****

**2D Game**

**2024**

**The Egyptian E-Learning University**

**Faculty of Computers and Information Technology**

**Department of Information Technology**

**Final Year Project**

**Team Members**

Mohamed Hesham Hafez : 20-01268

Abdelrahman Mahmoud : 20-00375

Ali Ashraf Ali : 20-01513

Omar Adel El Gazzar : 20-01298

Bassem Radi : 20-01151

Sherif Mostafa : 20-01363

**Supervised by**

**Dr.** Khaled Fathy

**Eng.** Mohamed Saeed

**Abstract**

This document provides a comprehensive overview of the development of a 2D game, outlining the technical implementation, tools, methodologies, and processes involved. The game is programmed in C# using the Unity game engine, chosen for its robust feature set and community support. Animation was facilitated through Unity Animator, Aseprite, and Photoshop, while Git and GitHub were employed for version control. The development process encompassed initial planning and design, core mechanics implementation, level design, UI creation, and animation. Rigorous testing, including unit and integration tests, ensured functionality and performance, while user feedback informed gameplay balance and UI enhancements. The game achieved stable performance metrics, including a consistent 60 fps frame rate and efficient resource utilization. Future work will focus on adding new features, expanding platform compatibility, and further enhancing the user experience. This documentation captures the key achievements, challenges, and lessons learned throughout the development journey, providing valuable insights for future projects.

**Acknowledgments**

First, we would like to thank Allah for helping us to complete this project successfully.

We are heartily thankful to Prof. Dr. Khaled Fathy who not only served as our

supervisors, but also guided, encouraged and challenged us throughout the project, and

guided us with great dedication, never accepting less than our best efforts, and allowing for

the completion and success of this project.

Also We would like to thank all teaching assistants at EELU for educating and

supporting us for four years.

We also owe our deepest gratitude to our families for their great support, which without, our work would not have been successful.

**Table of Contents :**

1. [**Abstract**](#Abstract)
2. [**Acknowledgements**](#Acknowledgments)
3. [**Table of Contents**](#Table_of_Contents)
4. [**List of Codes**](#Listofcodes)
5. [**List of Photos**](#ListofPhotos)
6. [**Introduction**](#Introduction)
   1. General
   2. Team Composition
   3. Objectives and Scope
   4. Structure of the Documentation
7. **[Background](#Background)**
   1. History of 2D Games
   2. Evolution and Significance of 2D Games
   3. Overview of Unity Engine
   4. Rationale for Choosing C# and Unity
   5. Previous Work and Inspirations
8. **[The Craft of Game Development and Design](#TheCraftofGameDevelopmentandDesign)**
   1. The Art of Game Design
   2. Mechanics and Dynamics
   3. Programming
   4. Art and Animation
   5. Sound Design
   6. User Experience (UX) Design
9. [**2D Game Development Using C# and Unity**](#TwoDGameDevelopmentUsingC)
   1. Introduction
   2. Game Concept and Design
   3. Game Mechanics
   4. Programming with C#
   5. Script Development
   6. Unity Overview
   7. Project Setup
   8. Level Design
10. [**Animation System**](#AnimationSystem)
    1. Character Animation
    2. Environment Animation
11. [**Algorithms and Code and Implementation**](#AlgorithmsandCodeAndImplementation)
    1. Introduction
    2. Overview of Enemy Tracking in Games
    3. Theoretical Background
    4. Implementation of Enemy Tracking Algorithm
    5. Code Implementation in Unity
    6. Testing and Optimization
    7. Implementation Overview
    8. Programming Language and Tools
    9. Game Development Process
    10. Animation
    11. Scripting and Logic
    12. Testing and Debugging
12. [**Results**](#Results)
    1. Game performance
    2. User feedback
    3. Visual and Audio Quality
13. [**Discussions**](#Discussions)
    1. Achievements
    2. Challenges and Solutions
    3. Lessons Learned
    4. Future Works
    5. References

**List of Codes**

**Example Snippet from the PlayerMovement Script C# :**

using UnityEngine;

public class PlayerMovement : MonoBehaviour {

    [Header("Movement Parameters")]

    [SerializeField] private float speed;

    [SerializeField] private float jumpPower;

    [Header("Coyote Time")]

    [SerializeField] private float coyoteTime;

//How much time the player can hang in the air before jumping

    private float coyoteCounter; //How much time passed since the player ran off the edge

    [Header("Multiple Jumps")]

    [SerializeField] private int extraJumps;

    private int jumpCounter;

    [Header("Wall Jumping")]

    [SerializeField] private float wallJumpX; //Horizontal wall jump force

    [SerializeField] private float wallJumpY; //Vertical wall jump force

    [Header("Layers")]

    [SerializeField] private LayerMask groundLayer;

    [SerializeField] private LayerMask wallLayer;

    [Header("Sounds")]

    [SerializeField] private AudioClip jumpSound;

    private Rigidbody2D body;

    private Animator anim;

    private BoxCollider2D boxCollider;

    private float wallJumpCooldown;

    private float horizontalInput;

    private void Awake() { //Grab references for rigidbody and animator from object

        body = GetComponent<Rigidbody2D>();

        anim = GetComponent<Animator>();

        boxCollider = GetComponent<BoxCollider2D>();}

    private void Update() {

        horizontalInput = Input.GetAxis("Horizontal");

        //Flip player when moving left-right

        if (horizontalInput > 0.01f)

            transform.localScale = Vector3.one;

        else if (horizontalInput < -0.01f)

            transform.localScale = new Vector3(-1, 1, 1);

        //Set animator parameters

        anim.SetBool("run", horizontalInput != 0);

        anim.SetBool("grounded", isGrounded()); //Jump

        if (Input.GetKeyDown(KeyCode.UpArrow)) Jump(); //Adjustable jump height

        if (Input.GetKeyUp(KeyCode.UpArrow) && body.velocity.y > 0)

            body.velocity = new Vector2(body.velocity.x

body.velocity.y / 2);

        if (onWall()) {

            body.gravityScale = 0;

            body.velocity = Vector2.zero;}

        else{

            body.gravityScale = 7;

            body.velocity = new Vector2(horizontalInput \* speed,

body.velocity.y);

            if (isGrounded()) {

                coyoteCounter = coyoteTime; //Reset coyote counter when on the ground

                jumpCounter = extraJumps; //Reset jump counter to extra jump value

            }

            Else

                coyoteCounter -= Time.deltaTime;

