Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «Алтайский государственный университет»

Школа развития цифровых компетенций «Digital Up» (цифровая кафедра)

**ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ**

**на итоговый проект «Разработка мобильной игры в жанре «Tower Defense» «Грань миров: Оборона»» по ДПП ПП «Школа развития цифровых компетенций «Digital Up» (Цифровая кафедра АлтГУ)»**

|  |  |  |  |
| --- | --- | --- | --- |
| п/п  № | Задание | Исполнитель | Рабочий график (план) выполнения |
| 1 | Изучить задание, принципы создания и работы боевой системы, а также необходимых навыков для ее разработки. | Воронова Т.А. Попова Е.Е.  Фейгина А.А. | 01.05.2025 – 11.05.2025 |
| 2 | Разработать программу в Unity, состоящую из нескольких уровней, на каждом из которых расположить арену с уникальными характеристиками, созданием противостояния между игроком и врагами, демонстрацией (заданием) определенных параметров, а именно, строительства башен, борьбы с противниками. | 12.05.2025 – 12.06.2025 |
| 3 | Сформировать выводы по результатам выполненной работы. | 12.06.2025 – 17.06.2025 |
| 4 | Подготовить отчет и видеопрезентацию о работе. | 17.06.2025 – 20.06.2025 |

Руководитель проекта   
преподаватель СПО Осыкин Д. А.

\_\_\_\_\_\_\_\_\_\_\_\_\_ /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / «\_\_\_ » \_\_\_\_\_\_\_\_\_ 2025 г.

(подпись)

СОГЛАСОВАНО:

Руководитель ДПП ПП

канд. физ.-мат. наук, доцент Козлов Д. Ю.

\_\_\_\_\_\_\_\_\_\_\_\_\_ /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_2025 г.

(подпись)

Министерство науки и высшего образования Российской Федерации

ФГБОУ ВО «Алтайский государственный университет»

Школа развития цифровых компетенций «Digital Up» (цифровая кафедра)

Отчет о выполнении группового задания по производственной практике по ДПП ПП

«Основы Gamedev и VR-разработки»

**«Разработка мобильной игры в жанре «Tower Defense» «Грань миров: Оборона»»**

Исполнители:

Воронова Т. А.

Попова Е. Е.

Фейгина А. А.

«\_\_\_\_» \_\_\_\_\_\_\_\_ 2025 г.

Руководитель проекта

преподаватель СПО Осыкин Д. А.

«\_\_\_\_» \_\_\_\_\_\_\_\_ 2025 г.

г. Барнаул, 2025

# Объект, предмет и цель проекта

Объектом является процесс разработки игры, ее проектирование, программирование, графическое оформление, тестирование.

Предметом выступают элементы разработки (механики, графика и программирование алгоритмов).

Цель проекта – разработка мобильной игры в жанре «Tower Defense» под названием «Грань миров: Оборона».

Игра должна предоставить игрокам возможность эффективно защищать поселения от агрессивных сил, исходящих из портала в теневом измерении, используя стратегии строительства и улучшения башен, а также управления ресурсами.

# Задачи проекта

# Для достижения поставленной цели требуется решить ряд задач:

# Изучить основные принципы разработки игр в жанре Tower Defense, включая механики строительства и улучшения башен, а также управление ресурсами.

# Реализовать основные игровые механики в Unity:

# Создать систему строительства башен с возможностью размещения на карте.

# Разработать механику улучшения башен, позволяющую увеличивать их характеристики (силу удара, радиус действия и т.д.).

# Внедрить систему управления ресурсами, необходимую для возведения и модернизации защитных сооружений.

# Создать уникальные уровни для различных типов поселений, включая их визуальное оформление и особенности геймплея.

# Провести тестирование и отладку механик игры для достижения сбалансированного игрового процесса.

# Сформировать выводы по результатам проведенной работы.

# Актуальность и востребованность проекта

Актуальность данного проекта высока благодаря популярности жанра Tower Defense, который привлекает широкий круг игроков благодаря сочетанию стратегического мышления и динамичного геймплея. Концепция защиты территорий от врагов, использующая магию, позволяет создать оригинальный игровой опыт, что отвечает современным трендам на рынке мобильных игр. Успешная реализация проекта может открыть возможности для будущих расширений, таких как добавление новых типов башен, врагов и уникальных уровней, а также обновлений, которые поддержат интерес игроков на длительное время.

# Общие сведения о проделанной работе

Этапы выполнения проекта:

* Проанализировать требования и разработать концепцию игры (основные механики, сюжетные элементы, взаимодействие игрока с окружением и управление ресурсами).
* Создать базовые классы и структуру данных для башен, врагов и ресурсов.
* Реализовать систему постройки и улучшения башен.
* Разработать механики управления ресурсами и экономической стратегии.
* Создать интерфейс для отображения состояния игры, ресурсов и построенных башен.
* Провести тестирование уровней на баланс и сложность.
* Сформировать выводы по работе.

Для разработки данной мобильной игры в жанре «Tower Defense» выбрана платформа Unity с использованием языка программирования C#. Это позволяет удобно управлять игровыми объектами, разрабатывать визуальные эффекты и интегрировать необходимые игровые механики. Используются встроенные возможности Unity, а также стандартные библиотеки C# для работы с данными, коллекциями и жестами на мобильных устройствах.

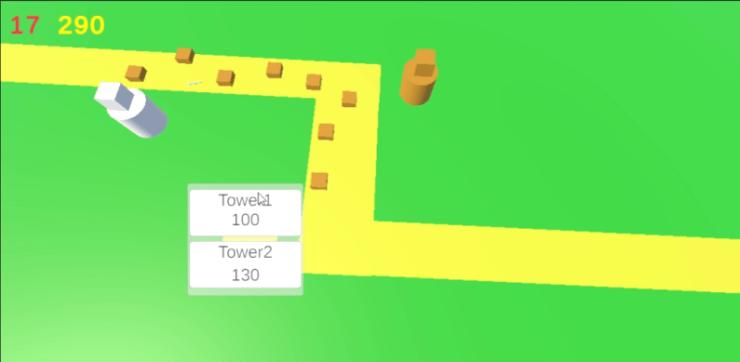
Создание базовых игровых объектов включает в себя разработку классов для башен, врагов и ресурсов. Каждая башня имеет такие свойства, как сила атаки, радиус действия, скорость стрельбы и стоимость. Эти характеристики позволяют эффективно противостоять нападающим врагам. Также предусмотрен класс для управления ресурсами, который отслеживает доступные игроку средства для постройки и улучшения башен.

Механика построения башен реализуется путем выделения специальных зон на карте, где игрок может размещать защитные сооружения. Игроку предоставляется возможность улучшать башни, увеличивая их боевые характеристики, что ключевым образом влияет на исход сражений. Параллельно с этим разработана система управления ресурсами, где игрок должен эффективно планировать затраты, чтобы успешно защитить свое поселение от волн врагов.

Игра состоит из 7-ми уровней различной сложности: начальный, промежуточные и конечный. Начальный уровень представляет собой простой этап, где игрок знакомится с основами и возможностями игры. В промежуточные уровни постепенно вводятся новые враги и тактики, требующие более глубокого стратегического подхода, повышается сложность прохождения. На конечном уровне игрок сталкивается с наиболее сильными противниками, которые могут атаковать с нескольких направлений одновременно, что создает высокий уровень вызова.

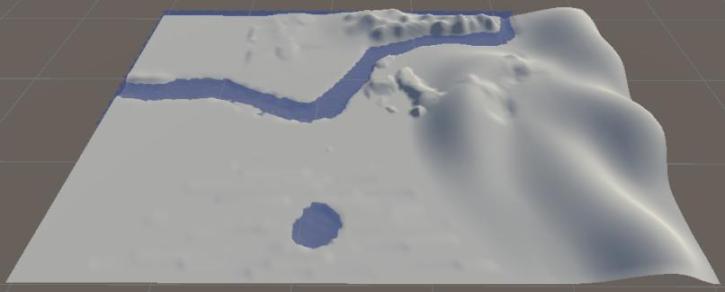
Кроме того, арены имеют уникальный дизайн, а уровень сложности варьируется в зависимости от типа поселения (леса, горы, пустыни). Это обеспечивает разнообразие в игровом процессе и позволяет игрокам адаптировать свою стратегию в зависимости от обстановки.

Для начала был создан прототип игры, основные элементы которого представлены ниже.



Наброски ландшафтов различных уровней:

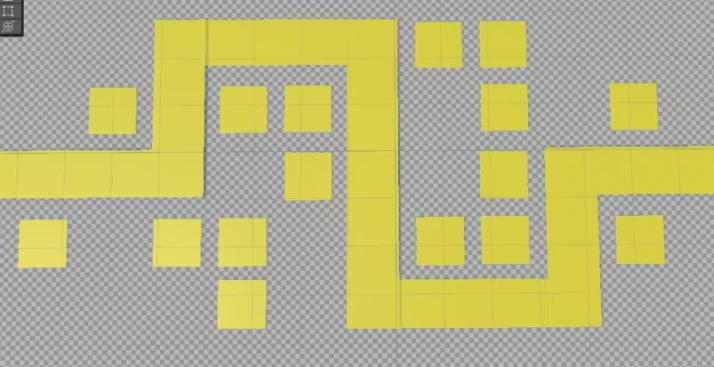






Построение уровней, включает создание дорожек, создание дизайна и его покраску. На рисунках представлены этапы разработки некоторых уровней.

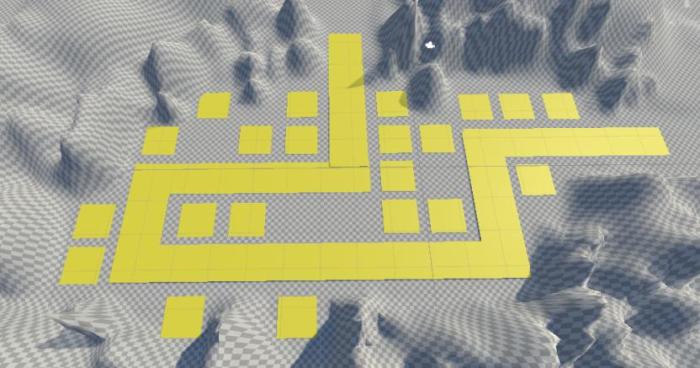
Построение второго уровня показано на рисунках ниже:





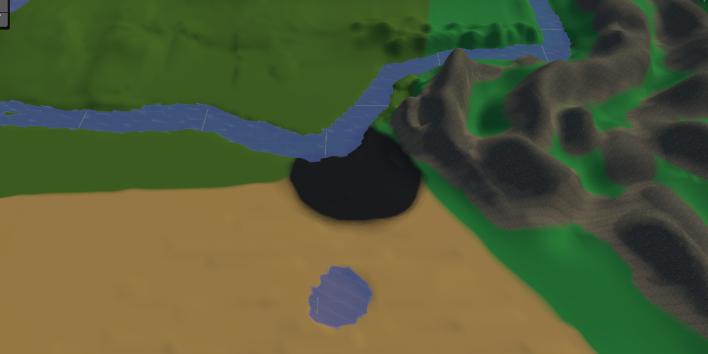


Построение уровня 3 отображено на следующих рисунках:

