JUMP CITY 2D

**Project submitted to the APSSDC**

**Bachelor of Technology In**

**Computer Science and Engineering -IOT**

**ADITYA COLLEGE OF ENGINEERING AND TECHNOLOGY**

**Submitted By**

**KOTA ADBUTH KUMAR -23MH1A4928**

****

**Under the guidance of**

**Mr. C. Sathish Kumar**

**July 2025**

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **S.NO** | **CONTENTS** | **PAGE NO** |
| 1 | Abstract | 3 |
| 2 | Introduction | 4 |
| 3 | System Requirements | 5-6 |
| 4 | System Architecture | 7 |
| 5 | Implementation | 8-17 |
| 6 | Result / Output | 18-21 |
| 7 | Conclusion | 21-22 |
| 8 | References | 23 |

1. **ABSTRACT**

The project titled **"JUMP CITY 2D"** is a 2D entertainment-based game developed using **Unity 2018.4.0f1**. The core objective of the game is to guide a player character in avoiding virus particles that continuously move towards them. Each successful dodge increases the player's score, which is displayed at the top of the screen. The game provides a simple yet engaging gameplay experience that tests the player's reflexes and timing.

The game is compatible with both WebGL and Windows platforms, making it accessible through web browsers as well as desktop systems. The development process involved the use of C# scripting, which is well-suited for Unity due to its strong cross-platform capabilities and support through advanced IDEs like Visual Studio. Visual assets were sourced from platforms such as Google and Mixamo, which contributed to the game's visual appeal and character animation.

This project helped in understanding key Unity concepts such as 2D game physics, sprite handling, asset integration, UI elements like score display, and scripting game logic using C#. The final outcome is a lightweight, interactive game aimed at providing fun and engagement through a simple but meaningful concept—staying safe by avoiding threats.

# INTRODUCTION

**Problem Statement**

Game development often requires a powerful and flexible platform that can handle both 2D and 3D environments, support multiple platforms, and provide a user-friendly interface for developers. Many beginner-level developers face challenges in creating engaging and interactive games due to limited knowledge of game engines, asset management, and scripting.

This project aims to address these challenges by developing a 2D game titled **“JUMP CITY 2D”** using the Unity game engine. Unity offers a rich environment for developing cross-platform games with robust support for 2D physics, sprite management, animation, and scripting through C#. The goal is to create a simple, fun, and visually appealing game where the player avoids virus particles, and the score increases as they progress, promoting both engagement and replayability.

The main problem is to effectively use Unity’s tools and features to design a functional and optimized 2D game that runs smoothly on Windows **and** WebGL, while ensuring the game logic, UI, and assets are well-integrated and responsive.

**Objectives**

The primary objectives of this project are:

1. Develop an engaging 2D game using Unity
2. Implement gameplay where the player must dodge and kill incoming enemy
3. Increase the player's score with collect coins and when required coin are collected we

win the game

1. Script game logic and functionality in C# for smooth, real-time interaction
2. Ensure cross-platform compatibility on Windows and WebGL
3. Integrate visually appealing assets sourced from platforms like Google and Unity
4. Design a simple, user-friendly interface for easy accessibility
5. Optimize performance to ensure smooth gameplay across various devices and browsers

# SYSTEM REQUIREMENTS

To successfully develop, test, and run the Jump City 2D Game, specific hardware and software components are essential. The following outlines the full system requirements categorized under Operating System, Hardware, Software, and Asset Sources.

