МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра Прикладная математика и информатика
(наименование)

02.03.03 Математическое обеспечение и администрирование информационных систем
(код и наименование направления подготовки / специальности)

Мобильные и сетевые технологии
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему: «Разработка однопользовательской 2Д игры в среде Unity»

Обучающийся Д. А. Пятков
(Инициалы Фамилия) (личная подпись)

Руководитель к.п.н., доцент, О.В. Оськина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант к.п.н., доцент, С.А. Гудкова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2024

Аннотация

Тема бакалаврской работы: «Разработка однопользовательской 2Д игры в среде Unity».

В работе проводится математический анализ, проектирование, реализация и тестирование игрового функционала, который включает в себя передвижение игрока, систему сражений, состоящую из вооружения игрока и врагов, а также динамическое соединение игровых локаций.

Для достижения целей рассматриваются различные подходы, шаблоны и методологии, по типу Model-View-Controller, Dependency Injection, твининг.

В работе проводится сравнение нескольких игровых движков, включающих в себя Unreal Engine, Godot Engine, Unity.

Работа может быть разделена на следующие логически взаимосвязанные части: анализ поставленных задач, построение математических моделей, проектирование и выделение архитектурных особенностей решения, реализация и тестирование разработанного функционала.

Бакалаврская работа состоит из 45 страниц текста, 37 рисунков и 21 источника.

Abstract

Theme of the bachelor’s work: «Development of a singleplayer 2D game in the Unity engine».

The paper includes mathematical analysis, design, implementation and testing of game functionality, which includes player movement, a battle system consisting of weaponry and enemies, as well as dynamic connection of in-game locations.

Various approaches, patterns and methodologies are considered in order to achieve the goals. These include model-view-controller, dependency injection and tweening.

The paper includes a comparison of several game engines such as Unreal Engine, Godot Engine, Unity.

The paper can be divided into the following logically interconnected parts: analysis of the assigned tasks, creation of mathematical models, design and architecture, realization and testing of implemented functionality.

The paper consists of 45 pages of text, 37 figures and 21 sources.

Оглавление

Введение

Видеоигры являются довольно распространённым способом проведения времени для самых разных поколений. Свою популярность они начали набирать в начале 1980-х годов, а сегодня всё также остаются релевантной и развивающейся формой развлечения, стремительно вытесняя другие формы по типу чтения книг и просмотра фильмов или сериалов. Об этом свидетельствует факт того, что в период с 2015 по 2022 год количество геймеров в мире выросло на один миллиард человек, со средним ростом в приблизительно 5.5% в год [12]. Для некоторых видеоигры также постепенно перешли из досуга в источник дохода с ростом современного вида спорта – киберспорта. Все эти факты безусловно говорят о том, что игры имели и сохраняют актуальность в современном мире.

Целью выпускной квалификационной работы является создание функционала для 2Д игры в жанрах roguelike и hack’n’slash. Упомянутый игровой функционал включает в себя:

- передвижение игрока
- враги, которые двигаются в сторону игрока
- динамическое соединение игровых локаций

Глава 1 Задачи выпускной квалификационной работы

1.1 Постановка технического задания на разработку игровых механик игры

Требования к игровому функционалу:

- реализация простого передвижения игрока
- реализация способности рывка игрока
- реализация простых врагов
- реализация системы боя
- реализация соединений между комнатами

Игрок может передвигаться в 8 направлениях и совершать рывок, который позволяет ему быстрее преодолевать короткие расстояния, но при этом ненадолго тормозить в конце.

Враги будут стоят на месте и активироваться при приближении игрока, после чего они будут передвигаться в его сторону.

Игровой мир состоит из определённых локаций, которые называются комнатами. Изначально комнаты закрыты, но могут быть динамически соединены при помощи определённых алгоритмов.

1.2 Теоретический анализ поставленных задач

Зададим вектор передвижения игрока как $\vec{c}$. Векторы движения игрока по горизонтали и вертикали соответственно будут $\vec{x}$ и $\vec{y}$. В этом случае мы можем задать вектор $\vec{c}$ как:

$$\vec{c}^2 = \vec{x}^2 + \vec{y}^2 \quad (1)$$

Формула (1) обязательна, так как если просто использовать векторы горизонтального и вертикального движения, игрок будет всегда передвигаться быстрее по диагонали нежели в одном направлении.

При совершении рывка, игрок будет терять контроль над управлением до тех пор, пока рывок не будет полностью завершён.

Враги имеют очень простую логику. Они стоят на месте и начинают двигаться в сторону игрока, если он входит в определённую область вокруг врага, которая определяется следующим образом:

$$(x - x_0)^2 + (y - y_0)^2 \leq R^2 \quad (2)$$

Формула (2) определяет, находится ли игрок в определённом радиусе от врага. Если уравнение истинно, враг начинается двигаться в сторону игрока.

Комнаты могут соединяться из любого направления в любое направление, то есть, например, может быть соединение комнаты из выхода с востока, к комнате с выходом, смотрящим на север. Всего это образует 16 сочетаний.

Глава 2 Проектирование и архитектура

2.1 Игровой движок

Для разработки игр в большинстве случаев используются так называемые игровые движки. Игровой движок – это специальное программное обеспечение, позволяющее разграничить функционал игры и фундаментальные задачи по типу создания окна игры, вывод графики на экран и так далее [1]. Функционал игры ложится на разработчика, в то время как остальные задачи решаются игровым движком.

Unity – самый популярный движок для мобильной и 2D разработки, но также может использоваться и для 3D разработки [3]. В мобильном секторе доля Unity составляет около 70 процентов [14]. Анонс и выход игрового движка в мир состоялся в 2005 году. Unity считается лёгким для освоения новичками, ввиду чего является популярным выбором для инди игр. Unity имеет очень обширный функционал, что позволяет разработчикам сфокусироваться на разработке игр используя по большей части только встроенные инструменты. Для создания скриптов в Unity можно использовать JavaScript и C#.

Unreal Engine – группа игровых движков, преимущественно использующийся для 3D разработки, но 2D разработка на движке также возможна [18]. Первый раз Unreal Engine был показан миру в 1998 году. Он использовался для разработки игры Unreal. Изначально движок был создан для разработки шутеров от первого лица, но с течением времени был адаптирован для других жанров игр и даже других сфер деятельности. В 2014 году Разработка в Unreal Engine осуществляется при помощи “чертежей”. Язык чертежей – C++.

Godot Engine – кроссплатформенный игровой движок в открытом доступе [11]. Используется преимущественно для 2D разработки, но также поддерживает и 3D разработку. Публичный выход движка состоялся в 2014

году, но он был разработан в 2001 году и использовался компаниями в Латинской Америке. Считается, что движки Godot и Unity во многом похожи, но Unity, ввиду своей более долгой истории, имеет более обширный функционал. Для разработки на Godot можно использовать GDScript , C# и C++.

Таблица 1 – Краткая характеристика игровых движков

Движок	**Язык**	**Дата выхода**	**Основные Сферы**
Unity	C#, JavaScript	2005	2D, 3D, мобильная
Unreal Engine	C++	1998	3D, фильмы и телевидение
Godot	C#, GDScript, C++	2014	2D

На основе сравнения таблицы 1 был сделан выбор в сторону Unity, так как в данной работе затрагивается разработка 2D проекта. Его более богатый функционал в этой сфере делает его более предпочтительным по сравнению с другими движками.

2.2 Подходы, методики, технологии

В процессе архитектуры и проектирования были выбраны определённые подходы и паттерны, чтобы достичь максимальной модульности и расширяемости. Архитектурные паттерны накладывают определённые ограничения на структуру кода и разделение логики, что обеспечивает чистоту кода и его поддерживаемость [5].

Dependency Injection – методика, при которой объект получает свои зависимости извне вместо того, чтобы создавать их самому [6]. Этот подход стремится разделить логику создания объектов и их использования, что ведёт к низкой связанности. Данная методика обеспечивает факт того, что объект-

клиент не знает, как создавать свои зависимости. Она также делает неясные зависимости ясными.

