Desenvolvimento com Motores de Jogos I

Aula 05 - Os Elementos Gráficos e a Câmera

Apresentação

Olá, galera! Chegamos à Aula 05 da nossa disciplina de Desenvolvimento com Motores de Jogos I! Já? Pois é! O tempo voa quando nos divertimos! E, nesta aula, nos divertiremos ainda mais: o assunto dessa vez é a câmera!

Como vimos em outras aulas, tudo o que fazemos dentro do mundo de um jogo é simulado para obtermos, através de alguma lógica, o resultado buscado. Isso funciona da mesma maneira com a câmera! É por meio dela que somos capazes de exibir aos jogadores o mundo do nosso jogo, além de fazê-los poder interagir com esse mundo e observar as suas mudanças.

Veremos, na aula de hoje, a importância da câmera para os nossos jogos e como podemos utilizá-la e configurá-la no Unity! Obviamente, já lidamos com uma câmera desde a nossa primeira aula, pois ela é necessária para exibir de tudo no jogo. Agora, conheceremos melhor os detalhes desse elemento e como ele é capaz de, sozinho, modificar o que o nosso jogo passa aos usuários. Prontos para mais essa jornada? Então preparem as baterias, escolham suas melhores lentes, peguem suas câmeras e vamos brincar de Pokémon Snap!

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Compreender a importância de uma câmera nos jogos digitais;
- Entender as mudanças que podem ser feitas nas câmeras para atender às especificidades de cada jogo;
- Conhecer as propriedades de câmera do Unity.

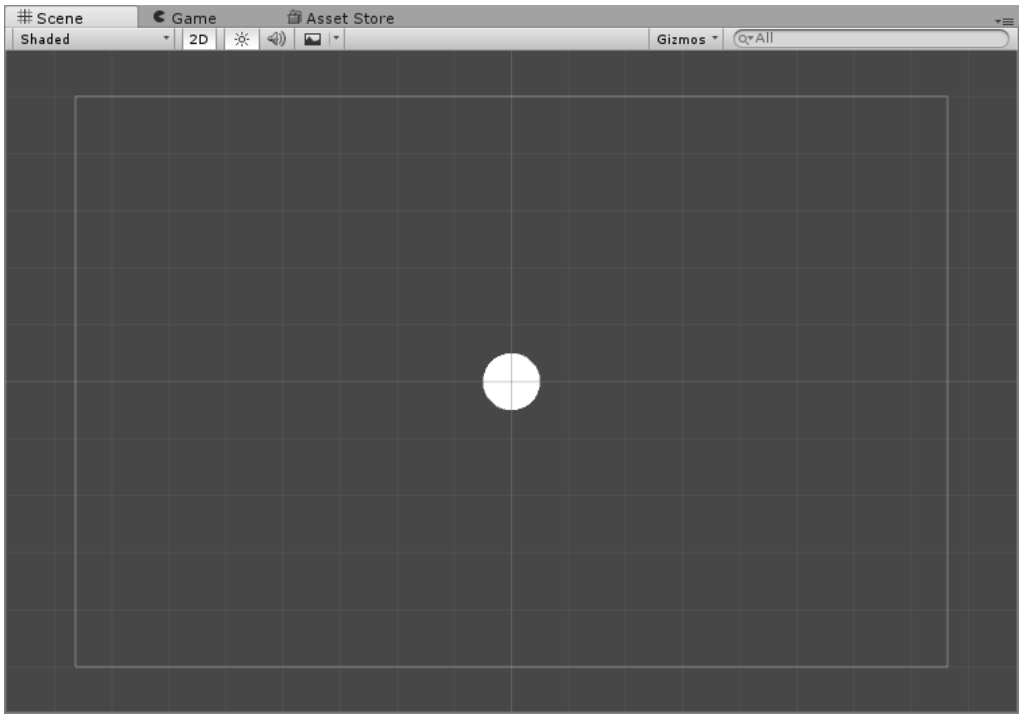
Elementos Gráficos e sua Relação com a Câmera

Quando falamos em jogos digitais, pensamos sempre em elementos gráficos sendo desenhados em uma tela. Isso é realmente o que acontece na maioria dos casos e é exatamente sobre eles que discutiremos ao longo desta aula. No entanto, a fim de que sejam exibidos, além da definição adequada de suas propriedades gráficas, precisamos de uma câmera para mostrar aos nossos usuários esses elementos gráficos, o que está ao seu redor e a cena em sua totalidade. Mais que isso, a utilização da câmera pode ser importante, também, para gerar efeitos interessantes ao nosso jogo, como cortes de câmera, movimentação do personagem, entre outros. Então, para podermos entender melhor tudo isso, começaremos a nossa conversa de hoje falando a respeito de um dos principais aspectos nos quais a câmera, os elementos gráficos e o mundo de objetos estão envolvidos: os espaços de coordenadas.

Sistemas de Coordenadas na Computação Gráfica

Quando falamos acerca de um objeto, como já vimos em aulas anteriores, tratamos primeiramente sobre a sua posição no espaço. Dizemos que um objeto tem uma posição X, uma Y e uma Z, e definimos um valor para cada uma delas. Vimos em outras aulas que, mesmo em 2D, a Z possui a sua importância, pois pode determinar o objeto que está à frente e qual deles pode ou não ser desenhado pela câmera. Mas perceba que esse espaço é absoluto e faz referência a onde o objeto está localizado no mundo. E na câmera? Onde está o objeto? Qual o espaço de coordenadas que funciona para ele? Os múltiplos espaços de coordenadas são utilizados exatamente objetivando facilitar esses entendimentos. Vejamos, na **Figura 1**, um círculo branco, localizado na posição (0, 0, 0) do **sistema de coordenadas do mundo**.

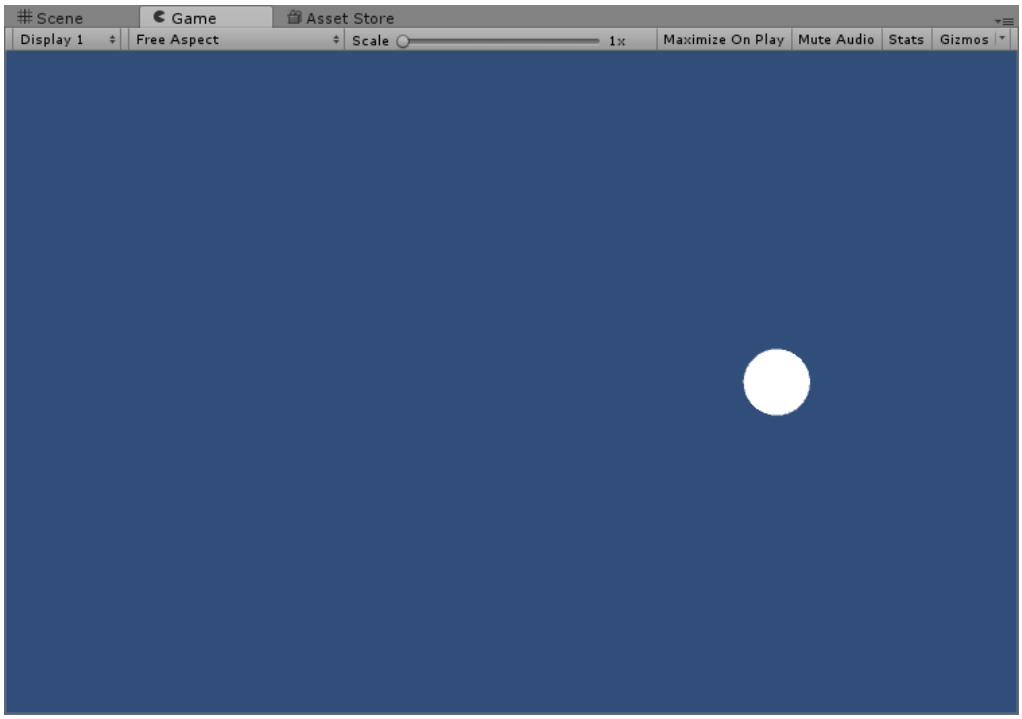
Figura 01 - Círculo localizado na origem do sistema de coordenadas do mundo.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 05 fev. 2017.

