Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования «КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА АНАЛИЗА ДАННЫХ И ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Направление: 38.03.05 – Бизнес-информатика

КУРСОВАЯ РАБОТА

Игра на Unity

Студент 3 курса
Группы 09-202
«___»_____2025 г. _____ Сидорова А.И.

Научный руководитель
старший преподаватель
«___»_____2025 г. _____ Тубальцева К.Е.

Казань – 2025

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
ГЛАВА 1. ИССЛЕДОВАНИЕ КОНЦЕПЦИИ 2D-ПЛАТФОРМЕРОВ И ТЕХНОЛОГИЙ ИХ РЕАЛИЗАЦИИ 5
1.1 Особенности жанра 2D-платформеров 5
1.2 Обзор движков для разработки 2D-игр 5
1.3 Обоснование выбора инструментов 7
ГЛАВА 2. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА 2D-ПЛАТФОРМЕРА «LOST LETTERS» 8
2.1. Требования к разрабатываемой игре 8
2.3. Диаграмма взаимодействия системы с пользователем 10
ГЛАВА 3. РАЗРАБОТКА ПРОГРАММНОГО ПРОДУКТА 12
3.1. Создание базовой сцены и управление игроком 12
3.2. Дизайн уровня и оформление окружения 13
3.3 Реализация врагов и боевой системы 15
3.4 Создание интерфейса и игровых экранов 18
ЗАКЛЮЧЕНИЕ 23
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 25

ВВЕДЕНИЕ

Современная игровая индустрия — это динамично развивающаяся сфера, где даже небольшие проекты могут найти свою аудиторию. Особой популярностью пользуются 2D-платформеры — они сочетают простоту освоения с глубоким геймплеем, а также дают возможность проявить ловкость и реакцию, что делает их идеальным выбором для большинства игроков. Такие игры подходят для тех, кто только начинает свой путь в разработке, что делает их отличным выбором для учебного проекта [2].

В данной работе представлен 2D-платформер «Lost Letters», созданный на движке Unity. В игре игрок берёт на себя роль призрака-почтальона., который доставляет письма между мирами живых и мёртвых, помогая душам найти умиротворение. На данный момент готов первый уровень, где есть система движения персонажа, препятствия, враги с простым ИИ, визуальное оформление и начальные диалоги. В дальнейшем сюжет можно развить.

Актуальность работы связана с растущим интересом к инди-играм, и пользой разработки для обучения. Современные 2D-платформеры всё чаще делают упор на сюжет и эмоциональную составляющую, что соответствует актуальным тенденциям игровой индустрии [2]. «Lost Letters» выделяется оригинальной идеей, которая до сих пор не была реализована в этом жанре. Создание проекта на Unity позволяет научиться строить игровую логику и на практике увидеть, как объекты в игре двигаются и взаимодействуют друг с другом [5].

Практическая ценность работы проявляется в изучении всех этапов создания игрового приложения, что помогает развить навыки программирования, тестирования, логики, дизайна и проектирования, необходимые для будущей профессии. Научная новизна заключается в исследовании возможностей Unity для разработки 2D-игр с историей, которая не только интересна, но и вызывает эмоции [2].

Цель курсовой работы — разработать игровой уровень в жанре 2D-платформера с базовым управлением, элементарной системой взаимодействия

между объектами и сюжетными сценами, используя Unity и язык программирования C#.

Задачи курсовой работы:

1. Изучить возможности Unity для создания 2D-платформеров
2. Спроектировать уровень и его интерфейс
3. Настроить движение персонажа и взаимодействие с объектами
4. Добавить врага с простым поведением
5. Внедрить вступительный диалог для сюжета
6. Выполнить тестирование и исправить ошибки

Объектом исследования стал процесс создания 2D-игр на Unity, а предметом — сам платформер, сделанный с использованием бесплатных ресурсов и инструментов движка [6].

Работа включает в себя описание этапов проектирования и разработки, иллюстрации ключевых фрагментов интерфейса, а также исходный код основных игровых компонентов. Реализованная игра показывает применение полученных знаний на практике и может быть доработана в будущем.

ГЛАВА 1. ИССЛЕДОВАНИЕ КОНЦЕПЦИИ 2D-ПЛАТФОРМЕРОВ И ТЕХНОЛОГИЙ ИХ РЕАЛИЗАЦИИ

1.1 Особенности жанра 2D-платформеров

Жанр 2D-платформеров является одним из самых популярных и узнаваемых направлений в игровой индустрии [1]. Основой классических платформеров является перемещение персонажа по двумерному миру, включающее преодоление препятствий, прыжки, сбором предметов и взаимодействием с врагами. Такие игры выделяются своей простотой и удобством управления, что делает их доступными для людей разных возрастов и с любым игровым опытом [2].

Важной составляющей платформеров является баланс между легкостью прохождения и сложностью игровых задач, что способствует поддержанию интереса и вовлеченности игроков. Несмотря на кажущуюся простоту, 2D-платформеры часто содержат разнообразные элементы — головоломки, сражения, сюжетные линии, сбор коллекционных предметов и другие. Это делает игровой процесс более разнообразным и добавляет ему глубины [3].

Современные 2D-платформеры успешно сочетают классические элементы жанра с новыми технологиями, включая более качественную графику с детализированными фонами, плавную анимацию движений и реалистичную физику [6]. Эти особенности делают игры визуально привлекательными, динамичными и реалистичными, что отвечает ожиданиям современных игроков и позволяет жанру сохранять свою популярность на игровом рынке.

1.2 Обзор движков для разработки 2D-игр

Для разработки 2D-игр существует множество игровых движков, каждый из которых имеет свои особенности, преимущества и ограничения. В данном разделе рассматриваются три популярных движка: Godot, Unreal Engine и Unity, чтобы выявить их сильные и слабые стороны при разработке 2D-платформеров.

Отметим для начала плюсы каждого из этих движков.

1. Все три игровых движка можно установить бесплатно.

2. Unity лидирует в кроссплатформенной разработке, поддерживая свыше 25 платформ, включая iOS, Android, Windows, macOS, Linux, PlayStation и Xbox. UE также поддерживает множество платформ, включая мобильные устройства и консоли, но с меньшим охватом. Godot имеет более ограниченный набор платформ по сравнению со своими конкурентами [5].

3. На Unity есть множество универсальных инструментов для разработки 2D- и 3D-игр, а также доступ к магазину ресурсов (Asset Store) для ускорения разработки [8]. UE выделяется мощными инструментами визуализации, включая визуальную систему Blueprints, которая позволяет создавать механики без программирования. Godot обеспечивает гибкость для 2D- и 3D-проектов благодаря простому языку GDScript, схожему с Python.

4. Unity создает компактные сборки (около 15 - 20 МБ). UE генерирует более крупные сборки (около 70 МБ), но обеспечивает высокую графическую производительность. Godot отличается легковесностью и минимальными требованиями к ресурсам [9].