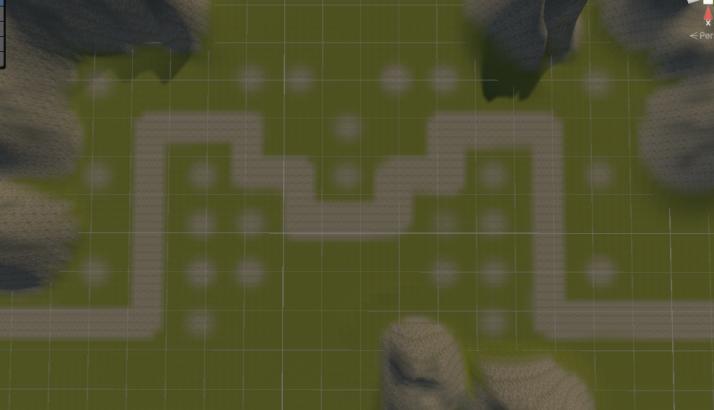




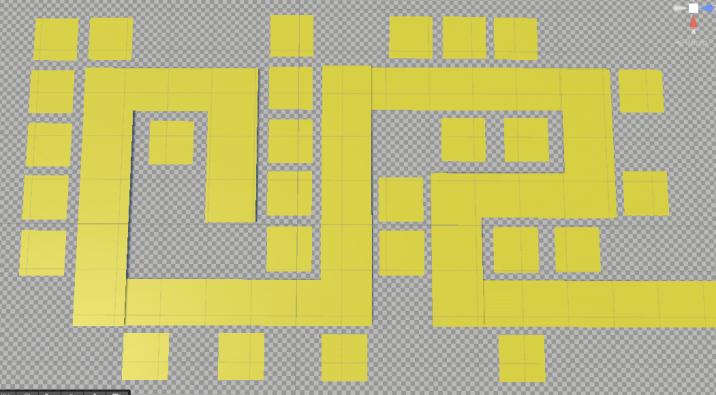
Набросок ландшафта главного меню:

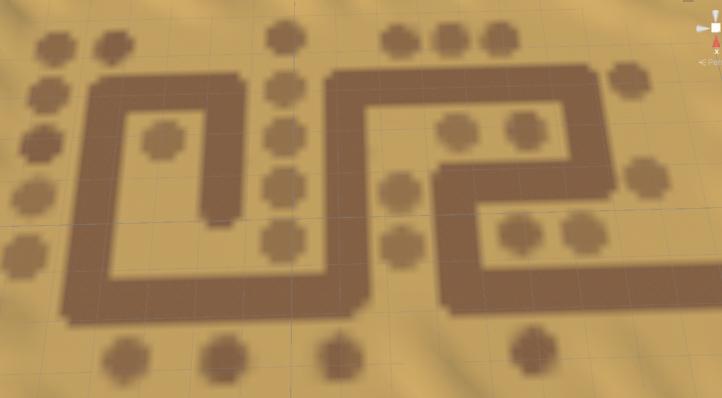


Далее покраска 4-го уровня:



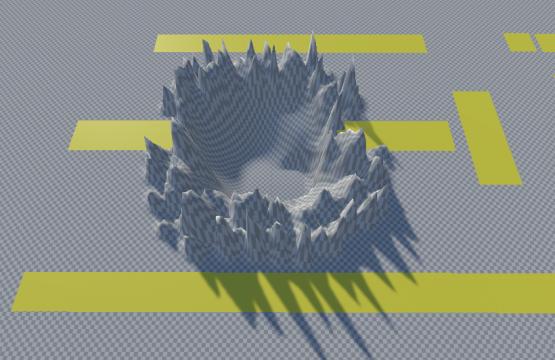
Построение объектов, дорожек и дизайна уровня 5:

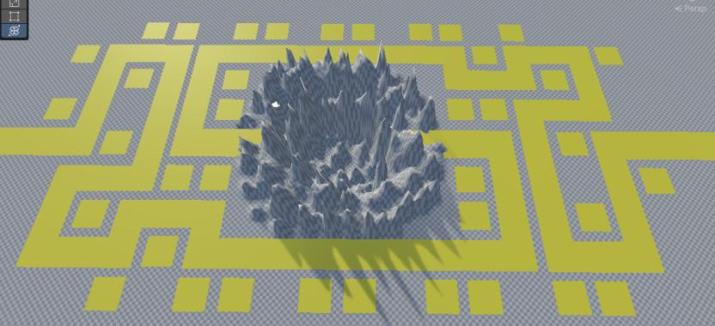


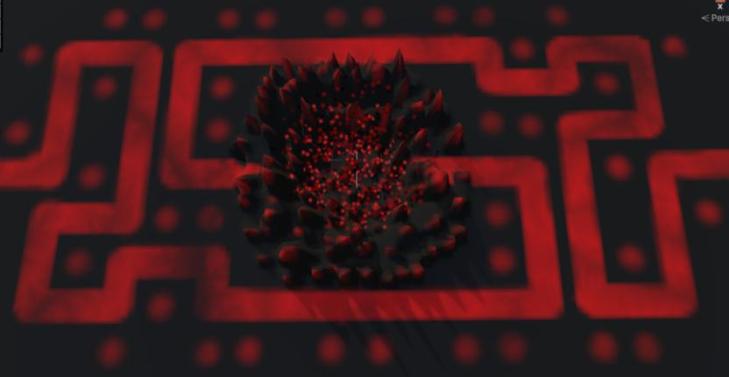




Финальный уровень (включает также построение портала):







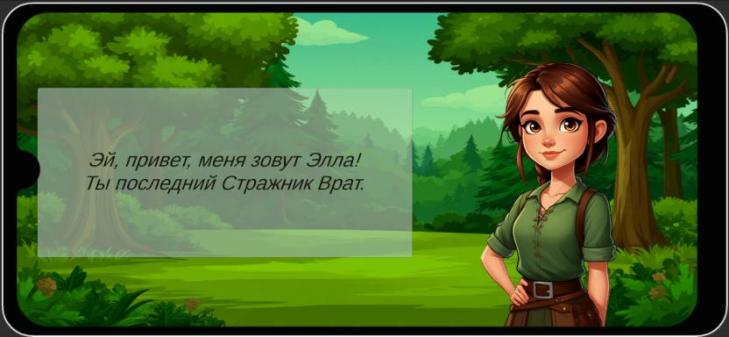
# Результаты проекта

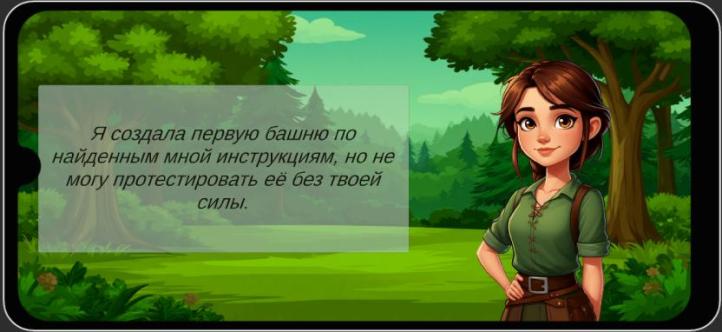
В результате разработки игры «Грань миров: Оборона» была создана комплексная система защиты, включающая в себя защиту от мобов. Игроки могут построить разнообразные башни и улучшать их для успешной защиты своих поселений.

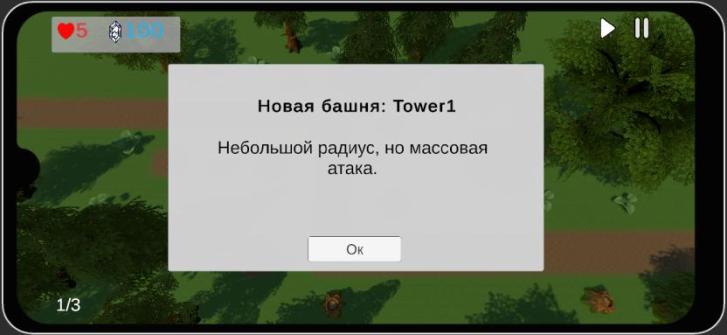
В начале игры открывается меню, где расположена карта с уровнями. По мере прохождения игры, уровни постепенно открываются.



На рисунках ниже показано начало игры, а именно часть вступления и первый уровень. Также, при необходимости, можно остановить игру, нажав на значок паузы в правом верхнем углу. Во время паузы, башни также можно строить.



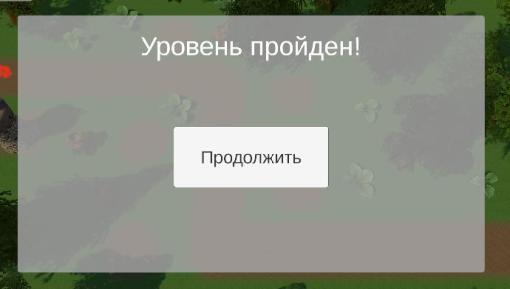




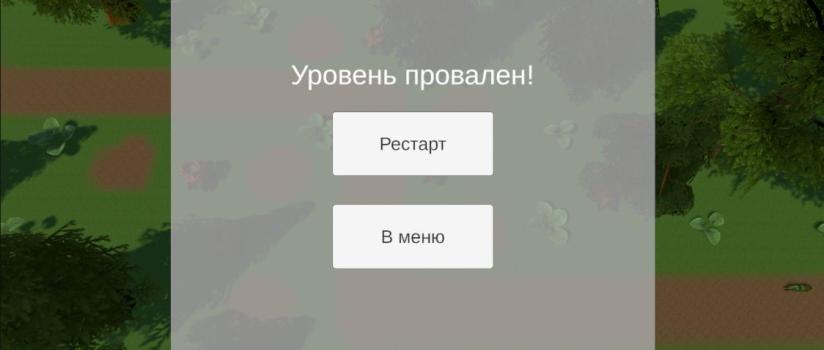




При удачном прохождении уровня, игрок получает соответствующее сообщение (показано на рисунке).

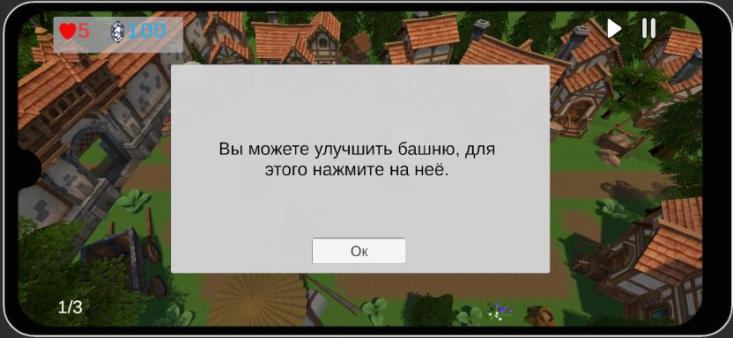


Если уровень не удалось пройти, предоставляется возможность выбора, при котором пользователь может пройти уровень повторно или выйти в главное меню.



