04

Organização e otimização

Transcrição

Vamos discutir um pouco sobre o que fizemos nos códigos e passar alguns significados. O último *script* que alteramos foi o de criação de zumbis (`GeradorZumbis.cs`). Nele, temos as variáveis:

- pública `Zumbi` ;
- pública, do tipo `float` , `TempoGerarZumbi` ;
- não pública `contadorTempo` .

Notem que o nome das variáveis **públicas** começa com letras **maiúsculas** e da **privada** com letra **minúscula**. Em todos os *scripts*, seguimos essa lógica, porque na linguagem C# há um padrão de nomenclatura no qual:

- variáveis públicas devem ter nomes com a primeira letra maiúscula;
- variáveis privadas devem ter nomes com a primeira letra minúscula.

Em nomes compostos, chamamos o início de cada palavra com letra maiúscula (`TempoGerarZumbi` , por exemplo) de *Pascal Case*, Caso de Pascal ou *Upper Camel Case* (Código ou Caso do Camelo, em português).

Quando o nome começa com a letra minúscula e as palavras seguintes são iniciadas com letra maiúscula (`contadorTempo` , por exemplo), são chamados de *CamelCase* (Código do Camelo, com letra minúscula) ou *lowerCamelCase*. Além disso, por padrão, o *script* define as variáveis não públicas com um acesso privado (`private`), ou seja, elas não podem ser utilizadas fora do *script* em que foram criadas. Ao contrário das variáveis públicas (`public`), que utilizamos em *scripts* diferentes daqueles em que foram declaradas.

Levando isso em consideração, é uma boa prática definir qual será o acesso — `public` ou `private` — das variáveis ao declará-las. Em alguns *scripts*, como `ControlaCamera.cs` , não especificamos a variável `distCompensar` como `private` . Faremos isso agora, adicionando tipo antes de `Vector3` :

```
private Vector3 distCompensar;
```

Discutiremos outro tópico interessante, observando o *script* `ControlaJogador.cs` . Em `FixedUpdate()` , utilizamos diversas vezes `GetComponent<Rigidbody>` para procurar e acessar componentes. No entanto, há uma forma mais fácil de fazermos isso. Podemos declarar uma variável no início do código, abaixo de `Vivo` . Como ela não será utilizada em outros *scripts*, iremos especificá-la como `private` , do tipo `Rigidbody` e a nomearemos como `rigidbodyJogador` . Notem que, por ser uma variável **privada**, a primeira letra no nome é **minúscula**.

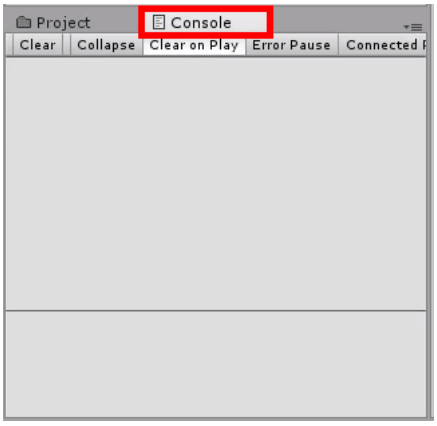
Na sequência, em `Start()` , abaixo de `timeScale` , colocamos a variável `rigidbodyJogador` e atribuímos (`=`) a ela o componente `Rigidbody` .

```
rigidbodyJogador = GetComponent<Rigidbody>();
```

Ou seja, buscamos somente uma vez o componente `Rigidbody` , no início do jogo (`Start`). Isso ajuda na performance, pois não é necessário buscar componentes o tempo todo. Declarada a variável no início do código, podemos utilizá-la para substituir todos `GetComponent<Rigidbody>` que aparecem ao longo dele. Para isso, utilizaremos os atalhos:

- duplo clique em cima da variável (`rigidbodyJogador`) para selecionar;
- "Ctrl + C" para copiar;
- "Ctrl + V" para colar.

Salvaremos e, se voltarmos à Unity, veremos que continua funcionando perfeitamente. Podemos, inclusive, consultar a aba "Console", que estará limpa, sem indicação de erros.



