Aula 11 – Scripting com C#: Fundamentos (Parte 1)

Bem-vindo à jornada de transformar ideias em realidade interativa! No universo do desenvolvimento de jogos 3D, a beleza dos gráficos e a complexidade dos modelos são apenas parte da equação. O verdadeiro coração de um jogo, aquilo que o torna vivo e responsivo às ações do jogador, reside na sua programação. É aqui que o scripting entra em cena, e para a vasta maioria dos jogos modernos criados com Unity, isso significa dominar o C#.

Imagine poder ditar exatamente como um personagem se move, como um inimigo reage, ou como um placar é atualizado. Tudo isso é possível através do scripting. Esta aula é o seu primeiro passo fundamental nesse caminho, desvendando os alicerces do C# dentro do ambiente Unity. Não se preocupe se você é novo na programação; vamos construir esse conhecimento juntos, passo a passo, como quem aprende a montar um quebra-cabeça complexo, começando pelas bordas.

Ao final desta aula, você não apenas entenderá a estrutura básica de um script C# no Unity, mas também será capaz de configurar seu ambiente de desenvolvimento, manipular dados simples e fazer seu jogo tomar decisões básicas. Vamos explorar as funções vitais que dão vida aos objetos do seu jogo e introduzir a lógica condicional que permite que seu mundo virtual reaja de forma inteligente. Prepare-se para dar os primeiros comandos ao seu universo digital, conectando a teoria à prática de forma direta e aplicável.

O Coração do Desenvolvimento: Configurando o Visual Studio para Unity

Antes de mergulharmos no código, precisamos garantir que temos as ferramentas certas e que elas estão afiadas. Pense no desenvolvimento de jogos como a construção de uma casa: você pode ter os melhores planos (seu design de jogo), mas sem um bom conjunto de ferramentas e um canteiro de obras organizado, o trabalho será lento e frustrante. No mundo do scripting C# para Unity, o Visual Studio (VS) é essa caixa de ferramentas essencial, o ambiente de desenvolvimento integrado (IDE) que nos permite escrever, depurar e organizar nosso código de forma eficiente.

A integração entre Unity e Visual Studio é um dos pilares da produtividade para desenvolvedores. Ele não é apenas um editor de texto; o VS oferece recursos poderosos como autocompletar código (IntelliSense), depuração passo a passo, realce de sintaxe e refatoração, que transformam a tarefa de escrever código de um desafio árduo em uma experiência fluida e intuitiva. Sem essa integração, você estaria escrevendo código "às cegas", perdendo tempo com erros simples e sem a assistência inteligente que acelera o processo.

Para garantir que seu ambiente esteja pronto para a ação, o primeiro passo é verificar se o Visual Studio está instalado corretamente e configurado para trabalhar com o Unity. Geralmente, ao instalar o Unity Hub, ele oferece a opção de instalar o Visual Studio Community Edition junto com as cargas de trabalho necessárias para o desenvolvimento de jogos. Caso não tenha feito isso ou precise verificar, abra o Visual Studio Installer e certifique-se de que a carga de trabalho "Desenvolvimento de jogos com Unity" esteja selecionada e instalada. Dentro do Unity, vá em Edit > Preferences > External Tools e confirme que o Visual Studio está definido como seu editor de script padrão. Essa pequena verificação garante que, ao dar um duplo clique em um script no Unity, ele se abrirá diretamente no VS, pronto para ser editado.

Desvendando a Estrutura de um Script C# (MonoBehaviour)

Com o Visual Studio configurado, é hora de criar nosso primeiro script e entender sua anatomia. No Unity, um script C# não é apenas um arquivo de texto com código; ele é uma peça fundamental que se conecta diretamente ao motor do jogo, permitindo que você controle o comportamento de objetos na sua cena. Cada script que você cria para interagir com objetos de jogo herda de uma classe especial chamada MonoBehaviour.