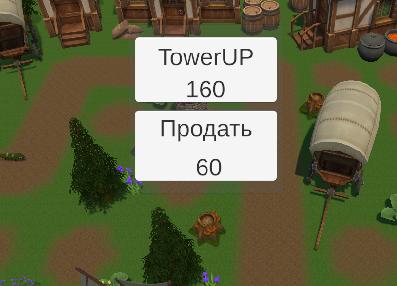
Переход на следующий уровень и его локация представлены на рисунках ниже



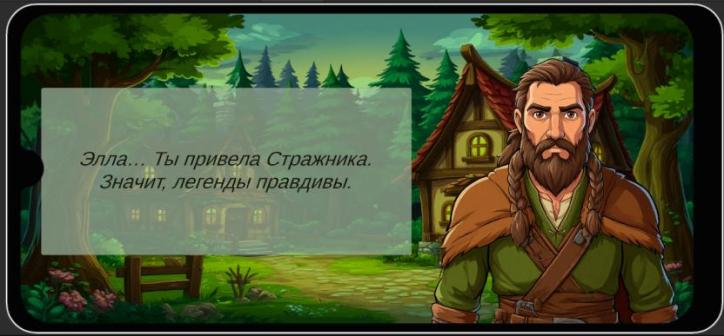




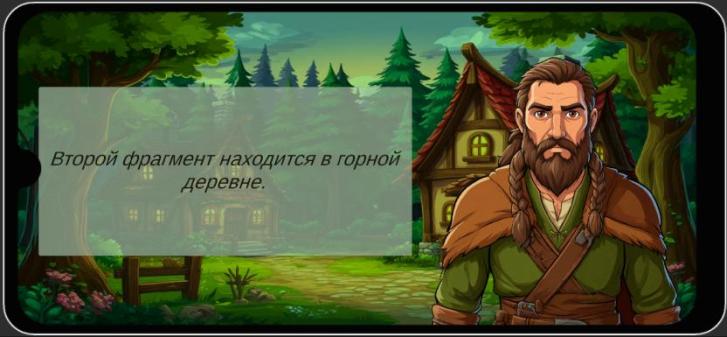
На данном уровне возможно улучшить башню за кристаллы, которые можно получить при удачной борьбе с мобами

После успешного прохождения каждого уровня следует диалог, для посвящения игрока в детали дальнейшего уровня







Начиная с 3-го уровня появляется возможность добавить вторую башню для более эффективной борьбы, Такая возможность, а также локация показаны на рисунках ниже



При построении можно выбрать вторую башню, а также улучшить или продать первую.

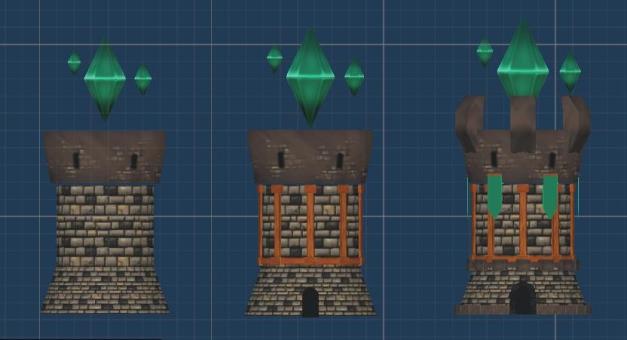
На следующих рисунках представлены часть диалога и локация 4 уровня.





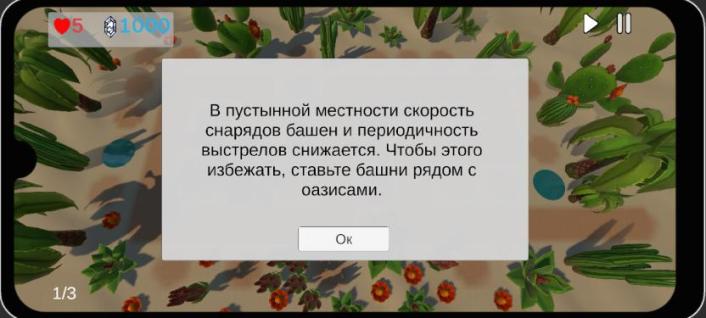
Улучшение башни отражается не только в усилении ее мощности, но и изменения ее внешнего вида, на рисунках отображены данные изменения, а также представлены различия при их разработке





На 5 и 6 уровнях, действия разворачиваются в пустыне, из-за чего возникает проблема с перегревом башен (вследствие чего снижается скорость и частота стрельбы), поэтому, для более эффективной работы их нужно располагать вблизи источников воды, о чем сообщают игроку. Предупреждение и локации показаны на рисунках.



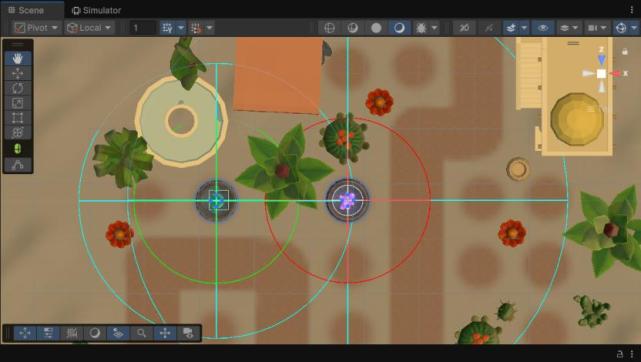




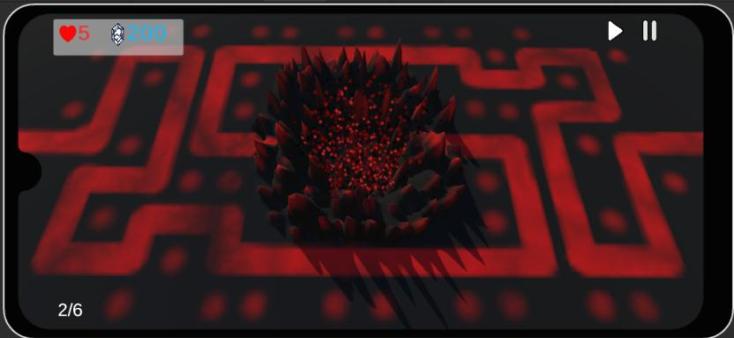




При разработке игры можно увидеть, как учитывалась особенность расположения башен от источников воды.

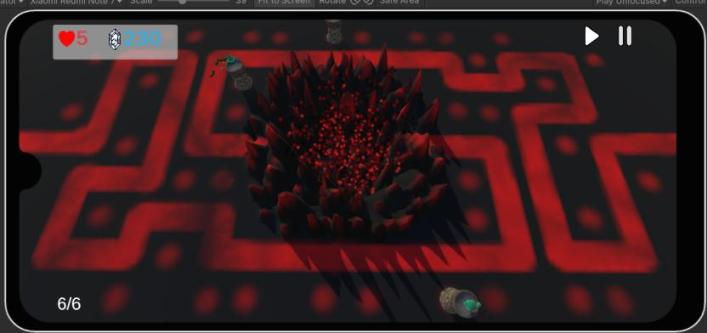


На 7-ом уровне является завершающим и более сложным по сравнению с предыдущими. Можно увидеть, что количество мобов и путей их атаки увеличилось.





Также возможно добавление башен.



При успешном прохождении последнего уровня игрока уведомляют об успешном завершении миссии.





На основе тестирования игровых механик можно сделать выводы о том, что игроки могут разрабатывать свои стратегии защиты, принимая оптимальные решения о постройке и улучшении башен. Разнообразие типов врагов и уровней сложности создаёт уникальные игровые ситуации, что делает каждую партию неповторимой. Элементы магии и технологий, воплощённые в игре, добавляют дополнительный интерес и глубину, позволяя игрокам взаимодействовать с окружающей средой.

В будущем концепция может быть доработана для внедрения новых типов врагов и башен, а также расширения игровых механик, что сделает игру ещё более увлекательной. Можно также рассмотреть возможность добавления многопользовательского режима, что позволит игрокам взаимодействовать друг с другом в реальном времени и усилит элементы стратегии.

**Приложение 1.**

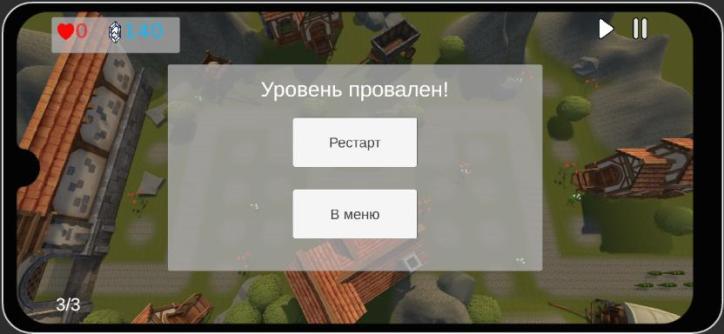
Скриншоты

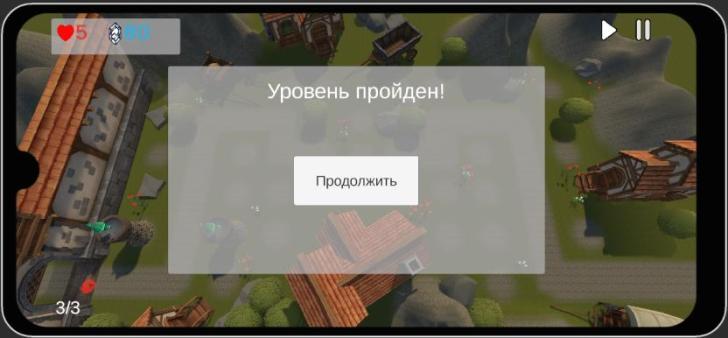
Рисунки для более четкого представления и понимания игры.



























