Unity C# Coding Instructional Packet

A Beginner's Guide to Programming in Unity 6

Created by Foxnova Studios

Table of Contents

1. Introduction to C# and Unity

Welcome to the world of game development! This guide will teach you the fundamentals of C# programming specifically tailored for Unity 6, the latest version of Unity's game engine.

C# (pronounced 'C Sharp') is a powerful, modern programming language developed by Microsoft. It's the primary language used in Unity for scripting game behavior, creating interactive experiences, and bringing your creative visions to life.

What You'll Learn

This packet covers everything from basic C# syntax to Unity-specific concepts like MonoBehaviour, GameObjects, components, and the Unity lifecycle. By the end, you'll be able to write scripts that control game objects, respond to player input, and create interactive gameplay.

> **Note:** This guide assumes no prior programming experience. If you're already familiar with programming, feel free to skip ahead to the Unity-specific sections.

2. Setting Up Your Environment

Before writing code, you need to set up Unity 6 and ensure your development environment is ready.

Installing Unity 6

1. Download Unity Hub from unity.com
2. Install Unity 6 through Unity Hub
3. Choose Visual Studio (recommended) or Visual Studio Code as your code editor
4. Create a new 3D or 2D project to begin

Creating Your First Script

1. In Unity, right-click in the Project window
2. Select Create > C# Script
3. Name it 'PlayerController' (no spaces)
4. Double-click the script to open it in your code editor

> **Tip:** Always use PascalCase for script names (capitalize the first letter of each word). The script name must exactly match the class name inside.

3. Using Visual Studio with Unity

Visual Studio is a powerful integrated development environment (IDE) that makes writing and debugging C# code easier. Unity integrates seamlessly with Visual Studio, providing features like code completion, syntax highlighting, and debugging tools. Mastering Visual Studio will significantly improve your productivity and code quality.

Setting Up Visual Studio

1. When you install Unity through Unity Hub, select Visual Studio as your code editor
2. Visual Studio will be installed automatically with Unity support and C# tools
3. In Unity, go to Edit > Preferences > External Tools
4. Set External Script Editor to Visual Studio
5. Check 'Regenerate project files' if Visual Studio doesn't recognize Unity classes

Opening Scripts from Unity

Simply double-click any C# script in Unity's Project window, and it will open in Visual Studio. Unity and Visual Studio communicate automatically through project files (.csproj and .sln files), so changes you make in Visual Studio are reflected in Unity when you save the file (Ctrl+S).

Understanding the Visual Studio Interface

When you first open Visual Studio, you'll see several panels:

• **Solution Explorer (right side):** Shows all your project files and scripts. This is like Unity's Project window. You can expand folders to find scripts.

• **Code Editor (center):** The main area where you write your code. Each open script appears as a tab at the top.

• **Error List (bottom):** Displays all compilation errors, warnings, and messages. Double-click an error to jump to that line of code.

• **Output Window (bottom):** Shows build information and Unity console messages when debugging.

Code Indentation and Formatting

Proper indentation makes your code readable and professional. Visual Studio can automatically format your code for you!

Automatic Indentation: Visual Studio automatically indents your code as you type. When you open a curly brace { and press Enter, the next line is automatically indented. When you close the brace }, it automatically moves back one level.

Manual Indentation:
• Press **Tab** to indent a line or selected lines one level to the right

• Press **Shift+Tab** to unindent (move left) a line or selected lines
• Select multiple lines and press Tab to indent them all at once

Auto-Format Entire File: This is the most important shortcut! Press **Ctrl+K, Ctrl+D** (hold Ctrl, press K, then press D) to automatically format your entire file. This fixes all indentation, spacing, and alignment issues instantly. Use this before saving your work!

Format Selection: Select code and press **Ctrl+K, Ctrl+F** to format only the selected lines.

Code Organization Best Practices

```
Visual Studio helps you organize your code with regions and proper spacing:

#region Variables
public float speed = 5.0f;
private int health = 100;
#endregion

#region Unity Lifecycle
void Start() { }
void Update() { }
#endregion
```

Regions can be collapsed (folded) by clicking the minus sign next to #region. This helps you focus on specific parts of your code without scrolling through everything.

Essential Visual Studio Features

IntelliSense (Auto-Complete): As you type, Visual Studio suggests code completions. This feature is incredibly powerful:
• Type 'trans' and IntelliSense suggests 'transform'
• Type 'transform.' and it shows all available properties and methods
• Press **Tab** or **Enter** to accept a suggestion
• Use **Arrow keys** to navigate through suggestions
• Press **Ctrl+Space** to manually trigger IntelliSense if it doesn't appear

Parameter Info: When you type a function name and open parenthesis, Visual Studio shows what parameters the function needs. Press **Ctrl+Shift+Space** to see parameter info.

Quick Info (Tooltips): Hover your mouse over any variable, function, or class name to see its documentation. This is extremely helpful for learning Unity's API!

Error Detection: Visual Studio underlines errors and warnings as you type:
• **Red squiggly lines:** Errors that prevent compilation. You must fix these!
• **Green squiggly lines:** Warnings. Your code will run but might have issues
• **Gray squiggly lines:** Suggestions for improvement
Hover over any squiggly line to see what's wrong and how to fix it.

