Name: Trung Kien Nguyen

Student ID: 104053642

SWINBURNE UNIVERSITY OF TECHNOLOGY

COS20007

Object Oriented Programming

REPORT

Research Project – Object Oriented Principles in Unity Game Development

TABLE OF CONTENTS

Object Oriented Programming in Unity Game Development

TRUNG KIEN NGUYEN – 104053642

Bachelor of Computer Science – AI major

Swinburne University of Technology – Hawthorn Campus – Semester 1 – Year 2023

ABSTRACT

Unity Game Engine is a powerful and widely used cross-platform game development platform that has revolutionized the world of interactive entertainment. Designed to create games for a variety of platforms, including PCs, consoles, mobile devices, and virtual reality, Unity offers developers a comprehensive set of Object-oriented tools, features, code styles to bring their creative visions to life.

INTRODUCTION

The goal of this research project is to examine some Unity's object-oriented features, and OOP concepts are applied in Unity game development. Through a combination of documentation, and real-world experimentation, this sheds light on the importance and usefulness of OOP principles in game design and code organization by examining how Unity makes use of them, especially promoting code readability, reusability, maintainability, and scalability, as well as its influence on the product's performance and optimization.

This report covers a total of four main sections, including the most basic things about Unity Engine, which is designed through the object-oriented model, as well as referring to some of its typical game projects, and conducting my own as an experiment. Following that, I discuss and give my opinion about some advanced features and functionality, and some potential limitations of OOP applied in the Unity game development, in terms of its codebase optimization, and overall game performance.

UNITY AND ITS OBJECT - ORIENTED STRUCTURE

For the research project, initially, I conducted an extensive review of Unity's official documentation [1], and relevant resources or literature on its OOP principles [3], to understand this engine's basic structure, as well as how it use objects and OOP concepts.

1. GameObject and Component

Every object created in Unity is a *GameObject* - the native type of object representing things in a game, including the concrete ones like the player(s), enemies, scenery, and items, or the more abstract ones such as UI, and audio manager. However, these GameObjects are just "containers" for *components*, and in most cases, they have no specific functionalities.

A GameObject containing a component has that component's functionality. For example, "`Transform`" defines the object's position, rotation, scales, while "`Rigidbody`" represent to physical body of the objects, allowing their interacting with various types of force within the game. As a lot of them are specialised in specific functions, they can be component-system hybrids as opposed to just simple components.

The Unity concept of component can present the principles of OOP. Taking an example of Inheritence in Unity's components, in general, "`Collider`" component represents the ability of an object (GameObject) to collide with the others. There are many of its "subcomponents" to present collider in different circumstances, including the object's shape of collision preferences ("`BoxCollider`", "`SphereCollider`", …).

2. Scripts

Unity helps developers create scripts, for the custom behavior(s) of GameObjects. In other words, each script in Unity, which normally represents a class or equivalent, can be considered a special component.

By default, the classes created inherits from `MonoBehaviour`, an Unity's base class. Through this inheritence, our class can inherits some useful methods, including "`Awake()`", "`Start()`", "`Update()`", "`FixedUpdate()`", …

Also, by using these scripts to present the object's behaviours and functionalities. We can apply the OO principles to enhance our codebase's readability, reusability, and updatability.

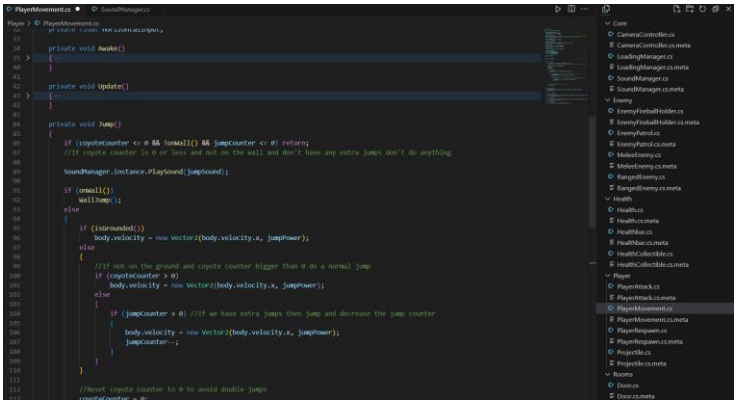
EXAMPLES AND EXPERIMENTS IN UNITY WITH OOP

In this stage, I have searched for some Unity projects' sources that demonstrate some of the principles of OOP.

