Level Up Your Code: Practical Unity Game Development (Week 4)

Welcome to Week 4! This week, we move from theory to action, focusing on essential practical skills for building functional, engaging game levels in Unity. We will cover everything from foundational design concepts to implementing core player systems and vital development tools.

Our focus is on creating a solid, working prototype. By the end of this session, you'll have a clear understanding of the full workflow: from initial level structure to setting up the player character with modern input and combat mechanics.

> This lecture is designed to be read and practiced. The detailed explanations and code examples will allow you to implement these concepts directly in your Unity projects. Let's make something great!

Chapter 1: Foundations – Level Design & Game Systems

1. Level Design: More Than Just Geometry

Level design is the process of creating environments—maps, stages, or worlds—that players interact with. It's not just about aesthetics; it's about guiding player experience, setting difficulty curves, and communicating the narrative.

- **Flow & Pacing:** Structure your level to control the player's movement and the rhythm of gameplay (e.g., quiet exploration followed by intense combat).
- **Visual Language:** Use lighting, color, and object placement to visually guide the player toward objectives and critical areas.
- **Three-Act Structure:** Many levels follow a mini-narrative: Introduction (safe space, tutorials), Conflict (main challenge, escalating threats), and Resolution (boss fight or objective completion).

2. Game Systems: The Engine of Gameplay

Game systems are the rules and mechanics that govern how the game works. Think of them as the interconnected cogs that define player capabilities, environmental interactions, and the core loop.

- **Core Loop:** The basic, repeating cycle of actions (e.g., Explore -> Fight -> Loot -> Upgrade -> Repeat).
- **Interconnectedness:** A change in one system (like player movement speed) often affects others (like combat difficulty or enemy detection range).
- **Examples:** Health/Damage System, Inventory System, Quest System, Input System.

A well-designed level uses geometric and visual cues (Level Design) to express and facilitate the underlying rules and interactions (Game Systems).

Development Workflow: Essential Tools for Every Developer

While creating levels and systems is fun, mastering development tools is crucial for efficiency and debugging.

3. Taking Screenshots: Capturing Key Moments

Screenshots are vital for documentation, bug reporting, marketing, and sharing progress. Unity offers simple built-in ways to capture the game view, but C# code allows for powerful, high-resolution captures.

Why Code for Screenshots?

- Capture at specific times (e.g., right when a bug occurs).
- Define custom resolutions (e.g., high-res marketing images).
- Save files automatically with custom naming conventions.

The key function is `ScreenCapture.CaptureScreenshot()`, which saves a PNG file to your project folder or a defined path.

C# Code Example: Screenshot Utility

```
using UnityEngine;

public class ScreenshotTool : MonoBehaviour
{
 // Assign a key in the Inspector
 public KeyCode screenshotKey = KeyCode.K;

 void Update()
 {
 if (Input.GetKeyDown(screenshotKey))
 {
 CaptureScreenshot();
 }
 }

 void CaptureScreenshot()
 {
 string folderPath = Application.dataPath + "/Screenshots/";
 // Ensure the folder exists
 if (!System.IO.Directory.Exists(folderPath))
 System.IO.Directory.CreateDirectory(folderPath);

 string fileName = "Screenshot_" +
 System.DateTime.Now.ToString("yyyy-MM-dd_HH-mm-ss") + ".png";

 ScreenCapture.CaptureScreenshot(folderPath + fileName);
 Debug.Log("Screenshot saved to: " + folderPath + fileName);
 }
}
```

Chapter 2: Project Setup – Starting Clean and Modern

Before writing player code, we need a modern and optimized Unity environment.

4. Create New Unity Project

Start fresh! Choose the 3D URP (Universal Render Pipeline) template for modern visuals and better performance across platforms. Name your project clearly and select a local directory. This is the foundation for all our work.

5. Get the New Input System

The legacy Unity Input Manager is outdated. We must use the modern Input System package for flexibility, controller mapping, and better separation of concerns. Install it via the Package Manager (Window > Package Manager).

Why the New Input System?

It allows you to define Actions (e.g., "Jump," "Move") that can be mapped to different physical controls (keyboard, gamepad) without changing code. It's asynchronous and cleaner to manage.

Step-by-Step: Installing the Input System

- Open **Window > Package Manager**.
- Select "Packages: Unity Registry" from the dropdown.
- Search for "Input System" and click **Install**.
- Unity will prompt you to enable the new Input System and restart the editor. Accept the restart.

Chapter 3: Player Core – Visual Setup and Basic Movement

The player character needs a visual representation and the ability to move through the world.

6. Player Setup: Visuals and Hierarchy

A clean hierarchy is essential for managing a complex character. Use an empty GameObject as the root, which holds the components (like the Character Controller and Rigidbody), and parent the visual model to it.

Root Object (Player)

Contains: `CharacterController`, `Rigidbody` (if needed), and the main Player script. Position: (0, 0, 0).

Child Object (Model/Mesh)

Contains: The 3D model, Animator, and materials. This allows us to rotate the visual model without affecting the physics/controller capsule.

