

WORKSHOP

CRIANDO JOGOS DO ZERO

AULA 2



[Youtube.com/crieseusjogos](https://www.youtube.com/crieseusjogos)



[@crieseusjogos](https://www.instagram.com/crieseusjogos)

POR TRÁS DAS CENAS

> INTRODUÇÃO

Na aula passada eu te mostrei os fundamentos essenciais de todo desenvolvedor de jogos. Entendemos também, os passos cruciais a serem tomados antes de se criar qualquer projeto, além de sabermos mais sobre esse mercado bilionário.

Então se você não viu ainda a primeira aula, recomendo que pare de ler este PDF agora mesmo e volte a [página do workshop](#). É essencial que você assista a primeira aula para entender o que falarei nessa aula, beleza?

Esta aula terá foco em te mostrar o que existe por trás de um jogo, basicamente tudo o que se faz necessário para termos um jogo completo.

Com isso, veremos sobre o game design, engines e linguagem de programação.

Nessa aula eu também vou quebrar alguns mitos que te impedem de se tornar um desenvolvedor de jogos reconhecido.

Então bora lá?

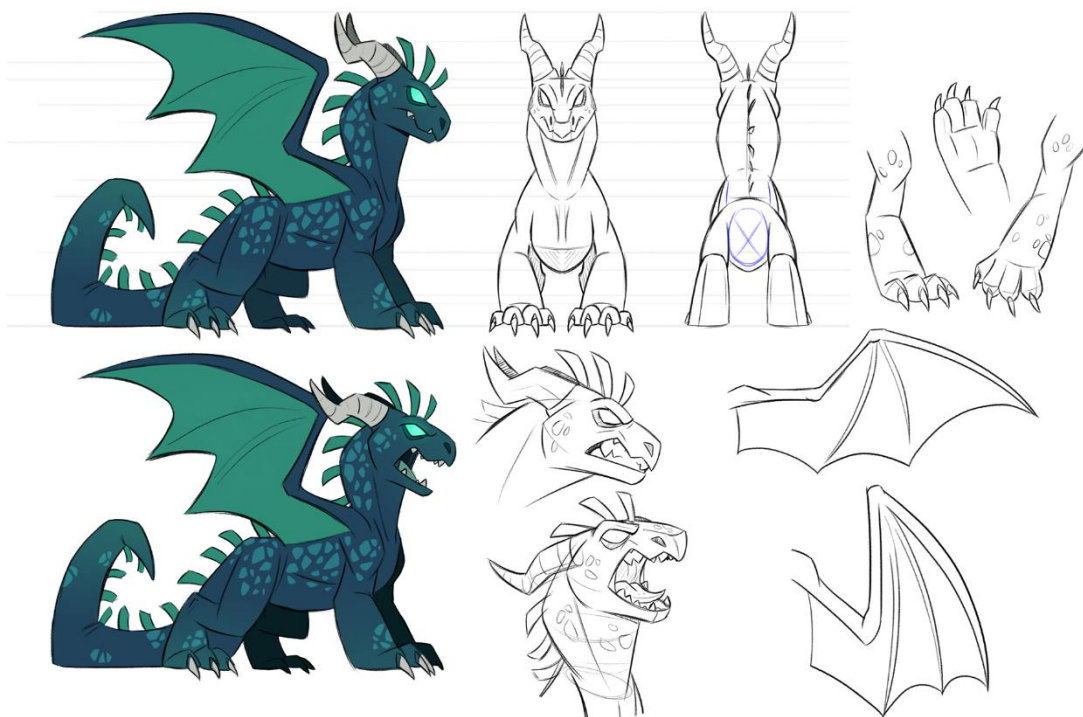
> CONCEITOS BÁSICOS DE GAME DESIGN

Neste momento eu gostaria muito dar uma ênfase nessa área que é, infelizmente, subestimada por muitos desenvolvedores de jogos e até mesmo confundida: o **game design**.