Несмотря на свои преимущества, каждый движок имеет ограничения:

1. Unity: компания прежде всего ориентирована на бизнес-партнеров, что может создавать сложности для индивидуальных разработчиков. Также есть сложности с обновлением версий. Проекты, созданные на более старых версиях движка, могут некорректно работать или требовать доработки при переходе на новые версии [5].

2. UE: для полноценной работы требуется знание языка C++, что создает барьер для новичков. Движок ориентирован на крупные проекты и больше подходит для разработки 3D игр, что делает его менее подходящим для тех, кто хочет выпустить простую 2D игру. Высокие системные требования также ограничивают его использование на слабых компьютерах [2].

3. Godot: Сообщество Godot меньше, чем у Unity и Unreal Engine. Это

может стать проблемой для новичков при поиске материалов и помощи [2].

1.3 Обоснование выбора инструментов

В процессе выбора среды разработки главным фактором стала возможность наглядно и удобно создавать логику 2D-игры, работать с графикой, анимацией и физикой [5]. В предыдущем параграфе были подробно рассмотрены такие движки, как Unreal Engine, Godot и Unity. Несмотря на все плюсы каждого из них, предпочтение было отдано Unity.

На выбор именно этой среды разработки повлияла не только ее популярность, но и то, что в движке используется язык C# [4]. Эти условия значительно упростили процесс написания кода и позволили сосредоточиться на продумывание игровой логики, а не на изучении языка программирования.

Еще одно ключевое приемущество – большое и активное сообщество разработчиков. Оно предоставляет множество учебных ресурсов, включая руководства, видеоуроки, готовые модели и скрипты [9]. В рамках курсовой работы не просто создается игра, а еще и приобретаются новые знания. Учитывая эти моменты, Unity стал лучшим выбором.

Также стоит отметить интерфейс среды. Он может показаться трудным для новичка, но на деле он очень удобен и прост. Редактор сцены позволяет легко создавать и перемещать игровые объекты. Кроме того, в Unity нетрудно настроить взаимодействие между объектами. Нужно просто добавить скрипты или компоненты, - и персонаж уже бегает по платформам и врезается в стены [2]. Также ускоряет работу и использование встроенных инструментов, например, Rigidbody2D для физики, Tilemap для создания уровней, Animator для анимаций и Prefab system для повторного использования объектов [6]. Они сокращают время разработки и позволяют направить больше сил на создание механик игры и ее общей организации.

Несмотря на более сильные визуальные возможности UE и простоту Godot, Unity стал оптимальным вариантом для написания курсовой работы.

ГЛАВА 2. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА 2D-ПЛАТФОРМЕРА «LOST LETTERS»

2.1. Требования к разрабатываемой игре

Игра Lost Letters — это классический 2D-платформер с добавлением небольшого сюжета. Цель игры — обеспечить увлекательный геймплей с простым и понятным управлением, атмосферным геймдизайном и стабильной работой без ошибок. Поэтому можно выделить следующие требования к игре [1]:

1. Игрок должен иметь возможность управлять персонажем, используя клавиатуру. Для этого нужно реализовать:
   - Движение влево/вправо через клавиши A/D или стрелки.
   - Прыжок (клавиша Пробел).
   - Удар по врагу (клавиша X).
2. Для погружения в сюжет следует добавить текстовые элементы. В них будет рассказана история персонажа.
3. Добавить врагов с элементарным ИИ: патрулирование, преследование, возвращение на точку патрулирования и атака игрока. Это добавит игре сложности и сделает ее более интересной [2].
4. Реализовать простой механизм атаки, для реализации взаимодействия персонажа с врагами.
5. Проработать точное и четкое взаимодействие персонажа с платформами, препятствиями, чтобы персонаж не застревал и не проваливался за сцену. Для этого используется физическая система Unity (копмоненты Rigidbody 2D, Box Collider 2D) [3].
6. Разработать логику смерти и возрождения персонажа является неотъемлемой частью игры. Стоит разработать логику данных событий.
7. Добавить завершение уровня при достижении конечной точки сцены или по истечению попыток прохождения, а также появление соответствующего экрана.
8. Разработать геймдизайн, чтобы по оформлению все

соответствовало с сюжетом и идей игры.

9. Настроить анимации для движущихся игровых объектов. Это сделает игру более динамичной.

10. Фон с эффектом параллакса сделает игру визуально лучше.

11. Создать несколько сцен и настроить переходы между ними.

12. Добавить UI-элементы: здоровье в виде сердечек, подсказки управления, пауза, завершение уровня. Это обеспечит лучший пользовательский опыт.

13. Изучить и добавить камеру, которая будет следовать за главным героем при его движении [6].

14. Игра должна быть стабильной, без вылетов и критических ошибок при выполнении базовых действий.

Эти требования обеспечат основу для создания увлекательного и стабильного платформера, соответствующего тематике игры.

2.2 Описание структуры игры

При разработке любого продукта важно иметь представление о его структуре, чтобы четко понимать, как должен выглядеть итоговый проект. Проектирование в разы упрощает работу, помогает избежать хаотичных решений и сохранить время, которого никогда не хватает.

Всего игра имеет 4 сцены:

1. Сцена Menu (главное меню). Первая сцена, которая отображается при запуске игры. Служит для навигации и содержит кнопки «Играть» и «Выход».

2. Сцена ChooseLvl (выбор уровня). Появляется после нажатия на предыдущей сцене кнопки «Играть». Перед открытием сцены появляется сюжетный монолог с историей главного героя. На самой сцене есть кнопка для выбора уровня и кнопка «Назад». После выбора уровня появляется небольшой сюжетный диалог. Пока реализован только один уровень, но структура позволяет в будущем добавить больше уровней.

3. Сцена FirstLvl (первый уровень). Основная игровая сцена, где происходит весь геймплей. Игрок управляет персонажем-призраком, которые перемещается по уровню и атакует врагов. После успешного прохождения уровня появляется соответствующая надпись и кнопка перехода к выбору уровня. Если игрок терпит поражение, то активируется сцена GameOver

4. Сцена GameOver (поражение). Сообщает игроку, что уровень провален. Появляется после истечения попыток прохождения. На сцене есть кнопка, которая перенаправляет игрока в главное меню.

Особенностью проекта стало использование встроенной в Unity системы анимаций [6]. Несколько игровых элементов имеют анимации на каждое свое состояние (передвижение, атака, смерть и другие). Это помогло создать визуально цельные образы, несмотря на использование простой 2D-графики в работе. Благодаря визуальному редактору Animator в Unity настройка переходов между состояниями была интуитивной и не требовала написание сложного кода [5]. Это значительно ускорило разработку и придало игре динамичности.

Для структурирования всех материалов, использованных при создании игры, проект логически разбит на папки. Созданы отдельные директории для спрайтов, сцен, скриптов, анимаций. Это сильно упростило навигацию и управление материалами во время разработки, позволяя быстро находить нужные файлы и поддерживать порядок.