Como esperado, um círculo posicionado na origem do sistema de coordenadas do mundo faz ele ficar centralizado na tela, a qual também está centralizada na origem. Mas o que acontece quando esse círculo vai para a nossa câmera? Conversamos em aulas anteriores que, ao apertar Play (|>), vemos na aba Game do Unity o jogo sendo executado da maneira como o jogador vê. E isso acontece porque passamos a ver o jogo através da câmera! Mas e se ela não estiver centrada na origem? Posicionemos a câmera, por exemplo, no ponto (-4, 0, -10). Vamos ver o efeito que isso tem em nosso objeto? Observe a **Figura 2**.

Figura 02 - Círculo localizado na origem de coordenadas do sistema do mundo, exibido por uma câmera descentrada



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 05 fev. 2017.

Wow! O círculo, que não foi movido em momento algum, não está mais no centro do sistema... Como pode? Pois é! Esse é exatamente o efeito causado pelos diferentes sistemas de coordenadas nos objetos que vemos em nosso jogo! O círculo, antes, parecia completamente centrado e, agora, está bem deslocado para direita. Isso acontece devido a ele não estar centrado no **sistema de coordenadas da câmera**! Esse sistema, uma vez que movemos a câmera para a posição (-4, 0, -10), está centrado nessa posição do **sistema de coordenadas do mundo** e não no (0, 0, 0), como normalmente acontece.

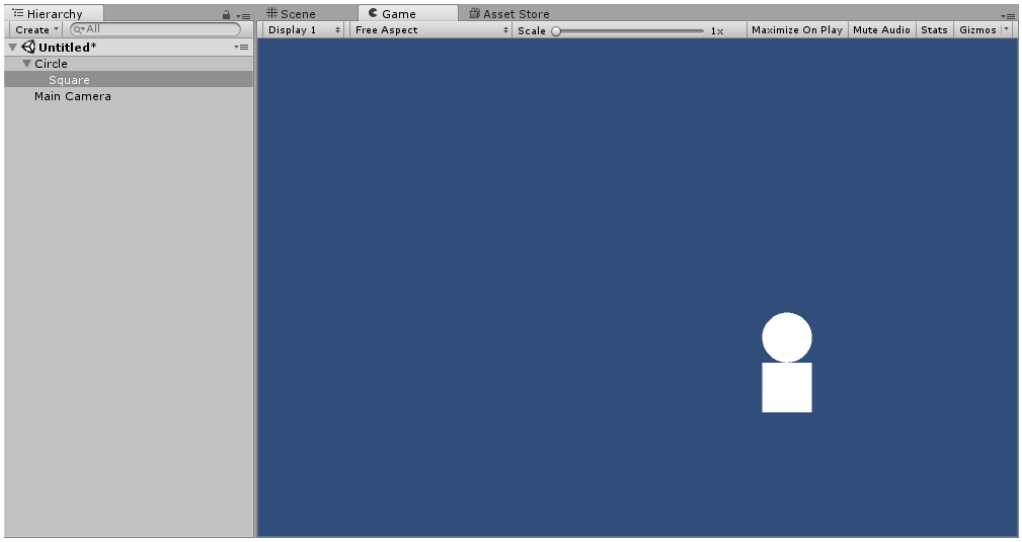
Consegue perceber a diferença? Podemos brincar com esses diferentes sistemas de coordenadas para conseguirmos efeitos visuais interessantes em nosso jogo, como discutiremos em seções mais à frente. Mas não é só isso! Ainda há mais sistemas de coordenadas que são utilizados para definir outros aspectos importantes ao tratarmos de elementos gráficos. Cada objeto, por exemplo, também possui o seu próprio sistema de coordenadas!

- Ah, mas para que eu quero utilizar o sistema de coordenadas de um objeto individual? Isso não deve servir para nada!

- Errado! Isso serve muito! Quer um exemplo bem claro? Podemos utilizar o sistema de coordenadas de um objeto para definir a posição de outro objeto! E como fazemos isso? Utilizando o sistema de hierarquia do Unity, já visto em outras aulas! Criaremos agora, como *filho* desse círculo mostrado nas figuras anteriores, um quadrado, o qual posicionaremos na posição (0, -1, 0), em relação ao **sistema de coordenadas do objeto**. Para ficar mais claro, moveremos também o círculo para (-1, -1, 0), retirando-o da origem do **sistema de coordenadas do mundo**. Vejamos o resultado na **Figura 3.**

Lembrando que, para colocarmos um objeto como *filho* de outro, basta clicar e arrastar esse objeto para cima do outro na hierarquia, como discutimos em aulas passadas.

Figura 03 - Objeto quadrado colocado como filho do círculo na hierarquia e seu posicionamento na visualização final da câmera.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 05 de fev de 2017.

Vejam, na aba Hierarchy, que o objeto Square está colocado como filho do Circle. Após movermos o círculo para (-1, -1, 0), esse objeto ficou em uma posição diferente em relação à **Figura 2**. Ao posicionarmos o quadrado em (0, -1, 0), em relação ao **sistema de coordenadas do objeto**, o objeto ficou, na verdade, na posição (-1, -2, 0), aparecendo na tela abaixo do círculo. Percebem a diferença?

Além disso tudo, o nosso jogo precisa ainda ser desenhado na tela, concordam? Afinal, é nela que ele será exibido para o nosso jogador. A tela pode, então, ser 1024x768, 1920x1080, 800x600 e por aí vai, ou seja, precisamos exibir, em uma faixa

de pixel variável, aquilo que está sendo montado pela câmera! Precisamos, também, desenhar o resultado no **sistema de coordenadas da tela.** Mas essa parte nós não nos preocuparemos muito e nem entraremos em detalhes agora. O Unity cuidará disso para nós e simplesmente aceitaremos aquilo que ele fizer! Veremos mais detalhes sobre o sistema de coordenadas da tela quando formos entender melhor a parte de capturar posicionamento de cliques, pois estes são gerados nesse sistema.

No fim das contas, todas essas trocas de espaço, todos esses sistemas de coordenadas e tudo visto até agora envolvem uma matemática bem interessante, a qual, infelizmente (ou não), não teremos tempo para ver nesta aula. O importante mesmo para entendermos aqui são os três sistemas de coordenadas utilizados pelo Unity para desenhar o nosso jogo e a exposição do sistema de coordenadas da tela, com o qual nos encontraremos novamente em uma oportunidade futura!

Então, concluindo essa primeira parte, pausaremos um pouco e assimilaremos bem os quatro sistemas de coordenadas usualmente utilizados em aplicações gráficas.

Sistema de Coordenadas do Mundo - Expressa as coordenadas absolutas do objeto em relação ao sistema de posicionamento de toda a cena. É centrado em (0, 0, 0) e possui o seu Y positivo para cima e o seu X positivo para a direita.

Sistema de Coordenadas da Câmera - Expressa as coordenadas do objeto em relação à câmera. É utilizado principalmente para mostrar o objeto na tela de jogo e não é manualmente configurado no Unity, sendo calculado automaticamente a partir da posição da câmera.