Pense no MonoBehaviour como o "passaporte" que seu script precisa para entrar e interagir com o mundo do Unity. Sem ele, seu código C# seria apenas lógica pura, sem saber como se comunicar com um GameObject, acessar seus componentes ou responder aos eventos do jogo. É essa herança que concede ao seu script a capacidade de ser anexado a um objeto na hierarquia do Unity e de acessar todas as funcionalidades inerentes ao motor, como as funções de ciclo de vida que veremos a seguir.

Quando você cria um novo script C# no Unity (clicando com o botão direito na pasta Project > Create > C# Script), ele já vem com uma estrutura básica. Essa estrutura inclui a declaração da classe do seu script, que herda de MonoBehaviour, e alguns blocos de código pré-definidos. O nome da classe deve ser o mesmo do arquivo para evitar problemas. Dentro dessa classe, você encontrará chaves {} que delimitam o corpo do seu script, onde toda a sua lógica e variáveis serão declaradas. É um modelo simples, mas poderoso, que serve como ponto de partida para qualquer comportamento que você queira adicionar ao seu jogo.

```
using UnityEngine; // Importa o namespace Unity para acessar suas classes
using System.Collections; // Importa o namespace para coleções (opcional, mas comum)

public class MeuPrimeiroScript : MonoBehaviour // Declaração da classe, herdando de MonoBehaviour
{
 // Variáveis e outras funções serão declaradas aqui

 void Start() // Função chamada uma vez no início
 {
 // Código de inicialização aqui
 }

 void Update() // Função chamada a cada frame
 {
 // Código de lógica de jogo aqui
 }
}
```

A Vida do Objeto: Funções Essenciais – Awake(), Start() e Update()

Agora que entendemos a estrutura básica de um script, vamos explorar as funções que dão vida a ele. No Unity, os objetos de jogo e seus scripts passam por um ciclo de vida bem definido, e certas funções são chamadas automaticamente em momentos específicos desse ciclo. Conhecer e utilizar Awake(), Start() e Update() é crucial para controlar o fluxo do seu jogo e garantir que as ações aconteçam na ordem e no momento certos.

Imagine que seu script é um ator em uma peça de teatro. Ele precisa saber quando entrar no palco, quando dizer suas primeiras falas e quando interagir com outros personagens. Awake(), Start() e Update() são como as marcações de palco e os sinais do diretor que guiam a performance do seu script. Cada uma dessas funções tem um propósito distinto e é chamada em uma fase específica da vida de um GameObject, permitindo que você inicialize variáveis, configure componentes ou execute lógicas contínuas.

A função Awake() é a primeira a ser chamada quando um script é carregado, mesmo antes de Start(). Ela é executada uma única vez, independentemente de o script estar habilitado ou não. Pense nela como o momento em que o ator se prepara nos bastidores, pegando seus adereços e se concentrando, antes mesmo de o espetáculo começar. É o lugar ideal para inicializar referências a outros componentes do mesmo GameObject ou a outros GameObjects na cena, garantindo que tudo esteja pronto antes que qualquer outra lógica comece.

```
public class ExemploCicloDeVida : MonoBehaviour
{
 void Awake()
 {
 Debug.Log("Awake: Eu sou chamado primeiro, mesmo se o script estiver desabilitado inicialmente!");
 // Ótimo para inicializar referências internas.
 }

 void Start()
 {
 Debug.Log("Start: Eu sou chamado uma vez, no primeiro frame em que o script está habilitado.");
 // Ótimo para inicializar variáveis que dependem de outros scripts já terem seus Awakes chamados.
 }

 void Update()
 {
 // Debug.Log("Update: Eu sou chamado a cada frame!"); // Cuidado para não encher o console!
 // Ótimo para lógica de jogo contínua, como movimento do jogador ou detecção de input.
 }
}
```

A Vida do Objeto: Funções Essenciais – Awake(), Start() e Update() (Continuação)

Após Awake(), a próxima função no ciclo de vida é Start(). Diferente de Awake(), Start() é chamada apenas uma vez, no primeiro frame em que o script está habilitado. Se um script estiver desabilitado no início, Start() só será executada quando ele for habilitado pela primeira vez. Esta função é perfeita para inicializar variáveis que podem depender de outros scripts já terem executado seus Awake() ou para configurar o estado inicial de um objeto após todas as suas dependências estarem prontas.