1. **Operating System Requirements**

|  |  |  |
| --- | --- | --- |
| **Platform** | **Minimum Requirement** | **Recommended** |
| **Windows** | Windows 10 (64-bit) | Windows 11 (64-bit) |
| **macOS** | macOS 10.15 (Catalina) or later | macOS Ventura or newer |

1. **Hardware Requirements**

|  |  |  |
| --- | --- | --- |
| **Component** | **Minimum Specification** | **Recommended Specification** |
| **Processor** | Intel Core i3 / AMD Ryzen 3 | Intel Core i5 / AMD Ryzen 5 or higher |
| **RAM** | 4 GB RAM | 8 GB or more |
| **Graphics** | Integrated Graphics (Intel UHD, Vega 8) | Dedicated GPU (NVIDIA GTX 1050 / AMD RX 560) |
| **Storage** | 1 GB free space for Unity & project files | SSD with 5 GB free space for assets & builds |
| **Display** | 1024×768 screen resolution | 1920×1080 Full HD or higher |

1. **Software Requirements**
   1. **Game Engine & Development Tools**

|  |  |
| --- | --- |
| **Software** | **Description** |
| **Unity Editor** | Unity 2018.4.0F1 LTS (or newer) |
| **Scripting Language** | C# with Mono Behaviour scripts for game logic |
| **IDE** | Visual Studio 2022 / Visual Studio Code |
| **Unity Modules** | WebGL Build Support, Windows Build Support |

* 1. **Browser Requirements for WebGL Builds**

|  |  |
| --- | --- |
| **Browser** | **Supported** |
| Google Chrome | Yes |
| Mozilla Firefox | Yes |
| Microsoft Edge | Yes |
| Safari (macOS) | Yes |