Такая структура не только делает игровой процесс понятным и не перегружает игрока лишними деталями, но и очень удобна для меня, как для разработчика. После формирования структуры игры начался самый интересный и увлекательный этап, ее реализация.

2.3. Диаграмма взаимодействия системы с пользователем

На рисунке 1 схематично представлен процесс взаимодействия игрока с системой. Пользователь запускает приложение, нажимает кнопку играть, знакомится с идей игры через небольшие текстовые вставки, выбирает

уровень, читает сюжет, переходит на игровую сцену, управляет персонажем, атакует врагов, продвигается вперед по игровой сцене и завершает уровень или проигрывает, после чего возвращается в главное меню или к выбору уровня.

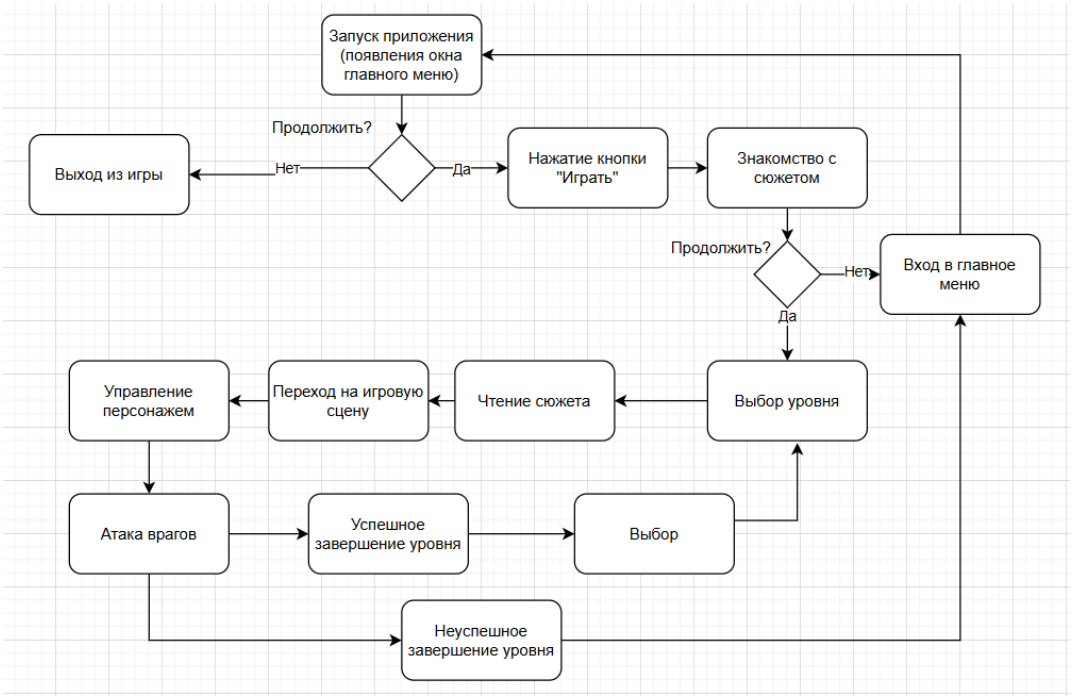

Рисунок 1 – схема взаимодействия пользователя с игрой

ГЛАВА 3. РАЗРАБОТКА ПРОГРАММНОГО ПРОДУКТА

3.1. Создание базовой сцены и управление игроком

В первую очередь было решено создать основную игровую сцену, так как это самый сложный этап. Первым шагом добавляем платформу, на которую поставим персонажа. Добавляем игровой объект на сцену и меняем в окне Inspector его длину и ширину так, чтобы образовалась платформа. Добавляем этой платформе компоненты физики, чтобы наш персонаж мог по ней ходить [6].

Далее добавим самого персонажа игрока. Также создаем игровой объект, добавляем ему компоненты Rigidbody2D и BoxColider2D, чтобы обеспечить физику столкновения и передвижения. Затем начинается написание первого скрипта в Unity. Для удобства создаем отдельную папку, в которой будем хранить все скрипты и добавляем наш скрипт, который будет отвечать за логику передвижения игрока (вправо/влево и прыжок) (см. Рисунок 2) [4].

```
private void Update() {
    if (!isAttacking) {
        movementX = Input.GetAxisRaw("Horizontal");
    }
    else {
        movementX = 0f;
    }
    movementX = Input.GetAxisRaw("Horizontal");

    if (Input.GetButtonDown("Jump") && IsGrounded() && !isAttacking) {
        Jump();
    }
```

Рисунок 2 — скрипт движения игрока

Чтобы сделать игру более динамичной и живой, после настройки управления приступаем к добавлению анимации. Скачиваем картинки (спрайты) персонажа и для удобства помещаем их в папку Player, созданную в папке Sprites [7, 8]. Также для хранения всех анимаций в одном месте создаем папку Animation и в ней папку Player. Добавляем туда объект AnimationController и анимации для покоя, ходьбы, прыжка. В окно Animation вставляем наши картинки для создания покадровой анимации. В окне

Animator настраиваем анимационный граф, добавляем условия перехода между состояниями, создав переменные isRunning, isWalking и т.д. (см. Рисунок 3) [5]. Эти переменные используются в коде для контроля анимациями.

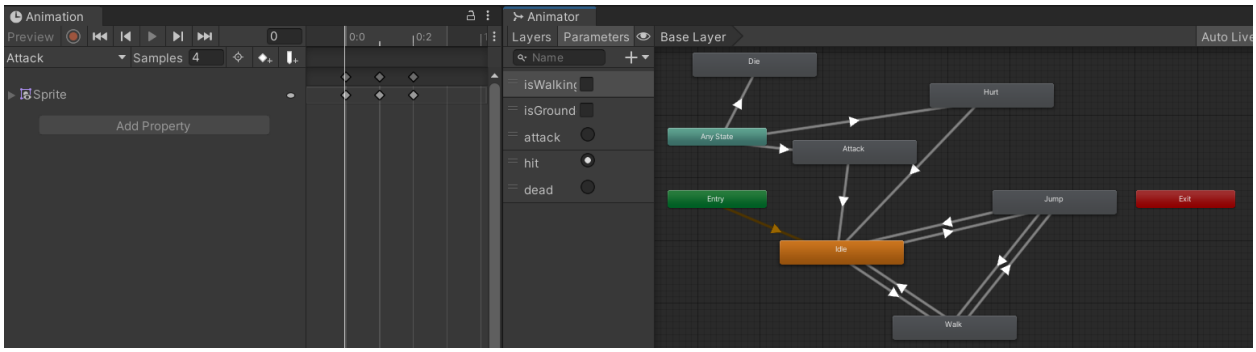

Рисунок 3 – анимационный граф игрока

Для того чтобы камера следовала за движением игрока и обеспечивала комфортный обзор уровня, используем компонент Cinemachine Virtual Camera. Камера настроена так, чтобы отслеживать позицию игрока и плавно перемещаться за ним в пределах уровня [6].

