**2D GAME DEVELOPMENT**

A mini project report submitted in the partial fulfillment for the award of degree of

**BACHELOR OF COMPUTER APPLICATIONS**

By

**YOGESH.M -** 212200826

**KARTHIKEYAN.N -** 212200788

**HARISH.K** - 212200784

Under the guidance of

**Mrs. S. Cynthia Juliet., M.C.A., M.Phil.,**

**Assistant Professor**



**JAYA COLLEGE OF ARTS AND SCIENCE**

**THIRUNINRAVUR– 602024.**

**APRIL – 2025.**

**BONAFIDE CERTIFICATE**

This to certify that the report entitled

**2D GAME DEVELOPMENT**

being submitted to the University of Madras, Chennai.

**By**

**YOGESH.M** - 212200826

**KARTHIKEYAN.N** - 212200788

**HARISH.K** - 212200784

in the partial fulfillment for the award of degree

of

**BACHELOR OF COMPUTER APPLICATIONS**

is a bonafide record work carried out by them

under the guidance and supervision.

Date:

**Mrs. S. Cynthia Juliet., M.C.A., M.Phil.,**

Assistant Professor,

Department of Computer Applications,

Jaya College of Arts and Science, Thiruninravur-602024.

Submitted for the viva-voice examination held on

**Internal Examiner External Examiner**

**DECLARATION**

This is to certify that the project entitled “**2D GAME DEVELOPMENT”** submitted to the University of Madras in partial fulfillment of the requirements for the degree of **BACHELOR OF COMPUTER APPLICATION** is a record of original mini project work done by me, under the guidance and supervision of **Mrs. S. Cynthia Juliet M.C.A., M.Phil.,** Assistant Professor**,** Department of Computer Applications, Jaya College of Arts & Science, Thiruninravur – 602024, and it has not form the basis for award of degree nor similar title to any candidate of any university.

**YOGESH.M** - 212200826

**KARTHIKEYAN.N** - 212200788

**HARISH.K** -212200784

BACHELOR OF COMPUTER APPLICATIONS,

JAYA COLLEGE OF ARTS AND SCIENCE, THIRUNINRAVUR – 602024.

**ACKNOWLEDGEMENT**

The Project work cannot be a one-man show. Although it is impossible to give individual thanks to all helpful faculty members and to those in connection. It takes this opportunity to express my gratitude for them.

As a start, my heartfelt thanks to God for giving his blessing and confidence to make this project a successful one.

I feel grateful to thank **Prof. A. KANAGARAJ., M.A., M.Phil.,** Founder and Chairman, Jaya Educational Trust, **Mrs. K. VIJAYAKUMARI., M.A., B.Ed.,** Secretary, Jaya Educational Trust, and **Er. K. NAVARAJ., M.E.,** Vice Chairman, Jaya Educational Trust for extending their support during course of my studies.

I am grateful to **Dr. P. GUHAN., M.C.A., M.Phil., Ph.D.,** Principal, Jaya College of Arts and Science, Thiruninravur – 602 024, for providing me with an environment to complete my project successfully.

I take pleasure in thanking **Mrs. S. Cynthia Juliet., M.C.A., M.Phil.,** Head of the Department**,** Department of Computer Applications, Jaya College of Arts and Science, Thiruninravur, for extending his support throughout the mini project.

My deepest thanks to **Mrs. S. Cynthia Juliet., M.C.A., M.Phil.,** Assistant Professor**,** Jaya College of Arts and Science, Thiruninravur, for her guidance and encouragement at each stage of my project.

I would also thank my Institution and faculty members without whom this project would have been a reality.

By

**YOGESH.M** - 212200826

**KARTHIKEYAN.N** -212200788

**HARISH.K** – 212200784

**TABLE OF CONTENTS**

**Page No**

**CHAPTER 1: ABSTRACT 01**

**1.1** Existing System

**1.2** Proposed System

**CHAPTER 2: SYSTEM REQUIREMENT & ANALYSIS 02**

**2.1** Problem Definition

**2.2** Requirement Specification

**2.3** Software & Hardware Requirements

**2.3.1** Hardware Requirements

**2.3.2** Software Requirements

**CHAPTER 3: SYSTEM DESIGNING 07**

**3.1** Data Flow Diagram

**3.2** Table Design

**CHAPTER 4: SYSTEM IMPLEMENTATION 12**

**4.1** Feasibility Test

**4.2** Coding Details

**4.3** Screen Shot

**CHAPTER 5: SYSTEM TESTING 39**

**5.1** Testing Procedure

**5.2** System Testing

**CHAPTER 6: PROJECT EVALUATION 45**

**6.1** Salient Feature

**6.2** Limitation of the System

**6.3** Future Scope of the Project

**CHAPTER 7: BIBLIOGRAPHY 47**

**CHAPTER - 1 ABSTRACT**

**OBJECTIVE:**

The main objective of this project is to be creating a 2D game in Unity, likely titled "MaskMan." The document outlines a structured approach, including setting up the Unity project, designing levels using tilemaps, implementing player movement and animations, adding UI elements, setting up collectibles and scoring, and integrating features like a parallax background, camera control, and death/respawn mechanics. It also includes future enhancements like sound effects, power-ups, additional levels, and a pause menu.

**1.1 EXISTING SYSTEM:**

The existing system consists of traditional platformer games with predefined levels, standard enemy mechanics, and basic 2D physics. These games often rely on basic graphics, predefined player interactions, and limited user engagement options. Platformer games have been around for decades, with classics such as Super Mario Bros, Sonic the Hedgehog, and Donkey Kong setting the foundation. However, these early systems had limitations in character movement, level design flexibility, and user interface customization. Modern game engines, such as Unity, provide developers with more advanced tools to enhance game physics, animations, and user experience.

**1.2. PROPOSED SYSTEM:**

The proposed system enhances the traditional 2D platformer experience by integrating modern features such as level selection and interactive elements while maintaining a nostalgic 2D platforming feel. The 2D MaskMan platformer game, developed using Unity and C#, will feature intuitive character controls, dynamic level progression, an engaging graphical interface, and interactive traps instead of enemies. The focus is on creating a challenging yet enjoyable experience where players must navigate through various obstacles and traps. The goal is to provide players with a smooth, responsive gameplay experience while preserving the essence of classic 2D platformers.

**CHAPTER - 2 SYSTEM REQUIREMENT & ANALYSIS**

**2.1 PROBLEM DEFINITION:**

Traditional 2D platformers emphasize enemy interactions, often restricting creative gameplay mechanics. This project introduces a trap-based challenge system where players rely on timing, reflexes, and logic instead of combat. The game encourages strategic movement and problem-solving to navigate obstacles. Each level progressively increases in difficulty, testing player adaptability. Smooth mechanics and interactive UI enhance the user experience. This approach offers a fresh take on platformer gameplay, making it engaging and skill-based.

2.1.1 SYSTEMS ANALYSIS**:**

The objective of the system analysis activity is to develop structured system specification for the proposed system. The structured system specification should describe what the proposed system would do; independent of the technology, which will be used to implement these requirements. The structured system specification will be used to implement these requirements. The structured system specification will be called the essential model (also known as logical model).

The essential model may itself consist of multiple models, modeling different aspect of the system. The data flow diagrams may model the data and their relationships and the state transition diagram may model time dependent behavior of the system. The essential model thus consists of the following.