Finalmente, temos a função Update(), que é o motor contínuo do seu jogo. Update() é chamada a cada frame do jogo, o que significa que ela é executada dezenas ou centenas de vezes por segundo, dependendo da performance do seu jogo e da taxa de quadros. É o local ideal para a lógica de jogo que precisa ser verificada ou executada constantemente, como a detecção de entrada do jogador, o movimento de objetos ou a verificação de colisões. No entanto, é crucial ser eficiente dentro de Update(), pois qualquer código pesado aqui pode impactar negativamente o desempenho do seu jogo.

Além dessas três, existem outras funções de ciclo de vida importantes, como FixedUpdate() (para física, chamada em intervalos fixos) e LateUpdate() (chamada após todos os Update() terem sido executados, útil para câmeras que seguem objetos). Para esta aula, o foco é em Awake(), Start() e Update() como os pilares. Compreender a diferença entre elas é como saber a diferença entre a preparação de um evento (Awake), o início do evento (Start) e o que acontece durante o evento (Update).

Awake()

Uma vez, quando o script é carregado (antes de Start). Inicialização de referências internas, configuração inicial de componentes.

Start()

Uma vez, no primeiro frame em que o script está habilitado. Inicialização de variáveis que dependem de outros scripts, configuração de estado inicial.

Update()

A cada frame do jogo. Lógica de jogo contínua, detecção de input, movimento, verificação de condições.

Os Blocos de Construção: Variáveis e Tipos de Dados (Parte 1)

Compreender como e quando o código é executado é apenas o começo. Para que seu jogo seja interativo e dinâmico, ele precisa armazenar e manipular informações. É aqui que entram as **variáveis** e os **tipos de dados**. Pense nas variáveis como caixas rotuladas onde você pode guardar diferentes tipos de informações. Cada caixa tem um nome (o nome da variável) e só pode guardar um tipo específico de coisa (o tipo de dado).

No contexto de um jogo, essas informações podem ser o placar do jogador, a velocidade de um carro, se uma porta está aberta ou fechada, ou o nome do personagem principal. Sem variáveis, seu jogo seria estático, incapaz de lembrar estados, contar pontos ou reagir a mudanças. Elas são a memória do seu programa, permitindo que ele mantenha o controle de tudo o que está acontecendo no mundo do jogo.

Vamos começar com um dos tipos de dados mais fundamentais: o **int**. O int é usado para armazenar números inteiros, ou seja, números sem casas decimais, como 1, 100, -5, 0. Em um jogo, você usaria um int para coisas como a pontuação do jogador, o número de vidas restantes, a quantidade de balas em um pente ou o nível atual. Declarar uma variável int é simples: você especifica o tipo (int), dá um nome à variável e, opcionalmente, atribui um valor inicial.

```
public class ExemploVariaveisInt : MonoBehaviour
{
    public int pontuacaoJogador = 0; // Variável para armazenar a pontuação
    public int vidasRestantes = 3; // Variável para armazenar o número de vidas
    private int inimigosDerrotados = 0; // Variável privada para contagem interna

    void Start()
    {
        Debug.Log("Pontuação inicial: " + pontuacaoJogador);
        Debug.Log("Vidas iniciais: " + vidasRestantes);

        pontuacaoJogador = 100; // Atualiza a pontuação
        vidasRestantes = vidasRestantes - 1; // Diminui uma vida

        Debug.Log("Nova pontuação: " + pontuacaoJogador);
        Debug.Log("Novas vidas: " + vidasRestantes);
    }
}
```

No exemplo acima, `public` significa que a variável pode ser vista e modificada no Inspector do Unity, o que é muito útil para testar e ajustar valores sem mexer no código.

Os Blocos de Construção: Variáveis e Tipos de Dados (Parte 2)

Enquanto os ints são perfeitos para números inteiros, o mundo dos jogos raramente é tão exato. Posições de objetos, velocidades, tempos e muitas outras grandezas frequentemente exigem valores com casas decimais. Para isso, usamos o tipo de dado **float**. O float (de "floating-point number") permite armazenar números reais, como 3.14, 0.5, -10.25.

