido ao tamanho da caixa invisível que guarda o texto. Podemos aumentar sua largura para que o texto utilize apenas uma linha. Para isso basta alterar o campo *Width* do *Rect Transform*. Altere o campo *Width* de *WinText* para 500. Altere o campo *Pos X* para centralizar o texto.

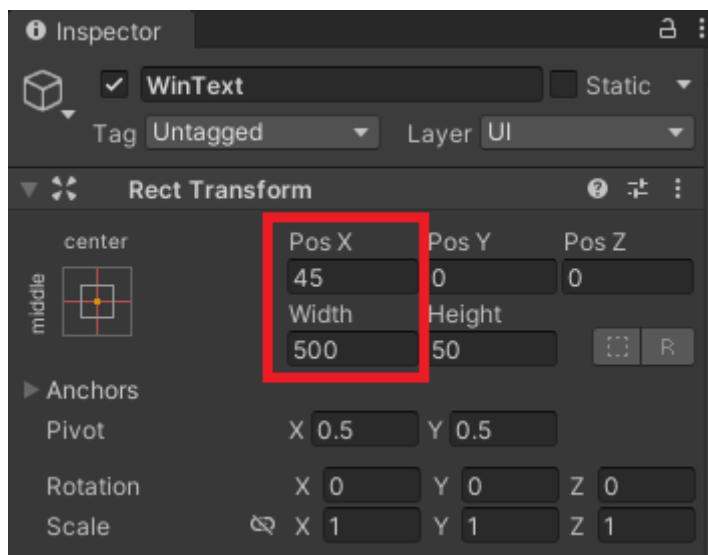


Figura 8.9 Alterações no Rect Transform de WinText.

8.4 Botões

Os botões são componentes essenciais para criar interfaces interativas em seus jogos ou aplicativos. Eles permitem que os usuários executem ações específicas ao clicarem ou tocarem neles.

No momento o jogo inicia com uma bola já posicionada no centro da tela, porém seria mais elegante se existisse um botão para iniciar a partida.

Na hierarquia exclua o *GameObject* da bola e em seguida adicione um botão a cena clicando com o botão direito no Canvas e UI > *Button - TextMeshPro* como na figura 8.10 e renomeie o botão para *StartGameButton*.

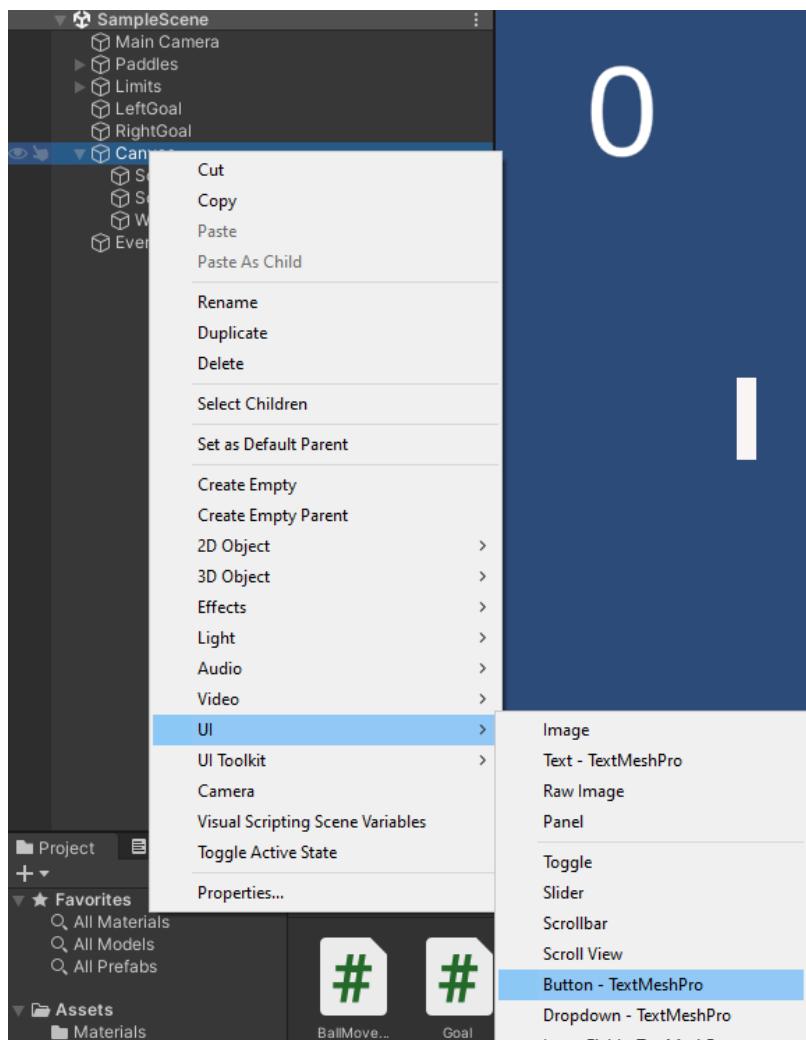


Figura 8.10 Adicionando um botão à cena.

Note que o botão é criado com um *TextMeshPro* aninhado, como mostra a imagem abaixo. Ele é responsável pelo texto que será exibido no botão. Altere seu texto para “Iniciar Jogo”.

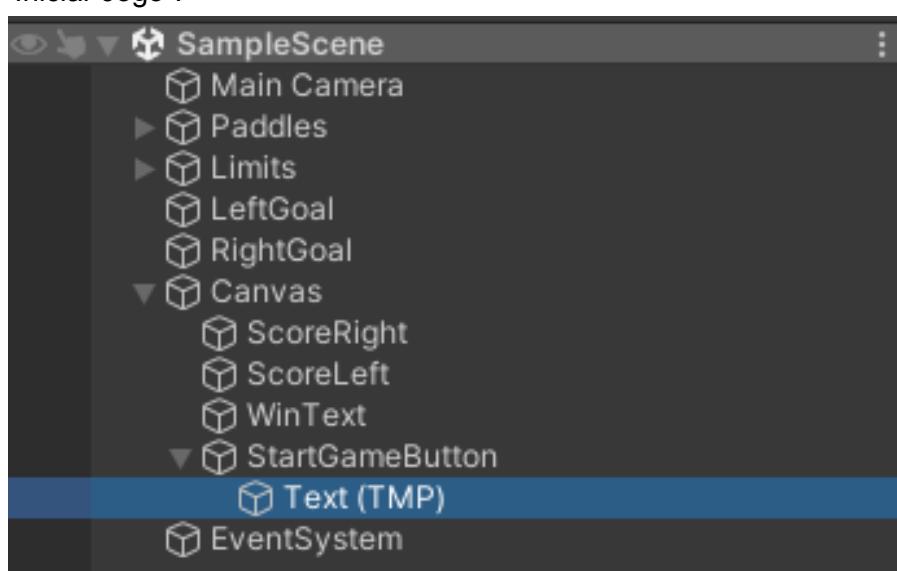


Figura 8.11 Botão na hierarquia.

A cena deve estar como mostra a figura 8.12.

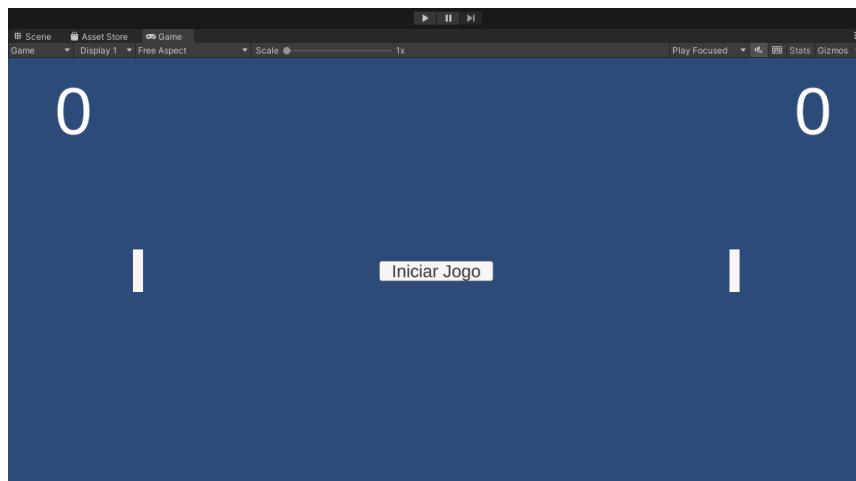


Figura 8.12 Botão na cena.

Ao testar o jogo nota-se que o botão não faz nada. Vamos mudar isso. Crie um Script chamado *GameManager*. Note que esse script possui um ícone diferente dos demais.

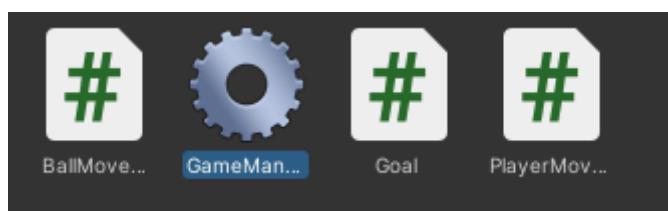


Figura 8.13 Script GameManager.

GameManager é um script muito comum em projetos de jogos na Unity, é o script responsável por monitorar o jogo. Crie um *GameObject* na hierarquia chamado *GameManager* e adicione o script *GameManager* à ele como indicado nas figuras 8.13 e 8.14.

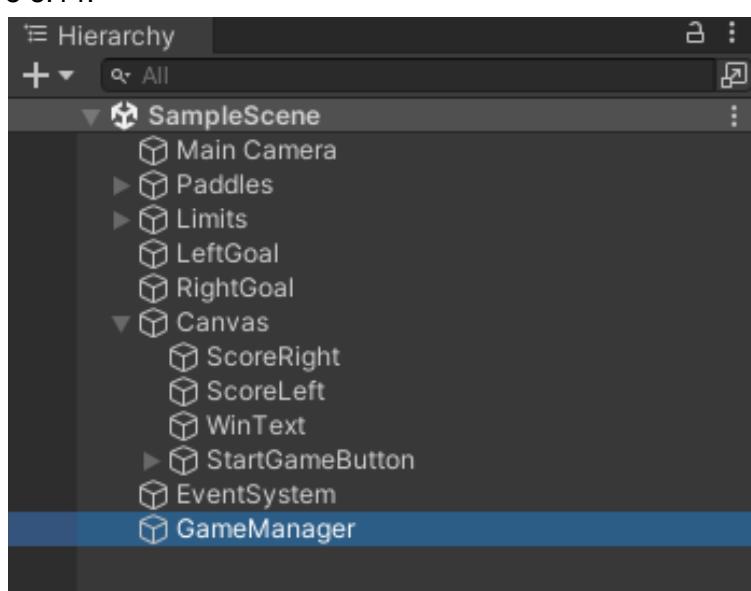


Figura 8.13 GameObject GameManager na hierarquia.