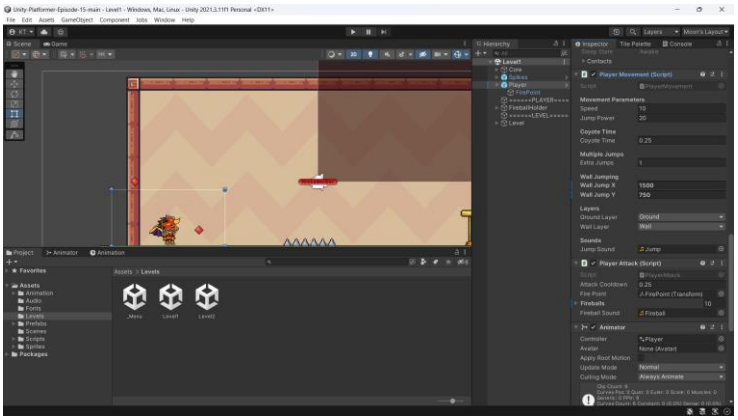
1. Unity 2D Platformer for Complete Beginners – Pandemonium

"Simple 2D platformer in Unity" [4]

This is one of my first tutorials since I got started with Unity. This 2D Platformer project is made with a simple Object-oriented codebase.



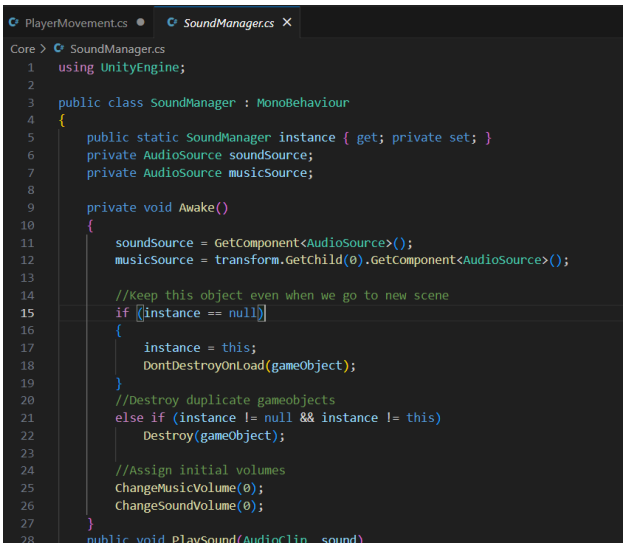
Considering this "`PlayerMovement`" script. This class is to create a component of movement for the GameObject "Player".



In the code of this class, some OOP's principles are basically presented:

- Abstraction: The class define how the player should move in the game, thus, it has the fields relating to that process, such as speed, jump power, wall jump forces, … .
- Inheritence: This class is also inherited from `MonoBehaviour`, allowing it to implement "`Awake()`" and "`Update()`" methods.

Take a script of "`SoundManager`", which presents very simply Singleton:



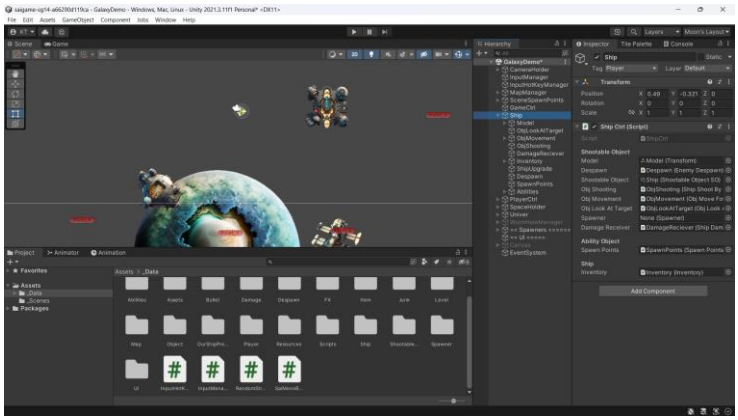
```cs
using UnityEngine;

public class SoundManager : MonoBehaviour
{
    public static SoundManager instance { get; private set; }
    private AudioSource soundSource;
    private AudioSource musicSource;

    private void Awake()
    {
        soundSource = GetComponent<AudioSource>();
        musicSource = transform.GetChild(0).GetComponent<AudioSource>();

        //Keep this object even when we go to new scene
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(gameObject);
        }
        //Destroy duplicate gameobjects
        else if (instance != null && instance != this)
            Destroy(gameObject);

        //Assign initial volumes
        ChangeMusicVolume(0);
        ChangeSoundVolume(0);
    }
    public void PlaySound(AudioClip sound)
```

Singleton is used to make the class "easily accessable to the others" – according to the author, as sound effects and background music will have the same volume respectively when they are played.

2. Unity 2D Spaceship shooter – Sai Game

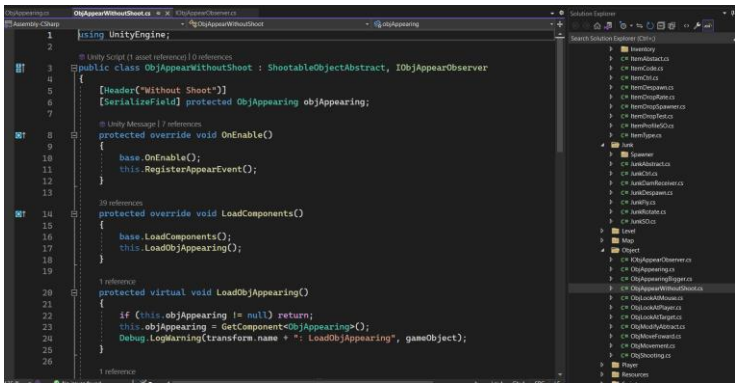
"*Tutorial Siêu to khổng lồ với Unity2D*" [5]

This is a tutorial project made by one of my favorite Vietnamese Game Developers, which is a 2D top-down shooting with spaceships.



In his code, besides concepts of OOP and patterns of Game programming, there is some interesting game programming patterns that have been applied.

Taking his "`ObjAppearing`" and the related scripts as the example for Observer pattern:

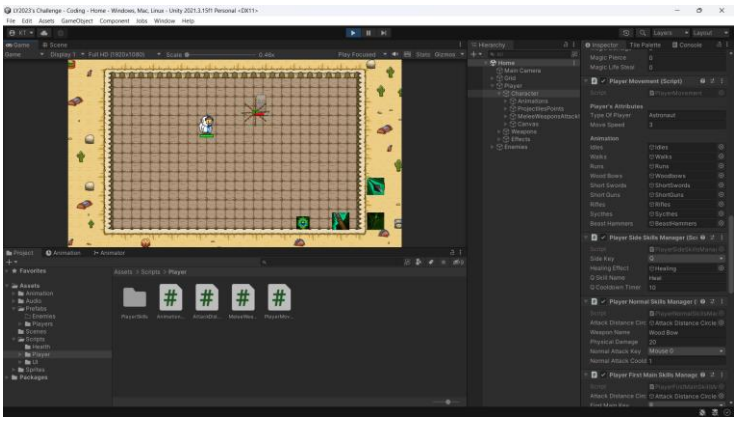


```cs
using UnityEngine;

public class ObjAppearWithoutShoot : ShootableObjectAbstract, IObjAppearObserver
{
    [Header("Without Shoot")]
    [SerializeField] protected ObjAppearing objAppearing;

    protected override void OnEnable()
    {
        base.OnEnable();
        this.RegisterAppearEvent();
    }

    protected override void LoadComponents()
    {
        base.LoadComponents();
        this.LoadObjAppearing();
    }

    protected virtual void LoadObjAppearing()
    {
        if (this.objAppearing != null) return;
        this.objAppearing = GetComponent<ObjAppearing>();
        Debug.LogWarning(transform.name + ": LoadObjAppearing", gameObject);
    }
```