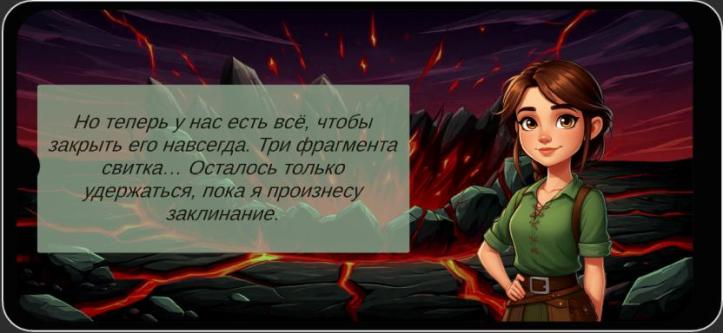


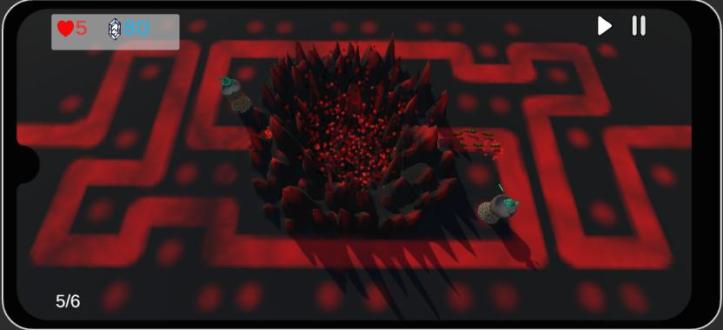


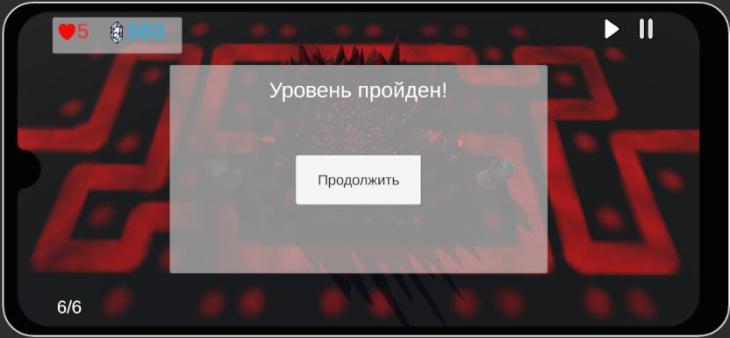














**Приложение 2.**Программный код

**Bullet.cs**

using UnityEngine;

[RequireComponent(typeof(AudioSource))]

public class Bullet : MonoBehaviour

{

public float speed;

public int damage;

public LayerMask whatIsSolid;

[Header("Effects")]

public ParticleSystem hitEffect;

public AudioClip hitSound;

[HideInInspector] public MovementMobs target;

[HideInInspector] public Tower tower;

private AudioSource audioSource;

void Start()

{

audioSource = GetComponent<AudioSource>();

if (audioSource == null)

{

audioSource = gameObject.AddComponent<AudioSource>();

}

audioSource.playOnAwake = false;

}

void Update()

{

if (target == null)

{

Destroy(gameObject);

return;

}

Vector3 dir = target.transform.position - transform.position;

transform.rotation = Quaternion.LookRotation(dir) \* Quaternion.Euler(90, 0, 0);

transform.position = Vector3.MoveTowards(transform.position, target.transform.position, speed \* Time.deltaTime);

if (Physics.Raycast(transform.position, dir.normalized, out RaycastHit hitInfo, 2f, whatIsSolid))

{

if (hitInfo.collider.CompareTag("Enemy") && hitInfo.collider.TryGetComponent<MovementMobs>(out var enemy))

{

PlayHitEffects(hitInfo.point);

enemy.TakeDamage(damage);

Destroy(gameObject);

}

}

}

private void PlayHitEffects(Vector3 hitPosition)

{

if (hitSound != null)

{

audioSource.PlayOneShot(hitSound);

}

if (hitEffect != null)

{

ParticleSystem effect = Instantiate(hitEffect, hitPosition, Quaternion.identity);

effect.Play();

Destroy(effect.gameObject, effect.main.duration);

}

}

}

**DialogueManager.cs**

using UnityEngine;

using UnityEngine.UI;

using TMPro;

using UnityEngine.InputSystem;

using UnityEngine.SceneManagement;

[System.Serializable]

public class DialogueLine

{

public enum SlideType { TextAndCharacter, ImageOnly }

public SlideType slideType = SlideType.TextAndCharacter;

[TextArea(3, 5)]

public string text;

public Sprite characterSprite;

public Sprite fullscreenImage;

public AudioClip soundEffect;

}

public class DialogueManager : MonoBehaviour

{

public DialogueLine[] dialogueLines;

public GameObject textPanel;

public TMP\_Text dialogueText;

public Image characterImage;

public GameObject tapPrompt;

public string nextSceneName = "level\_1";

public AudioSource audioSource;

public AudioClip defaultSlideSound;

private int currentLine = 0;

private float timeSinceLastTap = 0f;

private bool promptActive = false;

private PlayerInput playerInput;

private InputAction touchPressAction;

public Image fullscreenImage;

public enum DialogueType { Intro, Outro }

public DialogueType dialogueType = DialogueType.Intro;

public int associatedLevel = 1;

void Start()

{

playerInput = GetComponent<PlayerInput>();

if (playerInput == null)

{

playerInput = gameObject.AddComponent<PlayerInput>();

}

touchPressAction = playerInput.actions.FindAction("Fire", true);

if (touchPressAction == null)

{

Debug.LogError("Не найдено действие Fire в Input System");

}

if (audioSource == null)

{

audioSource = gameObject.GetComponent<AudioSource>();

if (audioSource == null)

{

audioSource = gameObject.AddComponent<AudioSource>();

audioSource.playOnAwake = false;

}

}

if (dialogueText.GetComponent<CanvasGroup>() == null)

{

dialogueText.gameObject.AddComponent<CanvasGroup>();

}

ShowLine(0);

if (tapPrompt != null) tapPrompt.SetActive(false);

}

void Update()

{

timeSinceLastTap += Time.deltaTime;

if (timeSinceLastTap > 5f && !promptActive)

{

ShowTapPrompt();

}

if (touchPressAction != null && touchPressAction.WasPressedThisFrame())

{

HandleTap();

}

}

void ShowLine(int index)

{

if (dialogueLines == null || index >= dialogueLines.Length || index < 0)

{

Debug.LogError("Invalid dialogue line configuration");

return;

}

var line = dialogueLines[index];

if (line == null)

{

Debug.LogError($"Dialogue line at index {index} is null");

return;

}

if (textPanel != null) textPanel.SetActive(false);

if (characterImage != null) characterImage.gameObject.SetActive(false);

if (fullscreenImage != null) fullscreenImage.gameObject.SetActive(false);

switch (line.slideType)

{

case DialogueLine.SlideType.TextAndCharacter:

if (textPanel != null) textPanel.SetActive(true);

if (dialogueText != null)

{

dialogueText.text = line.text ?? "";

dialogueText.gameObject.SetActive(true);

FadeInElement(dialogueText);

}

if (characterImage != null)

{

characterImage.sprite = line.characterSprite;

characterImage.gameObject.SetActive(line.characterSprite != null);

if (line.characterSprite != null) FadeInElement(characterImage);

}

break;

case DialogueLine.SlideType.ImageOnly:

if (fullscreenImage != null)

{

fullscreenImage.sprite = line.fullscreenImage;

fullscreenImage.gameObject.SetActive(line.fullscreenImage != null);

if (line.fullscreenImage != null) FadeInElement(fullscreenImage);

}

break;

}

if (audioSource != null && line.soundEffect != null)

{

audioSource.PlayOneShot(line.soundEffect);

}

timeSinceLastTap = 0f;

if (promptActive) HideTapPrompt();

}

void FadeInElement(Graphic element)

{

CanvasGroup cg = element.GetComponent<CanvasGroup>();

if (cg == null) cg = element.gameObject.AddComponent<CanvasGroup>();

cg.alpha = 0;

LeanTween.alphaCanvas(cg, 1f, 0.3f);

}

void PlaySlideSound(AudioClip clip)

{

if (audioSource == null) return;

AudioClip soundToPlay = clip != null ? clip : defaultSlideSound;

if (soundToPlay != null)

{

audioSource.PlayOneShot(soundToPlay);

}

}

void ShowTapPrompt()

{

if (tapPrompt == null) return;

promptActive = true;

tapPrompt.SetActive(true);

CanvasGroup cg = tapPrompt.GetComponent<CanvasGroup>();

if (cg == null) cg = tapPrompt.AddComponent<CanvasGroup>();

LeanTween.alphaCanvas(cg, 0.3f, 0.5f)

.setLoopPingPong();

}

void HideTapPrompt()

{

if (tapPrompt == null) return;

LeanTween.cancel(tapPrompt);

promptActive = false;

tapPrompt.SetActive(false);

}

void HandleTap()

{

timeSinceLastTap = 0f;

if (tapPrompt != null)

tapPrompt.SetActive(false);

if (currentLine < dialogueLines.Length - 1)

{

currentLine++;

ShowLine(currentLine);

}

else

{

string sceneName = SceneManager.GetActiveScene().name;

if (sceneName.EndsWith("\_2"))

{

LevelProgressManager.Instance.MarkLevelCompleted();

int currentLevel = LevelProgressManager.Instance.GetCurrentLevelNumber();

Debug.Log($"Completed level {currentLevel}. Loading MainMenu");

}

if (!string.IsNullOrEmpty(nextSceneName))

{

SceneManager.LoadScene(nextSceneName);

}

else

{

SceneManager.LoadScene("MainMenu");

Debug.LogWarning("Next scene name not specified! Loading MainMenu by default");

}

}

}

}

**GameManager.cs**

using UnityEngine;

using UnityEngine.SceneManagement;

using TMPro;

using System.Collections.Generic;

using System;

using UnityEngine.UI;

using System.Collections;

public class GameManager : MonoBehaviour

{

public enum LevelType { Normal, Desert }

public LevelType currentLevelType = LevelType.Normal;

public static GameManager Instance { get; private set; }

public static event Action<bool> OnGameEnded;

public int playerHealth;

public TMP\_Text healthText;

public int money;

public TMP\_Text moneyText;

[Header("UI Elements")]

public GameObject levelFailedPanel;

public GameObject levelCompletePanel;

[Header("Scene Names")]

public string mainMenuScene = "MainMenu";

public string dialogueScene = "DialogueScene";

[System.NonSerialized]

public List<MovementMobs> EnemyList = new List<MovementMobs>();

private WaveSpawner waveSpawner;

private bool isQuitting = false;

private bool gameEnded = false;

[Header("Input Blocking")]

public static bool IsInputBlocked { get; private set; }

public enum GameSpeed { Paused, Normal, Fast, VeryFast }

private GameSpeed currentGameSpeed = GameSpeed.Normal;

private GameSpeed lastSpeedBeforePause = GameSpeed.Normal;

[Header("UI Elements")]

public Button pauseButton;

public Image pauseButtonImage;

public Sprite pauseIcon;

public Sprite playIcon;

public Button speedButton;

public Image speedButtonImage;

public Sprite normalSpeedIcon;

public Sprite fastSpeedIcon;

public Sprite veryFastSpeedIcon;

private bool isPaused = false;

private bool isProcessingClick = false;

private void OnSceneLoaded(Scene scene, LoadSceneMode mode)

{

IsInputBlocked = false;

}

void OnEnable()

{

SceneManager.sceneLoaded += OnSceneLoaded;

}

void OnDisable()

{

SceneManager.sceneLoaded -= OnSceneLoaded;

}

public void TogglePause()

{

if (gameEnded || isProcessingClick) return;

isProcessingClick = true;

if (gameEnded) return;

isPaused = !isPaused;

if (isPaused)

{

lastSpeedBeforePause = currentGameSpeed;

Time.timeScale = 0f;

currentGameSpeed = GameSpeed.Paused;

pauseButtonImage.sprite = playIcon;

}

else

{

Time.timeScale = lastSpeedBeforePause switch

{

GameSpeed.Normal => 1f,

GameSpeed.Fast => 1.5f,

GameSpeed.VeryFast => 2f,

\_ => 1f

};

currentGameSpeed = lastSpeedBeforePause;

pauseButtonImage.sprite = pauseIcon;

}

UpdateSpeedUI();

StartCoroutine(ResetClickFlag());