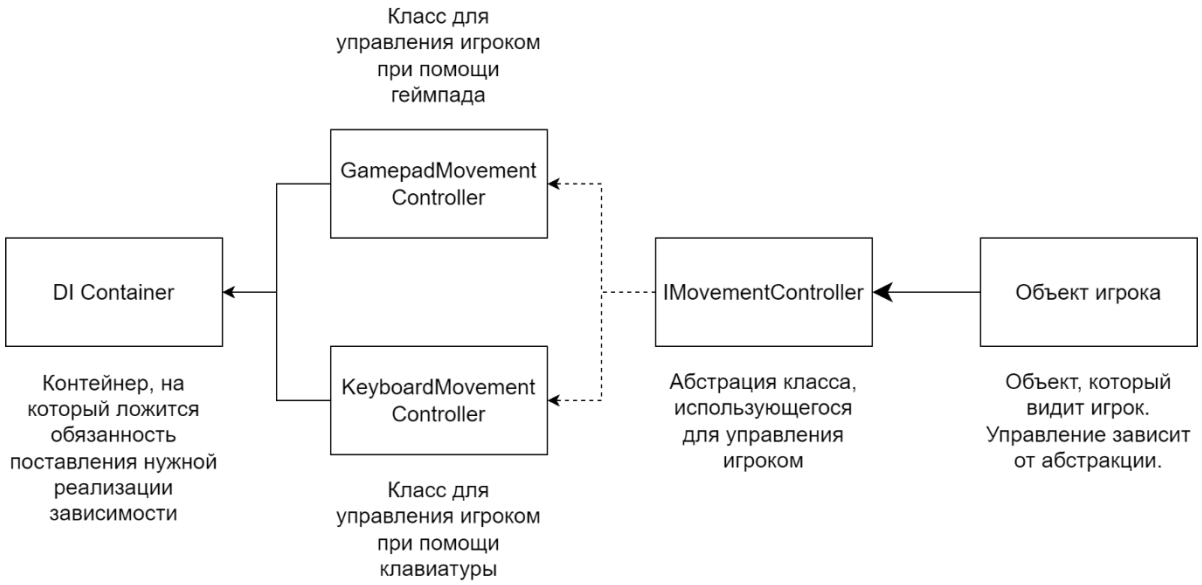


Рисунок 1 – Использование Dependency Injection в передвижении игрока

Как можно видеть на рисунке 1, игровой объект зависит от абстракции, которая в свою очередь может иметь нужную реализацию. Нужная реализация поставляется игровому объекту извне при помощи так называемого Dependency Injection контейнера. В Unity для реализации подхода Dependency Injection есть пакет Zenject. Это легковесный, высокопроизводительный фреймворк в открытом доступе, созданный специально для реализации данного подхода в Unity [21].

Следующий подход, который использовался для построения архитектуры игры – Model-View-Controller, сокращённо MVC. Он позволяет разделить логику отдельного функционала на три основные составляющие – модель, представление и контроллер [2]. Модель хранит в себе все данные и содержит бизнес логику, представление отображает изменения модели, контроллер является связующей точкой между моделью и представлением, а также обрабатывает пользовательский ввод.

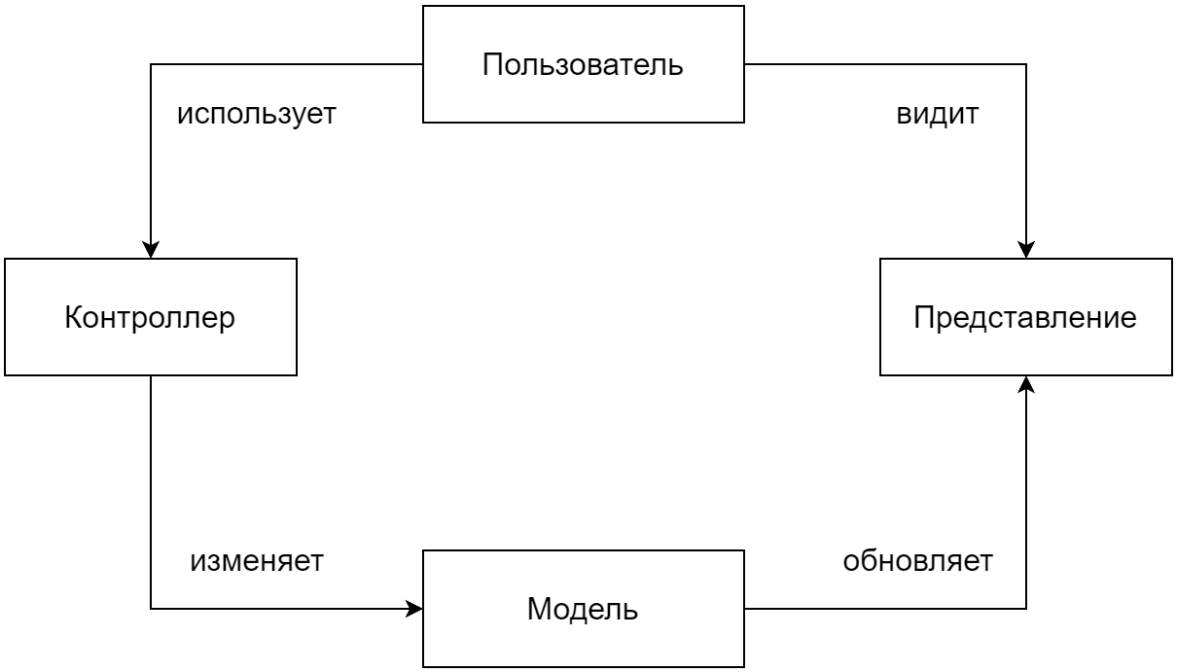


Рисунок 2 – Общая структура MVC

Рисунок 2 показывает общий случай структурирования отдельного функционала при помощи MVC. Контроллер обрабатывает пользовательский ввод, на основе которого изменяет модель. Модель при помощи события обновляет представление. Стоит отметить, что модель не знает о представлении. Представление получает изменения модели при помощи шаблона подписки-публикации.

MVC в Unity имеет некоторые особенности. В виде представления выступают сами игровые объекты. Контроллер не может быть самостоятельным, так как для проверки пользовательского ввода он должен вызываться каждый кадр, что возможно только в представлении. Контроллер также обрабатывает некоторые события, которые могут поступать только с представления, например, коллизии объектов [4]. Структура MVC в Unity представлена на рисунке 3.

