Designing a 2D Video Game to Aid Children with Learning Disabilities

by

Sabil Sadat Zahir
21301621

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

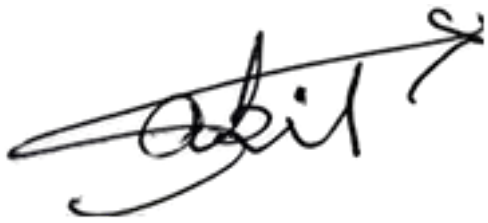
Department of Computer Science and Engineering
Brac University
October, 2025.

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:



Sabil Sadat Zahir

21301621

Approval

The thesis/project titled "Designing a 2D Video Game to Aid Children with Learning Disabilities" submitted by

1. Sabil Sadat Zahir (21301621)

of Summer, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science in October 9th, 2025.

Examining Committee:

Supervisor:
(Member)

Arif Shakil

Lecturer
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi

Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

This project aims to address the growing need for accessible, engaging educational tools to support children with learning disabilities (such as dyslexia and dyscalculia) and attention challenges (such as ADHD) by designing a short video game demo as a proof of concept. The goal of this project is to integrate game design principles with cognitive learning theories to enhance cognitive skills, attention, and emotional resilience among such children.
This project draws heavily on proven evidence from research papers, news articles, and studies on the impact of video games on learning disabilities. The game demo, inspired by the popular game series “Pokémon”, incorporates educational challenges in fifth-grade-level Mathematics, English Grammar, and Science, with immersive and intuitive gameplay, tailored to individual learning needs through adaptive difficulty and visual-spatial cues. The demo was subsequently shared with a group of testers, who provided qualitative feedback that assessed the game’s effectiveness in improving focus, problem-solving, and emotional regulation, with positive responses highlighting how the core quiz-based battle system mechanic provided a rewarding learning experience. This feedback ensured that the project aligns with the pre-existing research and studies on the topic. The project contains the base gameplay systems that can be expanded upon with future development and resources.

Keywords: Video games, Inclusive education, Learning disabilities, ADHD, Cognitive development, Emotional strength, 2D role-playing game, Unity, C, Educational technology

Acknowledgement

Special thanks to my Supervisor, Arif Shakil sir, for his kind support and advice throughout the entire process of making this project.
And I would like to thank my parents for their continuous love and support, as well as the teachings and values that they have instilled in me.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background

The following study was conducted based on some existing research papers, news reports and articles, as well as some scientifically proven concepts. However, at the present scope, the project serves as a proposal for the concept rather than an extensive proof.

1.2 Rational of the Study or Motivation

Children with learning disabilities often experience difficulty maintaining attention and motivation in traditional classroom settings. Conventional teaching methods may not cater to their cognitive needs or processing styles, resulting in disengagement and reduced learning outcomes. While educational games exist, few are specifically designed to address the attention challenges faced by children with conditions such as ADHD or dyslexia.
This research addresses that gap by introducing a modified, quiz-integrated game environment that merges play and pedagogy. By combining familiar gaming structures with targeted educational activities, the project seeks to demonstrate how game-based interventions can transform passive learning into interactive engagement.
The study is significant because it not only contributes to the academic discourse on inclusive learning design but also offers a prototype that could be adapted for classrooms, therapy programs, and home-based learning. It bridges the gap between entertainment and education, emphasizing the potential of serious games to enhance focus, motivation, and cognitive development in neurodiverse learners.

1.3 Problem Statement

Many existing tools fail to sustain the attention and motivation of children with learning disabilities. These learners often require multisensory, adaptive, and reward-based environments that reinforce concentration and gradual skill development.
The lack of games specifically designed to combine cognitive stimulation with subject-based learning creates a gap in accessible educational resources. Therefore, the core problem this study seeks to address is:

How can game design principles, particularly turn-based mechanics,

adaptive challenges, and interactive feedback, be utilized to create an educational game that enhances focus, engagement, and conceptual learning for children with learning disabilities?

By investigating this question, the study aims to explore the pedagogical and practical potential of gaming as a tool for education.

1.4 Objective

The primary objective of this study is to design, develop, and evaluate an educational game demo that supports attention and learning in children with learning disabilities. The demo adapts core mechanics from standard role-playing videogames to create a learning experience that integrates academic quizzes within an interactive gameplay environment. The specific objectives of the project are as follows:

- To learn and apply the fundamental principles of game development through the creation of a functional prototype.
- To modify traditional turn-based combat systems into quiz-based interactions that reinforce learning in subjects such as Math, English, and Science.
- To evaluate the effectiveness of the game in enhancing focus, engagement, and motivation among players.
- To collect and analyze user feedback to identify design improvements and educational outcomes.
- To demonstrate how educational game design can serve as a supportive tool for children with attention and learning difficulties.

The outcomes of this study aim to contribute both to the field of educational technology and to the growing body of research supporting the use of serious games in inclusive learning environments.

1.5 Methodology in Brief

This study employed a **design-based research** approach, combining elements of game development, user-centered design, and educational evaluation to explore how video games can support children with learning disabilities. The methodology consisted of two primary objectives: first, to gain a practical understanding of fundamental game development principles, and second, to design and implement a prototype game based on that understanding that provides an intuitive and engaging learning experience.
To achieve these goals, a modified version of the popular game series *Pokémon*, developed by Game Freak and published by Nintendo, was selected as the foundation for development. The choice of this game framework was motivated by several factors:

- **Comprehensive Mechanics:** The core mechanics of *Pokémon*, such as player movement, turn-based combat, and extensive dialogue systems, provide a solid foundation for understanding essential programming and game design concepts.

- **Child-Centered Appeal:** *Pokémon*'s accessibility, wide range of skill-based objectives, and cultural popularity among children made it an ideal model for creating an educational game targeted at younger audiences.
- **Adaptable Combat System:** The turn-based combat system, where players select attacks in alternating turns, was reimagined as a multiple-choice quiz mechanism. Players must answer subject-based questions correctly to perform attacks, integrating academic assessment into gameplay.
- **Educational Reinforcement through Game Types:** The elemental type system in *Pokémon* (e.g., Fire, Water, Grass) was repurposed to represent academic subjects such as Science, Grammar, and Mathematics. This approach provided both gameplay variety and motivation for players to explore different learning domains.

Data Collection: To evaluate the game's educational potential and usability, a survey was conducted among test players. Participants were asked to assess the game's learning effectiveness, engagement, and usability through Likert-scale and open-ended questions.
Data Analysis: Quantitative responses were analyzed using descriptive statistical methods, while qualitative feedback was thematically reviewed to identify recurring patterns related to engagement, focus, and learning outcomes.
Through this iterative design and evaluation process, the project explored how adapting familiar commercial game mechanics into an educational framework can enhance focus, motivation, and learning for children with attention or learning difficulties.

1.6 Scopes and Challenges

The primary challenge was that game development itself is a vast subject on its own, and requires the study and understanding of concepts beyond the traditional Computer Science topics provided in local curricula in Bangladeshi Universities.
Different game development engines require different scripting languages and an understanding of different tools. Moreover, a big video game project usually requires much more than just the ability to code, as artists and designers are required to create "assets" used in the games, writers are required for storytelling, and game directors are required to provide overall management of the development process. However, through accessing online learning resources, free assets and by applying basic coding concepts, I was able to create a very small-scale game demo. Considering time constraints, limited resources, the scope of the project could not be extended beyond a small explorable level with a basic gameplay loop.

1.7 Team Overview

The project was developed entirely by me as a solo developer. Project Supervisor provided direction and advice throughout the process.

1.8 Key Learnings and Insights

By working on this project, I was able to learn the basic fundamentals of game development and how the tools of a game engine can be enhanced through original coding

scripts.
I was also able to apply an idea, which I hope can have a great impact on society, particularly towards the marginalised children who suffer from learning and attention deficits through no fault of their own. Though I wish I could have made the project much more expansive and better, I believe I was able to lay the foundation to develop the concept into a fully actualised video game in the future that can be released for public access.

Chapter 2

Literature Review

2.1 Preliminaries

This section outlines foundational concepts for using video games to support children with learning disabilities, such as ADHD, dyslexia, and autism spectrum disorder. Gamified learning enhances cognitive, social, and emotional skills [5]. Serious games target specific goals, like improving attention or reading skills [2]. Howard Gardner's Theory of Multiple Intelligences suggests varied cognitive strengths, such as linguistic or logical-mathematical skills. Games can leverage these to address deficits. Executive function includes processes like attention, memory, and task-switching, critical for children with ADHD [4]. Emotional resilience, the ability to cope with failure, is fostered through trial-and-error mechanics [3].

2.2 Review of Existing Research

Recent studies highlight video games as effective educational tools for children with learning disabilities. These games provide immersive, customizable environments unlike traditional methods. Thomas and Woodyatt (2020) describe EndeavorRX, an FDA-approved game for ADHD. It improves focus and self-regulation by targeting executive functions [4]. Strong et al. (2011) studied Fast ForWord, a program for dyslexia. It enhances reading through exercises improving auditory discrimination and processing speed [1]. Khan (2021) notes that games like Unravel and Moving Out improve motor skills, communication, and organization by breaking tasks into manageable steps [5]. Noble (2020) discusses Cyberchase and Odd Squad, which boost math skills and resilience. Features like untimed play in Railway Hero promote inclusivity [3]. García-Redondo et al. (2019) found that a serious game using the Tree of Intelligences method improved visual attention in 44 students with ADHD after 28 sessions [2].

2.3 Summary of Key Findings

The literature consistently shows that game-based learning improves cognitive and emotional outcomes for children with learning disabilities. EndeavorRX achieves 30 percent focus gains for ADHD [4]. Fast ForWord supports rapid reading skill development in dyslexia [1]. Games also enhance motor, communication, and organizational skills. Interactive environments with immediate feedback and personalized pacing outperform

traditional methods, reducing anxiety and boosting motivation [6]. These findings support a quiz-based battle system to engage cognitive and emotional centers. The system could scale with progressive difficulty and broader subject coverage. However, gaps exist in longitudinal studies and applications in developing regions like Bangladesh. This project addresses these by creating a low-cost, accessible prototype.

Chapter 3

Requirements, Impacts and Constraints

3.1 Final Specifications and Requirements

This project was developed using Unity, a free-to-use and popular game engine created by Unity Technologies in 2005. The game engine utilizes C# programming language for scripting (i.e, for defining the behaviour and logic of game objects and components within a Unity project). The game is two-dimensional and sprite-based, viewed from an isometric point of view. As a result, it does not require any high-end specifications to run, and the only requirements are a Windows Operating System with at least some integrated graphics processing unit. All of the assets in the game were imported from free online resources and are mostly copyright-free (except for the Pokémon sprites and identical UI elements).

3.2 Societal Impact

If developed into a full game with original assets and themes, i.e., a new Intellectual Property (IP), it could potentially be used as a teaching tool for children with learning disabilities. It could help them learn in a more impactful way, bridging the gap between them and their peers.
Furthermore, it can also be used as an entertaining way to learn for all kinds of children as well, as it's meant to be an accessible product that aims to teach and cultivate knowledege.

3.3 Ethical Issues

The demo was made as a "proof-of-concept" and utilized assets and gameplay concepts from an already existing IP, that being the *Pokémon* franchise. To avoid copyright infringement and legal issues, the base of this project must be developed into a new concept with modified mechanics and characters.

3.4 Project Management Plan

The project was made sporadically over the course of 5-6 months, following a strategy similar to the *Kanban* process or model.

3.5 Economic Analysis

The project was made entirely using free assets and a free game engine. However, to make it into a full-fledged game, a team could be hired for separate tasks, such as artists and character designers for crafting the sprites and graphical elements of the game, a game director to manage the production of the game, a team of programmers to enhance the gameplay experience, a marketing team to advertise the finalized product, and so on.

Chapter 4

Proposed Methodology

4.1 Design Process or Methodology Overview

For this project, two objectives had to be completed. Firstly, I had to learn the basics of game development principles, and secondly, I had to craft a game that could utilize those principles to provide an intuitive learning experience.
To achieve these two objectives, I decided to create a modified version of Pokémon, a game series developed by Game Freak and published by Nintendo. The reasons why using this game as a base served the purpose of this project include:

- The fundamental gameplay mechanics of Pokémon, such as the movement, turn-based combat system, expansive dialogue and interaction systems, etc., all provide the quintessential groundwork and foundation that can be applied to any game, and thus, understanding how these mechanics can be applied and coded was an ideal way to learn game development. Furthermore, these aforementioned mechanics are genre-standard, meaning the code and principles for turn-based combat, dialogue, movement and so on is neither created nor copyrighted by the developers of Pokémon, and can be repurposed for any game project.

- The game has a mass appeal for children due to its inherent simplicity and accessibility, a wide variety of goals for different skill levels, and the broad cultural appeal of its characters and franchise. As the target demographic of this study is children, creating a game similar to Pokémon would be ideal.

- Pokémon games contain turn-based combat mechanics, meaning the player can attack an enemy and vice versa based on turns. The player has a selection of abilities they can use against the enemy. This system provides a framework that can be redesigned as a multiple choice quiz game.

- The combat system also contains stat-based effects, with different categories of Pokémon having benefits and weaknesses against opposing categories. This system could be reverse-engineered so that those Pokémon types (i.e., Fire, Water, Grass, and so on) could each represent a topic or subject (i.e., Science, History, Grammar, or Mathematics). Not only would this provide a varied gameplay experience, but it would also incentivize the player to learn different topics to progress in the game.

4.2 Preliminary Design or Design (Model) Specification

The project was finished sporadically over the course of approximately 6-7 months. While a strict design model was not implemented, the design process could be classified as similar to the *Kanban model*. This is a Japanese visual workflow method that focuses on focuses on step-by-step and continuous improvement and adaptation without conforming to strict deadlines. The method allows for focusing on individual tasks and tracking progress via visual tools such as a Kanban board.

4.3 Implementation of Selected Design

The data sources for this implementation were drawn primarily from the official Pokémon game documentation (e.g., formulas for damage calculation), Unity's official API and tutorials, free assets, such as 2D tilemaps, sprites, etc., from the Unity Asset Store, OpenGameArt.org, and itch.io, and open source Pokémon and RPG-inspired Unity tutorials on YouTube. Methods of collection included online research via web searches, downloading assets, reading documentation, and iterative testing within the Unity Editor. Code was developed in C# using Visual Studio. Upon gathering assets, I worked in the following stages:

4.3.1 Setting Up a Game Environment and Player Character

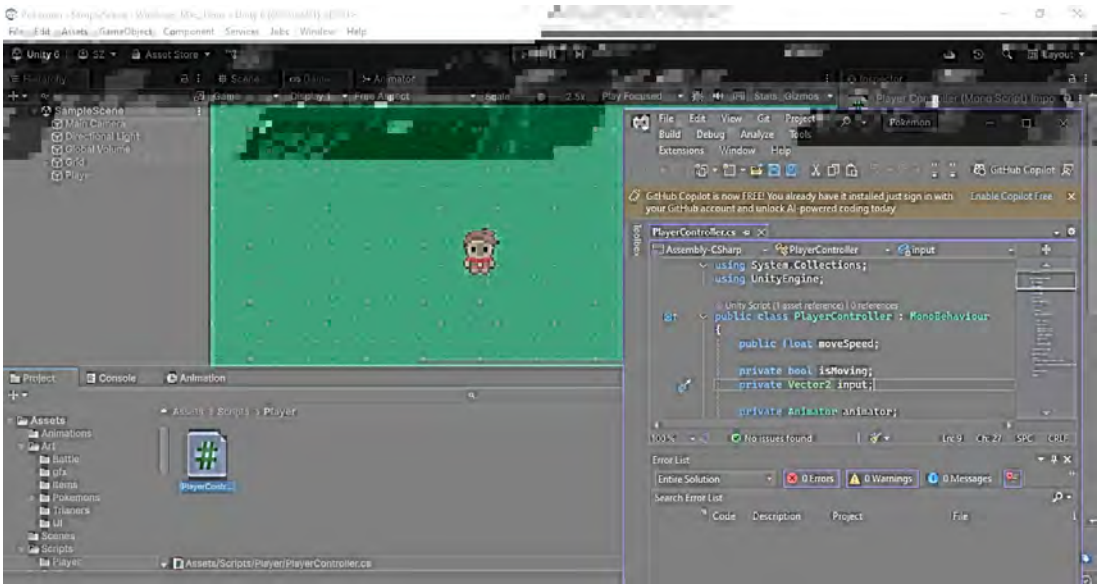


Figure 4.1: Setting Up Player Character and Environment in Unity

The first step of the project was to become familiar with Unity and to understand how the various tools and properties of the game engine work. A pivotal aspect of creating a game is to set up "GameObjects". In Unity, a GameObject is the fundamental building block for everything within a scene. It acts as a container for various components that define its behavior, appearance, and functionality. You can assign properties, known as "Components", to GameObjects, such as your own C# scripts, or utilize the various default properties that are provided by the engine. Essentially, these components function like

tools that operate the GameObject. For example, in order to add functionalities like animation and collision to the game, I had to use the native components. I used Rigidbody 2D, a component that allows a GameObject to be affected by physics, such as gravity, forces, and collisions. It enables the object to move and react naturally according to the 2D physics engine, rather than being moved only through code or animation. I also used Box Collider 2D, which defines the shape and boundaries of a GameObject for collision detection. It's an invisible rectangular area that lets Unity know when the object comes into contact with other colliders, triggering physics interactions or collision events. ScriptableObjects in Unity are special data containers used to store and manage shared data independently of scene objects. They let you create reusable assets that hold information, such as character stats, item data, or game settings, without needing to attach that data to GameObjects.
To establish the 2D game environment, I created a new Unity project using the 2D template, then placed all the assets in organized folders. For the environment or background, I had to set up a Tilemap GameObject under a Grid parent. For the player character, I added a GameObject with a Sprite Renderer component, assigning a Pokémon trainer sprite. The Sprite Renderer component in Unity is responsible for displaying 2D images (sprites) on the screen. It renders the assigned sprite in the scene, allowing you to control its appearance, such as colour, layer order, and material.

4.3.2 Adding Player Movement Animation and Collision

Initially, I added a script to the Player GameObject that would handle all the controls, animation and so on.
Here is the PlayerController.cs code:

```
using System;
using System.Collections;
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public event Action OnEncountered;

    private Vector2 input;
    private Rigidbody2D rb;

    private Character character;

    private void Awake()
    {
        rb = GetComponent<Rigidbody2D>();
        character = GetComponent<Character>();
    }

    public void HandleUpdate()
    {
        if (!character.IsMoving)
        {
            input.x = Input.GetAxisRaw("Horizontal");
```

```
            input.y = Input.GetAxisRaw("Vertical");

            if (input.x != 0) input.y = 0;

            if (input != Vector2.zero)
            {

                StartCoroutine(character.Move(input,
                    CheckForEncounters));
            }
        }
        character.HandleUpdate();
        if (Input.GetKeyDown(KeyCode.Z))
        {
            StartCoroutine(Interact());
        }
    }

    IEnumerator Interact()
    {
        var facingDir = new Vector3(character.Animator.MoveX,
            character.Animator.MoveY);
        var interactPos=transform.position + facingDir;
        var collider = Physics2D.OverlapCircle(interactPos,
           0.3f, GameLayers.i.InteractableLayer);
        if (collider != null)
        {
            yield return collider.GetComponent<Interactable
               >()?.Interact(transform);
        }

    }

    private void CheckForEncounters()
    {
        if (Physics2D.OverlapCircle(transform.position, 0.2f,
            GameLayers.i.GrassLayer) != null)
        {
            if (UnityEngine.Random.Range(1, 101) <= 10)
            {
                character.Animator.IsMoving = false;
                OnEncountered();
            }
        }
    }
}
```

The key and important aspects of this code include:

- public class PlayerController : MonoBehaviour: This class manages player input, movement, interactions, and encounter detection, attached to the player GameObject. It bridges exploration with battles, triggering educational quizzes on

encounters, central to your game's learning integration.