Camera Arm (Optional)

An empty object positioned behind the player, used to smoothly follow the character's movement and rotation without inheriting rotation directly.

Player Control: Implementing Modern Movement Input

7. Player Setup: Input Control using Action Maps

We will use the new Input System's `PlayerInput` component and C# events to handle movement. This separates control schemes from movement logic.

Step 1: Create an Input Action Asset

Right-click in the Project window > Create > Input Actions. This asset defines the **Action Maps** (e.g., 'Gameplay', 'Menu') and the specific **Actions** (e.g., 'Move', 'Look', 'Jump').

Step 2: Define the 'Move' Action

Create an Action named 'Move' with a Control Type of `Vector 2`. Bind it to WASD/Arrow keys (as a 2D composite) and the Left Stick of a gamepad. This provides a normalized vector (-1 to 1) for movement direction.

Step 3: Handle Movement in C#

Attach a `PlayerInput` component to the Player root object. Set its behavior to 'Invoke Unity Events'. This generates C# events we can subscribe to in our Player script.

C# Code Example: Subscribing to Input

```
// Requires: using UnityEngine.InputSystem;

public class PlayerMovement : MonoBehaviour
{
 private Vector2 moveInput;
 public float speed = 5f;
 private CharacterController controller;

 void Start()
 {
 controller = GetComponent();
 }

 // Called automatically by PlayerInput component when
"Move" action is performed
 public void OnMove(InputValue value)
 {
 moveInput = value.Get();
 }

 void FixedUpdate()
 {
 // Calculate direction relative to the world
 Vector3 direction = new Vector3(moveInput.x, 0,
moveInput.y);

 // Use CharacterController.Move for physics-safe movement
 controller.Move(direction * speed * Time.deltaTime);
 }
}
```

Core Systems Implementation: Health and Damage

A fundamental game system is tracking the player's health and handling damage events. This requires robust logic to ensure consistency and modularity.

8. Player Setup: Health and Damage System

We create a dedicated `HealthSystem` script that manages the current health value, maximum health, and includes public methods for applying damage and healing. This script can be reused for enemies, NPCs, and the player.

Health Variable

Use a public property or a serialized field for the current health and maximum health (e.g., `float currentHealth`). It's often helpful to use an Event or C# Action to notify UI elements when health changes.

Take Damage Method

The `TakeDamage(float damageAmount)` method is crucial. It subtracts the damage, clamps the health (ensuring it doesn't go below zero), and checks if the player has died. It should be called by external objects (like enemy attacks).

Handling Death

If `currentHealth <= 0`, trigger the death sequence (e.g., animation, sound, respawn/game over logic). This should be a clean, centralized process in the Health System.

C# Health System: Modular and Event-Driven

Using events (even simple debug logs) ensures that our health logic is decoupled from the UI or animation logic. This is key to professional, scalable game architecture.

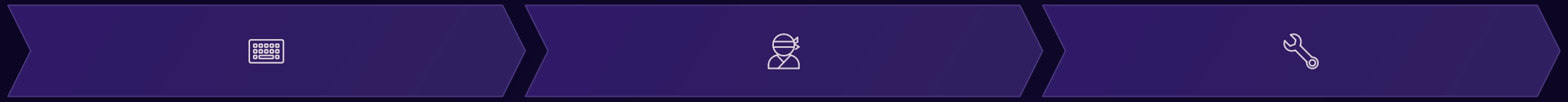
```
using UnityEngine;
using UnityEngine.Events;

public class HealthSystem : MonoBehaviour
{
    [SerializeField] private float maxHealth = 100f;
    private float currentHealth;

    // Unity Event for UI updates (optional, but highly recommended)
    public UnityEvent OnHealthChanged;

    void Awake()
{
    currentHealth = maxHealth;
}

public void TakeDamage(float damage)
{
    if (currentHealth <= 0) return; // Already dead

    currentHealth -= damage;
    currentHealth = Mathf.Clamp(currentHealth, 0, maxHealth);

    OnHealthChanged.Invoke(); // Notify listeners (UI, effects)

    if (currentHealth <= 0)
    {
        Die();
    }
    else
    {
        // Optional: Play hit reaction animation/sound
        Debug.Log(gameObject.name + " took " + damage + " damage. Health remaining: " + currentHealth);
    }
}

    private void Die()
    {
        Debug.Log(gameObject.name + " has died!");
        // Implement game over or respawn logic here
        // Example: gameObject.SetActive(false);
    }

    public float GetCurrentHealth()
    {
        return currentHealth;
    }
}
```

To test this, you could create a simple `DamageDealer` script on an enemy or a hazard that calls `player.GetComponent<HealthSystem>().TakeDamage(10);` when the player enters its trigger.

Player Combat Setup: Designing the Fight

9. Player Setup: Simple Fighting Mechanics

Setting up combat involves three main components: Input (when to attack), Animation (what the attack looks like), and Detection (what the attack hits and how much damage it deals).