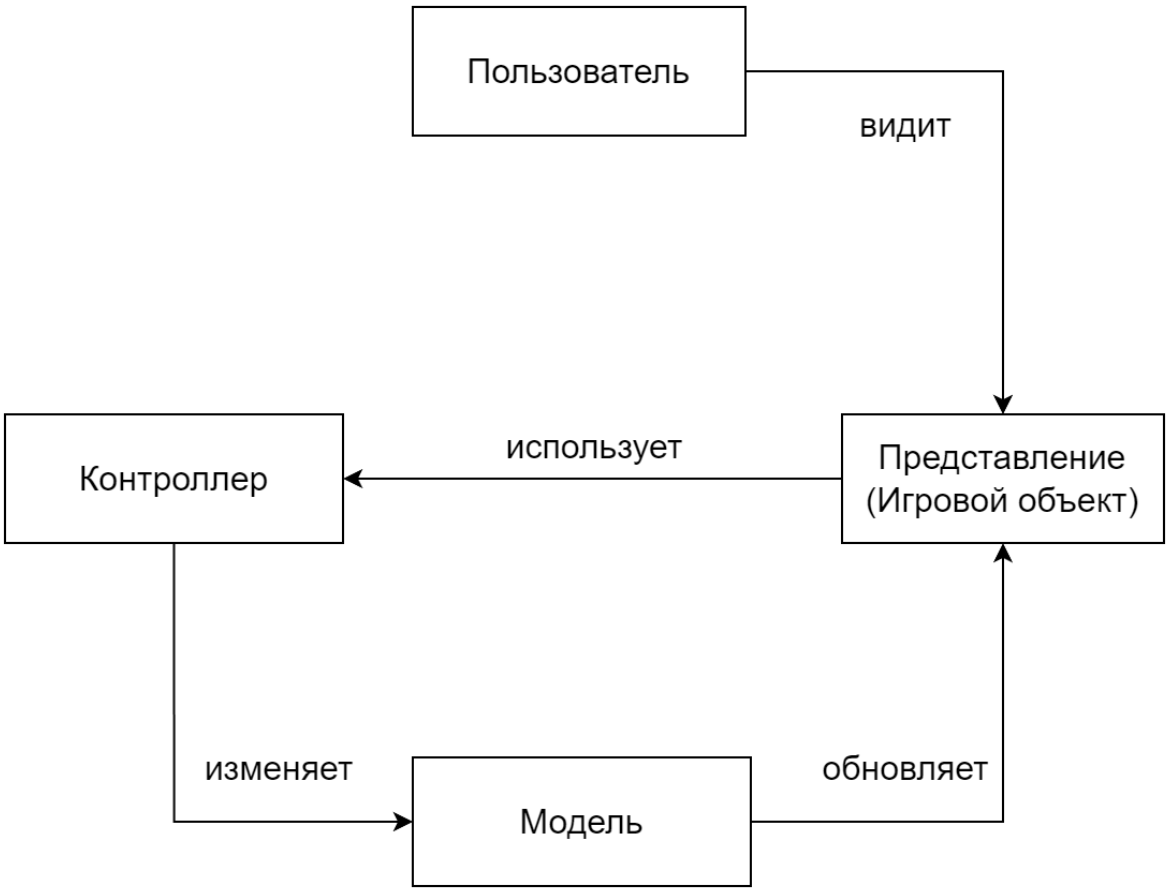


Рисунок 3 – Структура MVC в Unity

Как можно заметить на рисунке 3, пользователь больше не использует контроллер напрямую. Контроллер на каждом кадре обрабатывает пользовательский ввод при помощи особых классов, предоставленных Unity, например, класс Input [17].

Отдельно стоит упомянуть особую технику, которая называется “tweening”. Она позволяет создавать промежуточные кадры анимаций для иллюзии плавного движения [20]. Для того, чтобы сделать эти анимации нелинейными, используются различные функции сглаживания.

Рисунок 4 – Графики функций сглаживания [8]

На рисунке 4 представлены графики различных функций сглаживания от 0 до 1. Они придают анимациям нелинейность, то есть, анимируемый объект в разные моменты времени будет передвигаться и вращаться с разной скоростью, в зависимости от того, сколько времени прошло от начала анимации. Это можно представить следующим образом: прошло 100 мс от анимации длительностью 200 мс. Тогда на вход функции сглаживания будет поступать 0.5, а на выходе мы получим значение её завершённости или текущего состояния.

Самое популярное решение для твининга анимаций в Unity – DoTween [7]. В данном игровом проекте это используется для, например, анимации атак оружий. В общих чертах самую обычную атаку можно представить как взмах, удар и возвращение в исходное состояние. В виде твина это может выглядеть следующим образом:

а) Изменить относительное положение оружия на (-0.8; -0.3) и относительный поворот на 50 градусов со сглаживанием OutSine за 200 миллисекунд (взмах)
б) Изменить относительное положение оружия на (0.8; 0.3) и относительный поворот на -30 градусов со сглаживанием OutSine за 100 миллисекунд (удар)
в) Изменить относительное положение оружия на (0; 0) и относительный поворот поворот на 0 градусов со сглаживанием OutSine за 150 миллисекунд (возвращение в исходное положение)

Ещё один шаблон, который применяется в работе – конечный автомат, либо же машина состояний. Объект в любой момент времени может находиться в каком-либо состоянии, которое определяет его поведение [10]. Например, игрок может быть в состоянии ходьбы, в котором он может атаковать, но также может быть в состоянии плавания, в котором он не может атаковать. Этот шаблон полезен, так как он позволяет избежать избыточных проверок и добиться модульности решения.

В работе также применяется механизм пулинга объектов. Он позволяет повторно использовать часто применяющиеся объекты вместо создания новых [13]. Например, полупрозрачные копии игрока, которые появляются при рывке. Вместо того, чтобы удалять объект, он деактивируется, затем при надобности активируется снова с новыми параметрами (позицией, поворотом и так далее). Это позволяет не тратить ресурсы движка на создание новых объектов.

2.3 Проектирование передвижения

Передвижение игрока состоит из простого управления и рывков. Всего у игрока можно выделить три основных состояния:

- Состояние отдыха, в котором игрок стоит на месте
- Состояние передвижения, в котором игрок сам управляет игроком

– Состояние рывка, в котором игрок совершает короткий рывок

Разделение на состояния полезно, так как оно позволяет достичь следующих ограничений без избыточных проверок:

– Игрок может совершать рывок только если он находится в состоянии движения
– Игрок не может передвигаться и не может совершать рывок если он находится в состоянии рывка
– Игрок может атаковать только в состоянии движения и состоянии отдыха

Состояние передвижения можно охарактеризовать диаграммой, изображённой на рисунке 5

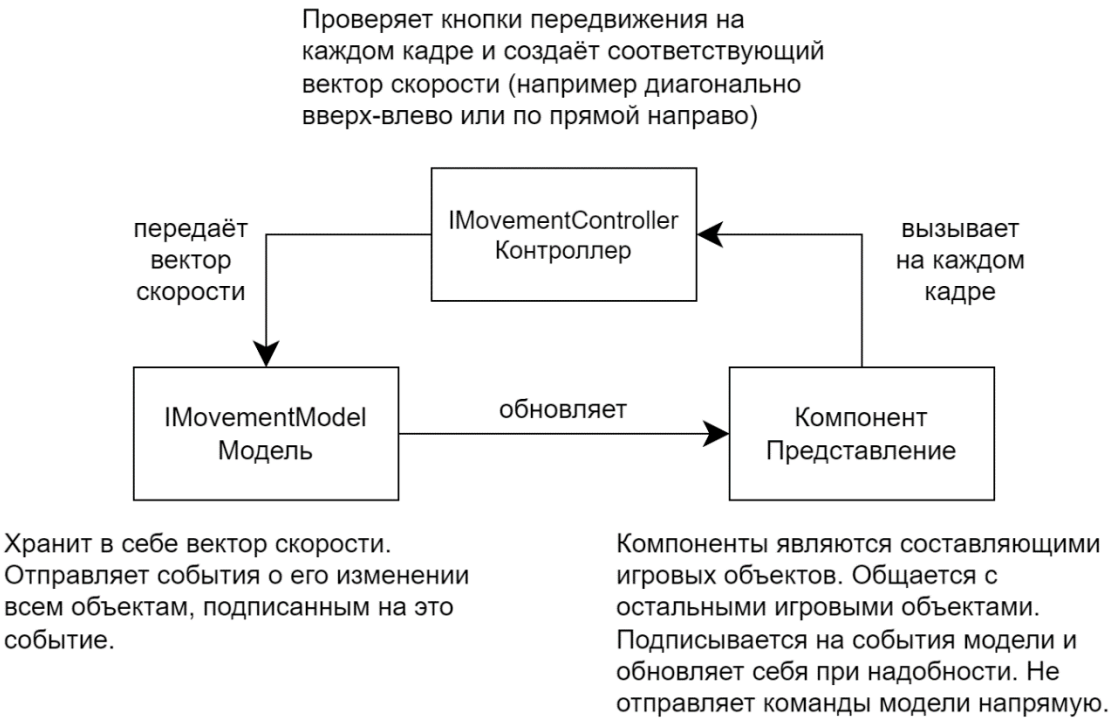


Рисунок 5 – Диаграмма архитектуры передвижения игрока

На рисунке 5 показано построение передвижения игрока. Контроллер на основе ввода пользователя создаёт вектор передвижения, длина которого всегда равна 1. Это достигается при помощи формулы 1.

Рывок при помощи твининга плавно перемещает игрока на определённую дистанции. Это простое перемещение объекта из одной точки в другую за определённое время со сглаживанием OutSine. Параллельно также создаётся последовательность твинов, которые создают полупрозрачные копии спрайта игрока для иллюзии убыстрённого движения.

2.4 Проектирование врагов

Самые простые враги в игре активируются при приближении игрока, а остальное время стоят на месте. В случае данной работы самые простые враги – это слизни. Они состоят из двух MVC модулей:

- Получение урона сущностями
- Передвижение

И имеют два основных состояния:

- Живой
- Умирающий

Передвижение слизней использует те же классы, что и используются для передвижения игрока. Единственная разница в контроллере. Для игрока контроллер – это клавиатурное управление, а для слизня – проверка расстояния до игрока и активация при выполнении условия.

Диаграмма работы модуля получения урона сущностями отображена на рисунке 6.