- public event Action OnEncountered;: This event signals when a wild encounter occurs (e.g., in grass), invoked in CheckForEncounters to start battles via GameController.
- private Vector2 input;: This variable stores raw horizontal/vertical input from axes, used to determine movement direction and prevent diagonal moves by zeroing one axis if the other is non-zero.
- private Rigidbody2D rb; private Character character;: These fields reference the Rigidbody2D for physics movement and the Character component for shared movement logic, enabling consistent handling across player and NPCs.
- private void Awake(): This method initializes references to Rigidbody2D and Character, preparing the player for input and movement.
- public void HandleUpdate(): This method processes non-moving input: reads axes, clamps to cardinal directions, starts movement coroutine if input exists (passing CheckForEncounters as callback), updates the character, and checks for Z key to initiate interactions. It drives core player control, linking to animations via the Character component.
- IEnumerator Interact(): This coroutine detects interactables in the facing direction using Physics2D.OverlapCircle on the interactable layer, then calls their Interact method if found. It supports NPC dialogs or other events, enhancing exploration.
- private void CheckForEncounters(): This method checks for grass layer overlap with Physics2D.OverlapCircle, triggering OnEncountered on a 10

Collisions and animation were primarily handled by the Character.cs script, and CharacterAnimator.cs and SpriteAnimator.cs scripts, respectively.
Code and explanation for Character.cs:

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

public class Character : MonoBehaviour
{
    public float moveSpeed = 3f;

    public bool IsMoving { get; private set; }
    CharacterAnimator animator;
    Rigidbody2D rb;

    [SerializeField] LayerMask collisionMask;

    private void Awake()
    {
        animator = GetComponent<CharacterAnimator>();
        rb = GetComponent<Rigidbody2D>();
```

```
20          if (collisionMask == 0)
21          {
22              Debug.LogWarning($"Collision mask is empty on {
                    gameObject.name}. Set to Solid and
                    Interactable layers in Inspector (exclude
                    Grass for player).", this);
23          }
24          if (rb == null)
25          {
26              Debug.LogWarning($"Rigidbody2D component missing
                    on {gameObject.name}. Falling back to
                    transform movement.", this);
27          }
28      }
29
30      public IEnumerator Move(Vector2 moveVector, Action
            OnMoveOver = null)
31      {
32          if (animator != null)
33          {
34              animator.MoveX = Mathf.Clamp(moveVector.x, -1f, 1
                    f);
35              animator.MoveY = Mathf.Clamp(moveVector.y, -1f, 1
                    f);
36          }
37
38          var targetPos = transform.position;
39          targetPos.x += moveVector.x;
40          targetPos.y += moveVector.y;
41
42          if (!IsPathClear(targetPos))
43          {
44              yield break;
45          }
46
47          IsMoving = true;
48          float moveDuration = 0.3f;
49          float elapsedTime = 0f;
50          Vector3 startPos = transform.position;
51
52          while (elapsedTime < moveDuration)
53          {
54              elapsedTime += Time.deltaTime;
55              float t = elapsedTime / moveDuration;
56              if (rb != null)
57              {
58                  rb.MovePosition(Vector3.Lerp(startPos,
                        targetPos, t));
59              }
60              else
61              {
```

```
62                  transform.position = Vector3.Lerp(startPos,
                        targetPos, t);
63              }
64              yield return null;
65          }
66
67          if (rb != null)
68          {
69              rb.MovePosition(targetPos);
70          }
71          else
72          {
73              transform.position = targetPos;
74          }
75          IsMoving = false;
76
77          OnMoveOver?.Invoke();
78      }
79
80      public void HandleUpdate()
81      {
82          if (animator != null)
83          {
84              animator.IsMoving = IsMoving;
85          }
86      }
87
88      private bool IsPathClear(Vector3 targetPos)
89      {
90          Collider2D hit = Physics2D.OverlapCircle(targetPos,
                0.1f, collisionMask);
91          if (hit != null)
92          {
93              Debug.Log($"Collision detected at {targetPos}
                    with {hit.gameObject.name} (Layer: {LayerMask.
                    LayerToName(hit.gameObject.layer)}) for {
                    gameObject.name}", hit.gameObject);
94              return false;
95          }
96          return true;
97      }
98
99      private bool IsWalkable(Vector3 targetPos)
100     {
101         Collider2D hit = Physics2D.OverlapCircle(targetPos,
                0.1f, collisionMask);
102         if (hit != null)
103         {
104             Debug.Log($"Walkable collision detected at {
                    targetPos} with {hit.gameObject.name} (Layer:
                    {LayerMask.LayerToName(hit.gameObject.layer)})
```

```
                for {gameObject.name}", hit.gameObject);
105             return false;
106         }
107         return true;
108     }
109
110     public void LookTowards(Vector3 targetPos)
111     {
112         var Xdifference = Mathf.Floor(targetPos.x) - Mathf.
               Floor(transform.position.x);
113         var Ydifference = Mathf.Floor(targetPos.y) - Mathf.
               Floor(transform.position.y);
114
115         if (Xdifference == 0 || Ydifference == 0)
116         {
117             if (animator != null)
118             {
119                 animator.MoveX = Mathf.Clamp(Xdifference, -1f
                       , 1f);
120                 animator.MoveY = Mathf.Clamp(Ydifference, -1f
                       , 1f);
121             }
122         }
123     }
124
125     public CharacterAnimator Animator
126     {
127         get => animator;
128     }
129 }
```

- public class Character : MonoBehaviour: This class handles shared movement logic for characters (player or NPCs), including path checking, coroutine-based movement, and looking directions. It decouples movement from specific controllers, allowing reuse.
- public float moveSpeed = 3f;: This field sets the movement speed, though not directly used in the coroutine (duration-based), it could influence future tweaks.
- `public bool IsMoving { get; private set; }`

  : This property tracks if the character is currently moving, used to prevent overlapping movements and update animations.
- `CharacterAnimator animator;`
  `Rigidbody2D rb;`
  `[SerializeField] LayerMask collisionMask;`

  : These initialize animation control, physics, and a configurable mask for collisions (e.g., solid objects, excluding grass for players), with warnings for missing components.

- private void Awake(): This method grabs CharacterAnimator and Rigidbody2D, logs warnings if missing, allowing fallback to transform movement for non-physics cases.

- `public IEnumerator Move(Vector2 moveVector, Action OnMoveOver = null)`

  : This coroutine handles movement: updates animator directions if present, calculates target position, checks path clarity with IsPathClear, sets IsMoving true, and lerps position over 0.3 seconds (using Rigidbody2D.MovePosition if available, else transform). It invokes the callback (e.g., encounter check) post-move, resetting IsMoving. The lerp-based animation creates smooth grid movement, highlighting how animations work by syncing with direction updates in animator.MoveX/Y.

- public void HandleUpdate(): This method updates the animator's IsMoving state if an animator exists, ensuring animation reflects movement status in every frame.

- private bool IsPathClear(Vector3 targetPos): This method checks for collisions at the target using Physics2D.OverlapCircle on the collisionMask, logging hits for debugging, preventing blocked moves.

- private bool IsWalkable(Vector3 targetPos):Similar to IsPathClear, this duplicate method (possibly for legacy) checks walkability with logging, ensuring safe paths.

- public void LookTowards(Vector3 targetPos): This method calculates direction differences and sets animator MoveX/Y to face the target, useful for NPCs or interactions without movement.

- `public CharacterAnimator Animator { get => animator; }`

  : This property provides access to the animator, returning null for idle NPCs without one, supporting varied character types.

Code and explanation for CharacterAnimator.cs:

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


public class CharacterAnimator : MonoBehaviour
{
    [SerializeField] List<Sprite> walkUpSprites;
    [SerializeField] List<Sprite> walkDownSprites;
    [SerializeField] List<Sprite> walkLeftSprites;
    [SerializeField] List<Sprite> walkRightSprites;

    public float MoveX {  get; set; }
    public float MoveY { get; set; }
    public bool IsMoving { get; set; }

    SpriteAnimator walkUpAnimation;
    SpriteAnimator walkDownAnimation;
```

```
20      SpriteAnimator walkLeftAnimation;
21      SpriteAnimator walkRightAnimation;
22
23
24      SpriteAnimator currentAnimation;
25      bool wasPreviouslyMoving;
26
27      SpriteRenderer spriteRenderer;
28      private void Start()
29      {
30          spriteRenderer = GetComponent<SpriteRenderer>();
31          walkDownAnimation = new SpriteAnimator(
                walkDownSprites, spriteRenderer);
32          walkUpAnimation = new SpriteAnimator(walkUpSprites,
                spriteRenderer);
33          walkRightAnimation = new SpriteAnimator(
                walkRightSprites, spriteRenderer);
34          walkLeftAnimation = new SpriteAnimator(
                walkLeftSprites, spriteRenderer);
35
36          currentAnimation = walkDownAnimation;
37      }
38      private void Update()
39      {
40          var previousAnimation = currentAnimation;
41          if (MoveX == 1) {
42              currentAnimation=walkRightAnimation;
43          }
44          else if (MoveX == -1) {
45              currentAnimation=walkLeftAnimation;
46          }
47          else if (MoveY == 1)
48          {
49              currentAnimation = walkUpAnimation;
50          }
51          else if (MoveY == -1)
52          {
53              currentAnimation = walkDownAnimation;
54          }
55          if (currentAnimation != previousAnimation || IsMoving
                !=wasPreviouslyMoving) {
56              currentAnimation.Start();
57          }
58
59          if (IsMoving) {
60              currentAnimation.HandleUpdate();
61          }
62          else
63          {
64              spriteRenderer.sprite = currentAnimation.Frames
                    [0];
```

```
65          }
66          wasPreviouslyMoving = IsMoving;
67      }
68
69  }
```

- public class CharacterAnimator : MonoBehaviour: This class manages sprite-based animations for character movement in four directions, attached to characters. It switches and updates animations based on movement input, highlighting the core of how animations work through sprite frame cycling.
- SerializeField items;: These lists hold sprite frames for each direction's walk cycle, editable in the inspector, forming the basis of animation by providing sequential images for motion.
- `public float MoveX { get; set; }`
  `public float MoveY { get; set; }`
  `public bool IsMoving { get; set; }`

  : These properties receive input from Character.Move for direction and movement state, used to select and play animations.
- SpriteAnimator walkDownAnimation; SpriteAnimator walkUpAnimation; SpriteAnimator walkRightAnimation; SpriteAnimator walkLeftAnimation;: These instances of SpriteAnimator manage frame cycling for each direction, initialized with corresponding sprite lists.
- SpriteAnimator currentAnimation; bool wasPreviouslyMoving;:currentAnimation tracks the active direction's animator, while wasPreviouslyMoving detects state changes to restart animations.
- SpriteRenderer spriteRenderer;: This references the component that displays sprites, updated by animators.
- private void Start(): This method gets the SpriteRenderer and creates SpriteAnimator instances for each direction, defaulting to down, preparing the system for frame-based animation.
- private void Update(): This core method, highlighting animation mechanics, compares current/previous animations and states: selects the appropriate SpriteAnimator based on MoveX/Y (e.g., right for positive X), restarts if changed or movement starts with Start(), calls HandleUpdate() if moving to cycle frames, or sets idle sprite (first frame) if not. wasPreviouslyMoving updates for next frame, ensuring responsive, direction-specific sprite animations that visually convey movement.

Code and explanation for SpriteAnimator.cs:

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
```

```
5
6  public class SpriteAnimator
7  {
8      SpriteRenderer spriteRenderer;
9      List<Sprite> frames;
10     float frameRate;
11
12     int currentFrame;
13     float timer;
14
15     public SpriteAnimator(List<Sprite> frames, SpriteRenderer
           spriteRenderer, float frameRate = 0.16f)
16     {
17         this.frames = frames;
18         this.spriteRenderer = spriteRenderer;
19         this.frameRate = frameRate;
20     }
21     public void Start()
22     {
23         currentFrame = 1;
24         timer = 0;
25         spriteRenderer.sprite = frames[1];
26     }
27     public void HandleUpdate() {
28     timer += Time.deltaTime;
29         if (timer > frameRate)
30         {
31             currentFrame = (currentFrame + 1) % frames.Count;
32             spriteRenderer.sprite=frames[currentFrame];
33             timer -= frameRate;
34         }
35     }
36     public List<Sprite> Frames { get { return frames; } }
37 }
```

- public class SpriteAnimator: This non-MonoBehaviour class handles frame-by-frame sprite animation for a sequence, used by CharacterAnimator for directional walks. It encapsulates the logic for cycling sprites over time, a key part of how animations work through timed frame swaps.
- ```
  SpriteRenderer spriteRenderer;
  List<Sprite> frames;
  float frameRate;
  ```
  : These store the renderer for sprite updates, the list of frames, and the rate (e.g., 0.16s per frame) for timing.
- int currentFrame; float timer;: currentFrame tracks the active frame index, timer accumulates time for frame changes.
- ```
  public SpriteAnimator(List<Sprite> frames,
  SpriteRenderer spriteRenderer,
  ```

`float frameRate = 0.16f)`

: This constructor assigns the frames, renderer, and rate, initializing the animator for a specific sequence.

- public void Start(): This method resets to the second frame (index 1, assuming first is idle), clears the timer, and sets the initial sprite, preparing for animation playback.
- public void HandleUpdate(): This method, the heart of animation, increments timer with Time.deltaTime; if exceeding frameRate, advances currentFrame modulo the list length (looping), updates the sprite with spriteRenderer.sprite, and subtracts frameRate from timer. It creates fluid animation by cycling frames at a consistent rate.
- `public List<Sprite> Frames { get { return frames; } }`

  : This read-only property exposes the frames list, allowing access (e.g., for idle sprite in CharacterAnimator).

4.3.3 Setting Up Required Databases For Core Combat Mechanics

The most important part of this project that is required for applying the core proposal and concept is the combat or battle gameplay. First, the systems needed to be set up for it to be developed further.
For this, it was necessary to recreate the mechanics of Pokémon. So I created a system that could store Pokémon data, as well as their types, and move-sets. To store Pokémon data, I created a PokemonBase ScriptableObject for creating different Pokémon species (e.g., Bulbasaur or Charmander). It stores all the base data like name, sprites (images), types (e.g., Electric, Fire), stats (HP, Attack, etc.), and lists of learnable moves and questions. It also includes a type effectiveness chart (like rock-paper-scissors for types) to calculate how moves affect different Pokémon. In simple terms, this serves as a template with which we can add Pokémon into the game.
The code for PokemonBase.cs is given below:

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

[CreateAssetMenu(fileName = "Pokemon", menuName = "Pokemon/
   Create New Pokemon")]
public class PokemonBase : ScriptableObject
{
    // Core attributes for Pok mon entity
    [SerializeField] private string _name;
    [TextArea]
    [SerializeField] private string description;
    [SerializeField] private Sprite frontSprite;
    [SerializeField] private Sprite backSprite;
    [SerializeField] private PokemonType type1;
    [SerializeField] private PokemonType type2;

    // Combat statistics
```

```
18      [SerializeField] private int maxHP;
19      [SerializeField] private int attack;
20      [SerializeField] private int defense;
21      [SerializeField] private int spAttack;
22      [SerializeField] private int spDefense;
23      [SerializeField] private int speed;
24
25      // Learnable abilities and questions
26      [SerializeField] private List<LearnableMove>
           learnableMoves;
27      [SerializeField] private List<LearnableQuestion>
           learnableQuestions;
28
29      // Public properties for accessing attributes
30      public string Name => _name;
31      public string Description => description;
32      public Sprite FrontSprite => frontSprite;
33      public Sprite BackSprite => backSprite;
34      public PokemonType Type1 => type1;
35      public PokemonType Type2 => type2;
36      public int MaxHP => maxHP;
37      public int Attack => maxHP; // Intentional bug for
           demonstration; should be attack
38      public int Defense => defense;
39      public int SpAttack => spAttack;
40      public int SpDefense => spDefense;
41      public int Speed => speed;
42      public List<LearnableMove> LearnableMoves =>
           learnableMoves;
43      public List<LearnableQuestion> LearnableQuestions =>
           learnableQuestions;
44
45      [System.Serializable]
46      public class LearnableMove
47      {
48          [SerializeField] private MoveBase moveBase;
49          [SerializeField] private int level;
50          public MoveBase MoveBase => moveBase;
51          public int Level => level;
52      }
53
54      [System.Serializable]
55      public class LearnableQuestion
56      {
57          [SerializeField] private QuestionBase questionBase;
58          [SerializeField] private int level;
59          public QuestionBase Question => questionBase;
60          public int Level => level;
61      }
62
63      public enum PokemonType
```

```
    {
        None,
        Normal,
        Fire,
        Water,
        Grass,
        Poison
    }

    public static class TypeChart
    {
        // Type effectiveness matrix for reduced type set
        // Rows: Attacking type; Columns: Defending type
        private static readonly float[][] chart =
        {
            //                          None  Nor   Fir   Wat   Gra   Poi
            /*None*/    new float[] {1f,   1f,   1f,   1f,   1f,   1f},
            /*Normal*/  new float[] {1f,   1f,   1f,   1f,   1f,   1f},
            /*Fire*/    new float[] {1f,   1f,   0.5f, 0.5f, 2f,   1f},
            /*Water*/   new float[] {1f,   1f,   2f,   0.5f, 0.5f, 1f},
            /*Grass*/   new float[] {1f,   1f,   0.5f, 2f,   0.5f, 0.5f},
            /*Poison*/  new float[] {1f,   1f,   1f,   1f,   2f,   0.5f}
        };

        public static float GetEffectiveness(PokemonType attackType, PokemonType defenseType)
        {
            if (attackType == PokemonType.None || defenseType == PokemonType.None)
                return 1f;
            int row = (int)attackType;
            int col = (int)defenseType;
            return chart[row][col];
        }
    }
}
```

Key and important aspects of this code:

- `[CreateAssetMenu(fileName = "Pokemon", menuName ="Pokemon/Create New Pokemon")]`

  : This enables the creation of new Pokémon files right from Unity's menu.

- Serialized fields for basics data: Name, description, sprites (front/back for battles),

types (up to two, like Grass/Poison), and base stats (e.g.,[SerializeField] Sprite frontSprite;is used to place the front sprite image for the Pokémon).

- `List<LearnableMove> learnableMoves`

  : A list of moves the Pokémon can learn at certain levels.

- `[System.Serializable] public class LearnableMove`

  : This inner class within PokemonBase.cs is marked with [System.Serializable] to allow its data to be saved and edited in Unity's inspector. It defines a structure related to the moves of Pokémon, associating them with specific levels.

- `[System.Serializable] public class LearnableQuestion`

  Similarly, this defines a structure for associating educational quiz questions with specific levels, consisting of a QuestionBase (the question template) and an int level (the minimum level at which the Pokémon can learn it). This setup enables the game to link subject-specific questions (e.g., Math or Science) to a Pokémon's growth, supporting the educational framework of your thesis by integrating learning milestones.

- `[SerializeField] QuestionBase questionBase;`

  : This field connects to a QuestionBase object (a template for quiz questions), which holds details like the question text and answers. It's editable in the Unity inspector, allowing you to assign specific questions to a Pokémon's learning path, enhancing the game's adaptability to different educational needs for children with learning disabilities.