//Start decreasing coyote counter when not on the ground

        }

    }

    private void Jump() {

        if (coyoteCounter <= 0 && !onWall() && jumpCounter<= 0)return;

        //If coyote counter is 0 or less and not on the wall and don't have any extra jumps don't do anything

        SoundManager.instance.PlaySound(jumpSound);

        if (onWall())

            WallJump();

        else{

            if (isGrounded())

                body.velocity=new Vector2(body.velocity.x, jumpPower);

            else { //If not on the ground and coyote counter bigger than 0 do a normal jump

                if (coyoteCounter > 0)

                    body.velocity = new Vector2(body.velocity.x,

jumpPower);

                else {

                    if (jumpCounter > 0) //If we have extra jumps then jump and decrease the jump counter

                    {

                        body.velocity = new Vector2(body.velocity.x,

jumpPower);

                        jumpCounter--;

                    }

                }

            } //Reset coyote counter to 0 to avoid double jumps

            coyoteCounter = 0;

        }

    }

    private void WallJump() {

        body.AddForce(new Vector2(-Mathf.Sign(transform.localScale.x)

\* wallJumpX, wallJumpY));

        wallJumpCooldown = 0; }

    private bool isGrounded() {

        RaycastHit2D raycastHit = Physics2D.BoxCast(

boxCollider.bounds.center, boxCollider.bounds.size, 0,

Vector2.down, 0.1f, groundLayer);

        return raycastHit.collider != null; }

    private bool onWall() {

        RaycastHit2D raycastHit = Physics2D.BoxCast(

boxCollider.bounds.center, boxCollider.bounds.size, 0, new

Vector2(transform.localScale.x, 0), 0.1f, wallLayer);

        return raycastHit.collider != null; }

    public bool canAttack() {

        return horizontalInput == 0 && isGrounded() && !onWall();

    }

}

**Example Snippet from the Projectile Script C# :**

using UnityEngine;

public class Projectile : MonoBehaviour {

    [SerializeField] private float speed;

    private float direction;

    private bool hit;

    private float lifetime;

    private Animator anim;

    private BoxCollider2D boxCollider;

    private void Awake() {

        anim = GetComponent<Animator>();

        boxCollider = GetComponent<BoxCollider2D>(); }

    private void Update() {

        if (hit) return;

        float movementSpeed = speed \* Time.deltaTime \* direction;

        transform.Translate(movementSpeed, 0, 0);

        lifetime += Time.deltaTime;

        if (lifetime > 5) gameObject.SetActive(false); }

    private void OnTriggerEnter2D(Collider2D collision) {

        hit = true;

        boxCollider.enabled = false;

        anim.SetTrigger("explode");

        if (collision.tag == "Enemy")

            collision.GetComponent<Health>()?.TakeDamage(1); }

    public void SetDirection(float \_direction) {

        lifetime = 0;

        direction = \_direction;

        gameObject.SetActive(true);

        hit = false;

        boxCollider.enabled = true;

        float localScaleX = transform.localScale.x;

        if (Mathf.Sign(localScaleX) != \_direction)

            localScaleX = -localScaleX;

        transform.localScale = new Vector3(localScaleX,

transform.localScale.y, transform.localScale.z);}

    private void Deactivate() {

        gameObject.SetActive(false);

    }

}

**The following code snippet EnemyProjectile Using C# :**

using UnityEngine;

public class EnemyProjectile : EnemyDamage{

[SerializeField] private float speed;

[SerializeField] private float resetTime;

private float lifetime;

private Animator anim;

private BoxCollider2D coll;

private bool hit;

private void Awake(){

anim = GetComponent<Animator>();

coll = GetComponent<BoxCollider2D>();}

public void ActivateProjectile() {

hit = false;

lifetime = 0;

gameObject.SetActive(true);

coll.enabled = true;}

private void Update() {

if (hit) return;

float movementSpeed = speed \* Time.deltaTime;

transform.Translate(movementSpeed, 0, 0);

lifetime += Time.deltaTime;

if (lifetime > resetTime)

gameObject.SetActive(false);

}

private void OnTriggerEnter2D(Collider2D collision){

hit = true;

base.OnTriggerEnter2D(collision); //Execute logic from parent script first

coll.enabled = false;

if (anim != null)

anim.SetTrigger("explode");//When the object is a fireball explode it

else

gameObject.SetActive(false); //When this hits any object deactivate arrow

}

private void Deactivate(){ gameObject.SetActive(false); } }

**Below is a Sample of the Player Control Script C# :**

using UnityEngine;

public class PlayerController : MonoBehaviour{

public float moveSpeed = 5f;

public float jumpForce = 10f;

private Rigidbody2D rb;

private bool isGrounded;

void Start(){ rb = GetComponent<Rigidbody2D>(); }

void Update() {

Move();

Jump();}

void Move() {

float moveInput = Input.GetAxis("Horizontal");

rb.velocity = new Vector2(moveInput \* moveSpeed, rb.velocity.y); }

void Jump(){ if (Input.GetKeyDown(KeyCode.Space) && isGrounded) {

rb.velocity = new Vector2(rb.velocity.x, jumpForce); } }

void OnCollisionEnter2D(Collision2D collision){

if (collision.gameObject.CompareTag("Ground")) { isGrounded = true; } }

void OnCollisionExit2D(Collision2D collision) {

if (collision.gameObject.CompareTag("Ground")) {isGrounded = false; } } }

**Below is a Sample Script for updating the Health Bar C# :**

using UnityEngine;

using UnityEngine.UI;

public class Healthbar : MonoBehaviour{

[SerializeField] private Health playerHealth;

[SerializeField] private Image totalhealthBar;

[SerializeField] private Image currenthealthBar;

private void Start(){

totalhealthBar.fillAmount = playerHealth.currentHealth / 10;}

private void Update(){

currenthealthBar.fillAmount = playerHealth.currentHealth / 10;