Sistema de Coordenadas do Objeto - Expressa as coordenadas a partir do centro do objeto em questão. É utilizado principalmente para definir outros objetos e ações em relação ao objeto principal. Segue a mesma lógica das coordenadas do mundo, porém, recentradas no objeto.

Sistema de Coordenadas da Tela - Indica a posição de um objeto na tela em que o jogo está sendo exibido. Esses valores são sensíveis a mudanças na resolução da tela e calculados automaticamente pelo Unity. Não serão importantes no momento.

Visto tudo isso, podemos entender melhor como a computação gráfica, de modo geral, trabalha com diversos sistemas de coordenadas. Há ainda outros autores que diferenciam alguns sistemas de coordenadas a mais, como o **upright**, o qual indica o posicionamento rotacional dos objetos, entre outros sistemas. Discutiremos, no entanto, apenas os citados acima, que serão o bastante para você entender bem os espaços do Unity e a importância deles. Falando nisso, vamos para o próxima etapa, começar a falar dos espaços do Unity em si? Gogogo!

Sistemas de Coordenadas do Unity

O Unity, como esperado, simplifica muito o sistema de coordenadas para não termos trabalho em fazer várias conversões ou mesmo nos enganarmos ao posicionar objetos em nosso jogo. Essa simplificação é feita em quatro principais espaços: **Screen Coordinates, GUI Coordinates, Viewport Coordinates** e **World Coordinates**. Esses quatro sistemas de coordenadas possuem peculiaridades e, no caso do World Coordinates, engloba mais de um sistema de coordenadas dos que vimos anteriormente. Não há nenhuma definição oficial do Unity quanto a esses espaços, ou como eles são implementados. Os pontos apresentados aqui são baseados em suas funcionalidades e no que a comunidade aceita como verdadeiro. Veremos alguns detalhes importantes de cada um deles.

World Coordinates

O **World Coordinates**, no Unity, representa, como alguns de vocês já podem ter observados anteriormente, três espaços diferentes: o do mundo, o do objeto e o da câmera. O que realmente acontece é todos os valores utilizados nos objetos posicionados na cena serem expressos em função de coordenadas do mundo. Apesar disso, os objetos também possuem um sistema de coordenadas local, chamado **Local Coordinates**. Esse sistema indica as coordenadas em função da própria rotação do objeto. Apesar disso ser um subsistema do **World Coordinates**, é também bem importante saber que ele existe.

A principal utilidade do sistema de coordenadas local é realizar movimentação em relação aos eixos do objeto. Imagine que temos um foguete sendo lançado em 45°. Para que ele se mova na direção do lançamento, precisaríamos alterar dois eixos em coordenadas globais. Já em coordenadas locais, basta aumentar o seu Y!

Isso, na verdade, vai alterar dois eixos em coordenadas globais da mesma forma, já que as coordenadas locais são apenas uma representação e não um sistema de coordenadas, mas fica bem mais fácil de pensar, não é?

GUI Coordinates

O **GUI Coordinates** representa o sistema de coordenadas da Graphical User Interface (GUI) do seu jogo. Esse sistema de coordenadas funciona de maneira similar ao **sistema de coordenadas de tela** que vimos anteriormente, mas é invertido em relação aos seus eixos. A origem do sistema se dá no topo esquerdo da tela, e o seu fim no canto inferior direito, atingindo neste canto os valores Screen.width e Screen.height.

Esse sistema de coordenadas é utilizado, como o próprio nome sugere, para a adição de elementos de interface ao seu jogo. Podemos posicionar barras de vida, textos, botões, entre outras coisas, utilizando os pontos desse sistema de coordenadas, sem interagir diretamente com outros sistemas. Às vezes é necessário, no entanto, que haja o desenho de uma parte da interface em um local específico, como a vida de uma unidade acima de sua cabeça, por exemplo. Nesses casos, será preciso utilizar, mais uma vez, uma das funções de conversão, como citamos anteriormente.

Mas, espere! Assim como um *boss* às vezes aparece no meio da fase só para deixar a sua mensagem e de repente some, nós não entraremos em detalhes de como isso funciona agora! Deixaremos o suspense para a aula de interface gráfica que teremos mais adiante. ;)

Viewport Coordinates

A classe **Viewport Coordinates** representa o sistema de coordenadas da tela em si, independentemente da resolução, tamanho, etc. É um sistema de coordenadas 2D que inicia em 0,0 no canto inferior esquerdo e vai até o ponto 1,1 no canto superior direito. Essas coordenadas são especialmente úteis para quando precisamos posicionar mais de uma câmera em nossa cena, por exemplo. Entraremos em mais detalhes sobre a utilização desse sistema em breve, ao falarmos da câmera do Unity propriamente dita.

Screen Coordinates

Por fim, as **Screen Coordinates** representam o sistema de coordenadas da tela em relação à resolução. Esse sistema desenhará, de acordo com o que for definido pelo **Viewport**, as informações na tela em si, baseado na resolução e no formato desta. O seu posicionamento varia de 0,0 no canto inferior esquerdo até Screen.width, Screen.height no canto superior direito. Ele mapeia exatamente o que vimos anteriormente como **sistema de coordenadas da tela**.

Com isso, finalizamos os tópicos referentes aos sistemas de coordenadas em nossa aula. Já é possível identificar que há sistemas diversos e que, em alguns casos, há a necessidade de interação entre eles. Esta deve ser feita por métodos como o ScreenToWorldPoint(), o qual veremos em aulas futuras, quando utilizarmos, de fato, esses métodos.

Agora que entendemos melhor esses espaços, tentaremos compreender um pouco mais sobre a nossa câmera no Unity, a qual é um elemento específico que possui diversas propriedades a serem utilizadas e alteradas.

Atividade 01

---

1. Para melhor fixar cada sistema de coordenadas, descreva, com suas palavras, os quatro sistemas de coordenadas utilizados pelo Unity.

O Elemento Camera

No desenvolvimento de jogos há um elemento tão importante que o próprio Unity já o adiciona previamente em qualquer nova cena criada em seus projetos: o elemento Camera, ou, em português, Câmera (sério!). Esse elemento é o responsável por capturar o seu mundo com todos os elementos deste e o exibir ao seu jogador, por meio do monitor. Para facilitar o entendimento do elemento Camera, pense que ele tem a mesma funcionalidade do objeto real utilizado em nosso dia a dia, ou seja, em uma cena de jogo a câmera tem o mesmo objetivo de uma câmera real, seja no

cinema, TV ou mesmo no quintal da sua casa. Você a utiliza para gravar, de determinado ângulo, em uma posição escolhida, com uma determinada quantidade de iluminação e de zoom, uma cena que está acontecendo à sua frente.

Ao pensar na câmera de um jogo fazendo alusão a uma câmera real, muitas ideias e associações são facilitadas, baseadas no que já conhecemos do mundo real. Podemos planejar a criação de efeitos utilizando o zoom, movimentação de câmera, giro de câmera, etc. As opções são diversas!

Em jogos, no entanto, às vezes também temos outros objetivos a serem alcançados com a utilização da câmera. Pense, por exemplo, no jogo do nosso amigo robô. O que acontece quando você aperta o D? O robô se move para a direita, correto?

E se move...

E se move...

E sai da tela! Adeus! :'(

Então, uma das tarefas dada por nós à câmera, por exemplo, pode ser simplesmente a de seguir o nosso personagem! Já que em alguns casos (e o nosso jogo se enquadra bem nestes) o objetivo é simplesmente controlar um personagem e fazê-lo se locomover ao longo de um cenário, podemos colocar a nossa câmera centrada nesse personagem e fazê-la ir para onde ele for, nos dando sempre uma boa noção de como podemos agir e de onde ele está. No final da aula, quando voltarmos a mexer em nosso joguinho, construiremos uma câmera desse tipo para melhorar a jogabilidade dele!