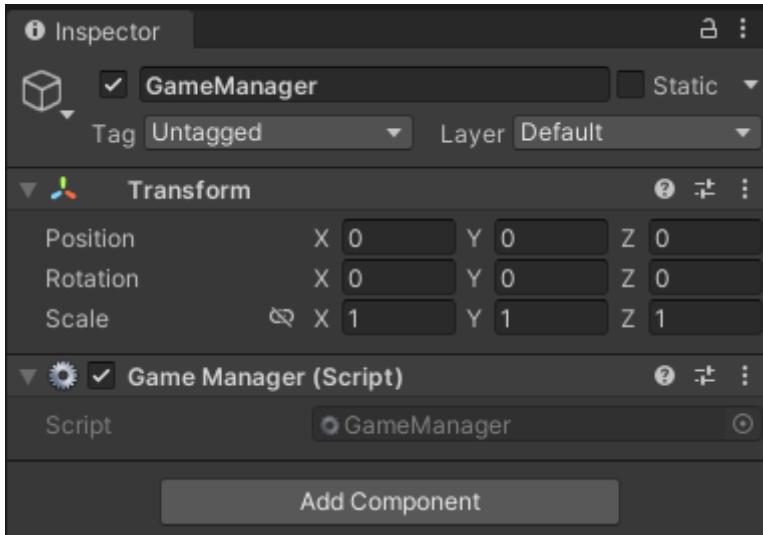


Figura 8.14 GameManager no inspector.

Abra o script *GameManager* para que possamos editá-lo. Apague os métodos *Start* e *Update* e cole o código a seguir:

```
public GameObject ball;

public GameObject leftGoal;

public GameObject rightGoal;

public void StartGame()
{
    Instantiate(ball, Vector3.zero, Quaternion.identity);
}
```

O método *StartGame* irá instanciar a Bola no centro da tela. Definimos o *GameObject* da bola com a variável *ball*. Note que o método *StartGame* é público, isso é necessário para que o botão de iniciar jogo possa utilizar esse método. Também criamos as variáveis *leftGoal* e *rightGoal*, elas serão úteis em instantes. Selecione o *GameManager* na hierarquia e preencha o campo *ball* com o prefab da bola e os campos *leftGoal* e *rightGoal* com os gameObjects *leftGoal* e *rightGoal* da hierarquia, como mostra a figura 8.15.

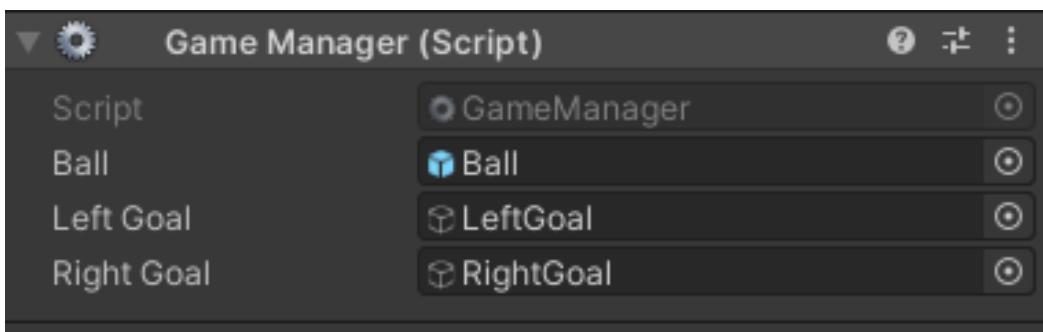


Figura 8.15 Campos de Game Manager.

Agora basta conectar o método *StartGame* ao evento de clique de *StartGameButton*. Selecione-o na hierarquia e no inspector desça a barra de rolagem até o final para ver o Componente *Button*, o mesmo da figura 8.16.

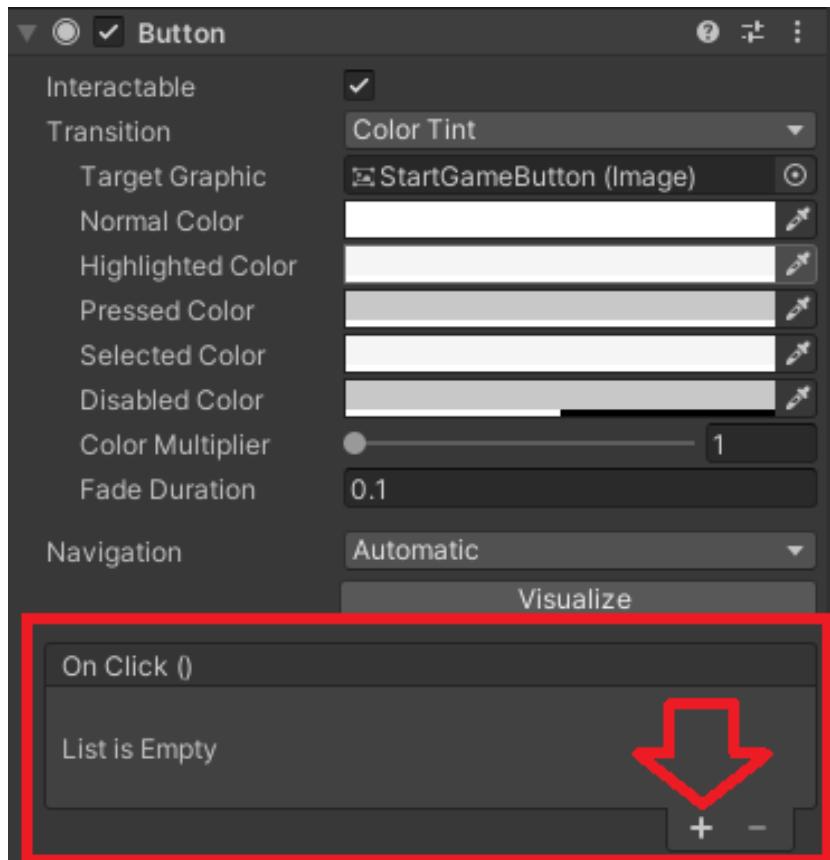


Figura 8.16 Componente Button.

O campo “On Click ()” contém a lista de métodos que ocorrem ao clicar no botão. No momento a lista está vazia. Para adicionar métodos primeiro é preciso clicar no botão indicado com a seta na figura 8.16.

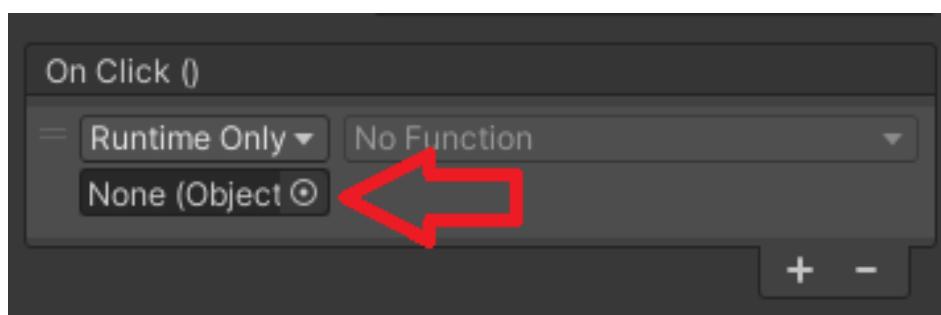


Figura 8.17 Botão de adicionar Objeto.

Clicando no botão indicado pela seta na figura 8.17 é possível selecionar qualquer *GameObject* da cena atual e acessar os métodos públicos de seus scripts e demais componentes. Selecione o *GameManager* como na figura 8.18. Note que para isso é necessário selecionar a aba *Scene*.

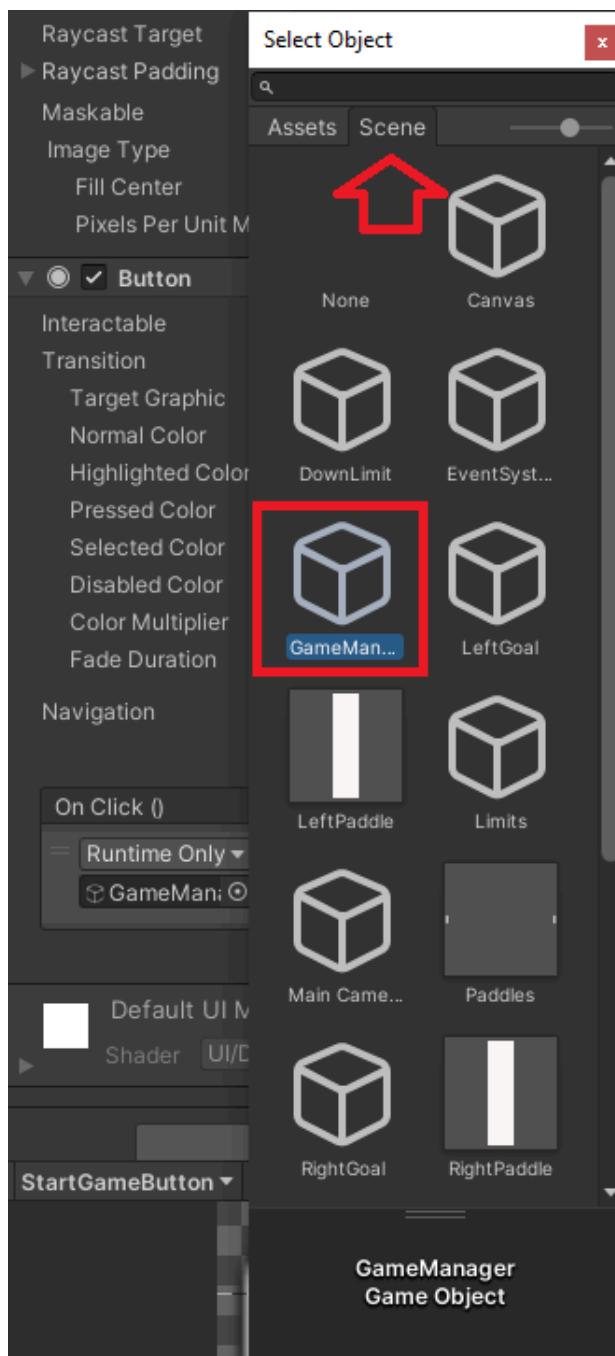


Figura 8.18 Adicionando GameManager.