Context diagram

Leveled data flow diagrams

Process specification for elementary bubbles

Data dictionary for the flow and stores on the DFDs.

**2.2 REQUIREMENT SPECIFICATION OF PROJECT:**

2.2.1 FRONT END DETAILS:

Front End tool is used In Unity, C# handles player controls, UI interactions, animations, and physics to create smooth gameplay. The Canvas system is used for menus and HUD, while Animator & scripts manage character animations. Cinemachine, parallax effects, and colliders enhance visuals and interactions for an engaging 2D platformer experience.

2.2.2 ABOUT UNITY**:**

Unity: Unity's frontend development focuses on creating visually appealing and interactive experiences for players. It handles game mechanics, animations, UI, and physics using C# scripts within the Unity engine. The player movement, jumping, and interactions with the environment are controlled through scripts attached to GameObjects, utilizing components like Rigidbody2D, Colliders, and Animator Controllers. Unity's Canvas system allows developers to design menus, HUDs, and score displays using elements like TextMeshPro, Buttons, and Sliders. Through event-driven programming, UI elements respond to player actions, such as starting a game, adjusting settings, or exiting. The Animator system ensures smooth character transitions between states like idle, running, and jumping, enhancing the overall fluidity of the game. Additionally, Unity's particle system adds visual effects such as explosions, fire, or dust to create a more immersive experience.

Beyond character control and UI, Unity also enhances visuals and gameplay flow using tools like Cinemachine and Parallax Scrolling. Cinemachine enables smooth and dynamic camera movements, automatically adjusting focus based on player actions. The parallax effect adds depth by making background layers move at different speeds, improving the visual appeal of 2D games. Unity’s physics system helps handle interactions between the player and objects, ensuring realistic behaviors such as collision detection, gravity effects, and environmental responses. The frontend also integrates sound and music, allowing background audio and sound effects to play at appropriate moments using AudioSource and AudioManager. Overall, Unity's frontend combines scripting, UI elements, animations, and physics to create an engaging and interactive gameplay experience.

.

2.2.3 WHY UNITY?

* + - User-Friendly Interface – Easy-to-use editor with drag-and-drop features.
    - C# Scripting – Powerful and flexible for gameplay mechanics.
    - Multiplatform Support – Build games for PC, mobile, consoles, VR, and AR.
    - Built-in Tools – Physics engine, animations, UI, and particle effects.
    - Cinemachine & AI – Smooth camera control and intelligent behaviors.
    - Asset Store – Access free and paid assets to speed up development.
    - Strong Community – Extensive tutorials, forums, and support resources.  Scalability – Suitable for indie developers and AAA studios.

2.2.4 BACK-END DETAILS**:**

In backend development, C# is commonly used for server-side logic, database management, and API development, especially with ASP.NET Core. The backend handles data processing, authentication, game logic, and communication between the client (frontend) and the server.

2.2.5 WHY C#?

C# is a powerful and versatile programming language widely used in game development, web applications, and enterprise software. It is the primary language for Unity, making it an essential choice for game developers due to its object-oriented structure, strong typing, and built-in support for game mechanics. C# provides high performance while maintaining ease of use, allowing developers to write clean, maintainable, and scalable code. Its rich standard library simplifies tasks such as file handling, networking, and UI development, making it suitable for both frontend and backend applications. Additionally, C# integrates seamlessly with .NET and ASP.NET Core, making it a preferred choice for backend development, where it handles server-side logic, authentication, and database management.

Beyond gaming, C# is widely used in enterprise applications, cloud computing, and software development due to its robust framework support and cross-platform capabilities. With the .NET Core and .NET 6+ ecosystem, C# applications can run on Windows, macOS, and Linux, increasing their flexibility. In game development, C# provides a structured way to manage game objects, physics, animations, and user inputs efficiently. It also supports multiplayer networking, enabling developers to create online games using frameworks

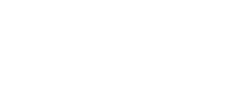
like Mirror, Photon, and Unity Netcode. The language’s garbage collection and memory management help optimize performance, reducing the risk of memory leaks. With strong community support, regular updates, and an extensive set of tools, C# remains a top choice for developers building everything from indie games to enterprise solutions.

**2.3 HARDWARE & SOFTWARE REQUIRMENTS:**

At Developer Side: Requires a high-performance PC with an Intel i7/Ryzen 7, 16GB+ RAM, and an SSD for fast development. Uses Unity, C# (Visual Studio), Photoshop/GIMP for UI, and Git for version control. Additional tools include Blender for 3D assets and Firebase/MySQL for online features.

At System Users Side: Needs a mid-range PC with an Intel i5/Ryzen 5, 8GB RAM, and GTX 1050+ for smooth gameplay. Requires Windows/macOS/Linux, DirectX 11+, and .NET Framework (if needed). The game runs on Unity Engine, ensuring optimized performance across different hardware.

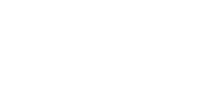
2.3.1 REQUIREMENT ANALYSIS STEPS:



Developmen

t

model

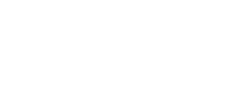


Context

A

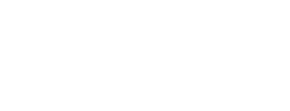
nalysi

s



Requirement

Specification



Finaliz

ing

The

Requirement

s

CONTEXT ANALYSIS

Shows how the game interacts with players, databases (if saving progress), input devices (keyboard/controller), and external APIs (if multiplayer or online features exist)

DEVELOPMENT MODEL

Represents the game development lifecycle (Agile model recommended) with stages like Concept Design → Prototyping → Development → Testing → Deployment.

REQUIREMENT SPECIFICATION

Lists functional requirements (character movement, UI interactions, trap mechanics) and non-functional

requirements (smooth performance, cross-platform support.

FINALIZING THE REQUIREMENTS

A structured chart displaying approved requirements, ensuring all gameplay mechanics, UI features, and

system optimizations are aligned before development starts.

2.3.2 SOFTWARE REQUIREMENT**:**

* + - * Operating System: Windows based system
      * Front end: C#, Unity
      * Back end: C#
      * Unity Engine: The primary game development engine used to design levels, implement physics, and manage assets.
      * C# Programming Language: Used for scripting game mechanics, handling user inputs, and managing interactions.
      * Visual Studio Code: The IDE for writing and debugging C# scripts.
      * Photoshop: Used for designing backgrounds, UI elements, and character sprites

2.3.3 HARDWARE REQUIREMENT:

* + - Processor: Intel Core i5 or higher
    - RAM: 8GB minimum
    - Graphics: NVIDIA GTX 1050 or equivalent
    - Storage: 2GB free space

**CHAPTER - 3 SYSTEM DESIGNING**

**3.1 DATA FLOW DIAGRAM:**

A directed arc or an arrow is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes, or between an external entity and a process, in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names.