Já em outros casos, a câmera pode ser utilizada para matar! :O

Quem já jogou em algum jogo uma fase conhecida como auto-scroller? Super Mario World, por exemplo. Nesse tipo de nível, a câmera se move em uma velocidade constante e o personagem deve, de acordo com esse movimento, escapar dos objetos para "se manter na tela". Caso consiga, o personagem avança com a tela. Caso não, ele perderá a fase assim que sair da câmera. Esse é apenas um exemplo de utilização de câmera, mostrando que esta pode interferir diretamente no gameplay do seu jogo.

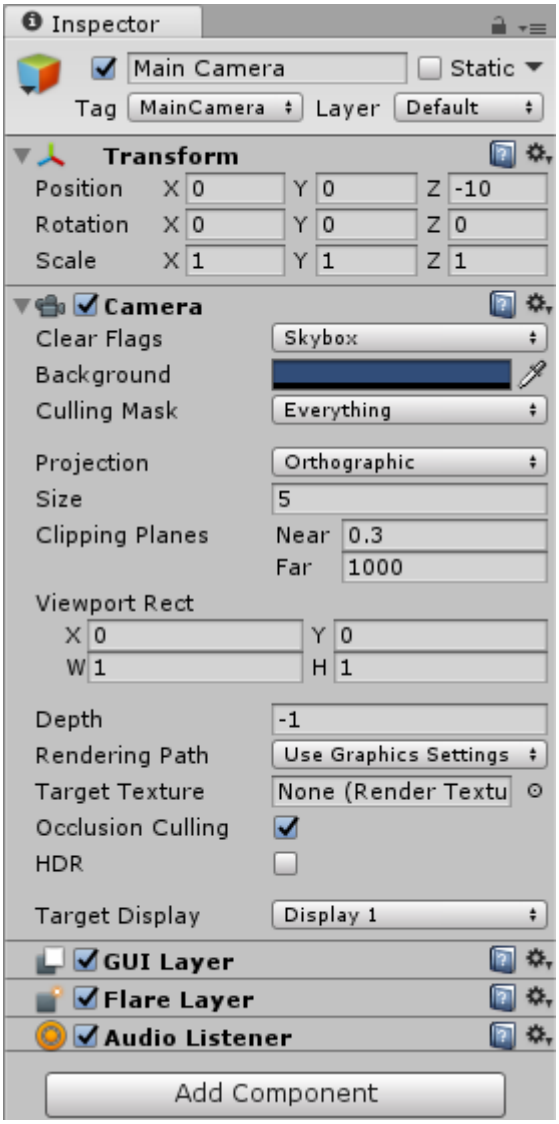
A câmera é tratada como um objeto normal no Unity. Na verdade, assim como qualquer outro objeto, o objeto Camera herda de GameObject e possui as mesmas funcionalidades que diversos outros objetos possuem. Por isso, o primeiro componente visto na câmera é o Transform, o qual contém posição, rotação e escala, como já vimos em outros objetos.

Outro detalhe interessante da câmera é ela vir, por padrão, com a tag MainCamera selecionada para ela, facilitando, assim, sua identificação em outros scripts e sua associação em geral. Perceba também que o nome **Main Camera** indica algo bem interessante: é possível ter mais de uma câmera por cena! O nome “Main”, ou “principal”, em português, demonstra que essa câmera é a responsável pela maior parte do desenho, mas ela pode não ser a única existente em uma cena.

Podemos utilizar mais de uma câmera na mesma cena para várias finalidades. Imagine, por exemplo, a Realidade Virtual. É necessário que haja uma câmera para cada olho do usuário do capacete, e podemos adicionar à nossa cena duas câmeras justamente a fim de cumprirem essa função. Outro exemplo de utilização de mais de uma câmera é quando queremos mostrar os retrovisores em um carro de corrida, ou mesmo mostrar uma segunda visualização de um elemento qualquer em um canto da tela, enquanto a câmera principal mostra os outros elementos, os conhecidos HUDs. São várias utilidades! O Unity permite configurarmos essas câmeras múltiplas de acordo com as propriedades do componente Camera, o qual é adicionado por padrão a todos os objetos do tipo Camera.

Com esses dados em mente, vamos, então, conhecer a câmera do Unity e todos os seus componentes. Observemos, na **Figura 4**, uma captura de tela do Unity contendo as propriedades que já vêm pré-configuradas ao se criar uma nova cena em um projeto qualquer dentro de nosso motor.

Figura 04 - Propriedades de uma câmera e como esta é inicializada em um projeto 2D no Unity.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 05 de fev 2017.

Como vemos na **Figura 4**, além do Transform citado anteriormente, temos o componente Camera já adicionado por padrão à nossa câmera. Esse componente configura diversas propriedades da câmera e permite que ela cumpra a sua função, seja de câmera principal ou de uma câmera secundária qualquer, responsável por detalhes pequenos na tela. Vejamos, a seguir, as propriedades mais importantes e como elas devem ser utilizadas.

Clear Flags

A propriedade Clear Flags determina a maneira como a tela será preparada para que os objetos incluídos na cena sejam desenhados posteriormente. Essa propriedade pode receber quatro valores diferentes, os quais influenciam diretamente no modo que o seu cenário é desenhado. Esses valores são:

- **Skybox** - Ao escolher esse valor para a propriedade, o Unity utiliza uma textura, a qual envolve todos os elementos presentes em sua cena para ser sempre esse o fundo do que está sendo desenhado na câmera. Imagine, nesse caso, a câmera e todos os elementos da cena estarem dentro de uma esfera e tudo que aparecesse ao fundo ser justamente a textura colocada nessa esfera. O Skybox deve ser definido ou, por padrão, será alterado para uma cor sólida. Utilizado principalmente em jogos 3D.

- **Solid Color** - A opção de cor sólida indica que todo o fundo, o qual não tenha nada desenhado sobre ele, aparecerá com uma cor sólida, determinada de acordo com a propriedade Background, explicada mais adiante.

- **Depth Only** - A propriedade Depth Only é utilizada principalmente quando há uma câmera criada exclusivamente para desenhar um elemento simples, o qual fique acima da imagem criada pela câmera principal, seja ele de interface, de HUD, ou qualquer outro, como uma arma em um jogo em primeira pessoa, estando tal arma sempre à frente. Essa opção deve ser utilizada para objetos estáticos e nunca em câmeras principais.

- **Don't Clear** - Essa opção quase nunca é utilizada em jogos. Nela, a câmera nunca limpará o frame anterior antes de desenhar o próximo, resultando em uma grande sobreposição de imagens que terminará por criar um grande borrão na tela.

Background

Essa propriedade representa a cor a ser aplicada ao fundo do que for desenhado pela câmera e não estiver sobreposto por nenhum elemento. Também pode ser utilizada caso o Skybox não esteja presente e a câmera esteja selecionada para esse modo.

Culling Mask

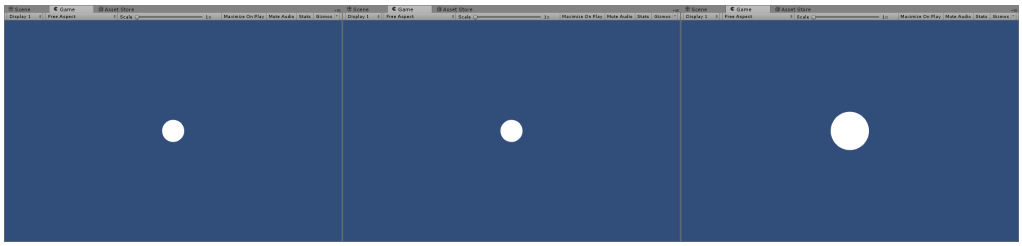
A propriedade Culling Mask é responsável por selecionar quais camadas (layers) serão desenhadas por aquela câmera especificamente. Ao criarmos elementos de interface gráfica, por exemplo, podemos (e devemos) definir uma camada de UI e colocar os elementos como parte dela. Feito isso, precisamos, então, posicionar a câmera de UI e permitir a ela renderizar apenas elementos dessa camada, utilizando a propriedade de Culling Mask, de modo a não mostrar nada mais que a UI.