O menu flutuante escrito *No Function* é onde podemos selecionar o método que queremos conectar ao evento de clique do botão. Clique no menu flutuante e selecione *GameManager > StartGame*.

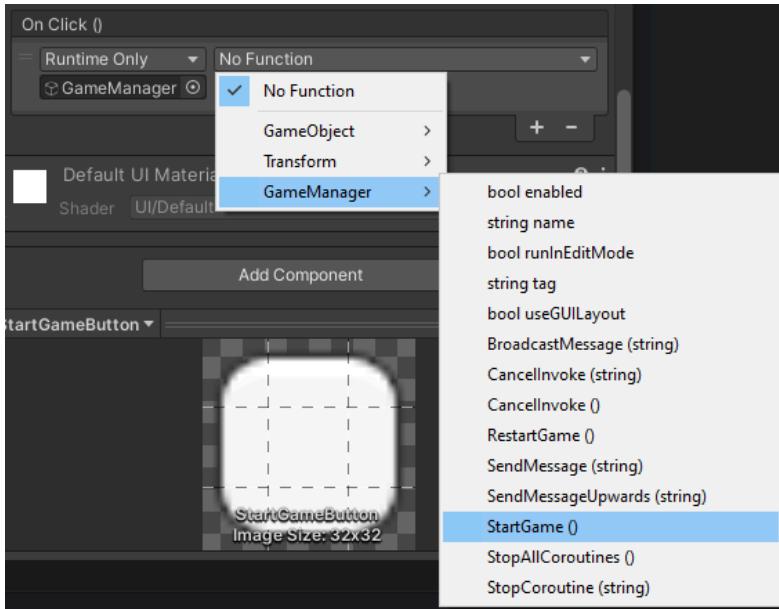


Figura 8.19 Selecionando StartGame.

Ao clicar no *play* e testar nota-se alguns problemas: ao clicar no botão para iniciar partida a bola surge, porém o botão continua na tela e se clicarmos múltiplas vezes no botão iniciar partida múltiplas bolas surgirão ao mesmo tempo. Para corrigir esse problema clique novamente no botão de adicionar métodos a lista, clique no botão de selecionar o *gameObject*, porém desta vez ao invé de selecionar o *GameManager* selecione o próprio *StartGameButton* e no menu flutuante onde está escrito *No Function* selecione *GameObject > SetActive*.

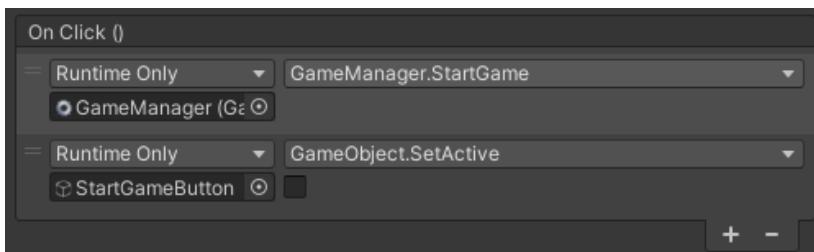


Figura 8.20 Adicionando evento de desativar Botão.

Atrelando o evento de clique de um botão ao método *SetActive* de um *GameObject* podemos ativar ou desativar esse *GameObject* da cena. Como a caixa de seleção da imagem acima está desmarcada, espera-se que ao clicar no botão o mesmo seja desativado da cena. Aperte o *play* e teste. Note que agora o botão desaparece da tela ao ser clicado e o jogo inicia normalmente.

No momento, para iniciar uma nova partida é necessário encerrar o teste e iniciar novamente, o que é análogo a abrir e fechar o jogo quando gerado o seu executável. Seria mais elegante ter um botão para reiniciar a partida após um dos jogadores vencer.

Na hierarquia, repita o processo de criar um botão e renomeie esse novo botão para *RestartButton*. Altere o texto de seu *TextMeshPro* para Reiniciar. Esse botão só deve

aparecer quando a partida terminar, portanto selecione *RestartButton* na hierarquia e no *inspector* desmarque a caixa de seleção ao lado do nome *RestartButton*, como mostra a figura 8.21.

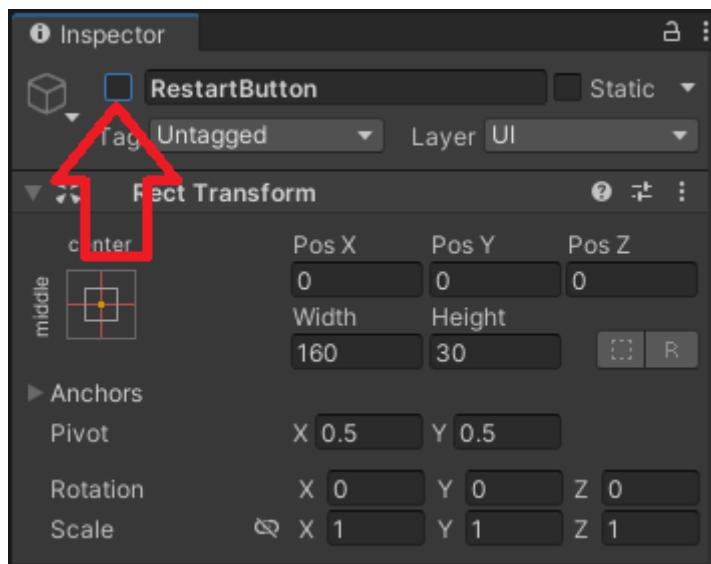


Figura 8.21 Desativando *RestartButton*.

Ao fazer isso, *RestartButton* não estará ativo na cena. Note que seu nome está em um tom mais fraco na hierarquia. Isso ocorre com todo *GameObject* que estiver inativo na cena.

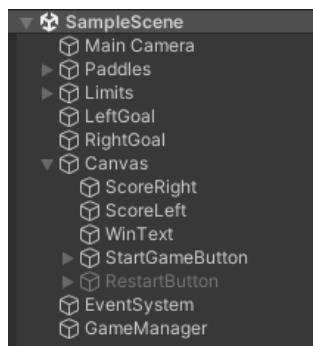


Figura 8.22 *GameObject* inativo.

Agora vamos fazer pequenas alterações no script *Goal* para que *RestartButton* apareça na tela junto a mensagem de vitória quando um dos jogadores vence a partida. Adicione uma variável pública do tipo *GameObject* chamada *restartButton* e altere o método *OnTriggerEnter2D* como mostra o script abaixo.

```
public GameObject restartButton;

void OnTriggerEnter2D(Collider2D other)
{
    Destroy(other.gameObject);
```

```

        score++;
        scoreText.text = score.ToString();

        if(score < 5)
        {
            Instantiate(ball, Vector3.zero, Quaternion.identity);
        }
        else
        {
            winText.text = "Vitória do lado " + playerPoint.ToString();
            restartButton.SetActive(true);
        }
    }
}

```

O método *SetActive* define se um *GameObject* estará ativo na cena, dependendo de seu argumento, true ou false. Utilizamos true pois queremos ativar *RestartButton* quando um dos jogadores vencer. No editor, atribua o *GameObject* *RestartButton* ao campo *Restart Button* de *LeftGoal* e *RightGoal*.

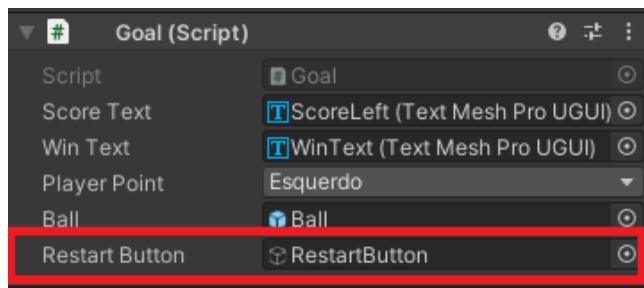


Figura 8.23 Campo RestartButton de Goal.

Aperte o *play* e teste. Note que ao término de uma partida surge o texto de vitória e o botão de reiniciar. O botão de reiniciar está na frente do texto de vitória. Corrija isso alterando o campo *Pos Y* do *Rect Transform* de *RestartButton* para -35.

O script *Goal* lida com a mensagem de vitória e as pontuações, criaremos um método que reinicie a pontuação e retire a mensagem de vitória. Adicione o seguinte método ao script *Goal*:

```

public void Restart()
{
    score = 0;
    scoreText.text = score.ToString();
    winText.text = "";
}

```

Ao clicar no botão de reiniciar o método acima deve ser chamado para *LeftGoal* e *RightGoal* e uma nova Bola deve ser instanciada. No script *GameManager*, adicione o seguinte método:

```

public void RestartGame()
{
}

```

```
    leftGoal.GetComponent<Goal>().Restart();  
    rightGoal.GetComponent<Goal>().Restart();  
  
    StartGame();  
}
```

Adicione o método *RestartGame* de *GameManager* ao evento de clique de *RestartButton* e o método *SetActive* de *RestartButton* como indica a imagem abaixo.