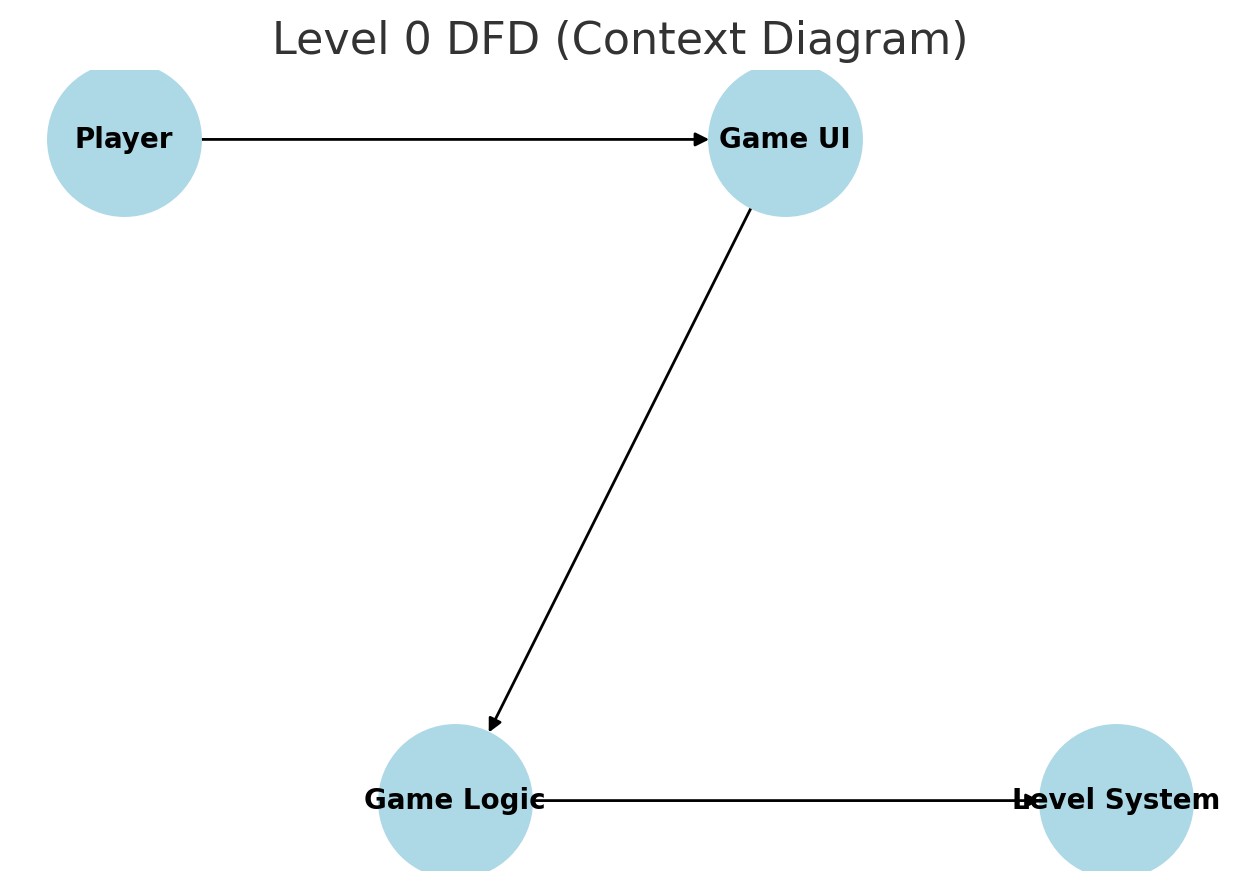
3.1.1 DATA STORE**:**

A data store represents a logical file. It is represented using two parallel lines. A logical file can represent either a data store symbol, which can represent either a data structure, or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire data of the data store and hence connecting to a data store need not be annotated with the name of the corresponding data items.

3.1.2 OUTPUT SYMBOL:

The output symbol is used when a hard copy is produced and the user of the copies cannot be clearly specified or there are several users of the output.

3.1.3 FLOWCHART:



*FIGURE 3.1.3.1 DFD (Context Diagram)*

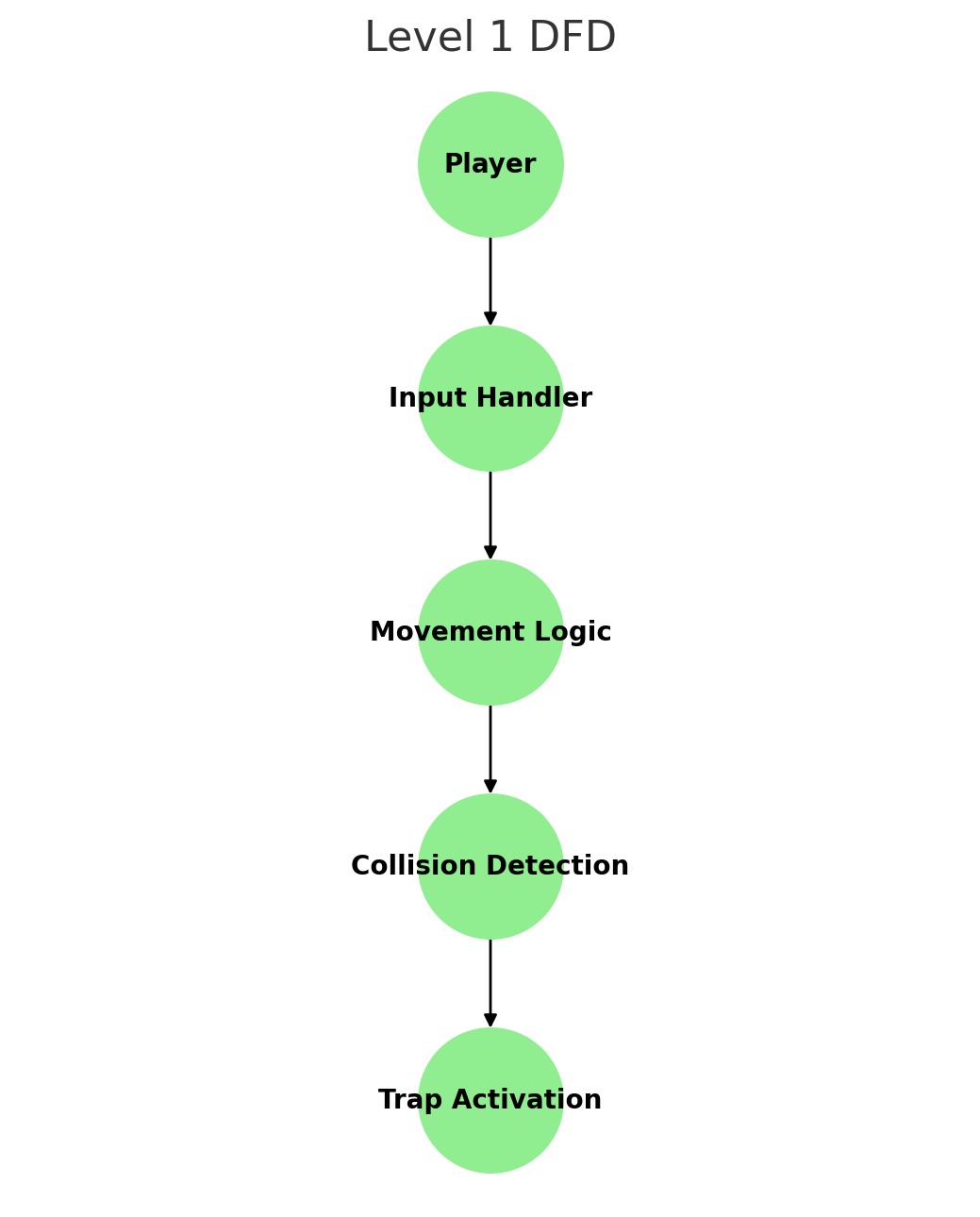


FIGURE 3.1.3.2 Level 1

**3.2 TABLE DESIGN**:

Player Table

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| **Player\_Id** | INT | Unique identifier for the player |
| **Name** | STRING | Player’s name |
| **Score** | INT | Current Score |
| **Level** | INT | Current level |