Imagine que você está controlando um carro em um jogo. A velocidade do carro não será sempre um número inteiro; ela pode ser 5.7 metros por segundo, ou a distância percorrida pode ser 123.45 unidades. Usar um float nesses casos é essencial para a precisão e a fluidez do movimento e dos cálculos. No C#, quando você atribui um valor decimal a um float, é uma boa prática adicionar um f ou F ao final do número (ex: 5.5f) para indicar explicitamente que é um float e não um double (outro tipo de número decimal, com maior precisão, mas que consome mais memória e não é o padrão para a maioria das operações no Unity).

A principal diferença entre int e float é a capacidade de armazenar valores fracionários. Se você precisa contar itens discretos (vidas, moedas), use int. Se precisa de medidas contínuas (posição, velocidade, tempo), float é a escolha certa. Misturar os dois sem cuidado pode levar a perda de precisão ou erros de compilação, então é importante escolher o tipo de dado correto para cada situação.

```
public class ExemploVariaveisFloat : MonoBehaviour
{
 public float velocidadeJogador = 5.0f; // Velocidade em unidades por segundo
 public float tempoRestante = 60.5f; // Tempo restante em segundos
 private float gravidade = 9.81f; // Valor da gravidade

 void Start()
 {
 Debug.Log("Velocidade inicial: " + velocidadeJogador);
 Debug.Log("Tempo restante: " + tempoRestante);

 velocidadeJogador = 7.25f; // Aumenta a velocidade
 tempoRestante -= 0.5f; // Diminui meio segundo

 Debug.Log("Nova velocidade: " + velocidadeJogador);
 Debug.Log("Novo tempo restante: " + tempoRestante);
 }
}
```

Os Blocos de Construção: Variáveis e Tipos de Dados (Parte 3)

Além de números inteiros e decimais, muitas vezes precisamos armazenar estados que são binários: verdadeiro ou falso, ligado ou desligado, sim ou não. Para isso, o C# nos oferece o tipo de dado **bool** (abreviação de "boolean"). Uma variável bool só pode ter dois valores possíveis: `true` (verdadeiro) ou `false` (falso).

Pense em um jogo onde você precisa saber se o jogador está pulando, se uma porta está aberta, se o jogo acabou, ou se um inimigo está vivo. Todas essas são perguntas que podem ser respondidas com um simples "sim" ou "não", "verdadeiro" ou "falso". O tipo bool é incrivelmente poderoso para controlar o fluxo do seu jogo, permitindo que você tome decisões baseadas no estado atual de diferentes elementos.

Declarar uma variável bool é tão direto quanto declarar um int ou um float. Você especifica o tipo bool, dá um nome à variável e atribui `true` ou `false` como seu valor inicial. Essas variáveis são a espinha dorsal da lógica condicional, que exploraremos em breve, pois elas são a base para qualquer "se isso, então aquilo" no seu código.

```
public class ExemploVariaveisBool : MonoBehaviour
{
 public bool jogoAcabou = false; // Indica se o jogo terminou
 public bool jogadorEstaPulando = false; // Indica se o jogador está no ar
 private bool portaAberta = false; // Estado de uma porta

 void Start()
 {
 Debug.Log("Jogo acabou? " + jogoAcabou);
 Debug.Log("Jogador está pulando? " + jogadorEstaPulando);

 // Simula o fim do jogo
 jogoAcabou = true;
 Debug.Log("Jogo acabou? " + jogoAcabou);

 // Simula o jogador pulando
 jogadorEstaPulando = true;
 Debug.Log("Jogador está pulando? " + jogadorEstaPulando);
 }
}
```

Os Blocos de Construção: Variáveis e Tipos de Dados (Parte 4)

Finalmente, para interagir com o jogador, exibir mensagens, nomes de personagens ou qualquer tipo de texto, precisamos do tipo de dado **string**. Uma string é uma sequência de caracteres, como letras, números e símbolos, que é tratada como um único valor de texto. No C#, as strings são sempre delimitadas por aspas duplas (").