In a nutshell, when handling appearing of enemy ships, he wants to make sure that the new spawn enemies will not be able to shoot our ship until they are completely spawned (their scale would gradually grow to the specific size). Therefore, the author considers "`ObjAppearing`", which is responsible for the appearance of the enemy, as a *(observable) subject*, and "`ObjAppearWithoutShoot`" implemented the interface "`IObjAppearObserver`" is an observer, which notifies the two events of the appearing's start and finish, and do the

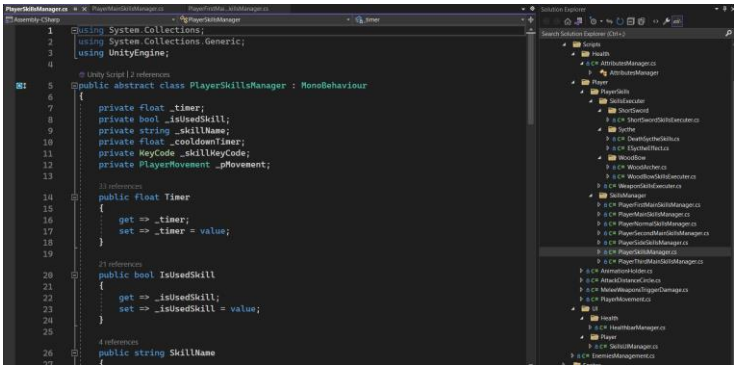
activate/deactivate the shootable functionality ("`ObjShooting`" and "`ObjLookAtTarget`" components).

3. My Unity prototype project - Lunar Year 2023's Challenge [6]



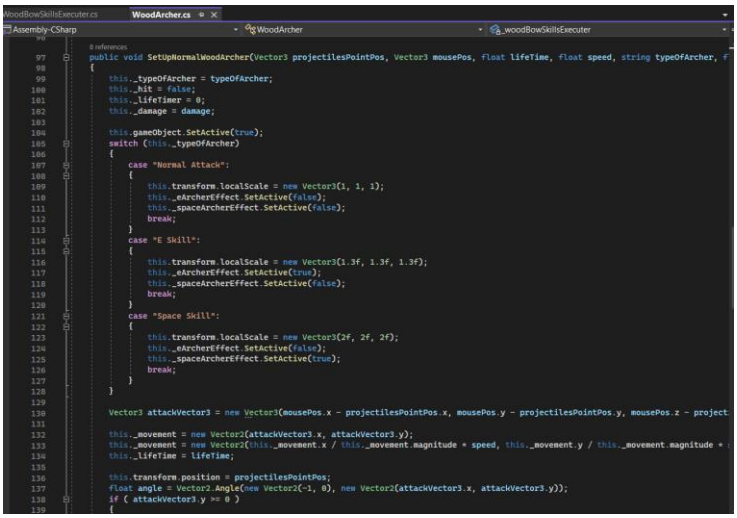
I have done a very simple prototype for this project, which is a 2D top-down Unity game appyling some OOP's principles in the codebase.

In this experimental project, I have demonstrated some key principles of OOP, especially the Inheritence and Polymorphism concept, since I have to implement for functional Skills for my Player, with some similar data and behaviour likes cooldown timer, skill name, skill functional key, … .