TABLE 3.2.1 TABLES FOR PLAYER

Game Progress Table

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| **Progress\_ID** | INT | Unique identifier for progress |
| **Player\_ID** | INT | Foreign key from Player Table |
| **Level** | INT | Last completed level |
| **Time\_Spent** | FLOAT | Time spent on the level |

TABLE 3.2.2 TABLES FOR GAME PROGRESS TABLE

Trap Mechanism Table

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Data Type** | **Description** |
| **Trap\_ID** | INT | Unique identifier for traps |
| **Type** | STRING | Type of trap (spike, saw, etc..) |
| **Position\_X** | FLOAT | X-coordinate of trap |
| **Position\_X** | FLOAT | Y-coordinate of trap |
| **Damage** | INT | Damage inflicted |

TABLE 3.2.2 TABLES FOR TRAP MECHANISM TABLE

# CHAPTER - 4 SYSTEM IMPLEMENTATION

**IMPLEMENTATION**

Implementation is the stage in the project where the theoretical design is turned into the working system and is giving confidence to the new system for the users i.e. will work efficiently and effectively. It involves careful planning, investigation of the current system and its constraints on implementation, design of method to achieve the changeover, an evaluation, of change over methods. A part from planning major task of preparing the implementation is education of users. The more complex system is implemented, the more involved will be the system analysis and design effort required just for implementation. An implementation coordinating committee based on policies of individual organization has been appointed. The implementation process begins with preparing a plan for the implementation for the system. According to this plan, the activities are to be carried out; discussions may regarding the equipment have to be acquired to implement the new system.

Implementation is the final and important phase. The most critical stage is in achieving a successful new system and in giving the users confidence that the new system will work and be effective. The system can be implemented only after thorough testing is done and if it found to working according to the specification. This method also offers the greatest security since the old system can take over if the errors are found or inability to handle certain types of transaction while using the new system.

At the beginning of the development phase a preliminary implementation plan is created to schedule and manage the many different activities that must be integrated into plan. The implementation plan is updated throughout the Development phase, culminating in a changeover plan for the operation phase. The major elements of implementation plan are test plan, training plan, equipment installation plan, and a conversion plan.

**There are three types of implementation:**

* Implementation of a computer system to replace a manual system.
* Implementation of a new computer system to replace an existing system.
* Implementation of a modified application to replace an existing one, using the same computer.

Successful implementation may not guarantee improvement in the organization using the new system, but improper installation will prevent it. It has been observed that even the best system cannot show good result if the analysts managing the implementation do not attend to every important detail. This is an area where the systems analysts need to work with utmost care.

**4.1 FEASIBILITY ANALYSIS:**

The trap-based 2D platformer game is feasible in terms of technology, cost, operations, and schedule. It is developed in Unity with C#, ensuring smooth physics and animations. The project is cost-effective, using Unity’s free version and open-source assets. The gameplay is engaging and challenging, focusing on reflex-based traps instead of combat. Development follows an Agile model, with an estimated timeline of 3–6 months.

* + 1. ECONOMIC FEASIBILITY

The trap-based 2D platformer game is economically feasible due to its low development costs and efficient resource management. Unity’s free version allows development without licensing fees, and open-source or affordable assets reduce expenses for game elements like sprites, sounds, and UI components. The primary costs involve developer time, optional premium assets, and potential marketing expenses if the game is to be commercially released. Additionally, monetization strategies like in-game ads, premium versions, or crowdfunding can help recover costs.

* + 1. TECHNICAL FEASIBILITY

The trap-based 2D platformer game is technically feasible as it utilizes Unity and C#, which provide a robust framework for handling physics, animations, and trap mechanics efficiently. The game requires a midrange PC with an Intel i5 processor, 8GB RAM, and a GTX 1050 GPU, making it accessible for both development and gameplay. Unity’s built-in tools like Rigidbody2D, Colliders, Tile maps, and Animator simplify implementation, while Visual Studio serves as the primary IDE for scripting and debugging. Additionally, crossplatform support ensures the game can be deployed on Windows, macOS, and even mobile platforms if required.

* + 1. OPERATIONAL FEASIBILITY

The trap-based 2D platformer game is operationally feasible as it offers an engaging, skill-based experience that aligns with player expectations. The game mechanics focus on timing, reflexes, and logic-based challenges, ensuring a fun and challenging experience without relying on combat. Its intuitive UI, smooth character controls, and progressive difficulty levels enhance usability and accessibility for a wide audience. From a development standpoint, Unity’s tools streamline game design, testing, and deployment, making operations manageable for developers. Additionally, the game’s scalability allows future updates, such as new levels, traps, or multiplayer features, ensuring long-term engagement.

**4.2 CODING DETAILS**

### Gamemanager.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class Gamemanager : MonoBehaviour

{ public void PlayGame()

{

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);

}

public void QuitGame()

{

Application.Quit();

}

}

### Menubt.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class menubt: MonoBehaviour

{

// Start is called once before the first execution of Update after the MonoBehaviour is created

public void MainMenu()

{

SceneManager.LoadScene("MainMenu");

### Mover.CS

using UnityEngine;

public class mover : MonoBehaviour

{

[SerializeField] Transform[] points; [SerializeField] float speed = 2f; int counter = 0;

// Start is called once before the first execution of Update after the MonoBehaviour is created void Start()

{

}

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

{

if (Vector3.Distance(transform.position, points[counter].position) < 0.1f)

{

counter++;

}

if (counter >= points.Length)

{ counter = 0;

}

transform.position = Vector3.MoveTowards(transform.position, points[counter].position, speed \* Time.deltaTime);

}

}

#### Options.CS

using UnityEngine;

using UnityEngine.SceneManagement;

public class options : MonoBehaviour

{

public void OpenLevel(int LevelID)

{

string Levelname = "Level" + LevelID; SceneManager.LoadScene(Levelname);

}

}

#### 

#### Playerhealth.CS

using UnityEngine; using UnityEngine.SceneManagement; public class playerhealth : MonoBehaviour

{

Animator animator;

[SerializeField] AudioSource deathAudio;

private void Start()

{

animator = GetComponent<Animator>();

}

private void OnCollisionEnter2D(Collision2D other)

{

if (other.transform.tag == "trap")

{

animator.SetTrigger("death");

GetComponent<playermovement>().enabled = false; deathAudio.Play();

}

}

public void RestartLevel()

{

SceneManager.LoadScene(SceneManager.GetActiveScene().name);

}

}

#### 

#### Playermoment.CS

using UnityEngine;

public class playermovement : MonoBehaviour

{

[SerializeField] float jumpForce = 18f; [SerializeField] float runSpeed = 5f; float dirX;

Rigidbody2D rb;

BoxCollider2D playerCollider2D; // Renamed to playerCollider2D

[SerializeField] LayerMask groundmask;

SpriteRenderer spriteRenderer; Animator animator; bool gamePaused = false;

[SerializeField] AudioSource jumpAudio;

private enum Movementstate { Idle, Run, Jump, Fall }

void Start()