3.2. Дизайн уровня и оформление окружения

После реализации базовой механики движения и анимации игрока, приступаем к левел дизайну. Создаем объект Grid и добавляем туда TileMap, которые будут отвечать за различное окружение. Основной инструмент, использованный для построения игрового мира — Tile Palette (см. Рисунок 4) [6]. С его помощью создаем ландшафт: земля, камни, дома и другие элементы. Редактор позволяет вручную «рисовать» уровень из маленьких повторяющихся изображений (тайлов), как мозаикой. Это ускоряет процесс и делает редактирование карты простым.

Рисунок 4 – окно Tile Pallete

Особое внимание уделяем фону. Для придания динамичности сцене используем эффект параллакса – это когда задние фоны двигаются медленнее переднего плана, создавая иллюзию объема [2]. Фоны расположены на отдельных слоях, каждый из которых настроен с разной скоростью движения в зависимости от положения камеры (см. Рисунок 5).

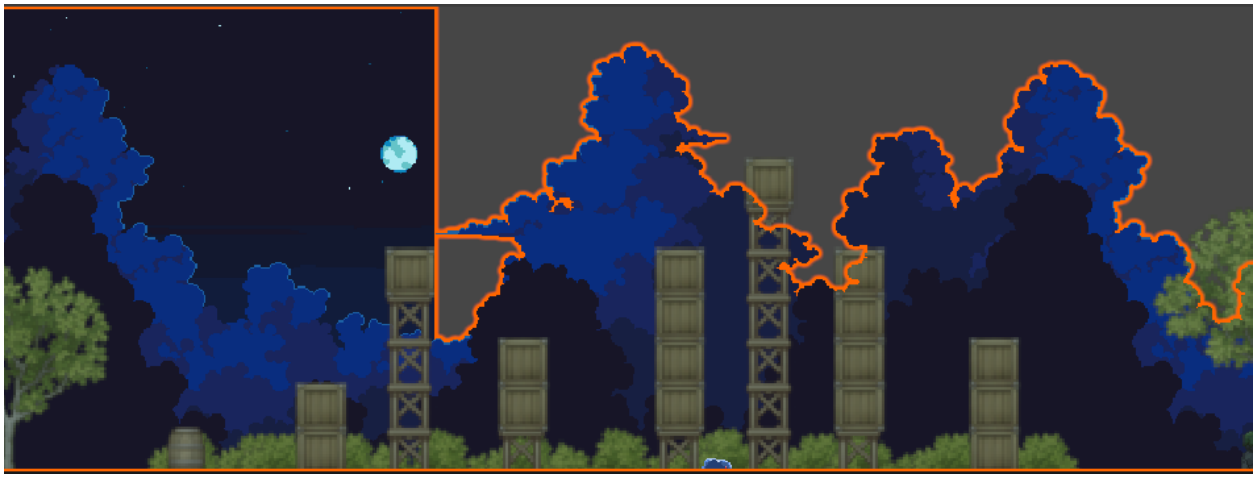

Рисунок 5 – фон

3.3 Реализация врагов и боевой системы

Для начала добавим врага, по аналогии с добавлением игрока. Его основная задача заключается в патрулирование территории вокруг определенной точки (см. Рисунок 6).

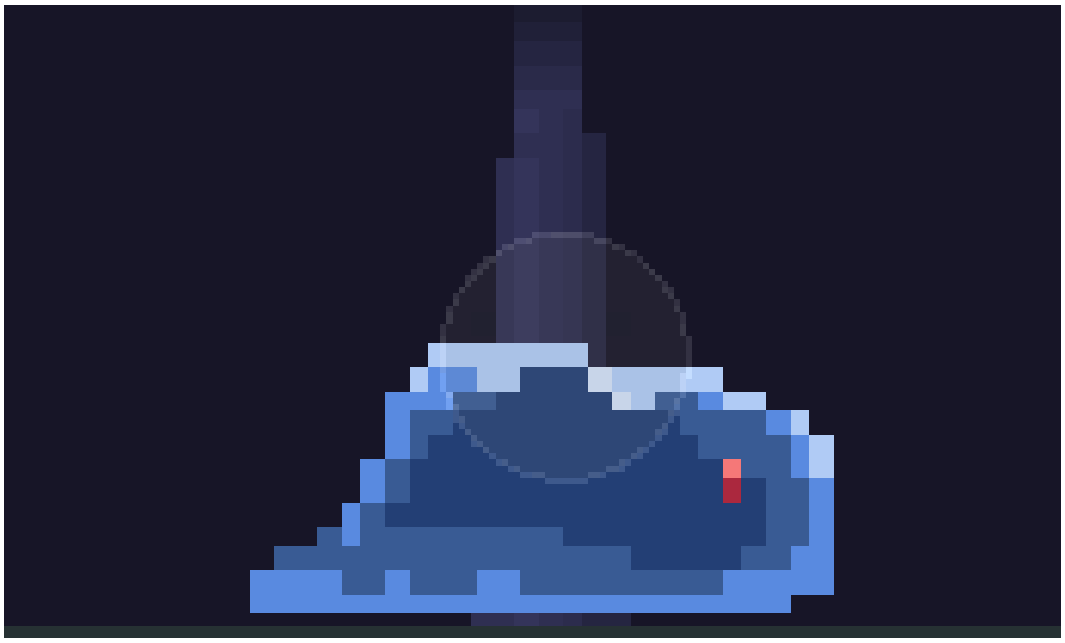

Рисунок 6 – точка, вокруг которой патрулирует враг

Также при приближении игрока на определенное расстояние, враг переходит в состояние преследования и начинает двигаться к игроку и атаковать его. Когда игрок отходит на безопасное расстояние, враг возвращается к своей точке патрулирования. Вся эта логика прописана в скрипте, код которого представлен в приложении 1.

Чтобы враг выглядел реалистично, добавим ему анимации ходьбы, атаки, получения урона и смерти, по аналогии с добавлением анимации игрока (см. Рисунок 7) [7].