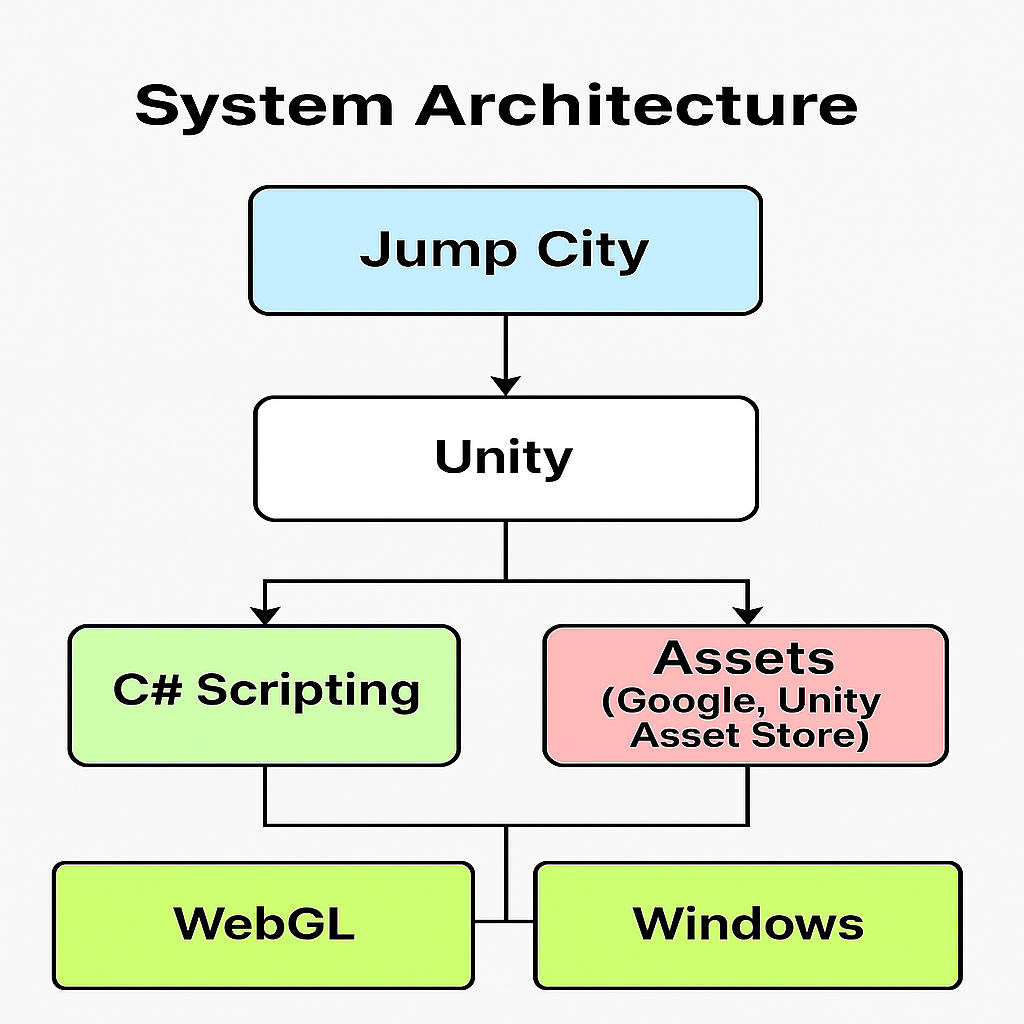
1. **Asset Sources**

To develop immersive and animated 2D content, external sources were used for character models, animations, and game environment objects.

|  |  |
| --- | --- |
| **Source** | **Description / Usage** |
| **Unity Asset Store** | Source of free and paid assets including terrain, UI icons, and particle effects. |

All external assets were either open-source, free for personal/educational use, or appropriately credited as per licensing terms.

# SYSTEM ARCHITECTURE

The system design of the JUMP CITY 2D Game follows a modular and component-based architecture, leveraging Unity’s built-in Game Object and Component system. This architecture allows for efficient updates, scalability, and ease of debugging.

# IMPLEMENTATION

The Jump City 2D Game was implemented using Unity and C# scripting. This section describes how various systems of the game were developed, including movement mechanics, obstacle generation, collision detection, scoring, and UI.

* 1. **Player Controller**

The player character moves forward automatically. The user can press up arrow to move up, and down arrow to move on. Rigid body physics is used for stopping gravity and freezing rotation.

Code   
using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI; // Ensure you have this namespace for UI elements

using UnityEngine.SceneManagement; // For scene management

using TMPro; // For TextMeshPro support

public class Player : MonoBehaviour

{

public int maxHealth = 3; // Maximum health of the player

public Animator Animation; // Reference to the Animation component for player animations

public float speed = 5.0f; // Speed of the player movement

public float jumpForce = 5.0f; // Force applied when the player jumps

private Rigidbody2D rb; // Reference to the Rigidbody2D component

public float move; // Variable to store horizontal input

private Vector2 originalScale;

public Vector2 repositions;// Position to reset the player to when they die

public Repositiondelay rd; // Delay before repositioning the player

public TextMeshProUGUI Healthtext; // Reference to the TextMeshProUGUI component for displaying health

public float attackRange = 0.5f; // Range within which the player can attack

public LayerMask enemyLayer; // Layer mask to identify enemies

public Transform attackPoint; // Point from which the player attacks

void Start()

{

rb = GetComponent<Rigidbody2D>();// Get the Rigidbody2D component attached to the player

rb.position = transform.position;

originalScale= transform.localScale; // Store the original scale of the player

repositions=transform.position; // Store the initial position of the player for respawningd

rd= FindObjectOfType<Repositiondelay>(); // Find the Repositiondelay script in the scene

}

// Update is called once per frame

void Update()

{

Healthtext.text = "Health: " + maxHealth; // Update the health text display

if (maxHealth <= 0) // If the player's health is zero or less

{

Die(); // Call the Die method

return; // Exit the Update method to prevent further actions

}

float currentSpeed = Input.GetKey(KeyCode.LeftShift) || Input.GetKey(KeyCode.RightShift) ? speed \* 2f : speed;

move = Input.GetAxis("Horizontal");

rb.velocity = new Vector2(move \* currentSpeed, rb.velocity.y);

// Flip the player

if (move > 0)

transform.localScale = new Vector2(originalScale.x, originalScale.y); // right

else if (move < 0)

transform.localScale = new Vector2(-originalScale.x, originalScale.y); // left

if (Input.GetButtonDown("Jump") && Mathf.Abs(rb.velocity.y) < 0.001f)

{

rb.AddForce(new Vector2(0, jumpForce), ForceMode2D.Impulse);