Go to Definition: Right-click on any function, variable, or class and select 'Go to Definition' (or press **F12**) to jump to where it's defined. This lets you explore Unity's source code and learn how things work! Press **Alt+Left**

Arrow to go back to where you were.

Find All References: Right-click on a variable or function and select 'Find All References' (or press **Shift+F12**) to see everywhere it's used in your project. This is great for understanding how your code connects together.

Rename (Refactoring): Right-click on a variable or function and select 'Rename' (or press **F2**). Type the new name and press Enter. Visual Studio automatically renames it everywhere in your project! This is much safer than manual find-and-replace.

Quick Actions (Light Bulb): When you see a light bulb icon (or when there's a squiggly line), click it or press **Ctrl+.** (period) to see suggested fixes and improvements. Visual Studio can often fix errors automatically!

Code Snippets - Writing Code Faster

Code snippets are templates that insert common code patterns. Type these shortcuts and press Tab twice:

Shortcut	What It Creates
prop + Tab Tab	Property with get/set
for + Tab Tab	For loop structure
foreach + Tab Tab	Foreach loop structure
if + Tab Tab	If statement structure
switch + Tab Tab	Switch statement structure
try + Tab Tab	Try-catch block
class + Tab Tab	New class definition

Keyboard Shortcuts - Master These!

Learning keyboard shortcuts will make you 5-10 times faster at coding. Start with these essentials:

Shortcut	Action	Why It's Important
Ctrl + S	Save current file	Save your work constantly!
Ctrl + K, Ctrl + D	Format entire document	Makes code professional & readable
Ctrl + K, Ctrl + F	Format selection	Clean up messy code sections
Ctrl + /	Comment/uncomment lines	Quickly disable code for testing
Ctrl + Space	Trigger IntelliSense	Get code suggestions
Ctrl + .	Quick actions (fixes)	Automatically fix errors
F12	Go to definition	Learn how functions work
Alt + Left/Right	Navigate backward/forward	Jump between code locations
Ctrl + F	Find in current file	Search for text
Ctrl + H	Find and replace	Change text throughout file
Ctrl + Shift + F	Find in all files	Search entire project
F2	Rename symbol	Safely rename variables
Shift + F12	Find all references	See where code is used
Ctrl + K, Ctrl + C	Comment selection	Comment multiple lines
Ctrl + K, Ctrl + U	Uncomment selection	Uncomment multiple lines
F5	Start debugging	Test code with breakpoints
Ctrl + Tab	Switch between open files	Navigate quickly
Ctrl + -	Navigate backward	Undo navigation jumps
Ctrl + Shift + -	Navigate forward	Redo navigation jumps

Practice Tip: Print out this shortcut list and keep it next to your computer. Try to use one new shortcut each day until they become second nature. Ctrl+K, Ctrl+D should become automatic!

Debugging in Visual Studio

Debugging is the process of finding and fixing errors in your code. Visual Studio's debugger is one of its most powerful features.

Setting Breakpoints: A breakpoint pauses your game at a specific line of code so you can inspect what's happening. To set a breakpoint:
1. Click in the gray margin to the left of a line number (a red dot appears)
2. Or place your cursor on a line and press **F9**
3. Click the red dot again (or press F9) to remove the breakpoint

Attaching to Unity: To use breakpoints, you must attach Visual Studio to Unity:
1. In Visual Studio, click 'Debug > Attach Unity Debugger' (or press **Alt+F5**)
2. Select your Unity Editor from the list and click OK
3. Or simply press **F5** which will attach automatically
4. Now run your game in Unity - it will pause at any breakpoints

When Paused at a Breakpoint:
• Hover over any variable to see its current value
• Check the 'Locals' window (Debug > Windows > Locals) to see all variables in scope
• Use the 'Watch' window to monitor specific variables
• Press **F10** to execute the current line and move to the next (Step Over)
• Press **F11** to step into a function call and see what happens inside
• Press **Shift+F11** to step out of the current function
• Press **F5** to continue running until the next breakpoint

Conditional Breakpoints: Right-click a breakpoint and select 'Conditions' to make it only pause when certain conditions are met (like when health reaches 0). This is incredibly useful for debugging specific scenarios!

Organizing Your Workspace

Visual Studio has many windows that can be arranged however you like. Here's a recommended setup for Unity development:

• Keep **Solution Explorer** docked on the right side for easy file navigation
• Keep **Error List** docked at the bottom - you'll check this constantly
• Open **Output** window (View > Output) for Unity console messages
• Close windows you rarely use to maximize code editor space
• You can drag any window's tab to dock it elsewhere or make it float
• Save your layout: Window > Save Window Layout

Working with Multiple Files

Real projects have many scripts. Here's how to work with multiple files efficiently:

• Press **Ctrl+Tab** to see all open files and quickly switch between them
• Press **Ctrl+,** (comma) to open 'Go to All' and search for any file, class, or function

• Right-click a tab and select 'New Vertical Tab Group' to view two files side by side
• Click the 'X' on a tab to close it, or press **Ctrl+F4**
• Right-click a tab and select 'Close All But This' to close other tabs

> **Pro Tip:** Use Ctrl+, (comma) constantly! Type any class, function, or file name to jump to it instantly. This is faster than navigating through Solution Explorer. For example, type 'PlayerController' and press Enter to open that script immediately.

