АКАДЕМИЈА ТЕХНИЧКО-УМЕТНИЧКИХ СТРУКОВНИХ СТУДИЈА
БЕОГРАД

ОДСЕК ВИСОКА ШКОЛА ЕЛЕКТРОТЕХНИКЕ И РАЧУНАРСТВА

Алекса Петровић

Имплементација једне 2Д игре у Unity окружењу

- завршни рад -

Београд, септембар 2024.

Кандидат: **Петровић Алекса**

Број индекса: **72/17**

Студијски програм: **НРТ**

Тема: **Имплементација једне 2Д игре у *Unity* окружењу**

Основни задаци:

1. Пројекат игре.

2. Развој модела и нивоа

3. Програмско решење

Ментор:

Београд, септембар 2024. године.

_____

др Зоран Ћировић, проф. ВИШЕР

РЕЗИМЕ:

У дипломском раду су приказани детаљи имплементације видео игре *Interstellar Lexicon*, која је израђена у *Unity* окружењу коришћењем програмског језика *C#*. Описана је структура фолдера и фајлова, примењене кодне конвенције, кључни системи који управљају логичким делом игре, као и корисничком интерфејсу.

Кључне речи: : видео игра, *Unity*, *C#*, кориснички интерфејс

ABSTRACT:

In this thesis, the implementation details of the video game *Interstellar Lexicon*, developed in the Unity environment using the C# programming language, are presented. The folder and file structure, applied coding conventions, key systems that manage the game's logic, as well as the user interface, are described.

Key words: video game, Unity, C#, user interface

САДРЖАЈ:

1. УВОД 1
2. ИМПЛЕМЕНТАЦИЈА 2
   2.1. Општи приступ имплементације 2
      2.1.1. Структура фолдера и фајлова унутар пројекта 2
      2.1.2. Конвенције унутар кода 3
   2.2. Системи у игри 3
      2.2.1. Систем за унос речи 3
      2.2.2. Систем непријатељских бродова 5
      2.2.3. Систем појачивача 9
      2.2.4. Систем пројектила 16
      2.2.5. Систем камера 18
      2.2.6. Менаџер игре 18
      2.2.7. Модификатори 20
      2.2.8. Систем резултата 22
      2.2.9. Графички кориснички интерфејс 24
3. ЗАКЉУЧАК 36
4. ИНДЕКС ПОЈМОВА 37
5. ЛИТЕРАТУРА 38
6. Прилози 39
   6.1. Усликан приказ игре 39
7. Изјава о академској честитости 40

1. УВОД

C# је објектно оријентисан, компајлиран програмски језик високог нивоа, широко коришћен у развоју Unity игара и апликација. Дизајниран је да буде једноставан, модеран и моћан, са снажном подршком за компонентно програмирање и управљање меморијом. У оквиру Unity окружења, омогућава креирање скрипти које управљају свим аспектима игре, од механике до графике.

Unity је свеобухватна платформа за развој игара и 3D апликација, широко призната због своје флексибилности и способности да подржава развој за више платформи, укључујући мобилне уређаје, конзоле и веб. Користи *C#* као главни програмски језик за скриптовање. *Unity* платформа омогућава креаторима да лако управљају свим аспектима игре, од физике и анимације до управљања ресурсима и корисничког интерфејса. Платформа је опремљена богатим екосистемом алата, као што су уређивач сцена, систем за анимацију, и подршка за *AR/VR*, чиме омогућава развој игара и апликација високог квалитета. Поред тога, *Unity Asset Store* нуди приступ великом броју ресурса и додатака који додатно убрзавају развојни процес.

Interstellar Lexicon је 2Д видео игра развијена коришћењем *Unity 2022.3.13f1* платформе и *C# 8.0*. Може се покренути на десктоп рачунару са *Windows* оперативним системом. Резултати се чувају локално на рачунару.

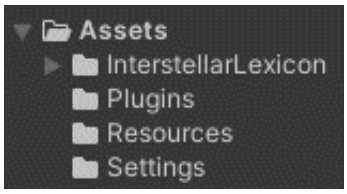
Игра је за једног играча, креирана у аркадном стилу – једноставан интерфејс и прогресивно повећање тежине. У игри играч има поглед из свемира на своју планету и ванземаљске непријатељске бродове који долазе ка њој са десне ивице екрана. Циљ му је да сакупи што више поена уништавањем тих бродова. Сваком броду је додељена насумична реч на енглеском која је видљива изнад брода, а играч мора да укуца реч и притисне дугме Enter како би испалио ракету са своје планете ка том броду да је уништи. Дужина речи одређује брзину, изглед, величину и јачину брода – бродови који имају дужу реч су већи и јачи али и спорији, док краћи бродови буду бржи али слабији. Бродови имају за циљ да се сударе са планетом, што доводи до експлозије и играчу губљење животних поена. Поред тога, појачивачи који такође садрже речи путују са доњег дела екрана до горњег; искуцавањем те речи, играч добија бонусе у игри. Сви ови објекти се непрекидно појављују све док играч не изгуби све животне поене. Генерисање је насумично, али у одређеном подручју које је успостављено кроз тестирање игре и проналажење баланса. Такође на генерисање утиче још фактора попут тежине коју је играч наместио и ниво играча у тренутној игри.

2. ИМПЛЕМЕНТАЦИЈА

2.1. ОПШТИ ПРИСТУП ИМПЛЕМЕНТАЦИЈЕ

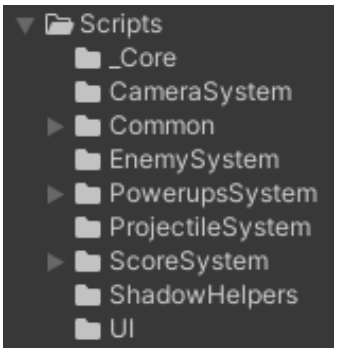
2.1.1. Структура фолдера и фајлова унутар пројекта

Пројекат је осмишљен са веома организованом структуром фолдера и фајлова, ради очувања чистоће и прегледности. Унутар фолдера Assets је креиран посебан фолдер са именом игре. У том фолдеру се налазе сви фајлови који не припадају додацима треће стране. На тај начин, одржава се јасна граница између увезених ресурса и сопствених ресурса, што олакшава управљање и проналажење фајлова.



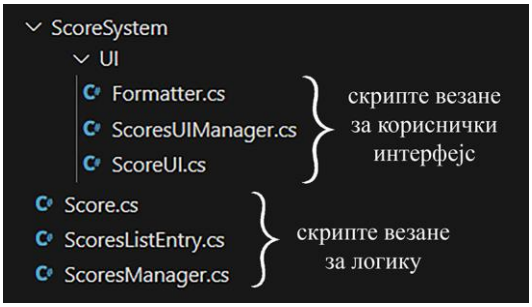
Слика 2.1 – Структура фолдера у корену пројекта

Поред тога, све скрипте су распоређене у различите фолдере унутар главног фолдера Scripts, према системима којима припадају. Ова структура омогућава лакше одржавање и управљање кодом, јер је свака скрипта груписана у складу са функцијом коју обавља.



Слика 2.2- Структура фолдера за скрипте

Како бисмо избегли мешање логике и корисничког интерфејса, скрипте које се односе на кориснички интерфејс сваког система смештене су у засебне фолдере унутар фолдера тог система. Пример овакве организације приказан је на слици испод.



Слика 2.3- Пример разсдвајања логичких скрипти од скрипти за кориснички интерфејс

2.1.2. Конвенције унутар кода

У коду, све написане скрипте налазе се унутар AP namespace-а. Овај приступ омогућава избегавање сукоба у именовању са скриптама из извезених ресурса. Већина скрипти су додатно раздвојена унутар посебних namespace-ова, који су најчешће еквивалентни фолдеру у којем се налазе. На пример, класа CameraShaker се налази у namespace-у AP.CameraSystem али и у фолдеру CameraSystem. Овај начин организовања кода не само да помаже у избегавању конфликата већ и олакшава навигацију и управљање скриптама у пројекту.

Поштован је стил писања тако да:

- За namespace-ове, класе, методе, догађаје и јавна својства користимо *pascal case*[1] формат.
- За класе чије постоји само једна инстанца, али и која није статична јер наслеђује од MonoBehaviour, додељује се суфикс Manager.
- За приватне променљиве користимо префикс m_.
    - У случају да је приватна променљива и статичка, уместо m_ користимо s_.
- За константе користимо *constant case*[2] формат.
- Унутар класе, на врху се налазе јавне променљиве, а затим приватне. Након тога следе методе које припадају Unity-јевом животном циклусу, затим јавне методе, и на крају приватне методе.

2.2. СИСТЕМИ У ИГРИ

2.2.1. Систем за унос речи

Систем за унос речи је кључни део пројекта који управља интеракцијом играча кроз куцање текста, што је основн

и механизам у игри. Његова примарна функција је да региструје унос тастатуре, генерише и прати промене унетих речи у реалном времену, и на крају шаље те речи другим системима у игри. Ово омогућава играчу да куца речи које се појављују на екрану и тако директно утиче на ток игре.

Сваки пут када играч унесе или избрише слово, систем ажурира тренутну реч и активира догађај *OnWordChanged*. Овај догађај омогућава другим деловима кода да реагују на промену речи, на пример, ажурирањем графичког приказа речи на екрану Уколико играч притисне један од тастера за потврду, као што су Enter или KeypadEnter, тренутна реч се шаље кроз догађај *OnWordSubmitted*, након чега се реч ресетује и играч

[1] Pascal case – пракса писања вишеречних спојева заједно, са великим словима на почетку речи (нпр. MyPascalCaseVariable)

[2] Constant case – пракса писања вишеречних спојева заједно, са великим словима и са доњом цртом између речи (нпр. MY_CONSTANT_CASE_VARIABLE)

почиње да уноси нову реч. На тај догађај реагују системи непријатељских бродова или појачивача, који проверавају да ли се унета реч подудара са неком инстанцом брода или појачивача.

