АКАДЕМИЈА ТЕХНИЧКО-УМЕТНИЧКИХ СТРУКОВНИХ СТУДИЈА БЕОГРАД

ОДСЕК ВИСОКА ШКОЛА ЕЛЕКТРОТЕХНИКЕ И РАЧУНАРСТВА

**Алекса Петровић**

**Имплементација једне 2Д игре у Unity oкружењу**

**- 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. годинe.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

др Зоран Ћировић, проф. ВИШЕР

Резиме:

Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

**Кључне речи:** : Xxxxxxxxxxxxxxxxxxxxxxxxxxxxx

ABSTRACT:

Xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

**Key words**: Xxxxxxxxxxxxxxxxxxxxxxxxxxxxx

САДРЖАЈ:

[1. УВОД 1](#_Toc177390095)

[2. Имплементација 2](#_Toc177390096)

[2.1. Општи приступ имплементације 2](#_Toc177390097)

[2.1.1. Структура фасцикли и датотека унутар пројекта 2](#_Toc177390098)

[2.1.2. Конвенције унутар кода 3](#_Toc177390099)

[2.2. Системи у игри 3](#_Toc177390100)

[2.2.1. Систем за унос речи 3](#_Toc177390101)

[2.2.2. Систем непријатељских бродова 5](#_Toc177390102)

[2.2.3. Систем појачивача 9](#_Toc177390103)

[2.2.4. Cистем пројектила 13](#_Toc177390104)

[2.2.5. Систем камера 15](#_Toc177390105)

[2.3. Xxxxxxxxxxxxxxxxxxxxxx 16](#_Toc177390106)

[3. МЕРЕЊА 17](#_Toc177390107)

[3.1.1. Zzzzzzzzzzzzzzzzzzzzzzzz 17](#_Toc177390108)

[4. ЗАКЉУЧАК 18](#_Toc177390109)

[5. ИНДЕКС ПОЈМОВА 19](#_Toc177390110)

[6. ЛИТЕРАТУРА 20](#_Toc177390111)

[7. Прилози 21](#_Toc177390112)

[8. изјава о академскoј честитости 22](#_Toc177390113)

# 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 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 у виду XML формата.

Игра је за једног играча, креирана у аркадном стилу – једноставан интерфејс и прогресивно повећање тежине. Инспирисана је игром MoonType, која је раније била доступна на Adobe Flash Player-у, али сада није у функцији због завршетка подршке за Flash Player.

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.

# Имплементација

## Општи приступ имплементације

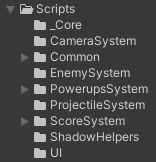
### Структура фасцикли и датотека унутар пројекта

Пројекат је осмишљен са веома организованом структуром фасцикли и датотека, ради очувања чистоће и прегледности. Унутар фасцикле *Assets* је креирана посебна фасцикла са именом игре. У њој се налазе све датотеке које не припадају додацима треће стране. На тај начин, одржава се јасна граница између увезених ресурса и сопствених ресурса, што олакшава управљање и проналажење датотека.



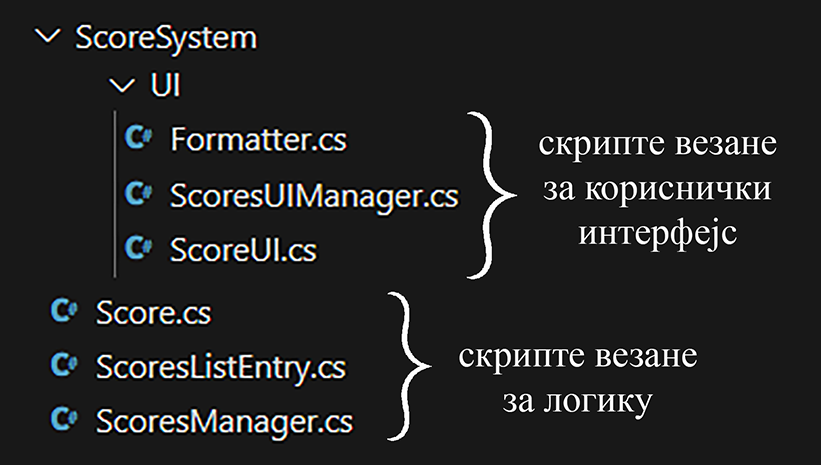
Слика 2.1 – Структура фолдера у корену пројекта

Поред тога, све скрипте су распоређене у различите фасцикле унутар главне фасцикле *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 фасцикли за скрипте

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



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

### Конвенције унутар кода

У коду, све написане скрипте налазе се унутар *AP* namespace-а, који представља иницијале мог имена. 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 фасцикли *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]](#footnote-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]](#footnote-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.