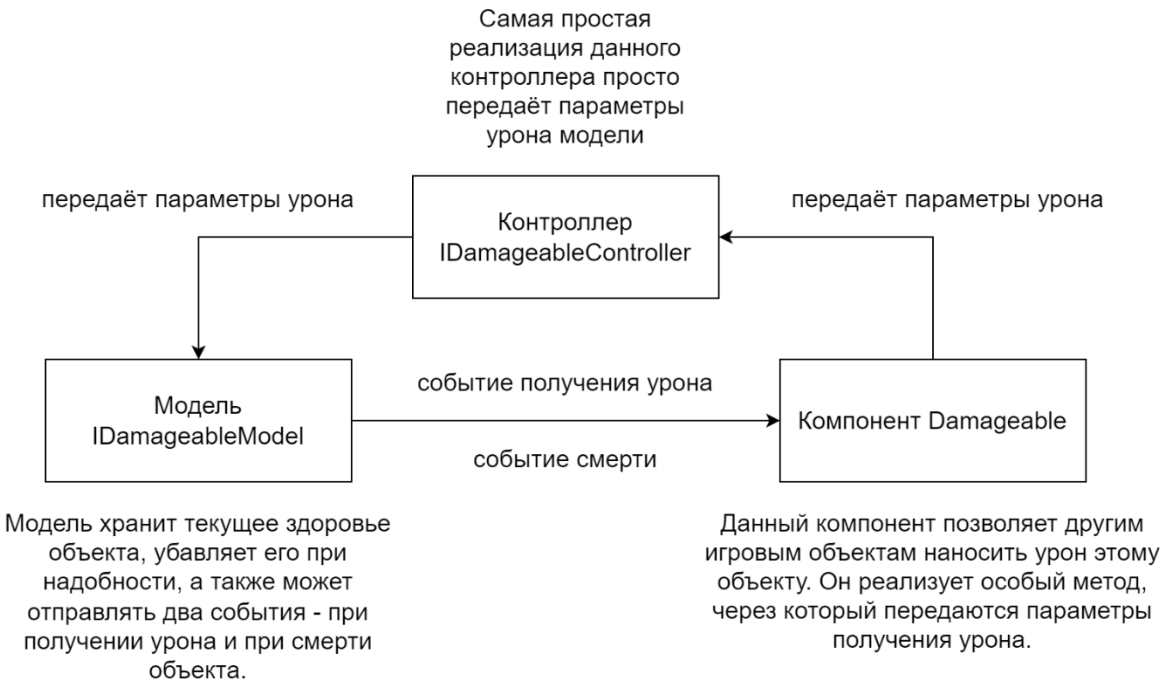


Рисунок 6 – MVC получения урона сущностями

На рисунке 6 представлен цикл работы MVC модуля получения урона. Параметры получения урона проходят по всем составляющим. В более сложной версии данного модуля было бы возможно, например, усиление отдачи моделью, либо же игнорирование параметров получения урона контроллером (например, если сущность может становиться неуязвимой). Представление получает два события – о получении урона и о смерти. При получении урона враг резко загорается белым, что даёт игроку знать о попадании. При смерти враг постепенно исчезает.

Игрок может наносить урон врагам при помощи оружий. Ранее был разобран пример твининга анимации атаки оружия. Во время анимации оружие также переходит в разные состояния, которые, как и в других случаях, задают определённое поведение. Диаграмма переходов состояний обычной атаки первоначального оружия игрока изображена на рисунке 7.

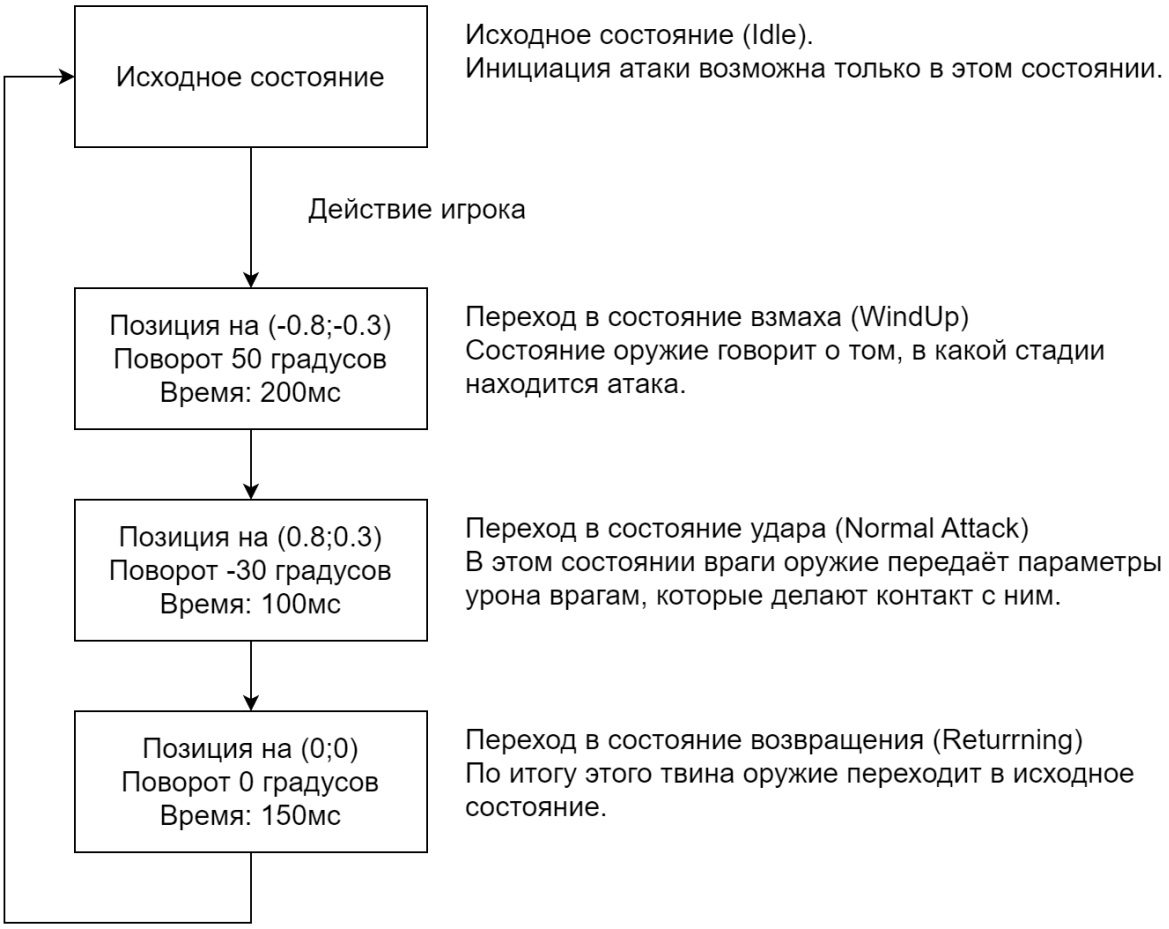


Рисунок 7 – Твининг атаки

На рисунке 7 изображён процесс атаки первоначального оружия. Все позиции и повороты являются относительными к родительскому объекту. Между игроком и оружием есть промежуточный объект, который позволяет задавать направление и позицию. Это позволяет с лёгкостью анимировать объекты в любом направлении.

2.5 Проектирование комнат и их соединений

Комнаты являются одним из двух ключевых составляющих генерации игрового мира. Второй составляющей являются соединения между комнатами. Вместе они полноценно образуют игровой мир, который состоит

из так называемых тайлов [15]. То есть, весь мир можно поделить на двумерный массив, и каждый тайл содержит свой особый спрайт [16].

Каждая комната имеет один или несколько выходов в любом направлении (восток, запад, север, юг). Комнаты соединятся в зависимости от позиций выходов и их направлений. На рисунке 8 приведён пример двух комнат с выходами.