Projection

Essa propriedade é uma das mais importantes quando se trata da configuração da câmera. Ela define se a câmera simulará, ou não, perspectiva de profundidade. Se o valor **Perspective** for selecionado para a propriedade, a câmera manterá os valores de profundidade dos objetos intactos e os desenhará na tela, fazendo você perceber quais objetos estão mais perto da tela e quais estão mais longe, conforme o tamanho, por exemplo.

Caso a propriedade **Ortographic** seja selecionada, todos os objetos serão desenhados como se fizessem parte de um mesmo plano, sem qualquer senso de perspectiva e de maneira completamente uniforme. Esse modo de câmera pode ser utilizado, por exemplo, em jogos 2D em geral. Também é interessante em alguns jogos com visão de cima, sejam de estratégia ou outros gêneros. A **Figura 5**, a seguir, mostra um círculo desenhado na posição (0,0,0), à esquerda e, em seguida, na posição (0,0,-5), no meio e à direita. Na imagem do meio, vemos a câmera ortográfica. Na direita, a câmera em perspectiva. Observe a diferença!

Figura 05 - Círculo em (0,0,0) com câmera ortogonal (esq.), círculo em (0,0,-5) com câmera ortográfica (centro) e círculo em (0,0,-5) com câmera em perspectiva (dir.).



Fonte: : Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 05 de fev. de 2017

Você nota a diferença entre as imagens da esquerda e do meio? Não? Olhe de novo! Ainda não vê? Pois é! Não há. A câmera ortográfica não diferencia o tamanho do objeto de acordo com a proximidade dela justamente por não ter uma noção de profundidade atrelada a ela. Isso é útil em jogos 2D, como os que desenvolveremos ao longo desta disciplina! Assim, utilizaremos apenas câmeras ortográficas de agora em diante, ok? Na verdade, as utilizamos desde sempre, pois são o padrão para projetos 2D! :D

Size (ou Field of View)

A propriedade Size, exclusiva das câmeras ortográficas, indica o tamanho do campo de visão que a câmera terá quando desenhar os objetos. Para efeitos práticos, imagine ser essa propriedade equivalente ao zoom de sua câmera. Quanto maior o valor da propriedade, menos zoom a câmera terá e mais elementos serão mostrados, com uma menor dimensão. Quanto menor o valor, mais zoom a câmera terá e maior serão os objetos mostrados, ao custo de mostrar menos objetos.

Caso sua câmera esteja configurada em perspectiva, a propriedade será substituída por Field of View, o qual tem a mesma ideia de Size, mas é expresso em ângulos do Frustum de visão. E o que é isso? Veremos em Motores II!

Clipping Planes

A propriedade Clipping Planes determina a área que será visível pela câmera dentro do mundo do nosso jogo. Essa propriedade é dividida em duas sub-propriedades: **Near** e **Far**. A sub-propriedade **Near** indica **a partir** de qual valor os objetos que estão à frente da câmera começarão a ser desenhados. Já a sub-

propriedade **Far** indica **até** qual valor de profundidade os objetos serão desenhados. A propriedade Clipping Planes tem os mesmos efeitos tanto em projeção ortográfica quanto em perspectiva.

A ideia é o **Near** ser utilizado para evitar objetos muito próximos à câmera serem desenhados e ocuparem toda a tela, escondendo todos os outros objetos que estão atrás dele. Já o **Far** deve ser utilizado para evitar ser gasto processamento em objetos tão afastados que nem representariam um pixel completo na tela se desenhados, ou seja, nem apareceriam. Os valores padrões para o Near e o Far são 0.3 e 1000, respectivamente.

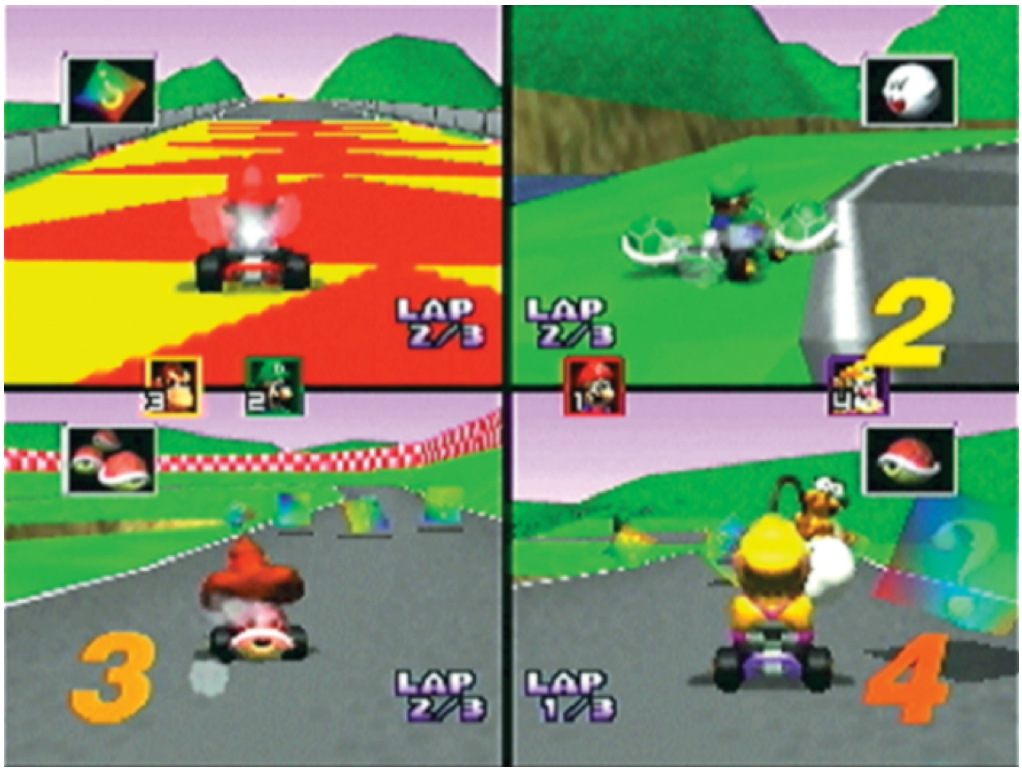
Viewport Rect

Essa propriedade, assim como a anterior, é dividida em sub-propriedades. No caso específico do Viewport Rect, elas indicam onde a câmera será desenhada na tela do usuário. Lembra das Viewport Coordinates citadas anteriormente? Aqui que elas se aplicam! Você deve indicar, através dessa propriedade, qual o X e o Y inicial em coordenadas de Viewport em que a câmera deve ser posicionada e também qual o tamanho e a largura da câmera (W - Width e H - Height).

> Lembre-se que as Viewport Coordinates variam apenas de 0 até 1, então, cuidado com isso ao escolher a posição na qual sua câmera estará.

Através dessa propriedade podemos criar câmeras que atuam apenas em alguma pequena parte da tela, como o retrovisor de um carro, por exemplo, as duas câmeras para a Realidade Virtual, ou até mesmo aquela clássica divisão em quatro telas para fazer um multiplayer local split screen (saudades).