4. C# Basics

Let's understand the structure of a basic C# script in Unity.

Anatomy of a Unity Script

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

public class PlayerController : MonoBehaviour
{
    // Your code goes here
}
```

Breaking It Down

using statements: These import namespaces that contain useful classes and functions. UnityEngine contains all Unity-specific code.

public class: This declares your script as a class. 'public' means it can be accessed by other scripts.

: MonoBehaviour: This means your class inherits from MonoBehaviour, which is the base class for all Unity scripts attached to GameObjects.

Curly braces {}: These define the beginning and end of code blocks.

// Comments: Lines starting with // are comments. They're notes for you and other developers and are ignored by the compiler.

5. Variables and Data Types

Variables are containers that store data. In C#, every variable has a specific data type that determines what kind of data it can hold.

Common Data Types

Data Type	Description	Example
`int`	Whole numbers	`int score = 100;`
`float`	Decimal numbers	`float speed = 5.5f;`
`bool`	True or false	`bool isAlive = true;`
`string`	Text	`string name = "Player";`
`Vector3`	3D position/direction	`Vector3 pos = new Vector3(0, 0, 0);`
`GameObject`	Reference to game object	`GameObject player;`

Declaring Variables

```
public class Example : MonoBehaviour
{
    // Public variables appear in the Inspector
    public int health = 100;
    public float moveSpeed = 5.0f;

    // Private variables are hidden
    private bool isGrounded = true;
    private string playerName = "Hero";
}
```

Important: Use 'f' after decimal numbers for float values (e.g., 5.5f). Public variables can be edited in the Unity Inspector, making them easy to adjust without changing code!

6. Control Structures

Control structures let you make decisions and repeat code based on conditions.

If Statements

```
if (health <= 0)
{
    Debug.Log("Player died!");
    Destroy(gameObject);
}
else if (health < 30)
{
    Debug.Log("Low health warning!");
}
else
{
    Debug.Log("Player is healthy");
}
```

For Loops

For loops repeat code a specific number of times:

```
// Spawn 10 enemies
for (int i = 0; i < 10; i++)
{
    Instantiate(enemyPrefab, new Vector3(i * 2, 0, 0), Quaternion.identity);
}
```

While Loops

While loops repeat code as long as a condition is true:

```
while (isAlive)
{
    // Keep playing the game
    UpdateGame();
}
```

Comparison Operators

Operator	Meaning	Example
==	Equal to	if (score == 100)
!=	Not equal to	if (health != 0)
<	Less than	if (speed < 10)
>	Greater than	if (score > 50)
<=	Less than or equal	if (health <= 20)

>=	Greater than or equal	if (level >= 5)
&&	AND (both true)	if (isAlive && hasKey)
||	OR (either true)	if (isDead || gameOver)

7. Functions and Methods

Functions (also called methods) are reusable blocks of code that perform specific tasks. They help organize your code and avoid repetition.

Creating a Function

```
// Function with no return value (void)
void PrintMessage()
{
    Debug.Log("Hello from my function!");
}

// Function with parameters
void TakeDamage(int damageAmount)
{
    health -= damageAmount;
    Debug.Log("Took " + damageAmount + " damage!");
}

// Function that returns a value
int CalculateScore(int kills, int deaths)
{
    int score = (kills * 100) - (deaths * 50);
    return score;
}
```

Calling Functions

```
void Start()
{
    // Call the function
    PrintMessage();

    // Call with parameters
    TakeDamage(20);

    // Store returned value
    int finalScore = CalculateScore(10, 3);
    Debug.Log("Final score: " + finalScore);
}
```

Function Components

Return type: What the function gives back (void means nothing, int means a whole number, etc.)

Function name: Use descriptive names in PascalCase (CalculateScore, MovePlayer)

Parameters: Values you pass into the function (in parentheses)

Function body: The code inside the curly braces

> **Best Practice:** Keep functions focused on one task. If a function is doing too many things, split it into multiple smaller functions. This makes your code easier to read and debug!

8. Unity-Specific C# Concepts

Unity extends C# with special features designed specifically for game development.

SerializeField Attribute

Use [SerializeField] to make private variables visible in the Inspector:

```
public class Player : MonoBehaviour
{
    [SerializeField] private int health = 100;
    [SerializeField] private float jumpForce = 5.0f;
}
```

Getting Components

Components are modular pieces of functionality. You often need to access other components:

```
public class PlayerMovement : MonoBehaviour
{
    private Rigidbody rb;

    void Start()
    {
        // Get the Rigidbody component on this GameObject
        rb = GetComponent<Rigidbody>();
    }
}
```

Finding GameObjects

```
// Find by name (slow, avoid in Update)
GameObject player = GameObject.Find("Player");

// Find by tag (better for frequently used objects)
GameObject enemy = GameObject.FindGameObjectWithTag("Enemy");

// Find all objects with tag
GameObject[] enemies = GameObject.FindGameObjectsWithTag("Enemy");
```

Instantiate and Destroy