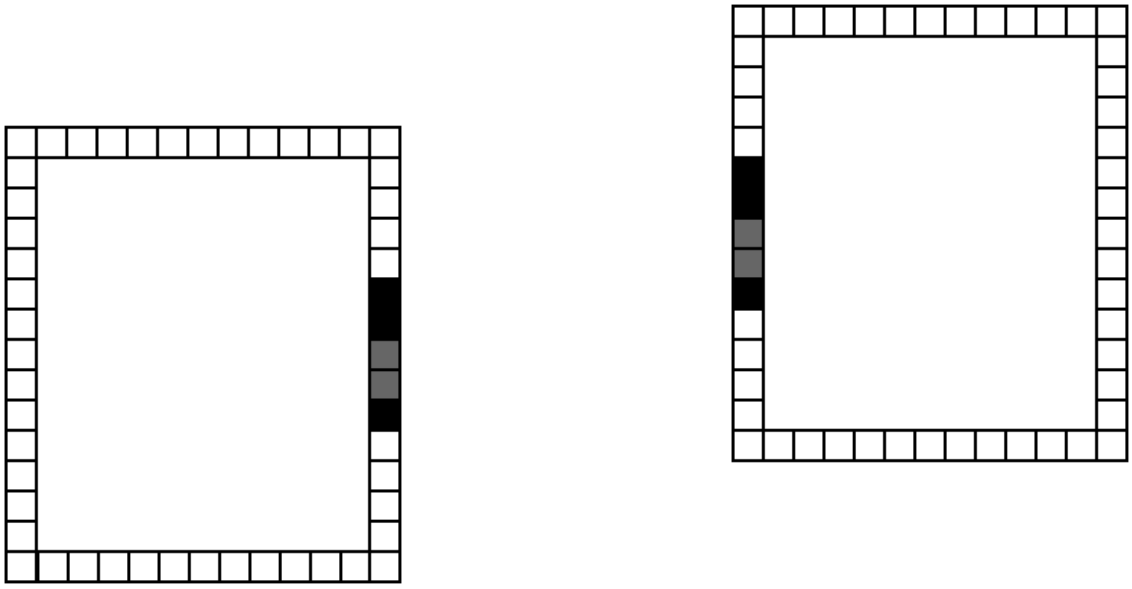


Рисунок 8 – Пример двух комнат с выходами

На рисунке 8 изображены границы двух комнат. Чёрным отмечены границы выходов, а серым – часть, по которой можно будет ходить в случае построения соединения через выход. Соединение этих двух комнат представлено на рисунке 9.

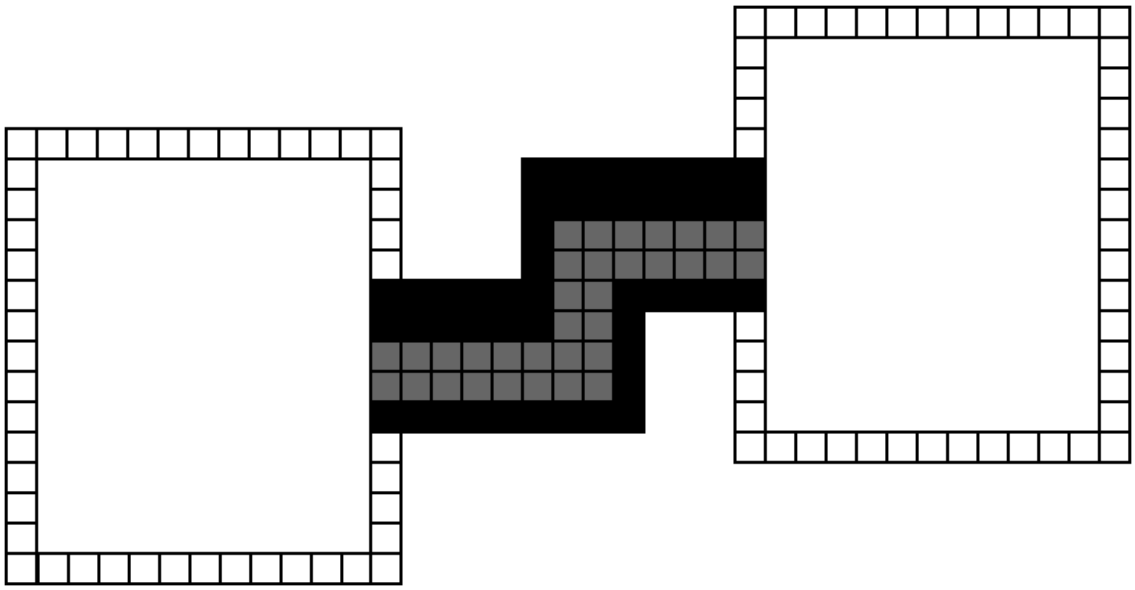


Рисунок 9 – Соединение между двумя комнатами

На рисунке 9 показано соединение восток-запад. По типу его можно отнести к соединению с выходами, смотрящими в противоположные стороны. Структурно соединение из рисунка 9 можно поделить на следующие составляющие (слева направо):

- Выход 1
- Коридор 1
- Поворот
- Коридор 2
- Выход 2

Как было упомянуто ранее, всего есть 16 сочетаний между направлениями выходов, которые можно поделить на три типа:

- Соединение комнат с противоположно смотрящими выходами
- Соединение комнат с выходами смотрящих в одну сторону
- Соединение комнат с выходами на разных параллелях

Глава 3 Реализация и тестирование игрового функционала

3.1 Разработка передвижения игрока

На уровне абстракций передвижение сущностей можно представить двумя интерфейсами, представленными на рисунках 10 и 11.

```
using UnityEngine;

public delegate void OnMoveHandler(Vector2 velocity);

public interface IMovementModel
{
    float MovementSpeed { get; }

    Vector2 Velocity { get; }

    event OnMoveHandler OnMove;

    void ChangeVelocity(Vector2 vector);
}
```

Рисунок 10 – Интерфейс IMovementModel

На рисунке 10 представлен интерфейс модели передвижения сущностей. Она хранит в себе вектор движения, скорость движения (так как вектор движения всегда имеет длину 1), событие OnMove, которое отправляется при изменении вектора движения, а также метод ChangeVelocity, который используется контроллером для передачи нового вектора движения. События в C# реализуют модель подписка-публикация [9], которая идеальна для MVC.

```
public interface IMovementController
{
    void Update();
}
```

Рисунок 11 – Интерфейс IMovementController

На рисунке 11 представлен довольно минимальный интерфейс контроллера передвижения. Функция Update на каждом кадре вызывается представлением.

На данный момент в проекте есть лишь одна реализация модели движения.

```
using System;
using UnityEngine;

public class SimpleMovementModel : IMovementModel
{
    public float MovementSpeed => _settings.MovementSpeed;

    public Vector2 Velocity { get; private set; }

    private readonly Settings _settings;

    public SimpleMovementModel(Settings settings)
    {
        _settings = settings;
    }

    public event OnMoveHandler OnMove;

    public void ChangeVelocity(Vector2 vector)
    {
        Velocity = vector.normalized;
        OnMove?.Invoke(vector);
    }

    [Serializable]
    public class Settings
    {
        public float MovementSpeed;
    }
}
```

Рисунок 12 – Реализация SimpleMovementModel

На рисунке 12 представлена минимальная реализация модели передвижения. Сериализуемый класс Settings позволяет задавать скорость передвижения сущности при помощи внешних объектов. Он поставляется извне при помощи Dependency Injection.

```
using UnityEngine;

public class KeyboardMovementController : IMovementController
{
    readonly IMovementModel _model;

    public KeyboardMovementController(IMovementModel model)
    {
        _model = model;
    }

    public void Update()
    {
        float hor = 0;
        float ver = 0;

        if (Input.GetKey(KeyCode.D))
            hor = 1;
        else if (Input.GetKey(KeyCode.A))
            hor = -1;

        if (Input.GetKey(KeyCode.W))
            ver = 1;
        else if (Input.GetKey(KeyCode.S))
            ver = -1;

        var vector = Vector3.ClampMagnitude(new Vector3(hor, ver, 0), 1);
        _model.ChangeVelocity(vector * _model.MovementSpeed);
    }
}
```

Рисунок 13 – Реализация контроллера клавиатурного

На рисунке 13 изображён контроллер клавиатурного управления. Он формирует вектор движения в зависимости от ввода игрока (управление WASD), затем вектор укорачивается до длины 1 при помощи ClampMagnitude.

