Unity Certified Associate: Programmer Courseware

By:

Tarun Kumar Poddar

Table of Contents

[1. Introduction to Unity 3](#_Toc191033315)

[2. Getting started with Unity 4](#_Toc191033316)

[2.1 Installation and Setup Unity 4](#_Toc191033317)

[2.2 Build Settings 4](#_Toc191033318)

[3. Scene Setup 5](#_Toc191033319)

[3.1 Integrate XR-Origin 5](#_Toc191033320)

[3.2 Integrate AR Session 5](#_Toc191033321)

[3.3 Set Rendering Pipeline 5](#_Toc191033322)

[4. Augmenting Objects 6](#_Toc191033323)

[4.1 Prefabs 6](#_Toc191033324)

[4.2 Adding Cube 6](#_Toc191033325)

[4.3 Plane Detection 6](#_Toc191033326)

[4.4 Raycast Manager 6](#_Toc191033327)

[4.4 Adding Physics 6](#_Toc191033328)

[4.5 Placing cube on detected plane 7](#_Toc191033329)

[5 Creating UI 9](#_Toc191033330)

[5.1 Add button on Canvas 9](#_Toc191033331)

[6. Replace cube by Foot Model 11](#_Toc191033332)

[7. Interactions with Object 12](#_Toc191033333)

[7.1 Transform an object by touch 12](#_Toc191033334)

# Introduction

**Overview**

Demonstrate core skills and competencies across programming, UI, debugging and asset management to help you obtain your first professional programming role with Unity.

**Prerequisites**

* 2-3 semesters of post-secondary Unity classwork or equivalent independent study
* Experience with a diverse range of Unity projects
* Importing assets or code, including from the Unity Asset Store or Unity Package Manager, and addressing conflicts that arise as a result
* Performing debugging of non-complex problems
* Interpreting pre-existing, well-documented code
* Integrating and modifying pre-existing well-documented code
* Building basic scene management, including loading scenes
* Creating, editing, and using Prefabs
* Deploying a basic build

# Syllabus

Unity Programming

* Evaluate code for integration into an existing system created/architected by a lead
* Apply coding standards and best practices as guided by senior programmers
* Determine code that would accomplish a specified interaction or programming logic
* Determine the process to implement transitions between scenes
* Save data between scenes and between sessions using approaches such as static variables and PlayerPrefs
* Obtain a defined result by using Unity API methods, given Unity’s API documentation
* Select the appropriate properties, scripts, and components of GameObjects for required tasks
* Explain the differences between basic inheritance and interfaces
* Choose the appropriate commonly used data structures for a specific situation including but not limited to lists, arrays, and dictionaries
* Choose the appropriate data types for a specific situation including but not limited to floats, bools, and strings
* Build an application to WebGL or a personal computer

UI

* Arrange UI components on the canvas according to a defined layout using anchors, pivots, and groups
* Identify the process required to display data in various UI elements
* Explain how to use the UnityEvent system to respond to User Input

Debugging

* Program debug messages to identify the possible causes of code failing to execute as expected
* Identify the cause of a compilation error, given a block of code
* Identify errors caused by a null variable
* Identify techniques required to refactor and improve code to fit defined coding standards
* Select appropriate profiling tools to identify the sources of performance problems

Asset Management

* Explain how to use prefabs in a scene
* Describe the process and outcomes for changing a nested prefab or prefab variant
* Explain the primary purposes of version control when working in Unity

# Course

## Course 1: Platformer

Identify the “Associate Programmer Platformer” project and its key elements.

Play the game and notice the things:

1. Skippable Beginning Cinematic
2. Avatar and Camera are Controllable
3. Player Can Jump and Collect Coins
4. Two Types of Enemies
   1. Red: Can follow the player
   2. Blue: Move in fix points only
5. Enemies Cause Player to be Shoved Back
6. Player Can Destroy Enemies
7. Moving Platforms and Unclimbable Slopes
8. The Game Resets Once the Player Touches the Spinning End Zone

### 3.1.1 Session 1 – Player Health System

Challenge 1

**Requirements**: Integrate a simple health and damage system for the player. The player needs to take damage when an enemy touches them but not when the player stomp/kill the enemy.

* Print out the player’s health.
* Player health should be edited in the inspector.

Key Elements:

1. **Best Place to Store the Player's Health State**
   * Typically, this would be within the **Player class or object**, as it directly represents the player's attributes and behaviors. This ensures encapsulation and easy access during gameplay.
2. **Best Mechanism for Updating the Player's Health**
   * A good mechanism would be a **dedicated method** like updateHealth(damageAmount) or takeDamage(amount) within the Player class. This allows centralized control over health logic, including clamping values, triggering animations, or checking for death.
3. **Best Place and Time to Use That Mechanism When the Player Comes in Contact with an Enemy**
   * The ideal time is **during collision detection** between the player and an enemy. This logic is often handled in the **game loop or physics engine**, where you can check for collisions and then call the health update method accordingly.