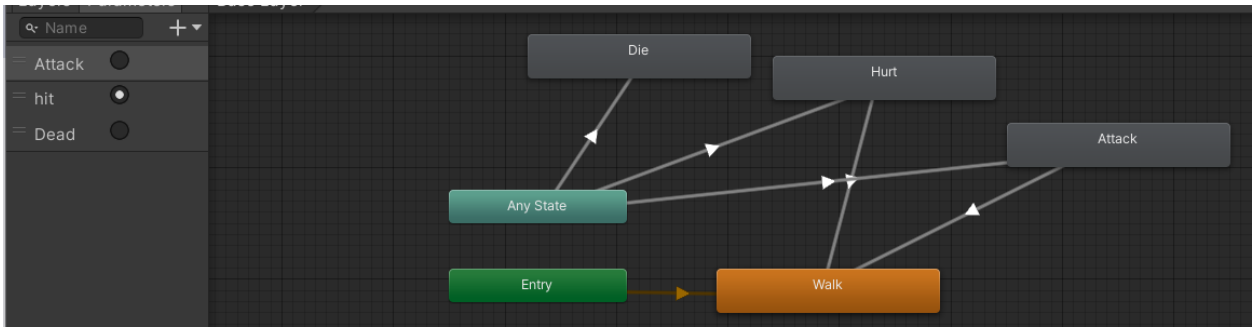

Рисунок 7 – анимационный граф врага

Затем реализуем систему боя. При слишком близком приближении врага к игроку, враг наносит урон. Это прописано в следующем коде (см. Рисунок 8) [4].

```
void TryAttack() {
    canAttack = false;
    healthScript.LastAttackTime = Time.time;
    anim.SetTrigger("Attack");
    Invoke(nameof(DealDamage), 0.4f);
    Invoke(nameof(ResetAttack), healthScript.attackCooldown);
}
Ссылок: 1
void ResetAttack() {
    canAttack = true;
}
Ссылок: 1
public void DealDamage() {
    if (player == null) {
        return;
    }

    if (playerHealth == null) {
        return;
    }

    float distanceToPlayer = Vector2.Distance(transform.position, player.position);
    if (distanceToPlayer <= healthScript.attackRange) {
        playerHealth.TakeDamage(healthScript.damageToPlayer);
    }
```

Рисунок 8 – скрипт атаки

Логика, которая связана с состоянием здоровья динамических игровых объектов (получение урона, смерть), прописана в скриптах PlayerHelth и EnemyHealth. Они представлены в приложении 2 и 3 соответственно.

Главному персонажу также добавляем дополнительные анимации: получение урона, атака, смерть. Чтобы наносить урон врагам, создан объект Splash Effect, который появляется при нажатии на кнопку атаки. В остальное время этот объект скрыт. Если Splash Effect задевает врага, то последний получает урон. Это прописано в скрипте, представленном на рисунке 9. На рисунке 10 приведена часть кода, отвечающая за логику атаки врага и вызов Slash Effect.

```
// Наносим урон врагу при попадании
Сообщение Unity | Ссылок: 0
private void OnTriggerEnter2D(Collider2D collision) {
    EnemyHealth enemy = collision.GetComponent<EnemyHealth>();
    if (enemy != null && !enemy.IsDead()) {
        Vector2 direction = (enemy.transform.position - transform.position).normalized;
        enemy.TakeDamage(damage,direction); // Наносим урон
        Debug.Log("Enemy damage");
    }
}
```

Рисунок 9 – скрипт нанесения урона врагу

```
void Attack() {
    if(isAttacking) return;
    anim.SetTrigger("attack");
    // Запуск анимации SlashEffect
    if (slashEffectPrefab != null) {
        DestroyAttack destroyAttackScript = slashEffectPrefab.GetComponent<DestroyAttack>();
        if (destroyAttackScript != null) {
            destroyAttackScript.ActivateSlashEffect(); // Активируем слеш-эффект
        }
    }
    isAttacking = true;
    // Обновляем время последней атаки
    lastAttackTime = Time.time;
    // Делаем паузу (задержка) перед сбросом флага атаки
    Invoke(nameof(ResetAttack), attackCooldown); // После кулдауна разблокируем возможность атаки
}
    Ссылок: 1
    void ResetAttack() {
    isAttacking = false;
}
```

Рисунок 10 – скрипт атаки врага игроком

Превращаем врага в префаб, чтобы расставить его по всей сцене [3]. Это позволит удобно копировать врага с уже готовыми настройками поведения, анимации и скриптов. Для интереса врагов расставляем в разные части уровня.

Когда у игрока не остается жизней, он умирает. Чтобы игра не заканчивалась так быстро, реализуем логику возрождения (см. Рисунок 15). Также для интереса добавим условие: если игрок умирает определенное число раз, то уровень считается проваленным и появляется окно «Game Over».

```
public void RespawnPlayer() {
    if (currentPlayer != null) Destroy(currentPlayer);

    currentPlayer = Instantiate(playerPrefab, spawnPoint.position, Quaternion.identity);
    cam.Follow = currentPlayer.transform;
}
```

Рисунок 15 – скрипт для возрождения игрока

3.4 Создание интерфейса и игровых экранов

Заканчивая работу с игровой сценой, добавим базовые UI-элементы, управляющие состоянием игры и отображением информации для игрока [3].

Интерфейс здоровья реализуем в виде сердечек, отображающих количество оставшихся жизней (см. Рисунок 11). При получении урона, одно сердечко исчезает. Элементы создаем в Canvas и обновляем скриптом [2].

Рисунок 11 – интерфейс здоровья

Для управления состоянием игры добавляем кнопку паузы. Чтобы эта кнопка при нажатии выполняла свои функции, создаем скрипт (см. Приложение 4). Скрипт добавляем к объекту Canvas. Для того чтобы привязать код к кнопке в Inspector настроим событие On Click(), выбрав нужный скрипт и метод (см. Рисунок 12).

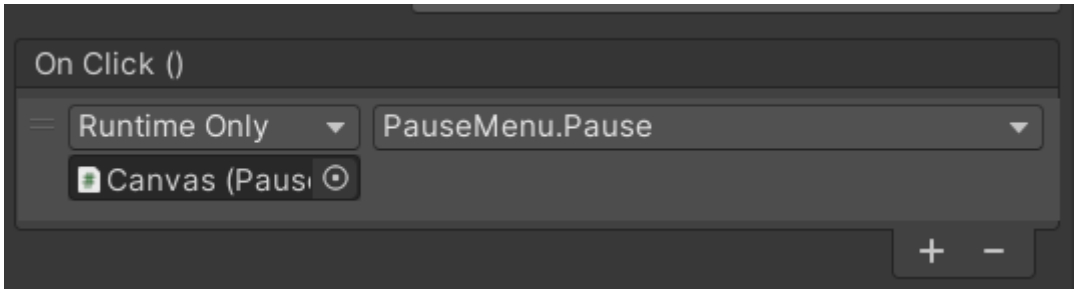

Рисунок 12 – событие On Click()

Аналогичные действия будут проделаны для всех остальных кнопок, имеющихся в игре. При нажатии на паузу отображается панель с кнопками для продолжения игры и для выхода из уровня (см. Рисунок 13). В этот момент время в игре останавливается (Time.timeScale = 0) [1].

Рисунок 13 – меню паузы

Для упрощения игры пользователям, добавим панель с подсказками по управлению персонажем, которая через некоторое время исчезает (см. Рисунок 14).

Рисунок 14 – подсказки

Чтобы успешно завершить уровень, игрок должен встать в определенную зону. Тогда появляется UI-панель завершения уровня (см. Рисунок 15). Она содержит кнопку перехода в меню выбора уровня и сообщение о том, что уровень пройден.