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

public abstract class PlayerSkillsManager : MonoBehaviour
{
    private float _timer;
    private bool _isUsedSkill;
    private string _skillName;
    private float _cooldownTimer;
    private KeyCode _skillKeyCode;
    private PlayerMovement _pMovement;

    public float Timer
    {
        get => _timer;
        set => _timer = value;
    }

    public bool IsUsedSkill
    {
        get => _isUsedSkill;
        set => _isUsedSkill = value;
    }

    public string SkillName
```

Also, I have implemented one of the famous game programming patterns, which is the *Object Pool*:
When player uses the Wood bow weapons, it can create some archers to shoot the enemies. If I spawn new archer GameObjects each time the player want to shoot, it may lead to underperformance of the game due to *memory fragmentation*.
Hence, I have initialized 20 deactivated archer GameObject, and when player need to summon an archer, it will take one from that list, and implement its own behaviour (speed, direction, …). In addition, I have made a mechanism of timer between attacks to make sure the total number of summoned archered at every time of the game is less that 20



```
public void SetUpNormalWoodArcher(Vector3 projectilesPointPos, Vector3 mousePos, float lifeTime, float speed, string typeOfArcher, f
{
    this._typeOfArcher = typeOfArcher;
    this._hit = false;
    this._lifeTimer = 0;
    this._damage = damage;

    this.gameObject.SetActive(true);
    switch (this._typeOfArcher)
    {
        case "Normal Attack":
        {
            this.transform.localScale = new Vector3(1, 1, 1);
            this._eArcherEffect.SetActive(false);
            this._spaceArcherEffect.SetActive(false);
            break;
        }
        case "E Skill":
        {
            this.transform.localScale = new Vector3(1.3f, 1.3f, 1.3f);
            this._eArcherEffect.SetActive(true);
            this._spaceArcherEffect.SetActive(false);
            break;
        }
        case "Space Skill":
        {
            this.transform.localScale = new Vector3(2f, 2f, 2f);
            this._eArcherEffect.SetActive(false);
            this._spaceArcherEffect.SetActive(true);
            break;
        }
    }

    Vector3 attackVector3 = new Vector3(mousePos.x - projectilesPointPos.x, mousePos.y - projectilesPointPos.y, mousePos.z - project

    this._movement = new Vector2(attackVector3.x, attackVector3.y);
    this._movement = new Vector2(this._movement.x / this._movement.magnitude * speed, this._movement.y / this._movement.magnitude *
    this._lifeTime = lifeTime;