}

if(Mathf.Abs(move) > 0f)

{

Animation.SetFloat("Run", 1f);

}

else

{

Animation.SetFloat("Run", 0f);

}

if (Input.GetMouseButtonDown(0)) // Check if the attack button is pressed (usually left mouse button or Ctrl key)

{

Animation.SetTrigger("Attack"); // Trigger the attack animation

}

}

private void OnTriggerEnter2D(Collider2D collision)

{

if (collision.gameObject.tag == "falldelector")

{

rd.Delay(); // Call the Delay method from Repositiondelay script

SceneManager.LoadScene("GameOver"); // Load the lose scene when the player falls

}

}

public void TakeDamage(int damage)

{

if (maxHealth <= 0) return; // If the player's health is already zero or less, do nothing

maxHealth -= damage; // Reduce the player's health by the damage amount

}

public void Attack()

{

Collider2D callinfo = Physics2D.OverlapCircle(attackPoint.position, attackRange, enemyLayer); // Check for enemies within the attack range

if(callinfo)

{

if(callinfo.GetComponent<enemy>() != null) // Check if the collided object has an Enemy component

{

callinfo.GetComponent<enemy>().TakeDamage(1); // Call the TakeDamage method on the enemy with a damage value of 1

}

}

}

private void OnDrawGizmosSelected()

{

if (attackPoint == null) return; // If attackPoint is not assigned, do nothing

Gizmos.color = Color.red; // Set the color for the Gizmos

Gizmos.DrawWireSphere(attackPoint.position, attackRange); // Draw a wire sphere to visualize the attack range

}

void Die()

{

SceneManager.LoadScene("GameOver"); // Load the GameOver scene when the player dies

}

}

* 1. **Enemy Spawner**

Enemy are spawned dynamically ahead of the player using Unity's Instantiate method.

Enemy petrol the area when player are with in the range then enemy attack the player.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class enemy : MonoBehaviour

{

public float health = 2f; // Health of the enemy

public Animator animator; // Reference to the Animator component

public bool facingRight = true; // Direction the enemy is facing

public float speed = 2f; // Speed of the enemy movement

public Transform checkpoint; // Reference to the checkpoint transform

public float distance = 0.5f; // Distance to the checkpoint

public LayerMask groundLayer; // Layer mask for the ground layer

public bool inRange = false; // Flag to check if the enemy is in range

public Transform player; // Reference to the player transform

public float attackRange = 3f; // Range within which the enemy can attack

public float reteverdistance = 2f; // Distance to return to the checkpoint

public float chaseSpeed = 3f; // Speed when chasing the player

public Transform attackpoint; // Point at which the enemy attacks

public float attackradius = 0.5f; // Radius of the attack area

public LayerMask attackLayer; // Layer mask for the attack area

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

if (health <= 0)

{

Die(); // Call the Die method if health is zero or less

return; // Exit the Update method to prevent further actions

}

if (Vector2.Distance(transform.position, player.position) <= attackRange)

{

inRange = true;

}

else

{

inRange = false;

}

if (inRange)

{

if(player.position.x > transform.position.x && !facingRight)

{

transform.eulerAngles = new Vector3(0, 0, 0); // Flip the enemy to face right

facingRight = true; // Update the facing direction

}

else if (player.position.x < transform.position.x && facingRight)

{

transform.eulerAngles = new Vector3(0, -180, 0); // Flip the enemy to face left

facingRight = false; // Update the facing direction

}

if (Vector2.Distance(transform.position, player.position) > reteverdistance)

{

animator.SetBool("Attack1", false); // Set the attack animation to true

transform.position = Vector2.MoveTowards(transform.position, player.position, chaseSpeed \* Time.deltaTime);

}

else

{

animator.SetBool("Attack1", true); // Set the attack animation to true

}

}

else

{

transform.Translate(Vector3.right \* Time.deltaTime \* speed);

RaycastHit2D hit = Physics2D.Raycast(checkpoint.position, Vector2.down, distance, groundLayer);

if (hit == false && facingRight)