Repetiremos o processo para aplicar o mesmo a `GetComponent<Animator>` . Criaremos a variável privada e do tipo `Animator` (`animatorJogador`), abaixo de `rigidbodyJogador` . Em `Start` , abaixo de `rigidbodyJogador` , atribuiremos a ela o valor de `GetComponent<Animator>` .

```
animatorJogador = GetComponent<Animator>();
```

Assim, quando precisarmos utilizar `Animator` , basta digitarmos `animatorJogador.` . É uma forma de tornar a leitura do código mais fácil, reduzindo o tamanho das linhas. Salvaremos `ControlaJogador.cs` da seguinte forma:

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

public class ControlaJogador : MonoBehaviour
{

    public float Velocidade = 10;
    Vector3 direcao;
    public LayerMask MascaraChao;
    public GameObject TextoGameOver;
    public bool Vivo = true;
    private Rigidbody rigidbodyJogador;

    private void Start()
    {
        Time.timeScale = 1;
        rigidbodyJogador = GetComponent<Rigidbody>();
    }

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

        float eixoX = Input.GetAxis("Horizontal");
```

```
            float eixoZ = Input.GetAxis("Vertical");

            direcao = new Vector3(eixoX, 0, eixoZ);

            if(direcao != Vector3.zero)
            {
                GetComponent<Animator>().SetBool("Movendo", true);
            }
            else
            {
                GetComponent<Animator>().SetBool("Movendo", false);
            }

            if(Vivo == false)
            {
                if(Input.GetButtonDown("Fire1"))
                {
                    SceneManager.LoadScene("game);
                }
            }

        }

        void FixedUpdate()
        {
            rigidbodyJogador.MovePosition
                (rigidbodyJogador.position +
                (direcao * Velocidade * Time.deltaTime));

            Ray raio = Camera.main.ScreenPointToRay(Input.mousePosition);
            Debug.DrawRay(raio.origin, raio.direction * 100, Color.red);

            RaycastHit impacto;

            if(Physics.Raycast(raio, out impacto, 100, MascaraChao))
            {
                Vector3 posicaoMiraJogador = impacto.point - transform.position;

                posicaoMiraJogador.y = transform.position.y;

                Quaternion novaRotacao = Quaternion.LookRotation(posicaoMiraJogador)

                rigidbodyJogador.MoveRotation(novaRotacao);

            }
        }
    }
```

Aplicaremos o que fizemos em `ControlaJogador.cs` aos *scripts* que contêm `GetComponent<>` . No caso, `ControlaInimigo.cs` e `Bala.cs` . Seguindo os mesmos passos de `ControlaJogador.cs` , declararemos as variáveis privadas `rigidbodyInimigo` e `animatorInimigo` no início do código. Na sequência, atribuiremos os respectivos valores a elas em `Start` e substituiremos `GetComponent<>` por elas. `ControlaInimigo.cs` ficará da seguinte forma:

```
public class ControlaInimigo : MonoBehaviour {
```

```
    public GameObject Jogador;
    public float Velocidade = 5;
    private Rigidbody rigidbodyInimigo;
    private Animator animatorInimigo;

    // Use this for initialization
    void Start () {
        Jogador = GameObject.FindWithTag("Jogador");
        int geraTipoZumbi = Random.Rage(1, 28);
        transform.GetChild(geraTipoZumbi).gameObject.SetActive(true);
        rigidbodyInimigo = GetComponent<Rigidbody>();
        animatorInimigo = GetComponent<Animator>();

    }

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

    }

    void FixedUpdate()
    {
        float distancia = Vector3.Distance(transform.position, Jogador.transform.position);

        Vector3 direcao = Jogador.transform.position - transform.position;

        Quaternion novaRotacao = Quaternion.LookRotation(direcao);
        rigidbodyInimigo.MoveRotation(novaRotacao);

        if (distancia > 2.5)
        {
            rigidbodyInimigo.MovePosition
                (rigidbodyInimigo.position +
                direcao.normalized * Velocidade * Time.deltaTime);

            animatorInimigo.SetBool("Atacando", false);
        }
        else
        {
            animatorInimigo.SetBool("Atacando", true);
        }

    }
}
```

Aplicando o processo em `Bala.cs`, teremos que adicionar o método `Start()`, que deletamos anteriormente. O código ficará da seguinte forma:

```
public class Bala : MonoBehaviour {

    public float Velocidade = 20;
    private Rigidbody rigidbodyBala;

    private void Start()
    {
        rigidbodyBala = GetComponent<Rigidbody>();
```

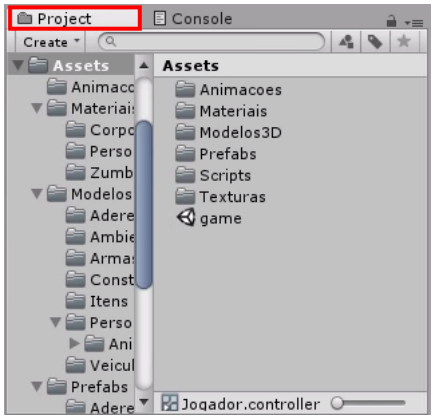
```
    }