```
// Create a copy of a prefab
GameObject bullet = Instantiate(bulletPrefab, transform.position,
transform.rotation);

// Destroy an object
Destroy(gameObject);

// Destroy after 3 seconds
Destroy(bullet, 3.0f);
```

Warning: Avoid using Find functions in Update() - they're slow! Instead, find objects once in Start() and store references to them.

9. MonoBehaviour Lifecycle

Unity calls certain functions automatically at specific times. Understanding this lifecycle is crucial for game development.

Essential Lifecycle Methods

Method	When Called	Common Uses
Awake()	Script instance created	Initialize references, set up data
Start()	Before first frame	Initialize after Awake, get components
Update()	Every frame	Input handling, non-physics movement
FixedUpdate()	Fixed time intervals	Physics calculations, Rigidbody movement
LateUpdate()	After all Updates	Camera following, final position adjustments
OnDestroy()	Object destroyed	Clean up, save data, unsubscribe events

Example Script

```
public class LifecycleExample : MonoBehaviour
{
    void Awake()
    {
        Debug.Log("Awake: Setting up");
    }

    void Start()
    {
        Debug.Log("Start: Ready to go!");
    }

    void Update()
    {
        // Called every frame (60+ times per second)
    }

    void FixedUpdate()
    {
        // Called at fixed intervals (physics)
    }
}
```

Order of Execution

1. **Awake()** - All Awake functions are called first
2. **Start()** - All Start functions are called after all Awakes
3. **Update()** - Called every frame
4. **FixedUpdate()** - Called at consistent intervals for physics

5. **LateUpdate()** - Called after all Updates

> **Key Insight:** Use FixedUpdate() for physics-based movement (Rigidbody), and Update() for input and non-physics code. This ensures smooth, consistent physics!

10. Working with GameObjects

GameObjects are the fundamental objects in Unity. Everything in your game is a GameObject with components attached to it.

Transform Component

Every GameObject has a Transform that controls position, rotation, and scale:

```
// Position
transform.position = new Vector3(0, 5, 0);
transform.position += Vector3.up * Time.deltaTime; // Move up

// Rotation
transform.Rotate(Vector3.up * 50 * Time.deltaTime); // Spin
transform.rotation = Quaternion.Euler(0, 90, 0);

// Scale
transform.localScale = new Vector3(2, 2, 2); // Double size
```

Vector3 Shortcuts

Vector3 Value	Coordinates	Description
`Vector3.zero`	(0, 0, 0)	Origin point
`Vector3.one`	(1, 1, 1)	All ones
`Vector3.up`	(0, 1, 0)	Upward direction
`Vector3.down`	(0, -1, 0)	Downward direction
`Vector3.forward`	(0, 0, 1)	Forward (Z-axis)
`Vector3.back`	(0, 0, -1)	Backward
`Vector3.right`	(1, 0, 0)	Right (X-axis)
`Vector3.left`	(-1, 0, 0)	Left

Moving Objects

```
public class SimpleMovement : MonoBehaviour
{
    public float speed = 5.0f;

    void Update()
    {
        // Move forward continuously
        transform.position += transform.forward * speed * Time.deltaTime;
    }
```

```
}
```

Critical: Always multiply movement by Time.deltaTime! This ensures smooth movement regardless of frame rate. Without it, movement speed varies based on computer performance.

11. Input Handling

Unity 6 supports both the legacy Input class and the new Input System. We'll cover the legacy Input class as it's simpler for beginners.

Keyboard Input

```
void Update()
{
    // Check if key is held down
    if (Input.GetKey(KeyCode.W))
    {
        transform.position += Vector3.forward * speed * Time.deltaTime;
    }

    // Check if key was just pressed (once)
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Jump();
    }

    // Check if key was just released
    if (Input.GetKeyUp(KeyCode.LeftShift))
    {
        StopSprinting();
    }
}
```

Input Axes

Unity's Input Manager provides axis values (-1 to 1) for smoother control:

```
void Update()
{
    // Get horizontal input (A/D or Left/Right arrows)
    float horizontal = Input.GetAxis("Horizontal");

    // Get vertical input (W/S or Up/Down arrows)
    float vertical = Input.GetAxis("Vertical");

    // Create movement vector
    Vector3 movement = new Vector3(horizontal, 0, vertical);
    transform.position += movement * speed * Time.deltaTime;
}
```

Mouse Input

```
void Update()
{
    // Left mouse button
    if (Input.GetMouseButtonDown(0))
    {
        Shoot();
    }

    // Right mouse button
    if (Input.GetMouseButton(1))
    {
        Aim();
    }