}

public void CycleSpeed()

{

if (isProcessingClick) return;

isProcessingClick = true;

if (currentGameSpeed == GameSpeed.Paused)

{

SetGameSpeed(GameSpeed.Normal, true);

return;

}

GameSpeed nextSpeed = currentGameSpeed switch

{

GameSpeed.Normal => GameSpeed.Fast,

GameSpeed.Fast => GameSpeed.VeryFast,

GameSpeed.VeryFast => GameSpeed.Normal,

\_ => GameSpeed.Normal

};

SetGameSpeed(nextSpeed, true);

StartCoroutine(ResetClickFlag());

}

private IEnumerator ResetClickFlag()

{

yield return new WaitForEndOfFrame();

isProcessingClick = false;

}

private void SetGameSpeed(GameSpeed speed, bool updateUI = false)

{

if (gameEnded || isPaused) return;

currentGameSpeed = speed;

Time.timeScale = speed switch

{

GameSpeed.Normal => 1f,

GameSpeed.Fast => 1.5f,

GameSpeed.VeryFast => 2f,

\_ => 1f

};

if (updateUI)

{

UpdateSpeedUI();

}

}

private void UpdateSpeedUI()

{

if (pauseButtonImage != null)

{

pauseButtonImage.sprite = currentGameSpeed == GameSpeed.Paused ? playIcon : pauseIcon;

}

if (speedButtonImage != null)

{

speedButtonImage.sprite = currentGameSpeed switch

{

GameSpeed.Normal => normalSpeedIcon,

GameSpeed.Fast => fastSpeedIcon,

GameSpeed.VeryFast => veryFastSpeedIcon,

\_ => normalSpeedIcon

};

var color = speedButtonImage.color;

color.a = currentGameSpeed == GameSpeed.Paused ? 0.5f : 1f;

speedButtonImage.color = color;

}

}

public static void SetInputBlock(bool blocked)

{

IsInputBlocked = blocked;

}

void Awake()

{

if (Instance == null)

{

Instance = this;

}

else

{

Destroy(gameObject);

return;

}

InitializeUI();

}

void Start()

{

if (pauseButton == null || speedButton == null)

{

Debug.LogError("Кнопки не назначены в инспекторе!");

return;

}

pauseButton.onClick.AddListener(TogglePause);

speedButton.onClick.AddListener(CycleSpeed);

isPaused = false;

Time.timeScale = 1f;

currentGameSpeed = GameSpeed.Normal;

UpdateSpeedUI();

SetGameSpeed(GameSpeed.Normal, updateUI: true);

waveSpawner = FindObjectOfType<WaveSpawner>();

DetermineLevelType();

}

private void DetermineLevelType()

{

currentLevelType = SceneManager.GetActiveScene().name.ToLower().Contains("desert")

? LevelType.Desert

: LevelType.Normal;

}

void InitializeUI()

{

healthText.text = playerHealth.ToString();

moneyText.text = money.ToString();

levelFailedPanel.SetActive(false);

levelCompletePanel.SetActive(false);

}

public void ChangeMoney(int count)

{

money += count;

moneyText.text = money.ToString();

}

public void ChangeHealth(int count)

{

playerHealth -= count;

healthText.text = playerHealth.ToString();

if (playerHealth <= 0)

{

LevelFailed();

}

}

public void AddEnemyOnList(MovementMobs enemy)

{

if (enemy == null) return;

if (!EnemyList.Contains(enemy))

{

EnemyList.Add(enemy);

}

}

public void SafeRemoveEnemyFromList(MovementMobs enemy)

{

if (isQuitting || enemy == null) return;

if (EnemyList.Contains(enemy))

{

EnemyList.Remove(enemy);

CheckLevelCompletion();

}

}

public void CheckLevelCompletion()

{

if (gameEnded) return;

if (waveSpawner == null) waveSpawner = FindObjectOfType<WaveSpawner>();

if (waveSpawner == null) return;

bool wavesDone = waveSpawner.AllWavesCompleted;

bool noActiveEnemies = EnemyList.Count == 0;

bool notSpawning = !waveSpawner.IsSpawning;

if (wavesDone && noActiveEnemies && notSpawning && playerHealth > 0)

{

LevelComplete();

}

}

private void LevelFailed()

{

if (!gameEnded && levelFailedPanel != null)

{

gameEnded = true;

Time.timeScale = 0f;

levelFailedPanel.SetActive(true);

SetInputBlock(true);

OnGameEnded?.Invoke(true);

}

}

private void LevelComplete()

{

if (!gameEnded && levelCompletePanel != null)

{

gameEnded = true;

Time.timeScale = 0f;

levelCompletePanel.SetActive(true);

SetInputBlock(true);

OnGameEnded?.Invoke(true);

}

}

public void RestartLevel()

{

Time.timeScale = 1f;

gameEnded = false;

SceneManager.LoadScene(SceneManager.GetActiveScene().name);

}

public void GoToMainMenu()

{

Time.timeScale = 1f;

gameEnded = false;

SceneManager.LoadScene(mainMenuScene);

}

public void ContinueToDialogue()

{

Time.timeScale = 1f;

gameEnded = false;

SceneManager.LoadScene(dialogueScene);

}

void OnApplicationQuit()

{

isQuitting = true;

}

}

**LevelManager.cs**

using UnityEngine;

using UnityEngine.UI;

using TMPro;

using UnityEngine.SceneManagement;

public class LevelManager : MonoBehaviour

{

[Header("Button Settings")]

[SerializeField] private GameObject[] levelButtons;

[SerializeField] private GameObject activeButtonPrefab;

[SerializeField] private GameObject inactiveButtonPrefab;

[Header("Settings UI")]

[SerializeField] private Canvas mainCanvas;

[SerializeField] private GameObject settingsPanel;

[SerializeField] private GameObject resetConfirmationPanel;

[SerializeField] private Button settingsButton;

[SerializeField] private Button closeSettingsButton;

[SerializeField] private Button quitButton;

[SerializeField] private Button resetProgressButton;

[SerializeField] private Button confirmResetButton;

[SerializeField] private Button cancelResetButton;

[Header("Debug")]

[SerializeField] private bool debugMode = true;

private void Start()

{

if (debugMode) Debug.Log("LevelManager initialization started");

if (!ValidateDependencies())

{

if (debugMode) Debug.LogError("Initialization failed - dependencies not met");

return;

}

SetupUIElements();

UpdateLevelButtons();

}

private void SetupUIElements()

{

settingsPanel.transform.SetAsLastSibling();

resetConfirmationPanel.transform.SetAsLastSibling();

settingsButton.onClick.AddListener(OpenSettings);

closeSettingsButton.onClick.AddListener(CloseSettings);

quitButton.onClick.AddListener(QuitGame);

resetProgressButton.onClick.AddListener(ShowResetConfirmation);

confirmResetButton.onClick.AddListener(ConfirmResetProgress);

cancelResetButton.onClick.AddListener(CancelResetProgress);

settingsPanel.SetActive(false);

resetConfirmationPanel.SetActive(false);

}

private void OpenSettings()

{

if (debugMode) Debug.Log("Opening settings panel");

settingsPanel.transform.SetAsLastSibling();

settingsPanel.SetActive(true);

SetLevelButtonsInteractable(false);

}

private void CloseSettings()

{

if (debugMode) Debug.Log("Closing settings panel");

settingsPanel.SetActive(false);

SetLevelButtonsInteractable(true);

}

private void ShowResetConfirmation()

{

if (debugMode) Debug.Log("Showing reset confirmation");

resetConfirmationPanel.transform.SetAsLastSibling();

resetConfirmationPanel.SetActive(true);

}

private void ConfirmResetProgress()

{

if (debugMode) Debug.Log("Resetting all progress");

ResetAllProgress();

resetConfirmationPanel.SetActive(false);

settingsPanel.SetActive(false);

SetLevelButtonsInteractable(true);

}

private void CancelResetProgress()

{

if (debugMode) Debug.Log("Cancel reset progress");

resetConfirmationPanel.SetActive(false);

}

private void SetLevelButtonsInteractable(bool interactable)

{

foreach (var button in levelButtons)

{

if (button != null)

{

var btnComponent = button.GetComponent<Button>();

if (btnComponent != null)

{

btnComponent.interactable = interactable;

}

}

}

}

private bool ValidateDependencies()

{

if (LevelProgressManager.Instance == null)

{

Debug.LogError("LevelProgressManager instance is null!");

return false;

}

if (levelButtons == null || levelButtons.Length == 0)

{

Debug.LogError("Level buttons array is not set or empty!");

return false;

}

if (activeButtonPrefab == null || inactiveButtonPrefab == null)

{

Debug.LogError("Button prefabs are not assigned!");

return false;

}

if (settingsPanel == null || resetConfirmationPanel == null ||

settingsButton == null || closeSettingsButton == null ||

quitButton == null || resetProgressButton == null ||

confirmResetButton == null || cancelResetButton == null)

{

Debug.LogError("Some UI elements are not assigned!");

return false;

}

if (debugMode) Debug.Log("All dependencies validated successfully");

return true;

}

public void UpdateLevelButtons()

{

if (debugMode) Debug.Log($"Updating {levelButtons.Length} level buttons");

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

{

if (levelButtons[i] == null)

{

Debug.LogWarning($"Button at index {i} is null, skipping");

continue;

}

Transform buttonParent = levelButtons[i].transform.parent;

if (buttonParent == null)

{

Debug.LogWarning($"Button at index {i} has no parent, skipping");

continue;

}

if (debugMode) Debug.Log($"Processing button {i} with parent {buttonParent.name}");

Vector3 position = levelButtons[i].transform.localPosition;

Vector3 scale = levelButtons[i].transform.localScale;

string name = levelButtons[i].name;

Destroy(levelButtons[i]);

levelButtons[i] = CreateLevelButton(i + 1, buttonParent);

levelButtons[i].transform.localPosition = position;

levelButtons[i].transform.localScale = scale;

levelButtons[i].name = name;

}

}

private GameObject CreateLevelButton(int levelNumber, Transform parent)

{

if (debugMode) Debug.Log($"Creating button for level {levelNumber}");

bool isUnlocked = LevelProgressManager.Instance.IsLevelUnlocked(levelNumber);

GameObject prefabToUse = isUnlocked ? activeButtonPrefab : inactiveButtonPrefab;

if (prefabToUse == null)

{

Debug.LogError($"Prefab for {(isUnlocked ? "active" : "inactive")} button is null!");

return null;

}

if (parent == null)

{

Debug.LogError("Parent transform is null!");

return null;

}

GameObject newButton = Instantiate(prefabToUse, parent);

if (isUnlocked)

{

SetupActiveButton(newButton, levelNumber);

}

return newButton;

}

private void SetupActiveButton(GameObject button, int levelNumber)

{

TMP\_Text textComponent = button.GetComponentInChildren<TMP\_Text>();

if (textComponent != null)

{

textComponent.text = levelNumber.ToString();

}

else

{

Debug.LogWarning("No TMP\_Text component found on active button");

}

Button buttonComponent = button.GetComponent<Button>();

if (buttonComponent != null)