В качестве представления используется главный объект сущности. Это объект PlayerView, который при помощи отдельных состояний управляет передвижением. Реализация компонента PlayerView изображена на рисунке 13.

```
using UnityEngine;
using Zenject;

public class PlayerView : MonoBehaviour
{
    private PlayerStateManager _stateManager;
    public Rigidbody2D Rigidbody { get; private set; }

    [Inject]
    public void Init(PlayerStateManager stateManager)
    {
        _stateManager = stateManager;
    }

    private void Start()
    {
        Rigidbody = GetComponent<Rigidbody2D>();
    }

    void Update()
    {
        _stateManager.Update();
    }
}
```

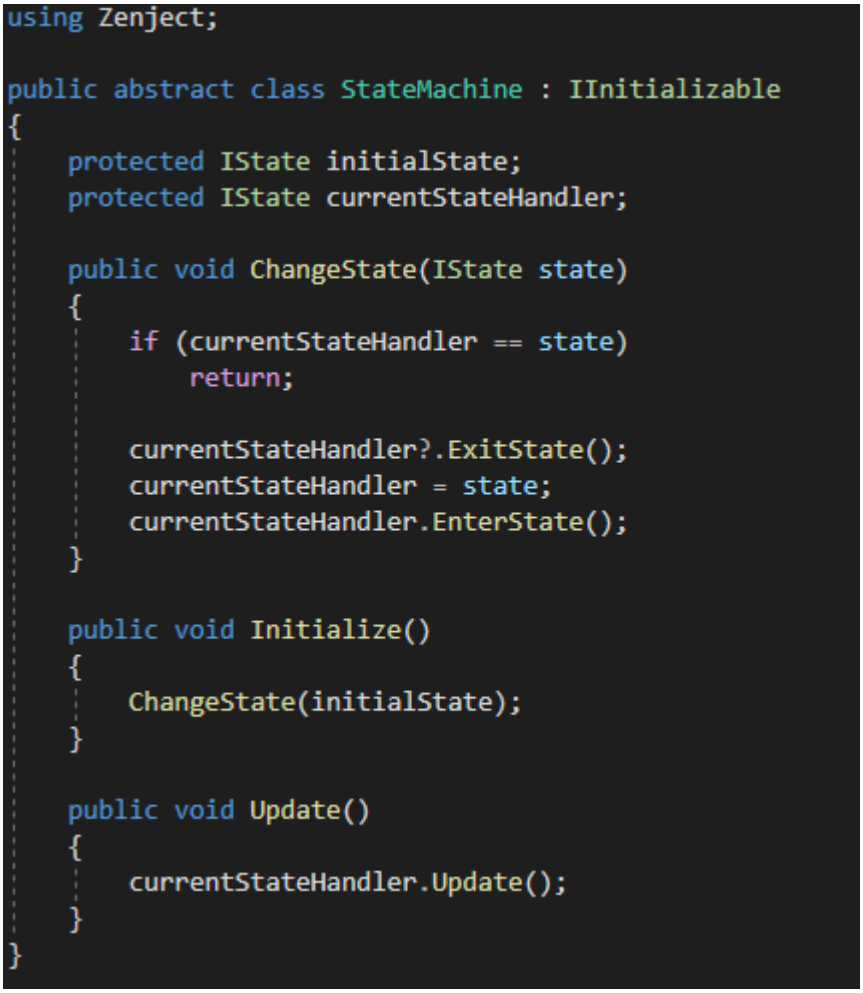
Рисунок 13 – Реализация компонента PlayerView

Компонент PlayerView, изображённый на риснуке 14 довольно минимальный, так как основная логика поведения содержится в классах состояний. На каждом кадре представление обновляет машину состояний игрока – PlayerStateManager.

```
using Zenject;

public class PlayerStateManager : StateMachine
{
    public PlayerIdleState Idle { get; private set; }
    public PlayerMovingState Moving { get; private set; }
    public PlayerDashingState Dashing { get; private set; }

    [Inject]
    public void Inject(
        PlayerIdleState idle,
        PlayerMovingState moving,
        PlayerDashingState dashing
        )
    {
        Idle = idle;
        Moving = moving;
        Dashing = dashing;

        initialState = idle;
    }
}
```

Рисунок 15 – Реализация PlayerStateManager

На рисунке 15 изображена реализация машины состояний игрока. Как было сказано ранее, игрок имеет 3 состояния. Класс PlayerStateManager наследует от абстрактного класса StateMachine, который является основой для любого класса машины состояний. Его реализация показана на рисунке 15.



```
using Zenject;

public abstract class StateMachine : IInitializable
{
    protected IState initialState;
    protected IState currentStateHandler;

    public void ChangeState(IState state)
    {
        if (currentStateHandler == state)
            return;

        currentStateHandler?.ExitState();
        currentStateHandler = state;
        currentStateHandler.EnterState();
    }

    public void Initialize()
    {
        ChangeState(initialState);
    }

    public void Update()
    {
        currentStateHandler.Update();
    }
}
```

Рисунок 16 – Реализация класса StateMachine

На рисунке 16 изображён основный класс для машин состояний. Класс инициализируется при помощи Dependency Injection контейнера, так как Zenject имеет особые интерфейсы для подобных задач. На каждом кадре при помощи функции Update вызывается обновление текущего состояния. При изменении состояния вызываются соответствующие функции каждого из состояний, так как они могут иметь дополнительную логику для входа и

выхода из них. Отдельные состояния игрока отображены на рисунках 17, 18 и 19 и 20.

```
using UnityEngine;

public class PlayerIdleState : IState
{
    private readonly IMovementModel _movementModel;
    private readonly IMovementController _movementController;
    private readonly IWeaponInputController _weaponController;
    private readonly ProximityItemController _itemPickupController;
    private readonly PlayerStateManager _stateManager;

    public PlayerIdleState(
        IMovementModel movementModel,
        IMovementController movementController,
        IWeaponInputController weaponController,
        PlayerStateManager stateManager,
        ProximityItemController itemPickupController)
    {
        _movementModel = movementModel;
        _movementController = movementController;
        _weaponController = weaponController;
        _stateManager = stateManager;
        _itemPickupController = itemPickupController;
    }

    public void EnterState()
    {
        _movementModel.OnMove += OnMove;
    }

    public void ExitState()
    {
        _movementModel.OnMove -= OnMove;
    }

    public void Update()
    {
        _movementController.Update();
        _weaponController.Update();
        _itemPickupController.Update();
    }

    private void OnMove(Vector2 vector)
    {
        if (vector != Vector2.zero)
            _stateManager.ChangeState(_stateManager.Moving);
    }
}
```

Рисунок 17 – Реализация состояния Idle

На рисунке 17 изображено состояние отдыха игрока. В состоянии отдыха игрок может атаковать и начать движение.

```
using UnityEngine;

public class PlayerMovingState : IState
{
    private readonly IMovementModel _movementModel;
    private readonly IMovementController _movementController;
    private readonly IWeaponInputController _weaponController;
    private readonly PlayerDashController _dashController;
    private readonly ProximityItemController _itemPickupController;

    private readonly PlayerDashModel _dashModel;

    private readonly PlayerStateManager _stateManager;
    private readonly PlayerView _view;

    public PlayerMovingState(IMovementModel movementModel,
        IMovementController movementController,
        IWeaponInputController weaponController,
        PlayerStateManager stateManager,
        PlayerView view,
        ProximityItemController itemPickupController,
        PlayerDashController dashController,
        PlayerDashModel dashModel)
    {
        _movementModel = movementModel;
        _movementController = movementController;
        _weaponController = weaponController;
        _stateManager = stateManager;
        _view = view;
        _itemPickupController = itemPickupController;
        _dashController = dashController;
        _dashModel = dashModel;
    }

    public void EnterState()
    {
        _movementModel.OnMove += OnMove;
        _dashModel.OnDash += OnDash;
    }

    public void ExitState()
    {
        _movementModel.OnMove -= OnMove;
        _dashModel.OnDash -= OnDash;
    }

    public void Update()
    {
        _movementController.Update();
        _weaponController.Update();
        _itemPickupController.Update();
        _dashController.Update();
    }

    private void OnMove(Vector2 velocity)
    {
        if (velocity == Vector2.zero)
            _stateManager.ChangeState(_stateManager.Idle);

        _view.Rigidbody.velocity = velocity;
        //_view.transform.position += (Vector3)velocity;
    }

    private void OnDash()
    {
        _stateManager.ChangeState(_stateManager.Dashing);
    }
}
```