Figura 06 - Quatro jogadores dividindo a mesma tela no jogo Mario Kart 64



Fonte: http://www.hardcoregamer.com/wp-content/uploads/2014/08/mariokart64-570x428.png

Ufa! Quantas propriedades esse componente tem, hein! E todas com alguma importância para o que queremos desenvolver. Algumas até se subdividindo. Bastante coisa mesmo! Vale a pena, agora, parar um pouco e dar uma olhada em cada propriedade calmamente, sem pressa, de modo a se familiarizar com os conceitos. Caso as dúvidas persistam, não fique tímido em procurar ajuda nos fóruns! Vai lá! ;)

Ah! E as propriedades mais avançadas não serão cobradas nos exercícios e estão como curiosidade na Leitura Complementar - pt.2. Dá uma olhada!

Atividade 02

---

1. Para fixar o que estudamos, vamos abrir o Unity e criar uma nova cena. Em seguida, criaremos um sprite de círculo e o adicionaremos à cena, centralizado. Feito isso, adicione a ele um Rigidbody 2D a fim de que possa cair ao utilizarmos o Play Mode. Por fim, altere as propriedades de câmera e brinque um pouco com o seu sprite para entender bem cada propriedade.

Técnicas de Movimentação da Câmera em 2D

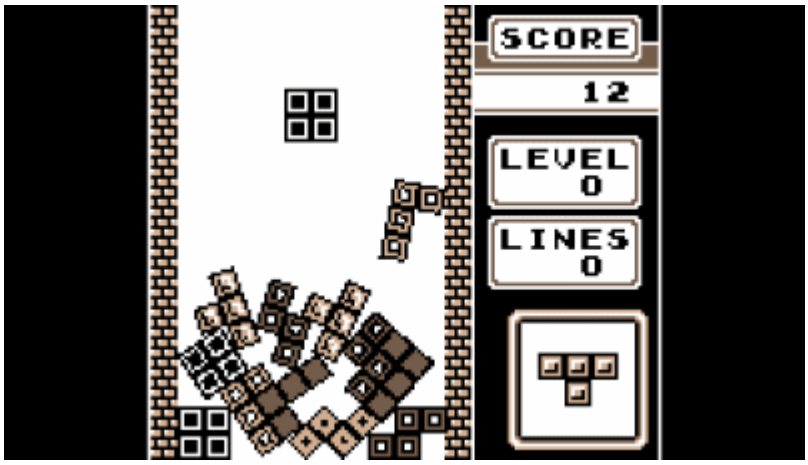
Agora que já conhecemos o componente Camera e suas propriedades, podemos passar a um ponto bem interessante relacionado às câmeras: as técnicas de movimentação delas.

Vimos até aqui a existência de diversos tipos de jogos que podemos desenvolver, mesmo em 2D. Cada um desses jogos requer um tipo específico de câmera e alguns detalhes importantes relacionados ao seu posicionamento. Utilizaremos esta seção para discutir alguns desses tipos de câmera e como podemos aplicá-los aos diferentes gêneros de jogos.

Câmera Estática

Até agora, devido ao padrão em que os projetos são criados no Unity, temos utilizado em nosso jogo uma câmera estática. Esse tipo de câmera não tem qualquer movimentação e fica em apenas um ponto, exibindo o que estiver à sua frente. Esse tipo de câmera, ainda assim, tem suas funcionalidades.

Figura 07 - Jogos de puzzle costumam utilizar câmeras estáticas.



Fonte: Giphy (https://media.giphy.com/media/5Tndtit6LsZmE/giphy.gif).

Apesar de não ser adequada para a utilização em jogos de plataforma, ou side-scrollers em geral, como vemos ao executar o nosso projeto, podemos, em jogos do tipo puzzle, utilizar uma câmera dessa exibindo todo o cenário para permitir ao jogador se situar bem no ambiente e tomar suas decisões de jogo conforme aquilo que está vendo, por exemplo.

Câmera Atrelada a um Objeto

Utilizando a ideia de hierarquia, como discutimos anteriormente, podemos também atrelar a câmera a um objeto gráfico e fazer este a levar para onde for, permitindo, assim, haver uma centralização do objeto selecionado sempre, uma vez que a câmera está atrelada a ele.

Esse tipo de comportamento pode ser visto em diversos tipos de jogos, como em primeira pessoa, em terceira pessoa, top-down e até mesmo side-scrollers, variando apenas a posição na qual a câmera e seu alvo são colocados. Ele pode ser feito de maneira muito simples no Unity, utilizando apenas a hierarquia. Entraremos em detalhes de como fazer esse tipo de câmera na seção seguinte, quando formos adicionar uma câmera assim ao nosso jogo.

Figura 08 - Câmera atrelada diretamente ao personagem.

Fonte: http://www.gamasutra.com/CamerasinSideScrollers.php

Movimentação de Câmera Através de Scripts

Outra opção que temos para controlar a câmera em nosso jogo é através da utilização de scripts. Somos capazes, utilizando scripts na câmera, de adicionar efeitos de movimentação a ela, além, simplesmente, de mantê-la estática ou fazê-la seguir um alvo de acordo com a posição deste. Podemos, por exemplo, adicionar efeitos de transição à câmera a fim de que ela se reposicione ao mudar de um lado para o outro, ou até mesmo criar um balanço natural durante a movimentação do personagem.

Figura 09 - Câmera seguindo o personagem em apenas um eixo, definida programaticamente.

Fonte: http://www.gamasutra.com/CamerasinSideScrollers.php

Perceba que a adição de scripts à câmera acontece, normalmente, da mesma maneira como fazemos ao lidar com outros GameObjects. Basta adicionar um componente novo do tipo script e adicionar a esse componente os códigos necessários para que a câmera aja conforme o esperado. Um bom exemplo de movimentação de câmera através de scripts pode ser encontrado no próprio projeto de exemplo que o Unity tem, em 2D. Caso seja de seu interesse, carregue esse exemplo, como vimos nas primeiras aulas, e dê uma olhada no script!

Adicionando Movimentação da Câmera ao Projeto DMJ I

Agora que já conhecemos bastante acerca das câmeras no Unity, vamos adicionar um novo comportamento à nossa câmera do projeto! Para vermos os dois tipos de câmera dos quais falamos, além da estática, já utilizada, adicionaremos a movimentação de câmera de duas maneiras ao nosso projeto: uma atrelando a câmera ao personagem e outra utilizando um script supersimples! Aí você escolhe qual delas deseja manter. Vamos lá?

Atrelando a Câmera ao Jogador

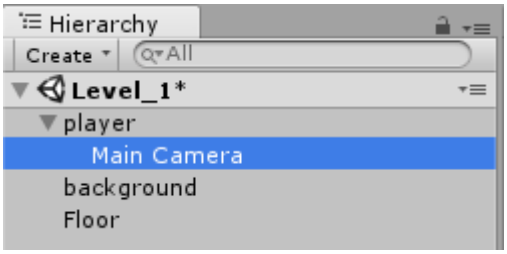
A primeira maneira a ser abordada é a mais simples. Para garantir que a câmera seguirá o nosso personagem ao longo de seu curso, simplesmente a adicionaremos como filha do personagem. Com isso, toda movimentação que for aplicada no

personagem também será aplicada na câmera. Isso é interessante até determinado ponto, pois quando ele pular ou mesmo cair, pode gerar algum desconforto. Vamos fazer desse modo e ficará a seu critério aproveitar a ideia ou não!

Primeiramente, retomaremos o projeto encerrado na aula anterior. Para isso, abra o projeto DMJ I no Unity e, em seguida, abra a cena Level_1. Lembrando: aqueles que se perderam ao longo da execução podem criar um novo projeto do zero e importar os assets disponíveis aqui.