O conceito básico de game design é a criação e planejamento de elementos, regras e dinâmicas de um jogo. Mas fique atento! Nada de confundir game design com design propriamente dito, são tarefas distintas.

Popularmente no Brasil, conhecemos o cara que cria artes para games como designer e isso causa muita confusão.

Enquanto o game design trata da organização do projeto inteiro, o “design” nada mais é do que as artes do seu projeto, como animações, modelos 3D, sprites e etc.



Exemplo de arte criada por um artista de jogo (popularmente chamado de designer)



Exemplo de GDD, documento criado por um Game Designer

Só para você ver a importância do que estou falando, o papel do game design em grandes empresas é realizado por um profissional específico, chamado de game designer.

Ele é o cérebro de todo o projeto. É ele quem amadurece a ideia do jogo, as interações, o enredo, as regras e todos os elementos que deverão existir no game. Tendo dito isso, agora nós conseguimos ter uma noção da importância de um bom game design em um jogo, não é mesmo?

Em projetos independentes, geralmente esse papel é feito pelo próprio desenvolvedor do jogo, o que não tem problema algum.

Logo, sabemos que o game design é de suma importância para um projeto de sucesso. Quando bem feito, afeta até mesmo o desenvolvimento positivamente. Já que por tudo estar devidamente documentado e definido, o artista e o programador podem focar seus esforços no que importa.

Só para você ter uma noção das técnicas de game design em um jogo, uma das premissas dessa área é a de que um jogo não pode ser chato, devendo haver um equilíbrio geral entre elementos.

É muito difícil manter aquela excitação inicial de um jogador ao começar a jogar um jogo novo por muito tempo. Aposto que você mesmo já tenha deixado muitos jogos de lado ao começar a jogar por não achar interessante o suficiente para você.

Isso pode ter acontecido por conta de uma pequena falha no planejamento do jogo em uma área específica. Agora, adivinha em qual área? Isso mesmo, o game design.

Para evitarmos que isso aconteça com nossos projetos, devemos considerar o que é interessante ou não para o jogador.

E esse interessante não significa que seu jogo precisa ter milhares de elementos acontecendo ao mesmo tempo ou gráficos extremamente realistas. Às vezes apenas fazer com que o jogador tenha que pensar em escolhas certas já poderá ser o suficiente. Quer um exemplo do que estou falando?



Jogo Copas

Em um jogo de baralho, mais especificamente Copas, por exemplo, o jogador deverá pensar em inúmeras possibilidades antes de realizar uma ação, tais como qual carta ele deverá

passar ao adversário, esperar ou não alguém jogar primeiro, qual carta jogar e etc.

E saca só, cada escolha dessa é automaticamente reavaliada a cada ação que ocorre no jogo, dependendo da sua mão e das jogadas que os adversários fizeram.

A cada instante o jogador é forçado a pensar em ações futuras e presentes, a fim de claro, vencer a partida. Legal, não é mesmo?

Aí entra o conceito do game design de que todo jogo deve haver ações que sejam extremamente interessantes ao jogador. Em outras palavras, seu jogo não pode, de forma alguma, ser chato. E chato não está relacionado a gráficos, vide o exemplo citado, mas sim a mecânica.

Com isso, podemos entender que os jogadores têm que aplicar seu tempo, concentração e habilidades em resolução de problemas aos desafios que seu jogo lança neles.

Deve haver uma pontuação para esses esforços, uma recompensa pelo seu investimento. Quando o jogo termina, os jogadores devem sair sentindo que a experiência foi significativa, que aquilo ali valeu a pena ter sido jogado.

Conseguiu perceber que absolutamente todos os jogos de sucesso possuíram um game design bem produzido, que os permitiram fazer sucesso muito além de aparências, mas sim através da mecânica introduzida por cada game? É exatamente esse ponto que eu queria que você compreendesse.

> AS ENGINES

Conforme já vimos rapidamente na aula passada, uma engine é o software que fornece aos desenvolvedores o conjunto necessário de recursos para se criar jogos de uma maneira rápida e eficiente.