    this.transform.position = projectilesPointPos;
    float angle = Vector2.Angle(new Vector2(-1, 0), new Vector2(attackVector3.x, attackVector3.y));
    if ( attackVector3.y >= 0 )
    {
```

DISCUSSION

1. OOP in console app and Unity game project

Besides the similarities in implementing and applying the fundamental principles, there are many differences in OOP structure in a typical console app compared to an Unity project's coding.

1.1. Terminologies and concepts confusion

Although both can be built and developed through OO principles, there is a clear difference in the core terms. One of the most popular confusions is the "**objects**" concept. When developing typical Object-oriented programs or projects, objects are the fundamental units

of the Object-oriented programs, which encapsulate data and behavior into a single entity. On the other hand, as I have mentioned, "object" in Unity or other Game Engines is frequently referred to as "GameObject". Although it is used to describe tangible or intangible objects we can interact with in the game world, to the developers, it is only an abstract container. Meanwhile, the things it contains, components, can be considered roughly the equivalent of "objects" in non-Unity programs.

Another possible misbelief is the **Inheritence**. In OOP, Inheritence refers to the ability of a class (child class) to inherit some attributes and behaviour of another (parent class). While this principle can be expressed in both cases to a similar extent, some people often confuse it with another of Unity's mechanics, along with confusing "objects" in OOP and Unity's GameObjects. In particular, Unity allows a GameObject to have one or more child GameObjects, which makes some of their properties such as position, scale, ..., dependent on the parent one, and this is totally different from the fundamental concepts of Inheritance in OOP.

1.2. Initialize instances of class

Normally, in a console app made with C#, we usually define a special method – *Constructor*, which is responsible for initializing objects of a class with the “new” keyword.

In Unity, when a GameObject contains a script that defines a class, it can be considered as having a component of that class, or in other words, an instance of this class has been automatically created and assigned to the GameObject like a component. The classes in Unity usually have no Constructor, instead, it uses the method “`Start()`” or “`Awake()`” inherited from `MonoBehaviour` class, to set up any necessary data (variables or fields).

2. OOP and Unity developing teams

By optimizing the Unity project's codebase, OOP ensures and enhances the efficiency and productivity of the development team.

2.1. Code Maintainability and Extensibility

Unity projects tend to get more complex over time, making maintenance difficult gradually. OOP's modular structure makes it easier to check and fix bugs or make enhancements, as developers can isolate specific classes or objects for testing or debugging purposes, without affecting the rest of the project, reducing the risk of unintended consequences.

In addition, these game projects often undergo changes and updates not only throughout the development process, but also after publishment. With OOP, developers may easily add new features or change existing ones without having to rewrite a sizable amount of code. New classes can be developed independently or as derivations of existing ones while still integrating seamlessly into the existing codebase by making use of principles like Inheritance and Polymorphism.

2.2. Team Collaboration

The development of games in Unity frequently involves a team of developers working on various game elements at once. Obeject-oriented principles make it easier to collaborate since it enables programmers to work independently and effectively on distinct classes or components without interfering with one another's code. Classes can conceal their core implementation through encapsulation, exposing only the essential interfaces for use by others.

2.3. Code Understandability

OOP promotes a modular approach to programming, allowing developers to organize their code into classes and objects. This makes the code of a member more readable and understandable to others, even to those who are not familiar with the project. It also facilitates effective communication within the team, making it easier to discuss and collaborate on code-related matters.

3. Programming Patterns in Unity Development

In general, Game programming patterns can be classified into the some following main groups: [7], [8]

3.1. Design patterns

Generally, in software development, *design patterns* are the generic, reusable coding and developing models to solve the commonly occurring issues. In terms of Unity or game development, design patterns are also widely applied in certain situations. Considering some of the most famous and popular design patterns in Unity that have presented in the experiments and examples part:

- *Singleton*: This pattern ensures a class has only one instance, and provides the global access to it.
    - This design pattern is so popular that many beginners overuse it, however, according to the "Game Programming Patterns", developers should avoid this pattern as much as possible because such global objects can cause performance issues.
- *Observer*: This pattern is entirely focused on what will occur after an event has happened.
    - In fact, to reduce developers' manual coding, Unity also has its own mechanisms, including UnityEvent and the related.
    - This is really useful if you want to avoid intricacy code by making classes independent of one another (loosing coupling). The key point is that the components causing the event (observable subjects) do not need to care about what methods or functionalities are attached to it. Thus, the developer can more easily identify the source of the bug if an event is triggered but nothing happens.

3.2. Sequencing patterns

Game Looping and *Updating* are the two significant parts of videogame development, including every game framework, library, or engine. In Unity, they are already built-in through its engine and mechanism.