    // Mouse position
    Vector3 mousePos = Input.mousePosition;
}
```

Tip: Use GetKeyDown for actions that happen once (jumping, shooting), and GetKey for continuous actions (moving, aiming). This prevents actions from repeating every frame!

12. Building Your First Player Controller

Now it's time to put everything together! In this chapter, you'll create a complete player controller from scratch that handles walking, running, and jumping. We'll build this step-by-step, explaining every line of code along the way.

Step 1: Setting Up Your Scene

Before writing code, let's prepare our Unity scene:

1. Create a new 3D project in Unity 6 (or open an existing one)
2. Right-click in the Hierarchy and select 3D Object > Plane (this will be the ground)
3. Right-click in Hierarchy and select 3D Object > Capsule (this will be the player)
4. Name the capsule 'Player' and position it at (0, 1, 0) so it's above the ground
5. Select the Player, click Add Component in the Inspector, and add a Rigidbody
6. In the Rigidbody component, set Constraints > Freeze Rotation X, Y, Z to prevent the player from tipping over
7. Create a new C# Script in the Project window and name it 'PlayerController'

> **Why These Components?** The Rigidbody component adds physics to our player, allowing gravity and collisions to work. Freezing rotations prevents the capsule from falling over when it moves, which would look strange for a character controller.

Step 2: Understanding the Code Structure

Double-click the PlayerController script to open it in Visual Studio. You'll see this default code:

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

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

    }

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

    }
}
```

Now we'll replace this with our complete player controller. Delete the default Start() and Update() methods - we'll write our own!

Step 3: Adding Variables

First, let's add variables to control our player's movement. Add these inside the class, above where Start() was:

```
public class PlayerController : MonoBehaviour
{
    // Movement speeds
    [SerializeField] private float walkSpeed = 5f;
    [SerializeField] private float runSpeed = 8f;
    [SerializeField] private float jumpForce = 5f;

    // Ground detection
    [SerializeField] private LayerMask groundLayer;
    private bool isGrounded;

    // Components
    private Rigidbody rb;
    private float currentSpeed;
}
```

Let's break this down:
• **[SerializeField]** makes private variables visible in the Inspector so we can adjust them
• **walkSpeed** and **runSpeed** control how fast the player moves
• **jumpForce** determines how high the player jumps
• **groundLayer** helps detect when we're touching the ground
• **isGrounded** tracks whether we're on the ground (can't jump in air!)
• **rb** will store a reference to our Rigidbody component
• **currentSpeed** will be either walkSpeed or runSpeed depending on if we're running

Step 4: Initializing in Start()

Now add the Start() method to get our Rigidbody component:

```
    void Start()
    {
        // Get the Rigidbody component attached to this GameObject
        rb = GetComponent<Rigidbody>();

        // Check if Rigidbody exists
        if (rb == null)
        {
            Debug.LogError("PlayerController requires a Rigidbody component!");
        }
    }
```

This code runs once when the game starts. It finds the Rigidbody and shows an error if it's missing. Always check for required components - it makes debugging much easier!

Step 5: Detecting Ground

Before we can jump, we need to know if we're on the ground. Add this method:

```
    void CheckGround()
    {
        // Cast a ray downward from the player's position
        // If it hits something 0.1 units below us, we're grounded
        isGrounded = Physics.Raycast(transform.position, Vector3.down, 1.1f,
groundLayer);
    }
```

What's happening here? Physics.Raycast shoots an invisible ray downward. If it hits something on the groundLayer within 1.1 units, we know we're on the ground. The 1.1 value accounts for the capsule's height (we're checking slightly below the bottom of the player).

Step 6: Handling Movement Input

Now let's add the Update() method that handles player input:

```
    void Update()
    {
        // Check if we're on the ground
        CheckGround();

        // Determine current speed (running or walking)
        if (Input.GetKey(KeyCode.LeftShift))
        {
            currentSpeed = runSpeed;
        }
        else
        {
            currentSpeed = walkSpeed;
        }

        // Handle jump input
        if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        {
            Jump();
        }
    }
```

Breaking it down:
• We call CheckGround() every frame to update our ground status
• If Left Shift is held, use runSpeed; otherwise use walkSpeed
• If Space is pressed AND we're grounded, call the Jump() method
• Note: GetKeyDown only triggers once when pressed, perfect for jumping!

Step 7: Physics-Based Movement

For smooth physics movement, we use FixedUpdate() instead of Update(). Add this method:

```
    void FixedUpdate()
    {
        // Get input from WASD or arrow keys
        float horizontal = Input.GetAxis("Horizontal"); // A/D or Left/Right
        float vertical = Input.GetAxis("Vertical"); // W/S or Up/Down

        // Create movement direction
        Vector3 movement = new Vector3(horizontal, 0f, vertical);
```

```
        movement = movement.normalized; // Prevent faster diagonal movement

        // Apply movement to Rigidbody
        Vector3 newVelocity = movement * currentSpeed;
        newVelocity.y = rb.linearVelocity.y; // Keep existing vertical velocity
(gravity/jump)
        rb.linearVelocity = newVelocity;
    }
```

Why FixedUpdate? Physics calculations happen at fixed intervals (default 50 times per second). Using FixedUpdate ensures smooth, consistent physics-based movement. Update() can run at variable frame rates (60, 120, 144 FPS), which would make physics unpredictable.

Step 8: Implementing Jump

Finally, let's add the Jump() method:

```
    void Jump()
    {
        // Apply upward force for jumping
        rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);

        // Optional: Play jump sound here
        // AudioSource.PlayClipAtPoint(jumpSound, transform.position);
    }
```

AddForce applies physics force to the Rigidbody. ForceMode.Impulse applies an instant burst of force, perfect for jumping. The force goes straight up (Vector3.up) multiplied by our jumpForce value.