Осим тога, систем такође подржава уклањање последњег слова из речи ако играч притисне тастере Backspace или Delete. Ова функција омогућава играчу да исправи грешке у куцању пре него што пошаље реч, чиме се повећава прецизност игре. Овако изгледа имплементација у коду:

```
private void Update()
{
    // Submit word.
    if (m_submitKeyCodes.AnyKeyDown() && CurrentWord.Length > 0)
    {
        OnWordSubmitted?.Invoke(CurrentWord);
        CurrentWord = string.Empty;
        OnWordChanged?.Invoke(CurrentWord);
    }

    // Delete letter.
    if (m_deleteKeyCodes.AnyKeyDown() && CurrentWord.Length > 0)
    {
        CurrentWord = CurrentWord.Remove(CurrentWord.Length - 1, 1);
        OnWordChanged?.Invoke(CurrentWord);
    }

    // Add letter.
    if (CurrentWord.Length == WordsManager.LongestWord) return;

    foreach (KeyCode letter in m_alphabetKeyCodes)
    {
        if (Input.GetKeyDown(letter))
        {
            CurrentWord += letter.ToString().ToLower();
            OnWordChanged?.Invoke(CurrentWord);
        }
    }
}
```

Један важан аспект овог система је да прати време које играч проводи куцајући реч, користећи променљиву *TimeSpentTyping*. Ово мерење времена је од користи за праћење ефикасности играча. Овако изгледа имплементација у коду:

```
private void LateUpdate()
{
    if (CurrentWord.Length == 0) return;
    TimeSpentTyping += Time.unscaledDeltaTime;
}
```

Један од кључних делова иницијализације система је метода InitializeAlphabetKeys(), који генерише листу свих валидних тастера за унос азбучних карактера од 'a' до 'z'. Ова листа омогућава систему да препозна било који притисак тастера који представља слово и да га дода тренутној речи коју играч куца. Овако изгледа имплементација у коду:

```
private void InitializeAlphabetKeys()
{
    for (char key = 'a'; key <= 'z'; key++)
        m_alphabetKeyCodes.Add((KeyCode)key);
}
```

На тај начин, систем за унос речи обезбеђује флексибилност и реактивност уноса текста у игри, омогућавајући играчима да директно комуницирају са објектима у игри и изазовима који им се пружају. Има јасну и структурисану логику која подржава различите акције играча, као што су унос, брисање и потврда речи, чиме се ствара динамична и интуитивна механика уноса.

У склопу оптимизације кода и унапређења читљивости, развили смо додатни функционалитет за рад са уносом корисника кроз класу InputExtensions. Ова класа омогућава ефикаснију проверу стања више тастера истовремено, што је особито корисно у ситуацијама где је потребно реаговати на притисак било којег од дефинисаних тастера у листи, које су у случају овог пројекта било који тастери за потврђивање речи или за брисане слова. Метода AnyKeyDown() проширује функционалност система за унос речи, омогућавајући да се са једноставним позивом провери да ли је било који од тастера у листи притиснут. Овако изгледа имплементација у коду:

```
public static class InputExtensions
{
    public static bool AnyKeyDown(this List<KeyCode> keys)
    {
        bool anyKeyPressed = false;

        foreach (KeyCode keyCode in keys)
        {
            if (!Input.GetKeyDown(keyCode)) continue;
            anyKeyPressed = true;
            break;
        }

        return anyKeyPressed;
    }
}
```

2.2.2. Систем непријатељских бродова

2.2.2.1. Управљање генерацијом непријатељских бродова

EnemyInstanceManager или менаџер инстанци непријатеља је класа која је централни део система непријатеља и задужена је за управљање појављивањем и

уништавањем свих непријатељских јединица током трајања игре. Ова скрипта се бави процесом инстанцирања нових непријатеља, њиховим почетним положајем, као и њиховим елиминисањем када их играч погоди или када сударе са планетом коју штити играч. Осим тога, ова класа чува листу свих активних непријатеља и одржава бројач уништених јединица.

Скрипта иницијализује и управља важним компонентама система, као што су препознавање речи, пуцање на непријатеље и генерисање нових јединица. Поред тога, методе као што су TryFindEnemy() и ShootEnemy() су одговорне за реакцију на исправно унете речи, при чему се непријатељске јединице деактивирају и уклањају из игре.

Метода StartInstantiatingCoroutine() је одговорна за континуирано генерисање непријатеља у насумичним интервалима дефинисаним у распону *instantiateDelayRange*. Овај распон се смањује како играч напредује кроз нивое, чиме се повећава учесталост појављивања непријатеља, а игра добија на тежини. Метода InstantiateEnemy() генерише новог непријатеља на основу насумично изабране речи из система речи (WordsManager). Овако изгледа имплементација у коду:

```
private IEnumerator StartInstantiatingCoroutine()
{
    var randomDelay = (instantiateDelayRange /
Modifiers.EnemyFrequency).GetRandomInRange();
    yield return new WaitForSeconds(randomDelay );
    InstantiateEnemy();
    StartCoroutine(StartInstantiatingCoroutine());
}
```

Метода InstantiateEnemy() подешава изглед и позицију новог непријатеља на основу параметара као што су брзина, величина, и графички ефекти (сенке), и поставља га на одговарајућу почетну позицију на екрану. Овако изгледа имплементација у коду:

```
private void InstantiateEnemy()
{
    var word = WordsManager.GetRandomWord();
    var enemyPreset =
EnemyPresetsManager.Instance.GetPresetDataBasedOnWord(word);

    var enemy = Instantiate(enemyPrefab, enemiesParent).GetComponent<Enemy>();
    enemy.word = word;
    enemy.gameObject.name = word;
    enemy.SetSpriteData(enemyPreset.mainSprite, enemyPreset.whiteSprite);
    enemy.transform.localScale = enemyPreset.scale * Vector3.one;
    enemy.transform.position = GetSpawnPosition(enemy);
    enemy.GetComponent<EnemyMovementController>().StartMoving(EnvironmentRefer
ences.Instance.planet, enemyPreset.speed * Modifiers.EnemySpeed);
    ShadowPathSetter.SetShadowPath(enemy.GetComponent<ShadowCaster2D>(),
enemyPreset.shadowCaster2DPath);
    Enemies.Add(enemy);
}
```

Метода TryFindEnemy() се активира када играч унесе реч и притисне тастер за потврду. Он пролази кроз све тренутне непријатеље и проверава да ли неки од њих има реч која се подудара са унетом речју. Овако изгледа имплементација у коду:

```
private void TryFindEnemy(string word)
{
    foreach (var enemy in Enemies)
    {
        if (!enemy.word.Equals(word)) continue;
        if (enemy.gettingShot) continue;
        ShootEnemy(enemy);
        GameManager.Instance.CorrectWordSubmitted();
    }
}
```

Када је реч успешно пронађена, метода ShootEnemy() јавља складном непријатељском броду да се припреми за пројектил који ће бити испаљен ка њему, повећава бројач уништених бродова за један и јавља систему пројектила да иницира секвенцу пуцања која укључује визуелне ефекте. Овако изгледа имплементација у коду:

```
public void ShootEnemy(Enemy enemy)
{
    enemy.GetReadyForShot();
    ProjectilesManager.Instance.ShootTarget(enemy.transform);
    EnemiesDestroyed++;
}
```

Метода GameOver() чисти све преостале непријатеље са екрана и зауставља генерацију нових непријатеља.

2.2.2.2. Систем подешавања непријатеља

Систем подешавања непријатеља (Enemy Presets System) одговоран је за прилагођавање карактеристика непријатеља на основу дужине речи коју је непријатељ добио. Ово омогућава динамичко подешавање брзине, величине, изгледа и других аспеката непријатеља у зависности од речи коју представља. У овом систему се користе две главне компоненте: EnemyLevelPresetData и EnemyPresetsManager.

EnemyLevelPresetData је ScriptableObject[3] који дефинише визуелне и функционалне карактеристике појединачног непријатеља. Ова класа садржи параметре као што су брзина, величина, примарни спрајт (главни изглед непријатеља) и спрајт беле боје који се користи у анимацији припреме за уништавање. Поред тога, ту је и *shadowCaster2DPath*, који омогућава подешавање сенке коју непријатељ пројектује на екрану. Овако изгледа имплементација у коду:

```
namespace AP.EnemySystem
{
    [CreateAssetMenu(fileName = "Enemy Level Preset Data", menuName = "")]
```

[3] ScriptableObject - контејнер за податке који се користе за чување велике количине података, независно од инстанци класа

```
public class EnemyLevelPresetData : ScriptableObject
{
    public float speed;
    public float scale = 1f;
    public Sprite mainSprite;
    public Sprite whiteSprite;
    public Vector3[] shadowCaster2DPath;
}
```

Овај објекат се креира унутар Unity инспектора и омогућава једноставну конфигурацију различитих врста непријатеља, без потребе за променом кода. На овај начин, дизајнери игре могу креирати неограничен број различитих непријатељских јединица са различитим параметрима у зависности од дужине речи коју играч мора унети.

Класа EnemyPresetsManager или менаџер подешавања непријатеља управља свим доступним подешавањима непријатеља и одлучује које ће подешавање бити додељено непријатељу на основу дужине речи. Она садржи листу подешавања подељених према дужини речи (*PresetsByWordLength*), што омогућава да различити непријатељи буду генерисани у зависности од броја карактера у речима. Када се реч додели непријатељу, систем користи метода GetPresetDataBasedOnWord() да одреди које подешавање ће бити примењено. Овако изгледа имплементација у коду:

```
public EnemyLevelPresetData GetPresetDataBasedOnWord(string word)
{
    var length = word.Length;
    var level = 0;

    for (int i = 0; i < presetsByWordLength.Count; i++)
    {
        if (length >= presetsByWordLength[i].range.x && length <
presetsByWordLength[i].range.y)
        {
            level = i;
            break;
        }
    }

    return presetsByWordLength[level].enemyLevelPresetData;
}
```

2.2.2.3. Систем понашања и кретања непријатеља

Систем понашања и кретања непријатеља у игри је кључан за реализацију интеракције са играчем. Непријатељи се понашају динамично, зависећи од речи коју играч треба да унесе. Ово одељак обухвата две главне скрипте који управљају овим понашањем: Enemy и EnemyMovementController.