Imagine a interface do usuário do seu jogo: o nome do jogador, a mensagem "Game Over", o diálogo de um personagem, a descrição de um item. Tudo isso é manipulado como string. Sem esse tipo de dado, a comunicação textual no seu jogo seria impossível, tornando a experiência do jogador muito limitada. As strings são flexíveis e podem ser concatenadas (juntadas), formatadas e manipuladas de diversas maneiras para criar mensagens dinâmicas e informativas.

Declarar uma string é similar aos outros tipos: você usa a palavra-chave string, dá um nome à variável e atribui um valor entre aspas duplas. É importante lembrar que, embora uma string possa conter números, eles serão tratados como texto e não poderão ser usados em operações matemáticas diretamente, a menos que sejam convertidos para um tipo numérico.

```
public class ExemploVariaveisString : MonoBehaviour
{
    public string nomeJogador = "Herói Anônimo"; // Nome do personagem
    public string mensagemBoasVindas = "Bem-vindo ao jogo!"; // Mensagem inicial
    private string statusJogo = "Em Andamento"; // Estado textual do jogo

    void Start()
    {
        Debug.Log(mensagemBoasVindas + " " + nomeJogador + "!"); // Concatenação de strings
        Debug.Log("Status atual: " + statusJogo);

        nomeJogador = "Cavaleiro Valente"; // Altera o nome do jogador
        statusJogo = "Missão Iniciada"; // Altera o status

        Debug.Log("Novo nome: " + nomeJogador);
        Debug.Log("Novo status: " + statusJogo);
    }
}
```

int

Números inteiros (sem casas decimais). Pontuação, número de vidas, quantidade de itens.

float

Números decimais (com casas decimais). Velocidade, posição, tempo, dano de ataque.

bool

Valores lógicos: true ou false. Jogo acabou, jogador pulando, porta aberta.

string

Sequência de caracteres (texto). Nome do jogador, mensagens de diálogo, descrições.

Tomando Decisões: Introdução à Lógica Condicional com if e else (Parte 1)

Agora que sabemos como armazenar diferentes tipos de informações, o próximo passo é fazer com que nosso jogo reaja a essas informações. Um jogo não é apenas uma sequência linear de eventos; ele é um sistema dinâmico que toma decisões com base no que está acontecendo. É aqui que entra a **lógica condicional**, a capacidade de executar diferentes blocos de código dependendo se uma condição é verdadeira ou falsa.

Pense na lógica condicional como um sistema de semáforos no trânsito. Se o semáforo está verde, você avança; se está vermelho, você para. Essa é a essência do comando `if`: "SE uma condição for verdadeira, ENTÃO faça algo". Sem essa capacidade de tomar decisões, seu jogo seria previsível e sem vida, incapaz de responder às ações do jogador ou às mudanças no ambiente.

No C#, a estrutura básica de uma declaração if é simples. Você escreve a palavra-chave `if`, seguida por uma condição entre parênteses (). Se essa condição for avaliada como `true`, o bloco de código dentro das chaves {} que segue o if será executado. Se a condição for `false`, esse bloco de código será ignorado. É o mecanismo fundamental para criar interatividade e inteligência no seu jogo, permitindo que ele se adapte a diferentes cenários.

```
public class ExemploIf : MonoBehaviour
{
    public int pontuacao = 80;

    void Start()
    {
        Debug.Log("Pontuação atual: " + pontuacao);

        // Se a pontuação for maior ou igual a 100, o jogador subiu de nível.
        if (pontuacao >= 100)
        {
            Debug.Log("Parabéns! Você subiu de nível!");
        }

        Debug.Log("Fim da verificação de nível.");
    }
}
```

Neste exemplo, se pontuacao fosse 120, a mensagem de "Parabéns!" apareceria. Se fosse 80, como no código, ela seria ignorada.