De modo geral, uma engine é um framework de desenvolvimento de jogos que suporta e reúne várias áreas importantes. Você pode importar gráficos e recursos 2D e 3D de outros softwares, tais como o Maya, 3DS Max ou Photoshop, montá-los em cenas, criando assim ambientações. Esses ambientes podem conter iluminação, áudios, efeitos especiais, física, animação e interatividade diversa.

As engines modernas permitem trabalhar com gráficos realmente incríveis de uma maneira simples. A importação de recursos é bastante rápida e é possível também trabalhar com arquiteturas de alto desempenho gráfico.

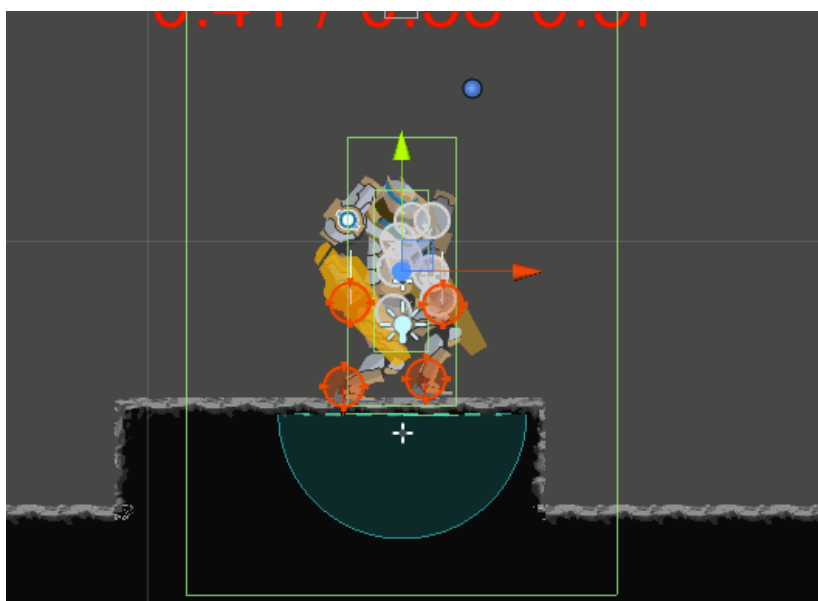


Cena de projeto desenvolvido na Unity

Outro ponto importante é a conectividade em games. Hoje em dia jogos online já são uma realidade, seja no PC ou em dispositivos móveis.

Felizmente, muitas engines fornecem componentes prontos, os famosos plugins, e APIs para serem utilizadas em seus projetos de uma maneira simplificada para a criação de games online e uso de multiplayer.

O sistema de física em uma engine é capaz de simular diversos tipos de gravidade para você. Não é necessário codificar todo um sistema de colisão, por exemplo. A engine simplesmente te dá isso pronto!



A interface de usuário é de extrema importância para o jogador do seu game. Pode acreditar, ela é a primeira coisa que o jogador irá ver ao baixá-lo.

E nada melhor do que trabalhar com interface de usuário de uma maneira em que seus esforços sejam minimizados para atingir seus objetivos. Muitas engines possuem um sistema personalizado de UI (interface de usuário), facilitando todo o processo.



Exemplo de Interface de Usuário feita na Unity

Por fim, temos os scripts. Talvez você nunca tenha tido um contato com uma engine, por conta disso, saiba que os scripts são a programação do seu jogo. Com eles nós podemos definir a lógica de todo o jogo, definindo o comportamento de cada elemento.

Praticamente tudo que exerce uma ação em seu jogo possuirá um script controlando ações por trás. Um movimento de personagem, a execução de uma animação, um efeito que se inicia, a colisão entre objetos, até mesmo um botão que executa uma ação... tudo isso é definido via script.

E como é a cara desse script? Saca só esse script que eu criei:

```
Assets > Scripts > C# Ball.cs
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  |
5  public class MeuScript : MonoBehaviour
6  {
7
8      // Start is called before the first frame update
9      void Start()
10     {
11
12     }
13
14     // Update is called once per frame
15     void Update()
16     {
17
18     }
19 }
20
```

Perceba a estrutura dele. Todos scripts são criados exatamente assim por padrão, a única coisa que muda de um para outro é o nome da classe, que corresponde ao mesmo nome do script!