Класа Enemy одговорна је за визуелно приказивање непријатеља, као и за интеракцију када је непријатељ погођен или припремљен за ударац. Она садржи основне

податке као што су реч која је додељена непријатељу и његов тренутни статус (нпр. да ли је погођен).

Примарна функција ове класе је да подеси изглед непријатеља и да контролише промене у визуелизацији када непријатељ буде спреман за ударац. Када се реч коју играч уноси поклопи са речју која је додељена непријатељу, непријатељ улази у статус "припрaван за ударац" кроз методу GetReadyForShot(), што активира визуелне ефекте као што је анимација преласка боје на белу. Овако изгледа имплементација у коду:

```
public void GetReadyForShot()
{
    gettingShot = true;
    m_colorTween = whiteSpriteRenderer.DOFade(1f,
s_colorTweenDuration).SetEase(s_colorTweenEase).Play();
    Destroy(m_title.gameObject);
}
```

Поред тога, сваки непријатељ има наслов који приказује реч коју треба унети да би непријатељ био поражен. Овај наслов се креира динамички и позиционира на екрану преко WordUI компоненте, која се инстнацира унутар Start() методе. Тиме се обезбеђује да сваки непријатељ има јасно видљив текст који играч мора правилно да унесе како би га уништио.

Док Enemy контролише визуелне аспекте и понашање непријатеља, EnemyMovementController управља кретањем непријатеља на екрану. Користећи библиотеку DOTween, ова скрипта омогућава глатко кретање непријатељских јединица према мети од своје стартне позиције.

Метода StartMoving() иницира кретање непријатеља према мети при подешеној брзини, што је параметар који је променљив и зависи од дужине речи. Овако изгледа имплементација у коду:

```
public void StartMoving(Transform target, float speed)
{
    m_moveTween = transform
        .DOMove(target.position, speed)
        .SetSpeedBased(true)
        .Play();
}
```

2.2.3. Систем појачивача

Појачивачи представљају специјалне механизме у игри који привремено или трајно пружају играчу одређене предности. Постоје две главне категорије појачивача: они који тренутно изврше одређену акцију при активирању, попут враћања животних поена или уништавања свих непријатељских бродова на екрану, и дуготрајни појачивачи, који трају одређени временски период пре него што се деактивирају. Ова два типа су представљена у хијерархијском систему наслеђивања, где сви појачивачи наслеђују базну апстрактну класу Powerup, док дуготрајни пауер-апови наслеђују апстрактну класу DurationalPowerup, која додатно дефинише временску дужину њиховог трајања.

Класа Powerup представља апстрактну базу свих појачивача у игри. Она дефинише основне елементе појачивача, као што су тип појачивача и визуелни ефекти који се активирају када је појачивач у функцији. Њена метода Activate() је виртуелна функција која се позива када појачивач постане активан, и она такође позива одговарајуће визуелне ефекте преко референце на инстанцу класе PowerupVisualsHandler, која управља визуелним приказима. Овако изгледа имплементација у коду:

```
using AP.PowerupsSystem.VisualEffects;
using UnityEngine;

namespace AP.PowerupsSystem
{
    public abstract class Powerup : MonoBehaviour
    {
        public PowerupType powerupType;
        public PowerupVisualsHandler powerupVisualsHandler;

        public virtual void Activate()
        {
            powerupVisualsHandler.OnActivate();
        }
    }
}
```

Класа DurationalPowerup проширује основну класу Powerup и додаје функционалност временски ограничених појачивача. Ови појачивачи не само да се активирају, већ након одређеног временског периода, аутоматски се деактивирају. Њена метода Activate() активира појачивач и покреће процес деактивације након истека временског периода који је одређен параметром *duration*. Овако изгледа имплементација у коду:

```
using System.Collections;
using UnityEngine;

namespace AP.PowerupsSystem
{
    public abstract class DurationalPowerup : Powerup
    {
        public float duration;

        public override void Activate()
        {
            base.Activate();
            StartCoroutine(DeactivatePowerupInSeconds(duration));
            PowerupsManager.Instance.isAnyPowerupActive = true;
        }

        public IEnumerator DeactivatePowerupInSeconds(float seconds)
```

```
        {
            yield return new WaitForSeconds(seconds);
            DeactivatePowerup();
        }

        public virtual void DeactivatePowerup()
        {
            powerupVisualsHandler.OnDeactivate();
            PowerupsManager.Instance.isAnyPowerupActive = false;
        }
    }
}
```

Обе класе, Powerup и DurationalPowerup, ослањају се на класе које су задужене за визуелне ефекте појачивача. Овај визуелни систем је имплементиран преко апстрактне класе PowerupVisualsHandler, која дефинише методе OnActivate() и OnDeactivate(). Ове методе су одговорне за приказ визуелних ефеката када је појачивач активиран и деактивиран, попут post-processing[4] ефеката. Овако изгледа имплементација у коду:

```
namespace AP.PowerupsSystem.VisualEffects
{
    public abstract class PowerupVisualsHandler : MonoBehaviour
    {
        public abstract void OnActivate();
        public abstract void OnDeactivate();
    }
}
```

Систем појачивача у игри је управљан преко класе PowerupsManager или менаџер појачивача, која служи као централни механизам за руковање свим активирањем и деактивирањем појачивача. Менаџер појачивача одржава листу свих типова појачивача у игри и води евиденцију о томе да ли је неки појачивач тренутно активан преко промењиве *isAnyPowerupActive*. Овако изгледа имплементација у коду:

```
namespace AP.PowerupsSystem
{
    public class PowerupsManager : GloballyAccessibleBase<PowerupsManager>
    {
        public bool isAnyPowerupActive = false;

        [SerializeField] private List<Powerup> powerups;

        private void Start()
        {
            GameManager.Instance.OnGameOver += DisableAllPowerups;
        }
```

[4] Post-processing - tehnike koje se primenjuju na slike ili video zapise nakon što je osnovna obrada završena.

```
        public void ActivatePowerup(PowerupType powerupType)
        {
            FindPowerupByType(powerupType).Activate();
        }

        private Powerup FindPowerupByType(PowerupType powerupType)
        {
            var index = 0;

            for (int i = 0; i < powerups.Count; i++)
            {
                if (powerups[i].powerupType != powerupType) continue;
                index = i;
                break;
            }

            return powerups[index];
        }

        private void DisableAllPowerups()
        {
            foreach (Powerup powerup in powerups)
            {
                DurationalPowerup target = powerup as DurationalPowerup;
                if (target == null) { continue; }

                target.StopAllCoroutines();
                target.DeactivatePowerup();
            }
        }
    }
}
```

У игри постоје четири конкретне имплементације основне класе Powerup или DurationalPowerup. Сваки од ових појачивача је јединствена по свом ефекту и доприноси различитим аспектима играња. У наставку ћемо детаљно описати све четири моћи и њихове имплементације у коду.

Први појачивач који ћемо описати је GainHealthPowerup, која омогућава играчу да обнови део својих животних поена. Ова моћ је корисна у тренуцима када је играчев живот на критично ниском нивоу и потребни су му додатни животни поени како би наставио да се бори против непријатеља.

Код за GainHealthPowerup је једноставан и јасно дефинисан. Ова моћ наслеђује основну класу *Powerup*, и њена главна функција је метода Activate(), која се позива када играч активира ову моћ. Када се моћ активира, она позива функцију из менаџера игре која додаје одређену количину животних поена играчу, а затим поставља промењљиву у менаџеру појачивача да ниједан појачивач није активан. Овако изгледа имплементација у коду:

```
namespace AP.PowerupsSystem
{
    public class GainHealthPowerup : Powerup
    {
        public int gainAmount;

        public override void Activate()
        {
            base.Activate();
            GameManager.Instance.GainHealth(gainAmount);
            PowerupsManager.Instance.isAnyPowerupActive = false;
        }
    }
}
```

Следећи појачивач који ћемо описати је PointsMultiplierPowerup. Овај појачивач омогућава играчу да на одређено време добија више поена него што би то иначе био случај. Ово може бити изузетно корисно када играч жели да брзо повећа своје бодове и напредује на табели са резултатима.

PointsMultiplierPowerup је наслеђен од класе DurationalPowerup, што значи да има временски ограничен ефекат. При активацији, овај појачивач подешава множилац за поене у менаџеру игре на вредност дефинисану у самом појачивачу. Након што време појачивача истекне, његов ефекат се поништава и множилац за поене враћа се на почетну вредност. Овако изгледа имплементација у коду:

```
namespace AP.PowerupsSystem
{
    public class PointsMultiplierPowerup : DurationalPowerup
    {
        public float multiplier;

        public const float ORIGINAL_MULTIPLIER = 1.0f;

        public override void Activate()
        {
            base.Activate();
            GameManager.Instance.pointsMultiplierFromPowerup = multiplier;
        }

        public override void DeactivatePowerup()
        {
            base.DeactivatePowerup();
            GameManager.Instance.pointsMultiplierFromPowerup =
ORIGINAL_MULTIPLIER;
        }
    }
}
```

Трећи појачивач који ћемо представити је SlowmotionPowerup. Овај појачивач омогућава играчу да привремено успори време у игри, чиме добија више времена да реагује на опасности или уништи непријатељске бродове. Ефекат успоравања је посебно користан у ситуацијама када је непријатељских бродова много или су брзи.