- GameLoop: All video games revolve around a loop, which is an endless while loop, and keeps the game processing, updating and rendering. This mechanism is applied by default by Unity Engine
- Update Method(s): This concept is closely related to GameLoop, it will process one frame of behaviour and functionality.
    - In Unity, there are two main types of this method, including "`Update()`" executed each frame, and "`FixedUpdate()`" executed at a specific rate defined in the editor. It also provide another one, "`LateUpdate()`", which is called after all other Update methods have been executed. They all are inherited from the class `MonoBehaviour`.
    - To optimize performance, usually each GameObject's component will have its own Update method(s).

3.3. Optimization patterns

Performance optimization is an essential part that affects all aspects of software. In terms of Game Development, Unity, like other Game Engines, allows developers to make improvements in game performance and speed through a variety of ways. Here are two common patterns among them:

- Data Locality: The fundamental principle is that the data structures should be set up so that the processing things are nearby one another in the memory.
    - Unity has its own technology to implement this idea, which is *Data-Oriented Technology Stack* (DOTS).
    - According to Game-Ace Studio, DOTS is a data-oriented paradigm, representing a system of Unity development with significant modifications to match the previous ones. [9]
- Object poolling: This has been mentioned in my prototype. Basically, because the game's performance will decrease significantly if keeping creating and destroying things, it is more effective to set up the GameObjects once, and activate/deactivate them as the game design.

4. OOP and Entity Component System

Entity Component System (ECS) is a data-oriented architectural framework and model associated with GameObjects in Game Development likes Unity. It allows Unity developers to build large-scale games with an "unprecedented level of control and determinism" – according to Unity. [2]

The mentioned system of game object and component in Unity is based on the Entity Component System, which has the concepts of *Entity* and *Component* respectively. Furthermore, a *system* in ECS utilises one or multiple components' inner data to present the entity's behaviour and functionality.

Although ECS concepts can present some Object-oriented features, there are some crucial differences between OOP and ECS. Firstly, in a pure object-oriented paradigm, all behavior and data of an Entity are part of the same object, whereas they are separated into Components and Systems in ECS. As a result, an entity's function and behaviour can be easily extended and updated by adding new components or modifying or removing the existing ones, enhancing the flexibility of the model. Secondly, ECS allows developers to store all entity data for each component in the same memory area. We can do coding quickly that repeatedly applies the same modification to all instances of a single component, which is an ideal solution for a CPU. Following that, with the ECS concept, it is also simple to run concurrently these transformation processes, as it enables developers to benefit from the multicore systems, and, reduce the amount of code supporting concurrency, in contrast to the complicated object-oriented concurrent programming.

CONCLUSION

In summary, object-oriented programming is applied as one of the fundamental paradigms for Unity Game development. Its concepts, including the four key principles and the related, enable the development of complex behaviors and enhance code reusability, leading to organized and maintainable projects. By analysing some resources, including papers, books, articles and documentation, some Case Studies of available Unity projects, as well as developing my own experimental one, I reinforce the OOP knowledge that I have learned during the semester and link it to one of my favorite game engines, Unity, as well as draw conclusions on the advanced features and issues related to OOP and Unity, such as programming patterns and ECS.

In my opinion, this research project has brought forth a lot of important ideas and views on taking advantage of the strengths of OOP, along with replacing or improving some limitations with other paradigms, contributing to help developers harness the full potential of Unity and create immersive gaming experiences.

Not only in OOP and Unity, the game industry is curently growing so fast that countless technologies, frameworks and game engines are born and updated every day.

As my passion of game development, in the future I will do much more complex and enormous researches and projects on Game Engines, along with related up-to-date features and technologies.

REFERENCES

[1] "Unity Scripting Reference," Unity Documentation. https://docs.unity3d.com/ScriptReference/index.html

[2] "Entity Component System for Unity," ECS for Unity. https://unity.com/ecs

[3] T. Holopainen, "Object-oriented programming with Unity," May 2016. https://www.theseus.fi/bitstream/handle/10024/108967/Holopainen_Timo.pdf?sequence=1&isAllowed=y

[4] Nickbota, "Unity 2D Platformer for Complete Beginners," Pandemonium, Oct. 19, 2022. https://github.com/nickbota/Unity-Platformer-Episode-15

[5] Sai Game, "Tutorial Unity2D siêu to khổng lồ," Học làm game Unity2D - Learn to make Unity2D game, Oct. 2022. https://bitbucket.org/saigame/sg14/src/master/

[6] Me (Trung Kien Nguyen), " Lunar Year 2023's Challenge," My experimental Unity prototype project for COS20007 Research Project, Jun. 04, 2023. https://github.com/moon22082004/005_Unity_Lunar_Year_2023_Challenge

[7] R. Nystrom, "Game Programming Patterns," 2011. http://gameprogrammingpatterns.com/

[8] Habrador, "Game programming patterns in Unity," Jun. 2020. https://github.com/Habrador/Unity-Programming-Patterns

[9] Game-Ace Creative Studio, "New Unity's DOTS System and What It Means for Game Development," Unity DOTS, Oct. 28, 2019. https://game-ace.com/blog/unity-dots-system/