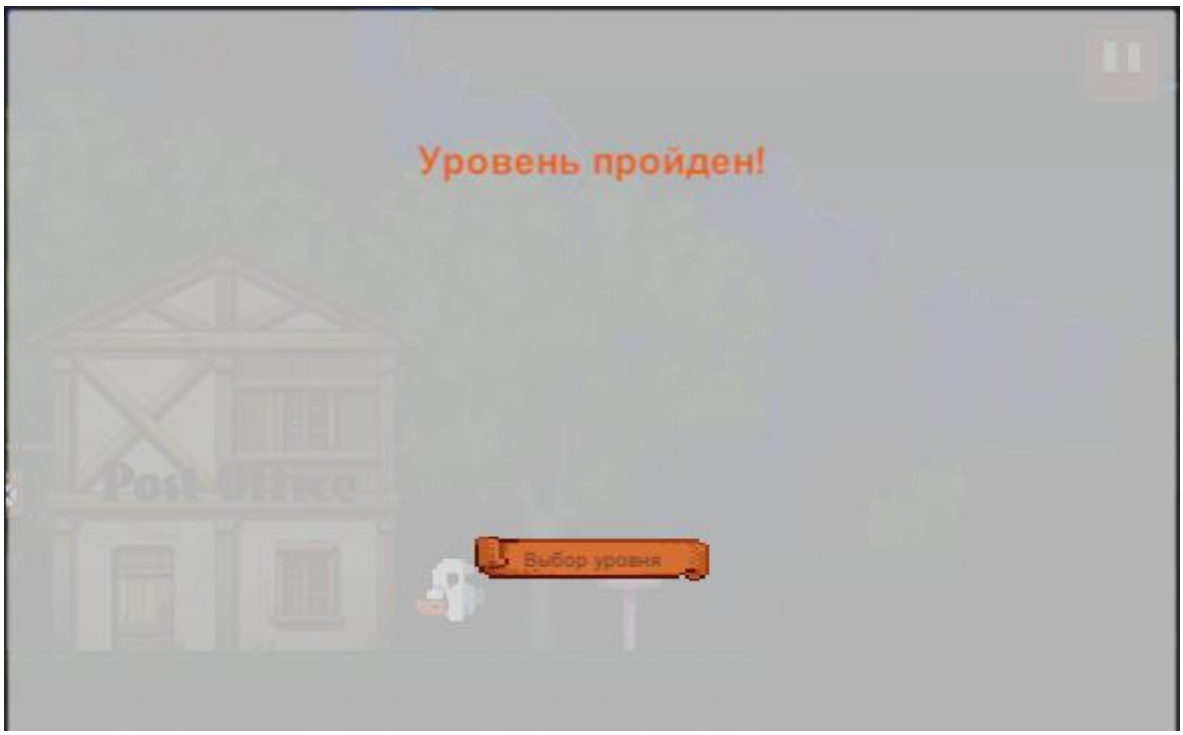

Рисунок 15 – успешное прохождение уровня

Кроме самой сцены уровня добавим еще три сцены. Главное меню, с кнопками «Играть», «Выбор уровня», «Выход» (см. Рисунок 16).

Рисунок 16 – главное меню

Экран выбора уровня, где пока что реализован только первый уровень (см. Рисунок 17).

Рисунок 17 – выбор уровня

Экран «Game Over», который появляется при неудачном завершении уровня (см. Рисунок 18).

Рисунок 18 – экран «Game Over»

Все сцены связаны между собой через SceneManager.LoadScene, а их интерфейс выполнен в одинаковом стиле, соответствующем идее игры.

Наконец, для добавления сюжета в сцену главного меню вставлена UI-панель, на которой построчно отображается текст и кнопка «Далее» (см. Рисунок 19).

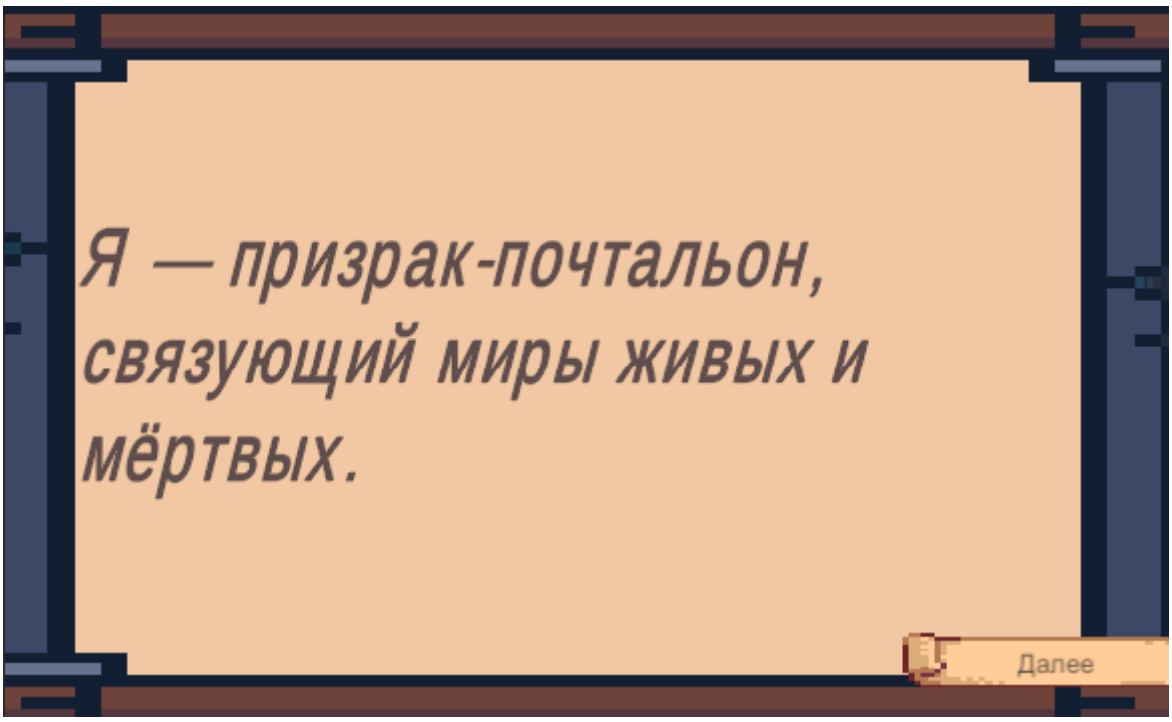

Рисунок 19 – вступительный сюжет

Код, отвечающий за отображение сюжетных вставок, представлен в приложении 5.

Также в сцену с выбором уровня по такой же схеме был добавлен диалог, освещающий идею первого уровня (см. Рисунок 20).