{

buttonComponent.onClick.RemoveAllListeners();

buttonComponent.onClick.AddListener(() => LoadLevel(levelNumber));

}

else

{

Debug.LogWarning("No Button component found on active button");

}

}

public void LoadLevel(int levelIndex)

{

if (LevelProgressManager.Instance.IsLevelUnlocked(levelIndex))

{

if (debugMode) Debug.Log($"Loading intro dialogue for level {levelIndex}");

SceneManager.LoadScene($"IntroDialogue{levelIndex}\_1");

}

else

{

Debug.LogWarning($"Attempted to load locked level {levelIndex}");

}

}

public void ResetAllProgress()

{

LevelProgressManager.Instance.ResetProgress();

UpdateLevelButtons();

if (debugMode) Debug.Log("All progress has been reset");

foreach (var button in levelButtons)

{

if (button != null)

{

button.transform.localScale = Vector3.zero;

LeanTween.scale(button, Vector3.one, 0.5f).setEase(LeanTweenType.easeOutBack);

}

}

}

public void QuitGame()

{

if (Application.isMobilePlatform)

{

SaveBeforeQuit();

AndroidJavaObject activity = new AndroidJavaClass("com.unity3d.player.UnityPlayer").GetStatic<AndroidJavaObject>("currentActivity");

activity.Call<bool>("moveTaskToBack", true);

}

else

{

#if UNITY\_EDITOR

UnityEditor.EditorApplication.isPlaying = false;

#else

Application.Quit();

#endif

}

}

private void SaveBeforeQuit()

{

LevelProgressManager.Instance.SaveProgress();

Debug.Log("Game data saved before quit");

}

}

**LevelProgressManager.cs**

using UnityEngine;

using UnityEngine.SceneManagement;

public class LevelProgressManager : MonoBehaviour

{

public static LevelProgressManager Instance { get; private set; }

private const string UNLOCKED\_LEVEL\_KEY = "UnlockedLevel";

private int \_unlockedLevel = 1;

void Awake()

{

if (Instance == null)

{

Instance = this;

DontDestroyOnLoad(gameObject);

LoadProgress();

Debug.Log($"Progress initialized. Unlocked up to level {\_unlockedLevel}");

}

else

{

Destroy(gameObject);

}

}

public bool IsLevelUnlocked(int level) => level <= \_unlockedLevel;

public void MarkLevelCompleted()

{

string sceneName = SceneManager.GetActiveScene().name;

if (sceneName.Contains("\_2"))

{

string levelStr = sceneName.Split('\_')[0].Replace("IntroDialogue", "");

if (int.TryParse(levelStr, out int completedLevel))

{

if (completedLevel >= \_unlockedLevel)

{

\_unlockedLevel = completedLevel + 1;

SaveProgress();

Debug.Log($"Level {completedLevel} completed! Unlocked level {\_unlockedLevel}");

}

}

}

}

public int GetCurrentLevelNumber()

{

string sceneName = SceneManager.GetActiveScene().name;

if (sceneName.Contains("IntroDialogue") && sceneName.Contains("\_"))

{

string levelStr = sceneName.Split('\_')[0].Replace("IntroDialogue", "");

if (int.TryParse(levelStr, out int level))

return level;

}

return -1;

}

private void LoadProgress() => \_unlockedLevel = PlayerPrefs.GetInt(UNLOCKED\_LEVEL\_KEY, 1);

public void SaveProgress()

{

PlayerPrefs.SetInt(UNLOCKED\_LEVEL\_KEY, \_unlockedLevel);

PlayerPrefs.Save();

}

public void ResetProgress()

{

\_unlockedLevel = 1;

PlayerPrefs.DeleteKey(UNLOCKED\_LEVEL\_KEY);

PlayerPrefs.Save();

Debug.Log("Progress reset to level 1");

}

}

**LewelTowerInfo.cs**

using UnityEngine;

using TMPro;

public class LevelTowerInfo : MonoBehaviour

{

[Header("Настройки для этого уровня")]

public bool showPanel = true;

[TextArea(3, 10)]

public string infoText = "Тут описание башни...";

[Header("Ссылки (перетащите вручную)")]

public GameObject towerInfoPanel;

public TMP\_Text textUI;

void Start()

{

if (showPanel)

{

textUI.text = infoText;

towerInfoPanel.SetActive(true);

Time.timeScale = 0f;

}

}

public void ClosePanel()

{

towerInfoPanel.SetActive(false);

Time.timeScale = 1f;

}

}

**MobSpawner.cs**

using UnityEngine;

public class MobSpawner : MonoBehaviour

{

public GameObject mobPrefab;

public Transform[] spawnPoints;

public float spawnInterval = 3f;

public int maxMobs = 10;

private float timer;

private int currentMobs;

private int lastSpawnPointIndex = -1;

void Update()

{

if (currentMobs >= maxMobs) return;

timer += Time.deltaTime;

if (timer >= spawnInterval)

{

SpawnMob();

timer = 0f;

}

}

void SpawnMob()

{

if (spawnPoints.Length == 0) return;

lastSpawnPointIndex = (lastSpawnPointIndex + 1) % spawnPoints.Length;

Transform spawnPoint = spawnPoints[lastSpawnPointIndex];

Instantiate(mobPrefab, spawnPoint.position, spawnPoint.rotation);

currentMobs++;

}

}

**MovementMobs.cs**

using UnityEngine;

using System.Linq;

public class MovementMobs : MonoBehaviour

{

private Transform[][] waypointGroups;

private Transform currentWaypoint;

private int currentPointIndex = 0;

public float baseSpeed = 3f;

private float actualSpeed;

[Header("Settings")]

public float rotationSpeed = 10f;

public float arrivalDistance = 0.1f;

[Header("Combat")]

public int health = 10;

public int loot = 1;

private GameManager gameManager;

private bool isBeingDestroyed = false;

private bool killedByPlayer = false;

public void Initialize(params Transform[] waypoints)

{

if (waypoints == null || waypoints.Length == 0)

{

Debug.LogError("No valid waypoints provided!");

SafeDestroy();

return;

}

this.waypointGroups = new Transform[][] { waypoints };

currentPointIndex = 0;

actualSpeed = Random.Range(baseSpeed \* 1f, baseSpeed \* 1f);

gameManager = GameManager.Instance;

if (gameManager == null)

{

Debug.LogWarning("GameManager instance not found");

SafeDestroy();

return;

}

gameManager.AddEnemyOnList(this);

SelectNextWaypoint();

}

private void SafeDestroy()

{

if (!isBeingDestroyed)

{

isBeingDestroyed = true;

Destroy(gameObject);

}

}

void Update()

{

if (isBeingDestroyed || currentWaypoint == null)

return;

MoveToWaypoint();

CheckWaypointReached();

}

private void MoveToWaypoint()

{

transform.position = Vector3.MoveTowards(

transform.position,

currentWaypoint.position,

actualSpeed \* Time.deltaTime

);

RotateTowards(currentWaypoint.position);

}

private void RotateTowards(Vector3 targetPosition)

{

Vector3 direction = (targetPosition - transform.position).normalized;

if (direction != Vector3.zero)

{

Quaternion targetRotation = Quaternion.LookRotation(direction);

transform.rotation = Quaternion.Slerp(

transform.rotation,

targetRotation,

rotationSpeed \* Time.deltaTime

);

}

}

private void CheckWaypointReached()

{

if (Vector3.Distance(transform.position, currentWaypoint.position) <= arrivalDistance)

{

currentPointIndex++;

SelectNextWaypoint();

}

}

private void SelectNextWaypoint()

{

var validGroups = waypointGroups

.Where(g => currentPointIndex < g.Length && g[currentPointIndex] != null)

.ToArray();

if (validGroups.Length == 0)

{

ReachedEndOfPath();

return;

}

int randomGroupIndex = Random.Range(0, validGroups.Length);

currentWaypoint = validGroups[randomGroupIndex][currentPointIndex];

}

private void ReachedEndOfPath()

{

if (isBeingDestroyed) return;

if (gameManager != null)

{

gameManager.ChangeHealth(1);

}

Die();

}

public void TakeDamage(int damage)

{

if (isBeingDestroyed) return;

health -= damage;

if (health <= 0)

{

killedByPlayer = true;

Die();

}

}

private void Die()

{

if (isBeingDestroyed) return;

isBeingDestroyed = true;

if (gameManager != null)

{

if (killedByPlayer)

{

gameManager.ChangeMoney(loot);

}

gameManager.SafeRemoveEnemyFromList(this);

}

Destroy(gameObject);

}

private void OnDestroy()

{

if (!isBeingDestroyed && gameManager != null)

{

gameManager.SafeRemoveEnemyFromList(this);

}

}

}

**Tower.cs**

using UnityEngine;

using System.Collections.Generic;

using UnityEngine.UIElements;

public enum TowerType { SingleTarget, MultiTarget }

public class Tower : MonoBehaviour

{

[Header("Тип башни")]

[SerializeField] private TowerType towerType = TowerType.SingleTarget;

[SerializeField] private int maxTargets = 3;

[Header("Настройки атаки")]

[SerializeField] private float attackRadius = 15f;

[SerializeField] private float startTimeBtwAttack = 1f;

[SerializeField] private GameObject bulletPrefab;

[SerializeField] private Transform shootPoint;

[Header("Ссылки")]

private GameManager gameManager;

private MovementMobs targetEnemy;

private float timeBtwAttack;

public GameObject nextTower;

public int cost;

[Header("Desert Effects")]

[SerializeField] private float desertSpeedMultiplier = 0.25f;

[SerializeField] private float desertFireRateMultiplier = 1.35f;

[SerializeField] private LayerMask oasisLayer;

[SerializeField] private float oasisCheckRadius = 3f;

private float originalFireRate;

private float originalBulletSpeed;

private bool isInDesert = false;

private bool hasOasisNearby = false;

[Header("Sound Effects")]

[SerializeField] private AudioClip shootSound;

private AudioSource audioSource;

void Start()

{

audioSource = GetComponent<AudioSource>();

if (audioSource == null)

{

audioSource = gameObject.AddComponent<AudioSource>();

}

audioSource.playOnAwake = false;

gameManager = FindObjectOfType<GameManager>();

originalFireRate = startTimeBtwAttack;

if (bulletPrefab != null && bulletPrefab.TryGetComponent<Bullet>(out var bullet))

{

originalBulletSpeed = bullet.speed;

}

isInDesert = gameManager != null && gameManager.currentLevelType == GameManager.LevelType.Desert;

}

void Update()

{

if (Time.timeScale == 0) return;

if (targetEnemy == null)

{

MovementMobs nearest = GetNearestEnemy();

if (nearest != null)

{

targetEnemy = nearest;

}

}

if (targetEnemy != null && Vector3.Distance(transform.position, targetEnemy.transform.position) > attackRadius)

{

targetEnemy = null;

}

timeBtwAttack -= Time.deltaTime;

if (timeBtwAttack <= 0 && targetEnemy != null)

{

timeBtwAttack = startTimeBtwAttack;

if (targetEnemy != null && targetEnemy.gameObject.activeInHierarchy)

{

Shoot();

}

}

if (!isInDesert) return;

CheckForOasis();