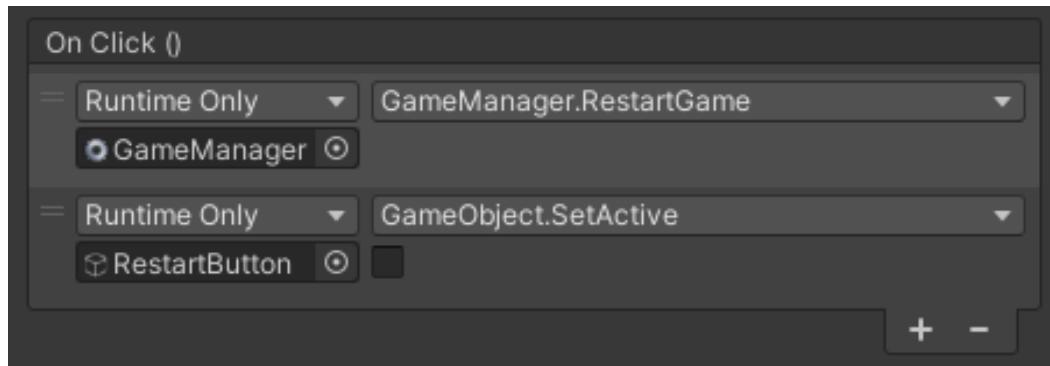


Figura 8.24 RestartButton.

Aperte o *play* e teste. Note que clicar no botão Reiniciar uma nova partida é iniciada.

10. Animações

Animação é o processo de dar vida e movimento a objetos ou personagens, criando a ilusão de que estão vivos, interagindo ou se transformando. Em jogos 2D isso é alcançado através da troca de imagens, alternando imagens para simular movimento. Na indústria essas imagens são comumente chamadas de *Sprites*.

9.1 Trabalhando com *Sprite Sheets*

Dentro da pasta sprites existe um arquivo de imagem chamado *ball_sprites*. Ele possui 6 *sprites* que representam a bola girando e 6 *sprites* que representam a bola estourando. Os *sprites* ao invés de estarem em arquivos separados estão em um arquivo só, chamado de *sprite sheet*. Para usar os *sprites* de forma separada precisaremos realizar uma operação chamada *slice*. Selecione o *sprite sheet* *ball_sprites* e observe o *inspector*.

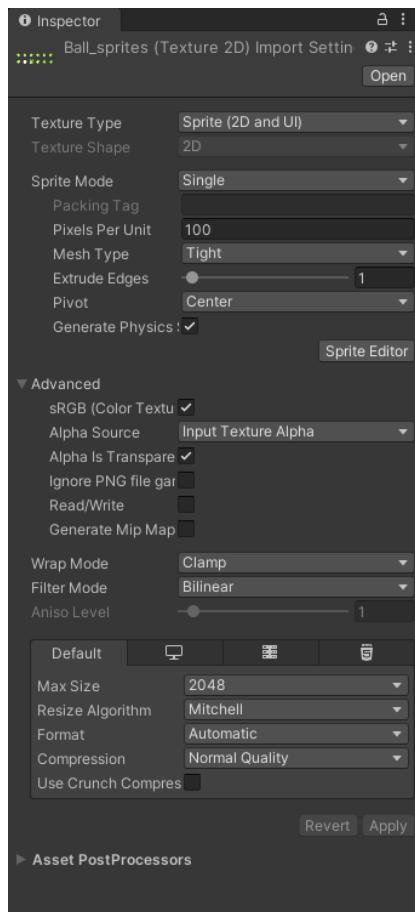


Figura 9.1 Sprite Sheet.

No momento o campo importante é *Sprite Mode*. Ao alterar seu valor de *Single* para *Multiple* e clicando em *Apply*, no canto inferior esquerdo, será possível realizar a operação *slice* no *sprite sheet* clicando em *Sprite Editor*. Ao clicar em *Sprite Editor*, a janela apresentada na figura 9.2 irá surgir.



Figura 9.2 Sprite Editor.

Ao clicar em *Slice* no canto superior direito, a mini janela apresentada na figura 9.3 irá surgir.

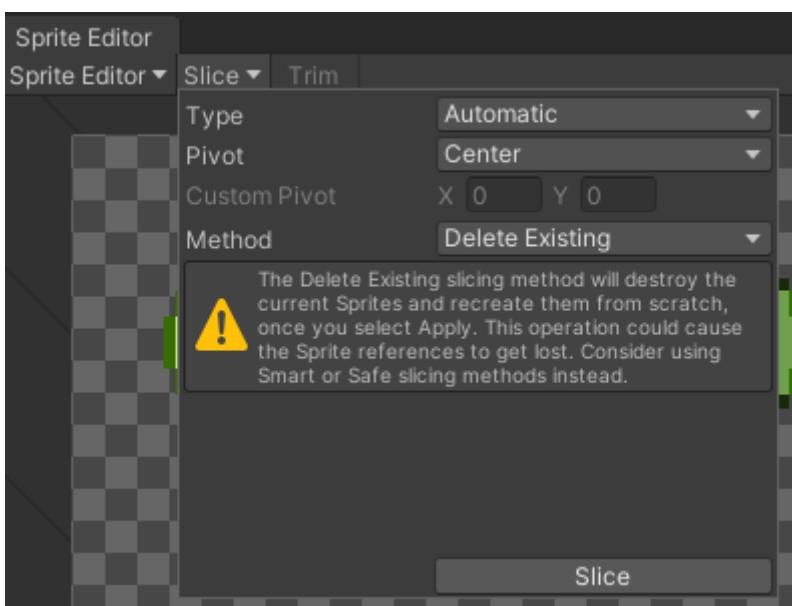


Figura 9.3 Janela de Slice.

O botão *Slice* realiza a operação de *Slice*, separando o *spritesheet* em *sprites* separados. Existem alguns métodos diferentes de se realizar o Slice, esse método é definido pelo campo *Type*. Por padrão, o *Type* selecionado é o *Automatic*. Este é o método mais simples e fácil, porém pouco recomendado por gerar sprites de tamanhos diferentes e poder confundir partículas na imagem com sprites separados. Se clicarmos em *Slice* com *Type Automatic* o *Slice* irá ocorrer como Mostrado na figura 9.4.



Figura 9.4 Slice Automático.

Note que os tamanhos estão diferentes e que duas partículas do quarto sprite da linha de baixo foram interpretadas como dois sprites separados. Este método é recomendado para Sprite Sheets desorganizadas. Selecionando o *Type Grid By Cell Size*, a janela de *Slice* se altera para a janela apresentada na figura 9.5.

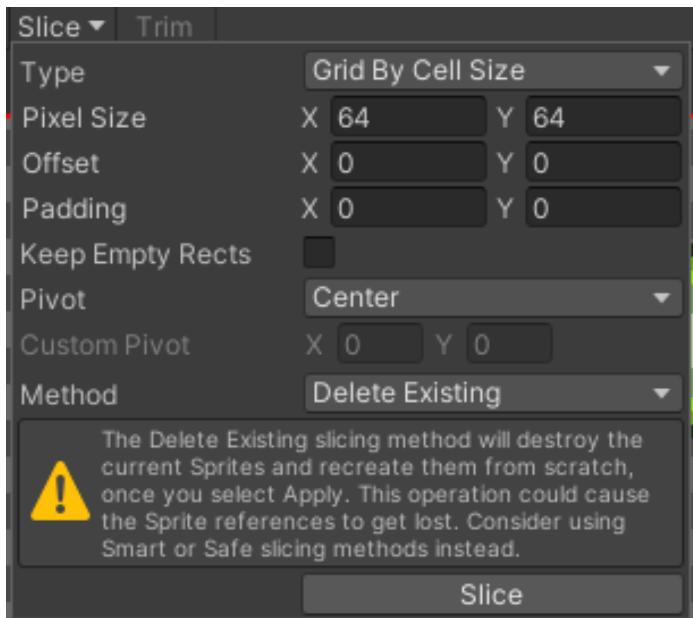


Figura 9.5 Janela de Slice By Cell Size.

Com esse método, os sprites possuem o mesmo tamanho em *pixels*. Os campos X e Y de *Pixel Size* definem respectivamente a largura e a altura dos sprites em *pixel*. Os sprites de *ball_sprites* foram feitos pensando em espaços de 32x32 *pixels*, portanto pode-se gerar sprites separados colocando o valor 32 para X e para Y. Ao fazer isso e clicar em *Slice*, os sprites são gerados como na figura 9.6.



Figura 9.6 Slice By Cell Size.

Este *Slice* percorre a *spritesheet* da direita para a esquerda e de cima para baixo cortando espaços com o tamanho especificado. Selezionando o *Type Grid By Cell Count*, a janela de *Slice* se altera para a janela apresentada na figura 9.7.

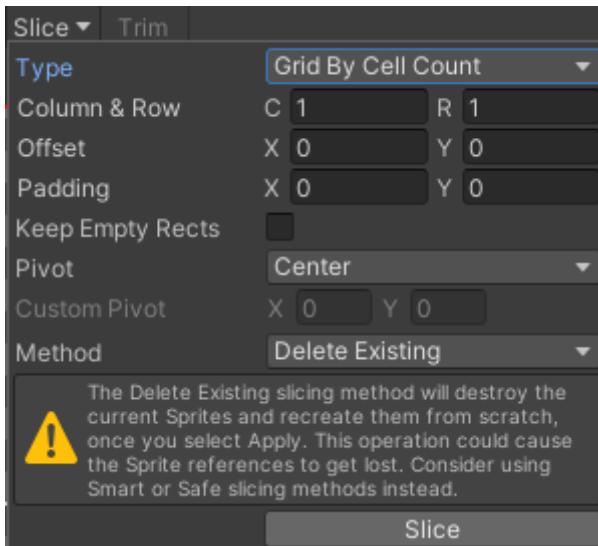


Figura 9.7 Janela de Slice By Cell Count.

Esse método corta a *sprite sheet* como uma matriz com número de colunas e linhas sendo definidas respectivamente pelos campos C e R de *Column & Row*. Os sprites de *ball_sprites* estão igualmente espaçados e estão organizados em 6 colunas e 2 linhas, portanto ao preencher os campos C e R com 6 e 2 respectivamente e clicando em *Slice*, os *sprites* são cortados corretamente. Após recortar os *sprites* é necessário clicar no botão *apply* no canto superior direito da janela *Sprite Editor*.