{

rb = GetComponent<Rigidbody2D>(); playerCollider2D = GetComponent<BoxCollider2D>(); // Use the renamed variable here spriteRenderer = GetComponent<SpriteRenderer>(); animator = GetComponent<Animator>();

}

void Update()

{

dirX = Input.GetAxisRaw("Horizontal");

// Handle Jumping if (Input.GetKeyDown(KeyCode.Space) && iSGrounded())

{

rb.linearVelocity = new Vector2(rb.linearVelocity.x, jumpForce); // Only apply vertical velocity change jumpAudio.Play();

}

// Handle Animations

HandleAnimation();

// Handle Pause functionality

PauseGame();

}

void PauseGame()

{ if (Input.GetKeyDown(KeyCode.Escape)) // Pause when Escape key is pressed

{

gamePaused = !gamePaused;

}

if (gamePaused)

{

Time.timeScale = 0f; // Stop time (pauses the game) AudioListener.pause = true; // Pause all audio

} else

{

Time.timeScale = 1f; // Resume time

AudioListener.pause = false; // Unpause audio

}

}

// FixedUpdate is used for physics updates, like movement private void FixedUpdate()

{

// Apply horizontal movement (keep vertical velocity unchanged) rb.linearVelocity = new Vector2(dirX \* runSpeed \* Time.deltaTime, rb.linearVelocity.y);

}

// Check if the player is grounded (touching the ground) bool iSGrounded()

{

return Physics2D.BoxCast(playerCollider2D.bounds.center, playerCollider2D.bounds.size, 0, Vector2.down, 0.1f, groundmask);

}

// Handle player animation based on movement state void HandleAnimation()

{

Movementstate state;

// Check running direction if (dirX > 0)

{

spriteRenderer.flipX = false; // Face right

state = Movementstate.Run;

} else if (dirX < 0)

{

spriteRenderer.flipX = true; // Face left state = Movementstate.Run;

} else

{

state = Movementstate.Idle; // Idle if no horizontal input

}

// Check if player is jumping or falling if (rb.linearVelocity.y > 0.1f)

{

state = Movementstate.Jump;

}

else if (rb.linearVelocity.y < -0.1f)

{

state = Movementstate.Fall;

}

// Set the animator's integer value for movement state animator.SetInteger("state", (int)state);

}

}

#### Replaybt.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class replaybt : MonoBehaviour

{ public GameObject REPLAY;

// Start is called once before the first execution of Update after the MonoBehaviour is created

public void Restart()

{

SceneManager.LoadScene("Level1");

}

}

#### Rotator.CS

#### 

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

public class rotator : MonoBehaviour

{

[SerializeField] float speed = 4f;

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

{

transform.Rotate(new Vector3(0f, 0f, 360f \* speed \* Time.deltaTime));

}

}

#### Scoremanager.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class ScoreManager : MonoBehaviour

{ private int score;

void Start()

{

// Load the previous score if it exists score = PlayerPrefs.GetInt("TotalScore", 0);

}

// Call this function to add score public void AddScore(int points)

{

score += points;

PlayerPrefs.SetInt("TotalScore", score); // Save the updated score

}

// Call this to get the total score public int GetTotalScore()

{ return score;

}

// You can reset the score (e.g., at the start of a new game) public void ResetScore()

{ score = 0;

PlayerPrefs.SetInt("TotalScore", score);

}

// Call this to load another scene public void LoadScene(int sceneIndex)

{

SceneManager.LoadScene(sceneIndex);

}

}

#### Stickplatform.CS

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

public class stickplatform : MonoBehaviour

{ private void OnTriggerEnter2D(Collider2D other){ if (other.transform.tag == "Player")

{

other.transform.SetParent(this.transform);

}

}

private void OnTriggerExit2D(Collider2D other){ if (other.transform.tag == "Player")

{ other.transform.SetParent(null);

}

}

}

#### Totalscore.Cs

using System.Collections; using System.Collections.Generic; using UnityEngine; public class totalscore : MonoBehaviour

{

[SerializeField] private int KiwiCollected; private void OnEnable()

{

if (PlayerPrefs.HasKey("Kiwi"))

{

KiwiCollected = PlayerPrefs.GetInt("Kiwi");

}

}

// Start is called before the first fram update

void Start()

{

KiwiCollected = 0;

} public int GetKiwi()

{

return KiwiCollected;

} private void OnTriggerEnter2D(Collider2D collision)

{ if (collision.CompareTag("fruit"))

{

// oh we ran into a collectible!

Destroy(collision.gameObject);

KiwiCollected++;

}

}

private void OnDestroy()

{ savePrefs();

}

void savePrefs()

{

// Set the PlayerPref of 'Coins' with the number of coins Collected

PlayerPrefs.SetInt("Kiwi", KiwiCollected);

PlayerPrefs.Save();

}

}

#### Camerfollow.CS

using UnityEngine;

public class cameraFollow : MonoBehaviour {

[SerializeField] Transform player;

// Start is called once before the first execution of Update after the MonoBehaviour is created void Start()

{

}

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

{

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

}

}

#### Collector.CS

#### 

#### 

using UnityEngine; using UnityEngine.UI;

public class collector : MonoBehaviour

{

[SerializeField] Text countText; // UI Text to show score

[SerializeField] AudioSource coinAudio; // Audio when collecting fruit private int countKiwi = 0; // Track collected kiwis

private void Start()

{

// Load the current kiwi count from PlayerPrefs (if any)

countKiwi = PlayerPrefs.GetInt("KiwiCount", 0); // Default to 0 if not set countText.text = "Kiwi: " + countKiwi; // Update the UI text

}

private void OnTriggerEnter2D(Collider2D other)

{ if (other.gameObject.CompareTag("fruit")) // Check if we hit a fruit { coinAudio.Play(); // Play sound countKiwi++; // Increment score countText.text = "Kiwi: " + countKiwi; // Update the score UI

// Save the score to PlayerPrefs

PlayerPrefs.SetInt("KiwiCount", countKiwi);

// Destroy the fruit object

Destroy(other.gameObject);

}

}

}

**Finalscenescore.CS**

using UnityEngine; using UnityEngine.UI;

public class FinalSceneScore : MonoBehaviour

{

public Text TotalScoreText; // UI Text component to display the score

void Start()

{

// Load the scores from PlayerPrefs int scene1Score = PlayerPrefs.GetInt("Kiwi", 0); // Scene 1 score (key "Kiwi") int scene2Score = PlayerPrefs.GetInt("Kiwi", 0); // Scene 2 score (key "Kiwi")

// Calculate the total score int totalScore = scene1Score + scene2Score; // Display the total score on the UI Text

TotalScoreText.text = "YOUR SCORE: " + totalScore.ToString();

}

// You can also create a function that resets the score if called public void ResetScores()

{

PlayerPrefs.DeleteKey("Scene1Score");

PlayerPrefs.DeleteKey("Scene2Score");

PlayerPrefs.Save();

}

}

#### Finalscore.CS

using UnityEngine; using UnityEngine.UI;

public class FinalScore : MonoBehaviour

{

[SerializeField] Text finalScoreText; // UI Text to show final score

void Start()

{

// Retrieve the total collected kiwi count (across both Level 1 and Level 2) int totalKiwi = PlayerPrefs.GetInt("KiwiCount", 0); // Load saved score (default 0)

// Display the total kiwi score in the UI Text finalScoreText.text = "YOUR SCORE: " + totalKiwi; }

public void ResetScore()