}

}

**Example of an Animation Script for a CameraController C# :**

using UnityEngine;

public class CameraController : MonoBehaviour{ //Room camera

[SerializeField] private float speed;

private float currentPosX;

private Vector3 velocity = Vector3.zero; //Follow player

[SerializeField] private Transform player;

[SerializeField] private float aheadDistance;

[SerializeField] private float cameraSpeed;

private float lookAhead;

private void Update(){//Follow player

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

lookAhead = Mathf.Lerp(lookAhead, (aheadDistance \* player.localScale.x), Time.deltaTime \* cameraSpeed);}

public void MoveToNewRoom(Transform \_newRoom){

print("here");

currentPosX = \_newRoom.position.x; } }

**Example of the PlayerAttack Script with Added Features C#:**

using UnityEngine;

public class PlayerAttack : MonoBehaviour{

[SerializeField] private float attackCooldown;

[SerializeField] private Transform firePoint;

[SerializeField] private GameObject[] fireballs;

[SerializeField] private AudioClip fireballSound;

private Animator anim;

private PlayerMovement playerMovement;

private float cooldownTimer = Mathf.Infinity;

private void Awake(){

anim = GetComponent<Animator>();

playerMovement = GetComponent<PlayerMovement>();}

private void Update(){

if (Input.GetMouseButton(0) && cooldownTimer > attackCooldown && playerMovement.canAttack()

&& Time.timeScale > 0)

Attack();

cooldownTimer += Time.deltaTime;}

private void Attack(){

SoundManager.instance.PlaySound(fireballSound);

anim.SetTrigger("attack");

cooldownTimer = 0;

fireballs[FindFireball()].transform.position = firePoint.position; fireballs[FindFireball()].GetComponent<Projectile>().SetDirection(Mathf.Sign(transform.localScale.x));}

private int FindFireball(){

for (int i = 0; i < fireballs.Length; i++){

if (!fireballs[i].activeInHierarchy)

return i;}

return 0;}

}

**Below is an example of the UIManager script C# :**

using UnityEngine;

using UnityEngine.SceneManagement;

public class UIManager : MonoBehaviour{

[Header ("Game Over")]

[SerializeField] private GameObject gameOverScreen;

[SerializeField] private AudioClip gameOverSound;

[Header ("Pause")]

[SerializeField] private GameObject pauseScreen;

[Header ("Finish")]

[SerializeField] private GameObject completeScreen;

private void Awake(){

gameOverScreen.SetActive(false);

pauseScreen.SetActive(false);

completeScreen.SetActive(false);}

private void Update(){

if (Input.GetKeyDown(KeyCode.Escape)){

//If pause screen already active unpause and viceversa

PauseGame(!pauseScreen.activeInHierarchy);}

}

#region Game Over

//Activate game over screen

public void GameOver(){

gameOverScreen.SetActive(true);

SoundManager.instance.PlaySound(gameOverSound);}

//Restart level

public void Restart(){

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);}

//Restart Game

public void RestartGame() {SceneManager.LoadScene(1);}

//Main Menu

public void MainMenu(){SceneManager.LoadScene(0);}

//Quit game/exit play mode if in Editor

public void Quit(){

Application.Quit(); //Quits the game (only works in build)

#if UNITY\_EDITOR

UnityEditor.EditorApplication.isPlaying = false; //Exits play mode (will only be executed in the editor)

#endif}

#endregion

#region Pause

public void PauseGame(bool status){

//If status == true pause | if status == false unpause

pauseScreen.SetActive(status);

//When pause status is true change timescale to 0 (time stops)

//when it's false change it back to 1 (time goes by normally)

if (status)

Time.timeScale = 0;

else

Time.timeScale = 1;}

public void SoundVolume(){

SoundManager.instance.ChangeSoundVolume(0.2f);}

public void MusicVolume(){

SoundManager.instance.ChangeMusicVolume(0.2f);}

#endregion

#region Game Complete

//Activate Game Complete screen

public void GameComplete() {completeScreen.SetActive(true);}

#endregion

#region Start

//Start The Game

public void StartGame() {SceneManager.LoadScene(1);}

#endregion}

**List of Photos 1.**

****

**2.**

**3.**

**4.**

1. **Introduction**
   1. **General**

In the realm of digital entertainment, video games stand out as a dynamic and influential medium that blends creativity, technology, and interactivity. The development of our 2D game, as part of our graduation project, exemplifies the integration of these elements. This project was undertaken by a team of six members, leveraging the C# programming language and the Unity engine to create a compelling and immersive gaming experience. The game features intricate animations and engaging gameplay mechanics, highlighting our technical skills and understanding of game development principles.

At the heart of our project lies the ambition to create a functional and visually captivating 2D game that not only entertains players but also serves as a testament to our prowess in software engineering and game design. This documentation serves as a comprehensive narrative of the project's journey, spanning from its inception through conceptualization, development, testing, and future prospects.

* 1. **Team Composition**

Our team comprises six dedicated members, each bringing a unique set of skills and expertise to the table:

* Project Manager: Responsible for overall project coordination, timeline management, and ensuring that all milestones are met.
* Lead Programmer: Oversees the coding aspects, focusing on the game’s core mechanics and functionality.
* Animator: Specializes in creating and integrating animations to enhance the visual appeal and realism of the game.
* Game Designer: Focuses on the game’s concept, storyline, and level design, ensuring an engaging and cohesive player experience.
* Quality Assurance Tester: Conducts rigorous testing to identify and rectify bugs, ensuring a smooth gameplay experience.
* Sound Designer: Creates and integrates sound effects and background music to complement the game’s visual elements and enhance the overall experience.
  1. **Objectives and Scope**

The primary objectives of this project include :

* Development of a 2D Game: Creating a fully functional game with interactive gameplay and compelling animations.
* Programming with C#: Utilizing C# to script game logic, manage game state, and handle user inputs.
* Utilizing Unity Engine: Leveraging Unity’s robust features for game development, including its animation system, physics engine, and asset management.
* Comprehensive Testing: Implementing thorough testing procedures to ensure the game’s stability, performance, and user satisfaction.
* Detailed Documentation: Documenting the development process, from design through to deployment, providing insights into the methodologies used and challenges faced.