Hint:

1. Identify what is controlling the player’s movement.
2. What code is responsible for pushing the player backward when player is touching the enemy.

Solution :

1. **PlayerMovement.cs** scriptis responsible for the player movement and stats.
2. Add a public field health in **Stats** struct: public float health;
3. Add method **UpdateHealth(float amount).**

public void UpdateHealth(float amount)

{

playerStats.health += amount;

Debug.Log($"Updated health by: {amount}, Player's current Health: {playerStats.health}");

}

1. **GetHit** scripthas logi**c** whengetting a hit by enemy or trap.
2. Update **TakeDamage()** method **:** playerMovementScript.UpdateHealth(-1);

**Challenge 1: Extra Challenge** Examining a Project & Adding a Health System We have all these coins floating around, let's put them to some good use. Design a system that allows you to collect 10 coins to add one health back to the player’s health. The player's health should still be capped at it’s starting base health. Consider how we can integrate this new system into our existing codeset and use the new functionality we created in this session’s challenge

Solution:

Update **CollectCoins** script:

if(ScoreManager.score % 100 == 0 && playerMovementScript.playerStats.health < 100)

{

playerMovementScript.UpdateHealth(1);

}

### 3.1.2 Session 2 – Buf Fix

Let’s fix some bugs now.

1. Fix ***PatrolEnemy*** script, uncomment the code and make sure it moves between 3 points A,B and C.
2. Fix EnemyExploding game object, enable it. It compiles but does not works in play mode.

Solution Fix 1:

Update the code:  
private void Update()

{

if (enemyStats.move == true)

{

Vector3 moveToPoint = patrolPoints[currentPatrolPoint].position;

transform.position = Vector3.MoveTowards(transform.position, moveToPoint, enemyStats.speed \* Time.deltaTime);

if (Vector3.Distance(transform.position, moveToPoint) < 0.01f)

{

currentPatrolPoint++;

if (currentPatrolPoint >= patrolPoints.Length)

{

currentPatrolPoint = 0;

}

}

}

}

Solution Fix 2:

Update ExplodeEnemy.cs

1. Update the update method:

private void Update()

{

Debug.Log($"Player distance :{Vector3.Distance(transform.position, playerTransform.position)}");

//Check to see how close the player is to the enemy

if (Vector3.Distance(transform.position, playerTransform.position) < enemyStats.explodeDist)

{

//Explode if player is within range

StartCoroutine("Explode");

}

}

2. Update the Explode method:

private IEnumerator Explode()

{

yield return new WaitForSeconds(0.2f);

Instantiate(enemyExplosionParticles, transform.position, Quaternion.identity);

Destroy(transform.parent.gameObject);

}

1. In Unity, Goto EnemyExploding gameobject > Enemy > Explode Enemy Script > Explode Dist = 5.

### 3.1.3 Session 3 – Refactor

Enable “Refactor Enemy” game object and observe in play.

**Challenge 3**

**Making it Better**

Exam Objectives Covered

● Identify techniques required to refactor and improve code to fit defined coding standards

Challenge Overview: Refactoring Enemy Code

Welcome to Challenge 3! I

n this session, you’ll refine your code optimization skills by refactoring

the code for a new enemy, the RefactorEnemy. This enemy combines the behaviors of the two

enemies you debugged in the previous challenge.

Your task is to refine elements of the existing

code to make it cleaner and more maintainable without altering the current functionality.

**Step 1: Examine RefactorEnemy’s Behavior**

Start by observing the RefactorEnemy's in-game behavior. Understanding its actions and

interactions will help you identify which parts of the code correspond to specific behaviors.

**Step 2: Understand the Code**

Navigate to RefactorEnemy.cs and thoroughly review the code. This will give you insight into

how the existing behaviors are implemented.

**Step 3: Evaluate and Refactor the Code**

Objective: Refactor the existing code to improve readability and flexibility without changing its

functionality.

**Instructions:**

1. Examine the Main Update Loop:

○ Simplify overly complex and long update loops by breaking out functionality into

methods.

○ Ensure that these new methods are concise and named descriptively to clarify

their purpose.

2. Simplify Nested If Statements:

○ Review nested if statements and consider alternatives to simplify the logic,

making it more apparent and easier to follow.

3. Review Scope and Access Modifiers:

○ Evaluate the scope and access modifiers for properties and methods. Ensure

that they are appropriately restricted to enhance encapsulation and security.

4. Move Patrol Behavior to a New Component:

○ Isolate the patrol behavior into a new component to decouple it from the main

class. This change will not only simplify RefactorEnemy.cs but also promote code

reuse and flexibility.

**Step 4: Verify the Refactored Code**

Once you have completed the refactoring:

● Ensure that the behavior of ***RefactorEnemy*** remains unchanged from its original

implementation.