SlowmotionPowerup је такође наслеђен од класе DurationalPowerup, што значи да његов ефекат траје ограничено време. При активацији, овај појачивач мења *Time.timeScale* на вредност дефинисану као *slowmotionTimescale*, чиме успорава све у игри. По истеку појачивача, вредност *Time.timeScale* се враћа на оригиналну вредност, чиме се игра враћа у нормалан ток. Овако изгледа имплементација у коду:

```
using UnityEngine;

namespace AP.PowerupsSystem
{
    public class SlowmotionPowerup : DurationalPowerup
    {
        public float slowmotionTimescale;

        private const float ORIGINAL_TIMESCALE = 1.0f;

        public override void Activate()
        {
            base.Activate();
            Time.timeScale = slowmotionTimescale;
        }

        public override void DeactivatePowerup()
        {
            base.DeactivatePowerup();
            Time.timeScale = ORIGINAL_TIMESCALE;
        }
    }
}
```

Последњи појачивач, који је можда и најмоћнији у игри, јесте ShootAllEnemiesPowerup. Овај појачивач омогућава играчу да тренутно уништи све непријатељске бродове који су присутни на екрану. Овај ефекат је посебно користан када је велики број непријатеља на путу ка планети и када је неопходно хитно реаговати како би се избегло оштећење.

ShootAllEnemiesPowerup је класа наслеђена од класе Powerup и функционише тако што пролази кроз све активне непријатеље у игри користећи листу *Enemies* из менаџера инстанци непријатеља, и за сваки непријатељски брод позива функцију ShootEnemy(). Ова функција одмах уништава непријатеља, као да је играч успешно написао одговарајућу реч за уништење. Овако изгледа имплементација у коду:

```
using AP.EnemySystem;

namespace AP.PowerupsSystem
```

```
{
    public class ShootAllEnemiesPowerup : Powerup
    {
        public override void Activate()
        {
            base.Activate();

            foreach (var enemy in EnemyInstanceManager.Instance.Enemies)
            {
                EnemyInstanceManager.Instance.ShootEnemy(enemy);
            }

            PowerupsManager.Instance.isAnyPowerupActive = false;
        }
    }
}
```

2.2.3.1. Носачи појачивача

У систему појачивача у оквиру ове игре, постоји посебан подсистем који се односи на управљање „носачима појачивача“ — објектима који носе одређени појачивач и чије уништавање активира дати појачивач. Носачи се креирају као независни објекти у игри који такође садрже реч коју играч мора правилно откуцати да би уништио носач и активирао појачивач.

Прва и најбитнија скрипта система носача појачивача јесте сам носач појачивача (PowerupCarrier). Ова скрипта је одговорна за сваки индивидуални носач појачивача у игри. Чува врсту појачивача и реч коју играч мора да откуца како би активирао појачивач. Такође генерише визуелни елемент са речју изнад самог носача. Затим имамо скрипту одговорну за кретање појачивача (PowerupCarrierMovementController) којој се додели почетна и крајња позиција за кретњу. Ако носач изађе из видног поља играча, систем га уништава.

Класа PowerupCarriersManager или менаџер носача појачивача је одговорна за управљање појављивањем, кретањем и уништавањем носача појачивача у игри, који носе одређене речи. Ова класа има кључну улогу у процесу активирања појачивача, јер када играч успешно унесе исправну реч, носач се уништава, а појачивач се активира.

По завршетку иницијализације игре, менаџер носача појачивача покреће coroutine[5] која је задужена за периодично генерисање носача појачивача. Интервал између појављивања носача зависи од вредности променљиве која контролише учесталост појављивања.

Менаџер новача појачивача је такође повезан са системом за унос речи, где путем методе TryFindCarrier() проверава да ли унета реч одговара речи на тренутном носачу. У случају успешног погађања, позива се метода из класе менаџера појачивача, који је задужен за активацију одговарајућег појачивача.

[5] Coroutine - посебан начин за извођење асинхроних операција, уместо да се извршавају све у једном тренутку

2.2.4. Систем пројектила

Систем пројектила у игри је задужен за управљање кретањем и колизијом пројектила са циљним објектима, што је кључно за механизам уништавања непријатељских бродова. Састоји се од две главне компоненте: скрипте Projectile, која је повезана са самим објектом пројектила у игри, и менаџера пројектила (ProjectilesManager), који управља свим пројектилима у игри, њиховим путањама и интеракцијама са метама.

Класа Projectile је задужена за кретање појединачног пројектила у игри. Пројектил се креће дуж путање коју одређује менаџер пројектила, а брзина и крива анимације такође се подешавају у зависности од потреба.

Када пројектил стигне до свог циља, позива догађај *OnHit*, који сигнализира успешан погодак. Овај догађај је важан јер је на њега претплаћен менаџер пројектила, који ће даље обрађивати оно што треба да се догоди након што пројектил погоди мету. Такође, при поготку, честице (визуелни ефекат) се искључују и уништавају након кратког кашњења, чиме се постиже визуелни ефекат експлозије. Овако изгледа имплементација у коду:

```
namespace AP.ProjectileSystem
{
    public class Projectile : MonoBehaviour
    {
        public event Action OnHit;

        [SerializeField] private ParticleSystem particles;

        private Tween m_pathTween = null;
        private const float PARTICLES_DESTRUCTION_DELAY = .35f;

        public void ShootTarget(Vector3[] path, float speed, AnimationCurve
curve)
        {
            m_pathTween = transform
                .DOPath(path, speed, PathType.CatmullRom)
                .SetSpeedBased(true)
                .SetEase(curve)
                .OnComplete(delegate
                {
                    OnHit?.Invoke();
                    particles.enableEmission = false;
                    particles.transform.parent = null;
                    Destroy(particles.gameObject,
PARTICLES_DESTRUCTION_DELAY);
                })
                .Play();
        }

        private void OnDestroy() => m_pathTween?.Kill();
```

```
    }
}
```

Менаџер пројектила управља свим пројектилима у игри и одговара за њихово испаљивање и погодак циља. Главна одговорност ове класе је да одреди путању пројектила, његову брзину и анимацију, као и да иницира испаљивање пројектила када је то потребно. Путем јавне методе ShootTarget(), друге класе у игри могу затражити испаљивање пројектила ка одређеној мети. Овако изгледа имплементација у коду:

```
public void ShootTarget(Transform target)
{
    var projectile = Instantiate(projectilePrefabs.GetRandomElement(),
projectilesParent).GetComponent<Projectile>();
    projectile.transform.position = RandomPointOnPlanet(planetRadius, planet);
    projectile.name = $"Projectile_{target.name}";

    Vector3[] projectilePath = new Vector3[]
    {
        (projectile.transform.position - planet.position).normalized *
projectileLaunchDistance + projectile.transform.position,
        target.position
    };

    projectile.OnHit += () => ProjectileHitTarget(target, projectile);
    projectile.ShootTarget(projectilePath, projectileSpeed, animationCurve);
}
```

Када пројектил погоди мету, позива се *OnTargetShot* догађај са метом као параметром. Ово омогућава другим деловима система, као што је менаџер инстанци непријатеља, да провере да ли је уништени објекат непријатељски брод. Овако изгледа имплементација у коду:

```
private void ProjectileHitTarget(Transform target, Projectile projectile)
{
    OnTargetShot?.Invoke(target);
    Destroy(projectile.gameObject);
}
```

У класи менаџер пројектила постоји метода RandomPointOnPlanet(), која генерише насумичну тачку на површини планете, али само на десној страни. Ова тачка представља почетну позицију пројектила приликом испаљивања. Након тога, креира се пут до одређене тачке која се налази на путањи од стартне позиције пројектила до нормале планете у тој тачки. Растојање до ове тачке се одређује променљивом *projectileLaunchDistance*. Коначно, последња тачка на путањи је сама мета.

```
private Vector3 RandomPointOnPlanet(float radius, Transform planet)
{
    var vector2 = UnityEngine.Random.insideUnitCircle.normalized * radius;
    return new Vector3(Mathf.Abs(vector2.x), vector2.y, 0f) +
planet.transform.position;
}
```

На овај начин, пројектил изгледа као да се прво лансира са планете, а затим се закључава на мету, као што би то урадила права ракета.

Укратко, систем пројектила у овој игри представља механизам за управљање испаљивањем пројектила и интеракцијом са метама. Кроз координацију између класа Projectile и ProjectilesManager, овај систем обезбеђује да се пројектил правилно креће дуж путање и активира жељени ефекат приликом поготка у мету.

2.2.5. Систем камера

Систем камера у овој игри је веома једноставан и садржи само једну компоненту чија је основна одговорност тресење камере када непријатељски брод погоди планету. Овај ефекат користи библиотеку DOTween за постизање визуелног тресења и игра важну улогу у томе да играч осети последицу удара непријатељског брода, чиме се повећава укупни доживљај игре.

У имплементацији, класа CameraShaker ослушкује догађај удара непријатеља у планету и активира тресење камере. Сваки пут када дође до удара, камера се затресе користећи подешене параметре као што су трајање, јачина и вибрација. Овако изгледа имплементација у коду:

```
namespace AP.CameraSystem
{
    public class CameraShaker : MonoBehaviour
    {
        [SerializeField] private float duration;
        [SerializeField] private float strength = 1;
        [SerializeField] private int vibrato = 10;

        private Tween m_cameraShake = null;

        private void Start() => PlanetBehaviour.Instance.OnHitByEnemy += _ =>
ShakeCamera();

        private void ShakeCamera()
        {
            if (m_cameraShake != null)
            {
                m_cameraShake.Rewind();
                m_cameraShake.Kill();
            }

            m_cameraShake = transform.DOShakePosition(duration, strength,
vibrato, 90f, false, true, ShakeRandomnessMode.Harmonic).Play();
        }
    }
}
```

2.2.6. Менаџер игре

Менаџер игре или GameManager је класа која представља централни елемент игре, задужен за праћење свих релевантних података током сесије игре. Главни задатак менаџера игре је да координише ток игре, осигурава праћење и ажурирање свих релевантних параметара и реагује на важне догађаје који се дешавају током игре, као што су уништавање непријатељског брода или оштећење планете од стране непријатеља. На овај начин, менаџер игре представља централни „мозак“ игре који осигурава да се све компоненте синхронизују и функционишу у складу са тренутним стањем игре.