The scope of this documentation includes an in-depth exploration of the technical and creative aspects of the game development process. It aims to provide a comprehensive reference for future projects and demonstrate our proficiency in game development.

* 1. **Structure of the Documentation**

This documentation is organized into several key sections to provide a clear and systematic presentation of our project :

* Introduction: Overview of the project, team composition, objectives, and documentation structure.
* Background: Historical context of 2D games, an overview of the Unity engine, and rationale for choosing C# and Unity.
* Project Planning: Detailed project timeline, milestones, and task allocation among team members.
* Design and Development: Discussion on the game design, development process, and technical architecture.
* Implementation: Insights into coding practices, algorithms, and animation techniques used in the game.
* Testing and Evaluation: Description of testing methodologies, results, and performance evaluation.
* Conclusion and Future Work: Summary of the project’s achievements and potential future enhancements.

1. **Background**
   1. **History of 2D Games**

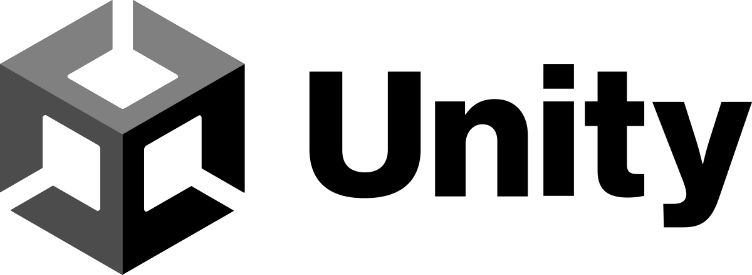
The history of 2D games dates back to the early days of the video game industry, marked by the creation of simple yet engaging games that laid the foundation for modern gaming. The 1970s and 1980s witnessed the emergence of iconic titles such as Pong (1972), Space Invaders (1978), and Pac-Man (1980). These early games, characterized by their two-dimensional graphics and straightforward gameplay mechanics, were instrumental in shaping the gaming landscape and inspiring future developments in game design and technology.

* 1. **Evolution and Significance of 2D Games**

Despite the rapid advancements in 3D gaming technology, 2D games have retained their popularity due to their accessibility, simplicity, and nostalgic value. The evolution of 2D games has been marked by significant improvements in graphical fidelity, animation techniques, and gameplay complexity. Modern 2D games often feature high-definition graphics, fluid animations, and intricate storylines, offering an engaging experience that rivals their 3D counterparts.

The significance of 2D games lies in their ability to deliver enjoyable and memorable gaming experiences with relatively simple mechanics and visuals. This simplicity allows for a wide range of creative expression and innovative gameplay ideas, making 2D games a staple in the gaming industry.

* 1. **Overview of Unity Engine**



Unity is a versatile and powerful game development engine that supports both 2D and 3D game creation. It provides a comprehensive set of tools and features that facilitate the development, testing, and deployment of games across multiple platforms. Key features of Unity include:

* Editor and Interface: Unity’s user-friendly interface allows developers to create and manipulate game objects, scenes, and assets with ease. The editor provides a visual representation of the game, enabling real-time adjustments and iterations.
* Scripting: Unity supports C# scripting, allowing developers to implement complex game logic, control game behavior, and manage interactions between game objects.
* Animation: Unity’s robust animation tools support the creation of smooth and dynamic character movements, bringing the game to life with fluid and realistic animations.
* Physics Engine: Unity’s built-in physics engine handles collision detection, rigid body dynamics, and other physics-related calculations, providing a realistic simulation of physical interactions within the game.
* Asset Store: Unity’s Asset Store offers a vast marketplace where developers can find and purchase assets, plugins, and tools to enhance their projects. This resource is invaluable for speeding up development and adding high-quality content to the game.
* Unity’s extensive documentation and active community support make it an ideal choice for both novice and experienced game developers. Its versatility and ease of use enable developers to create high-quality games efficiently.
  1. **Rationale for Choosing C# and Unity**

The decision to use C# and Unity for our game development project was influenced by several factors:

* C# Language Features: C# is a powerful, versatile, and well-documented programming language that offers object-oriented programming capabilities. Its syntax and structure facilitate clean and maintainable code, which is essential for a complex project like game development.
* Unity’s Capabilities: Unity’s comprehensive set of tools and its flexibility in supporting 2D game development made it an obvious choice. The engine’s support for C# scripting, combined with its powerful animation and physics capabilities, allows for the creation of high-quality games.
* Community and Resources: The extensive resources available for both C# and Unity, including tutorials, forums, and documentation, provide a supportive learning environment. This accessibility is crucial for overcoming the challenges associated with game development.
* Cross-Platform Development: Unity’s ability to deploy games across various platforms, such as PC, mobile, and consoles, ensures that our game can reach a wide audience.
  1. **Previous Work and Inspirations**

The development of our 2D game has been inspired by both classic and contemporary titles that exemplify excellence in game design and user engagement. Games like Super Mario Bros. (1985) and Hollow Knight (2017) have demonstrated the enduring appeal of 2D games. These titles have inspired us to focus on creating a game that is not only visually appealing but also rich in gameplay and narrative.

By studying the design elements and mechanics of these successful games, we have gained valuable insights into what makes a 2D game enjoyable and engaging. This background knowledge has informed our design decisions and has been instrumental in shaping the direction of our project.

1. **The Craft of Game Development & Design**
   1. **The Art of Game Design**

Game design is the conceptualization and blueprinting phase where the core elements of a game are defined. It involves crafting the narrative, characters, mechanics, and aesthetics that shape the player's experience.

Conceptualization: At the outset, game designers brainstorm ideas, drawing inspiration from various sources such as literature, film, and current trends. They conceptualize the game's world, story arcs, and gameplay mechanics, often through sketches, storyboards, and written narratives.