E, já para termos uma pequena introdução à lógica de programação, que tal entendermos agora mesmo como funciona essa estrutura?

> INTRODUÇÃO À PROGRAMAÇÃO EM C#

Neste workshop trabalharemos com a engine Unity. Trata-se da engine mais utilizada no mundo quando o assunto é desenvolvimento de games.

Além disso, sua versatilidade e fácil curva de aprendizado contribuem para que possamos dar os primeiros passos aqui mesmo no workshop, ok?

NOTA: *Em nosso canal no Telegram foi enviado um tutorial que ensina a instalar a Unity caso você tenha alguma dificuldade.*

Scripts

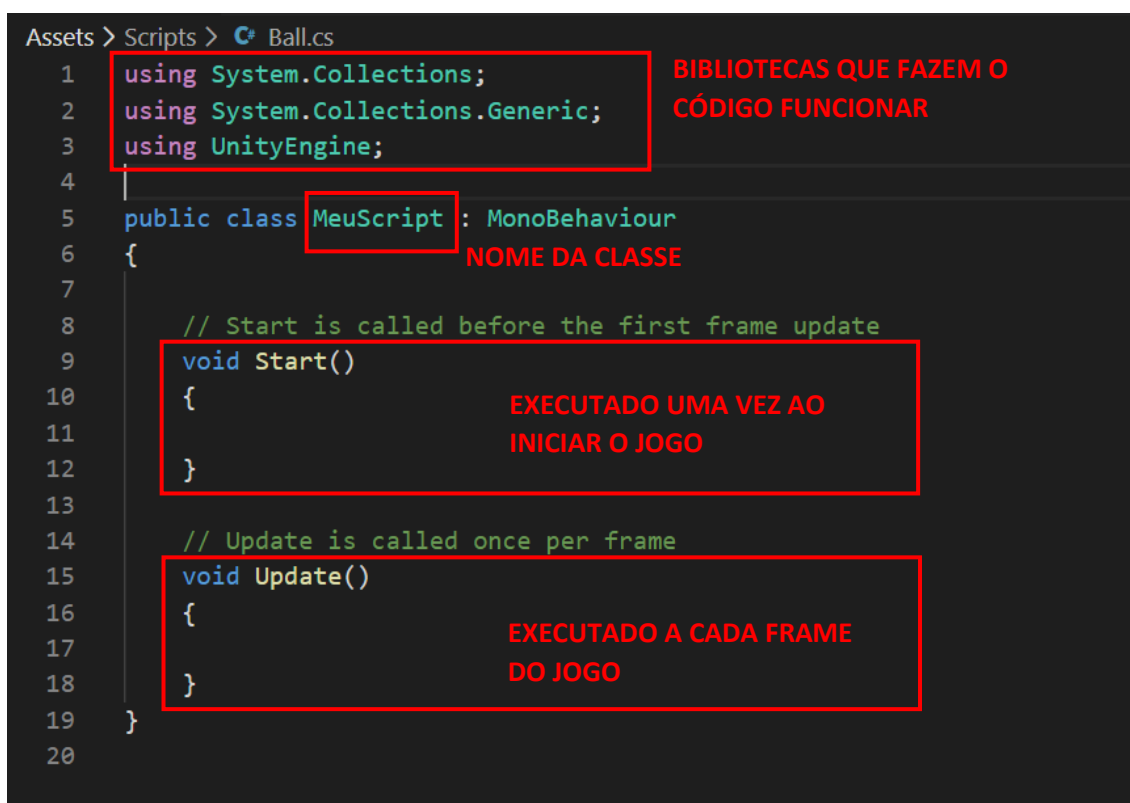
Um script é um arquivo de texto que faz uma conexão interna com a Unity. Então tudo o que é escrito dentro do script é lido pela Unity. Todos esses scripts derivam de uma classe interna da Unity, chamada **MonoBehaviour**.