{

transform.eulerAngles = new Vector3(0, -180, 0); // Flip the enemy to face left

facingRight = false; // Update the facing direction

}

else if (hit == false && facingRight == false)

{

transform.eulerAngles = new Vector3(0, 0, 0); // Flip the enemy to face right

facingRight = true; // Update the facing direction

}

}

}

public void Attack()

{

Collider2D Callinfo = Physics2D.OverlapCircle(attackpoint.position, attackradius, attackLayer); // Check for enemies in the attack area

if(Callinfo)

{

if(Callinfo.gameObject.GetComponent<Player>() != null) // Check if the collided object has a player component

{

Callinfo.gameObject.GetComponent<Player>().TakeDamage(1); // Call the TakeDamage method on the player component

}

}

}

public void TakeDamage(int damage)

{

if(health <= 0)

{

return; // If health is already zero or less, do nothing

}

health -= damage; // Reduce the enemy's health by the damage amount

}

private void OnDrawGizmosSelected()

{

if (checkpoint == null)

{

return;

}

Gizmos.DrawRay(checkpoint.position, Vector2.down \* distance);

Gizmos.color = Color.red; // Set the color of the gizmo to red

Gizmos.DrawWireSphere(transform.position, attackRange);

Gizmos.color = Color.green; // Set the color of the gizmo to green

if (attackpoint == null)

{

return;

}

// Draw a wire sphere at the attack point

Gizmos.DrawWireSphere(attackpoint.position, attackradius); // Draw a wire sphere at the attack point

}

void Die()

{

//animator.SetTrigger("Die"); // Trigger the die animation

Destroy(gameObject, 1f); // Destroy the enemy game object after 1 second

//Debug.Log("Enemy died!"); // Log a message when the enemy dies

}

}

* 1. **Camera Movement and Background**

Camera movement and Background scripts are used for camera and background are follows the Player movement in x directions.

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class camfollow : MonoBehaviour

{

public GameObject player; // Reference to the player GameObject

private Vector3 playervalue;

public float offset; // Offset from the player

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

playervalue= new Vector3(player.transform.position.x,transform.position.y,transform.position.z);

if(player.transform.localScale.x>0f)

{

playervalue= new Vector3(playervalue.x + offset, playervalue.y, playervalue.z);

}

else

{

playervalue= new Vector3(playervalue.x - offset, playervalue.y, playervalue.z);

}

transform.position = playervalue;

}

}

* 1. **Score Manager**

The score increases when Player collect the coins and coin rotates. The score is displayed using Unity’s UI Text or TextMeshPro.

**Code**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class coinrotator : MonoBehaviour

{

public float rotationSpeed = 100f;

public Repositiondelay rd; // Reference to the repositiondelay script

// Start is called before the first frame update

void Start()

{

rd = FindObjectOfType<Repositiondelay>(); // Find the repositiondelay script in the scene

}

// Update is called once per frame

void Update()

{

this.transform.Rotate(Vector3.up, rotationSpeed \* Time.deltaTime);

}

private void OnTriggerEnter2D(Collider2D other)

{

if(other.CompareTag("Player"))

{

rd.AddScore();// Call the AddScore method from Repositiondelay script

Destroy(gameObject); // Destroy the coin when collected

}

}

}

* 1. **UI Implementation (Buttons)**

The UI includes:(Game Scene)

* + - **Start Menu**
    - **Exit**
    - **Mute**

The UI includes: (Menu Scene)

* **Start**
* **Exit**

The UI include: (GameOver Scene)

* **Restart Button**

UI elements were created using Canvas and scripted to activate based on game state

**Code**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI; // Ensure you have this namespace for UI elements

using UnityEngine.SceneManagement; // For scene management

using TMPro; // For TextMeshPro if you are using it for UI text

public class Buttons : MonoBehaviour

{

// Start is called before the first frame update

void Start()

{

}

// Update is called once per frame

void Update()

{

}

public void StartGame()

{

SceneManager.LoadScene("Level"); // Replace "GameScene" with the name of your game scene

}

public void QuitGame()

{

// Quit the application

Application.Quit();

}

public void MenuGame()

{

SceneManager.LoadScene("Menu"); // Replace "Menu" with the name of your menu scene

}

public void MuteMusic()

{

AudioListener.volume = 0; // Mute the audio

}

}

* 1. **Background Music**

Background music is a audio source for the game which is used in Unity.