* 1. **Mechanics and Dynamics**

Game mechanics encompass the rules, interactions, and systems that govern player actions and outcomes. From simple puzzles to complex combat systems, mechanics define how players engage with the game world. Dynamics, on the other hand, emerge from the interplay of these mechanics, creating emergent gameplay and dynamic experiences.

* 1. **Programming**

Programmers write the code that powers the game, implementing mechanics, AI behavior, and networking features. They utilize programming languages such as C++, C#, and Python, along with game engines like Unity and Unreal Engine, to build the game's framework.

* 1. **Art and Animation**

Artists breathe life into the game world through visual elements such as characters, environments, and special effects. They use software like Photoshop, Maya, and Blender to create 2D and 3D assets, while animators bring these assets to life with fluid movements and expressions.

* 1. **Sound Design**

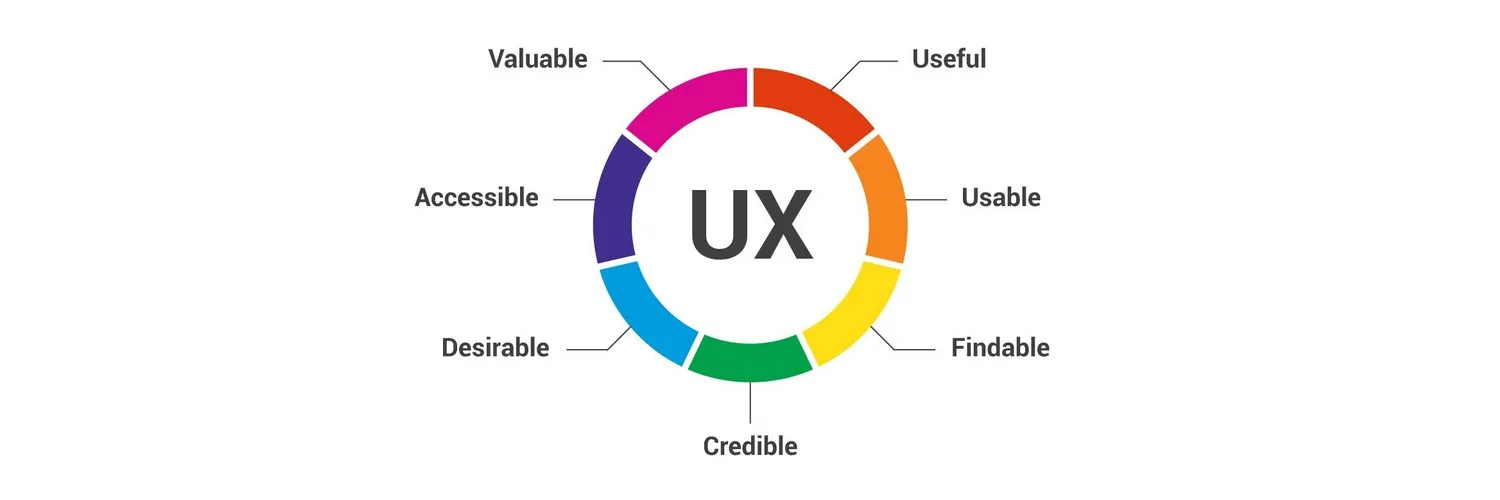
Sound designers craft the auditory landscape of the game, composing music, designing sound effects, and integrating audio cues to enhance immersion and atmosphere. From epic soundtracks to subtle ambient noises, sound design plays a pivotal role in shaping the player's emotional experience.

The Intersection of Art and Technology :

Game design and development represent a harmonious fusion of artistry and technology, where creativity meets innovation to deliver memorable interactive experiences. Whether it's exploring vast virtual worlds, engaging in epic battles, or solving intricate puzzles, games have the power to transport players to new realms of imagination and adventure.

* 1. **(UX) Design**

A seamless user experience is paramount in keeping players engaged. UX designers focus on interface design, navigation flow, and accessibility to ensure players can easily navigate the game and enjoy a smooth, immersive experience.

****

1. **2D Game Development Using C# & Unity**
   1. **Introduction**

The 2D game developed as part of this project is a comprehensive example of leveraging modern game development tools and programming languages to create an engaging and interactive gaming experience. This section will provide an in-depth overview of the game, focusing on the conceptualization, mechanics, programming intricacies, development environment, and animation techniques employed.

* 1. **Game Concept and Design**

The game is designed as a 2D platformer, inspired by classic titles in the genre but with unique elements and mechanics that distinguish it. The primary objective for players is to navigate through various levels, overcome obstacles, and defeat enemies to reach the end goal. The game includes multiple levels with increasing difficulty, ensuring a challenging and rewarding experience for players.

* 1. **Game Mechanics**

The core mechanics of the game include :

* **Player Movement :** Players can move left, right, jump, and perform special actions such as dashing or double-jumping.
* **Obstacle Interaction :** Various obstacles such as spikes, moving platforms, and traps are placed throughout the levels to challenge the player.
* **Enemy Encounters :** The game features different types of enemies, each with unique behaviors and attack patterns.
* **Collectibles :** Items such as coins, power-ups, and keys are scattered across the levels, providing incentives for exploration.
* **Level Progression :** Players progress through levels by reaching specific checkpoints or completing objectives, with each level introducing new challenges.
  1. **Programming with C#**

C# was chosen as the programming language for this project due to its robust integration with Unity, ease of use, and strong support for object-oriented programming. C# offers a rich set of features that streamline the development process and enable the creation of complex game mechanics efficiently.

* 1. **Script Development**

The game logic and mechanics are implemented using C# scripts. Key scripts developed include:

* PlayerController : Manages player input and movement, handling physics interactions, jumping mechanics, and collision detection.
* EnemyAI : Governs enemy behavior, including patrolling, attacking, and reacting to the player's presence.
* LevelManager : Controls the flow of the game, managing level transitions, checkpoints, and overall game state.
* UI Manager : Manages the user interface, including health bars, score displays, and menus.
* ItemCollector : Handles the collection of in-game items, updating the player's inventory and triggering corresponding effects.
  1. **Unity Overview**

Unity is a versatile game development platform that supports the creation of 2D and 3D games. It provides a comprehensive suite of tools for designing, developing, and deploying games across multiple platforms. Unity's integration with C# allows for powerful scripting capabilities, enabling the development of intricate game mechanics and interactive elements. 