Classe essa que controla as ações da engine. O que temos que fazer nesses scripts é literalmente escrevermos ações a serem realizadas de acordo com nossas necessidades e desejos.

Então, por exemplo, se eu quero que um objeto vá para frente sempre que eu pressione o botão “D”, nós temos que escrever, ou seja programar, isso dentro do script.

Mas, o importante a entendermos de cara sobre os scripts da Unity é a funcionalidade desses dois métodos criados por padrão, o método **Start()** e o método **Update()**.

Entenda por método, **um conjunto de tarefas** que podem ser realizadas sempre que esse método é chamado. Então dentro dos métodos nós devemos escrever as ações que desejamos que ele faça. Perceba, inclusive, que existe uma chave que abre e uma que fecha, indicando que o código deve ficar ali dentro:



Começando pelo start, nós podemos entender que tudo que está dentro do Start é chamado sempre que o jogo é inicializado.

Já o método Update é diferente. Ele é chamado a cada frame por segundo em que o seu jogo é rodado. Então imagine, por exemplo, que o método Update() possa ser chamado 60 vezes por segundo!

Com isso, é de extrema importância saber exatamente o que se está colocando dentro de um método Update(), pois senão você poderá cair em um loop infinito e seu computador travar. Não queremos isso, né?

Então em suma temos: o Start é chamado **UMA VEZ ao iniciar um jogo**; já o método Update é chamado **várias vezes por segundo**.

Quer ver uma coisa legal? Vou te mostrar na prática o funcionamento do script.

Saca só essa linha que eu digitei. Nessa linha eu estou dizendo à Unity que eu quero lançar uma mensagem escrita “**Olá, pessoal**” no console da engine, através do uso do **Debug.Log()**:

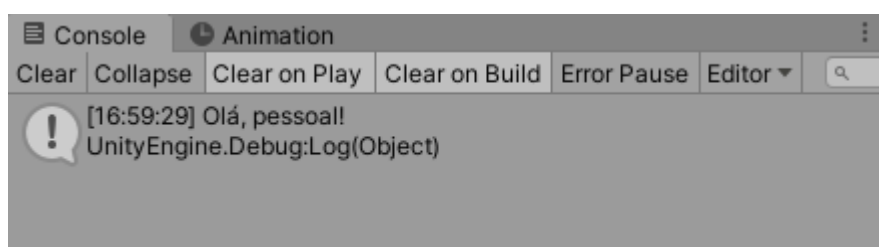
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

0 referências
public class MeuScript : MonoBehaviour
{
    // Start is called before the first frame update
    0 referências
    void Start()
    {
        Debug.Log("Olá, pessoal!");
    }

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

A janela Console na Unity é o local que nos indica mensagens do sistema, mensagens de erro e mensagens de alertas.

Agora veja o resultado do código digitado acima lá na Unity:



Olha só que legal! A Unity lançou para mim no console exatamente a mensagem que escrevi na linha! Isso aconteceu porque como coloquei o código dentro do Start(), assim que iniciei o modo de teste da Unity, modo esse que nos permite

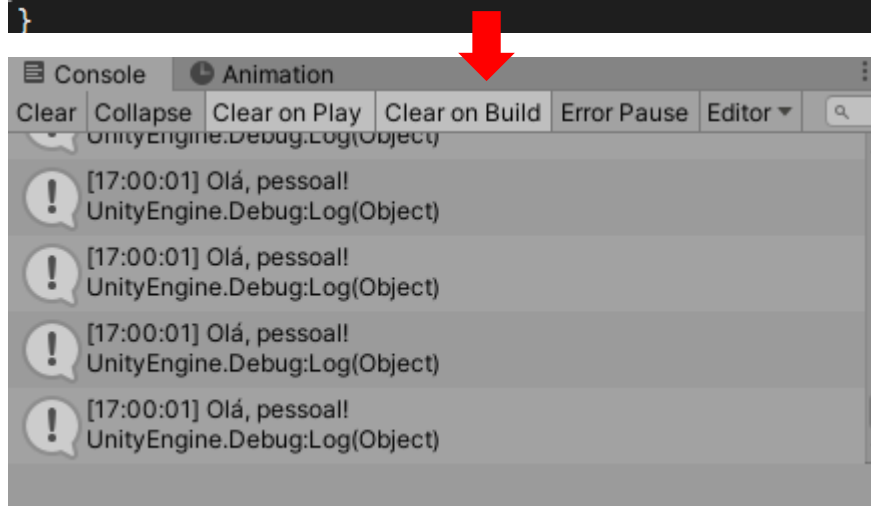
testar o jogo em tempo real, a engine lançou a mensagem apenas uma vez ao início.