Кључни подаци које менаџер игре прати јесу:

- Животни поени (*Health*) и број максималних животних поена (*MaxHealth*): Ови параметри контролишу колико оштећења играч може да прими пре него што изгуби. Када животни поени падну на 0, игра је завршена. Поред тога, параметар *HealthLeftNormalized* користи се за графичко приказивање преосталог здравља као процента у односу на максимално здравље.
- Бодови (Points): Овај параметар бележи укупан број бодова које је играч освојио током тренутне сесије. Бодови се повећавају уништавањем непријатељских бродова.
- Ниво (*Level*), Искуство (*Experience*) и Искуство до следећег нивоа (*ExperienceTillNextLevel*): Ови параметри прате напредак играча. Искуство се добија уништавањем бродова, а када играч сакупи довољно искуства, он прелази на следећи ниво. Параметар *ExperienceProgressNormalized* користи се за графички приказ напредовања играча до следећег нивоа.
- Тренутна комбинација (*CurrentCombo*) и Најдужа комбинација (*HighestCombo*): Ови параметри мере број узастопно уништених непријатељских бродова без примања оштећења. У случају да планета буде погођена, тренутна комбинација се ресетује, али ако је достигнута нова најдужа комбинација, она се ажурира.
- Трајање игре (*GameDurationInSeconds*): Параметар који прати колико дуго је игра трајала.
- Број речи у минути (*WordsPerMinute*): Овај параметар израчунава брзину писања играча на основу броја тачно написаних речи и времена које је провео куцајући.
- Статус игре (*IsGameOver*): Променљива која указује на то да ли је игра завршена.

Менаџер игре је такође задужен за реаговање на кључне догађаје у игри. Подешен је да слуша два догађаја, прво од којих је уништење непријатељског брода. Када је непријатељски брод уништен, менаџер игре обавља више корака:

- Израчунава и додељује број бодова на основу дужине речи везане за уништени брод.
- Играч добија искуство за сваку уништену реч, а ако достигне довољно искуства, повећава му се ниво. У том случају, покреће се догађај OnLevelUp који омогућава другим компонентама (најчешће визуелним) да реагују на повећање нивоа.
- Тренутна комбинација уништених бродова се повећава за један.

Овако изгледа имплементација у коду:

```
private void EnemyDestroyed(Enemy enemy)
{
    var wordLength = enemy.word.Length;
    Points += (int)(wordLength * pointsPerLetter * Modifiers.PointsModifier *
pointsMultiplierFromPowerup);
    GainExperience(wordLength);
    CurrentCombo++;
}
```

Други догађај на који је менаџер игре претплаћен јесте оштећење планете од стране непријатељског брода. У случају да непријатељски брод удари у планету, менаџер игре смањује животне поене играча. Ако животни поени падну на нулу или испод, игра је завршена и позива се догађај *OnGameOver* који обавештава све друге компоненте да престану са својим функцијама. Поред тога, ако игра није завршена, тренутна комбинација уништених бродова се ресетује и ажурира се највећа комбинација ако је достигнута нова најдужа комбинација. Овако изгледа имплементација у коду:

```
private void PlanetHit(Enemy enemy)
{
    LoseHealth(enemy.word.Length * DAMAGE_PER_LETTER);

    if (HighestCombo < CurrentCombo)
    {
        HighestCombo = CurrentCombo;
    }

    CurrentCombo = 0;
}
```

Функција SubscribeToEvents() је задужена за повезивање менаџера игре са релевантним догађајима у игри. Овако изгледа имплементација у коду:

```
private void SubscribeToEvents()
{
    EnemyInstanceManager.Instance.OnEnemyDestroyed += EnemyDestroyed;
    PlanetBehaviour.Instance.OnHitByEnemy += PlanetHit;
}
```

На овај начин, менаџер игре слуша када је непријатељ уништен или када је планета погођена и реагује у складу са тим.

Менаџер игре је срж механике игре, задужен за управљање подацима и реаговање на све кључне догађаје. Без ове компоненте, игра не би могла да прати напредак играча, нити би могла адекватно да одговори на уништавање непријатеља или оштећење планете. Као такав, ова компонента је неопходна за функционисање целокупне игре и повезивање осталих система у једну целину.

2.2.7. Модификатори

У игри је имплементиран концепт модификатора који омогућавају играчима да прилагоде неке параметре игре, а у зависности од те прилагодбе, добијају или губе множитељ поена (*PointsModifier*). Ова механика побољшава играчко искуство и пружа

играчима могућност да утичу на тежину игре према својим жељама. Ова функционалност подстиче играче да експериментишу са различитим стратегијама, чиме се продубљује интеракција са игром и повећава задовољство.

У оквиру статичке класе Modifiers, дефинисали смо неколико кључних параметара:

- *EnemySpeed*: Брзина непријатеља.
- *EnemyFrequency*: Учесталост појављивања непријатеља.
- *PowerupSpeed*: Брзина појачивача.
- *PowerupFrequency*: Учесталост појављивања појачивача.

Сваки од ових параметара има подразумевану вредност од 1. Када играч променује ове вредности, оне директно утичу на *PointsModifier*, што представља множитељ поена у игри. Овако изгледа имплементација у коду:

```
public static class Modifiers
{
    public static float PointsModifier { get; private set; } = 1f;

    public static float EnemySpeed { get; private set; } = 1f;
    public static float EnemyFrequency { get; private set; } = 1f;
    public static float PowerupSpeed { get; private set; } = 1f;
    public static float PowerupFrequency { get; private set; } = 1f;
}
```

Класа садржи методе за ресетовање параметара на подразумеване вредности и методе за подешавање појединачних модификатора:

- ResetToDefault(): Враћа све модификаторе на њихове подразумеване вредности и поново израчунава множитељ поена.
- EvaluatePointsModifier(): Израчунава нови множитељ поена на основу тренутних вредности модификатора. Формула за израчунавање узима у обзир производ брзине и учесталости непријатеља, као и однос брзине и учесталости појачивача.

Овако изгледа имплементација у коду:

```
public static class Modifiers
{
    public static void ResetToDefault()
    {
        EnemySpeed = 1f;
        EnemyFrequency = 1f;
        PowerupSpeed = 1f;
        PowerupFrequency = 1f;
        EvaluatePointsModifier();
    }

    public static void EvaluatePointsModifier()
```

```
    {
        float enemyModifiers = EnemySpeed * EnemyFrequency;
        float powerupsModifiers = PowerupSpeed / PowerupFrequency;

        // Calculate overall points modifier
        PointsModifier = enemyModifiers * powerupsModifiers;
    }
}
```

2.2.8. Систем резултата

Систем резултата представља важан аспект игре, обухватајући механизме који управљају и чувају перформансе играча након сваке партије. Овај систем је дизајниран да пружи структуриран и организован начин за бележење резултата, који играчима омогућава да прате свој напредак током времена. Када играч заврши игру, а резултати су уписани, систем се активира, чиме се иницира поступак чувања важних информација.

Када играч изгуби, што је неизбежно у бесконачној игри, активира се механизам за чување резултата. У овом тренутку, менаџер игре шаље обавештење класи менаџеру резултата (*ScoresManager*), који има задатак да обради нови резултат. Ова комуникација између компонената игре осигурава да се резултати додају у тачан редослед и да се правилно управља подацима о играчевом учинку. Ова интеракција не само да је функционална, већ и важна за одржавање структуре система. Овако изгледа имплементација у коду:

```
private void GameLost()
{
    OnGameOver?.Invoke();

    Score score = new Score(Points, WordsPerMinute, Modifiers.PointsModifier);
    ScoresManager.Instance.AddScore(score);
}
```

Када менаџер резултата прими обавештење о новом резултату, он додаје овај резултат у списак, а све резултате организује у опадајућем редоследу. Овакво уређивање резултата подразумева да играчи могу лако видети своје позиције у односу на друге, што ствара подстрек за побољшање. Важно је напоменути да се резултати бележе у класи Score, која обухвата неколико кључних променљивих. У ову класу спадају *Points*, који представља укупан број поена које је играч зарадио, *WPM* (words per minute), што указује на ефикасност и брзину куцања, као и *DifficultyMultiplier*, који је множитељ поена и одражава сложеност игре. Ови параметри чине основу система резултата и играчима пружају јасан увид у њихов учинак. Овако изгледа имплементација у коду:

```
namespace AP.ScoreSystem
{
    [System.Serializable]
    public class Score
    {
        public int Points { get; set; } = 0;
        public int WPM { get; set; } = 0;
        public float DifficultyMultiplier { get; set; } = 1f;
```

```
        public Score(int points, int wpm, float difficultyMultiplier)
        {
            Points = points;
            WPM = wpm;
            DifficultyMultiplier = difficultyMultiplier;
        }
    }
}
```