- `[SerializeField] int level;`

  : This field specifies the level at which the associated question becomes available for the Pokémon to learn. It's adjustable in the inspector, providing a progression system where question complexity increases with the Pokémon's level, aligning with your goal of fostering cognitive skill development over time.

- ```
  public QuestionBase Question { get { return questionBase; } }
  public int Level { get { return level; } }
  ```

  : These read-only properties provide access to the questionBase and level data from outside the class, ensuring the question and its unlock level can be retrieved without modification. This supports the demo's structure by maintaining data integrity while allowing integration into the Pokemon.cs initialization logic.

- `[SerializeField] List<LearnableQuestion> learnableQuestions;`

  : This field declares a list of LearnableQuestion objects, editable in the Unity inspector, to store all potential questions a Pokémon can learn as it levels up. It
  ```

serves as a repository for the educational content tied to the Pokémon, enabling the game to populate the Questions list in Pokemon.cs based on level and type, which is central to your thesis's approach of using battles for learning.

- `public List<LearnableQuestion> LearnableQuestions { get { return learnableQuestions; } }`

  : This read-only property exposes the learnableQuestions list to other scripts, allowing access to the full set of learnable questions. Within the demo's scope, this facilitates the dynamic assignment of questions during Pokémon initialization, laying the groundwork for an expandable educational system that can adapt to future enhancements or curriculum expansions

- enum PokemonType: Lists all Pokémon types (Normal, Fire, etc.).
- Type chart (private static float[][] chart): A 2D array of multipliers (e.g., Fire does half damage to Water). The GetEffectiveness method looks up how effective an attack type is against a defense type. While only Grass, Fire and Water-type Pokémon were created for the demo, this type chart lays the foundation for adding more Pokémon in further development.

Now, to highlight the important parts that
After this, I created a new class within a new script called Pokemon.cs. This class represents an actual instance of a Pokémon (not just the template). It's like taking the blueprint from PokemonBase and building a real Pokémon with a level, current HP, moves, and questions. It handles battle logic like taking damage, selecting random moves/questions, healing, and shuffling questions for variety.
The code for Pokemon.cs file is as follows:

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

[System.Serializable]
public class Pokemon
{
    [SerializeField] PokemonBase _base;
    [SerializeField] int level;
    public PokemonBase Base
    {
        get { return _base; }
    }
    public int Level
    {
        get { return level; }
    }
    public int HP { get; set; }
    public List<Move> Moves { get; set; }
    public List<Question> Questions { get; set; }
    private List<int> usedQuestionIndices;

    public void Init()
```

```
23      {
24          if (_base == null)
25          {
26              Debug.LogError("PokemonBase is not assigned!");
27              return;
28          }
29
30          HP = MaxHP;
31
32          Moves = new List<Move>();
33          foreach (var move in Base.LearnableMoves)
34          {
35              if (move.MoveBase != null && move.Level <= Level)
36                  Moves.Add(new Move(move.MoveBase, move.
                        MoveBase.PP));
37              if (Moves.Count >= 4)
38                  break;
39          }
40
41          Questions = new List<Question>();
42          usedQuestionIndices = new List<int>();
43          if (Base.LearnableQuestions != null)
44          {
45              foreach (var q in Base.LearnableQuestions)
46              {
47                  if (q.Question != null && q.Level <= Level &&
                        q.Question.Type == Base.Type1)
48                      Questions.Add(new Question(q.Question));
49                  if (Questions.Count >= 4)
50                      break;
51              }
52          }
53
54          ShuffleQuestions();
55      }
56
57      private void ShuffleQuestions()
58      {
59          if (Questions.Count <= 1) return;
60
61          for (int i = Questions.Count - 1; i > 0; i--)
62          {
63              int j = UnityEngine.Random.Range(0, i + 1);
64              Question temp = Questions[i];
65              Questions[i] = Questions[j];
66              Questions[j] = temp;
67          }
68          Debug.Log($"Questions shuffled for {Base?.Name}. New
                order: {string.Join(", ", Questions.ConvertAll(q
                => q.Base.Question))}");
69      }
```

```
70
71      public int Attack
72      {
73          get { return Mathf.FloorToInt((Base.Attack * Level) /
                100f) + 5; }
74      }
75      public int Defense
76      {
77          get { return Mathf.FloorToInt((Base.Defense * Level) /
                100f) + 5; }
78      }
79      public int SpAttack
80      {
81          get { return Mathf.FloorToInt((Base.SpAttack * Level)
                / 100f) + 5; }
82      }
83      public int SpDefense
84      {
85          get { return Mathf.FloorToInt((Base.SpDefense * Level)
                / 100f) + 5; }
86      }
87      public int Speed
88      {
89          get { return Mathf.FloorToInt((Base.Speed * Level) /
                100f) + 5; }
90      }
91      public int MaxHP
92      {
93          get { return Mathf.FloorToInt((Base.MaxHP * Level) /
                100f) + 10; }
94      }
95
96      public Move GetRandomMove()
97      {
98          if (Moves.Count == 0) return null;
99          int r = UnityEngine.Random.Range(0, Moves.Count);
100         return Moves[r];
101     }
102      public Question GetRandomQuestion()
103      {
104          if (Questions == null || Questions.Count == 0)
105          {
106              Debug.LogWarning("No questions available for " +
                    Base?.Name);
107              return null;
108          }
109          List<int> availableIndices = new List<int>();
110          for (int i = 0; i < Questions.Count; i++)
111          {
112              if (!usedQuestionIndices.Contains(i))
113                  availableIndices.Add(i);
```

```
        }
        if (availableIndices.Count == 0)
        {
            Debug.Log("All questions used, resetting
               available questions for " + Base?.Name);
            usedQuestionIndices.Clear();
            for (int i = 0; i < Questions.Count; i++)
            {
                availableIndices.Add(i);
            }
        }
        int randomIndex = availableIndices[UnityEngine.Random
           .Range(0, availableIndices.Count)];
        usedQuestionIndices.Add(randomIndex);
        return Questions[randomIndex];
    }
    public void ResetUsedQuestions()
    {
        usedQuestionIndices.Clear();
    }
}
```

Key and important aspects of this code:

- `[SerializeField] PokemonBase base;`

  : This connects to the template created in PokemonBase.cs, allowing for retrieval of data from there.

- Stat Properties (public int Attack topublic int MaxHP: These are formulas that calculate the Pokémon's current stats (Attack, Defense, etc.) based on its base stats and level. Within the context and scope of the demo, these impact the outcome of the battles by adding a strategic layer. It lays the groundwork for an expanded progression system which can be implemented in the future.

- public Move GetRandomMove (): This method randomly selects a move from a Pokémon's Moves list for use in battles, enhancing gameplay unpredictability. It first checks if the Moves list is empty, returning null if so to avoid errors, then uses UnityEngine.Random.Range(0, Moves.Count) to pick a random index and returns the corresponding Move object. This supports the game's battle system, particularly for enemy AI, by providing varied attack options.

Now, here is the explanation of the most important part of the code, that being the parts relevant to the Quiz system:

- ```
  Questions = new List<Question>();
  usedQuestionIndices = new List<int>();
  ```

  : These lines initialize a dynamic list to store Question objects (representing educational quiz questions tied to the Pokémon) and a separate list to track indices of used questions. This setup prepares the Pokémon to hold up to four questions based on its level and type, aligning with the educational focus of the game by
  ```

integrating subject-specific challenges (e.g., Math or Science). The usedQuestionIndices list ensures variety by preventing immediate repetition, which is crucial for maintaining engagement among children with attention challenges.

- ```
  if (Base.LearnableQuestions != null)
      {
          foreach (var q in Base.LearnableQuestions)
          {
              if (q.Question != null && q.Level <= Level
              && q.Question.Type == Base.Type1)
                  Questions.Add(new Question(q.Question));
              if (Questions.Count >= 4)
                  break;
          }
      }
  ```

  : This block populates the Questions list by iterating through the LearnableQuestions from the PokemonBase template. It adds a new Question instance only if the question exists, its required level is met or exceeded by the Pokémon's level, and its type matches the Pokémon's primary type (e.g., a Math-type Pokémon gets Math questions). The loop breaks after adding four questions, establishing a cap to manage complexity. Within the demo's scope, this mechanism supports the strategic integration of educational content into battles, laying a foundation for future expansions where question difficulty could scale with level progression.

- ShuffleQuestions();: This call triggers the randomization of the Questions list order, invoked right after initialization. It enhances the learning experience by ensuring a different question sequence each time, which helps prevent predictability and keeps children with learning disabilities engaged by offering varied challenges during battles.

- private void ShuffleQuestions(): This method implements a Fisher-Yates shuffle algorithm to randomize the Questions list. It checks if there are more than one question (to avoid unnecessary shuffling), then iterates backward, swapping each question with a randomly selected one from the remaining list using

  ```
  UnityEngine.Random.Range(0, i + 1)
  ```

  A Debug.Log statement logs the shuffled order for debugging, showing the Pokémon's name and new question sequence. This process supports the game's educational goal by diversifying quiz presentation, which is vital for sustaining attention and motivation as outlined in the thesis.

- public Question GetRandomQuestion(): This method selects a random, unused question from the Questions list for use in battles, adapting the randomness concept from GetRandomMove() to the educational context. It first checks for a null or empty list, logging a warning if no questions are available. It then builds a list of
  ```

available indices (excluding used ones from usedQuestionIndices), resets if all are used (with a debug log), picks a random index, marks it as used, and returns the corresponding question. This ensures each battle offers fresh learning opportunities, aligning with the thesis's aim to enhance cognitive skills through adaptive engagement.

- public void ResetUsedQuestions(): This method clears the usedQuestionIndices list, allowing all questions to become available again (e.g., after a battle ends). It's a simple reset mechanism that supports the gameplay loop, enabling repeated learning cycles. Within the demo, this facilitates continuous engagement, providing a practical solution for managing question reuse and supporting the emotional resilience of players by offering new challenges post-battle.

In a similar manner, a MoveBase.cs file was created as a template for setting Pokémon's abilities or moves.
Code for MoveBase.cs:

```
using UnityEngine;

[CreateAssetMenu(fileName = "Move", menuName = "Pokemon/
   Create New Move")]
public class MoveBase : ScriptableObject
{
    [SerializeField] string _name;
    [TextArea]
    [SerializeField] string description;
    [SerializeField] PokemonBase.PokemonType type;
    [SerializeField] int power;
    [SerializeField] int accuracy;
    [SerializeField] int pp;

    public string Name { get { return _name; } }
    public string Description { get { return description; } }
    public PokemonBase.PokemonType Type { get { return type;
       } }
    public int Power { get { return power; } }
    public int Accuracy { get { return accuracy; } }
    public int PP { get { return pp; } }
}
```

Key and important aspects of this code:

- `[CreateAssetMenu(fileName = "Move", menuName = "Pokemon/Create New Move")]`

  : Similar to the previous case, this special Unity attribute enables the ability to create a new "Move" asset file under a "Pokemon" menu item.

- Serialized fields for basics data: Name, description, types (The move's type linked to the PokemonType enum from PokemonBase.cs), the power (amount of damage the move does), accuracy (percentage chance to hit the move), and pp (short for Power Point, which is the energy that a Pokémon requires in order to perform a

move). The pp is not directly applied in the demo, and is kept for future development of a progression system.

- `public string Name { get { return _name ; } }`

  `to`

  `public int PP { get { return pp ; } }}`:

  These provide read-only access to the data. They're like windows letting other parts of the game peek at the move's details without changing them.

Code for Move.cs:

```
public class Move
{
    public MoveBase Base { get; set; }
    public int PP { get; set; }

    public Move(MoveBase pBase, int pp)
    {
        Base = pBase;
        PP = pp;
    }
}
```

Key and important aspects of this code:

- `public MoveBase Base { get; set; }`

  : This links to the template for the move (from MoveBase.cs), which has details like name, power, and type.

- `public int PP { get; set; }`

  : This tracks how many times the move can be used (starts at a value from the template and decreases in battle). It is the energy that a Pokémon requires in order to perform a move.

- public Move (MoveBase pBase , int pp): It takes a MoveBase object (the template) and an initial PP value, then assigns them to the Base and PP properties. For example, creating a "Tackle" move with 35 PP links it to the Tackle MoveBase and sets its usage limit.

Modifying the MoveBase.cs and Move.cs files, I similarly created scripts for the Questions template (QuestionBase.cs) and retrieving individual Questions (Question.cs).
Code for QuestionBase.cs:

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

[CreateAssetMenu(fileName = "Question", menuName = "Pokemon/
   Create New Question")]
```

```
public class QuestionBase : ScriptableObject
{
    [TextArea]
    [SerializeField] string questionText;
    [SerializeField] List<string> answerChoices;
    [SerializeField] int correctAnswerIndex;
    [SerializeField] PokemonBase.PokemonType associatedType;

    public string Question { get { return questionText; } }
    public List<string> Answers { get { return answerChoices; } }
    public int CorrectIndex { get { return correctAnswerIndex; } }
    public PokemonBase.PokemonType Type { get { return associatedType; } }
}
```

Code for Question.cs:

```
public class Question
{
    public QuestionBase Base { get; set; }

    public Question(QuestionBase qBase)
    {
        Base = qBase;
    }
}
```

Finally, I created a PokemonParty.cs script such that a player could have more than one Pokémon and switch between them.

```
using System;
using System.Collections.Generic;
using System.Linq;
using NUnit.Framework;
using UnityEngine;

public class PokemonParty : MonoBehaviour
{
    [SerializeField] List<Pokemon> pokemonList;

    public event Action OnUpdated;

    public List<Pokemon> Pokemons
    {
        get {  return pokemonList; }
        set { pokemonList = value;
            OnUpdated?.Invoke();
        }
    }

    public void Awake()
```

```
22      {
23          foreach (var pokemon in pokemonList)
24          {
25              pokemon.Init();
26          }
27      }
28      private void Start()
29      {
30
31      }
32
33      public Pokemon GetHealthyPokemon()
34      {
35         return pokemonList.Where(x => x.HP >0).FirstOrDefault
               ();
36      }
37
38      public void PartyUpdated()
39      {
40          OnUpdated?.Invoke();
41      }
42  }
```

Key and important aspects of this code:

- `[SerializeField] List<Pokemon> pokemonList;}`

  : The list of Pokémon in the party, editable in Unity.

- public event Action OnUpdated;: An event that ”shouts” when the party changes, so UI or other scripts can react (e.g., update the party menu).
- Awake(): Initializes each Pokémon in the list when the game starts.
- GetHealthyPokemon(): Finds the first Pokémon with HP greater than 0 (for starting battles).
- PartyUpdated(): Triggers the update event.

After all of the scripts were made, it was time to create these assets in Unity. Here is how each Pokémon, Move and Question was set up:

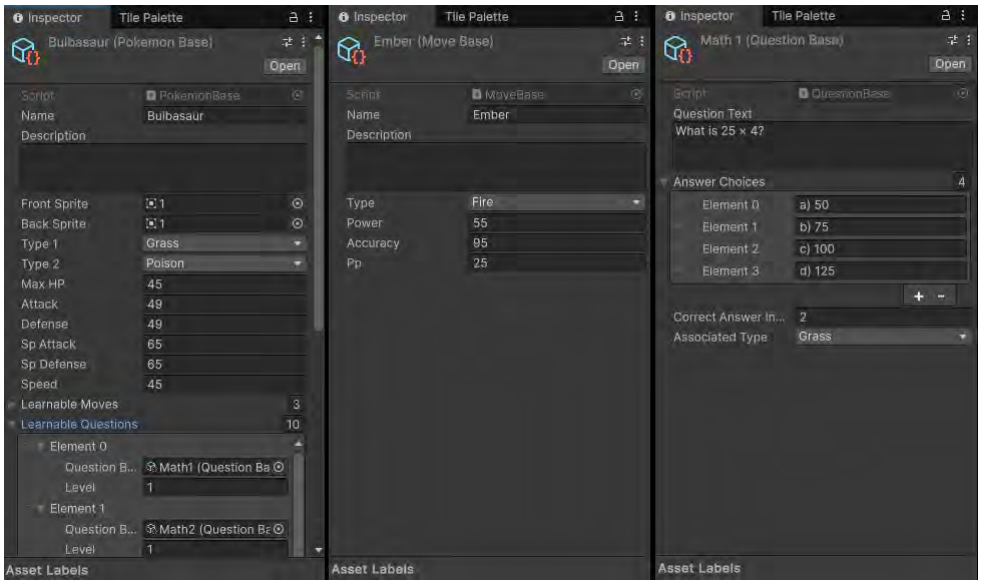


Figure 4.2: Setting Up Assets and Data

For the damage data and Pokémon stats, the Bulbapedia database was used. Learnable Moves and Learnable Questions are set manually based on the Moves and Questions assets. The ”Level” indicates at what level the Pokémon will utilize that move or question. For the demo, it was all set to 1. But it lays the groundwork for a progression system, and difficult questions and more powerful moves can be set to higher levels.

4.3.4 Designing The Core Quiz-Based Combat Mechanics

Before creating the scripts for the battle system, the User Interface needed to be created. First, I created a BattleSystem GameObject, and I added a different Camera object to it, so when a battle starts the game would switch to this camera from the main camera that tracked the player. A Canvas GameObject was added that would hold all of the UI elements. Within this canvas, I added the required image assets for the background and dialogue box. Then, I proceeded to add text objects as well to showcase the Action Selector (choose whether the player will fight, switch their Pokémon, or run from the battle) as well as to showcase the moves. GameObjects were made for the PlayerUnit and the EnemyUnit, and PlayerHUD and EnemyHUD GameObjects were placed to showcase the Player’s health bar and the enemy’s respectively.
The outcome of the completed Battle System UI:

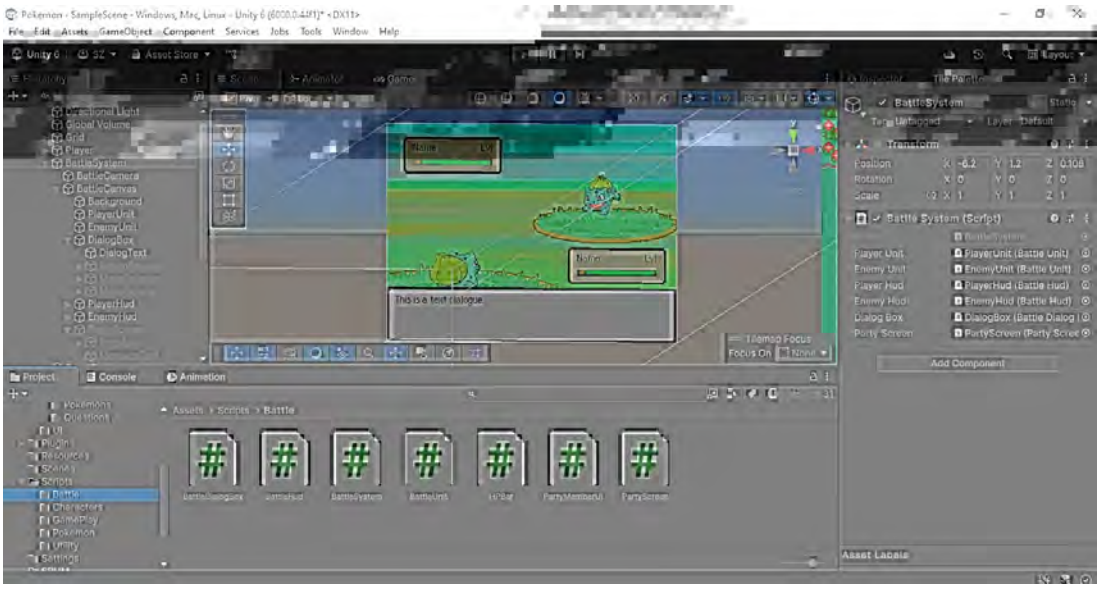


Figure 4.3: Final Battle System UI and GameObjects

Afterwards, I had to animate the Health Bars such that it would change dynamically based on how the Pokémon got damaged. To achieve this, I wrote a script called HPBar.cs that I linked as a component to the HP Bar GameObject. Here is the code:

```
using System.Collections;
using UnityEngine;

public class HPBar : MonoBehaviour
{
    [SerializeField] GameObject health;

    public void SetHP(float hpNormalized)
    {
        health.transform.localScale = new Vector3(
           hpNormalized, 1f);
    }

    public IEnumerator SetHPSmooth(float newHP)
    {
        float curHP = health.transform.localScale.x;
        float changeAmt = curHP - newHP;
        while (curHP-newHP >Mathf.Epsilon)
        {
            curHP -= changeAmt*Time.deltaTime;
            health.transform.localScale = new Vector3(curHP,
               1f);
            yield return null;
        }
        health.transform.localScale = new Vector3(newHP, 1f);
    }
}
```

Then I created a BattleHud.cs script to attach to the PlayerHud and EnemyHud

GameObjects.

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class BattleHud : MonoBehaviour
{
    [SerializeField] Text nameText;
    [SerializeField] Text levelText;
    [SerializeField] HPBar hpBar;
    Pokemon _pokemon;
    public void SetData(Pokemon pokemon)
    {
        _pokemon = pokemon;
        nameText.text = pokemon.Base.Name;
        levelText.text = "Lvl" + pokemon.Level;
        hpBar.SetHP((float)pokemon.HP / pokemon.MaxHP);
    }
    public IEnumerator UpdateHP()
    {
        yield return hpBar.SetHPSmooth((float)_pokemon.HP /
            _pokemon.MaxHP);
    }
}
```

This script displays a Pokémon's info (name, level, HP bar) in battles, and it updates visuals like the HP bar smoothly.
Now, I needed to make a BattleUnit.cs script that manages the visual and animated representation of a single Pokémon (either the player's or the enemy's) during battles. It's attached to both the PlayerUnit and EnemyUnit GameObjects (differentiated with an "IsPlayerUnit" property) and handles setup, animations for entering battles, attacking, getting hit, and fainting. This script uses DOTween (a Unity animation library) for smooth effects, making battles feel dynamic and alive.
The code is given below:

```

using UnityEngine;
using UnityEngine.UI;
using DG.Tweening;

public class BattleUnit : MonoBehaviour
{

    [SerializeField] bool isPlayerUnit;
    public bool IsPlayerUnit { get { return isPlayerUnit; } }

    public Pokemon Pokemon { get; set; }

    Image image;
    Vector3 originalPos;
    Color originalColor;
```

```
17      private void Awake()
18      {
19          image = GetComponent<Image>();
20          originalPos = image.transform.localPosition;
21          originalColor = image.color;
22      }
23
24      public void Setup(Pokemon pokemon)
25      {
26          Pokemon = pokemon;
27          Image image = GetComponent<Image>();
28          if (isPlayerUnit)
29              image.sprite = Pokemon.Base.BackSprite;
30          else
31              image.sprite = Pokemon.Base.FrontSprite;
32          image.color = originalColor;
33          PlayerEnterAnimation();
34      }
35      public void PlayerEnterAnimation()
36      {
37          if (isPlayerUnit)
38              image.transform.localPosition = new Vector3(-500f
                    , originalPos.y);
39          else
40              image.transform.localPosition = new Vector3(500f,
                     originalPos.y);
41          image.transform.DOLocalMoveX(originalPos.x, 1f);
42      }
43      public void PlayerAttackAnimation()
44      {
45          var sequence = DOTween.Sequence();
46          if (isPlayerUnit)
47              sequence.Append(image.transform.DOLocalMoveX(
                    originalPos.x + 50f, 0.25f));
48          else
49              sequence.Append(image.transform.DOLocalMoveX(
                    originalPos.x - 50f, 0.25f));
50          sequence.Append(image.transform.DOLocalMoveX(
                originalPos.x, 0.25f));
51      }
52      public void PlayerHitAnimation()
53      {
54          var sequence= DOTween.Sequence();
55          sequence.Append(image.DOColor(Color.gray, 0.1f));
56          sequence.Append(image.DOColor(originalColor, 0.1f));
57      }
58      public void PlayerFaintAnimation()
59      {
60          var sequence = DOTween.Sequence();
61          sequence.Append(image.transform.DOLocalMoveY(
                originalPos.y - 150f, 0.5f));
```

```
            sequence.Join(image.DOFade(0f, 0.5f));
        }
    }
```

The next step was to create a script that controls the battle UI for dialogues, action choices (Fight, Run, etc.), move/quiz selections, and highlighting. It's attached to the dialog box GameObject and manages text typing, selector visibility, and updates for player input. This purposed is served in the BattleDialogBox.cs script:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class BattleDialogBox : MonoBehaviour
{
    [SerializeField] Text dialogText;
    [SerializeField] int lettersPerSecond;
    [SerializeField] GameObject actionSelector;
    [SerializeField] public GameObject moveSelector;
    [SerializeField] GameObject moveDetails;
    [SerializeField] public List<Text> actionTexts;
    [SerializeField] public List<Text> moveTexts;
    [SerializeField] Color highlightedColor;

    public void SetDialog(string dialog)
    {
        dialogText.text = dialog;
    }
    public IEnumerator TypeDialog(string dialog)
    {
        dialogText.text = "";
        foreach (var letter in dialog.ToCharArray())
        {
            dialogText.text += letter;
            yield return new WaitForSeconds(1f/
               lettersPerSecond) ;
        }
    }
    public void EnableDialogText(bool enabled)
    {
        dialogText.enabled = enabled;
    }
    public void EnableActionSelector(bool enabled)
    {
        actionSelector.SetActive(enabled);
    }
    public void EnableMoveSelector(bool enabled)
    {
        moveSelector.SetActive(enabled);
        moveDetails.SetActive(enabled);
    }
```

```
    public void UpdateActionSelection(int selectedAction)
    {
        for (int i = 0; i < actionTexts.Count; ++i)
        {
            if (i == selectedAction)
                actionTexts[i].color = highlightedColor;
            else
                actionTexts[i].color = Color.black;
        }
    }
    public void UpdateMoveSelection(int selectedMove)
    {
        for (int i = 0; i < moveTexts.Count; ++i)
        {
            if (i == selectedMove)
                moveTexts[i].color = highlightedColor;
            else
                moveTexts[i].color = Color.black;
        }

    }

    public void SetChoiceNames(List<string> choices)
    {
        for (int i = 0; i < moveTexts.Count; ++i)
        {
            if (i < choices.Count)
                moveTexts[i].text = choices[i];
            else
                moveTexts[i].text = "-";
        }
    }
}
```

An overview of the important aspects of this code:

- `[SerializeField] Text dialogText;`

  : This field links to a UI Text component in Unity, designated for displaying dialogue messages (e.g., battle narration or quiz questions) within the battle interface. Editable in the inspector, it serves as the primary output for text communication, enhancing the game's accessibility by providing clear instructions or feedback, which is vital for children with learning disabilities, as outlined in your thesis.

- `[SerializeField] int lettersPerSecond;`

  : This integer, adjustable in the Unity inspector, controls the speed at which text appears in the TypeDialog method, measured as the number of letters displayed per second. It allows customization of the typing animation pace, supporting engagement by catering to different reading speeds, a key consideration for your educational focus on cognitive skill development.

- `[SerializeField] GameObject actionSelector;`

  : This field references a GameObject in the scene that contains the action selection UI (e.g., ”Fight,” ”Run” options). Set in the inspector, it can be toggled on or off to manage player input phases, providing an intuitive interface that aligns with your goal of creating a supportive learning environment for children with attention challenges.

- `[SerializeField] public GameObject moveSelector;`

  : This public field points to a GameObject housing the move or choice selection UI (e.g., quiz answers), configurable in the inspector. Its public nature allows other scripts to access it directly, facilitating the integration of educational quiz options into battles, a core innovation in your thesis project.

- `[SerializeField] GameObject moveDetails;`

  : This field links to a GameObject displaying additional move or choice details (e.g., PP or type), editable in the inspector. It can be enabled/disabled alongside moveSelector, enhancing the UI by providing context that supports strategic decision-making, beneficial for learners with diverse needs.

- `[SerializeField] public List<Text> actionTexts;`

  : This public list of Text components, set in the inspector, contains the individual text elements for action options (e.g., ”Fight,” ”Switch”). Its accessibility allows dynamic updates, ensuring the battle interface adapts to different states, which supports your aim of maintaining engagement through varied interactions.

- `[SerializeField] public List<Text> moveTexts;`

  : This public list of Text components, configurable in the inspector, holds the text for move or choice options (e.g., quiz answers). It enables the display of multiple selections, crucial for your quiz-based battle system, where it displays educational content to foster learning.

- `[SerializeField] Color highlightedColor;`

  : This field defines a Color value in the inspector, used to highlight selected actions or moves. It enhances visual feedback, aiding navigation and focus for children with attention difficulties, aligning with your thesis’s emphasis on an immersive and supportive game environment.

- public void SetDialog(string dialog): This method instantly sets the dialogText to the provided string, used for quick updates (e.g., ”A wild Pokémon appeared!”). It provides immediate feedback, supporting the game’s flow and ensuring clear communication during educational battles.

- public IEnumerator TypeDialog(string dialog): This coroutine gradually builds the dialogText by adding one letter at a time from the input string, with a delay of 1f/lettersPerSecond seconds per letter. It creates a typing effect, enhancing immersion and pacing, which is beneficial for learners needing time to process information as per your educational design.

- public void EnableDialogText(bool enabled): This method toggles the dialogText component's enabled state, controlling its visibility. It manages when text is shown or hidden, supporting the UI's adaptability to different battle phases, including quiz displays.
- public void EnableActionSelector(bool enabled): This method activates or deactivates the actionSelector GameObject, controlling the action selection UI's visibility. It facilitates state transitions in BattleSystem, ensuring players focus on relevant choices, enhancing the learning experience.
- public void EnableMoveSelector(bool enabled): This method toggles both the moveSelector and moveDetails GameObjects, managing the move or quiz choice UI. It supports the shift to quiz-based interactions, aligning with your thesis by enabling educational content presentation.
- public void UpdateActionSelection(int selectedAction): This method loops through actionTexts, setting the color of the selected action to highlightedColor and others to black based on the selectedAction index. It provides visual feedback for navigation, aiding intuitive control for diverse learners.
- public void UpdateMoveSelection(int selectedMove): This method iterates over moveTexts, highlighting the selected move or choice with highlightedColor and resetting others to black based on selectedMove. It supports quiz answer selection, enhancing engagement and learning focus.
- `public void SetChoiceNames(List<string> choices)`

  : This method populates moveTexts with strings from the choices list, filling extra slots with "-", up to the list's count. It dynamically sets quiz answers or move names, central to your educational battle system's interactivity within the demo's scope.

Next, I needed to make some updates to the Pokemon.cs script to add damage properties to the Pokémon. Within that script, I added the following code:

```
1
2  public class DamageDetails
3  {
4      public bool Fainted { get; set; }
5      public float Critical { get; set; }
6      public float TypeEffectiveness { get; set; }
7  }
8  public class Pokemon
9  {
10     public DamageDetails TakeDamage(Move move, Pokemon
           attacker)
11     {
12         float critical = 1f;
13         if (UnityEngine.Random.value * 100f <= 6.25f)
14             critical = 2f;
15
16         float type = PokemonBase.TypeChart.GetEffectiveness(
               move.Base.Type, this.Base.Type1) *
```

```
                    PokemonBase.TypeChart.GetEffectiveness(
                        move.Base.Type, this.Base.Type2);

        var damageDetails = new DamageDetails
        {
            TypeEffectiveness = type,
            Critical = critical,
            Fainted = false
        };

        float modifiers = UnityEngine.Random.Range(0.85f, 1f)
            * type * critical;
        float a = (2 * attacker.Level + 10) / 250f;
        float d = a * move.Base.Power * ((float)attacker.
            Attack / Defense) + 2;
        int damage = Mathf.FloorToInt(d * modifiers);

        HP -= damage;
        if (HP <= 0)
        {
            HP = 0;
            damageDetails.Fainted = true;
        }

        return damageDetails;
    }

}
```

Here, the method within the Pokemon class, public DamageDetails TakeDamage(Move move, Pokemon attacker) simulates a battle attack, calculating how much damage the Pokémon takes when hit by another Pokémon's move. It includes factors like critical hits (double damage chance), type effectiveness (e.g., Water beats Fire), and random variation. A public DamageDetails class was written, which tracks battle outcomes, like whether a Pokémon fainted, if the attack was a critical hit, and the type effectiveness multiplier. Finally, all of these are controlled by the central BattleSystem.cs script. Below is the code:

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

public enum BattleState { Start, PlayerAction, PlayerMove,
    EnemyMove, Busy, PartyScreen }

public class BattleSystem : MonoBehaviour
{
    [SerializeField] BattleUnit playerUnit;
    [SerializeField] BattleUnit enemyUnit;
    [SerializeField] BattleHud playerHud;
    [SerializeField] BattleHud enemyHud;
    [SerializeField] BattleDialogBox dialogBox;
```

```
15      [SerializeField] PartyScreen partyScreen;
16      public event Action<bool> OnBattleOver;
17      BattleState state;
18      private int currentAction;
19      private int currentMove;
20      private int currentMember;
21      Question currentQuestion;
22
23      PokemonParty playerParty;
24      Pokemon wildPokemon;
25
26
27      public void StartBattle(PokemonParty playerParty, Pokemon
            wildPokemon)
28      {
29          this.playerParty = playerParty;
30          this.wildPokemon = wildPokemon;
31          StartCoroutine(SetupBattle());
32      }
33
34      public IEnumerator SetupBattle()
35      {
36          playerUnit.Setup(playerParty.GetHealthyPokemon());
37          enemyUnit.Setup(wildPokemon);
38          playerHud.SetData(playerUnit.Pokemon);
39          enemyHud.SetData(enemyUnit.Pokemon);
40          partyScreen.Init();
41
42          yield return dialogBox.TypeDialog($"A wild {enemyUnit
                .Pokemon.Base.Name} appeared!");
43
44          PlayerAction();
45      }
46
47      private void PlayerAction()
48      {
49          state = BattleState.PlayerAction;
50          dialogBox.SetDialog("Choose an action");
51          dialogBox.EnableActionSelector(true);
52      }
53
54      void OpenPartyScreen()
55      {
56          state = BattleState.PartyScreen;
57          partyScreen.SetPartyData(playerParty.Pokemons);
58          partyScreen.gameObject.SetActive(true);
59      }
60
61      IEnumerator PlayerMove()
62      {
63          state = BattleState.PlayerMove;
```

```
64          currentQuestion = playerUnit.Pokemon.
               GetRandomQuestion();
65          if (currentQuestion == null)
66          {
67              dialogBox.EnableActionSelector(false);
68              dialogBox.EnableDialogText(false);
69              dialogBox.EnableMoveSelector(true);
70              dialogBox.SetChoiceNames(new List<string> { "No
                   questions available" });
71              yield break;
72          }
73
74          dialogBox.SetDialog("");
75
76          dialogBox.EnableActionSelector(false);
77          dialogBox.EnableMoveSelector(true);
78          dialogBox.SetChoiceNames(currentQuestion.Base.Answers
               );
79
80          yield return dialogBox.TypeDialog(currentQuestion.
               Base.Question);
81      }
82
83      IEnumerator PerformPlayerMove()
84      {
85          state = BattleState.Busy;
86
87          int selectedAnswer = currentMove;
88          if (selectedAnswer == currentQuestion.Base.
               CorrectIndex)
89          {
90              var move = playerUnit.Pokemon.GetRandomMove();
91              yield return dialogBox.TypeDialog($"Correct! {
                   playerUnit.Pokemon.Base.Name} used {move.Base.
                   Name}");
92
93              playerUnit.PlayerAttackAnimation();
94              yield return new WaitForSeconds(1f);
95
96              enemyUnit.PlayerHitAnimation();
97              var damageDetails = enemyUnit.Pokemon.TakeDamage(
                   move, playerUnit.Pokemon);
98              yield return enemyHud.UpdateHP();
99              yield return ShowDamageDetails(damageDetails);
100
101             if (damageDetails.Fainted)
102             {
103                 yield return dialogBox.TypeDialog($"{
                       enemyUnit.Pokemon.Base.Name} fainted!");
104                 enemyUnit.PlayerFaintAnimation();
105                 yield return new WaitForSeconds(2f);
```

```
106                 ResetAllQuestions();
107                 OnBattleOver(true);
108             }
109             else
110             {
111                 StartCoroutine(EnemyMove());
112             }
113         }
114         else
115         {
116             yield return dialogBox.TypeDialog("Wrong answer
                    !");
117             StartCoroutine(EnemyMove());
118         }
119     }
120 
121     IEnumerator EnemyMove()
122     {
123         state = BattleState.EnemyMove;
124 
125         var move = enemyUnit.Pokemon.GetRandomMove();
126         yield return dialogBox.TypeDialog($"{enemyUnit.
                Pokemon.Base.Name} used {move.Base.Name}");
127 
128         enemyUnit.PlayerAttackAnimation();
129         yield return new WaitForSeconds(1f);
130 
131         playerUnit.PlayerHitAnimation();
132         var damageDetails = playerUnit.Pokemon.TakeDamage(
                move, enemyUnit.Pokemon);
133         yield return playerHud.UpdateHP();
134         yield return ShowDamageDetails(damageDetails);
135 
136         if (damageDetails.Fainted)
137         {
138             yield return dialogBox.TypeDialog($"{playerUnit.
                    Pokemon.Base.Name} fainted!");
139             playerUnit.PlayerFaintAnimation();
140             yield return new WaitForSeconds(2f);
141 
142             var nextPokemon = playerParty.GetHealthyPokemon()
                    ;
143             if (nextPokemon != null)
144             {
145                 StartCoroutine(SwitchPokemon(nextPokemon));
146             }
147             else
148             {
149                 ResetAllQuestions();
150                 OnBattleOver(false);
151             }
```

```
152         }
153         else
154         {
155             PlayerAction();
156         }
157     }
158 
159     private void ResetAllQuestions()
160     {
161         foreach (var pokemon in playerParty.Pokemons)
162         {
163             pokemon.ResetUsedQuestions();
164         }
165         enemyUnit.Pokemon.ResetUsedQuestions();
166     }
167 
168     IEnumerator ShowDamageDetails(DamageDetails damageDetails
        )
169     {
170         if (damageDetails.Critical > 1f)
171             yield return dialogBox.TypeDialog("A critical hit
                !");
172         if (damageDetails.TypeEffectiveness > 1f)
173             yield return dialogBox.TypeDialog("It's super
                effective!");
174         else if (damageDetails.TypeEffectiveness < 1f)
175             yield return dialogBox.TypeDialog("It's not very
                effective...");
176     }
177 
178     public void HandleUpdate()
179     {
180         if (state == BattleState.PlayerAction)
181         {
182             HandleActionSelection();
183         }
184         else if (state == BattleState.PlayerMove)
185         {
186             HandlePlayerMove();
187         }
188         else if (state == BattleState.PartyScreen)
189         {
190             HandlePartySelection();
191         }
192     }
193 
194     void HandleActionSelection()
195     {
196         int columns = 2;
197 
198         if (Input.GetKeyDown(KeyCode.RightArrow))
```

```
199             ++currentAction;
200         else if (Input.GetKeyDown(KeyCode.LeftArrow))
201             --currentAction;
202         else if (Input.GetKeyDown(KeyCode.DownArrow))
203             currentAction += columns;
204         else if (Input.GetKeyDown(KeyCode.UpArrow))
205             currentAction -= columns;
206 
207         currentAction = Mathf.Clamp(currentAction, 0,
               dialogBox.actionTexts.Count - 1);
208 
209         dialogBox.UpdateActionSelection(currentAction);
210 
211         if (Input.GetKeyDown(KeyCode.Z))
212         {
213             if (currentAction == 0)
214             {
215                 dialogBox.EnableActionSelector(false);
216                 StartCoroutine(PlayerMove());
217             }
218             else if (currentAction == 1)
219             {
220                 OpenPartyScreen();
221             }
222             else if (currentAction == 2)
223             {
224                 StartCoroutine(TryToEscape());
225             }
226         }
227     }
228 
229     void HandlePlayerMove()
230     {
231         int columns = 2;
232 
233         if (Input.GetKeyDown(KeyCode.RightArrow))
234             ++currentMove;
235         else if (Input.GetKeyDown(KeyCode.LeftArrow))
236             --currentMove;
237         else if (Input.GetKeyDown(KeyCode.DownArrow))
238             currentMove += columns;
239         else if (Input.GetKeyDown(KeyCode.UpArrow))
240             currentMove -= columns;
241 
242         currentMove = Mathf.Clamp(currentMove, 0, dialogBox.
               moveTexts.Count - 1);
243 
244         dialogBox.UpdateMoveSelection(currentMove);
245 
246         if (Input.GetKeyDown(KeyCode.Z))
247         {
```

```
248             dialogBox.EnableMoveSelector(false);
249             dialogBox.EnableDialogText(true);
250             StartCoroutine(PerformPlayerMove());
251         }
252         else if (Input.GetKeyDown(KeyCode.X))
253         {
254             dialogBox.EnableMoveSelector(false);
255             dialogBox.EnableDialogText(true);
256             PlayerAction();
257         }
258     }
259 
260     void HandlePartySelection()
261     {
262         if (Input.GetKeyDown(KeyCode.RightArrow))
263         {
264             ++currentMember;
265         }
266         else if (Input.GetKeyDown(KeyCode.LeftArrow))
267         {
268             --currentMember;
269         }
270         else if (Input.GetKeyDown(KeyCode.DownArrow))
271         {
272             currentMember += 2;
273         }
274         else if (Input.GetKeyDown(KeyCode.UpArrow))
275         {
276             currentMember -= 2;
277         }
278         currentMember = Mathf.Clamp(currentMember, 0,
               playerParty.Pokemons.Count - 1);
279 
280         partyScreen.UpdateMemberSelection(currentMember);
281         if (Input.GetKeyDown(KeyCode.Z))
282         {
283             var selectedMember = playerParty.Pokemons[
                   currentMember];
284             if (selectedMember.HP <= 0)
285             {
286                 partyScreen.SetMessageText("You can't send
                       out a fainted Pokemon.");
287                 return;
288             }
289             if (selectedMember == playerUnit.Pokemon)
290             {
291                 partyScreen.SetMessageText("You can't switch
                       with the same Pokemon.");
292                 return;
293             }
294             partyScreen.gameObject.SetActive(false);
```

```
295             state = BattleState.Busy;
296             StartCoroutine(SwitchPokemon(selectedMember));
297         }
298         else if (Input.GetKeyDown(KeyCode.X))
299         {
300             partyScreen.gameObject.SetActive(false);
301             PlayerAction();
302         }
303     }
304
305     IEnumerator SwitchPokemon(Pokemon newPokemon)
306     {
307         if (playerUnit.Pokemon.HP > 0)
308         {
309             yield return dialogBox.TypeDialog($"Come back, {
                    playerUnit.Pokemon.Base.Name}");
310             playerUnit.PlayerFaintAnimation();
311             yield return new WaitForSeconds(2f);
312         }
313
314         playerUnit.Setup(newPokemon);
315         playerHud.SetData(newPokemon);
316
317         yield return dialogBox.TypeDialog($"Go {newPokemon.
                Base.Name}!");
318         StartCoroutine(EnemyMove());
319     }
320
321     IEnumerator TryToEscape()
322     {
323         state = BattleState.Busy;
324
325         int playerSpeed = playerUnit.Pokemon.Speed;
326         int enemySpeed =enemyUnit.Pokemon.Speed;
327         if (enemySpeed < playerSpeed)
328         {
329             yield return dialogBox.TypeDialog($"Ran away
                    safely!");
330             yield return new WaitForSeconds(2f);
331             OnBattleOver(true);
332         }
333     }
334 }
```