Рисунок 18 – Реализация состояния Moving

На рисунке 18 изображена реализация состояния движения. При остановке движения игрок обратно переходит в состояние отдыха. Нахождение в состоянии движения гарантирует, что вектор скорости игрока ненулевой, что делает возможным рывок, который совершается в направлении движения.

```
public class PlayerDashingState : IState
{
    private readonly IMovementModel _movement;
    private readonly PlayerStateManager _stateManager;
    private readonly PlayerDashModel _dash;
    private readonly IWeaponInputController _weaponController;

    private readonly SpriteRenderer _sr;
    private readonly PlayerView _view;
    private readonly WeaponView _weapon;

    private readonly IPool<SpriteRenderer> _queue;

    public PlayerDashingState(IMovementModel movement,
        PlayerStateManager stateManager,
        PlayerDashModel dash,
        SpriteRenderer sr,
        PlayerView view,
        WeaponView weapon,
        IWeaponInputController weaponController,
        IPool<SpriteRenderer> dashQueue)
    {
        _movement = movement;
        _stateManager = stateManager;
        _dash = dash;
        _sr = sr;
        _view = view;
        _weapon = weapon;
        _weaponController = weaponController;
        _queue = dashQueue;
    }
```

Рисунок 19 – Конструктор и поля класса состояния Dashing

На рисунке 19 изображены поля и конструктор состояния рывка. IPool<SpriteRender> является пулом полупрозрачных теней, которые создаются при рывке игрока.

```
public void EnterState()
{
    _weapon.gameObject.SetActive(false);

    var position = _view.transform.position;
    var destination = new Vector3(
        position.x + _movement.Velocity.x * _dash.Distance,
        position.y + _movement.Velocity.y * _dash.Distance,
        position.z);

    var sequence = DOTween.Sequence();
    sequence.Append(_view.Rigidbody.DOMove(destination, _dash.Duration).SetEase(Ease.OutSine));

    var trailSequence = DOTween.Sequence();
    for (int i = 0; i < _dash.Afterimages; i++)
    {
        trailSequence.AppendCallback(() => DoTrail());
        trailSequence.AppendInterval(_dash.Duration / _dash.Afterimages);
    }

    sequence.AppendCallback(() => _stateManager.ChangeState(_stateManager.Moving));
}

private void DoTrail()
{
    var sequence = DOTween.Sequence();
    var sr = _queue.Dequeue().GetComponent<SpriteRenderer>();

    sequence.AppendCallback(() => {
        sr.sprite = _sr.sprite;
        sr.color = Color.white;
        sr.gameObject.transform.position = _view.transform.position;
    });

    sequence.Append(sr.DOFade(0, _dash.AfterimageDuration));
    sequence.AppendCallback(() => _queue.Enqueue(sr));
}

public void ExitState()
{
    _weapon.gameObject.SetActive(true);
}

public void Update()
{
    _weaponController.Update();
}
```

Рисунок 20 – Методы класса состояния Dashing

На рисунке 20 изображены основные методы состояния рывка. При входе в состояние создаётся твин, который передвигает игрока. По завершении твина игрок обратно переходит в состояние движения. Параллельно с первым также проигрывается твин, создающий полупрозрачные тени игрока.

Рисунок 21 – Полупрозрачные тени при рывке

На рисунке 21 можно видеть игрока в состоянии рывка. Тени создаются с определённой прозрачностью, после чего затухают до полной невидимости.

3.2 Разработка врагов

Движение слизней во многом дублирует движение игрока, за исключением контроллера.

```
using UnityEngine;

public class ProximityMovementController : IMovementController
{
    private readonly IMovementModel _model;
    private readonly Transform _transform;
    private readonly int _mask;

    public ProximityMovementController(IMovementModel model, Transform transform)
    {
        _model = model;
        _transform = transform;
        _mask = LayerMask.GetMask("Player");
    }

    public void Update()
    {
        Collider2D collider = Physics2D.OverlapCircle(_transform.position, 3, _mask);

        if (collider is null)
        {
            _model.ChangeVelocity(Vector3.zero);
            return;
        }

        Vector2 dir = (collider.transform.position - _transform.position).normalized;
        _model.ChangeVelocity(_model.MovementSpeed * dir);
    }
}
```

Рисунок 22 – Реализация контроллера слизней

На рисунке 22 изображён контроллер движения слизней. Если игрок приближается достаточно близко к слизню, слизень начинает двигаться в его сторону.

Рисунок 23 – Движение слизня в сторону игрока

На рисунке 23 можно видеть активированного слизня, движущегося в сторону игрока. Если игрок выходит из зоны сканирования слизня, слизень обратно переходит в состояние отдыха.

Помимо движения, слизни также могут получать урон от игрока. Для этого реализована отдельная MVC система под названием Damageable. В этом случае интерфейсы идеальны, так как позволяют создавать любые объекты, которые могут получать урон [19].

```
using UnityEngine;
using Zenject;

public class Damageable : MonoBehaviour, IDamageable
{
    private FlashEffect _damageFlash;

    private IDamageableController _controller;
    private IDamageableModel _model;

    private Rigidbody2D _rigidbody;

    public int Id => GetInstanceID();

    [Inject]
    public void Inject(FlashEffect flash, Rigidbody2D rigidbody, IDamageableController controller, IDamageableModel model)
    {
        _damageFlash = flash;
        _controller = controller;
        _model = model;
        _rigidbody = rigidbody;
        Debug.Log(_rigidbody is null);
        Debug.Log("injected");
    }

    private void OnEnable()
    {
        _model.OnDamageTaken += OnDamageTaken;
    }

    private void OnDisable()
    {
        _model.OnDamageTaken -= OnDamageTaken;
    }

    public void ReceiveDamageInstance(DamageInstance damage)
    {
        _controller.HandleDamageReceived(damage);
    }

    private void OnDamageTaken(DamageInstance damage)
    {
        _rigidbody.AddForce(damage.KnockbackDirection);
        Debug.Log(damage.KnockbackDirection);
        _damageFlash.DoFlash();
    }
}
```

Рисунок 24 – Реализация представления Damageable

На рисунке 24 представлена реализация компонента Damageable. Компонент позволяет другим объектам системы передавать параметры урона этому объекту. Данный объект затем проходит через контроллер, модель и обратно возвращается представлению в виде события получения урона. На основе этого события объект загорается ярким белым цветом для индикации получения урона.

Рисунок 25 – Индикация получения урона

На рисунке 25 представлена индикация получения урона врагом в виде резкого загорания в белый и быстрого, но постепенного затухания. Это поведение реализуется классом FlashEffect.

```
using DG.Tweening;
using System;
using UnityEngine;
using Zenject;

public class FlashEffect
{
    private Sequence _flashSequence;
    private Settings _settings;

    [Inject]
    public void Inject(SpriteRenderer sr, Settings settings)
    {
        _settings = settings;
        _flashSequence = DOTween.Sequence();

        var flashMat = sr.material;

        _flashSequence.Append(flashMat.DOFloat(1, "_FlashAmount", _settings.InitialFlashDuration));
        _flashSequence.AppendInterval(_settings.FullFlashDuration);
        _flashSequence.Append(flashMat.DOFloat(0, "_FlashAmount", _settings.FadeOutDuration));

        _flashSequence.SetAutoKill(false);
        _flashSequence.SetLink(sr.gameObject);

        _flashSequence.Pause();
    }

    public void DoFlash()
    {
        _flashSequence.Restart();
    }

    [Serializable]
    public class Settings
    {
        public float InitialFlashDuration;
        public float FullFlashDuration;
        public float FadeOutDuration;
    }
}
```

Рисунок 26 – Реализация класса FlashEffect

На рисунке 26 представлена реализация класса FlashEffect. Он использует ранее упомянутый твининг для загорания и затухания врага определённым цветом. Спрайты врагов используют особый шейдер, который делает это загорание возможным.