Step 9: The Complete Script

Here's the entire PlayerController script put together. Copy this into your Visual Studio:

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    // Movement speeds
    [SerializeField] private float walkSpeed = 5f;
    [SerializeField] private float runSpeed = 8f;
    [SerializeField] private float jumpForce = 5f;

    // Ground detection
    [SerializeField] private LayerMask groundLayer;
    private bool isGrounded;

    // Components
    private Rigidbody rb;
    private float currentSpeed;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        if (rb == null)
        {
            Debug.LogError("PlayerController requires a Rigidbody!");
        }
    }

    void Update()
    {
        CheckGround();

        // Set speed based on shift key
        currentSpeed = Input.GetKey(KeyCode.LeftShift) ? runSpeed : walkSpeed;

        // Jump when space pressed and grounded
        if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        {
            Jump();
        }
    }

    void FixedUpdate()
    {
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(h, 0f, v).normalized;
        Vector3 newVelocity = movement * currentSpeed;
```

```
        newVelocity.y = rb.linearVelocity.y;
        rb.linearVelocity = newVelocity;
    }

    void CheckGround()
    {
        isGrounded = Physics.Raycast(transform.position, Vector3.down, 1.1f, groundLayer);
    }

    void Jump()
    {
        rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
    }
}
```

Step 10: Setting Up in Unity

Now let's configure everything in Unity:

1. Save your script in Visual Studio (Ctrl+S)
2. Go back to Unity (it will compile automatically)
3. Drag the PlayerController script onto your Player GameObject
4. Select the Player in the Hierarchy
5. In the Inspector, find the PlayerController component
6. You'll see Walk Speed, Run Speed, Jump Force, and Ground Layer fields
7. For Ground Layer, click the dropdown and select 'Default' (or create a 'Ground' layer)
8. Make sure your ground plane is on the same layer you selected

Step 11: Testing Your Controller

Press the Play button in Unity and test your controller:

• **WASD or Arrow Keys:** Move the player
• **Left Shift + Movement:** Run (notice the speed difference!)
• **Spacebar:** Jump (only works when on the ground)

Try adjusting the values in the Inspector while playing to see how they affect movement. This is one of Unity's best features - real-time tweaking!

Step 12: Troubleshooting Common Issues

Player falls through the ground:
• Make sure your ground has a Collider component
• Check that the Ground Layer is set correctly

Can't jump:
• Verify isGrounded is true (add a Debug.Log to check)
• Make sure Ground Layer matches your ground object's layer
• Try increasing the raycast distance in CheckGround() to 1.5f

Movement feels slippery:
• Select your Player's Rigidbody
• Increase Drag to 1 or 2
• You can also set linearDamping in code: rb.linearDamping = 5f;

Player tips over:
• Make sure Rigidbody Constraints has X, Y, Z rotation frozen

Congratulations! You've just built your first complete player controller from scratch! This is a major milestone. You now understand how to combine input, physics, and game logic. Try enhancing it by adding: double jump, wall running, crouching, or even a stamina system!

13. Creating Unity Editor Windows

Custom Editor Windows extend Unity's interface with your own tools. These are powerful for creating level editors, asset managers, batch processors, and development utilities. Editor scripts only run in the Unity Editor, never in your built game.

> **Important:** Editor scripts must be placed in a folder named 'Editor' inside your Assets folder. Create this folder if it doesn't exist: right-click Assets > Create > Folder, name it 'Editor'.

Understanding Editor Scripts

Editor scripts use the UnityEditor namespace and inherit from EditorWindow instead of MonoBehaviour. They have access to special Editor APIs that let you create custom interfaces and tools.

Creating Your First Editor Window

Let's create a simple tool that helps you organize your scene. Create a new C# script in the Editor folder and name it 'ObjectOrganizer':

```
using UnityEngine;
using UnityEditor;

public class ObjectOrganizer : EditorWindow
{
    // Create menu item to open this window
    [MenuItem("Tools/Object Organizer")]
    public static void ShowWindow()
    {
        // Creates and displays the window
        GetWindow<ObjectOrganizer>("Object Organizer");
    }

    // Called to draw the window's GUI
    void OnGUI()
    {
        GUILayout.Label("Object Organization Tools", EditorStyles.boldLabel);

        if (GUILayout.Button("Create Folder Structure"))
        {
            CreateFolderStructure();
        }
    }

    void CreateFolderStructure()
    {
        Debug.Log("Creating folder structure...");
        // Your folder creation code here
    }
}
```

Breaking it down:
• **[MenuItem]** creates a new menu option in Unity's top menu bar
• **ShowWindow()** must be static - Unity calls it to open your window
• **GetWindow<>()** creates or brings focus to your window
• **OnGUI()** is called to draw the window's interface
• **GUILayout** automatically positions UI elements vertically

After saving this script, go to Unity and click Tools > Object Organizer in the menu bar. Your custom window appears! You can dock it like any Unity window.

Building a More Advanced Tool

Let's create a useful tool: a Prefab Spawner that helps you quickly place objects in your scene. Create a new script in the Editor folder named 'PrefabSpawner':