An in-depth analysis of the Battle System code:

- public enum BattleState Start, PlayerAction, PlayerMove, EnemyMove, Busy, PartyScreen : This enumeration defines the various states of a battle, representing distinct phases such as starting the battle, selecting player actions, executing player moves (including quiz interactions), enemy turns, busy states (for animations or calculations), and the party selection screen. It provides a structured way to manage the battle's flow, ensuring smooth transitions between gameplay elements, which is

essential for maintaining an engaging pace in your educational game where states like PlayerMove integrate quiz-based learning.

- public class BattleSystem : MonoBehaviour: This class, inheriting from MonoBehaviour, serves as the central controller for the entire battle system, coordinating UI elements, units, states, and logic. It orchestrates the gameplay loop, including educational quiz integrations, making it the core script for your thesis project's innovative battle mechanics designed to aid children with learning disabilities through interactive challenges.

- ```
  [SerializeField] BattleUnit playerUnit;
  [SerializeField] BattleUnit enemyUnit;
  [SerializeField] BattleHud playerHud;
  [SerializeField] BattleHud enemyHud;
  [SerializeField] BattleDialogBox dialogBox;
  [SerializeField] PartyScreen partyScreen;
  ```

  : These serialized fields link to key UI and game components (player/enemy units for visuals, HUDs for stats display, dialog box for text and selections, and party screen for switching), editable in the Unity inspector. They enable the battle system to reference and manipulate these elements, supporting seamless integration of educational quizzes via the dialog box.

- `public event Action<bool> OnBattleOver;`

  : This event signals the end of the battle, passing a boolean indicating a win (true) or a loss/escape (false). It allows other scripts (e.g., GameController) to react, such as returning to free roam, which closes the gameplay loop and supports repeated learning sessions.

- BattleState state; private int currentAction; private int currentMove; private int currentMember; Question currentQuestion;: These private variables track the current battle state, selected action/move (or quiz answer), party member, and the active quiz question. currentQuestion is particularly key for your educational adaptation, storing the quiz object during player turns to facilitate learning integration.

- PokemonParty playerParty; Pokemon wildPokemon;: These fields store references to the player's party and the wild enemy Pokémon, enabling access to their data (e.g., stats, questions) throughout the battle, crucial for quiz-based mechanics tied to Pokémon types.

- public void StartBattle(PokemonParty playerParty, Pokemon wildPokemon): This method initializes the battle by assigning the player's party and wild Pokémon, then starts the SetupBattle coroutine. It sets the stage for educational interactions by preparing the Pokémon that will provide quiz questions.

- public IEnumerator SetupBattle(): This coroutine sets up the battle scene by initializing player and enemy units with healthy Pokémon, configuring HUDs, initializing the party screen, and displaying an introductory dialog (e.g., "A wild [name] appeared!"). It then calls PlayerAction to begin the player's turn, establishing the battle's initial state and preparing for quiz integrations in subsequent phases.
  ```

- private void PlayerAction(): This method transitions to the PlayerAction state, sets a dialog prompt (”Choose an action”), and enables the action selector UI. It serves as the entry point for player decisions, leading into quiz-based moves or other actions like switching or escaping.
- void OpenPartyScreen(): This method shifts to the PartyScreen state, populates the party screen with Pokémon data, and activates it. It facilitates switching Pokémon, which can be strategic in your educational context by allowing selection of ones with specific subject-typed questions.
- IEnumerator PlayerMove(): This coroutine handles the player’s move phase, a critical part of your quiz integration. It sets the state to PlayerMove, retrieves a random question from the player’s Pokémon using GetRandomQuestion() (which ensures variety and prevents repetition, key for engagement). If no question is available, it displays a fallback message; otherwise, it disables the action selector, enables the move selector, populates choices with the question’s answers via SetChoiceNames, and types the question text. This method is thoroughly central to your educational design, transforming traditional moves into interactive quizzes that teach Math, English, or Science, with the yield ensuring the dialog types smoothly before player input.
- IEnumerator PerformPlayerMove(): This coroutine executes the player’s selected move based on the quiz outcome, marking a pivotal educational mechanic. It sets the state to Busy, checks if the selected answer (currentMove) matches the correct index (currentQuestion.Base.CorrectIndex). If correct, it picks a random move, displays success feedback, performs animations (attack via PlayerAttackAnimation, enemy hit via PlayerHitAnimation), calculates and applies damage using TakeDamage, updates the enemy HUD with a smooth HP animation, and checks for fainting (triggering win if yes). If incorrect, it shows failure feedback without damage. It then proceeds to the enemy move or ends the battle, reinforcing learning through immediate consequences—correct answers deal damage, promoting cognitive skills and resilience as per your thesis.
- IEnumerator RunMoves (BattleUnit sourceUnit, BattleUnit targetUnit, Move move): This coroutine (truncated) handles move execution for both player and enemy, including dialog, animations, damage calculation, HP updates, and fainting checks. In your quiz context, it’s invoked after a correct answer, blending traditional battle logic with educational outcomes.
- IEnumerator CheckForBattleOver(BattleUnit faintedUnit): This coroutine (truncated) checks if a unit fainted, plays faint animation, displays faint message, and ends the battle with win/loss via OnBattleOver. It supports the loop by transitioning out of battles, allowing return to exploration for more learning opportunities.
- IEnumerator EnemyMove(): This coroutine manages the enemy’s turn, selecting a random move, displaying it, performing animations and damage on the player, updating HUD, and checking for player faint (loss). It provides opposition to the player’s quiz-based strategy, maintaining challenge.

- void HandleUpdate(): This method processes input based on the current state—e.g., arrow keys and Z/X for selections in PlayerAction, PlayerMove, or PartyScreen. For quiz phases (PlayerMove), it handles answer selection, confirmation (Z for submit, triggering PerformPlayerMove), or cancel (X back to actions), ensuring intuitive controls for educational interactions.
- void HandlePartySelection(): This method manages party screen input (arrow keys for navigation, Z to switch, X to cancel), with checks for invalid selections (fainted or same Pokémon). It updates selection visually and starts SwitchPokemon on confirm, supporting strategic depth.
- IEnumerator SwitchPokemon(Pokemon newPokemon): This coroutine recalls the current Pokémon (with dialog and faint animation if needed), sets up the new one, updates HUD, and displays a send-out dialog before enemy move. It enables team management, tying into resilience by allowing recovery from quiz failures.
- IEnumerator TryToEscape(): This coroutine attempts escape based on speed comparison (success if player faster), with dialog and OnBattleOver(true) on success. It offers an alternative to battles, useful for pacing educational sessions.

The completed and fully functional Battle System thus looks like this:

Figure 4.4: Completed Battle System with Dynamic Health Bars and Functioning Quiz System

One final feature was to add the Party Selection UI that would also dynamically display all the Pokémon the player has, along with their current health.
For this, a PartyMemberUI.cs script was created:

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;
public class PartyMemberUI : MonoBehaviour
{
```

```
    [SerializeField] Text nameText;
    [SerializeField] Text levelText;
    [SerializeField] HPBar hpBar;
    [SerializeField] Color highlightedColor;
    Pokemon _pokemon;
    public void SetData(Pokemon pokemon)
    {
        _pokemon = pokemon;
        nameText.text = pokemon.Base.Name;
        levelText.text = "Lvl" + pokemon.Level;
        hpBar.SetHP((float)pokemon.HP / pokemon.MaxHP);
    }
    public void SetSelected(bool selected)
    {
        if (selected)
        {
            nameText.color = highlightedColor;
        }
        else
        {
            nameText.color = Color.black;
        }
    }
}
```

- public class PartyMemberUI : MonoBehaviour: This class, attached to individual UI slots on the party screen, manages the display of a single Pokémon's data (name, level, HP bar) and selection highlighting. It supports the party management system, crucial for strategic switching in your educational battles.
- `[SerializeField] Text nameText;`

  : This serialized field links to a UI Text component for displaying the Pokémon's name, set in the Unity inspector, providing clear identification for players.
- `[SerializeField] Text levelText;`

  : This serialized field references a UI Text component for showing the Pokémon's level, editable in the inspector, helping players track progress in the educational context.
- `[SerializeField] HPBar hpBar;`

  : This serialized field connects to an HPBar component, set in the inspector, to visualize the Pokémon's health, reflecting quiz outcomes in battles.
- `[SerializeField] Color highlightedColor;`

  : This serialized field, configurable in the inspector, defines the color for highlighting selected Pokémon, enhancing visual feedback for navigation, important for accessibility.
- `Pokemon _pokemon;`

: This private field stores the Pokémon instance associated with this UI slot, used to populate and update display data.

- public void SetData(Pokemon pokemon): This method assigns the Pokémon, sets nameText to the Pokémon's name, levelText to "Lvl" plus level, and hpBar to the normalized HP (HP/MaxHP). It initializes the UI, showing players their team's status, which reflects learning progress in quiz battles.
- public void SetSelected(bool selected): This method changes nameText's color to highlightedColor if selected, or black if not, providing visual feedback for party navigation, aiding intuitive control for children with learning disabilities.

As well as a script for showing the party screen, PartyScreen.cs:

```
using NUnit.Framework;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PartyScreen : MonoBehaviour
{
    [SerializeField] Text messageText;
    PartyMemberUI[] members;
    List<Pokemon> pokemons;

    public void Init()
    {
        members = GetComponentsInChildren<PartyMemberUI>();
    }
    public void SetPartyData(List<Pokemon> pokemons)
    {
        this.pokemons = pokemons;
        for (int i = 0; i < members.Length; i++)
        {
            if (i < pokemons.Count)
            {
                members[i].SetData(pokemons[i]);
            }
            else
            {
                members[i].gameObject.SetActive(false);
            }
        }
        messageText.text = "Choose a Pokemon.";
    }
    public void UpdateMemberSelection(int selectedMember)
    {
        for (int i = 0; i < pokemons.Count; i++)
        {
            if (i == selectedMember)
            {
```

```
                members[i].SetSelected(true);
            }
            else
            {
                members[i].SetSelected(false);
            }
        }
    }
    public void SetMessageText(string message)
    {
        messageText.text = message;
    }
}
```

- public class PartyScreen : MonoBehaviour: This class, attached to the party screen UI, manages the display and selection of the player's Pokémon team, enabling switching during battles. It supports strategic choices in your educational game, where switching can align with subject-specific quiz strategies.
- `[SerializeField] Text messageText;`

  : This serialized field references a UI Text component for displaying instructions or feedback (e.g., "Choose a Pokemon."), editable in the inspector, guiding players through party management.
- `PartyMemberUI[] members;`

  : This array stores PartyMemberUI components for each party slot, initialized in Init, handling individual Pokémon displays.
- `List<Pokemon> pokemons;`

  : This field temporarily stores the player's Pokémon list, used to populate the UI during party screen activation.
- public void Init(): This method populates the members array with child PartyMemberUI components, preparing the screen for displaying the player's team.
- `public void SetPartyData(List<Pokemon> pokemons)`

  : This method assigns the Pokémon list, iterates through members, calls SetData for each valid Pokémon, and hides extra slots. It sets messageText to "Choose a Pokemon.", initializing the UI for team selection, reflecting quiz battle outcomes.
- public void UpdateMemberSelection(int selectedMember): This method loops through Pokémon, calling SetSelected on the corresponding PartyMemberUI to highlight the selected member, providing visual feedback for navigation, crucial for accessibility.
- public void SetMessageText(string message): This method updates messageText with the provided string, used for feedback like "You can't switch with a fainted Pokemon.", guiding players and supporting learning resilience.

The outcome of these codes is as follows:

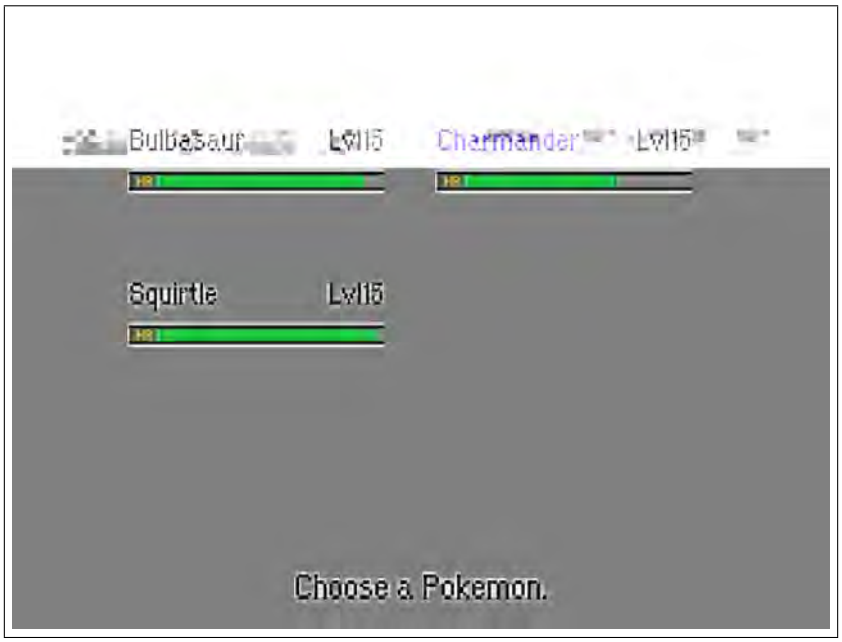


Figure 4.5: Dynamically Updating Party Screen

4.3.5 Adding Non-Player Characters and Dialogue System

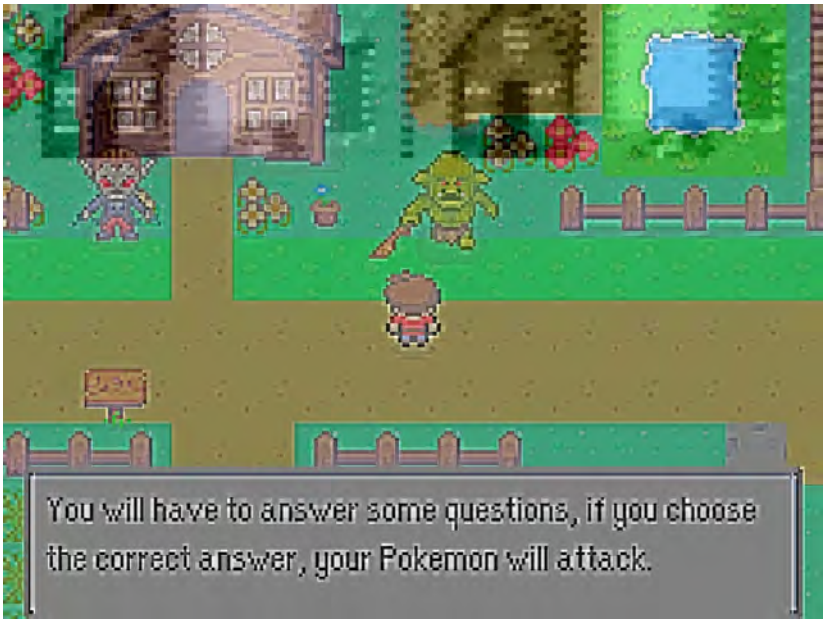


Figure 4.6: Instance of Interaction and Dialogue with an NPC

To make the game more enjoyable, I added NPCs and a simple dialogue system.
The idea was that several NPCs could provide instructions to the player, while one might have the ability to heal the player's Pokémon.
First, I created an NPCController.cs script for the NPC handling:

```
1 using System;
```

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

public class NPCController : MonoBehaviour, Interactable
{
    [SerializeField] Dialog dialog;
    [SerializeField] List<Vector2> movementPattern;
    [SerializeField] float timeBetweenPattern;
    Character character;

    NPCState state;
    float idleTimer = 0f;
    int currentPattern = 0;

    Healer healer;

    public void Awake()
    {
        character = GetComponent<Character>();
        healer = GetComponent<Healer>();
    }

    public IEnumerator Interact(Transform initiator)
    {
        if (state == NPCState.Idle)
        {
            state = NPCState.Dialog;
            character.LookTowards(initiator.position);

            if (healer != null)
            {
                yield return healer.Heal(initiator, dialog);
                    // Combine healer and dialog logic
            }
            else
            {
                yield return DialogManager.Instance.
                    ShowDialog(dialog);
            }

            idleTimer = 0f;
            state = NPCState.Idle;
        }
    }

    private void Update()
    {
        if (state == NPCState.Idle)
        {
            idleTimer += Time.deltaTime;
```

```
51              if (idleTimer > timeBetweenPattern)
52              {
53                  idleTimer = 0f;
54                  if (movementPattern != null &&
                        movementPattern.Count > 0)
55                  {
56                      StartCoroutine(walk());
57                  }
58              }
59              character.HandleUpdate();
60          }
61      }
62
63      IEnumerator walk()
64      {
65          state = NPCState.Walking;
66          var oldPosition = transform.position;
67          yield return character.Move(movementPattern[
                currentPattern]);
68          if (transform.position != oldPosition)
69          {
70              currentPattern = (currentPattern + 1) %
                    movementPattern.Count;
71          }
72          state = NPCState.Idle;
73      }
74
75      public enum NPCState { Idle, Walking, Dialog }
76  }
```

- public class NPCController : MonoBehaviour, Interactable: This class, inheriting from MonoBehaviour and implementing the Interactable interface, manages the behavior of non-player characters (NPCs) in your game, such as those providing dialog or healing services. It integrates with your educational Pokémon game by facilitating interactions that can enhance learning, such as quizzes or health recovery, aligning with your thesis's focus on supporting children with learning disabilities.

- `[SerializeField] Dialog dialog;`

  : This serialized field references a Dialog asset in the Unity inspector, containing text or choices for NPC interactions (e.g., a healing prompt or lore). It enables customizable NPC conversations, supporting educational narratives or instructions, which are key for engagement.

- `[SerializeField] List<Vector2> movementPattern;`

  : This serialized list stores movement directions (e.g., up, left) as Vector2 values, editable in the inspector, defining the NPC's patrol path. It allows for dynamic behavior, adding life to the game world and maintaining player interest.

- `[SerializeField] float timeBetweenPattern;`

: This serialized float, set in the inspector, determines the delay (in seconds) between movement pattern steps, controlling the NPC's pacing. It supports varied interaction timing, which can be adjusted for accessibility.

- Character character;: This field references the Character component attached to the NPC, handling shared movement logic, ensuring consistent behavior across entities.