Após abrir o nosso projeto, precisamos apenas clicar na Main Camera e arrastá-la até o player, no Hierarchy, como já vimos em outras oportunidades. Feito isso, a câmera será posicionada como filha do player, criando, assim, um nível a mais na hierarquia dele, como vemos na **Figura 10**.

Figura 10 - Câmera aninhada no player como filha.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 05 de fev. de 2017

Depois, se você clicar em play, verá que a câmera já está seguindo o jogador. O problema, no entanto, diz respeito ao jogador estar centralizado na câmera, de modo a perder espaço, mostrando o que está abaixo do chão, o qual, na prática, não teria utilidade alguma.

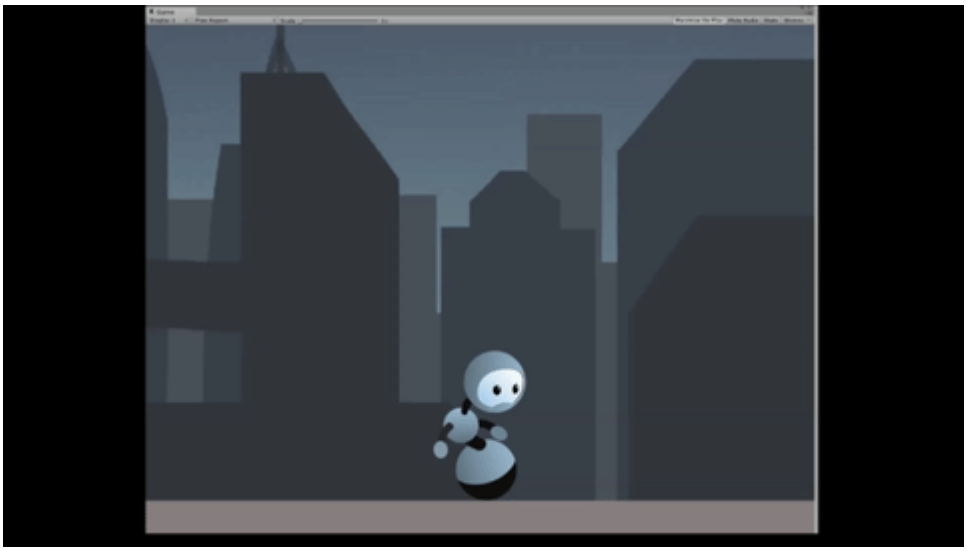
Para evitar isso acontecer, configuraremos a posição da câmera em relação ao player, o que. Perceba que isso é feito a fim de criar jogos em primeira pessoa e em terceira pessoa. Basta reposicionar a câmera de maneira que ela fique no local desejado e pronto!

No nosso caso, é necessário somente selecionar a câmera e alterar o Transform dela para o Y ter o valor 2.75. Feito isso, a câmera se posicionará um pouco acima do player, mas ainda seguindo-o e exibindo apenas a parte do cenário mais

interessante ao jogador. Fácil, não? Para melhorar o efeito geral do cenário, altere também o background para Y = 3.75 e a escala para (3,3,3). Isso fará com que ele se posicione melhor para o personagem poder se mover, sem perder espaço.

Veja na **Figura 11** como é o comportamento dessa câmera.

Figura 11 - Personagem se movendo com a câmera atrelada.



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

Perceba que, ao levantar o personagem do chão, a câmera sobe junto ao personagem e nós perdemos um pouco a noção de onde está o chão. Isso é uma das peculiaridades de se utilizar esse tipo de câmera. Experimente um pouco e veja o que acha!

Adicionando um Script à Câmera

Uma outra opção que temos para a câmera ser capaz de seguir o personagem é adicionar um script a ela . O script pode ser feito de várias maneiras e utilizar diversas técnicas para atingir o objetivo de seguir o jogador. Faremos aqui, no entanto, um script bem básico, apenas para mostrar como é o funcionamento da ideia.

Mais uma vez abriremos o nosso projeto onde o deixamos na aula passada, ou mesmo salvaremos separadamente a cena criada na seção anterior e começaremos outra nesta seção. Feito isso, selecionaremos a câmera principal e adicionaremos um novo script C# nela. Coloquei o nome do script CameraMovementScript. Fique à

vontade para escolher o nome do seu! Lembre-se que, assim como vimos em Java, na disciplina de Programação Estruturada, o nome do arquivo e da classe deverão sempre ser iguais! Se trocar o nome do arquivo, preste atenção ao nome da classe. E não esqueça de mover o script para a pasta adequada nos assets!