{

PlayerPrefs.DeleteKey("KiwiCount"); // Reset score

// Or use PlayerPrefs.DeleteAll() to clear all data

}

}

#### Finish.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class finish : MonoBehaviour

{

[SerializeField] AudioSource finishAudio;

private void OnTriggerEnter2D(Collider2D other)

{

if (other.gameObject.tag == "Player")

{

Invoke("NextLevel", 1.25f); finishAudio.Play();

}

}

void NextLevel()

{

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex+1);

}

}

#### Lvl1.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class lvl1 : MonoBehaviour

{ public GameObject REPLAY;

// Start is called once before the first execution of Update after the MonoBehaviour is created

public void Lvl1()

{

SceneManager.LoadScene("Level1");

}

}

#### Lvl2.CS

using UnityEngine; using UnityEngine.SceneManagement;

public class lvl2 : MonoBehaviour

{ public GameObject REPLAY;

// Start is called once before the first execution of Update after the MonoBehaviour is created

public void Lvl2()

{

SceneManager.LoadScene("Level2");

}

}

**4.3 SCREEN SHOT:**



FIGURE 4.3.1 MAIN MENU



FIGURE 4.3.2 LEVEL OPTIONS



FIGURE 4.3.3 PLAYER OR PLAYERMOVEMENT



FIGURE 4.3.4 TRAPS



FIGURE 4.3.5 COLLECTABLE

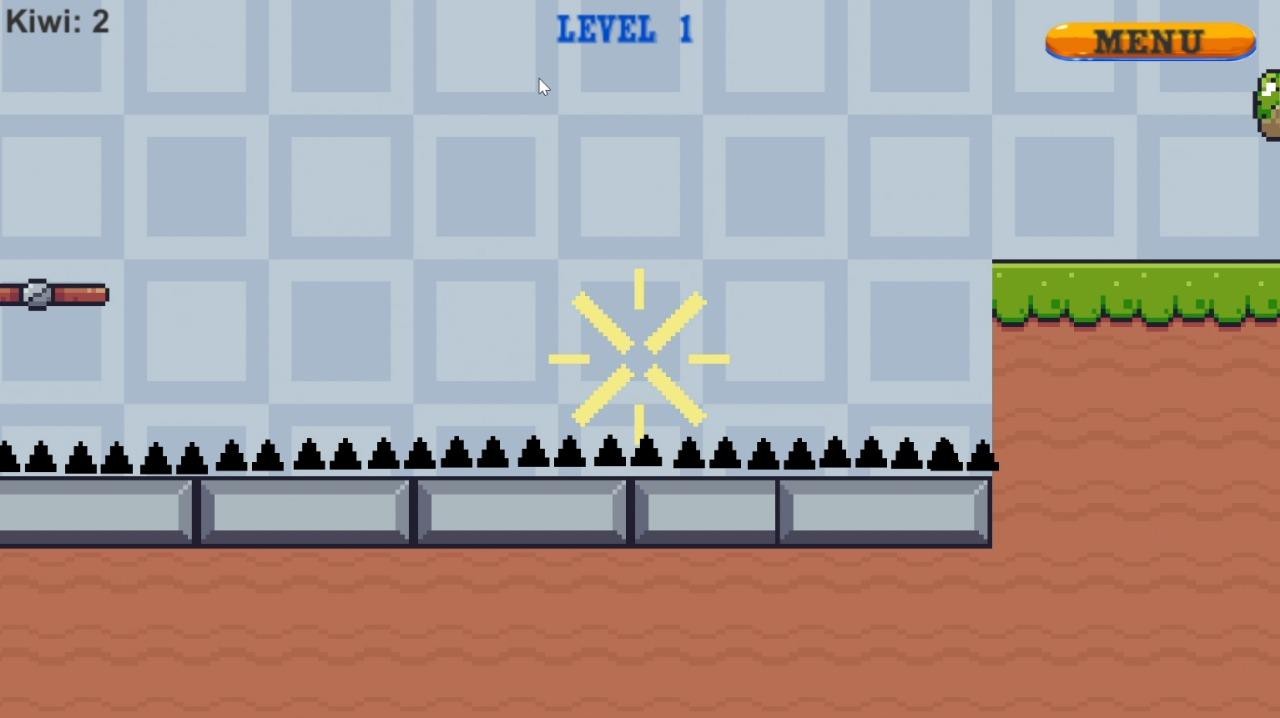


FIGURE 4.3.6 PLAYER DEATH



FIGURE 4.3.7 EXIT MENU

**CHAPTER – 5 SYSTEM TESTING**

**TESTING:**

The trap-based 2D platformer game undergoes rigorous testing to ensure smooth gameplay, bug-free mechanics, and an engaging user experience. The testing phase includes unit testing to check individual game components, such as player movement, collision detection, and trap mechanics. Functional testing ensures that all features, including UI interactions, level progression, and sound effects, work as intended. Performance testing evaluates frame rates, load times, and memory usage across different devices to optimize efficiency. User testing is conducted with players to gather feedback on difficulty balance, controls, and overall enjoyment. After identifying and fixing bugs, the game goes through final testing before release, ensuring a polished and highquality product.

**5.1 TESTING PROCEDURE:**

The testing phase of the trap-based 2D platformer game ensures smooth gameplay, bug-free mechanics, and an enjoyable user experience. It starts with unit testing, where individual components like player movement, traps, and collision detection are tested separately. This is followed by functional testing, which checks if core features like jumping, level progression, and UI interactions work correctly.

Next, performance and compatibility testing are conducted to evaluate frame rates, memory usage, and loading times across different devices. This ensures the game runs smoothly on various operating systems and hardware configurations without major issues. Any performance bottlenecks are identified and optimized for better efficiency.

Finally, user testing is performed by real players to gather feedback on controls, difficulty, and overall experience. Based on this feedback, necessary adjustments are made, and bug fixing is carried out. After a final round of testing, the game is polished and prepared for release, ensuring a high-quality and stable product.

5.1.1 STRUCTURAL TESTING**:**

Structural testing focuses on evaluating the internal structure and logic of the game’s code to ensure it functions correctly. This type of testing involves examining the underlying C# scripts used in Unity, including player movement, trap mechanics, and physics interactions. Developers perform white-box testing, which allows them to check code execution paths, loops, and conditions to identify logical errors or inefficiencies.

Different techniques such as control flow testing and data flow testing are used to verify that the game processes inputs correctly and triggers appropriate responses. Test cases are designed to cover all possible scenarios, including edge cases like unexpected player actions or rapid button presses that could cause unexpected behavior. By analyzing the structure of functions and modules, developers can optimize code efficiency and reduce redundancy.

Once structural issues are identified, necessary improvements are made to enhance game stability and performance. Automated testing tools can be used to simulate interactions and check for memory leaks or infinite loops that could crash the game. After fixing errors, the game undergoes another round of testing to ensure all structural components work seamlessly, leading to a well-optimized and stable platformer experience.

1. Code Execution Testing – Evaluates C# scripts in Unity to verify the correct execution of player movement, traps, and physics interactions.

1. Control Flow Testing – Analyzes loops, conditions, and execution paths to detect logical errors and ensure efficient processing.

1. Data Flow Testing – Checks how variables and data move within the code to prevent unexpected crashes or incorrect calculations.