Након што су резултати успешно забележени, долази до важне фазе у систему: чувања података. Када играч одлучи да напусти апликацију, менаџер резултата иницира процес чувања резултата у XML[6] фајл. Овај корак је кључан, јер осигурава да сви подаци буду сачувани и доступни за будуће коришћење. Користећи XmlSerializer[7] и FileStream[8], систем управља подацима на ефикасан начин, што значи да се резултати могу лако учитати и анализирати. Чување у XML формату не само да побољшава структуру података, већ и чини њихово будуће коришћење једноставнијим. Овако изгледа имплементација у коду:

```
private void SaveScores()
{
    XmlSerializer xmlSerializer = new(typeof(ScoresListEntry));
    FileStream fileStream =
File.Create(Path.Combine(Application.persistentDataPath, FILE_NAME));
    ScoresListEntry scoresListEntry = new(Scores);
    xmlSerializer.Serialize(fileStream, scoresListEntry);
    fileStream.Close();
}
```

Када играч поново покрене апликацију, менаџер резултата покушава да учита постојећи scores.xml фајл, који се налази у Application.persistentDataPath + FILE_NAME (где је *FILE_NAME* назив фајла, тј. scores.xml). Ова процедура осигурава да су сви резултати доступни сваки пут када играч покрене игру, што омогућава континуирано праћење напредка. Играчи тако могу да виде своје претходне перформансе и упоређују их са новим резултатима, што их подстиче да улажу више напора у постизање бољих резултата. Овако изгледа имплементација у коду:

```
private void LoadScores()
{
    Scores = new List<Score>();

    if (!File.Exists(Path.Combine(Application.persistentDataPath, FILE_NAME)))
    {
        return;
    }
```

[6] XML - текстуални формат за представљање структурираних података.
[7] XMLSerializer – класа која серијализује и десеријализује објекте у и из XML докумената.
[8] FileStream – класа која пружа *Stream* за фајл, подржавајући и синхроне и асинхроне операције читања и писања.

```
    XmlSerializer xmlSerializer = new XmlSerializer(typeof(ScoresListEntry));
    FileStream fileStream =
File.Open(Path.Combine(Application.persistentDataPath, FILE_NAME),
FileMode.Open);
    Scores = new((xmlSerializer.Deserialize(fileStream) as
ScoresListEntry).scores);
    fileStream.Close();
}
```

Систем резултата у овој игри не представља само механизам за чување података, већ и важан мотиватор за играче. Пружајући увид у напредак и позицију у односу на друге, ова механика охрабрује играче да се враћају и усавршавају своје вештине. На тај начин, играчи не само да се такмиче сами са собом, већ и са другима, што додаје нову димензију играчком искуству. Систем резултата стога значајно доприноси динамици игре, чинећи је занимљивијом и изазовнијом, као и подстичући ангажовање и конкурентност унутар игре.

2.2.9. Графички кориснички интерфејс

Графички кориснички интерфејс (Graphical User Interface - GUI) је врста интерфејса који омогућава корисницима да комуницирају са уређајем преко графичких елемената попут икона и индикатора. У видео играма се чешће користи термин head-up display (HUD), који потиче од истоимених дисплеја на савременим ваздухопловима.

2.2.9.1. Менији

Играч при уласку у игру види главни мени који је приказан на слици 2.x. На менију se налазе три дугмета: *Play* (Играј), *Highscores* (најбољи резултати) и *Exit* (Изађи из игре). Играч може да кликне на било које дугме са левим тастером миша.

Слика 2.4- Приказ главног менија са дугмади за започетак нове игре, виђање најбољих резултата и излазак из игре

Притиском дугмета за играње или најбоље резултате покреће анимацију померања свих менија улево или удесно. У класи MainMenuContainersAnimator постоје три методе које су одговорне за анимације клизања свих менија док истовремено

осигурава да не може да се покрене нова анимација ако је једна већ покренута: DisplayHighscores(), DisplayMainMenu() и DisplayModifiers(). Овако изгледа имплементација у коду:

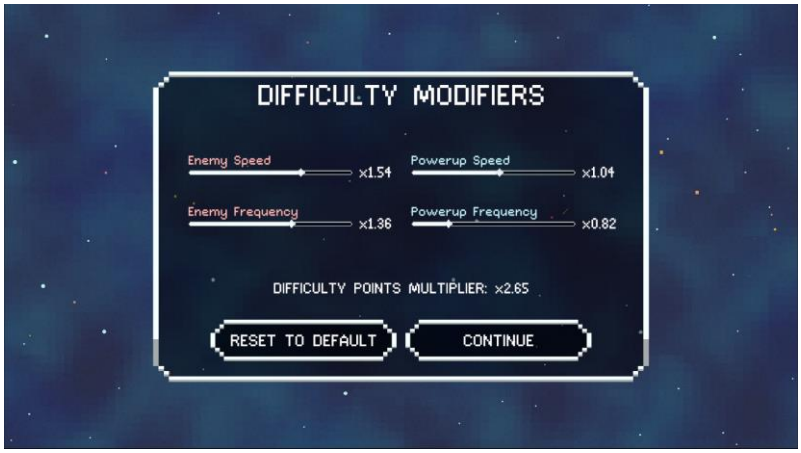
```
public void DisplayHighscores()
{
    if (m_menuState == MenuState.Highscores || m_isAnimating) { return; }

    m_isAnimating = true;

    m_sequence = DOTween.Sequence();
    m_sequence.Append(menuContainer.DOLocalMove(Vector2.right *
REFERENCE_WIDTH, duration).SetEase(ease));
    m_sequence.Join(highscoresContainer.DOLocalMove(Vector2.zero,
duration).SetEase(ease));
    m_sequence.Join(modifiersContainer.DOLocalMove(Vector2.right * 2 *
REFERENCE_WIDTH, duration).SetEase(ease));
    m_sequence.OnComplete(() =>
    {
        m_menuState = MenuState.Highscores;
        m_isAnimating = false;
    }).Play();
}

public void DisplayModifiers()
{
    if (m_menuState == MenuState.Modifiers || m_isAnimating) { return; }

    m_isAnimating = true;

    m_sequence = DOTween.Sequence();
    m_sequence.Append(menuContainer.DOLocalMove(Vector2.left *
REFERENCE_WIDTH, duration).SetEase(ease));
    m_sequence.Join(highscoresContainer.DOLocalMove(Vector2.left * 2 *
REFERENCE_WIDTH, duration).SetEase(ease));
    m_sequence.Join(modifiersContainer.DOLocalMove(Vector2.zero,
duration).SetEase(ease));
    m_sequence.OnComplete(() =>
    {
        m_menuState = MenuState.Modifiers;
        m_isAnimating = false;
    }).Play();
}

public void DisplayMenu()
{
    if (m_menuState == MenuState.Menu || m_isAnimating) { return; }
```

```
    m_isAnimating = true;

    m_sequence = DOTween.Sequence();
    m_sequence.Append(menuContainer.DOLocalMove(Vector2.zero,
duration).SetEase(ease));
    m_sequence.Join(highscoresContainer.DOLocalMove(Vector2.left *
REFERENCE_WIDTH, duration).SetEase(ease));
    m_sequence.Join(modifiersContainer.DOLocalMove(Vector2.right *
REFERENCE_WIDTH, duration).SetEase(ease));
    m_sequence.OnComplete(() =>
    {
        m_menuState = MenuState.Menu;
        m_isAnimating = false;
    }).Play();
}
```

Притиском дугмета за најбоље резултате нас води на нови мени са анимацијом преклизавања свих менија удесно. На овом менију можемо видети све сачуване резултате од претходних играња. Такође, постоји дугме *Return* (назад) чијим активирањем нас враћа на главни мени.

Слика 2.5- Приказ менија са најбољим резултатима

Притиском дугмета за играње нас води на нови мени са анимацијом преклизавања свих менија у лево. На овом се налази панел са клизачима за подешавање модификатора за следећу игру. У реалном времену у току подешавања, играч може видети како то исто подешавање утиче на множитељ поена.



Слика 2.6 - Приказ менија са модификаторима

Стиском дугмета *Reset To Default* (Врати на подразумевано стање) враћамо све параметре на подразумеване вредности, док стиском дугмета *Continue* (Настави) продужујемо у саму игру. Класа која је одговорна за модификаторе у контексту корисничког интерфејса се назива ModifiersUIManager. У њој се налазе референце за клизаче и текст које се налазе на менију модификатора. У методи Awake()[9] су постављени слушаоци догађаја за промене вредности на сваком клизачу. Мењањем вредности било ког клизача се позива одговарајућа метода која јавља статичкој класи Modifiers да ажурира модификаторе и множитељ поена, након чека се ажурирају и текстуална поља која приказују те исте вредности. Овако изгледа имплементација у коду:

```
private void Awake()
{
    enemySpeedSlider.onValueChanged.AddListener(_ => UpdateEnemySpeed());
    enemyFrequencySlider.onValueChanged.AddListener(_ =>
UpdateEnemyFrequency());
    powerupSpeedSlider.onValueChanged.AddListener(_ => UpdatePowerupSpeed());
    powerupFrequencySlider.onValueChanged.AddListener(_ =>
UpdatePowerupFrequency());
}
```

При завршетку игре, на екрану се појављује мени за крај игре. На панелу се налазе статички подаци о игри, као што је приказано на слици 2.x. Два дугмета су приказана на њему. Притискањем дугмета *Restart* (започни опет) ми започињемо игру испочетка, док стискањем дугмета *Main Menu,* ми *се* враћамо на главни мени.

[9] Awake() - метода у животном циклусу компоненти *Unity* платформе. Позива се када се објекат први пут иницијализује.

Слика 2.7- Приказ менија за крај игре

2.2.9.2. HUD

HUD је метода приказивања битних информација играчу као део корисничког интерфејса игре. HUD се приказује докле год је играч жив и док игра траје. Уместо да се сви контролери за HUD налазе у једној скрипти, раздвојене су у више њих, како би свака имала своју специфичну одговорност. Овај приступ прати принцип *Single Responsibility* из SOLID[10] принципа дизајна, где свака класа или модул треба да има један задатак.