- ```
  NPCState state;
  float idleTimer = 0f;
  int currentPattern = 0;
  ```

  : These variables track the NPC's state (Idle, Walking, Dialog), time since last action, and current movement pattern index, respectively. They manage the NPC's behavior cycle, including interaction readiness.

- Healer healer;: This field references a Healer component (if present), enabling healing functionality alongside dialog, enhancing gameplay by supporting Pokémon recovery post-quiz battles.

- public void Awake(): This method initializes character and healer components, ensuring the NPC is ready for movement and interaction logic, preparing it for educational or support roles.

- public IEnumerator Interact(Transform initiator): This method, required by Interactable, handles player interaction. When idle, it sets the state to Dialog, makes the NPC face the player, and either triggers healing (via healer.Heal if present) or shows the dialog (via DialogManager). It resets the idle timer and state post-interaction, supporting learning through dialog or health restoration.

- private void Update(): This method updates the NPC's behavior when idle: increments idleTimer, triggers movement if exceeding timeBetweenPattern, and calls character.HandleUpdate for animation updates. It ensures the NPC remains active, enhancing the game's immersive environment.

- IEnumerator walk(): This coroutine handles NPC movement: sets state to Walking, moves to the next pattern position using character.Move, updates currentPattern if movement succeeds (looping via modulo), and returns to Idle. The animation works by leveraging Character.Move's lerp-based movement over 0.3 seconds, syncing with CharacterAnimator for directional sprite changes, creating a smooth patrol effect.

- `public enum NPCState { Idle, Walking, Dialog }`

  : This enumeration defines the NPC's possible states, guiding its behavior (waiting, moving, or dialoging). It structures the interaction and movement flow, supporting a responsive game world.

For the dialogue system, at first, I had to make sure the player could interact with an NPC to initiate conversations. For that, I created an Interactable.cs file. This is referenced in the NPCController.cs and PlayerController.cs scripts, and initiates the interaction function that opens the dialogue box.
  ```

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

public interface Interactable
{
    IEnumerator Interact(Transform initiator);

}
```

- public interface Interactable: This interface defines a contract for objects that players can interact with (e.g., NPCs, signs), requiring an Interact method. It standardizes interaction behavior, enabling your game to trigger educational dialogs or actions during exploration.
- IEnumerator Interact(Transform initiator): This method, implemented by interactable objects, takes the initiator's transform (e.g., player position) and returns a coroutine for interaction logic (e.g., showing dialog). It supports flexible interactions, such as initiating quizzes or healing, aligning with your thesis's goal of engaging learning experiences.

Now, the next step was to build the dialogue system. The main dialogue is handled with the DialogManager.cs code:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class DialogManager : MonoBehaviour
{
    [SerializeField] GameObject dialogBox;
    [SerializeField] ChoiceBox choiceBox;
    [SerializeField] Text dialogText;
    [SerializeField] int lettersPerSecond;

    public event Action OnShowDialog;
    public event Action OnCloseDialog;

    public static DialogManager Instance { get; private set;
       }

    private void Awake()
    {
        Instance = this;
    }

    public bool IsShowing { get; private set; }

```

```
26      public IEnumerator ShowDialogText(string text, bool
            waitForInput = true, bool autoClose = true)
27      {
28          OnShowDialog?.Invoke();
29          IsShowing = true;
30          dialogBox.SetActive(true);
31          yield return TypeDialog(text);
32          if (waitForInput)
33          {
34              yield return new WaitUntil(() => Input.GetKeyDown
                    (KeyCode.Z));
35          }
36          if (autoClose)
37          {
38              CloseDialog();
39          }
40      }
41
42      public void CloseDialog()
43      {
44          dialogBox.SetActive(false);
45          IsShowing = false;
46          OnCloseDialog?.Invoke();
47      }
48
49      public IEnumerator ShowDialog(Dialog dialog, List<string>
            choices = null, Action<int> onChoiceSelected = null)
50      {
51          yield return new WaitForEndOfFrame();
52          OnShowDialog?.Invoke();
53          IsShowing = true;
54          dialogBox.SetActive(true);
55
56          foreach (var line in dialog.Lines)
57          {
58              yield return TypeDialog(line);
59              yield return new WaitUntil(() => Input.GetKeyDown
                    (KeyCode.Z));
60          }
61
62          if (choices != null && choices.Count > 1)
63          {
64              yield return choiceBox.ShowChoices(choices,
                    onChoiceSelected);
65          }
66
67          CloseDialog();
68      }
69
70      public IEnumerator TypeDialog(string line)
71      {
```

```
72          dialogText.text = "";
73          foreach (var letter in line.ToCharArray())
74          {
75              dialogText.text += letter;
76              yield return new WaitForSeconds(1f /
                    lettersPerSecond);
77          }
78      }
79  }
```

- public class DialogManager : MonoBehaviour: This singleton class manages dialog display, including text typing and choice selection, used for NPC interactions and system messages like the exit prompt. It's key for delivering educational instructions or quiz context in your game.
- `[SerializeField] GameObject dialogBox;`
  `[SerializeField] ChoiceBox choiceBox;`
  `[SerializeField] Text dialogText;`
  `[SerializeField] int lettersPerSecond;`

  : These serialized fields reference the dialog UI, ChoiceBox for selections, text component, and typing speed, all editable in the inspector. They configure the dialog system for clear, paced presentation.
- public event Action OnShowDialog; public event Action OnCloseDialog;: These events notify other systems (e.g., GameController) when dialogs start or end, enabling state changes (e.g., to Dialog state), ensuring smooth integration with gameplay.
- public static DialogManager Instance get; private set; : This singleton property ensures a single DialogManager instance, accessible globally, facilitating consistent dialog management across scenes.
- public bool IsShowing get; private set; : This property tracks if a dialog is active, used to manage input and visibility, preventing conflicts during educational interactions.
- public IEnumerator ShowDialogText(string text, bool waitForInput = true, bool autoClose = true): This coroutine shows a single text line, invokes OnShowDialog, activates the dialog box, types the text, waits for Z input if required, and closes if autoClose is true. It supports concise messages, such as quiz instructions, with a typing effect for engagement.
- public void CloseDialog(): This method hides the dialog box, clears IsShowing, and invokes OnCloseDialog, resetting the state for gameplay continuation, ensuring a smooth return to exploration or battles.
- `public IEnumerator ShowDialog(Dialog dialog,`
  `List<string> choices = null,`
  `Action<int> onChoiceSelected = null)`

: This coroutine displays a multi-line dialog, showing each line with typing, waiting for Z input, and optionally presenting choices via ChoiceBox. It supports complex interactions like NPC conversations or quiz setups, enhancing educational delivery.

- public IEnumerator TypeDialog(string line): This coroutine types a line character by character at lettersPerSecond speed, creating an engaging, readable effect for instructions or questions, crucial for accessibility in your learning-focused game.

Then, a Dialog.cs code to handle the dialogue texts:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

[System.Serializable]
public class Dialog
{
    [SerializeField] List<string> lines;
    public List<string> Lines {
        get { return lines; }
    }

}
```

- `[System.Serializable] public class Dialog`

  : This serializable class defines a dialog asset, allowing it to be created and edited in the Unity inspector. It serves as a container for conversation or instructional text, supporting narrative and educational content delivery.

- `[SerializeField] List<string> lines;`

  : This serialized field stores a list of dialog lines, editable in the inspector, representing sequential text (e.g., NPC speech or quiz instructions). It enables customizable content for learning interactions.

- `public List<string> Lines { get { return lines; } }`

  : This read-only property exposes the dialog lines to other scripts, ensuring controlled access for display by DialogManager, supporting clear communication in your educational design.

Then, I handled the ability to give the player some dialogue choices. This choice system is implemented when the player wants to exit the game, as well as the interaction with the Healer NPC.
First, I created ChoiceBox.cs script:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
```

```
6
7  public class ChoiceBox : MonoBehaviour
8  {
9      [SerializeField] ChoiceText choiceTextPrefab;
10
11     bool choiceSelected = false;
12     List<ChoiceText> choiceTexts;
13     int currentChoice;
14
15     private void OnValidate()
16     {
17         if (choiceTextPrefab != null && choiceTextPrefab.
               GetComponent<Text>() == null)
18         {
19             Debug.LogError($"ChoiceTextPrefab on {gameObject.
                   name} must have a Text component.", this);
20         }
21         if (choiceTextPrefab != null && choiceTextPrefab.
               GetComponent<ChoiceText>() == null)
22         {
23             Debug.LogError($"ChoiceTextPrefab on {gameObject.
                   name} must have a ChoiceText component.", this
                   );
24         }
25     }
26
27     public IEnumerator ShowChoices(List<string> choices,
           Action<int> onChoiceSelected)
28     {
29         if (choices == null || choices.Count == 0)
30         {
31             Debug.LogWarning($"ChoiceBox: No choices provided
                   to ShowChoices on {gameObject.name}.", this);
32             onChoiceSelected?.Invoke(-1);
33             yield break;
34         }
35
36         if (choiceTextPrefab == null)
37         {
38             Debug.LogError($"ChoiceTextPrefab is not assigned
                   on {gameObject.name}. Assign a valid prefab
                   with ChoiceText and Text components.", this);
39             onChoiceSelected?.Invoke(-1);
40             yield break;
41         }
42
43         gameObject.SetActive(true);
44         choiceSelected = false;
45         currentChoice = 0;
46
47         foreach (Transform child in transform)
```

```
48          {
49              Destroy(child.gameObject);
50          }
51          choiceTexts = new List<ChoiceText>();
52          foreach (var choice in choices)
53          {
54              var choiceTextObj = Instantiate(choiceTextPrefab,
                    transform);
55              var choiceTextComponent = choiceTextObj.
                   GetComponent<ChoiceText>();
56              if (choiceTextComponent == null)
57              {
58                  Debug.LogError($"Instantiated ChoiceText
                       prefab on {gameObject.name} lacks
                       ChoiceText component.", choiceTextObj);
59                  Destroy(choiceTextObj.gameObject);
60                  continue;
61              }
62              if (choiceTextComponent.TextField != null)
63              {
64                  choiceTextComponent.TextField.text = choice;
65              }
66              else
67              {
68                  Debug.LogError($"Instantiated ChoiceText
                       prefab on {gameObject.name} lacks Text
                       component.", choiceTextObj);
69                  Destroy(choiceTextObj.gameObject);
70                  continue;
71              }
72              choiceTexts.Add(choiceTextComponent);
73          }
74
75          if (choiceTexts.Count == 0)
76          {
77              Debug.LogError($"No valid ChoiceText instances
                   created on {gameObject.name}. Check
                   ChoiceTextPrefab configuration.", this);
78              gameObject.SetActive(false);
79              onChoiceSelected?.Invoke(-1);
80              yield break;
81          }
82
83          for (int i = 0; i < choiceTexts.Count; i++)
84          {
85              choiceTexts[i].SetSelected(i == currentChoice);
86          }
87
88          yield return new WaitUntil(() => choiceSelected);
89
90          onChoiceSelected?.Invoke(currentChoice);
```

```
91          gameObject.SetActive(false);
92      }
93
94      private void Update()
95      {
96          if (choiceTexts == null || choiceTexts.Count == 0)
97          {
98              return;
99          }
100
101         if (Input.GetKeyDown(KeyCode.DownArrow))
102         {
103             currentChoice = Mathf.Min(currentChoice + 1,
                    choiceTexts.Count - 1);
104         }
105         else if (Input.GetKeyDown(KeyCode.UpArrow))
106         {
107             currentChoice = Mathf.Max(currentChoice - 1, 0);
108         }
109
110         for (int i = 0; i < choiceTexts.Count; i++)
111         {
112             choiceTexts[i].SetSelected(i == currentChoice);
113         }
114
115         if (Input.GetKeyDown(KeyCode.Z))
116         {
117             choiceSelected = true;
118             Debug.Log($"ChoiceBox: Selected choice {
                    currentChoice} on {gameObject.name}");
119         }
120     }
121 }
```

- public class ChoiceBox : MonoBehaviour: This class, attached to a UI GameObject, manages a dynamic choice selection interface for multiple-choice inputs, such as quiz answers in your educational battles. It instantiates and updates text options, handles navigation, and processes selections, directly supporting the interactive learning mechanics central to your thesis.
- `[SerializeField] ChoiceText choiceTextPrefab;`

  : This serialized field references a prefab with a ChoiceText component, used to create individual choice text elements. Editable in the Unity inspector, it ensures consistent UI creation for quiz answers or dialog choices.
- `bool choiceSelected = false;`
  `List<ChoiceText> choiceTexts;`
  `int currentChoice;`

  : These variables track whether a choice has been selected, store the list of instantiated ChoiceText objects, and hold the currently selected choice index. They manage the state and navigation of the choice UI, crucial for quiz interactions.

- private void OnValidate(): This method runs in the Unity editor to validate the choiceTextPrefab, logging errors if it lacks Text or ChoiceText components. It ensures proper setup, preventing runtime issues in the educational UI.
- `public IEnumerator ShowChoices(List<string> choices, Action<int> onChoiceSelected)`

  : This coroutine displays a list of choices (e.g., quiz answers), creating a ChoiceText instance for each, setting their text, and initializing selection. It handles navigation via Update, waits for a choice with Z, then invokes the callback with the selected index and hides the UI. For your quiz system, it presents answers (e.g., Math problem options), enabling interactive learning by capturing user input, with robust error handling for invalid setups.
- private void Update(): This method processes input when choices are shown: DownArrow/UpArrow adjusts currentChoice within bounds, updates selection visuals via SetSelected, and Z confirms the choice. It ensures intuitive navigation, supporting accessibility for children with learning challenges.

And then, I created a ChoiceText.cs script to handle the choice text elements:

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ChoiceText : MonoBehaviour
{
    Text text;

    private void Awake()
    {
        text = GetComponent<Text>();
        if (text == null)
        {
            Debug.LogError($"No Text component found on {
               gameObject.name}. Ensure the ChoiceText prefab
                has a Text component.", this);
        }
    }

    public Text TextField => text;

    public void SetSelected(bool selected)
    {
        if (text != null)
        {
            text.color = selected ? GetHighlightedColor() :
               Color.black;
        }
        else
```

```
        {
            Debug.LogWarning($"Cannot set selected state on {
                gameObject.name}: No Text component.", this);
        }
    }

    private Color GetHighlightedColor()
    {
        if (GlobalSettings.Instance == null)
        {
            Debug.LogWarning($"GlobalSettings.Instance is
                null on {gameObject.name}. Using default
                highlight color (yellow).", this);
            return Color.green;
        }
        return GlobalSettings.Instance.HighlightedColor;
    }
}
```

- public class ChoiceText : MonoBehaviour: This class, attached to choice text prefabs, manages individual choice UI elements (e.g., quiz answer options). It handles text display and selection highlighting, crucial for the interactive quiz system.
- Text text;: This field references the UI Text component for displaying choice text, initialized in Awake.
- private void Awake(): This method gets the Text component, logging an error if missing, ensuring the prefab is properly configured for choice display.
- `public Text TextField => text;`

  : This read-only property provides access to the Text component, allowing external scripts (e.g., ChoiceBox) to set choice text, supporting quiz answer presentation.
- public void SetSelected(bool selected): This method changes the text color to a highlighted color (from GlobalSettings) if selected, or black if not, logging warnings if components are missing. It provides visual feedback for choice navigation, enhancing accessibility for your target audience.
- private Color GetHighlightedColor(): This method retrieves the highlight color from GlobalSettings, defaulting to green if unavailable, ensuring consistent UI styling for selected quiz answers.

Finally, I added an NPC who could heal the Pokemon of the player. As seen in the NPCController.cs code above, a Healer component is already placed. That is a reference to this Healer.cs code:

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

```

```
public class Healer : MonoBehaviour
{
    public IEnumerator Heal(Transform player, Dialog dialog)
    {
        int selectedChoice = 0;
        yield return DialogManager.Instance.ShowDialog(dialog
            , new List<string> { "Yes", "No" }, (choiceIndex)
            => selectedChoice = choiceIndex);

        var playerParty = player.GetComponent<PokemonParty>()
            ;
        if (playerParty == null)
        {
            Debug.LogError($"PokemonParty component missing
                on {player.gameObject.name}", player.
                gameObject);
            yield break;
        }

        if (selectedChoice == 0)
        {
            foreach (var pokemon in playerParty.Pokemons)
            {
                pokemon.Heal();
                Debug.Log($"Healed {pokemon.Base.Name} to {
                    pokemon.HP} HP");
            }
            playerParty.PartyUpdated();
            yield return DialogManager.Instance.
                ShowDialogText("Your Pok mon are fully healed
                !");
        }
        else
        {
            yield return DialogManager.Instance.
                ShowDialogText("Come back anytime!");
        }
    }
}
```

- public class Healer : MonoBehaviour: This class, attached to NPC GameObjects with healing functionality, manages the process of restoring the player's Pokémon party to full health. It integrates with the dialog system to offer an interactive healing option, supporting gameplay continuity and player engagement in your educational Pokémon-inspired game, particularly after quiz-based battles that may deplete health.
- public IEnumerator Heal(Transform player, Dialog dialog): This coroutine handles the healing interaction, taking the player's transform and a dialog asset as parameters. It presents a choice to heal, processes the selection, heals the party if confirmed, and displays appropriate feedback, reinforcing the supportive

environment for children with learning disabilities by ensuring they can recover and continue learning.