Agora veja o resultado se eu colocar a mesma linha no Update():

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

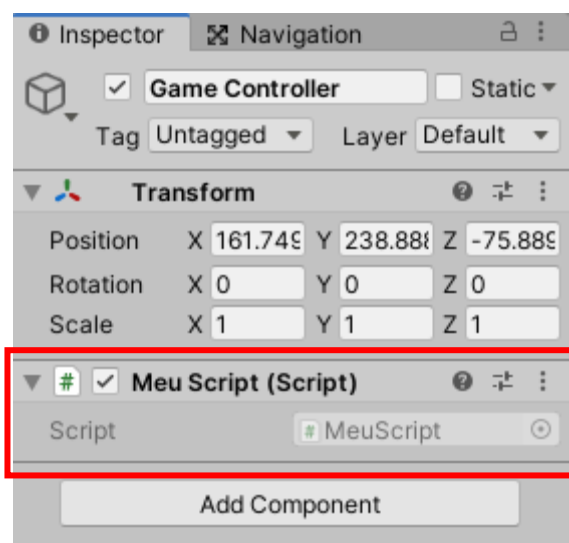
0 referências
public class MeuScript : MonoBehaviour
{
    // Start is called before the first frame update
    0 referências
    void Start()
    {
    }

    // Update is called once per frame
    0 referências
    void Update()
    {
        Debug.Log("Olá, pessoal!");
    }
}
```



A Unity agora está lançando milhares de mensagens por segundo e ficará assim infinitamente se eu deixar. Então essa é a diferença prática entre o método `Start()` e o `Update()`.

Com o script pronto, é hora de colocá-lo para funcionar.



Dentro da Unity o que temos que fazer é extremamente simples, basta anexar o script a qualquer objeto que esteja na cena e pronto! A programação está feita!

Meus parabéns. Você já sabe o que é um método e também já sabe o básico de um script, assim já está começando com o pé direito a ter uma noção de programação. Ah, só para informar, a linguagem de programação utilizada nos scripts da Unity é o C#.

Que tal aprofundarmos um pouco mais nesse tal de C#? Garanto que te explicarei de uma maneira simples e rápida. Além disso, é essencial para que possamos conseguir entender bem a próxima aula, combinado?

O que é uma linguagem de programação?

A definição de linguagem de programação é um conjunto de instruções ou regras para comunicação com um computador. É só pensarmos numa receita de bolo. Na receita, deve-se seguir os passos para o bolo ficar pronto e sem nenhum problema, assim também ocorre na programação.

No nosso caso, a linguagem de programação é utilizada para realizar uma comunicação com a engine. Então, conforme já vimos, através de instruções que são nossos scripts, a linguagem manda a Unity realizar alguma ação.

Mas quais instruções são essas? Como eu as defino?

Para o nosso workshop e principalmente para essa aula é necessário entendermos 4 conceitos básicos da programação orientada a objetos, que são: definição de **variáveis, métodos, classes e objetos**.

Variáveis

Uma variável **armazena um dado na memória do computador**. Esse dado pode ter vários tipos, como um valor numérico, sendo positivo ou negativo; um valor booleano, sendo verdadeiro ou falso e caracteres, sendo textos mesmo.

Esses tipos de variáveis mencionadas são encontradas em praticamente todas linguagens de programação.

Mas, para a Unity, nós temos algumas bem específicas além dessas citadas, que nos permitem manipular diversos componentes específicos dentro da engine.

Como por exemplo, a variável do tipo `GameObject`, que nos permite acessar dados de algum objeto que esteja no jogo e a

variável do tipo Transform, que nos permite acessar dados de posições de algum objeto.

Além disso, temos também, por exemplo, a variável do tipo Rigidbody, que nos permite acessar o componente de gravidade de algum objeto e por aí vai.

Ao declararmos uma variável, estamos dizendo que gostaríamos de armazenar o valor dela para usarmos posteriormente.