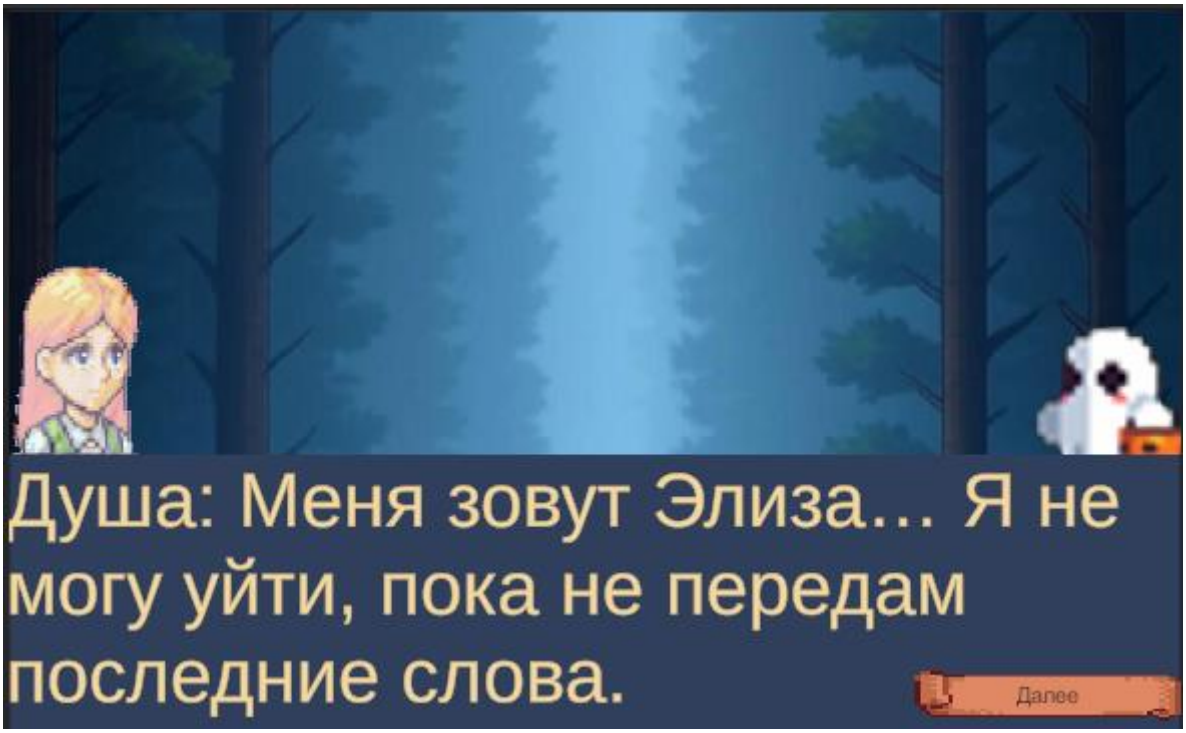

Рисунок 20 – часть сюжетного диалога для первого уровня

ЗАКЛЮЧЕНИЕ

В курсовой работе была разработана и реализована 2D-игра в жанре платформер под названием «Lost Letters». Проект реализован с помощью игрового движка Unity и языка программирования C#. Игра представляет собой небольшую сюжетную историю о призраке-почтальоне, доставляющем письма между мирами мертвых и живых.

Проект прошел ключевые этапы создания игры: анализ требований, проектирование, реализация. В ходе работы над курсовой работой было освоено создание сцен, анимаций, взаимодействие объектов, разработка боевой системы и проектирование пользовательского интерфейса. Для реализации использовались инструменты Unity: Animator для создания плавных анимаций, Canvas для удобного интерфейса и добавления сюжетных вставок, Rigidbody2D и Colider2D для реалистичной физики, Cinemachine для динамичного управления камерой и Tilemap для построения уровней [6]. Работа над проектом также позволила применить на практике теоретические знания, навыки программирования на языке C# [4].

Много внимания было уделено не только технической стороне, но и художественной части проекта. Общий визуальный стиль, плавные анимации, сюжет, эффекты параллакса и удобный интерфейс делают игровой процесс более захватывающим [5]. Такой подход добавил творчества в процесс, разбавлял рутинную работу с кодом.

Проект имеет множество возможностей для дальнейшего развития. Можно создать побольше уровней, разнообразить игровой процесс, добавив новые механики. Также есть возможность расширить сюжет, сильнее раскрыть историю главного персонажа через диалоги, сделать повествование интереснее. Помимо этого можно добавить звуковые эффекты и фоновую музыку, чтобы усилить атмосферу [1].

Еще одна интересная перспектива – выложить игру в интернет, чтобы в нее могли поиграть все желающие и оставить отзывы [7]. Это может вдохновить на создание новых идей и выпуска обновлений.

Таким образом, работа над проектом не только способствовала улучшению знаний и навыков, но и стала первым шагом к созданию полноценного игрового продукта.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гейг, М. Разработка игр на Unity 2018 за 24 часа: пошаговое руководство / Майк Гейг; пер. с англ. — Москва: Эксмо, 2019. — 480 с.
2. Гибсон, Дж. Unity и C#. Геймдев от идеи до реализации: [руководство] / Джереми Гибсон; пер. с англ. — Санкт-Петербург: Питер, 2020. — 512 с.
3. Корнилов, А. Unity. Полное руководство: [справочное издание] Андрей Корнилов. — Москва: Эксмо, 2022. — 704 с.
4. Трофимов, В. Ю. Программирование на языке C# в Unity 2021 : учебное пособие / В. Ю. Трофимов. — Москва: ДМК Пресс, 2021. — 288 с.
5. Хокинг, Дж. Unity в действии: многоплатформенная разработка с использованием C# / Джозеф Хокинг ; пер. с англ. — Москва : ДМК Пресс, 2021. — 400 с.
6. Официальная документация Unity (версия 2021+). URL: https://docs.unity3d.com/ru
7. Itch.io [Электронный ресурс]. — URL: https://itch.io
8. Unity Asset Store [Электронный ресурс]. — URL: https://assetstore.unity.com.
9. Unity Technologies. Официальная документация Unity [Электронный ресурс]. — URL: https://docs.unity3d.com/ru.

ПРИЛОЖЕНИЕ 1

```
void Update() {
   //расстояние от точки которую нужно патрулировать, до врага меньше расстояния на которое
патрулирует враг
   if (Vector2.Distance(transform.position, point.position) < positionOfPatrol && angry == false)
      chill = true; //спокойное состояние так как он просто патрулирует эту область и рядом нет
игрока
   //расстояние от врага до гг меньше дозволенного
   if (Vector2.Distance(transform.position, player.position) < stoppingDistance) {
      angry = true;
      chill = false;
      goBack = false;
   }
   if (Vector2.Distance(transform.position, player.position) > stoppingDistance) {
      goBack = true;
      angry = false;
   }
   if (angry && canAttack) {
      float distanceToPlayer = Vector2.Distance(transform.position, player.position);
      if (distanceToPlayer <= healthScript.attackRange) {
         Debug.Log("Try attack");
         TryAttack();
      }
   }
   if (moveingRight)
      transform.localScale = new Vector3(1f, 1f, 1f);
   else
      transform.localScale = new Vector3(-1f, 1f, 1f);
}
//состояния персонажа
void Chill() {
   if (transform.position.x > point.position.x + positionOfPatrol) {moveingRight = false;}
   else if (transform.position.x < point.position.x - positionOfPatrol) {moveingRight = true;}
   float moveDirection = moveingRight ? 1 : -1;
   rb.velocity = new Vector2(moveDirection * speed, rb.velocity.y);
}
void Angry() {
   if (player == null) return;
   float targetX = player.position.x;
   float newX = Mathf.MoveTowards(transform.position.x, targetX, speed * Time.deltaTime);
   rb.velocity = new Vector2((newX - transform.position.x) / Time.deltaTime, rb.velocity.y);
   if (player.position.x > transform.position.x) moveingRight = true;
   else moveingRight = false;
}
void GoBack() {
   float targetX = point.position.x;
   float newX = Mathf.MoveTowards(transform.position.x, targetX, speed * Time.deltaTime);
   rb.velocity = new Vector2((newX - transform.position.x) / Time.deltaTime, rb.velocity.y);
   if (player.position.x > transform.position.x) moveingRight = true;
   else moveingRight = false;
}
```