Após realizar o slice e salvar as alterações clique no ícone de seta em *ball_sprites* na *project view* para ver os sprites gerados.



Figura 9.8 Sprites gerados pelo Slice.

9.2 Criando uma animação

Os seis primeiros *sprites* pertencem à animação da bola girando. A forma mais simples de criar uma animação é selecionando os *sprites* pertencentes a essa animação e arrastá-los para a hierarquia. Com a tecla Control pressionada clique em cada um dos seis primeiros *sprites* de *ball_sprites*.

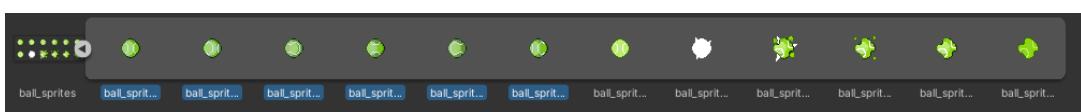


Figura 9.9 Selecionando Sprites da bola girando.

Com os *sprites* selecionados arraste-os com o mouse para a hierarquia. Ao fazer isso, a Unity abrirá uma janela para escolher onde salvar a nova animação. Crie uma pasta

chamada Animations e salve essa animação com o nome *ball_idle*, como indicado na imagem abaixo.

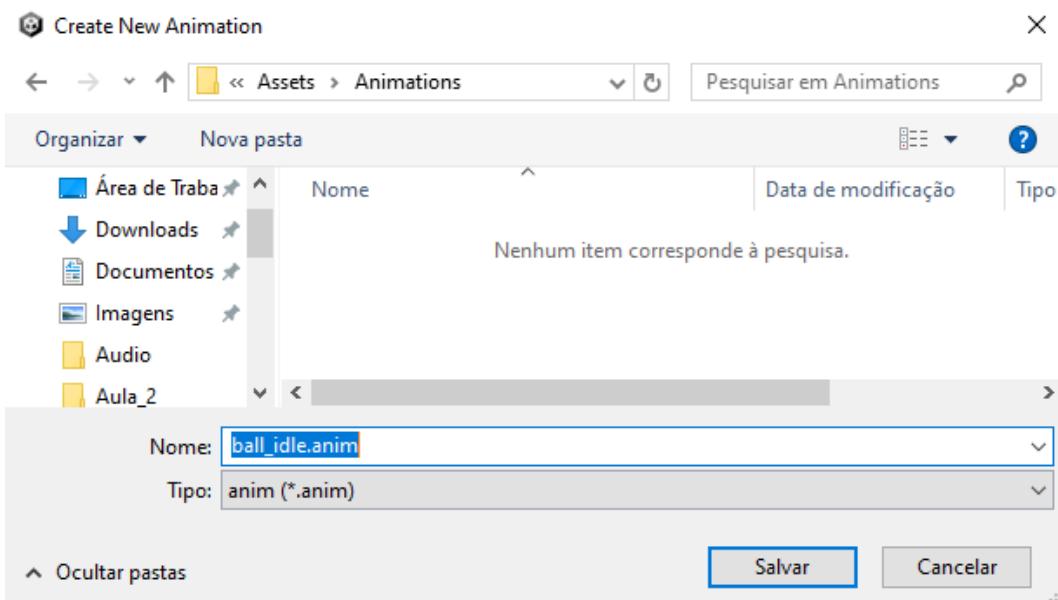


Figura 9.10 Salvando Animação.

Note que agora existe um *GameObject* chamado *ball_sprites_0* na cena. Ele possui o componente responsável por reproduzir a animação. Selecione *ball_sprite_0* e no *inspector* altere sua posição Y para 2, assim você terá uma melhor visualização. Aperte *play* e observe a animação da bola girando. Observe o componente *Animator* de *ball_sprites_0*.

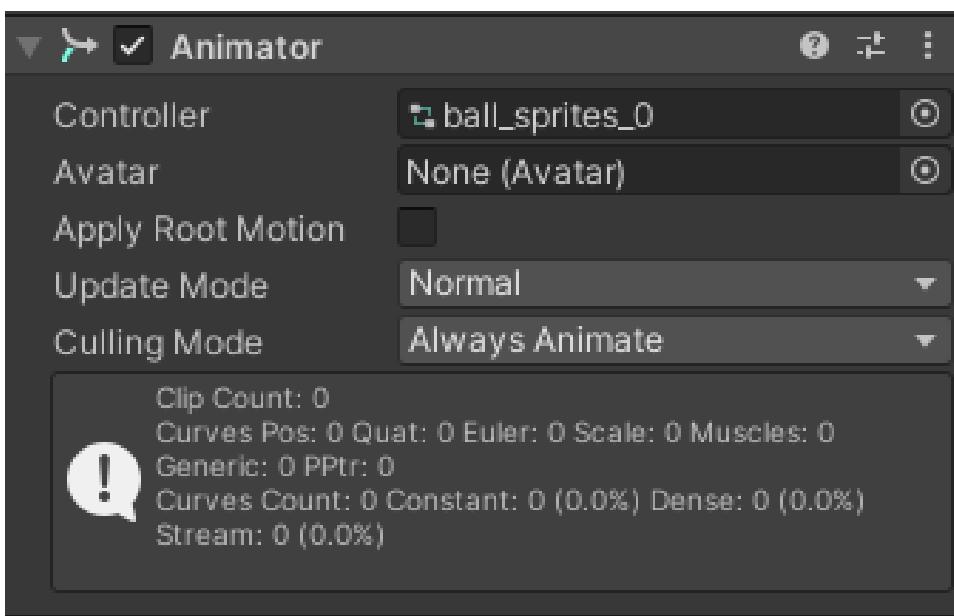


Figura 9.11 Componente animator.

Ele possui um asset do tipo *controller* no seu campo *Controller*. Esse asset foi criado junto à animação *ball_idle* na pasta *Animations*.

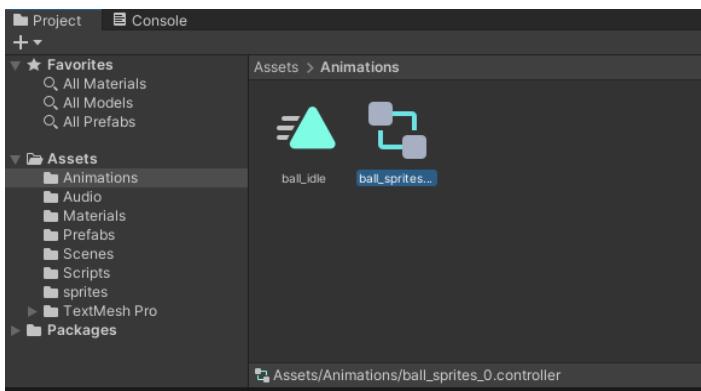


Figura 9.11 Controller criado.

Renomeie controller *ball_sprites_0* na *project view* para *ball_animator*. Precisamos aplicar essa animação ao prefab da bola. Selecione o prefab *Ball* e clique em *Add Component > Miscellaneous > Animator*.

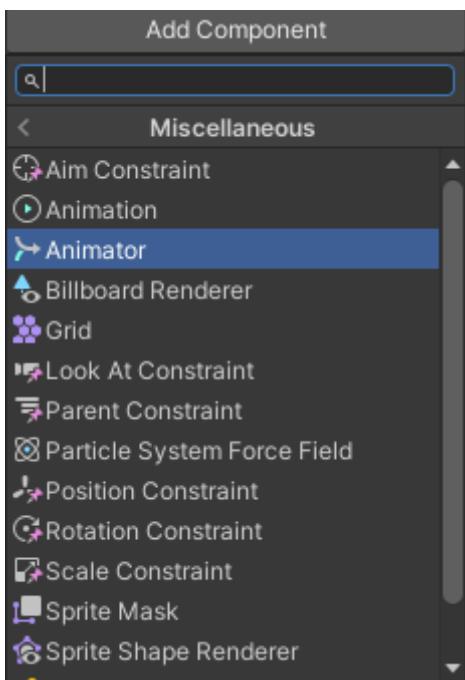


Figura 9.3 Adicionando animator.

No campo *controller* do *Animator* selecione *ball_animator*. Exclua o *GameObject* *ball_sprites_0* da hierarquia, aperte o play e teste. Agora as bolas possuem uma animação de bola girando.

11. Áudio

Sons são uma parte muito importante de um jogo. Geram imersão, representam um ambiente, proporcionam *feedback* para o jogador e podem provocar diversas emoções. A Unity possui um conjunto robusto de ferramentas para lidar com áudio.

10.1 Componentes do sistema de Áudio

Para que os sons funcionem em uma cena, você precisa de três elementos: o *audio listener*, o *audio source* e o *audio clip*. O *audio listener* é o componente mais básico de um sistema de áudio. Sua única responsabilidade é "ouvir" o que está acontecendo na cena. Uma maneira fácil de pensar nele é como se fosse o "ouvido" do seu mundo. Por padrão, toda cena começa com um *audio listener* anexado à Câmera Principal.



Figura 10.1 Audio Listener.

Não há propriedades disponíveis para o *audio listener*, e você não precisa fazer nada para que ele funcione. É uma prática comum colocar o *audio listener* em qualquer objeto do jogo que represente o jogador. Note que, se você colocar um *audio listener*

em outro objeto, será necessário removê-lo da Câmera Principal. Apenas um único *audio listener* é permitido por cena.

O *audio listener* escuta os sons, mas é o *audio source* que realmente emite o som. Esse componente pode ser colocado em qualquer objeto da cena (até mesmo no objeto que contém o *audio listener*). O *audio source* possui muitas propriedades e configurações, que serão abordadas em sua própria seção mais adiante.

O último item necessário para o áudio funcionar é o *audio clip*. Como você pode imaginar, o *audio clip* é o arquivo de som que é reproduzido pelo *audio source*. Cada *clip* possui algumas propriedades que podem ser configuradas para alterar a forma como o Unity os reproduz. O Unity suporta os seguintes formatos de áudio: .aif, .aiff, .wav, .mp3, .ogg, .mod, .it, .s3m e .xm.

Juntos, esses três elementos proporcionam uma experiência sonora à sua cena.