    // Update is called once per frame
    void FixedUpdate()
    {
        rigidbodyBala.MovePosition
            (rigidbodyBala.position +
            (direcao * Velocidade * Time.deltaTime));
    }

     void OnTriggerEnter(Collider objetoDeColisao)
    {
        if(objetoDeColisao.tag == "Inimigo")
        {
        Destroy(objetoDeColisao.gameObject);
        }

        Destroy(gameObject);
    }
}
```

A partir de agora, sempre que utilizarmos `Rigidbody` e `Animator`, criaremos a variável para atribuir a eles o valor `GetComponent<Rigidbody>` e acessar o componente de forma mais prática.

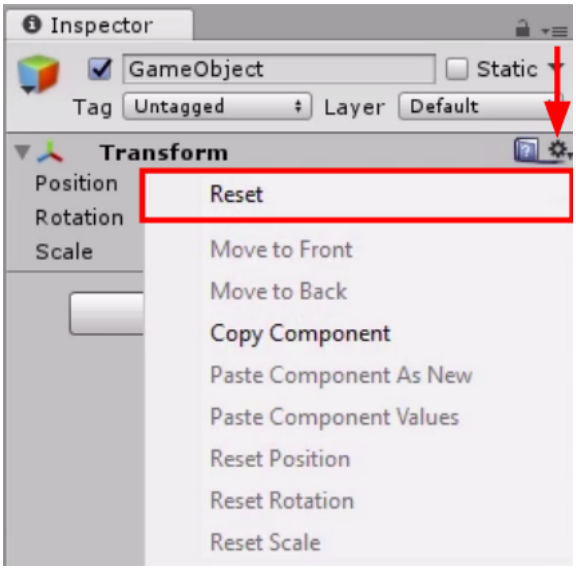
Outro tópico que abordaremos será a **organização** do jogo. Colocaremos os objetos referentes a materiais, na pasta "Project > Assets > **Materiais**". Criaremos uma pasta, clicando com o botão direito do mouse e selecionando "Create > Folder", para armazenar as animações ("Animacoes"):

- "Jogador";
- "TiroMascara";
- "Zumbi".

Dessa forma, "Project" ficará mais organizada.



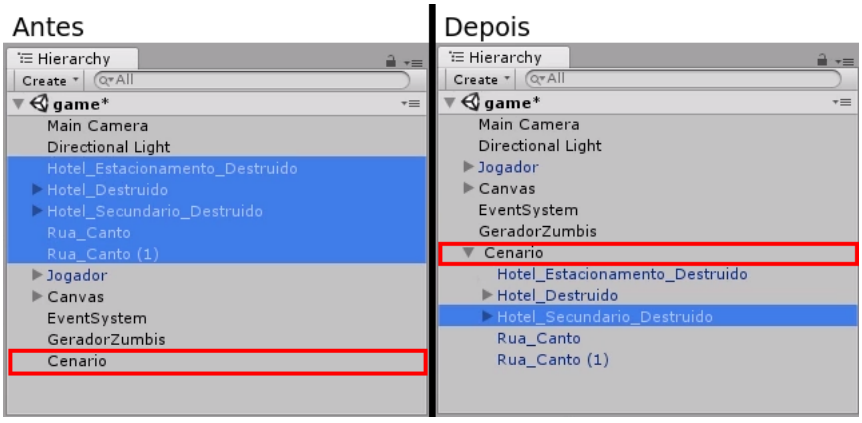
Organizaremos "Hierarchy" também. Começaremos clicando em "Create > Create Empty". No final da lista, aparecerá "GameObject". Iremos selecioná-lo e, em "Inspector", atribuiremos `0` aos três eixos (X, Y e Z) de "Transform > Position". Também podemos zerar os campos de "Transform" clicando na engrenagem do canto superior direito e selecionando a opção "Reset".



Renomearemos "GameObject" como "Cenario" e moveremos para dentro dele todos os objetos que pertecem ao cenário:

- "Hotel_Estacionamento_Destruido";
- "Hotel_Destruido";
- "Hotel_Secundario_Destruido";
- "Rua_Canto;
- "Rua_Canto(1);

Clicando no primeiro e no último, pressionando a tecla "Shift", e arrastando-os.



Os objetos ficam organizados, facilitando a visualização deles.

Assim, vimos formas de **otimizar** os códigos e tornar o projeto mais **organizado**, antes de continuar o desenvolvimento do jogo.