## Системи у игри

### Систем за унос речи

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

public event Action<string> OnWordChanged;

public event Action<string> OnWordSubmitted;

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.

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.

private void LateUpdate()

{

    if (CurrentWord.Length == 0) return;

    TimeSpentTyping += Time.unscaledDeltaTime;

}

Један од кључних делова инициализације система је метод 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.

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 нашем случају било који тастери за потврдђивање 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.

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;

    }

}

### Систем непријатељских бродова

#### Управљање генерацијом непријатељских бродова

Скрипта EnemyInstanceManager 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. Метод InstantiateEnemy генерише новог непријатеља на основу насумично изабране речи из система речи (WordsManager).

private IEnumerator StartInstantiatingCoroutine()

{

var randomDelay = (instantiateDelayRange / Modifiers.EnemyFrequency).GetRandomInRange();

yield return new WaitForSeconds(randomDelay );

InstantiateEnemy();

StartCoroutine(StartInstantiatingCoroutine());

}

Метод 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:

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(EnvironmentReferences.Instance.planet, enemyPreset.speed \* Modifiers.EnemySpeed);

    ShadowPathSetter.SetShadowPath(enemy.GetComponent<ShadowCaster2D>(), enemyPreset.shadowCaster2DPath);

    Enemies.Add(enemy);

}

Метод 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:

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, метод 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.

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.

#### 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 који дефинише визуелне и функционалне карактеристике појединачног непријатеља. Ова класа садржи параметре као што су брзина, скала (величина), примарни спрајт (главни изглед непријатеља) и спрајт беле боје који се користи у анимацијама припреме за уништавање. Поред тога, ту је и shadowCaster2DPath, који омогућава подешавање сенке коју непријатељ пројектује на екрану.

namespace AP.EnemySystem

{

    [CreateAssetMenu(fileName = "Enemy Level Preset Data", menuName = "")]

    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 управља свим доступним пресетима непријатеља и одлучује који ће пресет бити додељен 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 метод GetPresetDataBasedOnWord да одреди који ће пресет бити примењен:

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;

}

#### 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 "приправан за ударац" (GetReadyForShot), што активира визуелне ефекте као што је анимација преласка боје на белу, креирана уз помоћ DOTween библиотеке:

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 скрип 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:

public void StartMoving(Transform target, float speed)

{

    m\_moveTween = transform

        .DOMove(target.position, speed)

        .SetSpeedBased(true)

        .Play();

}

### 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 класу PowerupVisualsHandler, која управља визуелним приказима.

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.

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[[3]](#footnote-3) ефеката.

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, која служи као централни механизам за руковање свим активирањем и деактивирањем појачивача. PowerupsManager одржава листу свих типова појачивача у игри и води евиденцију о томе да ли је неки појачивач тренутно активан преко променљиве isAnyPowerupActive

namespace AP.PowerupsSystem

{

    public class PowerupsManager : GloballyAccessibleBase<PowerupsManager>

    {

        public bool isAnyPowerupActive = false;

        [SerializeField] private List<Powerup> powerups;

        private void Start()

        {

            GameManager.Instance.OnGameOver += DisableAllPowerups;

        }

        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();

            }

        }

    }

}

#### 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 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 покреће coroutine[[4]](#footnote-4) која је задужена за периодично генерисање носача појачивача. Интервал између појављивања носача зависи од вредности променљиве која контролише учесталост појављивања.

PowerupCarriersManager 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 метод из класе PowerupsManager, који је задужен за активацију одговарајућег појачивача.

### Cистем 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 ProjectilesManager, који управља свим пројектилима у игри, њиховим путањама и интеракцијама са метама.

Скрипта Projectile је задужена за кретање појединачног пројектила у игри. Помоћу библиотеке DOTween, пројектил се креће дуж путање коју одређује ProjectilesManager, 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 ProjectilesManager, 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.

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();

    }

}

Скрипта ProjectilesManager 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.