10.2 Áudio 2D e 3D

Um conceito importante sobre áudio é a ideia de som 2D e 3D. Audio clips 2D são os tipos mais básicos de áudio. Eles tocam no mesmo volume, independentemente da proximidade do áudio listener em relação ao *audio source* na cena. Sons 2D são mais indicados para menus, alertas, trilhas sonoras ou qualquer áudio que precise ser ouvido sempre da mesma forma. No entanto, a maior vantagem dos sons 2D também é sua maior limitação. Imagine se todos os sons do seu jogo tocassem no mesmo volume, independentemente de onde você estivesse. Isso rapidamente se tornaria caótico.

O áudio 3D resolve os problemas do áudio 2D. Esses *audio clips* possuem algo chamado *roll-off*, que determina como os sons ficam mais baixos ou mais altos dependendo da proximidade do *audio listener* ao *audio source*. Em sistemas de áudio avançados, como o Unity, sons 3D podem até simular o efeito Doppler. Se você busca um áudio realista em uma cena cheia de fontes sonoras diferentes, o áudio 3D é a melhor escolha.

A dimensionalidade dos *audio clips* é selecionada nas configurações individuais de cada arquivo de som. Para este projeto focaremos em áudio 2D.

10.3 Audio Source

Como mencionado anteriormente, os *audio sources* são os componentes que realmente reproduzem audio clips em uma cena. Para adicionar um *audio source* clique com o botão esquerdo do mouse na hierarquia e selecione *Audio > Audio Source* como indicado na figura 10.2.

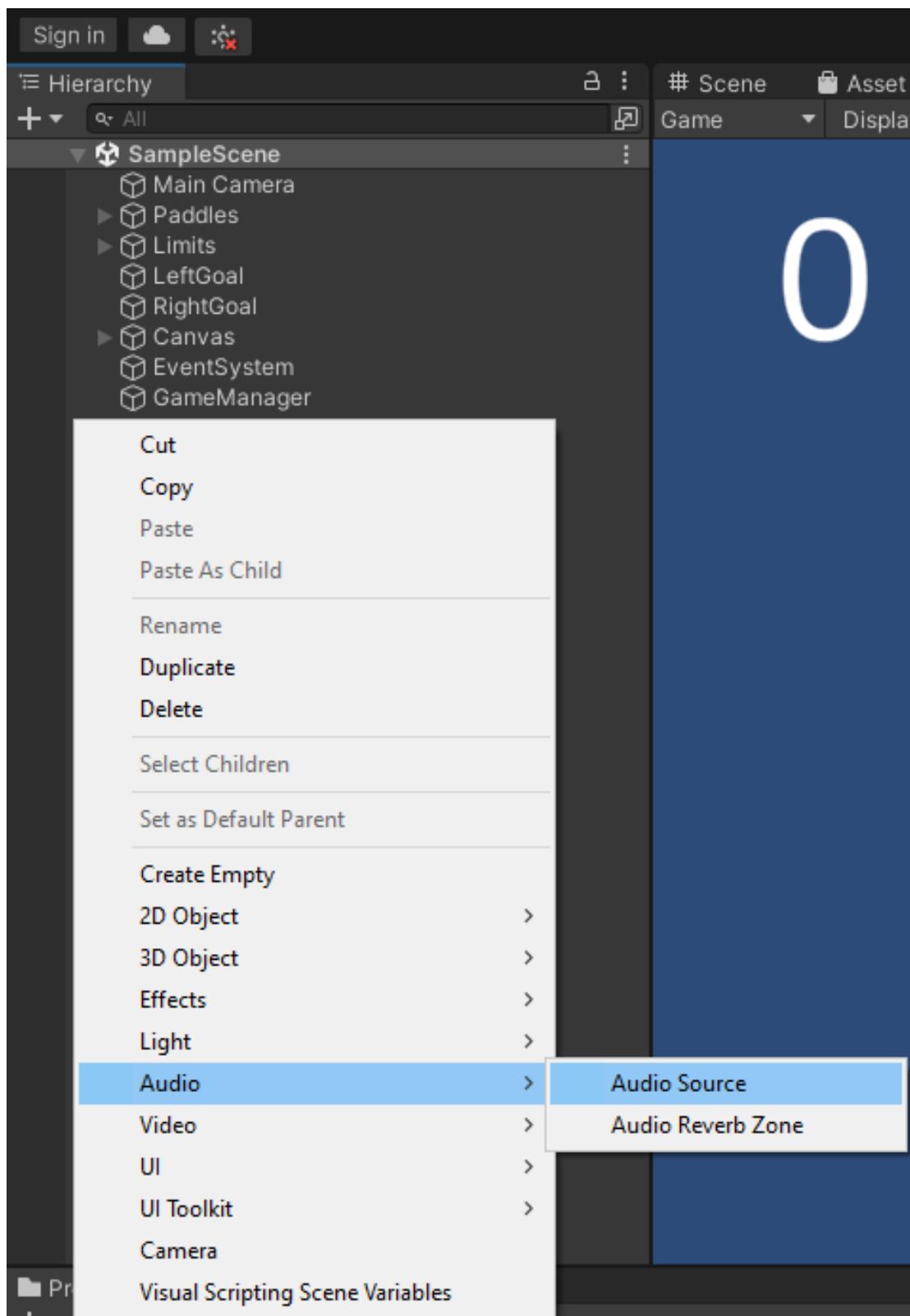


Figura 10.2 Adicionando um Audio Source.

Adicione um *audio source* na cena e nomeie seu *GameObject BackgroundMusic*. Esse *audio source* será responsável por adicionar uma música ao jogo. Observe o componente *Audio Source* no *inspector*.

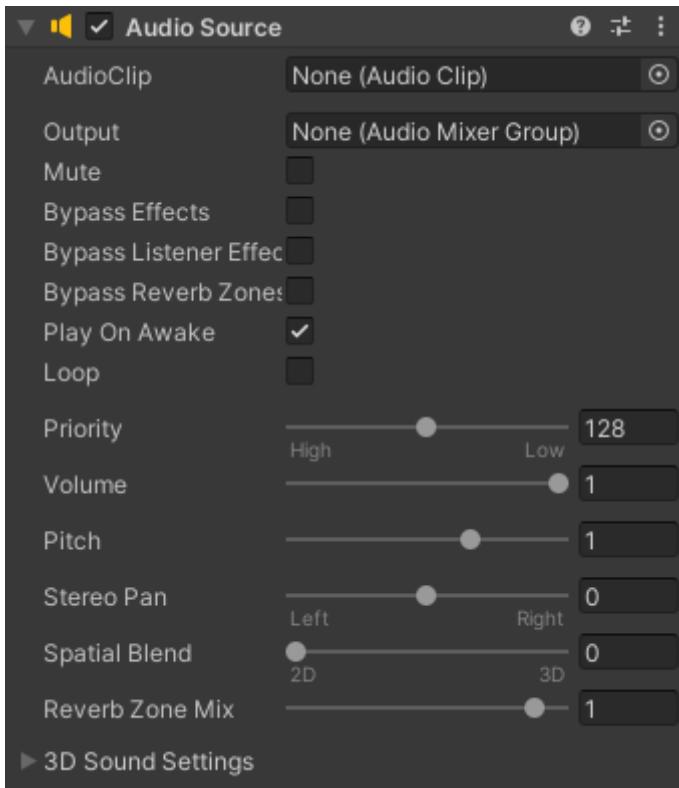


Figura 10.3 Componente Audio Source.

Principais campos:

Audio Clip: O arquivo de som real a ser reproduzido.

Output: Opcionalmente, direciona o clipe de som para um *Audio Mixer*.

Mute: Determina se o som está silenciado.

Bypass Effects: Determina se os efeitos de áudio são aplicados a esta fonte. Selecionar essa propriedade desativa os efeitos.

Bypass Listener Effects: Determina se os efeitos do *audio listener* são aplicados a esta fonte. Selecionar essa propriedade desativa os efeitos.

Bypass Reverb Zones: Determina se os efeitos das zonas de reverberação (*reverb zones*) são aplicados a esta fonte. Selecionar essa propriedade desativa os efeitos.

Play On Awake: Determina se o áudio começará a ser reproduzido automaticamente assim que a cena for carregada.

Loop: Determina se a fonte de áudio reiniciará o clipe de áudio quando ele terminar de tocar.

Priority: A importância da fonte de áudio. O valor 0 é o mais importante, e 255 é o menos importante. Use 0 para músicas, para garantir que elas sempre sejam reproduzidas.

Volume: O volume da fonte de áudio, onde 1 equivale a 100% do volume.

Pitch: O tom da fonte de áudio.

Stereo Pan: Define a posição no campo estéreo do componente 2D do som.

Spatial Blend: Define o quanto o mecanismo 3D afeta a fonte de áudio.

Reverb Zone Mix: Define a quantidade do sinal de saída que é direcionada para as zonas de reverberação (reverb zones).

3D Sound Settings: Configurações aplicadas a clipes de áudio 3D.

Na pasta *Audio* tem um *Audio Clip* chamado *Background*, que contém a música que deverá tocar durante o jogo. Para adicionar *Background* ao *Audio Source* de *BackgroundMusic* existem dois métodos. Com o *GameObject BackgroundMusic* selecionado na hierarquia basta arrastar o *Audio Clip Background* de *Project* para o campo *AudioClip* do componente *Audio Source* no *Inspector*. O outro método é clicar no botão em forma de bola ao lado do campo *AudioClip* e selecionar *Background* na janela que abre em seguida, como indica a figura 10.4.