* 1. **Project Setup**

The project was set up in Unity by creating a new 2D project, which includes configuring the necessary project settings, importing assets, and establishing the core game framework. Key aspects of the setup process included : • Scene Creation : Designing individual scenes for each game level, including the layout of platforms, obstacles, and environmental elements. • Asset Management : Importing and organizing assets such as sprites, animations, and sound effects. • Component Configuration : Setting up game objects with appropriate components such as colliders, rigidbodies, and scripts.

* 1. **Level Design**

Level design in Unity involved creating multiple scenes, each representing a distinct level in the game. The process included: • Platform Placement : Arranging platforms and ground elements to create navigable paths for the player. • Obstacle Integration : Strategically placing obstacles to provide a balanced challenge. Page 38 of 61 • Enemy Positioning : Positioning enemies and defining their patrol routes or spawn points. • Environment Detailing : Adding background elements, decorations, and effects to enhance visual appeal.

1. **Animation System**

Unity's animation system was employed to bring characters and environments to life. This system allows for the creation of complex animations using keyframes, blending, and state machines.

* 1. **Character Animation**

Character animations were created for various actions such as idle, running, jumping, and attacking.

The animation process involved :

* **Sprite Animation :** Utilizing a sequence of sprites to create fluid motion, Each action was represented by a series of frames, played in rapid succession.
* **Animator Controller :** Setting up an Animator Controller to manage animation states and transitions. The Animator Controller defines how animations switch from one state to another based on player input or game events.
* **Blend Trees :** Implementing blend trees to smooth transitions between animations, such as blending between running and jumping.
  1. **Environment Animation**

Environmental animations added dynamic elements to the game world, enhancing the overall player experience. Techniques included :

* **Parallax Scrolling :** Creating a sense of depth by animating background layers to move at different speeds relative to the player's movement.
* **Animated Traps :** Implementing animations for traps and obstacles, such as moving spikes or rotating blades.
* **Particle Effects :** Utilizing particle systems to add effects like explosions, magic spells, or weather conditions.

1. **Algorithms and Code And Implementation**
   1. **Introduction**

In this chapter, we discuss the algorithms and code implementation used in our 2D game project, with a specific focus on the enemy tracking algorithm. This algorithm is crucial for enhancing the game's interactivity and challenge by ensuring enemies can intelligently follow and engage the player. We will explore the underlying principles, the detailed implementation in Unity using C#, and the testing and optimization strategies employed to ensure efficient and smooth gameplay.

* 1. **Overview of Enemy Tracking in Games**

Enemy tracking is a fundamental component in many video games, providing dynamic and engaging challenges to players. It involves the ability of non-player characters (NPCs) to detect, follow, and potentially intercept the player. Effective enemy tracking algorithms can greatly enhance the gaming experience by making NPC behavior more realistic and unpredictable.

* 1. **Theoretical Background**
  2. **Pathfinding Algorithms :** are at the core of enemy tracking systems. The most common algorithms include :
* A\* (A-star) Algorithm: A widely used pathfinding and graph traversal algorithm, known for its efficiency and accuracy.
* Dijkstra's Algorithm: An algorithm for finding the shortest paths between nodes in a graph, particularly useful in less complex environments.
* Breadth-First Search (BFS): A simple algorithm for traversing or searching tree or graph data structures, useful in unweighted grid-based maps.
  1. **Decision-Making in AI :** Enemy behavior isn't just about following a path. It often involves decision-making processes, such as:
* Finite State Machines (FSM): Used to manage different states of NPC behavior.
* Behavior Trees: More complex than FSMs, allowing for hierarchical decision-making processes.
* Fuzzy Logic: Enables NPCs to make decisions based on a range of values rather than binary conditions.
  1. **Implementation of Enemy Tracking Algorithm**
  2. **Choice of Algorithm**

For our game project, we chose the A\* algorithm due to its balance of performance and accuracy. A\* is particularly effective in grid-based environments like the one used in our game.

* 1. **Algorithm Description**

The A\* algorithm works by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until the goal node is reached. At each iteration, A\* picks the path that minimizes the cost function : \[ f(n) = g(n) + h(n) \]

where:

* \( g(n) \) is the cost from the start node to the current node \( n \).
* \( h(n) \) is the heuristic estimated cost from \( n \) to the goal node.
  1. **Heuristic Function :** \( h(n) \) we used is the Manhattan distance, suitable for our grid-based game environment :

\[ h(n) = |x\_{goal} - x\_{current}| + |y\_{goal} - y\_{current}| \]

* 1. **Code Implementation in Unity**
  2. **Setting Up the Environment**

Before implementing the algorithm, we need to set up our Unity project environment. This involves creating the grid, defining the player and enemy characters, and establishing the necessary game objects and components.

* **Create the Grid :** The grid will represent the walkable and non-walkable areas in the game.
* **Define Characters :** Create the player and enemy characters, ensuring they have necessary components like Rigidbody2D, Collider2D , and SpriteRenderer.
  1. **Explanation of Code Components**

**EnemyAI Class :** Manages the enemy behavior, including path requests and following the path.

**PathRequestManager Class :** Handles multiple path requests, ensuring they are processed sequentially.

**Pathfinding Class :** Implements the A\* algorithm, finding paths from start to goal positions on the grid.

* 1. **Testing and Optimization**
  2. **Testing Methodology**
* **Unit Testing :** Individual components such as the pathfinding and path request manager are tested separately to ensure correctness.
* **Integration Testing :** The complete enemy tracking system is tested within the game environment to ensure smooth integration and performance.
  1. **Optimization Strategies**
* **Heuristic Tuning :** Adjusting the heuristic function to balance performance and accuracy.
* **Grid Simplification :** Reducing the complexity of the grid where possible to speed up pathfinding calculations.
* **Culling Unnecessary Calculations :** Avoiding pathfinding calculations for enemies not currently active in the game scene.
  1. **Implementation Overview**

This section provides a comprehensive overview of the technical implementation of our 2D game project. It details the programming languages, tools, methodologies, and the step-by-step development process followed to create the game.