● Test thoroughly in-game to verify that all functionalities are intact and no new issues

have been introduced.

Good luck with the refactor! This exercise will help you develop a keen eye for optimization and

maintainability in your code, critical skills for any proficient programmer. Happy coding!

**Solution:**

1. Refactor if statements to a method.

2. Check Struct usage and keep only editor specific fields there.

3. Extract components to a class that can be reusable like Patrol behavior, explode behavior and slip check.

4. Keep structs on top of the class for best practice.

5. Keep the ***update*** method readable – extract methods.

**Challenge 3:** **Extra Challenge** Making it Better

Create a new enemy based on the Refactored Enemy that adds another behaviour, instead of chasing the enemy, it will flee from the enemy when his health is at its max. Consider how to add this functionality in a legible, flexible way that works within the existing framework

**Solution:**

Update ChaseEnemy :

private void ChaseEnemy()

{

// Flee away from the player at max health.

float health = player.GetComponent<PlayerMovement>().playerStats.health;

Debug.Log($"health: {health}");

if (health >= 100)

{

Debug.Log("Fleeing");

// Calculate direction away from the player

Vector3 fleeDirection = (transform.position - player.transform.position).normalized;

// Move enemy away from the player

transform.position += fleeDirection \* enemyStats.chaseSpeed \* Time.deltaTime;

// Optionally, look in the direction of movement

if (fleeDirection != Vector3.zero)

transform.rotation = Quaternion.LookRotation(fleeDirection);

}

else // Chase the player

{

Debug.Log("Chasing");

sight.position = player.transform.position;

transform.LookAt(sight);

transform.position = Vector3.MoveTowards(transform.position, player.transform.position, Time.deltaTime \* enemyStats.chaseSpeed);

}

## 3.2 Course 2: Ski Game

### 3.2.1 Session 1: Scene setup

Follow ***Game Design Document***

ProBuilder

1. Create game design scene using prefab models.

Solution:

1. Create empty game objects for organizing:
   1. GENERAL: For Main camera and Lightings
   2. PLAYER: Player related objects
   3. LEVEL: Game objects like trees, ramps, slope etc.
2. Add a **RigidBody** component to the player.
   1. Freeze Rotation in X,Y and Z.
3. Add **Constant Force** component to the player
   1. Set Force: -20 in Z.
4. Add a following camera to the player
   1. Under PLAYER > R.click > Cinemachine > Targetted Camera > Follow Camera
   2. Rename to Player Camera
   3. Set Tracking Object > Player.
   4. Set Cinemachine Follow > Follow offset: 0, 5, 10
   5. Add **Cinemachine Brain** to Main Camera.
5. Add invisible walls
   1. Add empty game object: Invisible walls
   2. Add a new empty game object > Add **Box Collider** and adjust shape and size.
6. Add **Finish Line** game object
   1. Create a new game object, adjust scale and position.
   2. Add Box Collider and adjust its dimensions.
   3. Add **Sprite Renderer ,** set **sprite: square** and set its color.
7. Add background
   1. Window > Rendering > Lighting > set Lighting Settings Asset: Settings.lighting in Assets/Sprites.
   2. Goto Environment tab > set Skybox Material: Tundra in Assets/Skyboxes.
   3. Main Camera > Clear Flags: Skybox
   4. Directional Light > Adjust color > Generate

### 3.2.2 Session 2 : Scripting

Instructions:

1. The skier should move by changing the velocity of the attached Rigidbody component
2. The skier should always be moving in the direction they are facing
3. While touching the ground, the skier should be able to turn 90° in either direction
4. The skier should stop moving if they are turned at a full 90° angle
5. While in the air, the skier should not be able to turn
6. A simple animation controller should respond to the speed of the skier. Animations can be found in Animations folder.

Create PlayerController.cs

Update the unity:

1. Remove **Const Forces** component from Player, **PlayerController** will handle it now.
2. Use Ground Check to check if the player is on the ground.
   1. Create an empty gameObject under **Player >** Rename to “GroundCheck” > Reset values.
   2. Set y: -0.5 so that it sits just below the player.
   3. Use an icon for ground check.
   4. Set the gameobject in the player script.
   5. Set ground layers in unity set **layer** of ramps to **Ground.**
3. Add **Animator** component and set animator controller:
   1. Slow to fast.
   2. Fast to slow
   3. Add a parameter **playerSpeed = 1100**
   4. Set speed: 3
4. PlayerController.cs
   1. Add a struct **Stats:** speed, maxSpeed,minSpeed, turnSpeed, turnAcc,turnDeacc.
   2. Add public properties:
      1. playerStats
      2. KeyCode, A,D
      3. Bool isMoving
      4. Transform GroundCheck
      5. LayerMask groundLayers
      6. RigidBody rb
      7. Animator animator

Refer PlayerController.cs for implementation.