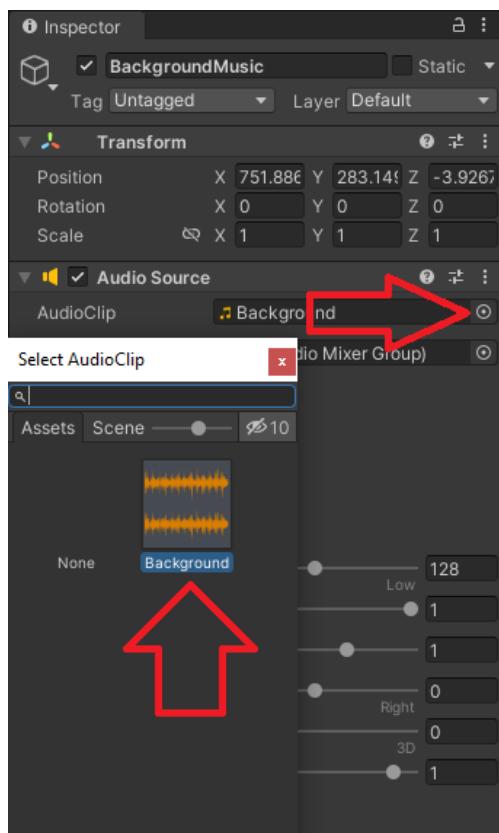


Figura 10.4 Adicionando AudioClip ao Audio Source.

Em jogos digitais as músicas costumam tocar repetidamente sem parar, por isso o campo *Loop* deve ser marcado. O *Audio Source* de *BackgroundMusic* deve estar como a figura 10.5.

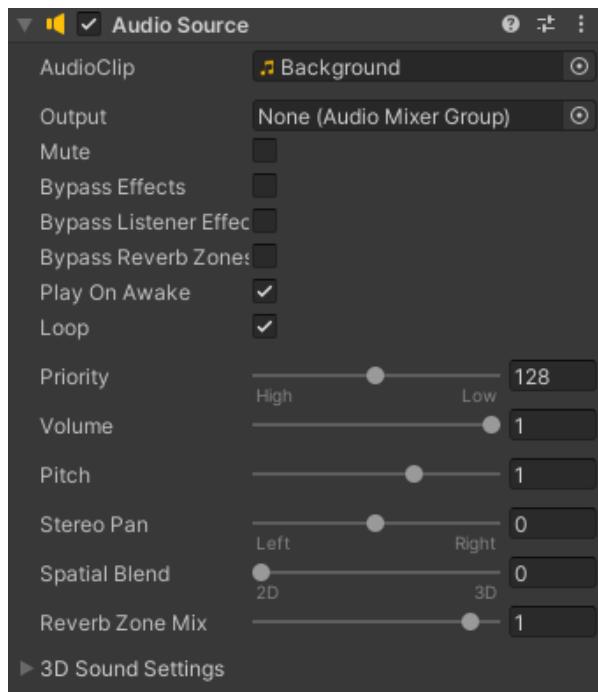


Figura 10.5 BackgroundMusic.

Aperte o *play* e teste. A música deve estar tocando sem parar.

10.4 Audio Scripting

Reproduzir áudio de um *Audio Source* assim que ele é criado é útil, supondo que essa seja a funcionalidade desejada. No entanto, se for necessário esperar para reproduzir um som em um momento específico ou reproduzir sons diferentes a partir do mesmo *Audio Source*, será necessário utilizar scripts. Felizmente a manipulação dos *Audio Sources* é uma tarefa bastante simples utilizando a *Scripting API* da Unity.

A classe *AudioSource* permite a manipulação dos *Audio Sources* . Veja abaixo como criar uma variável *AudioSource* .

```
private AudioSource audioSource;

void Start()
{
    audioSource = GetComponent<
```

Para tocar um som basta utilizar o método *Play*, para interromper esse som utiliza-se o método *Stop*.

```
audioSource.Play();  
audioSource.Stop();
```

O atributo *isPlaying* informa se o som está tocando no momento.

```
if(audioSource.isPlaying)  
{  
    //realiza uma ação caso o som esteja tocando  
}
```

Seria interessante a bola reproduzir um som de impacto toda vez que ela bate em nas barras ou nos limites da tela. Adicione um *Audio Source* ao prefab da bola. Para isso selecione o prefab da bola, clique em *Add Component > Audio > Audio Source* e desmarque a opção *Play On Awake*. No Campo *AudioClip* selecione o Audio Clip *impact_000*.

No script *BallMovement* acrescente a variável *audioSource* e faça as alterações descritas abaixo no método *Start*.

```
private AudioSource audioSource;  
  
void Start()  
{  
    rb = GetComponent<Rigidbody2D>();  
    Invoke("LaunchBall", 3f);  
    audioSource = GetComponent<}
```

Existe um método *callback* da Unity que verifica quando o *colisor 2D* de um *GameObject* toca em outro *colisor 2D*, é o método *OnCollisionEnter2D*. Vamos utilizar esse método para tocar o som de impacto. Adicione o método abaixo ao script *BallMovement*.

```
void OnCollisionEnter2D(Collision2D col)  
{  
    audioSource.Play();  
}
```

Aperte o *play* e teste. Note que agora um som de impacto toca sempre que a bola encosta nos limites da tela ou em uma das barras.

Alterar o *Audio Clip* de *Audio Source* utilizando Scripts é uma tarefa bastante simples. Basta utilizar uma variável do tipo *AudioClip* e alterar o campo *clip* da variável do tipo *AudioSource*, como no exemplo abaixo.

```
public AudioClip newClip;

void Method()
{
    audioSource.clip = newClip;
    audioSource.Play();
}
```

Seria interessante se o som produzido pela bola ao tocar em algo fosse aleatório. Primeiramente vamos utilizar uma estrutura conhecida como *array*. Um *array* é uma coleção de valores de um mesmo tipo e é criado adicionando um abrir e fechar de colchetes após o nome de um tipo de variável. Neste caso queremos ter uma coleção de *AudioClips*, portanto criaremos uma *Array* de *AudioClips*. Acrescente a seguinte linha após a variável *audioSource*.

```
public AudioClip[] impactSounds;
```

Os *colchetes* após *AudioClip* indicam que a variável *impactSounds* não é apenas um *AudioClip*, mas um *array* de *AudioClips*, ou seja, uma coleção com um ou mais *AudioClips*. No editor da Unity selecione o prefab da bola no projeto e observe o Script *Ball Movement* no Inspector. Note o campo *Impact Sounds*.

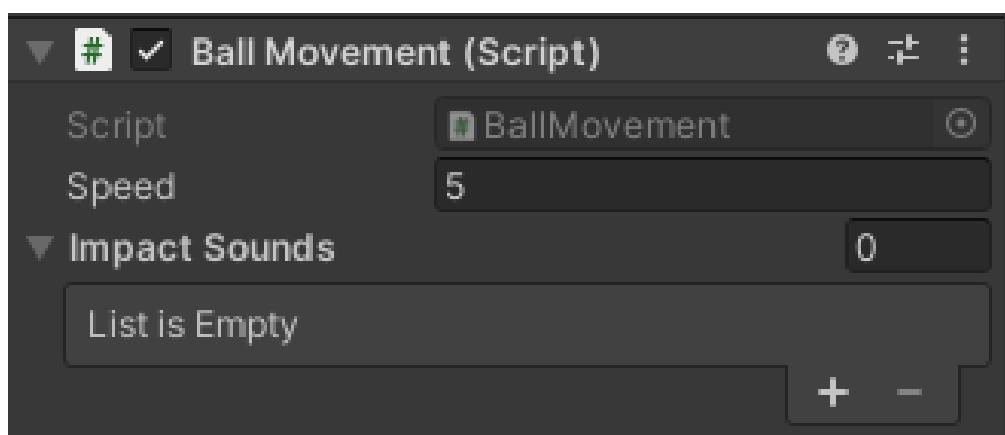


Figura 10.6 Campo Impact Sounds.

O valor 0 indica quantos campos o array possui 0 campos no momento. Podemos adicionar campos utilizando o botão com sinal de cruz ou alterando o valor numérico que no momento é 0. Altere o valor numérico para 5 e aperte enter. Note que 5 campos do tipo *AudioClip* surgem.

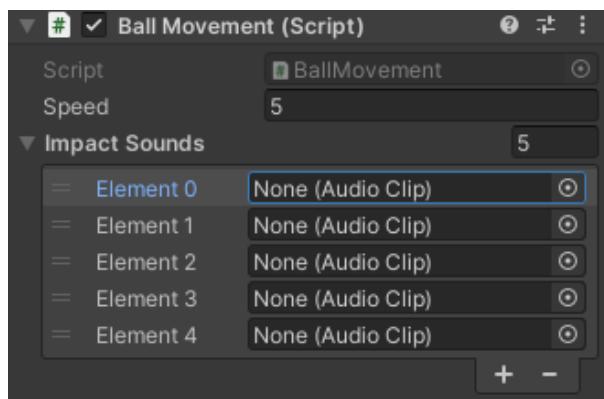


Figura 10.7 Campos de Impact Sound criados.

Selecione para os campos os *Audio Clips* *imapct_000* à *impact_004*, como indica a figura 10.8.

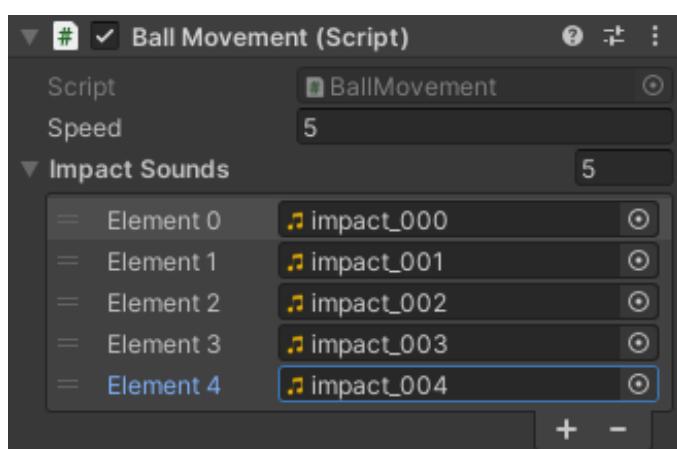


Figura 10.8 Campos de Impact Sound preenchidos.