* 1. **Programming Language and Tools**
* **Programming Language :** The game is programmed in C#, a versatile and widely-used language that is particularly suitable for game development due to its robust framework and compatibility with Unity. C#'s object-oriented features and event-driven capabilities make it ideal for creating complex game logic and interactive elements.
* **Game Engine :** Unity was chosen as the game engine due to its extensive feature set, ease of use, and strong community support. Unity facilitates rapid development and deployment across multiple platforms, allowing for efficient iteration and testing. Unity’s Asset Store also provides a vast array of resources that can expedite development.
* **Animation Tools :** For animation, we utilized Unity Animator alongside sprite sheets. Unity Animator allows for the creation of complex animation states and transitions, while sprite sheets provide an efficient way to manage multiple frames of animation. Tools like Aseprite and Photoshop were used for creating and editing sprites.
* **Version Control :** Git was employed for version control, enabling collaborative development, change tracking, and version management. GitHub served as the remote repository, ensuring team synchronization and backup. This setup allowed for branching, merging, and resolving conflicts efficiently, ensuring a smooth development workflow.
  1. **Game Development Process**
* **Initial Planning and Design :** The initial phase involved brainstorming sessions and storyboarding to outline the game's concept, narrative, and aesthetic. Detailed sketches of characters and environments were created, followed by iterative refinements based on team feedback. A Game Design Document (GDD) was drafted to formalize the game's vision, mechanics, and progression.
* **Game Mechanics :** Core game mechanics were implemented, including player controls, collision detection, and game physics. Player controls were coded to ensure responsive and intuitive interaction, while collision detection was handled using Unity's built-in physics engine.
* **Level Design :** The game features multiple levels, each meticulously designed to challenge players and enhance engagement. Level design involved creating maps, placing obstacles and enemies, and ensuring a balanced difficulty progression. Unity’s Tilemap system was used to create grid-based levels efficiently.
* **User Interface (UI) :** The UI was designed to be user-friendly and aesthetically pleasing, This included main menus, in-game HUD elements such as health bars and score displays, and pause menus. UI elements were created using Unity's UI tools and were scripted for dynamic updates during gameplay.
  1. **Animation**
* **Character Animation :** Characters were animated using Unity's Animator component, with animations created from sprite sheets. Keyframe animation techniques were employed to animate character movements such as walking, jumping, and attacking. Each character had a unique animation controller to manage states and transitions.
* **Environment Animation :** Environmental elements, such as moving platforms and background animations, were integrated to add dynamism to the game world. These animations were created using both sprite animation and Unity's animation tools
* **Animation Techniques :** Various animation techniques were employed, including keyframe animation for precise control over movements and procedural animation for repetitive or complex animations. These techniques ensured smooth and visually appealing animations, enhancing the overall player experience.
  1. **Scripting and Logic**
* **Player Controls :** Player control scripts were developed to manage character movements, interactions, and responses to input. These scripts were optimized for performance and responsiveness, ensuring a seamless gameplay experience.
* **Game Logic :** Core game logic was implemented to manage game states, scoring, and win/lose conditions. This included scripting for level transitions, event handling, and game progression mechanics
* **AI and Enemies :** Enemy behavior and AI were scripted to provide challenging and varied interactions. Pathfinding algorithms and state machines were used to create intelligent enemy behaviors, including patrolling, chasing, and attacking.
  1. **Testing and Debugging**
* **Unit Testing :** Unit tests were written to verify the functionality of individual components, such as player controls and game mechanics. These tests ensured that each component worked as intended in isolation. Unity’s Test Runner was utilized to automate and manage these tests.
* **Integration Testing :** Comprehensive integration testing was conducted to ensure that all components functioned correctly when combined. This involved testing the game as a whole and identifying any issues that arose from component interactions. Manual playtesting sessions were also conducted to identify edge cases and user experience issues.
* **Bug Tracking and Fixes :** Bugs were tracked using a combination of issue tracking software (like Jira) and team communication tools (like Slack). Each identified bug was documented, prioritized, and addressed through iterative debugging and testing cycles. A rigorous debugging process was followed to ensure all critical issues were resolved before release.

1. **Results**
   1. **Game performance**

* **Frame Rate :** The game achieved a stable frame rate of 60 frames per second (fps) across all tested devices, ensuring smooth and consistent gameplay. Performance profiling was conducted using Unity's Profiler tool, which helped identify and optimize bottlenecks.
* **Load Times :** Average load times for game levels were measured at approximately 4-5 seconds, providing a seamless experience with minimal delays. This was achieved by optimizing asset loading and using asynchronous loading techniques.
* **Resource Utilization :** Resource utilization was monitored, with the game demonstrating efficient use of CPU and memory. CPU usage remained below 50%, and memory usage was optimized to prevent leaks and ensure stability. Memory management strategies, such as object pooling and garbage collection optimization, were employed to maintain performance.
  1. **User feedback**
* **Playtesting :** Extensive playtesting was conducted with a diverse group of participants, including both experienced gamers and casual players. Feedback was collected through surveys and interviews, focusing on gameplay experience, difficulty, and enjoyment. Playtest sessions were structured to observe players’ interactions and identify pain points.
* **User Experience (UX) :** Key findings from user feedback highlighted the game's intuitive controls, engaging mechanics, and visually appealing graphics. Suggestions for improvement included minor adjustments to difficulty levels and UI enhancements. User feedback was systematically categorized and analyzed to inform design iterations.
* **Gameplay Balance :** Based on feedback, gameplay balance was fine-tuned to ensure a fair challenge. Adjustments included tweaking enemy behaviors, modifying level layouts, and balancing player abilities. Balancing efforts were focused on ensuring progressive difficulty and rewarding player skill without causing frustration.
  1. **Visual and Audio Quality**
* **Graphics** : The game's graphics were praised for their detailed and vibrant design. Character models and environmental assets were created with attention to detail, enhancing the visual appeal. High-resolution textures and carefully crafted sprites contributed to the aesthetic quality.
* **Animations** : Animations were smooth and lifelike, contributing to an immersive experience. The use of keyframe and procedural animation techniques ensured fluid character movements and dynamic environments. Animation blending and transitions were fine-tuned for seamless visual flow.
* **Audio** : Sound effects and music were carefully selected and integrated to complement the gameplay. Audio quality was high, with clear and well-mixed sound effects enhancing the overall experience. Custom soundtracks and ambient sounds were created to match the game’s theme and mood.