```
O referências
public class MeuScript : MonoBehaviour
{
    GameObject meuObjeto;
    Transform outroObjeto;
    Rigidbody meuRigidbody;

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

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

Métodos

Aqui já temos uma vaga ideia, não é mesmo? Afinal, vimos anteriormente que o Start e o Update de um script já são métodos.

Um método armazena um conjunto de instruções a serem executadas quando o mesmo é chamado.

Então eu posso ter, por exemplo, um método que faça o meu personagem atacar, um outro que faça ele movimentar, outro que faça ele pegar moedas e outro que o faça pular. Os métodos, além de tudo, nos permitem organizar o código.

```
O referências
void Atacar()
{
  ...
}

O referências
void Movimentar()
{
  ...
}

O referências
void PegarMoedas()
{
  ...
}

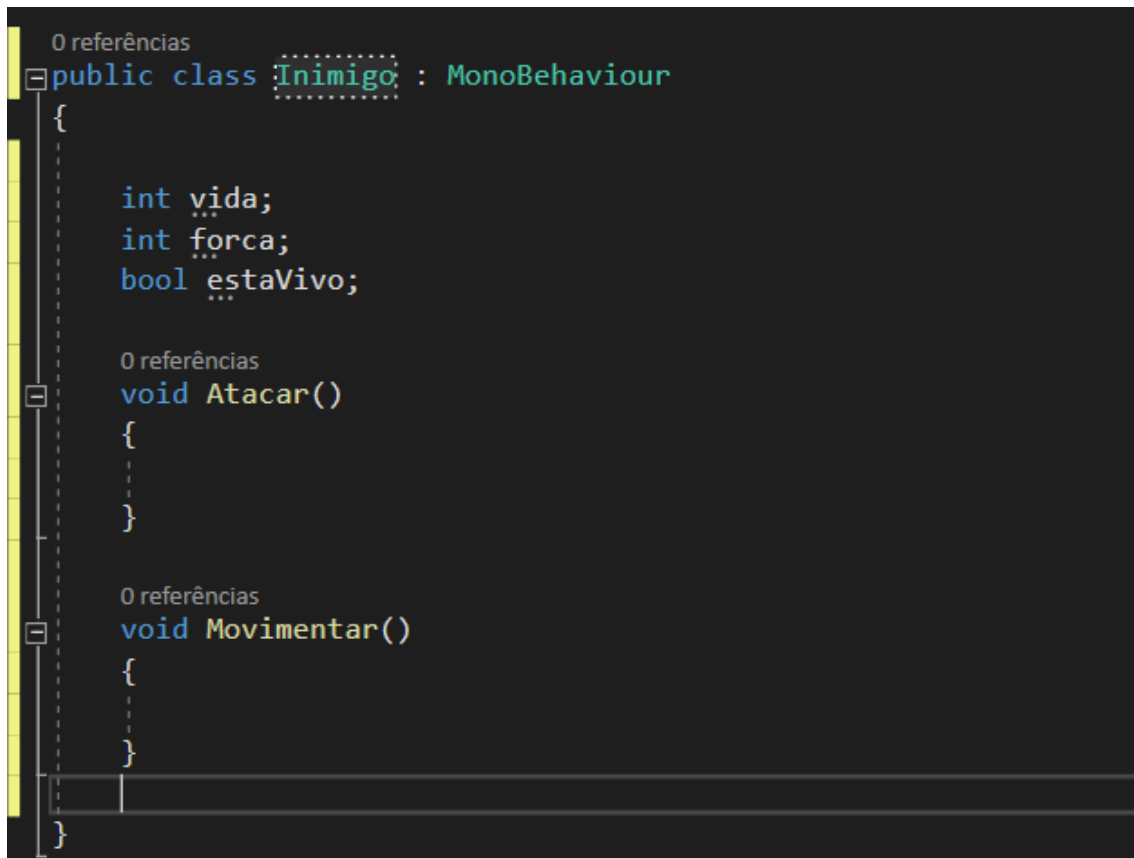
O referências
void Pular()
{
  ...
}
```