Code

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class SoundManager : MonoBehaviour

{

public AudioSource audioSource; // Reference to the AudioSource component

public static AudioClip jumpsound,attacksound,GameOversound; // Array to hold multiple audio clips

// Start is called before the first frame update

void Start()

{

audioSource = GetComponent<AudioSource>();

jumpsound = Resources.Load<AudioClip>("jump"); // Load the jump sound clip

attacksound = Resources.Load<AudioClip>("attack"); // Load the attack sound clip

GameOversound = Resources.Load<AudioClip>("GameOver"); // Load the game over sound clip

}

// Update is called once per frame

void Update()

{

}

public static void PlaySound(string clip)

{

switch (clip)

{

case "jump":

AudioSource.PlayClipAtPoint(jumpsound, Camera.main.transform.position); // Play the jump sound

break;

case "attack":

AudioSource.PlayClipAtPoint(attacksound, Camera.main.transform.position); // Play the attack sound

break;

case "GameOver":

AudioSource.PlayClipAtPoint(GameOversound, Camera.main.transform.position); // Play the game over sound

break;

default:

Debug.LogWarning("Sound clip not found: " + clip); // Log a warning if the clip is not found

break;

}

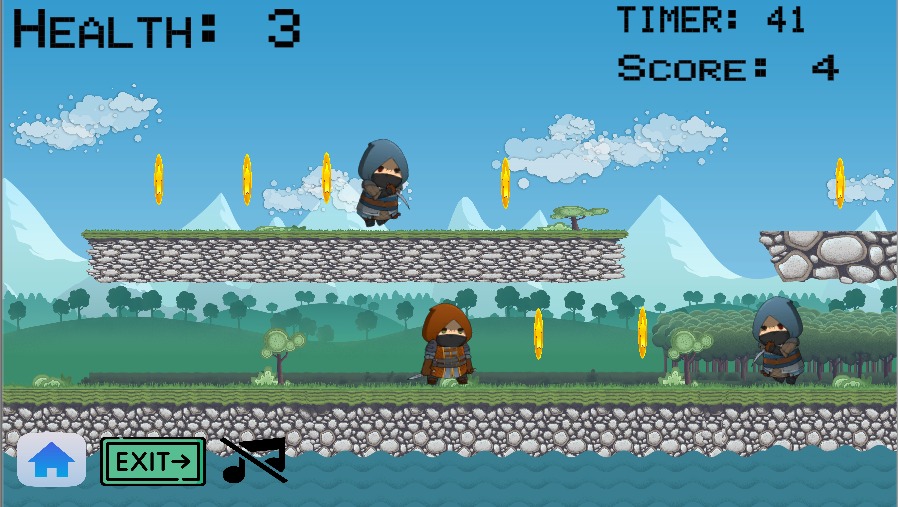
}

}

* 1. **Screenshots**

****

**MAIN MENU SCREEN**



**GAMEPLAY**

****

**GAMEOVER**

1. **RESULT / OUTPUT**

The Jump City 2D Game was successfully designed and implemented using Unity and C#. The game functions as expected and meets all the core objectives of a standard player movement and collisions : continuous gameplay, real-time obstacle generation, and score tracking.