У доњем делу екрана се налази панел који је проширен од леве до десне стране екрана. У том панелу се налазе најбитније информације везане за игре. Са леве стране тог панела су животни поени приказани на два начина. Први начин је текстуалан, где је приказан број тренутних животних поена и максималан број животних поена. Други начин је у облику клизача, познатог као *Health Bar,* који визуелно представља колико животних поена играч има у односу на максималан број животних поена. Овај клизач се пуни или празни, пружајући играчу брзу и јасну визелну повратну информацију о њиховом стању током игре.

Слика 2.8- Приказ животних поена у HUD-у

За контролу приказа животних поена је одговорна класа HealthBar, која садржи референцу за текст и за клизач. Чим се започне сесија, инстанца ове класе у сцени се претплати за догађај *OnHealthChange* унутар менаџера игре. Када се призове тај догађај, HealthBar реагује позивањем методе UpdateHealth() у којој ажурира вредности текстуалног приказа животних поена и клизача. Овако изгледа имплементација у коду:

```
namespace AP.UI
{
    public class HealthBar : MonoBehaviour
```

[10] SOLID - акроним за пет принципа дизајна који имају за циљ да учине објектно оријентисане дизајне разумљивијим, флексибилнијим и лакшим за одржавање.

```
    {
        [SerializeField] private TextMeshProUGUI healthValue;
        [SerializeField] private Slider healthSlider;

        private void Start()
        {
            SubscribeToEvents();
            UpdateHealth();
        }

        private void SubscribeToEvents()
        {
            GameManager.Instance.OnHealthChange += _ => UpdateHealth();
        }

        private void UpdateHealth()
        {
            healthValue.text =
$"{GameManager.Instance.Health}/{GameManager.Instance.MaxHealth}";
            healthSlider.value = GameManager.Instance.HealthLeftNormalized;
        }
    }
}
```

Тренутни ниво и искуство су приказани десно од животних поена, на сличан начин као и сам систем животних поена. Ниво је представљен у текстуалном формату, док искуство има свој клизач који приказује колико искуства тренутно имамо у односу на потребно искуство за следећи ниво. Када играч напредује у нивоу, текст се визуелно ажурира на начин да се текст увећа и обоји у зелено, а затим врати на оригиналну величину и боју, чиме дајемо визуелни сигнал да је играч напредовао.

Класа која је одговорна за овај приказ зове се ExperienceBar. Она садржи референцу на текст за приказ нивоа и на клизач за приказ искуства. Прислушкује догађај *OnTargetShot* од менаџера пројектила, и у тренутку призивања тог догађаја позива се метода UpdateExperience(). Поред тога, прислушкује и догађај *OnLevelUp* од менаџера игре, и у тренутку призивања тог догађаја позива се метода UpdateLevel(). Она користи *DOTween* библиотеку за анимацију текста. Овако изгледа имплементација у коду:

```
namespace AP.UI
{
    public class ExperienceBar : MonoBehaviour
    {
        [SerializeField] private TextMeshProUGUI levelValue;
        [SerializeField] private Slider experienceSlider;

        [Header("Level Up Text Sequence Data")]
        [SerializeField] private Vector3 levelUpScale;
        [SerializeField] private Color levelUpColor;
        [SerializeField] private float duration;
```

```
        [SerializeField] private Ease ease;

        private Sequence m_levelUpSequence = null;

        private void Awake()
        {
            m_levelUpSequence = DOTween.Sequence();
            Tween scaleUpAndBack = levelValue.transform.DOScale(levelUpScale,
duration / 2f).SetLoops(2, LoopType.Yoyo).SetEase(ease);
            Tween changeColorAndBack = levelValue.DOColor(levelUpColor,
duration / 2f).SetLoops(2, LoopType.Yoyo).SetEase(ease);
            m_levelUpSequence.Append(scaleUpAndBack).Join(changeColorAndBack);
        }

        private void Start()
        {
            SubscribeToEvents();
            UpdateExperience();
        }

        private void SubscribeToEvents()
        {
            ProjectilesManager.Instance.OnTargetShot += _ =>
UpdateExperience();
            GameManager.Instance.OnLevelUp += UpdateLevel;
        }

        private void UpdateExperience()
        {
            experienceSlider.value =
GameManager.Instance.ExperienceProgressNormalized;
        }

        private void UpdateLevel(int level)
        {
            levelValue.text = $"LEVEL {level + 1}";
            m_levelUpSequence.Rewind();
            m_levelUpSequence.Play();
        }
    }
}
```

На десној страни панела се налазе информације о томе колико поена је играч скупио у том тренутку игре. Састоји се од текста који служи као етикета која назначује да су поени у питању, и испод тога се налази број поена у посебно форматираном облику.

Слика 2.9 - Приказ освојених поена у HUD-у

Класа која је одговорна приказ број поена на екрану се назива PointsCounter. Она садржи референцу на текст за приказ број поена. Прислушкује догађај *OnTargetShot* од менаџера пројектила, и у тренутку призивања тог догађаја позива се метода UpdateCounter(). У тој методи се ажурира број поена приказан на екрану у посебном формату. Овако изгледа имплементација у коду:

```
namespace AP.UI
{
    public class PointsCounter : MonoBehaviour
    {
        [SerializeField] private TextMeshProUGUI counter;

        private void Start() => ProjectilesManager.Instance.OnTargetShot += _
=> UpdateCounter();

        private void UpdateCounter()
        {
            counter.text =
Formatter.FormatPoints(GameManager.Instance.Points);
        }
    }
}
```

Форматирање броја поена препуштамо класи Formatter која је статичка класа са методама за форматирање текста. Унутар методе UpdateCounter() призивамо методу FormatPoints() из класе Formatter која нам врати *string* са одређеним бројем нулама испред самог броја поена. Овако изгледа имплементација у коду:

```
namespace AP.ScoreSystem.UI
{
    public static class Formatter
    {
        private const int DEFAULT_DIGITS_PREPENDED = 7;

        public static string FormatPoints(int points, int digitsPrepended =
DEFAULT_DIGITS_PREPENDED)
        {
            string format = $"{{0,{digitsPrepended}:D{digitsPrepended}}}";
            return string.Format(format, points);
        }
    }
}
```

Са леве стране броја поена налази се *WPM (words per minute)* играча у том тренутку игре. Састоји се од текста који служи као етикета која назначује да је у питању број речи по минути, и испод тога се налази број речи по минути.

Слика 2.10 - Приказ броја речи по минути у HUD-у

Класа која је одговорна за приказ број речи по минути на екрану се назива WordsPerMinuteDisplayer. Она садржи референцу на текст за приказ број речи по минути. Садржи само једну методу LateUpdate() која се позива сваки фрејм, и у њој се ажурира приказ броја речи по минути на екрану. Овако изгледа имплементација у коду:

```
using UnityEngine;
using TMPro;

namespace AP.UI
{
    public class WordsPerMinuteDisplayer : MonoBehaviour
    {
        [SerializeField] private TextMeshProUGUI wpm;

        private void LateUpdate() => wpm.text =
GameManager.Instance.WordsPerMinute.ToString();
    }
}
```

На средини панела користимо текстуални елемент како бисмо приказали текст који играч откуца. Овим се омогућава визуелна повратна информација у реалном времену.

Слика 2.11- Приказ примера откуцане речи у HUD-у

Класа која је одговорна за приказ откуцаног текста се назива TypedTextUIController. Она садржи референцу на текстуални елемент и ажурира тај текст када се призива догођај *OnWordChanged* од менаџера за куцање на коју је ова класа претплаћена. Овако изгледа имплементација у коду:

```
namespace AP.UI
{
    public class TypedTextUIController : MonoBehaviour
```

```
    {
        [SerializeField] private TextMeshProUGUI textMeshProUGUI;

        private void Start() => TypingManager.Instance.OnWordChanged +=
UpdateText;
        private void UpdateText(string text) => textMeshProUGUI.text = text;
    }
}
```

Текстуални елемент за приказ откуцане речи у хијерархији објеката садржи још један дечји елемент, који служи као тачка уметања (*insertion point*). Овај елемент визуелно истиче да је у питању динамички текст који ће се мењати у зависности од играчевог куцања, пружајући му јасну повратну информацију. Класа која је одговорна за понашање тачке уметања се назива InsertionPointBehaviour. Она садржи референцу на слику тачке уметања и кроз корутину је активира и деактивира после одређеног времена. Овако изгледа имплементација у коду:

```
namespace AP.UI
{
    public class InsertionPointBehaviour : MonoBehaviour
    {
        [SerializeField] private Image image;
        [SerializeField] private float duration;

        private void Start() => StartCoroutine(IPBlinkCoroutine());

        private IEnumerator IPBlinkCoroutine()
        {
            yield return new WaitForSeconds(duration);
            image.enabled = !image.enabled;
            StartCoroutine(IPBlinkCoroutine());
        }
    }
}
```

Са горње стране екрана и у средини постоје два елемента корисничког интерфејса. Први је у питању текст који показује колико секунди у реалном времену траје тренутна игра. Класа која је одговорна за приказ и ажурирање овог текста се назива GameDurationTimer. Ова класа на крају сваког фрејма приступа менаџеру игре да извуче број секунди које су прошле од почетка игра кроз варијаблу *GameDurationInSeconds*. Затим тај број претвара у текст посебно форматиран тако да приказује минуте и секунде, са двотачком између њих.

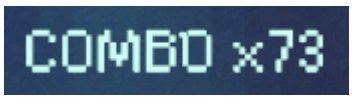
Слика 2.12- Приказ текста који показује трајање игре

```
namespace AP.UI
{
    public class GameDurationTimer : MonoBehaviour
```

```
    {
        [SerializeField] private TextMeshProUGUI timer;

        private void Start() => GameManager.Instance.OnGameOver += () =>
Destroy(gameObject);

        private void LateUpdate()
        {
            timer.text =
FormatTime((int)GameManager.Instance.GameDurationInSeconds);
        }

        public static string FormatTime(int secondsElapsed)
        {
            string minutes = string.Format("{0,2:D2}", secondsElapsed / 60);
            string seconds = string.Format("{0,2:D2}", secondsElapsed % 60);
            return $"{minutes}:{seconds}";
        }
    }
}
```

Други елемент у питању је бројач комбинације, који се појављује у виду текста "*Combo xБрој*" када је активан комбо, а не приказује ништа када нема низа уништених непријатеља. Комбо бројање се ажурира сваки пут када играч успешно уништи непријатеља или када непријатељ погоди планету.