Tomando Decisões: Introdução à Lógica Condicional com if e else (Parte 2)

A vida raramente é uma questão de "faça isso se for verdade e nada se for falso". Muitas vezes, precisamos de uma alternativa: "SE uma condição for verdadeira, FAÇA ISSO; CASO CONTRÁRIO, FAÇA AQUILO". É para esses cenários que o comando `else` entra em jogo, complementando o if e criando um caminho alternativo para quando a condição principal não é atendida.

Imagine um sistema de portas em um jogo. SE o jogador tiver a chave, a porta abre; CASO CONTRÁRIO, a porta permanece trancada e uma mensagem de erro aparece. O else nos permite definir esse comportamento padrão ou alternativo, garantindo que o jogo sempre tenha uma resposta para qualquer situação, tornando a experiência mais completa e robusta.

O `else` é sempre usado em conjunto com um if. Ele não tem uma condição própria; ele simplesmente executa seu bloco de código se a condição do if associado for `false`. Essa combinação if-else é a base para a maioria das decisões binárias em qualquer programa, permitindo que você crie bifurcações no fluxo de execução do seu script, direcionando o jogo para diferentes caminhos com base nos estados atuais.

```
public class ExemploIfElse : MonoBehaviour
{
public int vidaJogador = 50;

void Start()
{
Debug.Log("Vida atual do jogador: " + vidaJogador);

// Se a vida for menor ou igual a 0, o jogo acaba. Caso contrário, o jogo continua.
if (vidaJogador <= 0)
{
Debug.Log("Game Over! Suas vidas acabaram.");
// Aqui você chamaria uma função para reiniciar o jogo ou mostrar a tela de Game Over.
}
else
{
Debug.Log("Você ainda tem vida! Continue jogando.");
// Aqui o jogo continuaria normalmente.
}

Debug.Log("Fim da verificação de vida.");
}
}
```

Tomando Decisões: Lógica Condicional Aninhada e else if

A vida, e os jogos, raramente se resumem a apenas duas opções. Muitas vezes, precisamos lidar com múltiplos cenários, onde uma condição leva a outra, ou onde há várias possibilidades mutuamente exclusivas. Para esses casos, o C# oferece a estrutura `else if` e a capacidade de aninhar declarações if.

Imagine um sistema de classificação de jogadores baseado em pontuação: se a pontuação é menor que 50, é "Bronze"; se é entre 50 e 99, é "Prata"; e se é 100 ou mais, é "Ouro". Usar uma série de ifs independentes não funcionaria bem, pois um jogador com 120 pontos seria "Bronze", "Prata" e "Ouro" ao mesmo tempo, o que não faz sentido. O else if resolve isso, permitindo que você teste múltiplas condições em sequência, onde apenas uma delas será executada.

A estrutura `else if` é como uma série de perguntas encadeadas: "SE é isso? Não? ENTÃO SE é aquilo? Não? ENTÃO SE é outra coisa?". O primeiro if ou else if cuja condição for verdadeira terá seu bloco de código executado, e todas as outras condições subsequentes na mesma cadeia if-else if-else serão ignoradas. Isso garante que apenas um caminho seja seguido, tornando sua lógica clara e eficiente para lidar com múltiplos estados ou classificações.

```
public class ExemploElseIf : MonoBehaviour
{
public int pontuacaoJogador = 75;

void Start()
{
Debug.Log("Pontuação do jogador: " + pontuacaoJogador);

if (pontuacaoJogador < 50)
{
Debug.Log("Seu rank é: Bronze");
}
else if (pontuacaoJogador < 100) // Esta condição só é testada se a anterior for falsa
{
Debug.Log("Seu rank é: Prata");
}
else // Este bloco só é executado se todas as condições anteriores forem falsas
{
Debug.Log("Seu rank é: Ouro");
}

Debug.Log("Verificação de rank concluída.");
}
}
```

Operadores de Comparação e Lógicos

Para que as condições dentro dos nossos ifs, else ifs e elses funcionem, precisamos de ferramentas para comparar valores e combinar múltiplas condições. Essas ferramentas são os **operadores de comparação** e os **operadores lógicos**. Eles são a linguagem que usamos para fazer perguntas ao nosso programa, como "Este valor é maior que aquele?" ou "Ambas as coisas são verdadeiras ao mesmo tempo?".