1. Boundary & Edge Case Testing – Tests unusual player inputs, extreme conditions, and rapid actions to identify potential glitches.

1. Debugging & Optimization – Fixes inefficient code, memory leaks, and performance issues to enhance game stability before release.

5.1.2 FUNCTIONAL TESTING

Functional testing ensures that all game features work as intended and provide a smooth player experience. It begins with verifying core mechanics such as player movement, jumping, trap activation, and level progression. This testing ensures that controls respond correctly to user inputs, whether using a keyboard, mouse, or game controller.

Any issues with lag, unresponsive buttons, or broken mechanics are identified and fixed to maintain fluid gameplay.

Another crucial aspect is game mechanics testing, where interactions between traps, physics, animations, and collision detection are examined. This prevents bugs like a player passing through obstacles or traps failing to trigger at the right moment. Additionally, the UI and HUD are tested to ensure that menus, health bars, score displays, and notifications function correctly and provide accurate information.

Finally, end-to-end testing is conducted to simulate a full gameplay session, ensuring a seamless experience from start to finish. Testers play through different levels, identifying unexpected bugs, balancing issues, or inconsistencies in difficulty. Once all issues are resolved, the game is refined for performance, usability, and stability, ensuring a high-quality final product.

5.1.3 TEST DATA USED

Test data is essential in ensuring that all game mechanics function correctly in the trap-based 2D platformer. One key aspect is player input data, where various user actions such as jumping, running, stopping, and interacting with traps are tested under different conditions. This helps verify that all controls respond smoothly and without delay. Additionally, multiple input combinations, such as pressing movement and jump keys simultaneously, are tested to check for unexpected behaviors like movement lag or animation glitches. Edge cases, such as holding keys for an extended time or rapidly pressing buttons, are also analyzed to ensure stability.

Another important area of testing involves collision and physics data. The game world consists of platforms, walls, obstacles, and traps, all of which must interact correctly with the player and the environment. Testing involves simulating different scenarios where the player collides with objects, ensuring they respond appropriately based on game physics. For example, if a player jumps onto a moving platform, the game should correctly register the player's movement with it. Similarly, if the player collides with a trap, the appropriate action, such as reducing health or restarting the level, should occur.

Additionally, gravity effects, acceleration, and object interactions are tested to ensure smooth physics behavior throughout the game.

Other critical test data include level progression, trap activation, and UI elements. The game must properly save and load progress at different checkpoints to prevent data loss or incorrect state resets. Traps are tested under different conditions, such as triggering at various speeds and from different angles, to ensure they provide a fair yet challenging experience. The UI and score display are also tested to ensure accurate updates when the player earns points, takes damage, or progresses to a new level. These tests help refine the game, ensuring it offers a seamless and engaging experience without unexpected issues.

* Player input data is tested by simulating actions like jumping, running, and interacting with traps to ensure smooth and responsive controls.
* Collision and physics data are examined by checking interactions between the player, platforms, walls, and traps to ensure realistic game mechanics.
* Level progression data is tested by saving and loading game states at checkpoints to verify correct transitions and progress retention.
* Trap activation data is analyzed under different conditions, such as varying speeds and angles, to maintain a fair and consistent challenge.
* UI and score display data are tested to ensure accurate updates for menus, health bars, timers, and score counters during gameplay.

5.1.4 TEST CASES

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Case ID** | **Test Scenario** | **Test Steps** | **Expected Result** | **Status** |
| **TC01** | Player movement | Move left, right, and  jump | Character moves smoothly | Pass/Fail |
| **TC02** | Collision with obstacles | Run into walls and platforms | Character stops upon impact | Pass/Fail |
| **TC03** | Trap activation | Step on a trap | Trap triggers correctly | Pass/Fail |
| **TC04** | Falling off platforms | Move off the edge of a platform | Character falls and  respawns | Pass/Fail |
| **TC05** | Level progression | Complete a level and enter the next one | Next level loads successfully | Pass/Fail |
| **TC06** | Health reduction | Take a damage from a trap | Health decreases as expected | Pass/Fail |
| **TC07** | UI functionality | Navigate menu, start game, pause  game | Buttons work as expected | Pass/Fail |
| **TC08** | Score calculation | Complete actions that earn points | Score updates correctly | Pass/Fail |
| **TC09** | Save and load game | Save game at checkpoint, then reload | Game loads at correct state | Pass/Fail |
| **TC10** | Performance testing | Play the game for extend | No lag, crashes, or glitches | Pass/Fail |

## 5.2 SYSTEM TESTING:

System testing ensures that the entire game functions as a complete and integrated system. It involves testing all components, such as player mechanics, level progression, traps, UI, and performance, working together without errors. The game is tested under different environments and hardware specifications to ensure it runs smoothly on various systems. Developers also verify that external dependencies like audio files, animations, and saved game states function correctly across different playthroughs.

One key aspect of system testing is end-to-end gameplay testing, where the game is played from start to finish to identify any issues with mechanics, level transitions, or overall experience. Load testing is conducted to check how the game performs under stress, such as multiple enemies or complex trap activations at the same time. Additionally, compatibility testing ensures the game runs on different operating systems, screen resolutions, and controller types without issues.

Finally, security and data integrity are verified to prevent potential game crashes, glitches, or exploits. Save and load features are tested to ensure progress is stored correctly and retrieved without corruption. Performance metrics, such as frame rate stability, memory usage, and CPU load, are monitored to optimize the game for a smooth experience. After system testing, any detected bugs are fixed before the game is finalized for release.

* The game is tested as a complete system to ensure all features work together without errors. o Different hardware and software environments are used to check compatibility and performance.
* End-to-end gameplay testing is performed to identify any issues from start to finish. o Load testing is conducted to examine how the game handles multiple interactions simultaneously. o Screen resolution and controller compatibility tests ensure smooth functionality across devices. o Save and load features are tested to verify that game progress is stored and retrieved correctly.
* Performance metrics like frame rate, memory usage, and CPU load are monitored for optimization. o Security testing checks for potential crashes, glitches, or exploits that could impact gameplay. o The final system testing phase ensures a stable, polished, and ready-to-release game.

5.2.1 GOALS OF THE PROPOSED SYSTEM:

### 1. Enhance Gameplay Experience

The main goal of the system is to create a trap-based 2D platformer that shifts the focus from traditional enemy combat to obstacle-based challenges. This encourages players to think strategically and master movement-based mechanics rather than relying on attacking opponents. The game is designed to be engaging and immersive, keeping players interested with unique challenges at every stage.

### 2. Improve Player Skills

To successfully navigate through levels, players must develop strong timing, reflexes, and logical thinking. The game introduces different traps and obstacles that require quick decision-making and precision-based actions. By progressively increasing the difficulty, the system ensures that players improve their skills naturally as they advance through the game.

### 3. Ensure Smooth Performance

Performance optimization is crucial to maintaining a seamless and lag-free gaming experience. The system is designed to run efficiently on different devices by optimizing movement physics, animations, and interactions. By ensuring a stable frame rate and responsive controls, the game provides a smooth and enjoyable experience for all players.

### 4. Create Interactive Level Design

The game’s level design is structured to gradually introduce new traps, mechanics, and interactive elements. This keeps gameplay fresh and exciting while ensuring a steady learning curve. Each level presents unique challenges that push players to explore different strategies, improving overall engagement and satisfaction.