```
using UnityEngine;
using UnityEditor;

public class PrefabSpawner : EditorWindow
{
    // Variables to store user input
    private GameObject prefabToSpawn;
    private int spawnCount = 10;
    private float spawnRadius = 5f;
    private bool randomRotation = true;

    [MenuItem("Tools/Prefab Spawner")]
    public static void ShowWindow()
    {
        GetWindow<PrefabSpawner>("Prefab Spawner");
    }

    void OnGUI()
    {
        // Title
        GUILayout.Label("Prefab Spawning Tool", EditorStyles.boldLabel);
        EditorGUILayout.Space();

        // Prefab selection field
        prefabToSpawn = (GameObject)EditorGUILayout.ObjectField(
            "Prefab to Spawn", prefabToSpawn, typeof(GameObject), false);

        // Number slider
        spawnCount = EditorGUILayout.IntSlider("Spawn Count", spawnCount, 1, 100);

        // Radius slider
        spawnRadius = EditorGUILayout.Slider("Spawn Radius", spawnRadius, 1f, 50f);

        // Toggle for random rotation
        randomRotation = EditorGUILayout.Toggle("Random Rotation", randomRotation);

        EditorGUILayout.Space();

        // Spawn button
        if (GUILayout.Button("Spawn Prefabs", GUILayout.Height(30)))
        {
            SpawnPrefabs();
        }

        // Clear button
        if (GUILayout.Button("Clear All Spawned", GUILayout.Height(30)))
        {
            ClearSpawnedObjects();
        }
    }

    void SpawnPrefabs()
    {
        if (prefabToSpawn == null)
        {
            EditorUtility.DisplayDialog("Error", "Please select a prefab!", "OK");
            return;
        }

        // Create parent object
        GameObject parent = new GameObject("Spawned Objects");
        Undo.RegisterCreatedObjectUndo(parent, "Spawn Prefabs");

        for (int i = 0; i < spawnCount; i++)
        {
            // Random position within radius
```

```
            Vector2 randomCircle = Random.insideUnitCircle * spawnRadius;
            Vector3 spawnPos = new Vector3(randomCircle.x, 0, randomCircle.y);

            // Random rotation if enabled
            Quaternion rotation = randomRotation ?
                Quaternion.Euler(0, Random.Range(0f, 360f), 0) :
                Quaternion.identity;

            // Instantiate prefab
            GameObject obj = (GameObject)PrefabUtility.InstantiatePrefab(prefabToSpawn);
            obj.transform.position = spawnPos;
            obj.transform.rotation = rotation;
            obj.transform.SetParent(parent.transform);
        }

        Debug.Log($"Spawned {spawnCount} objects!");
    }

    void ClearSpawnedObjects()
    {
        GameObject parent = GameObject.Find("Spawned Objects");
        if (parent != null)
        {
            Undo.DestroyObjectImmediate(parent);
            Debug.Log("Cleared all spawned objects!");
        }
    }
}
```

Understanding the Advanced Features

EditorGUILayout.ObjectField: Creates a drag-and-drop field for Unity objects. The 'false' parameter means we only accept assets from the Project window, not scene objects.

EditorGUILayout.IntSlider: Creates a slider for integer values. Much better than typing numbers!

EditorUtility.DisplayDialog: Shows popup messages. Use this for errors, confirmations, or success messages.

Undo.RegisterCreatedObjectUndo: Critical! This makes your object creation undoable with Ctrl+Z. Always register undo for editor operations that modify the scene.

PrefabUtility.InstantiatePrefab: Better than Instantiate() in editor scripts because it maintains the prefab connection, allowing you to apply changes back to the prefab.

Common Editor GUI Elements

GUI Element	Code Example	Use Case
`Label`	`GUILayout.Label("Text");`	Display text
`Button`	`GUILayout.Button("Click");`	Trigger actions
`TextField`	`EditorGUILayout.TextField("Name", str);`	Text input
`IntField`	`EditorGUILayout.IntField("Value", num);`	Number input
`FloatField`	`EditorGUILayout.FloatField("Speed", f);`	Decimal input
`Toggle`	`EditorGUILayout.Toggle("Enable", bool);`	Checkbox
`Slider`	`EditorGUILayout.Slider("Volume", v, 0, 1);`	Range value
`ColorField`	`EditorGUILayout.ColorField("Color", col);`	Color picker
`ObjectField`	`EditorGUILayout.ObjectField("Obj", obj, t);`	Asset reference
`Space`	`EditorGUILayout.Space();`	Add spacing
`HelpBox`	`EditorGUILayout.HelpBox("Info", type);`	Info messages

Best Practices for Editor Windows

1. **Always support Undo:** Use Undo.RegisterCreatedObjectUndo, Undo.RecordObject, etc.
2. **Validate input:** Check for null references and invalid values before processing
3. **Provide feedback:** Use Debug.Log or EditorUtility.DisplayDialog to inform users
4. **Save window state:** Use EditorPrefs to remember settings between sessions
5. **Add help text:** Use EditorGUILayout.HelpBox to explain what your tool does

6. **Handle errors gracefully:** Wrap operations in try-catch blocks

> **Real-World Uses:** Custom editor windows are used for batch renaming objects, generating procedural levels, managing save data, creating dialogue editors, building asset bundles, and countless other workflow improvements. Learning this skill will make you much more productive!

14. Practical Examples

Let's put it all together with complete, working examples you can use in your games.

Example 1: Player Controller

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 5.0f;
    [SerializeField] private float jumpForce = 5.0f;
    private Rigidbody rb;
    private bool isGrounded = true;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Movement
        float h = Input.GetAxis("Horizontal");
        float v = Input.GetAxis("Vertical");
        Vector3 move = new Vector3(h, 0, v) * moveSpeed;
        rb.linearVelocity = new Vector3(move.x, rb.linearVelocity.y, move.z);

        // Jump
        if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
        {
            rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
            isGrounded = false;
        }
    }

    void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Ground"))
        {
            isGrounded = true;
        }
    }
}
```