Input Action

Define an 'Attack' action in your Input Action Asset, mapped to the left mouse button or a gamepad face button. Use the `OnAttack` event in C# to trigger the start of the attack sequence.

Animation Trigger

When the input is detected, set an Animator trigger parameter (e.g., "Attack"). The animation system then plays the attack sequence. This is where the visual feedback occurs.

Hit Detection

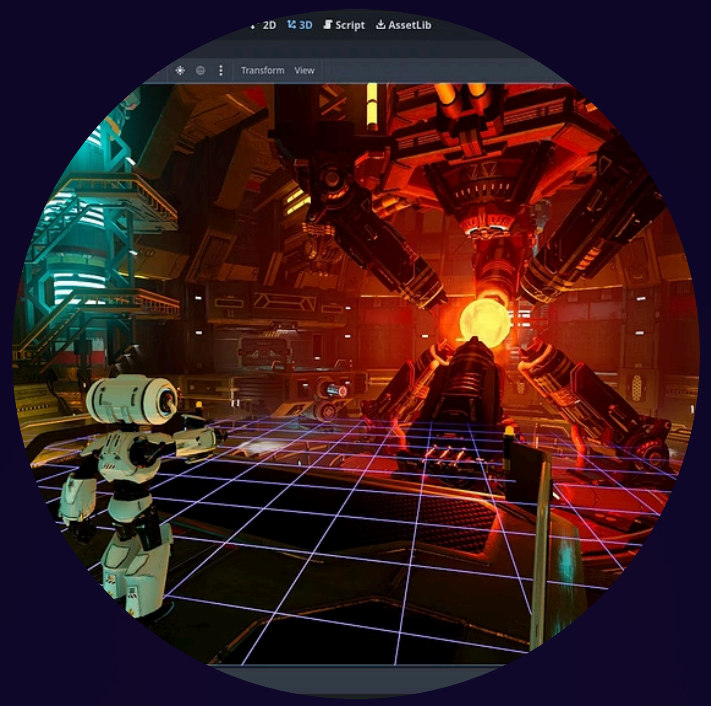
The actual damage-dealing moment should be linked to the animation. Use an `Animation Event` placed at the exact frame the weapon connects to trigger a short-lived collision box (or sphere cast) that checks for colliders with a `HealthSystem` component.

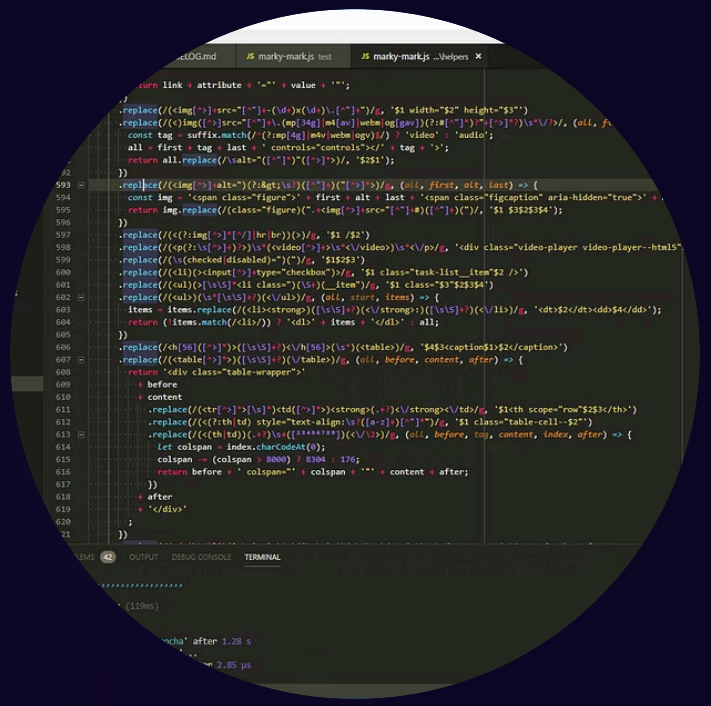
This decoupling ensures the attack input is instantaneous, but the damage is tied precisely to the visual impact point, making combat feel responsive and fair.

> **Pro Tip:** Use the `Physics.OverlapSphere` or `Physics.BoxCast` methods in C# for efficient, precise hit detection without relying on constantly active colliders.

10. Additional Resources & Next Steps

Game development is a continuous learning process. Here are some essential resources and a look ahead to keep your momentum going.





Unity Documentation

The primary source of truth. Always check the official manual and scripting API when encountering new components or methods (like `ScreenCapture` or `InputSystem`). Master the documentation search!

Unity Learn

Contains free, guided projects and tutorials on almost every topic, from introductory concepts to advanced visual effects and networking. Excellent for hands-on learning.

Community Forums

The Unity Forums and communities like Reddit's `r/Unity3D` are invaluable for troubleshooting niche errors and seeing how professional and hobbyist developers solve common problems.

Your Challenge: Apply the Health and Input Systems to your player character this week!

Remember the core principle of game development: start small, iterate often, and ensure every system you build is testable and modular. Good luck with your implementation!