### 5. Develop a User-Friendly Interface

A well-designed UI and HUD system ensures that players can easily navigate menus, access settings, and monitor game elements like health, timers, and progress. The system is designed to be intuitive and visually appealing, making it accessible to both new and experienced gamers.

### 6. Implement a Reliable Save and Load System

Saving and loading progress is essential to providing a smooth experience for players. The system ensures that checkpoints and saved progress function correctly, allowing players to resume gameplay without losing achievements.

A well-implemented save system prevents frustration and encourages longer play sessions.

### 7. Ensure Cross-Platform Compatibility

To reach a wider audience, the game is designed to run smoothly on different operating systems, screen sizes, and hardware configurations. Whether played on a mid-range PC or a high-performance gaming setup, the system ensures consistent performance and playability.

### 8. Integrate High-Quality Visuals and Audio

A well-designed game must have visually appealing graphics and immersive sound effects. The system incorporates detailed backgrounds, smooth animations, and responsive sound cues that enhance the overall gaming experience.

High-quality visuals and audio contribute to player immersion and enjoyment.

### 9. Conduct Thorough Testing for Stability

A key goal of the system is to ensure a bug-free and stable release. Through rigorous testing, potential issues such as glitches, performance drops, and crashes are identified and fixed. This ensures that players can enjoy a polished and error-free gaming experience.

### 10. Increase Replay Value and Engagement

To keep players engaged, the system includes features like challenges, achievements, and score-based rewards. These elements encourage replay ability, motivating players to improve their performance and discover hidden challenges in each level. By offering incentives for mastery, the game maintains long-term engagement and enjoyment.

# CHAPTER - 6 PROJECT EVALUATION

**6.1 SALIENT FEATURE:**

* Trap-based gameplay focusing on reflexes and strategy instead of combat.
* Smooth controls for precise jumping, running, and dodging.
* Progressive levels with increasing difficulty and new mechanics.
* Dynamic environments with moving platforms and timed obstacles.
* High-quality visuals and immersive sound effects.
* Save and load system for progress retention.
* Cross-platform compatibility for smooth performance on different devices.
* User-friendly interface with an intuitive menu and HUD.
* Challenging gameplay with scoring, leaderboards, and achievements.
* Bug-free experience through rigorous testing and optimization.

**6.2 LIMITATION OF THE SYSTEM**

The game relies heavily on trap-based mechanics, which may not appeal to players who prefer combatoriented platformers. Since enemy interactions are minimal, some players might find the gameplay lacking in variety compared to traditional platformers.

Additionally, performance could be affected on lower-end hardware, leading to occasional frame drops or lag, which might impact the overall experience.

The complexity of trap timing and precision-based movements could make it difficult for casual players to progress, potentially causing frustration. Players who struggle with fast reflex-based challenges may find the game too demanding.

Furthermore, the save and load system may not support mid-level saves, requiring players to complete an entire level in one go, which could be challenging in longer levels.

Multiplayer or co-op options are limited, which may restrict social gaming experiences. Compatibility issues might also arise with certain screen resolutions or unusual hardware setups, requiring additional optimizations.

Lastly, the game may require frequent updates to fix bugs, improve mechanics, and introduce new content

to maintain player engagement and ensure a smooth gameplay experience.

LIMITATIONS OF THE SYSTEM

1. The game’s focus on traps may not appeal to combat-loving players.
2. Limited enemy interactions could make gameplay feel repetitive.
3. Performance issues may occur on lower-end devices.
4. High difficulty might frustrate casual players.
5. Save and load options may not support mid-level saves.
6. Lack of multiplayer limits social gameplay.
7. Compatibility issues may arise on certain hardware setups.

6.2.1 PROJECT LEGACY

The project legacy ensures that the trap-based 2D platformer remains relevant and adaptable for future improvements. It provides a strong foundation for expanding gameplay mechanics, introducing new levels, and optimizing performance. The codebase and design principles allow future developers to enhance features, fix bugs, and integrate updates without major restructuring.

The project documentation ensures smooth knowledge transfer, making it easier for new developers to understand the system and build upon it. The game’s scalability allows for future compatibility with different platforms, improved graphics, and additional content such as multiplayer modes or customization features. With continuous updates and community engagement, the project can maintain long-term player interest and expand its audience.

By establishing a structured development framework, the project legacy supports future improvements while preserving core gameplay mechanics. It enables developers to refine user experience, introduce creative challenges, and ensure sustained engagement. The game’s modular design also allows for seamless adaptation to evolving technology and gaming trends, keeping it relevant for years to come.

**6.3 FUTURE SCOPE OF THE PROJECT**

The future scope of the project includes expanding the game with new levels, mechanics, and challenges to enhance player engagement. Additional traps and interactive elements can be introduced to keep the gameplay fresh and exciting. Future updates may also include difficulty settings, allowing players to customize their experience based on their skill level.

Cross-platform compatibility can be improved by optimizing the game for mobile devices, consoles, and different operating systems. Multiplayer or co-op modes can be implemented to provide a more interactive and social gaming experience. Enhanced AI-driven trap behaviors and dynamic level generation can also be explored to create a more unpredictable and immersive environment.

Regular updates with bug fixes, performance optimizations, and graphical enhancements will help maintain the game’s longevity. Community-driven features such as leaderboards, user-generated content, and mod support can increase replay ability. Expanding the project with downloadable content or expansions can keep the player base engaged, ensuring the game remains relevant in the evolving gaming industry.

6.3.1 SCOPE AND FEASIBILITY

* The project focuses on developing a trap-based 2D platformer with strategic obstacles instead of combat.
* Smooth character movement, engaging level design, and interactive traps enhance the player experience.
* Performance optimization ensures accessibility across different hardware configurations.
* Future expansions may include new levels, mechanics, and multiplayer features.
* Technically feasible with Unity and C#, utilizing existing tools for efficient development.
* Economically viable with minimal investment, as most development tools are free or costeffective.
* Operationally practical, catering to platformer enthusiasts and offering a challenge-driven experience.
* The structured design allows for scalability, enabling future updates and expansions.  Continuous improvements based on player feedback ensure long-term sustainability.
* - The game has the potential to attract and maintain a dedicated player base over time.

**CHAPTER- 7 BIBLIOGRAPY**

This document contains provisions which, through reference in this text, constitute provisions of the present document.

1. Google Search Engine for various searching
2. Unity Documentation – Official site for Unity Game Engine
3. C# Programming Guide – Microsoft Docs for scripting reference
4. Game Development Patterns – Reference for structure and design techniques

**CONCLUSION:**

The development of this 2D platformer game has provided an innovative approach to gameplay by focusing on trap-based challenges rather than traditional combat. The project successfully integrates smooth character movement, interactive UI, and progressively difficult levels to engage players. By using Unity and C# for development, the game ensures a seamless and optimized experience across different hardware configurations.

Through careful planning, requirement analysis, and implementation, the project has achieved its primary objectives of providing an engaging and skill-based platformer. The feasibility analysis confirms that the game is technically and economically viable, making it a sustainable project for further development. Various testing procedures, including functional and structural testing, have ensured that the game runs efficiently without major bugs or performance issues.

In the future, the game can be expanded with additional levels, new mechanics, and multiplayer features to enhance the overall experience. Regular updates and community feedback will play a crucial role in the

continued success of the game. With ongoing improvements and innovations, this project has the potential to attract and retain a dedicated player base, making it a valuable contribution to the platformer gaming genre.