Os operadores de comparação são usados para comparar dois valores e sempre retornam um resultado bool (true ou false). Eles são a base de qualquer condição. Por exemplo, para saber se a pontuação do jogador é igual a 100, usamos `==`. Para saber se a vida é menor que zero, usamos `<=`. Sem esses operadores, não poderíamos formular as perguntas que o if precisa para tomar suas decisões.

Já os operadores lógicos nos permitem combinar múltiplas condições bool em uma única condição mais complexa. O operador `&&` (AND lógico) significa que *ambas* as condições devem ser verdadeiras para que o resultado final seja verdadeiro. O operador `||` (OR lógico) significa que *pelo menos uma* das condições deve ser verdadeira. E o operador `!` (NOT lógico) inverte o valor de uma condição, transformando true em false e vice-versa. Dominar esses operadores é como aprender a construir frases complexas em uma nova língua, permitindo que você expresse lógicas de jogo muito mais sofisticadas.

```
public class ExemploOperadores : MonoBehaviour
{
    public int moedas = 10;
    public int chaves = 1;
    public bool portaTrancada = true;

    void Start()
    {
        // Operadores de Comparação
        if (moedas >= 10) // Maior ou igual
        {
            Debug.Log("Você tem moedas suficientes!");
        }

        if (chaves == 1) // Igual a
        {
            Debug.Log("Você tem uma chave!");
        }

        // Operadores Lógicos
        if (moedas >= 10 && chaves >= 1) // Ambas as condições devem ser verdadeiras
        {
            Debug.Log("Você tem moedas E chaves suficientes!");
        }

        if (moedas < 5 || chaves == 0) // Pelo menos uma condição deve ser verdadeira
        {
            Debug.Log("Você precisa de mais moedas OU chaves!");
        }

        if (!portaTrancada) // Se a porta NÃO estiver trancada
        {
            Debug.Log("A porta está aberta!");
        }
    }
}
```

== (Igual a)

`if (vida == 0)`

!= (Diferente de)

`if (nomeJogador != "Convidado")`

< (Menor que)

`if (tempoRestante < 10.0f)`

> (Maior que)

`if (pontuacao > 1000)`

&& (E lógico)

`if (estaNoChao && estaPulando)`

|| (Ou lógico)

`if (temChave || temSenha)`

! (Não lógico)

`if (!inimigoVisivel)`

Boas Práticas em Scripting C# para Unity

Escrever código que funciona é um bom começo, mas escrever código que é fácil de ler, entender e manter é o que realmente diferencia um desenvolvedor profissional. No desenvolvimento de jogos, especialmente em equipes, a clareza e a organização do seu código são tão importantes quanto a sua funcionalidade. Adotar boas práticas desde o início economizará horas de depuração e frustração no futuro.

Imagine que seu código é um manual de instruções. Se o manual estiver mal escrito, confuso e desorganizado, ninguém conseguirá usá-lo, nem mesmo você depois de um tempo. Boas práticas de scripting são como as regras de um bom manual: elas garantem que seu código seja legível, consistente e fácil de colaborar. Isso inclui desde a forma como você nomeia suas variáveis e funções até como você estrutura seus arquivos e adiciona comentários.

Algumas das práticas mais importantes incluem:

1. **Nomenclatura Clara e Consistente:** Use nomes descritivos para variáveis, funções e classes (ex: velocidadeJogador em vez de v). Siga convenções como camelCase para variáveis e PascalCase para classes e métodos.
2. **Comentários Úteis:** Explique o "porquê" de um trecho de código complexo, não apenas o "o quê". Comentários são para clarear intenções, não para reescrever o código.
3. **Evitar "Magic Numbers":** Em vez de usar números literais diretamente no código (ex: if (vida <= 0)), declare-os como constantes ou variáveis com nomes significativos (ex: const int VIDA_MINIMA = 0;). Isso torna o código mais fácil de entender e modificar.
4. **Modularidade:** Divida seu código em funções e classes menores e mais focadas. Cada função deve fazer uma única coisa bem feita. Isso facilita a depuração e a reutilização.
5. **Organização de Pastas:** Mantenha seus scripts organizados em pastas lógicas dentro do projeto Unity (ex: Scripts/Player, Scripts/Enemy).