Example 2: Health System

```
using UnityEngine;

public class Health : MonoBehaviour
{
    [SerializeField] private int maxHealth = 100;
    private int currentHealth;

    void Start()
    {
        currentHealth = maxHealth;
    }

    public void TakeDamage(int damage)
    {
        currentHealth -= damage;
        Debug.Log("Health: " + currentHealth);

        if (currentHealth <= 0)
        {
            Die();
```

```
        }
    }

    public void Heal(int amount)
    {
        currentHealth = Mathf.Min(currentHealth + amount, maxHealth);
        Debug.Log("Healed! Health: " + currentHealth);
    }

    void Die()
    {
        Debug.Log(gameObject.name + " died!");
        Destroy(gameObject);
    }
}
```

Example 3: Simple Enemy AI

```
using UnityEngine;

public class EnemyAI : MonoBehaviour
{
    [SerializeField] private float moveSpeed = 3.0f;
    [SerializeField] private float detectionRange = 10.0f;
    private Transform player;

    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }

    void Update()
    {
        // Calculate distance to player
        float distance = Vector3.Distance(transform.position, player.position);

        // Chase player if in range
        if (distance < detectionRange)
        {
            // Look at player
            Vector3 direction = (player.position - transform.position).normalized;
            transform.LookAt(player);

            // Move towards player
            transform.position += direction * moveSpeed * Time.deltaTime;
        }
    }
}
```

Example 4: Collectible Item

```
using UnityEngine;

public class Collectible : MonoBehaviour
{
    [SerializeField] private int pointValue = 10;
    [SerializeField] private float rotationSpeed = 50.0f;

    void Update()
    {
        // Rotate for visual effect
        transform.Rotate(Vector3.up * rotationSpeed * Time.deltaTime);
    }

    void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            // Add points (assuming GameManager exists)
            Debug.Log("Collected! +" + pointValue + " points");

            // Destroy the collectible
            Destroy(gameObject);
        }
    }
}
```

Try It: Copy these examples into Unity and experiment! Change values in the Inspector, add Debug.Log statements to see what's happening, and modify the code to create your own variations.

15. Best Practices and Next Steps

Coding Best Practices

1. Name things clearly: Use descriptive names like 'playerHealth' instead of 'pH'. Your future self will thank you!

2. Comment your code: Explain why you're doing something, not just what. Comments help you remember your logic later.

3. Keep it simple: Start with simple code that works, then improve it. Don't try to make everything perfect the first time.

4. Test frequently: Test your code often as you write it. It's easier to fix small bugs than to debug a massive script.

5. Use SerializeField: Make values adjustable in the Inspector so you can tweak them without changing code.

6. Avoid code in Update when possible: Update runs every frame (60+ times per second). Heavy calculations here can slow your game.

Common Beginner Mistakes

• Forgetting to multiply movement by Time.deltaTime
• Using Find functions inside Update (very slow!)
• Not checking for null references before using them
• Forgetting the 'f' after float numbers (e.g., 5.5f)
• Script name not matching the class name
• Not attaching scripts to GameObjects

Debugging Tips

```
// Use Debug.Log to print to the Console
Debug.Log("Player health: " + health);

// Draw debug lines in the Scene view
Debug.DrawRay(transform.position, Vector3.forward * 10, Color.red);

// Pause the game when something happens
if (health <= 0)
{
    Debug.Break(); // Pauses in editor
}
```

Next Steps in Your Journey

1. Unity Learn: Unity's official learning platform has free tutorials and courses.

2. Unity Documentation: The Unity Scripting API reference is your best friend. Look up any function you're curious about!

3. Build small projects: Make simple games like Pong, a platformer, or a simple shooter. Learning by doing is the best way!

4. Join the community: Unity forums, Reddit (r/Unity3D), and Discord servers are great places to ask questions and learn.

5. Study other code: Look at free Unity assets and tutorial projects to see how experienced developers structure their code.

Advanced Topics to Explore

- Coroutines (for timed sequences)
- Events and delegates (for communication between scripts)
- Scriptable Objects (for data management)
- Object pooling (for performance)
- Animation controllers (for character animations)
- Unity's new Input System
- Cinemachine (for camera control)
- NavMesh AI (for pathfinding)

Remember: Every expert was once a beginner. Programming is a skill that improves with practice. Don't get discouraged by errors - they're learning opportunities! Keep experimenting, keep building, and most importantly, have fun creating games!

Happy Coding!

- The Foxnova Studios Team