- int selectedChoice = 0;: This variable tracks the player's dialog choice (0 for "Yes," 1 for "No") during the healing prompt, initialized to 0 and updated via a callback from DialogManager, enabling decision-based interaction.

- ```
  yield return DialogManager.Instance.ShowDialog(dialog,
  new List<string>
  { "Yes", "No" },
  (choiceIndex) => selectedChoice = choiceIndex);
  ```

  : This line triggers a dialog with "Yes"/"No" choices using DialogManager, capturing the player's selection via a callback. It engages players in a decision-making process, enhancing interactivity and supporting cognitive engagement as per your thesis's goals.

- ```
  var playerParty = player.GetComponent<PokemonParty>();
  ```

  : This retrieves the PokemonParty component from the player GameObject, which contains the player's Pokémon list. It's essential for accessing and modifying Pokémon health during healing.

- ```
  if (playerParty == null)
  { Debug.LogError(...); yield break; }
  ```

  : This checks if the PokemonParty component exists, logging an error and exiting if missing, ensuring robust error handling to prevent runtime issues in the healing process.

- ```
  if (selectedChoice == 0)
  { foreach (var pokemon in playerParty.Pokemons)
  { pokemon.Heal(); Debug.Log(...); } playerParty.PartyUpdated();
  yield return DialogManager.Instance.ShowDialogText
  ("Your Pokémon are fully healed!"); }
  ```

  : If the player chooses "Yes" (0), this block iterates through the party's Pokémon, calls Heal() on each to restore full HP, logs the healing for debugging, and notifies the party of updates via PartyUpdated(). It then displays a confirmation message via DialogManager, providing clear feedback that supports emotional resilience by rewarding players with recovery after challenging quiz battles.

- ```
  else { yield return DialogManager.Instance.ShowDialogText
  ("Come back anytime!"); }
  ```

  :If the player chooses "No" (1), this displays a friendly message via DialogManager, maintaining a positive tone and encouraging continued exploration, which aligns with creating a supportive learning environment.
  ```

How the healer NPC interaction looks within the demo:

Figure 4.7: Dialogue Choices with Healer NPC

4.3.6 Completing The Game World and Final Touches

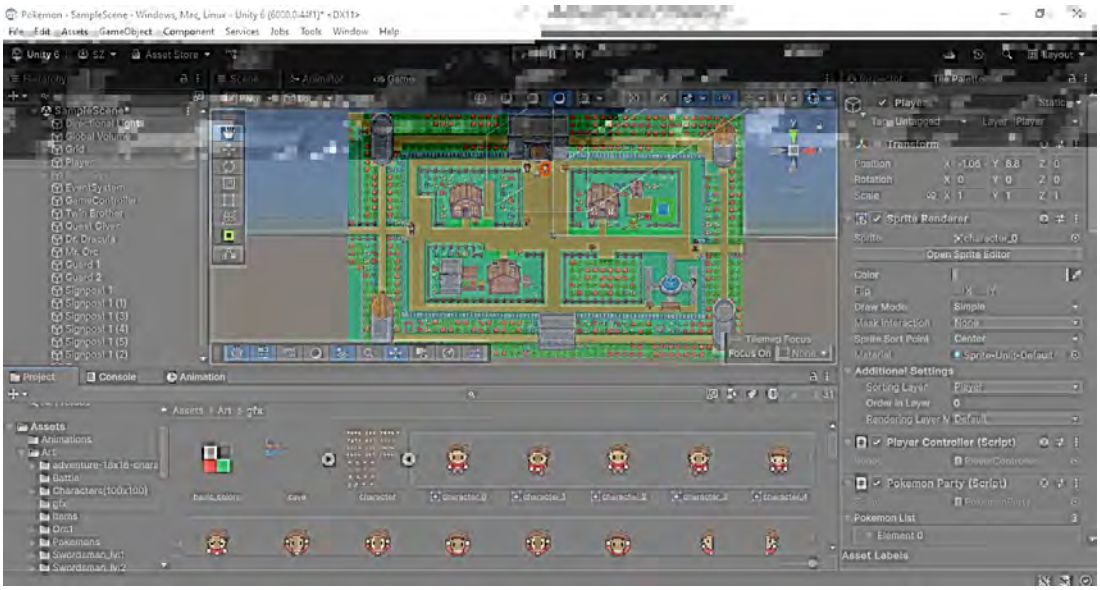


Figure 4.8: The Final Stages of Development of The Demo

To complete the project, all of these scripts needed to work in tandem with one another.

The GameController.cs script controls the player movement, transition to combat, and dialogue toggles:

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

public enum GameState { FreeRoam, Battle, Dialog }

public class GameController : MonoBehaviour
{
    [SerializeField] PlayerController playerController;
    [SerializeField] BattleSystem battleSystem;
    [SerializeField] Camera worldCamera;
    [SerializeField] Dialog exitDialog;

    GameState state;

    private void Start()
    {
        playerController.OnEncountered += StartBattle;
        battleSystem.OnBattleOver += EndBattle;

        DialogManager.Instance.OnShowDialog += () =>
        {
            state = GameState.Dialog;
        };
        DialogManager.Instance.OnCloseDialog += () =>
        {
            if (state == GameState.Dialog)
            {
                state = GameState.FreeRoam;
            }
        };
    }

    void StartBattle()
    {
        state = GameState.Battle;
        battleSystem.gameObject.SetActive(true);
        worldCamera.gameObject.SetActive(false);
        var playerParty = playerController.GetComponent<
           PokemonParty>();
        var wildPokemon = FindAnyObjectByType<MapArea>().
           GetComponent<MapArea>().GetRandomWildPokemon();
        battleSystem.StartBattle(playerParty, wildPokemon);
    }

    void EndBattle(bool won)
    {
        state = GameState.FreeRoam;
```

```
48          battleSystem.gameObject.SetActive(false);
49          worldCamera.gameObject.SetActive(true);
50      }
51
52      private void Update()
53      {
54          if (state == GameState.FreeRoam)
55          {
56              playerController.HandleUpdate();
57              if (Input.GetKeyDown(KeyCode.Escape))
58              {
59                  StartCoroutine(ShowExitDialog());
60              }
61          }
62          else if (state == GameState.Battle)
63          {
64              battleSystem.HandleUpdate();
65          }
66          else if (state == GameState.Dialog)
67          {
68              //
69          }
70      }
71
72      private IEnumerator ShowExitDialog()
73      {
74
75          int selectedChoice = 0;
76          yield return DialogManager.Instance.ShowDialogText("
               Do you want to exit the game?");
77          yield return DialogManager.Instance.ShowDialog(
               exitDialog, new List<string> { "Yes", "No" }, (
               choiceIndex) => selectedChoice = choiceIndex);
78
79          if (selectedChoice == 0) // Yes
80          {
81              yield return DialogManager.Instance.
                   ShowDialogText("Exiting game...");
82              Application.Quit();
83  #if UNITY_EDITOR
84              UnityEditor.EditorApplication.isPlaying = false;
85  #endif
86          }
87          else // No
88          {
89              yield return DialogManager.Instance.
                   ShowDialogText("Returning to game...");
90          }
91      }
92
93      internal void SetActive(bool v)
```

```
94      {
95          throw new NotImplementedException();
96      }
97  }
```

- `public enum GameState { FreeRoam, Battle, Dialog }`

  : This enumeration defines the game's primary states: free roaming (exploration), battling (including quiz interactions), and dialog (for NPC conversations or system messages). It structures the game's flow, ensuring smooth transitions, such as from exploration to battles triggered by encounters, which is essential for your educational design where battles integrate learning.

- public class GameController : MonoBehaviour: This class, inheriting from MonoBehaviour, acts as the main game manager, handling state changes, battle initiation, dialog events, and the escape menu. It coordinates core systems like player movement, battles, and dialogs, providing a high-level framework for your Pokémon-inspired game with educational elements.

- ```
  [SerializeField] PlayerController playerController;
  [SerializeField] BattleSystem battleSystem;
  [SerializeField] Camera worldCamera;
  [SerializeField] Dialog exitDialog;
  ```

  : These serialized fields reference key components: the player controller for movement, battle system for combat, world camera for exploration view, and a dialog asset for the exit prompt. They are editable in the Unity inspector, enabling easy configuration and integration of educational battles.

- GameState state;: This variable tracks the current game state, used to determine active behaviors (e.g., player input in FreeRoam or battle logic in Battle), ensuring the game responds appropriately to events like encounters or dialogs.

- private void Start(): This method subscribes to events: OnEncountered from the player controller to start battles, OnBattleOver from the battle system to end them, and dialog show/close events from DialogManager to switch states. It initializes the game's event-driven structure, supporting seamless shifts to educational battle modes.

- void StartBattle(): This method transitions to the Battle state, activates the battle system and deactivates the world camera, retrieves the player's party and a random wild Pokémon, and starts the battle. It triggers the educational quiz mechanics within battles, aligning with your thesis by initiating learning opportunities.

- void EndBattle(bool won): This method resets to FreeRoam, deactivates the battle system, and reactivates the world camera, concluding battles and returning to exploration for continued gameplay loops.

- private void Update(): This method checks the current state: in FreeRoam, it handles player updates and escape key presses to show the exit dialog; in Battle, it delegates to the battle system; in Dialog, it allows dialog handling. It ensures real-time input processing, with the escape feature providing a user-friendly exit option.
  ```

- private IEnumerator ShowExitDialog(): This coroutine displays an exit confirmation dialog using DialogManager, captures the choice, and either quits the application (with editor support) on ”Yes” or returns to the game on ”No”. It enhances usability, preventing accidental exits during educational sessions.
- internal void SetActive(bool v): This method throws a NotImplementedException, serving as a placeholder for potential future activation logic, though not used in the current demo.

The game operates on several layers that have been set on Unity. The Background is in the Default Layer, the Player Character is in the playerLayer, the NPCs are in the interactableLayer the Objects such as houses and fences are in the solidObjectsLayer, and the Grass where Pokémon are encountered is in the grassLayer.
Here is the GameLayers.cs code:

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

public class GameLayers : MonoBehaviour
{
    [SerializeField] LayerMask solidObjectsLayer;
    [SerializeField] LayerMask interactableLayer;
    [SerializeField] LayerMask grassLayer;
    [SerializeField] LayerMask playerLayer;

    public static GameLayers i { get; set; }
    private void Awake()
    {
        i = this;
    }

    public LayerMask SolidLayer { get => solidObjectsLayer; }
    public LayerMask InteractableLayer { get =>
       interactableLayer; }
    public LayerMask GrassLayer { get => grassLayer; }
    public LayerMask PlayerLayer { get => playerLayer; }
}
```

- public class GameLayers : MonoBehaviour: This singleton class, inheriting from MonoBehaviour, centralizes access to layer masks used for collision and interaction detection throughout the game. It supports the exploration mechanics by defining layers for objects, interactables, grass, and the player, ensuring smooth navigation and encounter triggers that lead to educational battles.
- `[SerializeField] LayerMask solidObjectsLayer;`

  : This serialized field, editable in the Unity inspector, defines a layer mask for solid objects (e.g., walls, trees) that block movement. It’s used in collision checks to prevent characters from passing through obstacles, maintaining a consistent game world.

- `[SerializeField] LayerMask interactableLayer;`

  : This serialized field specifies a layer mask for interactable objects (e.g., NPCs, signs), set in the inspector. It enables detection for interactions like initiating dialogs or healing, which can support educational content delivery.

- `[SerializeField] LayerMask grassLayer;`

  : This serialized field, configurable in the inspector, marks the layer for grass areas where wild Pokémon encounters occur. It's critical for triggering battles with quiz-based mechanics, aligning with your thesis's focus on integrating learning through combat.

- `[SerializeField] LayerMask playerLayer;`

  : This serialized field defines the player's layer, set in the inspector, used for specific collision or interaction checks, ensuring the player is distinguished from other entities.

- `public static GameLayers i { get; set; }`

  : This static property establishes a singleton instance, set in Awake, allowing global access to layer masks from scripts like PlayerController for consistent collision logic.

- private void Awake(): This method sets the singleton instance (i) to the current object, ensuring only one GameLayers exists, providing a reliable reference for layer checks throughout the game.

- `public LayerMask SolidLayer { get => solidObjectsLayer; }`

  : This read-only property exposes the solidObjectsLayer, allowing scripts to access it for collision checks, such as in Character.IsPathClear, to prevent movement through walls.

- `public LayerMask InteractableLayer { get => interactableLayer; }`

  : This property provides access to the interactableLayer, used in scripts like PlayerController.Interact to detect NPCs or objects, facilitating educational interactions.

- `public LayerMask GrassLayer { get => interactableLayer; }`

  : This property, likely containing a typo (should return grassLayer), exposes the grass layer for encounter checks in PlayerController.CheckForEncounters, triggering battles with educational quizzes.

- `public LayerMask PlayerLayer { get => playerLayer; }`

  : This property provides access to the playerLayer, enabling specific checks for the player, supporting targeted interactions or collision logic.

Using the engine, I placed the assets to create a small explorable map. I added a total of six NPCs, who provide instructions to the player, along with a few "signposts" that use the same NPC controller code, just without the animations.
However, for added depth, I assigned one of the NPCs with the ability to heal the player's Pokémon party.

Finally, the main objective of the game was to fight enemy Pokémon within a fixed area, that being the garden area in the level. The elements of the garden were put in the aformentioned grassLayer. However, the Pokémon must be generated randomly within that area. This is handled by the MapArea.cs file:

```
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;

public class MapArea : MonoBehaviour
{
    [SerializeField] List<Pokemon> wildPokemon;

    public Pokemon GetRandomWildPokemon()
    {
        var wildPokemons = wildPokemon[Random.Range(0,
           wildPokemon.Count)];
        wildPokemons.Init();
        return wildPokemons;
    }
}
```

- public class MapArea : MonoBehaviour: This class, attached to a map area GameObject, manages the pool of wild Pokémon available for encounters in that area. It's crucial for initiating battles, which incorporate your quiz-based learning system, by providing random opponents.
- `[SerializeField] List<Pokemon> wildPokemon;`

  : This serialized list, editable in the Unity inspector, holds Pokemon objects representing possible wild Pokémon for the area. It allows customization of encounter pools, supporting varied educational challenges tied to Pokémon types.
- public Pokemon GetRandomWildPokemon(): This method selects a random Pokémon from wildPokemon using Random.Range, initializes it with Init(), and returns it. It ensures battles start with a fresh, randomized opponent, triggering quiz-based mechanics in BattleSystem, aligning with your thesis's goal of engaging learning through diverse encounters.

4.3.7 Sending The Demo to Testers

After completion of the demo, I shared it with several people and conducted a survey. The survey included questions regarding the gameplay, how effective it is in creating an intuitive and interactive learning experience, and so on.
A total of 13 people filled up the survey. Among them, 15.4% claimed to have some kind of learning disability or knew of someone who did. However, due to time and resource constraints, not enough data could be collected pertaining to the actual target demographic of the project (i.e., neuro-divergent and learning disabled children), and that is left for future consideration and development.

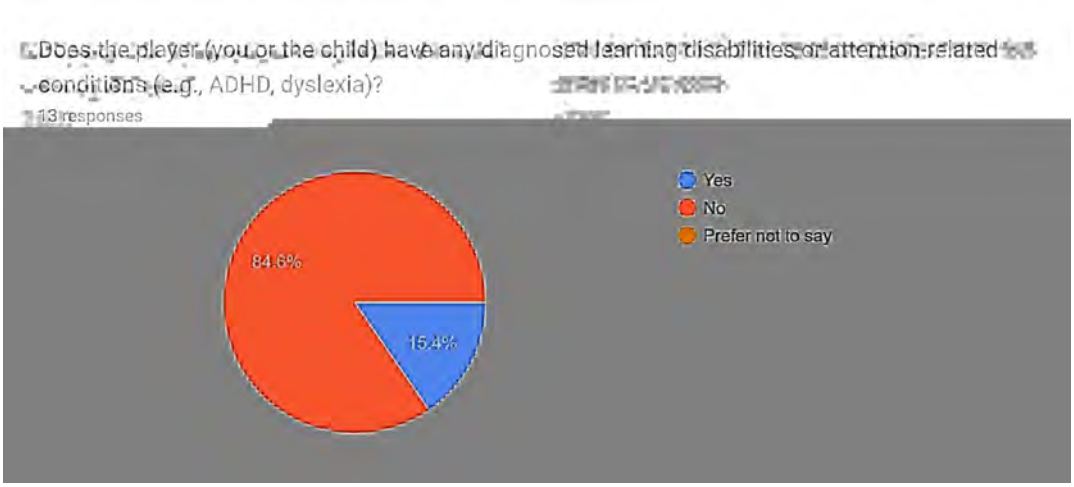


Figure 4.9: Survey Responders

The results of this survey are explored further in detail in the Result Anaylsis section.

Chapter 5

Result Analysis

5.1 Performance Evaluation

The survey results demonstrate an overall positive evaluation of the game's potential to support learning among children with attention or learning difficulties. The key evaluation metrics included **accuracy**, **efficiency**, and **reliability**, as assessed through quantitative ratings and qualitative feedback.
Performance Accuracy and Effectiveness: About 23% respondents reported some gameplay bugs.

Figure 5.1: Gameplay Issues Report

Their comments included "There was some issues with the responsiveness", and "It can be optimized further". One user reported an aspect ratio issue when an external software, such as an FPS (Frames Per Second) Counter was running simultaneously with the demo, stating, "Faced slight issue with aspect ratio, likely due to rivatuner fps counter clash. And bug with rivatuner didn't let me close it to retest."
Participants rated the ease of control between 1 and 3 on a 5-point scale (with 1 being Very Easy and 5 being Very Hard), reflecting an intuitive gameplay experience with minimal learning curve.

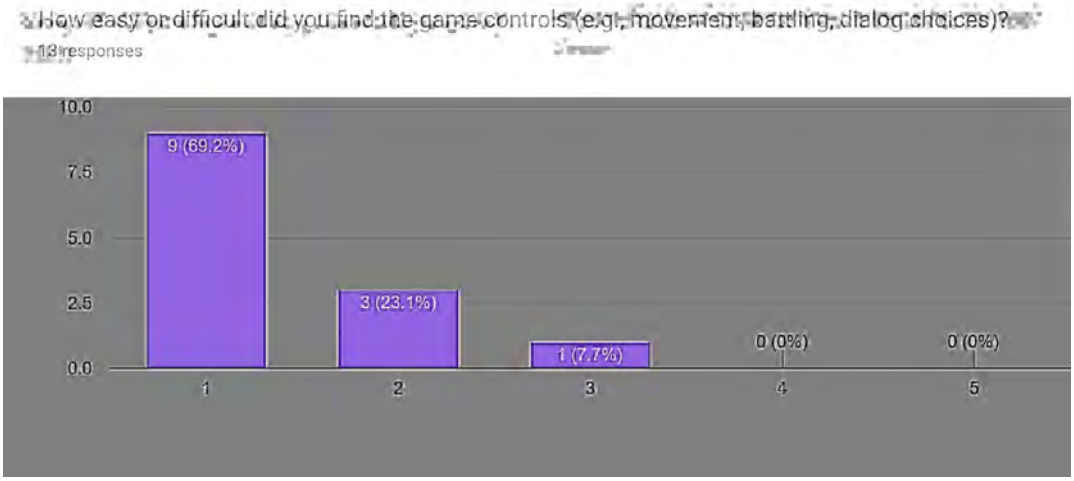


Figure 5.2: Difficulty of Controls

In terms of how effective it was to the core problem statement and theme of the project, most of the feedback was incredibly positive. 76.9% of the testers implied that the gameplay mechanics were helpful in learning.