ПРИЛОЖЕНИЕ 2

```
public class PlayerHealth : MonoBehaviour {
    [Header("Health Settings")]
    public int currentLives;
    public int playerLives = 3; //max
    public static int numberOfRespawn = 3; //сколько раз можно умереть
    [Header("References")]
    public Animator anim;
    private bool isDead = false;
    private PlayerRespawnManager respawnManager;
    private Slider healthBar; // Статическая ссылка на HealthBar
    void Start() {
        respawnManager = GameObject.FindObjectOfType<PlayerRespawnManager>();
        currentLives =playerLives; // Инициализируем здоровье
        GameObject healthBarObject = GameObject.FindWithTag("HealthBar");
        if (healthBarObject != null) {
            healthBar = healthBarObject.GetComponent<Slider>();
            if (healthBar != null) {
                healthBar.maxValue = playerLives;
                healthBar.value = currentLives;
            }
    }
    public void TakeDamage(int damage) {
        if (isDead) return;
        currentLives -= damage;
        Debug.Log($"Player took {damage} damage, remaining lives: {currentLives}");
        if (healthBar != null) {
            healthBar.value = currentLives;
        }
        anim.SetTrigger("hit");
        if (currentLives <= 0) {
            Die();
        }
    }
    void Die() {
      if (isDead) return;
      isDead = true;
      // Включаем анимацию смерти
      anim.SetTrigger("dead");
      // Отключаем управление
      var controller = GetComponent<PlayerComtroller>();
      if (controller != null) {
          controller.enabled = false;
      }
      numberOfRespawn--;
      if (numberOfRespawn <= 0) {
          Invoke(nameof(LoadGameOverScene),2f);
      }
      else {
          if (respawnManager != null) {
              respawnManager.OnPlayerDeath();
          }
          Destroy(gameObject, 1.5f); // игрок исчезнет — затем появится новый
      }
    }
}
```

ПРИЛОЖЕНИЕ 3

```
public class EnemyHealth : MonoBehaviour {
    public int maxHealth = 3;
    private int _currentHealth;
    private bool _isDead = false;

    private Animator _anim;

    public float attackCooldown = 1f;
    private float _lastAttackTime = 0f;
    public int damageToPlayer = 1;
    public float attackRange = 2f;

    public bool isKnockbacking = false;
    private float knockbackTimer = 0f;
    private const float knockbackDuration = 0.2f;

    void Start() {
        _currentHealth = maxHealth;
        _anim = GetComponent<Animator>();
    }
    public void TakeDamage(int damage, Vector2 attackDirection) {
        if (_isDead) return;
        Debug.Log("Attack");
        _currentHealth -= damage;
        _anim.SetTrigger("hit");
         Knockback(attackDirection);
        if (_currentHealth <= 0) {
            Invoke(nameof(Die), 0.15f); //подождем пока отлетит
        }
    }
    void Die() {
        if (_isDead) return;
        Debug.Log("die");
        _isDead = true;
        _anim.SetTrigger("Dead");
        GetComponent<Collider2D>().enabled = false;
        Rigidbody2D rb = GetComponent<Rigidbody2D>();
        rb.velocity = Vector2.zero;
        rb.constraints = RigidbodyConstraints2D.FreezeAll;
        Destroy(gameObject, 1.5f); // Уничтожаем через 2 секунды

    }
    void Knockback(Vector2 attackDirection) {
        Rigidbody2D rb = GetComponent<Rigidbody2D>();
        // Сброс скорости и толчок
        rb.velocity = Vector2.zero;
        rb.AddForce(attackDirection.normalized * 5000f, ForceMode2D.Impulse);
        // Отключить патруль и ИИ
        GetComponent<PatrolerEnemy>().enabled = false;
        // Включить обратно через 0.2 сек
        Invoke(nameof(EnableAI), 0.2f);
    }
}
```

ПРИЛОЖЕНИЕ 4

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class PauseMenu : MonoBehaviour
{
    public bool pauseGame;
    public GameObject pauseGameMenu;
    void Update()
    {
        if(Input.GetKeyDown(KeyCode.Escape)) {
            if(pauseGame) {
                Resume();
            }
            else {
                Pause();
            }
        }

    }
    public void Resume() {
        pauseGameMenu.SetActive(false); //панель паузы неактивна
        Time.timeScale = 1f; //игра в норм режиме
        pauseGame = false; //игра не на паузе
    }
    public void Pause() {
        pauseGameMenu.SetActive(true);
        Time.timeScale = 0f;
        pauseGame = true;
    }
}
```

ПРИЛОЖЕНИЕ 5

```
using TMPro;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;
public class GameManager : MonoBehaviour
{
    public GameObject gameIntroPanel;    // Панель с описанием игры
    public TextMeshProUGUI introText;    // Текст описания
    public Button continueButton;        // Кнопка "Далее"

    private string[] introLines = new string[]
    {
        "Я — призрак-почтальон, связующий миры живых и мёртвых.",
        "Моя задача — доставлять письма, которые несут последние слова, полные любви и боли.",
        "Я помогаю душам обрести покой, но этот путь полон слёз и опасностей...",
        "Ты готов отправиться в это путешествие?"
    };
    private int currentLineIndex = 0;

    private void Start() {
        // Скрываем панель в начале
        if (gameIntroPanel != null) {
            gameIntroPanel.SetActive(false);
        }

        // Настраиваем кнопку "Далее"
        if (continueButton != null) {
            continueButton.onClick.AddListener(ShowNextIntroLine);
        }
    }
    void StartIntro() {
        if (gameIntroPanel != null) {
            gameIntroPanel.SetActive(true);
        }
        currentLineIndex = 0;
        ShowNextIntroLine();
    }
    void ShowNextIntroLine() {
        if (currentLineIndex < introLines.Length) {
            introText.text = introLines[currentLineIndex];
            currentLineIndex++;
        }
        else {
            // Переходим на сцену выбора уровня
            ChooseLvl();
        }
    }

}
```