Adotar essas práticas desde o início é um investimento no seu futuro como desenvolvedor. Um código limpo e bem estruturado é mais fácil de depurar, de estender e de trabalhar em equipe, permitindo que você se concentre na criatividade e na resolução de problemas complexos, em vez de lutar contra a confusão.

CONSOLIDAÇÃO

Chegamos ao fim da primeira parte da nossa imersão em Scripting com C# para Unity. Nesta aula, você deu os primeiros passos cruciais para dar vida aos seus jogos. Começamos configurando o ambiente de desenvolvimento com o Visual Studio, garantindo que você tenha as ferramentas certas para codificar com eficiência. Em seguida, desvendamos a estrutura fundamental de um script C# no Unity, entendendo o papel do MonoBehaviour como a ponte entre seu código e o motor do jogo.

Exploramos as funções essenciais do ciclo de vida de um script – Awake(), Start() e Update() – aprendendo quando e por que cada uma é chamada, e como usá-las para controlar o fluxo do seu jogo. Mergulhamos nos blocos de construção básicos da programação: as variáveis e os tipos de dados (int, float, bool, string), que permitem ao seu jogo armazenar e manipular informações dinamicamente. Finalmente, introduzimos a lógica condicional com if e else, e os operadores de comparação e lógicos, capacitando seu jogo a tomar decisões e reagir a diferentes cenários.

Em prática:

1. Certifique-se de que seu Visual Studio está configurado para Unity.
2. Crie um novo script e observe sua estrutura básica.
3. Experimente usar Debug.Log em Awake(), Start() e Update() para ver a ordem de execução.
4. Declare variáveis de diferentes tipos e altere seus valores.
5. Crie um script simples que use if-else para exibir uma mensagem baseada em uma condição (ex: vida do jogador).

Próxima Aula: Na Aula 12 – Scripting com C#: Fundamentos (Parte 2), aprofundaremos ainda mais, explorando conceitos como loops, arrays, funções personalizadas e a introdução a classes e objetos, expandindo ainda mais seu arsenal de desenvolvimento.

Recursos Adicionais:

- **Documentação Oficial do Unity:** Para detalhes aprofundados sobre qualquer função ou componente.
- **C# Yellow Book:** Um guia gratuito e abrangente para a linguagem C#.
- **Tutoriais em Vídeo (Brackeys, CodeMonkey):** Ótimos para ver exemplos práticos e visuais.

Autoavaliação

1. Qual das seguintes funções é chamada uma única vez, no primeiro frame em que o script está habilitado, e é ideal para inicializações que dependem de outros scripts já terem executado suas preparações iniciais?

A) Awake()
B) Update()
C) Start()
D) FixedUpdate()

2. Para armazenar a pontuação de um jogador, que é sempre um número inteiro, qual tipo de dado C# seria o mais apropriado?

A) float
B) string
C) bool
D) int

3. Considere o seguinte trecho de código:

```
int valorA = 10;
int valorB = 5;
bool condicao = (valorA > 5 && valorB < 10);
```

Qual será o valor final da variável condicao?
A) true
B) false
C) 1
D) 0

4. Qual é o propósito principal da estrutura else if em uma lógica condicional?

A) Executar um código se a condição principal for falsa, sem testar outras.
B) Permitir testar múltiplas condições em sequência, onde apenas uma será executada.
C) Inverter o valor de uma condição booleana.
D) Repetir um bloco de código várias vezes.

5. Explique a importância de utilizar o MonoBehaviour como classe base para scripts no Unity e como isso difere de um script C# "puro".

Gabarito:

1. C
2. D
3. A
4. B

> **NOTA IMPORTANTE:** As informações regulatórias/legais/técnicas desta aula estão atualizadas até 2025. Consulte sempre fontes oficiais para verificar alterações.