* 1. **Functional Results**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Status** | **Description** |
| **Player Movement** | Completed | Player moves up and down using RightArrow and LeftArrow and Rotation in Z is freezed. |
| **Enemies Spawning** | Completed | Enemies spawn at regular intervals to challenge the player. |
| **Collision Detection** | Completed | Player collides with Enemy and when Player Health is zero it triggers game over or restart. |
| **Score System** | Completed | Score updates dynamically on over time by crossing Player collect the coins. |
| **UI System** | Completed | Score, Timer, health and Menu, Exit, Mute button implemented with Unity UI. |
| **Game State Management** | Completed | Handles game start, restart, and game over states cleanly. |
| **Performance** | Smooth | Runs without lag on systems meeting minimum requirements. |

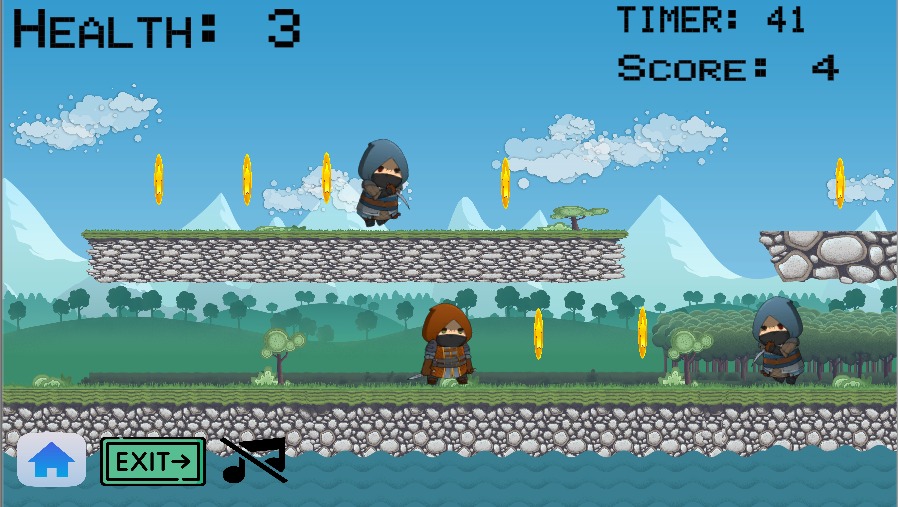
* 1. **User Interface Output**
     + **Start Screen** – Displays game title and “Start” button.
     + **Gameplay Screen** – Shows the live score, Timer, Health at the top and Menu, Exit, Mute Button at the Bottom.
     + **Game Over Screen** – Displays “Restart” button and “Exit” button upon collision.
  2. **Sample Screenshots**

1. **Main Menu**

****

Start screen with "Play" and "Exit" button

1. **Gameplay Screen**

****

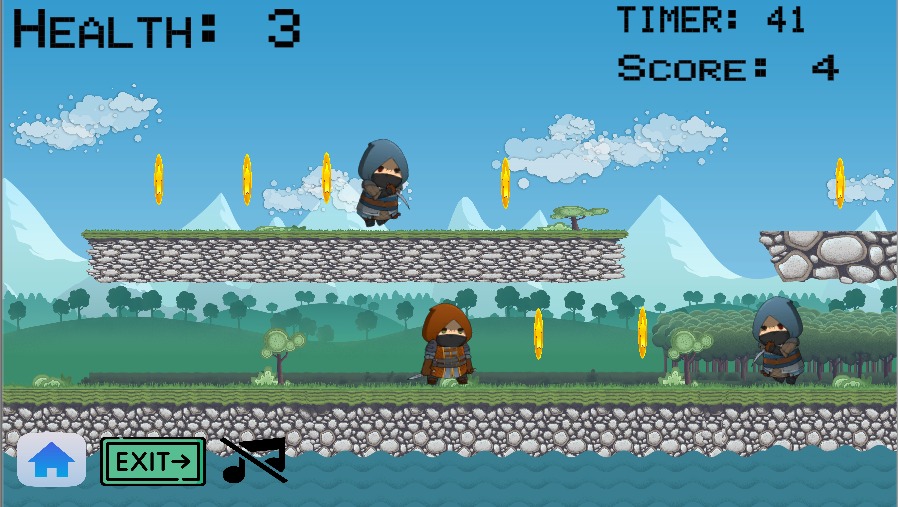
Player moving forward and escaping and kill the Enemies and collect the coins

1. **Collision Event**

****

Player Health and Timer reached “0”— Game-over Scenes appears.