Quando se quer acessar um valor específico de um *array* utilizasse o nome desse *array* e a posição da variável entre colchetes. Por exemplo, para alterar o *audio clip* da variável *audioSource* para o *audioClip* da posição 2 de *impactSounds* utiliza-se o script abaixo.

```
audioSource.clip = impactSounds[2];
```

Um ponto importante sobre as posições dos valores de um *Array*, a primeira posição sempre é 0, ou seja, um *array* com 5 valores possui valores nas posições 0, 1, 2, 3 e 4. Para saber quantos valores um *array* possui basta utilizar o atributo *Length*. Por exemplo, se *impactSounds* possuir 5 *AudioClips*, o script abaixo atribui o valor 5 à variável *number*.

```
int number = impactSounds.Length;
```

Altere o método *OnCollisionEnter2D* como mostra o script abaixo.

```
void OnCollisionEnter2D(Collision2D col)
{
```

```
        AudioSource.clip      =      impactSounds [Random.Range (0,
impactSounds.Length) ];
    AudioSource.Play ();
}
```

O script acima altera o *Audio Clip* de *audioSource* para um *audio clip* presente em *impactSounds* em uma posição aleatória entre 0 e o tamanho do array *impactSounds* subtraído por 1, e em seguida toca esse som. Aperte o *play* e teste. Note que agora um som aleatório toca sempre que a bola toca nos limites da tela ou em uma das barras.

11. Geração de Executável

Até o momento o jogo produzido através deste tutorial só existe dentro do editor da unity, porém espera-se que o jogo esteja disponível de forma externa à unity, como um executável para windows por exemplo. Neste capítulo será gerado uma versão executável para windows chamada de build.

No menu superior clique em *File > Build Settings* como indica a figura 11.1.

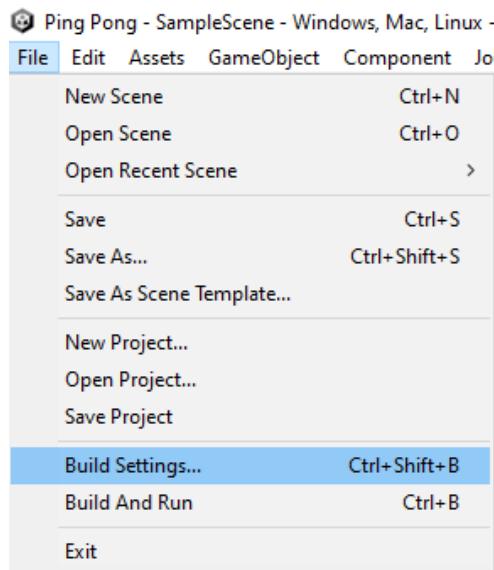


Figura 11.1 Abrindo Build Settings.

Ao fazer isso, a janela apresentada na figura 11.2 irá aparecer.

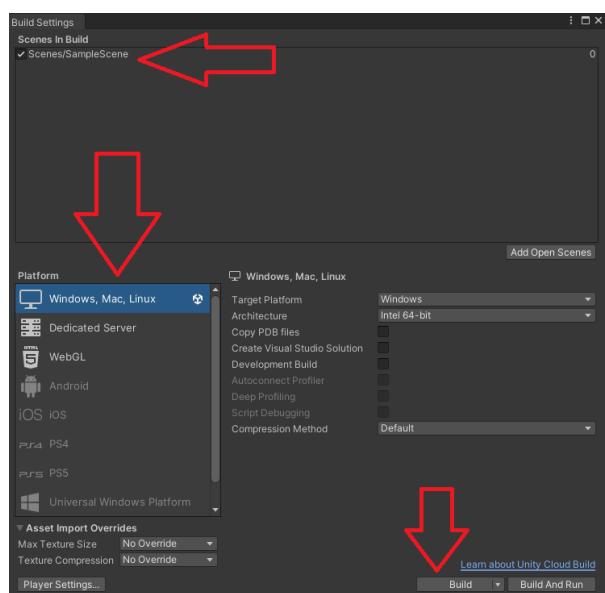


Figura 11.2 Build Settings.

O campo superior indica as cenas que estarão presentes na *build*. Abaixo deste campo à esquerda temos o local onde indicamos para qual plataforma a build será gerada. Mantenha a plataforma *Windows*, *Mac*, *Linux*. Na parte inferior temos o botão *Build*, clique nele para poder gerar a *build*. Será necessário escolher em qual pasta estará sua *build*. Crie uma pasta chamada *build* e selecione-a.

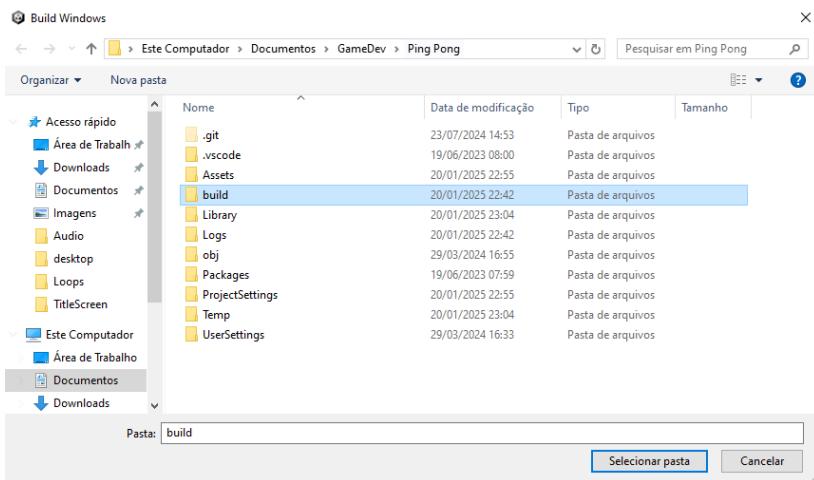


Figura 11.3 Pasta build.

Aguarde o processo de geração da *build* que será indicado por uma barra. Após o processo de criação da *build* a pasta selecionada deve abrir com algumas pastas e dois executáveis, o *UnityCrashHandler* e o jogo. Abra o jogo e teste-o.

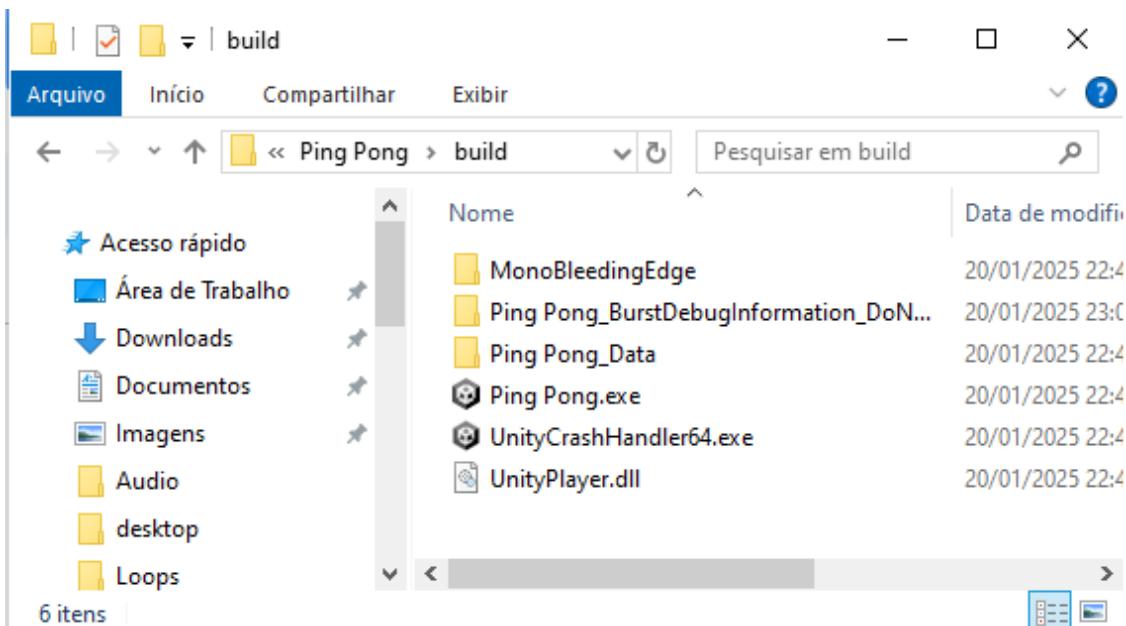


Figura 11.4 Build.

12. Encerramento

Este é o fim deste tutorial. Você criou seu primeiro jogo na unity, aprendeu sobre conceitos básicos da Unity como suas *views*, *GameObjects*, Prefabs, Scripts, o sistema de física 2D, interfaces, animações e sons. O próximo passo é desenvolver novos jogos para continuar seu processo de aprendizado.

Você pode acessar uma versão para windows do executável do resultado esperado através deste link: <https://romuboy.itch.io/ping-pong>

Você pode acessar o projeto final esperado através do seguinte repositório: <https://github.com/ROMUBOY/Ping-Pong-TCC/tree/main>

Referências

- gamesindustry.biz. (2024). ***GamesIndustry.biz presents... The Year In Numbers 2024.*** Disponível em: <https://www.gamesindustry.biz/gamesindustrybiz-presents-the-year-in-numbers-2024>. Acesso em: 06 de fevereiro de 2025.
- Abragames. (2024). ***2023 Indústria Brasileira de Games.*** Disponível em: https://www.abragames.org/uploads/5/6/8/0/56805537/2023_relat%C3%B3rio_final_v4_3.2_-_ptbr.pdf. Acesso em: 30 de setembro de 2024.
- DevMedia. (2023). ***Desenvolvendo Jogos em Unity 3D: Tutorial Completo.*** Disponível em: <https://www.devmedia.com.br/desenvolva-jogos-com-a-unity-3d/29125>. Acesso em: 30 de setembro de 2024.
- Unity Technologies. (2023). ***Game Development Resources, Case Studies & Articles.*** Disponível em: <https://unity.com/resources>; Acesso em 30 de setembro de 2024.
- Unity Technologies. (2023). ***Unity - Scripting API.*** Disponível em: <https://docs.unity3d.com/ScriptReference/>. Acesso em: 30 de setembro de 2024.
- Kenney. ***Thousands of completely free game assets for you to use.*** Disponível em: <https://kenney.nl>. Acesso em: 30 de setembro de 2024.
- Freesound. ***Find any sound you like.*** Disponível em: <https://freesound.org>. Acesso em: 30 de setembro de 2024.