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 EnemyInstanceManager, да провере да ли је уништени објекат непријатељски брод.

private void ProjectileHitTarget(Transform target, Projectile projectile)

{

    OnTargetShot?.Invoke(target);

    Destroy(projectile.gameObject);

}

У ProjectilesManager постоји методa RandomPointOnPlanet, којa 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 нашој 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.

### Систем камера

Систем камерa у нашем пројекту 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.

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();

        }

    }

}

### Менаџер игре

Менаџер игре или GameManager представља централни елемент игре, задужен за праћење свих релевантних података током сесије игре. Главни задатак GameManager-а 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, GameManager представља централни „мозак“ игре који осигурава да се све компоненте синхронизују и функционишу у складу са тренутним стањем игре.

Кључни подаци које GameManager 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.

GameManager 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, GameManager 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.

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 GameManager претплаћен јесте оштећење планете од стране непријатељског брода. У случају да непријатељски брод удари у планету, GameManager 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.

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 GameManager-а са релевантним догађајима у игри.

private void SubscribeToEvents()

{

    EnemyInstanceManager.Instance.OnEnemyDestroyed += EnemyDestroyed;

    PlanetBehaviour.Instance.OnHitByEnemy += PlanetHit;

}

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

GameManager 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.

### Модификатори

У игри цмо имплементирали концепт модификатора који омогућавају играчима да прилагоде неке параметре игре, а у зависности од те прилагодбе, добијају или губе множитељ поена. 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 статичне 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.

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.

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;

    }

}

### 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, GameManager шаље обавештење ScoresManager-у, 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.

private void GameLost()

{

    OnGameOver?.Invoke();

    Score score = new Score(Points, WordsPerMinute, Modifiers.PointsModifier);

    ScoresManager.Instance.AddScore(score);

}

Када ScoresManager 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.

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, ScoresManager иницира процес чувања резултата у XML[[5]](#footnote-5) датотеку. Овај корак је кључан, јер осигурава да сви подаци буду сачувани и доступни за будуће коришћење. Користећи XmlSerializer и FileStream, 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.

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, ScoresManager 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.

private void LoadScores()

{

    Scores = new List<Score>();

    if (!File.Exists(Path.Combine(Application.persistentDataPath, FILE\_NAME)))

    {

        return;

    }

    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 нашој 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.

Табела 2.1 – Ggggggggggggggggggggggggggggg gggggggggggggggggg gggggggggggggggggg.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| xx | XXXXXXXXXXXXXX | Xxxxx | Xxxx | Xxxxx | Xxxx |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx ggggggggg xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

## Xxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx fffffffffffffff xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

# МЕРЕЊА

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

### Zzzzzzzzzzzzzzzzzzzzzzzz

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

# ЗАКЉУЧАК

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

# ИНДЕКС ПОЈМОВА

# ЛИТЕРАТУРА

[1] Б. Раковић, *Електроника – Линеарна интегрисана кола*, Грађевинска књига, Београд, 1983.

[2]D. Davis, E. Patronis Jr, *Sound system Engineering,*pp. 202-212, Focal Press, 2006.

[3] S.H. Linkwits, "Active Crossover Networks for Noncoincident Drivers", *J. Audio Eng. Soc.*, vol. 24, no. 1, pp. 2-8, 1976.

[4]J.J. Lopez, B. Pueo and M. Cobos, “Conventional and distributed mode loudspeaker arrays for the application of wave-field synthesis to video conference” *124th Conv. Audio Eng. Soc.*,Amsterdam, May 2008.

[5] J. Murray, „A Primer On Ethernet Cabling For Digital Audio” <http://www.prosoundweb.com/article/a_primer_on_ethernet_cabling_for_digital_audio/>, преузето: април 2016.

# Прилози

# изјава о академскoј 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 овaj 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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Pascal case – пракса писања вишеречних спојева заједно, са великим словима на почетку

   речи (нпр. MyPascalCaseVariable) [↑](#footnote-ref-1)
2. Constant case – пракса писања вишеречних спојева заједно, са великим словима и са доњом цртом између речи (нпр. MY\_CONSTANT\_CASE\_VARIABLE) [↑](#footnote-ref-2)
3. Post-processing - tehnike koje se primenjuju na slike ili video zapise nakon što je osnovna obrada završena. [↑](#footnote-ref-3)
4. Coroutine - посебан начин за извођење асинхроних операција, уместо да се извршавају све у једном тренутку [↑](#footnote-ref-4)
5. XML - текстуални формат за представљање структурираних података. [↑](#footnote-ref-5)