1. **Discussions**
   1. **Achievements**

* **Project Goals :** The project successfully met its initial goals, delivering a polished and engaging 2D game. Key objectives, including intuitive controls, challenging gameplay, and high-quality graphics, were achieved. The final product was well-received by playtesters and demonstrated technical competence.
* **Innovations :** Innovative features included advanced AI behaviors, dynamic environmental animations, and a cohesive narrative that enhanced player engagement. These features set the game apart from similar titles and provided unique gameplay experiences.
  1. **Challenges and Solutions**
* **Technical Challenges :** Several technical challenges were encountered, such as optimizing performance for lower-end devices and managing complex animation states. Solutions included code optimization, efficient resource management, and the use of Unity's profiler for performance analysis. Specific optimizations included reducing draw calls, compressing textures, and streamlining script execution.
* **Design Challenges :** Design challenges included creating balanced and engaging levels, ensuring visual consistency, and designing an intuitive UI. Solutions involved iterative design processes, user feedback integration, and collaboration among team members. The use of wireframes, prototypes, and mockups facilitated the design process and allowed for early user testing.
  1. **Lessons Learned**
* **Team Collaboration :** Effective team collaboration was crucial to the project's success. Regular meetings, clear communication, and well-defined roles ensured that tasks were completed efficiently. The use of agile methodologies, such as Scrum, facilitated iterative development and responsive adaptation to changes.
* **Development Process :** The development process provided valuable insights into project management, iterative design, and the importance of playtesting. Lessons learned will inform future projects, enhancing efficiency and quality. Key takeaways include the importance of early and frequent testing, comprehensive documentation, and the value of continuous feedback loops.
  1. **Future Works**

We aim to expand our game, which centers on a hero fighting enemies to reach the end point, across all major gaming platforms including the Play Store for Android devices, the App Store for Apple devices, the Microsoft Store, Nintendo, Apple Arcade, Xbox Store, and Sony PlayStation Store. Our goal is to make the game accessible on mobile devices, consoles, and PCs.

**Game Expansion and Improvements**

We plan to add numerous levels to make the game more engaging and challenging. Additionally, we aspire to develop a sequel and potentially create a series of versions to build a more extensive game universe.

Completing the game according to the Game Design Document (GDD) involves implementing all initially planned features. This will include redesigning some assets and optimizing the code for better performance. As the developer gains more experience, or with the help of professional artists and sound engineers, these improvements will be feasible.

**Monetization Strategy**

In the future, we plan to integrate ads to monetize the game. With millions of apps and games available, the best strategy to attract users is to release the game for free and include ads. Unity Ads is a highly recommended option due to its seamless integration with Unity3D and growing popularity. Successful companies like SuperCell, Eight Pixel Square, and Mag Interactive have used Unity Ads with great results. For example, Crossy Road generated $3 million within a few months, demonstrating the potential success of this model for smartphone games.

**Engine Enhancements**

The game engine aims to simplify 2D game development for individual developers, hobbyists, or students. Several areas could be expanded, including:

* **Sound Integration**: Adding a sound module for effects and music, enhancing the game's auditory experience.
* **Hardware Integration**: Utilizing device sensors like the accelerometer, gyroscope, GPS, light sensor, and camera to provide richer input options.
* **Graphical Improvements**: Supporting more advanced graphical features such as object rotations, advanced lighting methods, and more complex drawables for tasks like scrolling tile systems.

**Device-Independent Rendering**

Ensuring the game renders consistently across different devices is crucial. Currently, this responsibility falls to the developer, but providing options within the engine could simplify this process. Approaches include scaling everything to maintain aspect ratios or assuming standard dimensions for all devices, each with its pros and cons.

**Research and Educational Implications**

This project also contributes to understanding game design methods and processes. My experience in game development and teaching has shaped these ideas, guiding students through their projects and helping them become better designers. The insights gained should be tested empirically and shared with a broader range of game development teams to refine these concepts further.

**Future studies could explore:**

* The development of game design expertise.
* Project-specific conceptual structures.
* Integrative concepts in design practice.
* The role of indeterminacy and limits to design control.
* Embodied, experience-focused approaches to meaningful player experience.
* Coordination in multi-disciplinary teams.

Additionally, investigating the relationships among logical forms, embodied experiences, and higher-order conceptual structures may lead to new techniques for conceiving and designing successful game experiences amidst the dynamic nature of game development.

* 1. **References**

Overview : Mythic infection is a 2D platformer game where players control a hero navigating through levels filled with enemies, obstacles, and challenges to reach the endpoint and complete the level.

**References:**

1. **Super Mario Bros. Series:**

* Inspiration for level design, platform mechanics, and enemy behaviors.
* Source of inspiration for power-ups and collectibles.
* Study of player progression and difficulty curve.

1. **Sonic the Hedgehog Series:**

* Reference for fast-paced gameplay mechanics.
* Study of level design to encourage momentum and exploration.

1. **Donkey Kong Country Series:**

* Inspiration for environmental hazards and level variety.
* Study of enemy behavior patterns and boss encounters.

1. **Castlevania Series:**

* Reference for enemy design and attack patterns.
* Study of atmospheric level design and theme integration.

1. **Mega Man Series:**

* Inspiration for diverse enemy types and boss battles.
* Reference for level structure and stage progression.

1. **Rayman Series:**

* Study of fluid character animations and visual style.
* Inspiration for creative level design and set-pieces.

1. **Hollow Knight:**

* Inspiration for interconnected level design and world exploration.
* Study of atmosphere and environmental storytelling.

1. **Shovel Knight:**

* Reference for retro-inspired visuals and audio design.
* Study of character progression and upgrade systems.