ApplyCurrentEffects();

}

private void CheckForOasis()

{

bool newOasisState = Physics.CheckSphere(transform.position, oasisCheckRadius, oasisLayer);

if (newOasisState != hasOasisNearby)

{

hasOasisNearby = newOasisState;

ApplyCurrentEffects();

}

}

private void ApplyCurrentEffects()

{

if (!isInDesert)

{

startTimeBtwAttack = originalFireRate; // Нормальная скорость атаки

return;

}

if (hasOasisNearby)

{

startTimeBtwAttack = originalFireRate; // Оазис рядом — нормальная скорость

}

else

{

startTimeBtwAttack = originalFireRate \* desertFireRateMultiplier; //В пустыне — медленнее стрельба

}

}

private List<MovementMobs> GetEnemiesInRange()

{

List<MovementMobs> enemiesInRange = new List<MovementMobs>();

foreach (MovementMobs enemy in gameManager.EnemyList)

{

if(enemy != null && Vector3.Distance(transform.position, enemy.transform.position) <= attackRadius)

{

enemiesInRange.Add(enemy);

}

}

return enemiesInRange;

}

private MovementMobs GetNearestEnemy()

{

MovementMobs nearestEnemy = null;

float smallesDistance = float.PositiveInfinity;

foreach(MovementMobs enemy in GetEnemiesInRange())

{

if (Vector3.Distance(transform.position, enemy.transform.position) < smallesDistance)

{

smallesDistance = Vector3.Distance(transform.position, enemy.transform.position);

nearestEnemy = enemy;

}

}

return nearestEnemy;

}

private void Shoot()

{

if (bulletPrefab == null || shootPoint == null) return;

if (towerType == TowerType.SingleTarget)

{

FireBullet(targetEnemy);

}

else if (towerType == TowerType.MultiTarget)

{

List<MovementMobs> enemiesInRange = GetEnemiesInRange();

int targetsToShoot = Mathf.Min(maxTargets, enemiesInRange.Count);

for (int i = 0; i < targetsToShoot; i++)

{

if (i < enemiesInRange.Count)

{

FireBullet(enemiesInRange[i]);

}

}

}

}

private void FireBullet(MovementMobs enemy)

{

if (GameManager.Instance != null && Time.timeScale == 0) return;

GameObject newBullet = Instantiate(bulletPrefab, shootPoint.position, shootPoint.rotation);

Bullet bulletComponent = newBullet.GetComponent<Bullet>();

if (bulletComponent != null)

{

bulletComponent.target = enemy;

bulletComponent.tower = this;

if (isInDesert && !hasOasisNearby)

{

bulletComponent.speed = originalBulletSpeed \* desertSpeedMultiplier;

}

else

{

bulletComponent.speed = originalBulletSpeed;

}

}

else

{

Debug.LogError("У префаба снаряда нет компонента Bullet!");

}

if (shootSound != null)

{

audioSource.PlayOneShot(shootSound);

}

}

private void OnDrawGizmosSelected()

{

Gizmos.color = Color.cyan;

Gizmos.DrawWireSphere(transform.position, attackRadius);

if (isInDesert)

{

Gizmos.color = hasOasisNearby ? Color.green : Color.red;

Gizmos.DrawWireSphere(transform.position, oasisCheckRadius);

}

}

}

**TowerManager.cs**

using TMPro;

using UnityEngine;

using UnityEngine.EventSystems;

using UnityEngine.InputSystem;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

[RequireComponent(typeof(PlayerInput))]

public class TowerManager : MonoBehaviour

{

[Header("Context Menus")]

public GameObject contextMenu;

public GameObject contextMenuUp;

public GameObject contextMenuMaxUp;

public RectTransform menuParent;

private GameObject selectedBuildPoint;

private GameObject selectedTower;

private Camera mainCamera;

private PlayerInput playerInput;

private InputAction touchPositionAction;

private InputAction touchPressAction;

private bool inputBlocked = false;

[Header("Level Settings")]

public bool isLevel1 = false;

public bool isLevel2 = false;

[Header("Debug")]

[SerializeField] private bool debugMode = true;

private static TowerManager \_instance;

void Start()

{

string sceneName = SceneManager.GetActiveScene().name;

isLevel1 = sceneName.Contains("level\_1");

isLevel2 = sceneName.Contains("level\_2");

mainCamera = Camera.main;

playerInput = GetComponent<PlayerInput>();

InitializeInputActions();

HideAllContextMenus();

GameManager.OnGameEnded += HandleGameEnded;

}

void Awake()

{

if (\_instance != null && \_instance != this)

{

Destroy(gameObject);

return;

}

\_instance = this;

DontDestroyOnLoad(gameObject);

FullReset();

}

void OnEnable()

{

SceneManager.sceneLoaded += OnSceneLoaded;

GameManager.OnGameEnded += HandleGameEnded;

}

void OnDisable()

{

SceneManager.sceneLoaded -= OnSceneLoaded;

GameManager.OnGameEnded -= HandleGameEnded;

}

private void FullReset()

{

inputBlocked = false;

selectedBuildPoint = null;

selectedTower = null;

mainCamera = null;

if (playerInput != null)

{

playerInput.actions = null;

playerInput.enabled = false;

playerInput.enabled = true;

}

HideAllContextMenus();

}

private void OnSceneLoaded(Scene scene, LoadSceneMode mode)

{

FullReset();

InitializeForNewScene();

EnsureEventSystemExists();

}

private void InitializeForNewScene()

{

mainCamera = Camera.main;

playerInput = GetComponent<PlayerInput>();

string sceneName = SceneManager.GetActiveScene().name;

isLevel1 = sceneName.Contains("level\_1");

isLevel2 = sceneName.Contains("level\_2");

if (debugMode)

Debug.Log($"Инициализация для {sceneName}. isLevel1: {isLevel1}, isLevel2: {isLevel2}");

}

private void EnsureEventSystemExists()

{

if (FindObjectOfType<EventSystem>() == null)

{

var eventSystem = new GameObject("EventSystem");

eventSystem.AddComponent<EventSystem>();

eventSystem.AddComponent<StandaloneInputModule>();

if (debugMode)

Debug.Log("Создан новый EventSystem для сцены");

}

}

void OnDestroy()

{

GameManager.OnGameEnded -= HandleGameEnded;

}

void InitializeInputActions()

{

touchPressAction = playerInput.actions.FindAction("Fire");

touchPositionAction = playerInput.actions.FindAction("Point");

if (touchPressAction == null || touchPositionAction == null)

{

Debug.LogError("Required Input Actions not found!");

enabled = false;

}

}

void HandleGameEnded(bool gameEnded)

{

inputBlocked = gameEnded;

HideAllContextMenus();

SetButtonsInteractable(contextMenu, !gameEnded);

SetButtonsInteractable(contextMenuUp, !gameEnded);

SetButtonsInteractable(contextMenuMaxUp, !gameEnded);

}

void SetButtonsInteractable(GameObject menu, bool interactable)

{

if (menu == null) return;

var buttons = menu.GetComponentsInChildren<Button>(true);

foreach (var button in buttons)

{

button.interactable = interactable;

}

}

void HideAllContextMenus()

{

if (contextMenu != null) contextMenu.SetActive(false);

if (contextMenuUp != null) contextMenuUp.SetActive(false);

if (contextMenuMaxUp != null) contextMenuMaxUp.SetActive(false);

}

void Update()

{

if (inputBlocked || GameManager.IsInputBlocked) return;

HandleTouchInput();

}

void HandleTouchInput()

{

if (inputBlocked || GameManager.IsInputBlocked) return;

if (touchPressAction.WasPressedThisFrame())

{

Vector2 touchPosition = touchPositionAction.ReadValue<Vector2>();

if (IsPointerOverUI(touchPosition)) return;

ProcessRaycast(touchPosition);

CheckForMenuClose(touchPosition);

}

}

private bool IsPointerOverUI(Vector2 screenPosition)

{

if (EventSystem.current.IsPointerOverGameObject())

return true;

return IsClickInsideAnyMenu(screenPosition) ||

(GameManager.Instance != null &&

(IsClickOnPanel(GameManager.Instance.levelFailedPanel, screenPosition) ||

IsClickOnPanel(GameManager.Instance.levelCompletePanel, screenPosition)));

}

bool IsClickOnPanel(GameObject panel, Vector2 position)

{

return panel != null && panel.activeSelf &&

RectTransformUtility.RectangleContainsScreenPoint(

panel.GetComponent<RectTransform>(),

position

);

}

bool IsClickInsideAnyMenu(Vector2 touchPosition)

{

return (contextMenu != null && contextMenu.activeSelf && IsPositionInMenu(contextMenu, touchPosition)) ||

(contextMenuUp != null && contextMenuUp.activeSelf && IsPositionInMenu(contextMenuUp, touchPosition)) ||

(contextMenuMaxUp != null && contextMenuMaxUp.activeSelf && IsPositionInMenu(contextMenuMaxUp, touchPosition));

}

bool IsPositionInMenu(GameObject menu, Vector2 position)

{

return RectTransformUtility.RectangleContainsScreenPoint(

menu.GetComponent<RectTransform>(),

position

);

}

void ProcessRaycast(Vector2 touchPosition)

{

if (inputBlocked) return;

Ray ray = mainCamera.ScreenPointToRay(touchPosition);

if (Physics.Raycast(ray, out RaycastHit hit))

{

if (hit.collider.CompareTag("BuildPoint"))

{

selectedBuildPoint = hit.collider.gameObject;

ShowContextMenu(touchPosition);

}

else if (hit.collider.CompareTag("Tower"))

{

selectedTower = hit.collider.gameObject;

ShowContextMenuUp(touchPosition);

}

else if (hit.collider.CompareTag("TowerMaxUp"))

{

selectedTower = hit.collider.gameObject;

ShowContextMenuMaxUp(touchPosition);

}

}

}

void CheckForMenuClose(Vector2 touchPosition)

{

if (contextMenu != null && contextMenu.activeSelf && !IsPositionInMenu(contextMenu, touchPosition))

contextMenu.SetActive(false);

if (contextMenuUp != null && contextMenuUp.activeSelf && !IsPositionInMenu(contextMenuUp, touchPosition))

contextMenuUp.SetActive(false);

if (contextMenuMaxUp != null && contextMenuMaxUp.activeSelf && !IsPositionInMenu(contextMenuMaxUp, touchPosition))

contextMenuMaxUp.SetActive(false);

}

private Vector2 GetAdjustedMenuPosition(Vector2 screenPosition, RectTransform menuRect)

{

Vector2 menuSize = menuRect.rect.size;

Vector2 canvasSize = menuParent.rect.size;

RectTransformUtility.ScreenPointToLocalPointInRectangle(

menuParent,

screenPosition,

null,

out Vector2 localPoint

);

float halfMenuWidth = menuSize.x / 2;

float halfMenuHeight = menuSize.y / 2;

if (localPoint.x + halfMenuWidth > canvasSize.x / 2)

{

localPoint.x = canvasSize.x / 2 - halfMenuWidth;

}

else if (localPoint.x - halfMenuWidth < -canvasSize.x / 2)