Класа која управља овим бројачем се зове ComboCounter. Када играч повећа комбо, ова класа не само да ажурира текст, већ и покреће анимацију у којој се бројач увећава и враћа на оригиналну величину. За ову анимацију се користи библиотека *DOTween*, која омогућава да се елемент текста увећа за одређени фактор и потом врати у нормалну величину, што даје ефекат „пулсирања“ броја комбоа.



Слика 2.13 - Приказ бројача који показује тренутни број погођених непријатеља у низу

Овако изгледа имплементација у коду:

```
using UnityEngine;
using TMPro;
using DG.Tweening;
using AP.ProjectileSystem;

namespace AP.UI
{
    [RequireComponent(typeof(TextMeshProUGUI))]
    public class ComboCounter : MonoBehaviour
    {
        [Header("Scale Tween Data")]
```

```
        [SerializeField] private Vector3 scaleIncrease;
        [SerializeField] private float scaleDuration;
        [SerializeField] private Ease scaleEase;

        private TextMeshProUGUI m_counter;
        private Tween m_counterPunchTween = null;

        private void Awake()
        {
            m_counter = GetComponent<TextMeshProUGUI>();
            m_counterPunchTween = m_counter.transform.DOScale(scaleIncrease,
scaleDuration).SetEase(scaleEase).SetLoops(2,
LoopType.Yoyo).SetRelative(true);
        }

        private void Start()
        {
            SubscribeToEvents();
        }

        private void SubscribeToEvents()
        {
            PlanetBehaviour.Instance.OnHitByEnemy += _ => UpdateCounter();
            ProjectilesManager.Instance.OnTargetShot += _ => UpdateCounter();
        }

        private void UpdateCounter()
        {
            var combo = GameManager.Instance.CurrentCombo;

            if (combo == 0)
            {
                m_counter.text = string.Empty;
            }
            else
            {
                m_counter.text = $"COMBO x{combo}";
                m_counterPunchTween.Rewind();
                m_counterPunchTween.Play();
            }
        }
    }
}
```

3. ЗАКЉУЧАК

У раду је урађено као што следи:

- Успостављена је структура пројекта и кода који омогућава лаку проширивост, са јасно дефинисаним конвенцијама за организацију фолдера и писменост кода ради одрживости и модуларности.
- Осмишљен је механизам интеракције играча преко типкања, где играч куца речи како би уништио непријатељске бродове или активирао почајиваче.
- Имплементиран је систем непријатељских бродова који управља генерацијом, подешавањем и понашањем непријатељских бродова.
- Имплементиран је систем појачивача, са четири различита појачивача која обогаћују искуство играња, као и менаџери унутар тог система који контролишу генерацију носача појачивача, и активацију и деактивацију појачивача.
- Развијен је менаџер игре који управља свим релевантним информацијама у игри, ук ључујући животне поене, поене, ниво и искуства током трајања једне игре.
- Развијен је систем за управљање резултатима који омогућава чување и учитавање највиших постигнутих резултата, као и праћење напредовања играча кроз игре, чиме се обезбеђује мотивација за побољшање учинка у наредним играма.
- Креиран је графички кориснички интерфејс (HUD) који омогућава приказ животних поена, нивоа и искуства играча кроз текстуалне и визуелне елементе, уз динамичне анимације приликом пораста нивоа и низа уништених непријатеља.

4. ИНДЕКС ПОЈМОВА

U

Unity, 1, 3, 8, 27, 38

К

класа, 3, 5, 7, 14, 15, 18, 19, 23, 28, 31, 32, 33, 34
кориснички интерфејс, 2, 24, 36

О

објекат, 8, 17, 27

П

променљива, 3
променљиве, 3, 11, 15
променљиву, 4, 12

5. ЛИТЕРАТУРА

[1] Robert Nystrom, „Game Programming Patterns“, https://gameprogrammingpatterns.com/

[2] „C# Definition“, https://sr.wikipedia.org/wiki/C_Sharp, преузето: септембар 2024.

[3] „Unity Definition“, https://bs.wikipedia.org/wiki/Unity, преузето: септембар 2024.

[4] „GUI Definition“, https://www.linfo.org/gui.html, преузето: септембар 2024.

[5] „XML Definition“, https://en.wikipedia.org/wiki/XML, преузето: септембар 2024.

[6] „FileStream Definition“, https://learn.microsoft.com/en-us/dotnet/api/system.io.filestream?view=net-8.0, преузето: септембар 2024.

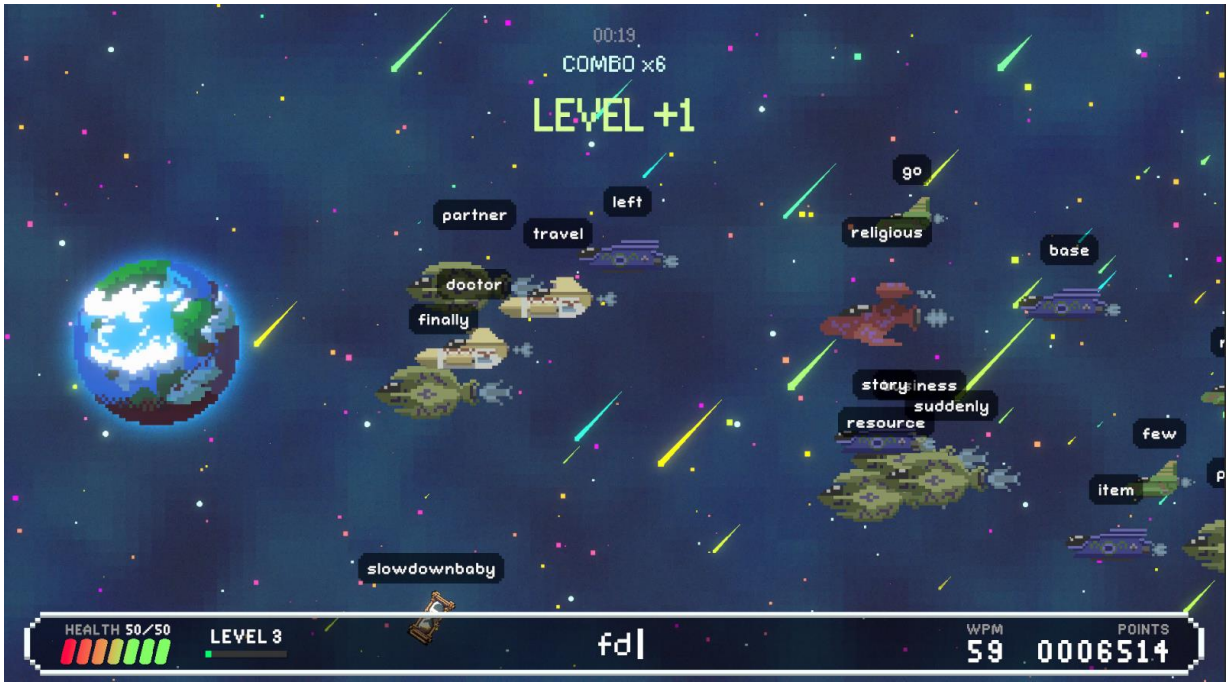
[7] „XMLSerializer Definition“, https://learn.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer?view=net-8.0, преузето: септембар 2024.

[8] „Post-processing Definition“, https://en.wikipedia.org/wiki/Video_post-processing, преузето: септембар 2024.

[9] „SOLID Design Principles Definition“, https://en.wikipedia.org/wiki/SOLID, преузето: септембар 2024.

6. ПРИЛОЗИ

6.1. УСЛИКАН ПРИКАЗ ИГРЕ



7. ИЗЈАВА О АКАДЕМСКОЈ ЧЕСТИТОСТИ

ИЗЈАВА О АКАДЕМСКОЈ ЧЕСТИТОСТИ

Студент (име, име једног родитеља и презиме): _____

Број индекса: _____

Под пуном моралном, материјалном, дисциплинском и кривичном одговорношћу изјављујем да је завршни рад, под насловом:

_____

_____

1. резултат сопственог истраживачког рада;
2. да овај рад, ни у целини, нити у деловима, нисам пријављивао/ла на другим високошколским установама;
3. да нисам повредио/ла ауторска права, нити злоупотребио/ла интелектуалну својину других лица;
4. да сам рад и мишљења других аутора које сам користио/ла у овом раду назначио/ла или цитирао/ла у складу са Упутством;
5. да су сви радови и мишљења других аутора наведени у списку литературе/референци који је саставни део овог рада, пописани у складу са Упутством;
6. да сам свестан/свесна да је плагијат коришћење туђих радова у било ком облику (као цитата, парафраза, слика, табела, дијаграма, дизајна, планова, фотографија, филма, музике, формула, веб-сајтова, компјутерских програма и сл.) без навођења аутора или представљање туђих ауторских дела као мојих, кажњиво по закону (Закон о ауторском и сродним правима), као и других закона и одговарајућих аката Академије техничко-уметничких струковних студија Београд;
7. да је електронска верзија овог рада идентична штампаном примерку овог рада и да пристајем на његово објављивање под условима прописаним актима Академије техничко-уметничких струковних студија Београд;
8. да сам свестан/свесна последица уколико се докаже да је овај рад плагијат.

У Београду, __. __. 202_. године

Својеручни потпис студента

_____