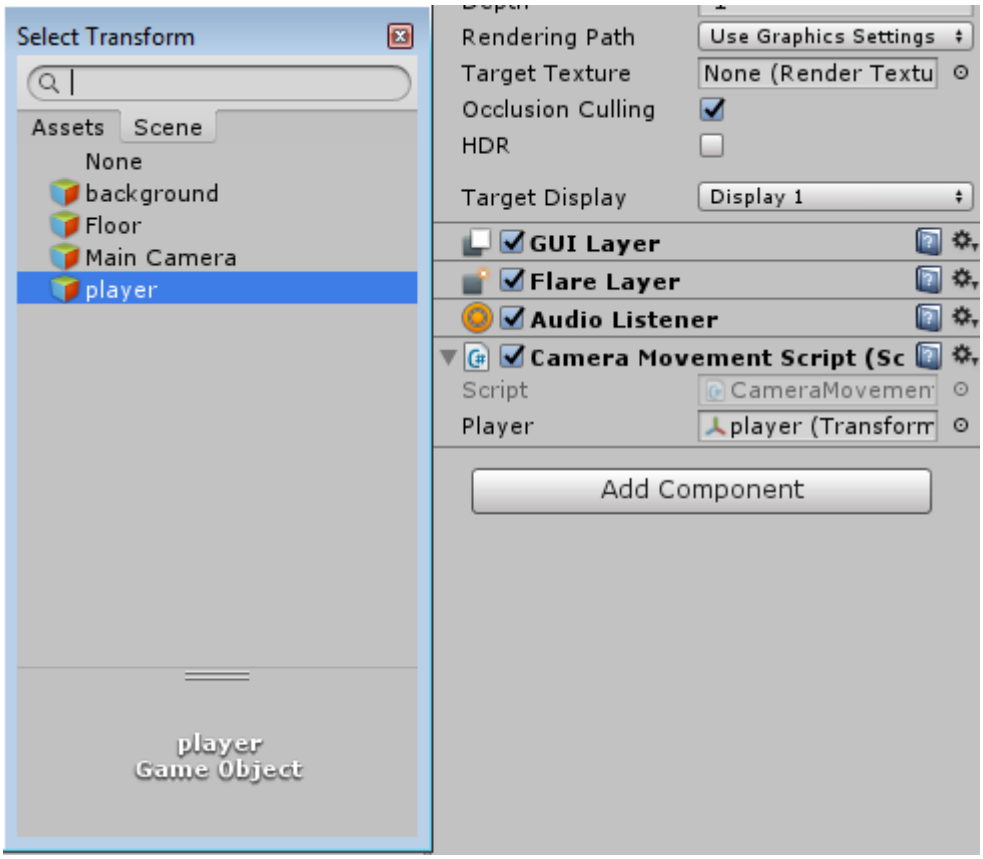
Logo após, adicionaremos à câmera o script demonstrado na **Listagem 01**, a seguir.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class CameraMovementScript : MonoBehaviour {
6
7      public Transform player;
8
9      // Use this for initialization
10     void Start () {
11     }
12
13     // Update is called once per frame
14     void Update () {
15         transform.position = new Vector3(player.position.x, transform.position.y, transform.position.z)
16     }
17 }
```

Listagem 01 - Script para que a câmera possa seguir o personagem.

Perceba que o script tem um public Transform player, o qual deve ser configurado no editor para referenciar o objeto que gostaríamos de seguir. A **Figura 12** mostra o player sendo selecionado como objeto a ser seguido.

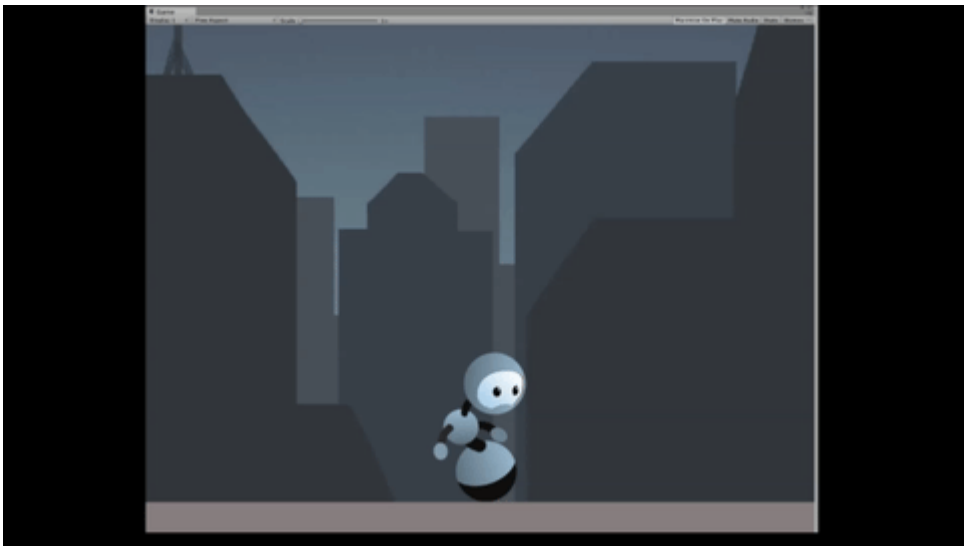
Figura 12 - Player sendo selecionado como objeto a ser seguido.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/. Acesso em: 05 de fev. de 2017.

Falando do script em si, há apenas uma linha de código no Update que simplesmente pega a posição em X do player e modifica a posição da câmera para essa posição. Perceba que o Y nunca é alterado. Isso criará uma câmera um pouco diferente, parecida com um joguinho ali... um tal de Super Mario Bros. Veja na **Figura 13** o resultado dessa escolha de câmera.

Figura 13 - Câmera seguindo o jogador por script.



Fonte: Captura de tela do Unity. Disponível em: https://unity3d.com/pt/ Acesso em: 05 de fev. de 2017.

Perceba que quando o jogador começa a subir, a câmera não o acompanha, permitindo-lhe até mesmo sair da tela por cima. Quando configurarmos o jogador para pular adequadamente, isso não acontecerá, dado o limite do pulo, mas, por enquanto, já conseguimos ver que há uma diferença entre as câmeras.

E por hoje é só, pessoal! Espero que tenham aprendido bastante sobre os elementos gráficos e principalmente sobre a câmera do Unity, seu funcionamento e seus usos. Qualquer dúvida, é só conferir lá no fórum, pois estaremos felizes em trocar uma ideia! Até a próxima!

Leitura Complementar

Manual do Unity para a Câmera: https://docs.unity3d.com/Manual/class-Camera.html

Vídeo oficial sobre as propriedades da câmera: https://unity3d.com/tutorials/cameras

Artigo completo, em inglês, indicando tipos diversos de câmera, incluindo os estudados nesta aula: http://www.gamasutra.com/CamerasinSideScrollers.php

---

Depth

Essa propriedade determina qual a posição da câmera na ordem de desenho. Deve ser utilizada quando há mais de uma câmera na cena. As câmeras que possuem o maior valor de Depth são desenhadas acima das câmeras que possuem o menor valor, ou seja, se você quer desenhar uma GUI, como havíamos discutido anteriormente, essa câmera deve possuir o maior valor de Depth, garantindo, assim, estar sempre por cima das outras câmeras no desenho dos frames.

Rendering Path

Essa propriedade é um pouco mais avançada, portanto não entraremos em detalhes no momento. Ela define qual o método de renderização a ser utilizado pela câmera. Para entender melhor, precisaríamos abordar materiais, iluminação, shaders e outros assuntos que não cabem a esta disciplina. Deixaremos sempre o valor padrão! Aos interessados, o material do Unity, referenciado no fim da aula, contém mais informações sobre essa propriedade.

Target Texture

Essa opção nos permite selecionar, como o alvo de renderização dessa câmera, uma **Render Texture** que tenha sido adicionada à nossa cena. Com isso, a câmera perde suas capacidades de desenhar na tela e passa a trabalhar exclusivamente

para desenhar nessa textura. Isso pode ser útil para simular um espelho, por exemplo, no qual o seu conteúdo é atualizado na tela de acordo com o que estiver em sua frente a cada momento.

Occlusion Culling

Essa propriedade determina se objetos escondidos por trás de outros objetos e, por esse motivo, não mostrados na tela, devem ou não ser desenhados pela câmera. Caso esteja habilitado, menos objetos serão desenhados, de acordo com a capacidade da câmera de visualizá-los ou não. Isso pode melhorar a performance, uma vez que muitos deles não serão desenhados desnecessariamente.

HDR

Essa é mais uma propriedade que não discutiremos aqui, por se relacionar a aspectos mais complexos do motor. HDR significa High Dynamic Range, e ativar essa propriedade habilita tal funcionalidade para essa câmera. Basicamente, as cores são representadas em valores de 0 a 1 para Red, Green e Blue, a cada pixel. Habilitando a HDR, criam-se novas opções de valores para essas cores, tornando-as mais próximas do que temos na vida real. Aos curiosos, recomendo procurar o funcionamento de monitores HDR e como isso se aplica ao Unity!

Target Display

Essa última propriedade diz respeito à qual dos displays conectados à máquina a câmera deverá renderizar quando for desenhar o seu conteúdo. Os valores podem variar de 1 a 8 e correspondem ao número do display que a câmera desenhará quando for acionada.

Resumo

Na aula de hoje, conhecemos os sistemas de coordenadas utilizados em computação gráfica em geral. Em seguida, conhecemos os sistemas de coordenadas que o Unity utiliza. Vimos também o elemento câmera, principal assunto da aula, de modo a conhecer cada propriedade do componente Camera.

Para finalizar, aprendemos algumas técnicas de movimentação de câmera em 2D e vimos como podemos implementar duas dessas técnicas em nosso projeto do joguinho do robô. Mais uma vez, aqueles que desejarem podem obter o código atualizado desta aula no seguinte link.

Com isso, concluímos mais uma etapa da construção do nosso jogo! Para a nossa próxima aula, temos mais um assunto muito importante planejado: mecânicas de pulo! Vamos fazer o personagem tirar o pé do chão!

Autoavaliação

1. Cite os quatro principais sistemas de coordenadas utilizados em computação gráfica.
2. Cite os sistemas de coordenadas do Unity e informe onde cada um deles deve ser utilizado.
3. Você concorda com a utilização do nome Local Coordinates?
4. Quais os principais sistemas de movimentação de câmera? Pense no seu jogo 2D favorito. Qual a movimentação de câmera que ele utiliza?

Referências

Documentação oficial do Unity - Disponível em: https://docs.unity3d.com/Manual/index.html.

Tutoriais oficiais do Unity - Disponível em: https://unity3d.com/pt/learn/tutorials.

RABIN, Steve. **Introdução ao Desenvolvimento de Games**, Vol 2. CENGAGE.