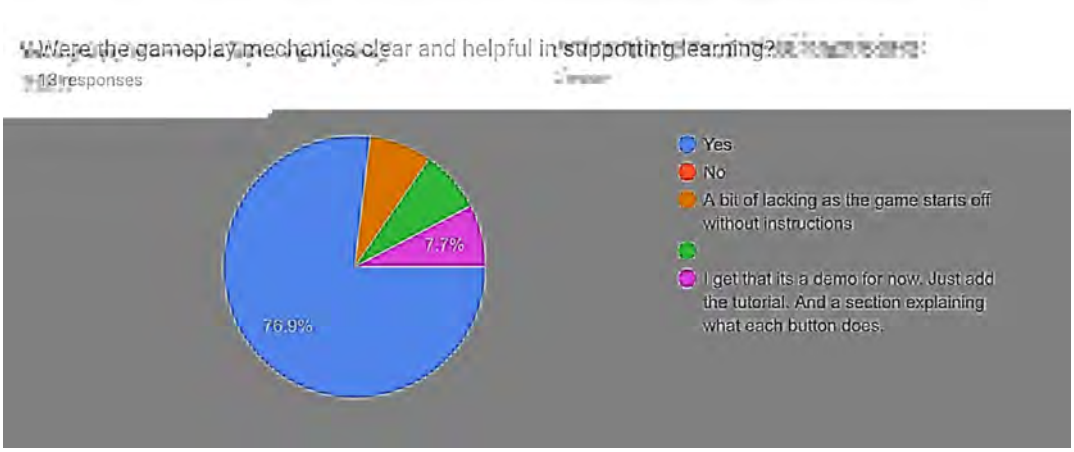


Figure 5.3: Gameplay Effectiveness

Meanwhile, 53.8% of the testers agreed that the core Quiz Game system and the questionnaire would be helpful for the target demographic.

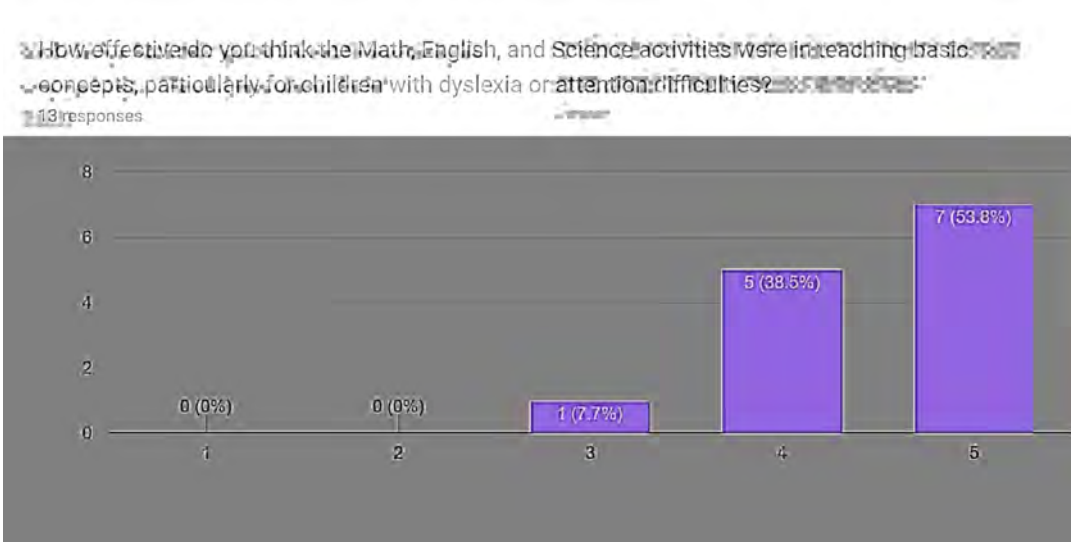


Figure 5.4: Question Effectiveness

In terms of general engagement, 61.5% of the testers rated the quiz game mechanics as a 4/5 (on a scale of 1 to 5).

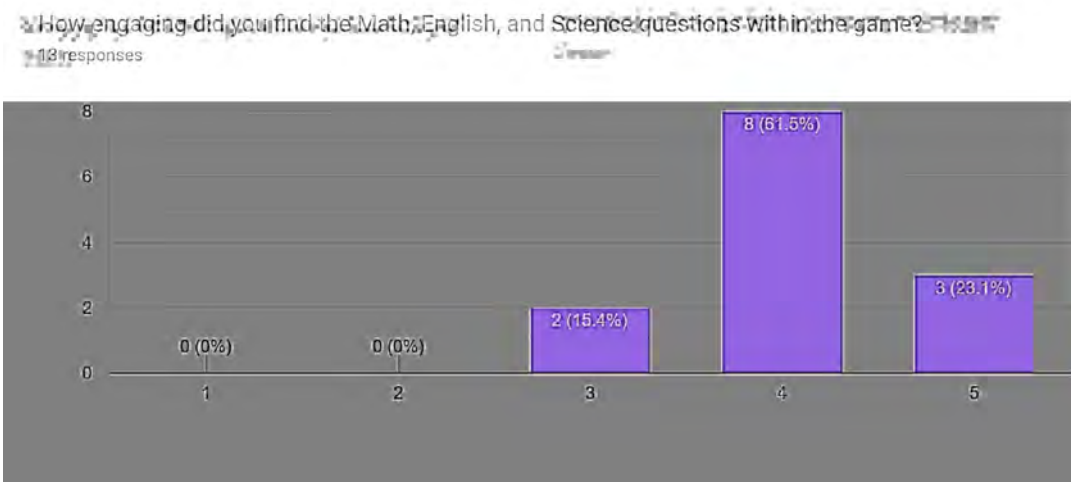


Figure 5.5: Quiz Mechanics Engagement Level

Finally, when rating how the gameplay mechanics implemented in this demo can support emotional resilience and provide a more rewarding learning experience, a staggering 46.2% rated it as most effective.

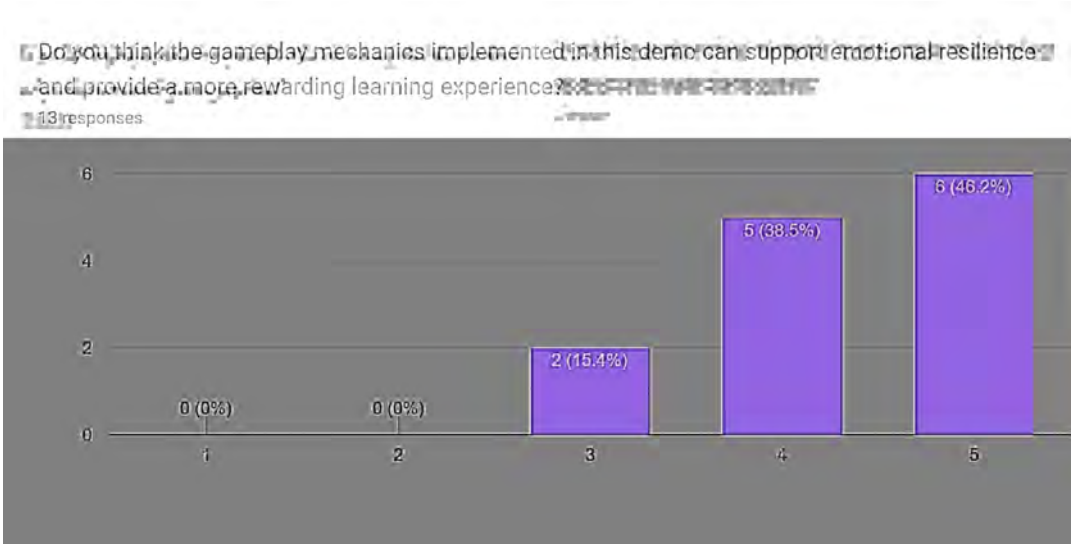


Figure 5.6: Gameplay Effectiveness in Developing Emotional Resilience and Rewarding Learning Experience

Some of the positive feedback of the demo included:

- ”This is a thoughtful project. Simple yet impactful. Best wishes!”
- ”It was a fun time! Great potential to expand further, hope to see it developed further.”
- ”I actually have enjoyed the quiz part a lot. Try to include more difficult quizzes with the player leveling up. And also about the turn-based thing, yes you can try to include that.”
- ”I think it’s a really fun game for children and yeah I think it will be effective for children with learning disabilities. But I do recommend to add some additional guidelines to the game so the children don’t get confused or get stuck while playing. (Like which keys to use for what purpose or maybe a quick intro at the beginning of the game)”
- ”Yeah definitely, Kids can learn a lot from video games, I mean games that targeted towards kids and integrating educational based questions can help more”
- ”Yes, I think the quiz-based battle system is a pretty innovative idea with a lot of potential. The gameplay experience was enjoyable. I suspect that this type of learning environment, activates the emotion centres of the brain, which in turn enhances learning.”
- ”Yes, I think it’s a fun learning. These days kids are so much into content consumption from different medias. These type of game and quiz can make them enjoy the journey of learning and also keep away from junk content consumption.”
- ”I think that the quiz based battle system is very fun and intuitive and having to switch between types is a clever mechanic to get to answer wider range of questions. And I do believe people with learning disability will have a very fun time with a visual style of quizzing.”

- ”It is a revolutionary attempt in helping children with ADHD/dyslexia to learn and respond to prompts and surroundings. It is a very effective approach in fun based learning for children facing learning problems, far more effective than conventional style of learning.”
- ”Yes it does make learning very enjoyable. The quiz based battle system has a lot of potential and could even be branched to different difficulty levels if players want to test their knowledge on variety of topics.”
- ”A good initiative on teaching. I am a grown man and I was invested in finding the right triangle.”
- ”The quiz based system is undoubtedly a great system as it’s really helpful for children to learn. Also, this really makes the learning much more enjoyable and effective for children with learning disabilities.”

Testing Methods: Both quantitative (Likert-scale responses) and qualitative (open-ended comments) data collection methods were employed to evaluate user engagement, usability, and learning outcomes.

5.2 Analysis of Design Solutions

The game design emphasized the integration of quiz-based battles and interactive NPC dialogue to stimulate focus and reinforce academic content.
Evaluation Criteria:

- **Accuracy:** Educational quizzes were relevant and correctly aligned with Math, English, and Science topics.
- **Efficiency:** Respondents found the battle quiz system engaging and effective. Comments such as “Kids can learn a lot from video games” and “It’s a revolutionary attempt in helping children with ADHD” highlight the design’s success.
- **Feasibility & Cost:** The prototype demonstrated low-cost feasibility for educational use both in classrooms and at home.
- **Performance:** Ratings for educational effectiveness averaged between 4 and 5 on a 5-point scale.
- **Strengths:** Engaging gameplay, smooth mechanics, and strong potential to improve attention and retention.
- **Weaknesses:** Feedback suggested including audio cues, more extensive NPC interactions, and adaptive progression systems for enhanced motivation.

5.3 Final Design Adjustments

Based on the survey, the testers recommended these for future updates:

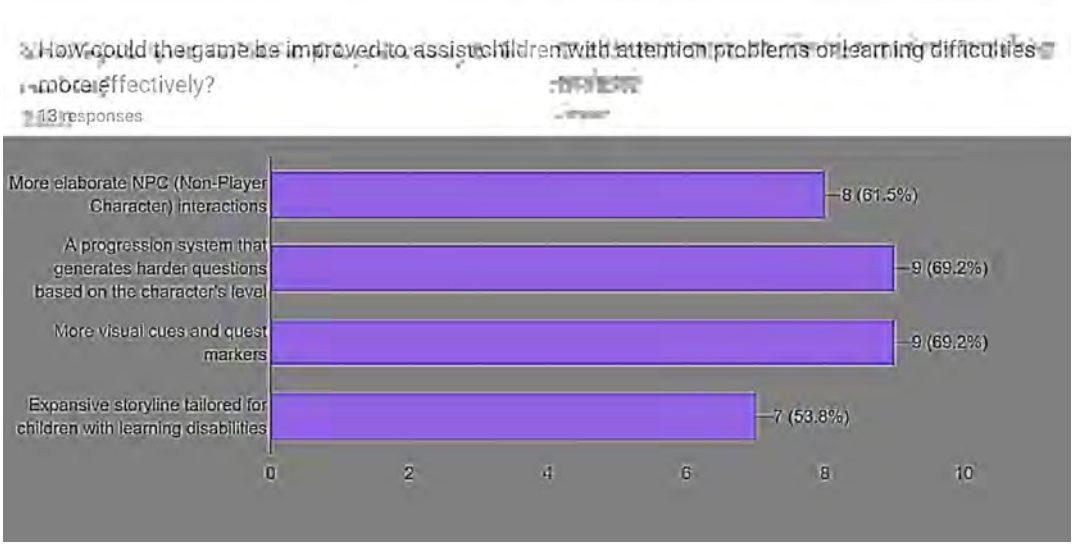


Figure 5.7: Future Development Features

Based on performance evaluation and user feedback, the following improvements are proposed:

- Add background music and sound effects to sustain attention.
- Introduce adaptive difficulty levels to maintain engagement.
- Expand NPC interactions for greater narrative and emotional depth.
- Optimize performance for smoother play on lower-end devices.
- Implement a reward-based progression system to reinforce learning achievements.

5.4 Statistical Analysis

Due to the limited sample size, descriptive statistics were used to summarize findings. These included mean, mode, and frequency analysis for Likert-scale data, and percentage-based distributions to identify response trends. Cross-tabulations between engagement and learning disability status indicated consistent enjoyment and perceived learning effectiveness across all participant types.
Future large-scale evaluations could incorporate inferential statistical tests such as *ANOVA* or *Chi-square* analyses to determine significant differences between groups (e.g., diagnosed versus non-diagnosed players).

5.5 Comparisons and Relationships

Compared to traditional learning tools, the prototype game exhibited greater engagement and focus enhancement. Participants emphasized that integrating quiz elements into RPG-style battles sustained attention better than static worksheets or conventional quizzes. Relative to existing educational games, the hybrid approach of combining academic content with turn-based combat demonstrated higher potential for maintaining motivation and supporting adaptive learning. The results align with prior research emphasizing that

gamified learning systems improve focus and cognitive engagement among neurodiverse learners.

5.6 Discussions

Findings suggest that educational games featuring interactive mechanics can effectively enhance focus, motivation, and conceptual understanding among children with learning disabilities. Participants reported positive experiences regarding the integration of academic content and gameplay, confirming the game's potential as a supportive learning tool.

Implications:

- Educational games can create a motivating, low-pressure environment for neurodiverse learners.
- Quiz-based battle mechanics successfully merge entertainment and learning, improving retention without overstimulation.

Limitations:

- The small sample size limits the generalizability of findings.
- Broader testing with educators and diverse learner groups is needed for validation.
- Long-term learning retention and emotional outcomes were not measured in this phase.

Overall, the results validate the conceptual design and highlight key directions for further refinement and large-scale implementation of the educational game.

Chapter 6

Conclusion

6.1 Summary of Findings

Figure 6.1: Recommendation Rate

The findings from the user survey indicate that the educational game prototype was well-received and effective in promoting engagement, focus, and conceptual understanding among players. Across all responses, participants demonstrated high levels of satisfaction with both the gameplay mechanics and the educational content.
Most players found the controls simple and intuitive, reporting scores between 1 and 2 on the difficulty scale. Engagement with academic content was rated highly, averaging between 4 and 5, particularly for the Math, English, and Science quiz components. All respondents agreed that the gameplay mechanics were helpful in supporting learning, and 100% of participants stated they would recommend the game for educational use, particularly for children with learning disabilities.
Open-ended feedback emphasized the positive impact of combining interactive quizzes with role-playing game (RPG) mechanics, suggesting that the integration of turn-based battles and academic content enhanced motivation and sustained focus. Only one participant reported minor optimization issues, indicating strong technical stability.
Collectively, the results validate the concept that well-designed educational games can bridge the gap between entertainment and learning, especially for neurodiverse students such as those with ADHD or dyslexia.

6.2 Contributions to the Field

This study contributes to the studies of gamified or gaming-based learning and its application for children with learning and attention difficulties. The prototype demonstrates how integrating educational quizzes into an engaging RPG framework can serve as an effective tool to reinforce academic concepts while maintaining student interest.
The primary contributions include:

- Demonstrating the feasibility of merging turn-based combat systems with subject-based quizzes to improve focus and retention.
- Providing user-based evidence that educational games can enhance motivation and emotional resilience in learners with attention challenges.
- Highlighting design strategies, such as audio-visual reinforcement and adaptive progression, that can be applied to other inclusive educational game designs.

This research thus supports the pedagogical potential of serious games as interactive learning aids, particularly for children who struggle with traditional classroom methods.

6.3 Recommendations for Future Work

In the previous chapter, several suggestions for future development were gathered from the testers. However, some more areas can be explored for improvement and future research:

- **Expanded Testing:** Conduct larger-scale evaluations across diverse demographics, including children formally diagnosed with ADHD, dyslexia, or other learning differences, to validate the observed effects statistically.
- **Longitudinal Studies:** Implement long-term assessments to measure learning retention, behavioral changes, and emotional resilience over extended play periods.
- **Enhanced Design Features:** Incorporate adaptive learning algorithms that dynamically adjust question difficulty, and expand storyline depth to enhance motivation.
- **Educator Integration:** Collaborate with teachers and therapists to align game content with educational curricula and therapeutic goals. Perhaps add a gameplay option that can allow educators to add their own questions and use it as a tool for taking exams of their students.
- **Technical Optimization:** Improve performance for low-spec devices and explore mobile or web-based deployment to increase accessibility.

In conclusion, the study highlights the substantial promise of educational gaming as a supportive intervention for children with learning disabilities. By integrating academic content within engaging, interactive experiences, such games can cultivate focus, resilience, and confidence, contributing meaningfully to inclusive education and digital learning innovation.

Bibliography

[1] Gemma K. Strong, Carole J. Torgerson, David Torgerson, and Charles Hulme. "A systematic meta-analytic review of evidence for the effectiveness of behavioural interventions for children diagnosed with Attention Deficit Hyperactivity Disorder (ADHD)". In: *Psychological Bulletin* 137 (2011). Accessed: 2024-10-19, pp. 1–23. URL: https://doi.org/10.1037/a0022092.

[2] Patricia García-Redondo, Trinidad García, Débora Areces, José Carlos Núñez, and Celestino Rodríguez. "How video games promote cognitive processes in learning disabilities". In: *Frontiers in Psychology* (July 2019). Accessed: 2024-10-17. URL: https://doi.org/10.3389/fpsyg.2019.01524.

[3] Steve Noble. *Supporting your child with disabilities with digital learning games*. Accessed: 2024-10-17. Nov. 2020. URL: https://www.pbs.org/parents/thrive/supporting-your-child-with-disabilities-with-digital-learning-games.

[4] N. Thomas and A. Woodyatt. *FDA approves video game as treatment for ADHD*. Accessed: 2024-10-17. June 2020. URL: https://edition.cnn.com/2020/06/16/health/adhd-fda-game-intl-scli-wellness.

[5] Saniya Khan. *Can video games help kids with learning disabilities?* Accessed: 2024-10-17. Sept. 2021. URL: https://www.edtechreview.in/trends-insights/trends/can-video-games-help-kids-with-learning-disabilities.

[6] Education360Journal. *6 Proven benefits of game-based learning for children with learning disabilities*. Accessed: 2024-10-17. May 2023. URL: https://education360journal.medium.com/6-proven-benefits-of-game-based-learning-for-children-with-learning-disabilities-59fe67dd2cbc.