Classe

Uma classe nada mais é do que uma forma de agrupar métodos e variáveis juntas! Então dê só uma olhada nesse exemplo. Aqui nessa classe (que é nosso script) temos métodos e temos também variáveis.

As classes representam um tipo de alguma coisa. Por exemplo, se tivermos uma classe chamada Inimigo, sabemos que ela terá

variáveis e métodos correspondentes às ações dos inimigos em nosso jogo.



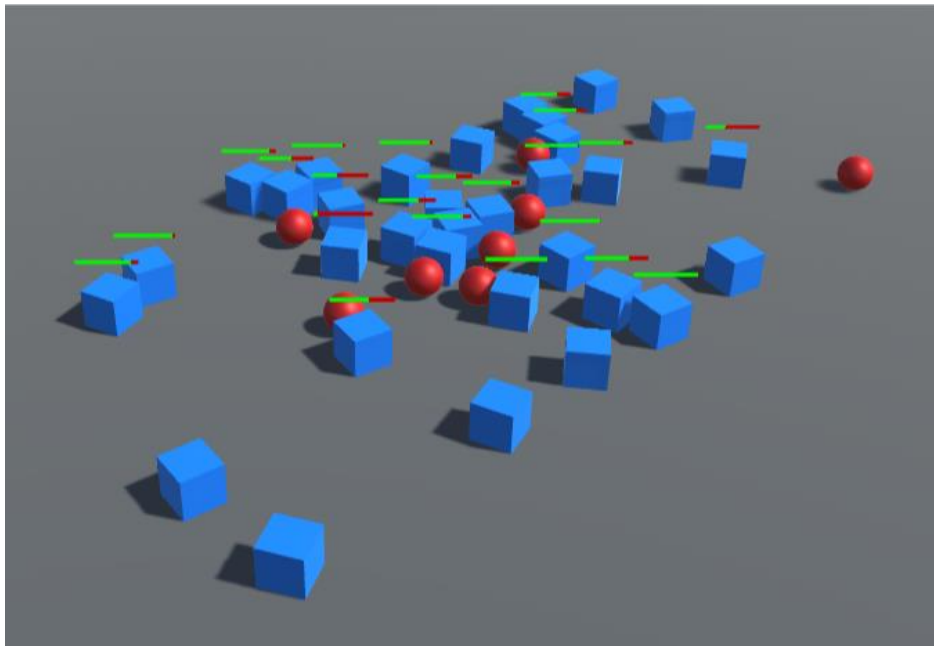
```
0 referências
public class Inimigo : MonoBehaviour
{
    int vida;
    int força;
    bool estaVivo;

    0 referências
    void Atacar()
    {
    }

    0 referências
    void Movimentar()
    {
    }
}
```

Objetos

Logo, se temos uma classe chamada Inimigo, podemos instanciar (ou seja, criar uma cópia na cena do jogo) de vários inimigos idênticos, todos oriundos da mesma classe, mas cada um será um objeto próprio, tendo uma vida própria.



Vários inimigos na cena

No exemplo citado, podemos ter vários personagens na cena, todos cópias da mesma classe mas cada um tendo sua própria vida. Se eu destruir todos os inimigos nessa cena e deixar só dois, os dois continuarão intactos, pois cada um tem sua própria existência, digamos.