{

localPoint.x = -canvasSize.x / 2 + halfMenuWidth;

}

if (localPoint.y + halfMenuHeight > canvasSize.y / 2)

{

localPoint.y = canvasSize.y / 2 - halfMenuHeight;

}

else if (localPoint.y - halfMenuHeight < -canvasSize.y / 2)

{

localPoint.y = -canvasSize.y / 2 + halfMenuHeight;

}

return localPoint;

}

void ShowContextMenu(Vector2 screenPosition)

{

if (contextMenu == null || inputBlocked) return;

if (isLevel1 || isLevel2)

{

Transform firstButton = contextMenu.transform.GetChild(0);

Button button = firstButton.GetComponent<Button>();

if (contextMenu.transform.childCount > 1)

{

TMP\_Text priceText = contextMenu.transform.GetChild(1).GetComponent<TMP\_Text>();

if (priceText != null && button != null)

{

if (button.onClick.GetPersistentEventCount() > 0)

{

GameObject towerPrefab = button.onClick.GetPersistentTarget(0) as GameObject;

if (towerPrefab != null)

{

Tower tower = towerPrefab.GetComponent<Tower>();

if (tower != null)

{

priceText.text = tower.cost.ToString();

}

}

}

}

else

{

Debug.LogError("Не найден TextCost или компонент Button!");

}

}

for (int i = 2; i < contextMenu.transform.childCount; i++)

{

contextMenu.transform.GetChild(i).gameObject.SetActive(false);

}

}

contextMenu.SetActive(true);

RectTransform menuRect = contextMenu.GetComponent<RectTransform>();

Vector2 adjustedPosition = GetAdjustedMenuPosition(screenPosition, menuRect);

menuRect.localPosition = adjustedPosition;

}

void ShowContextMenuUp(Vector2 screenPosition)

{

if (contextMenuUp == null || selectedTower == null || inputBlocked) return;

Tower tower = selectedTower.GetComponent<Tower>();

if (tower == null) return;

if (isLevel1)

{

ShowContextMenuMaxUp(screenPosition);

return;

}

if (isLevel2)

{

contextMenuUp.transform.GetChild(0).gameObject.SetActive(true);

contextMenuUp.transform.GetChild(1).gameObject.SetActive(true);

TMP\_Text upgradePriceText = contextMenuUp.transform.Find("TextCost")?.GetComponent<TMP\_Text>();

TMP\_Text sellPriceText = contextMenuUp.transform.Find("TextCostSell")?.GetComponent<TMP\_Text>();

if (upgradePriceText != null)

upgradePriceText.text = Mathf.RoundToInt(tower.cost \* 1.6f).ToString();

if (sellPriceText != null)

sellPriceText.text = Mathf.RoundToInt(tower.cost \* 0.6f).ToString();

}

else

{

TMP\_Text upgradePriceText = contextMenuUp.transform.Find("TextCost")?.GetComponent<TMP\_Text>();

TMP\_Text sellPriceText = contextMenuUp.transform.Find("TextCostSell")?.GetComponent<TMP\_Text>();

if (upgradePriceText != null)

upgradePriceText.text = Mathf.RoundToInt(tower.cost \* 1.6f).ToString();

if (sellPriceText != null)

sellPriceText.text = Mathf.RoundToInt(tower.cost \* 0.6f).ToString();

}

contextMenuUp.SetActive(true);

Vector2 adjustedPosition = GetAdjustedMenuPosition(screenPosition, contextMenuUp.GetComponent<RectTransform>());

contextMenuUp.GetComponent<RectTransform>().localPosition = adjustedPosition;

}

void ShowContextMenuMaxUp(Vector2 screenPosition)

{

if (contextMenuMaxUp == null || selectedTower == null || inputBlocked) return;

var tower = selectedTower.GetComponent<Tower>();

if (tower == null) return;

contextMenuMaxUp.transform.GetChild(1).GetComponent<TMP\_Text>().text = ((int)Mathf.Round(tower.cost \* 0.6f)).ToString();

contextMenuMaxUp.SetActive(true);

Vector2 adjustedPosition = GetAdjustedMenuPosition(screenPosition, contextMenuMaxUp.GetComponent<RectTransform>());

contextMenuMaxUp.GetComponent<RectTransform>().localPosition = adjustedPosition;

}

public void BuildTower(GameObject towerPrefab)

{

if (inputBlocked || towerPrefab == null || selectedBuildPoint == null) return;

var tower = towerPrefab.GetComponent<Tower>();

if (tower == null) return;

if (GameManager.Instance.money - tower.cost >= 0)

{

GameManager.Instance.ChangeMoney(-tower.cost);

GameObject newTower = Instantiate(towerPrefab, selectedBuildPoint.transform.position, Quaternion.identity);

if (isLevel1)

{

newTower.tag = "TowerMaxUp";

Tower towerComponent = newTower.GetComponent<Tower>();

if (towerComponent != null)

{

towerComponent.nextTower = null;

}

}

Destroy(selectedBuildPoint);

contextMenu.SetActive(false);

}

}

public void SellTower(GameObject BuildPoint)

{

if (inputBlocked || selectedTower == null || BuildPoint == null) return;

var tower = selectedTower.GetComponent<Tower>();

if (tower == null) return;

GameManager.Instance.ChangeMoney((int)Mathf.Round(tower.cost \* 0.6f));

contextMenuUp.SetActive(false);

contextMenuMaxUp.SetActive(false);

Destroy(selectedTower);

Instantiate(BuildPoint, selectedTower.transform.position, Quaternion.identity);

}

public void TowerUp()

{

if (inputBlocked || selectedTower == null) return;

var tower = selectedTower.GetComponent<Tower>();

if (tower == null || tower.nextTower == null) return;

int upgradeCost = (int)Mathf.Round(tower.cost \* 1.6f);

if (GameManager.Instance.money - upgradeCost >= 0)

{

GameManager.Instance.ChangeMoney(-upgradeCost);

contextMenuUp.SetActive(false);

Destroy(selectedTower);

Instantiate(tower.nextTower, selectedTower.transform.position, Quaternion.identity);

}

}

}

**WavwSpawner.cs**

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

using TMPro;

using System.Linq;

public class WaveSpawner : MonoBehaviour

{

[System.Serializable]

public class TrackConfig

{

public Transform[] spawnPoints;

public string waypointTag;

public string altWaypointTag;

public Wave[] waves;

[HideInInspector] public Transform[] waypoints;

[HideInInspector] public Transform[] altWaypoints;

}

[System.Serializable]

public class Wave

{

public GameObject mobPrefab;

public int mobCount;

public float spawnRate;

}

public TrackConfig[] tracks;

public float timeBetweenWaves = 2f;

public bool loopWaves = false;

[Header("UI")]

public TMP\_Text waveCounterText;

private int[] currentWaveIndices;

private bool[] isSpawning;

private List<GameObject>[] activeMobsPerTrack;

private int[] lastSpawnPointIndices;

void Start()

{

InitializeTracks();

StartAllTrackSpawners();

}

private void InitializeTracks()

{

currentWaveIndices = new int[tracks.Length];

isSpawning = new bool[tracks.Length];

activeMobsPerTrack = new List<GameObject>[tracks.Length];

lastSpawnPointIndices = new int[tracks.Length];

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

{

tracks[i].waypoints = GetSortedWaypoints(tracks[i].waypointTag);

tracks[i].altWaypoints = GetSortedWaypoints(tracks[i].altWaypointTag);

activeMobsPerTrack[i] = new List<GameObject>();

lastSpawnPointIndices[i] = -1;

}

}

private Transform[] GetSortedWaypoints(string tag)

{

return GameObject.FindGameObjectsWithTag(tag)

.Where(go => go != null)

.OrderBy(go => go.name)

.Select(go => go.transform)

.ToArray();

}

private void StartAllTrackSpawners()

{

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

{

StartCoroutine(TrackSpawnLoop(i));

}

}

IEnumerator TrackSpawnLoop(int trackIndex)

{

TrackConfig track = tracks[trackIndex];

while (currentWaveIndices[trackIndex] < track.waves.Length)

{

Wave currentWave = track.waves[currentWaveIndices[trackIndex]];

currentWaveIndices[trackIndex]++;

UpdateWaveCounter();

yield return StartCoroutine(SpawnWave(trackIndex, currentWave));

yield return StartCoroutine(WaitForAllMobsDefeated(trackIndex));

if (currentWaveIndices[trackIndex] < track.waves.Length)

{

yield return new WaitForSeconds(timeBetweenWaves);

}

}

if (loopWaves)

{

currentWaveIndices[trackIndex] = 0;

StartCoroutine(TrackSpawnLoop(trackIndex));

}

}

IEnumerator SpawnWave(int trackIndex, Wave wave)

{

isSpawning[trackIndex] = true;

activeMobsPerTrack[trackIndex].Clear();

for (int i = 0; i < wave.mobCount; i++)

{

GameObject mob = SpawnMob(trackIndex, wave.mobPrefab);

if (mob != null) activeMobsPerTrack[trackIndex].Add(mob);

yield return new WaitForSeconds(1f / wave.spawnRate);

}

isSpawning[trackIndex] = false;

}

GameObject SpawnMob(int trackIndex, GameObject mobPrefab)

{

TrackConfig track = tracks[trackIndex];

if (track.spawnPoints.Length == 0 || mobPrefab == null)

return null;

lastSpawnPointIndices[trackIndex] =

(lastSpawnPointIndices[trackIndex] + 1) % track.spawnPoints.Length;

Transform spawnPoint = track.spawnPoints[lastSpawnPointIndices[trackIndex]];

GameObject mob = Instantiate(mobPrefab, spawnPoint.position, spawnPoint.rotation);

MovementMobs movement = mob.GetComponent<MovementMobs>();

if (movement != null)

{

if (lastSpawnPointIndices[trackIndex] == 0)

{

movement.Initialize(track.waypoints);

}

else

{

movement.Initialize(track.altWaypoints);

}

}

return mob;

}

IEnumerator WaitForAllMobsDefeated(int trackIndex)

{

while (true)

{

activeMobsPerTrack[trackIndex].RemoveAll(mob => mob == null);

if (activeMobsPerTrack[trackIndex].Count == 0)

yield break;

yield return null;

}

}

private void UpdateWaveCounter()

{

if (waveCounterText != null)

{

int totalWaves = tracks.Sum(t => t.waves.Length);

int completedWaves = currentWaveIndices.Sum();

waveCounterText.text = $"{completedWaves}/{totalWaves}";

}

}

public bool AllWavesCompleted

{

get

{

if (currentWaveIndices == null || tracks == null) return false;

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

{

if (currentWaveIndices[i] < tracks[i].waves.Length)

{

return false;

}

}

return true;

}

}

public bool IsSpawning

{

get

{

if (isSpawning == null) return false;

return isSpawning.Any(x => x);

}

}}

**Приложение 3.**

Ссылка на репозиторий: <https://github.com/Voronova1/TD_Voronova_Popova_Feygina>

Ссылка на видеопрезентацию и видео прохождения игры:

<https://drive.google.com/drive/folders/1A404YQtLICcf48HKvS6-1gWFgw6j7esN?usp=sharing>