1. **Score Display**

****

Real-time score shown at the top of the screen with increasing values.

* 1. **Performance Testing**

|  |  |
| --- | --- |
| **Test Environment** | **Result** |
| Unity Editor (Play Mode) | No lag |
| Windows Build (.exe) | Stable FPS |
| WebGL Export (Browser) | Smooth performance on Chrome/Edge |

**Note:** Optimized prefabs and limited simultaneous spawns were used to maintain frame rate.

# CONCLUSION

The Jump City 2D Game successfully demonstrates the core concepts of 2D game development using Unity. This project integrates multiple components including player movement, physics- based interactions, real-time obstacle and, scoring, and user interface design. It serves as a strong foundation for understanding Unity’s component-based architecture and the practical use of C# scripting in a real-time interactive environment.

Key outcomes of the project include:

* Functional background with player gameplay.
* Smooth movements control (Right and Left).
* Efficient spawning of Enemies.
* Real-time score tracking and responsive UI system.

By developing the 2D game **"Jump City 2D"**, I gained a practical understanding of various core concepts in the Unity game engine. I learned how to set up a 2D game environment, manage scenes, and handle assets such as sprites, backgrounds, and animations. Through the use of **C#** scripting, I implemented player controls, movement, collision detection, and

scoring logic, which helped me understand how scripting integrates with Unity’s component- based system.

I explored **Rigid body 2D** and **Box Collider 2D** components to apply physics and detect interactions between the player and Enemies. I also worked with the Canvas System to create a user interface that displays the player’s score in real-time during gameplay.

Unity's **Inspector panel**, **Hierarchy**, and **Scene view** helped me visually arrange and fine- tune game objects, while the **Animator and Animation** tools allowed me to create smooth transitions and effects. Additionally, I understood how Unity handles cross-platform builds, and I successfully exported my game for both **WebGL** and **Windows** platforms.

Overall, this project gave me hands-on experience in Unity's 2D game development workflow—from asset integration and scripting to building and deploying the final game.

**Future Scope**

While the current version of the game meets its primary objectives, there are several opportunities for enhancement:

**Gameplay Improvements**

* + Adding More Attack functions to destroy Enemies.
  + Introduce difficulty scaling over time (faster obstacles, complex patterns).
  + Implement multiple lanes or tracks with lane switching.

**Player Features**

* + Add different player characters with different effects.
  + Giving limited lifelines.

**Environment & Graphics**

* + Include different background themes (ex: Colors and effects).
  + Adding specific effects while the score increases and Player dies.

**Platform Publishing**

* + Optimize for Android/iOS deployment.
  + Add touch controls for mobile compatibility.

**Analytics & Leaderboard**

* + Integrate score leaderboard using online services (Facebook).
  + Add player analytics to track performance.

MY GAME URL: <https://kotaadbuthkumar.itch.io/jump-city>

# REFERENCES

The following resources, tools, and platforms were used during the development of the Jump City 2D Game. These sources provided essential assets, documentation, and development support throughout the project lifecycle.

**Technical Resources**

* **Unity Documentation** – [https://docs.unity3d.com](https://docs.unity3d.com/)

Official documentation for Unity engine, scripting, components, and build settings.

* **C# Scripting in Unity** – [https://learn.unity.com](https://learn.unity.com/)

Unity Learn platform for tutorials on scripting and gameplay mechanics.

**Asset Resources**

* **Unity Asset Store** – [https://assetstore.unity.com](https://assetstore.unity.com/)

Used to download 2D models, UI elements, particle effects, and sample background music.

* **OpenGameArt.org** – [https://opengameart.org](https://opengameart.org/)

Free and open 2D/3D assets including props, obstacles, textures, and audio.