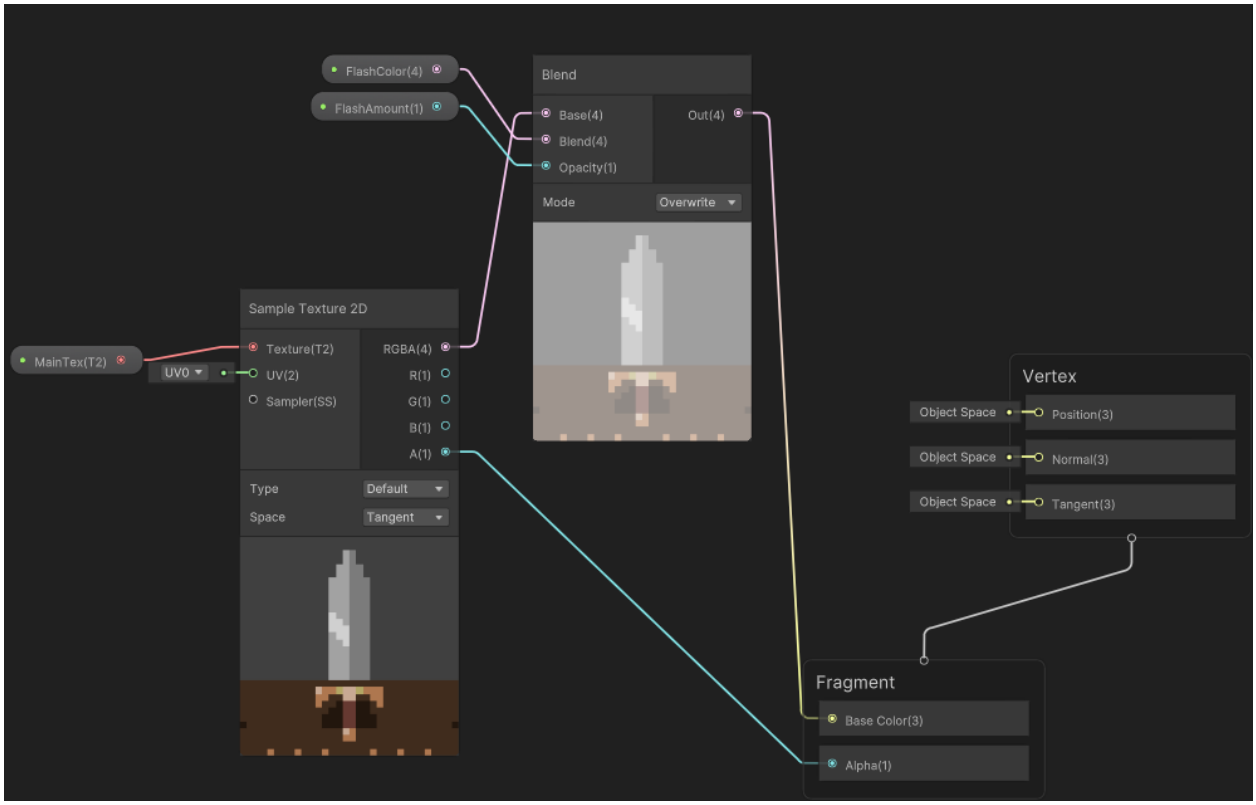
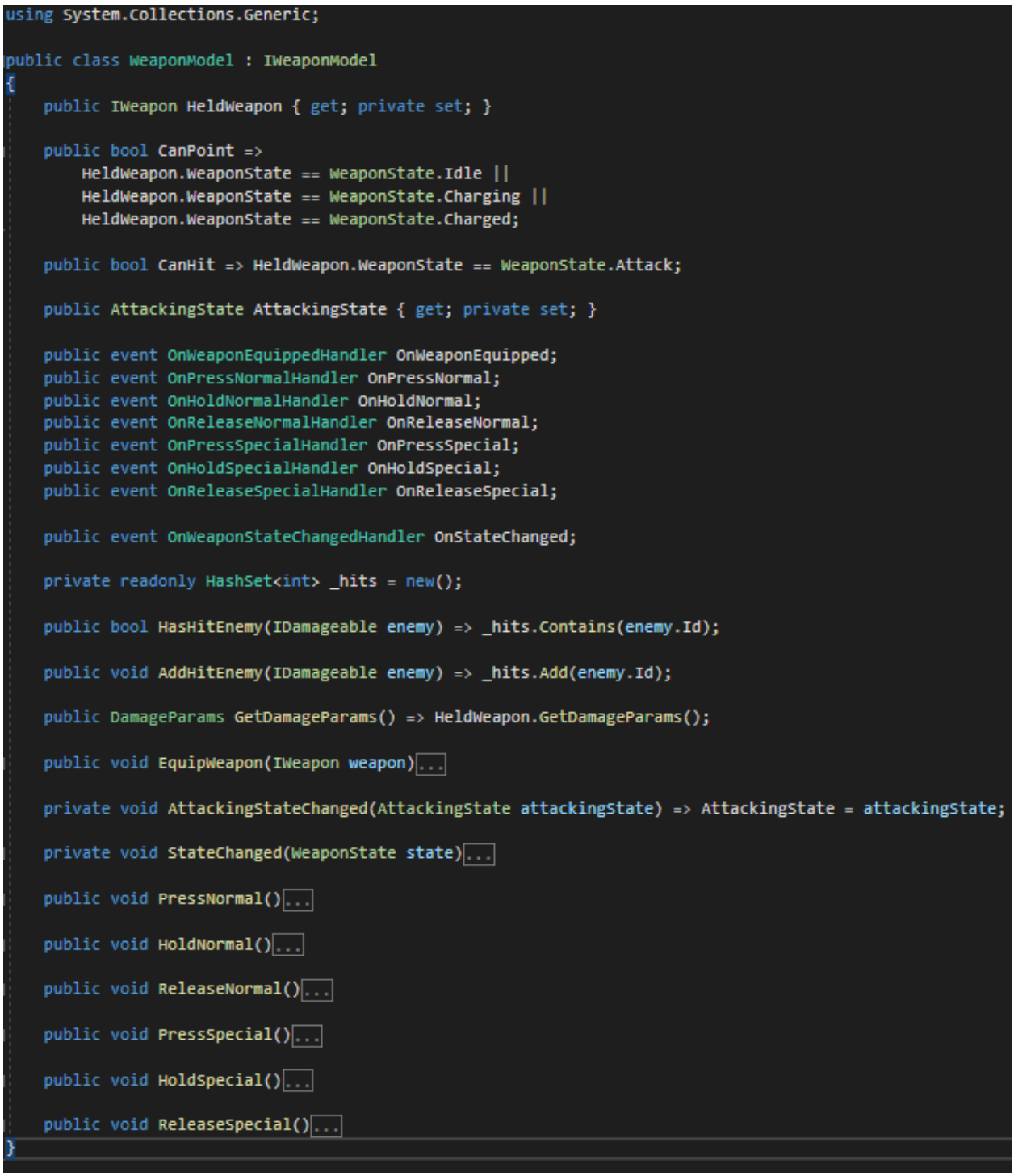


Рисунок 27 – Шейдер для загорания спрайтов

На рисунке 27 изображён шейдер, делающий загорание спрайтов возможным. Для наглядности значение загорания выставлено в 50%. При помощи Blend мы можем смешать наш исходный спрайт с другим цветом. FlashAmount, подключённый к каналу Opacity, определяет непрозрачность нового цвета.

Для атак игрока используется отдельная MVC система. Модель хранит в себе текущее оружие, подобранное игроком, контроллер считывает весь ввод с мышки (нажатие, удержание, отпуск правой и левой кнопки мыши), представление анимирует атаку.



```
using System.Collections.Generic;

public class WeaponModel : IWeaponModel
{
    public IWeapon HeldWeapon { get; private set; }

    public bool CanPoint =>
        HeldWeapon.WeaponState == WeaponState.Idle ||
        HeldWeapon.WeaponState == WeaponState.Charging ||
        HeldWeapon.WeaponState == WeaponState.Charged;

    public bool CanHit => HeldWeapon.WeaponState == WeaponState.Attack;

    public AttackingState AttackingState { get; private set; }

    public event OnWeaponEquippedHandler OnWeaponEquipped;
    public event OnPressNormalHandler OnPressNormal;
    public event OnHoldNormalHandler OnHoldNormal;
    public event OnReleaseNormalHandler OnReleaseNormal;
    public event OnPressSpecialHandler OnPressSpecial;
    public event OnHoldSpecialHandler OnHoldSpecial;
    public event OnReleaseSpecialHandler OnReleaseSpecial;

    public event OnWeaponStateChangedHandler OnStateChanged;

    private readonly HashSet<int> _hits = new();

    public bool HasHitEnemy(IDamageable enemy) => _hits.Contains(enemy.Id);

    public void AddHitEnemy(IDamageable enemy) => _hits.Add(enemy.Id);

    public DamageParams GetDamageParams() => HeldWeapon.GetDamageParams();

    public void EquipWeapon(IWeapon weapon)...

    private void AttackingStateChanged(AttackingState attackingState) => AttackingState = attackingState;

    private void StateChanged(WeaponState state)...

    public void PressNormal()...

    public void HoldNormal()...

    public void ReleaseNormal()...

    public void PressSpecial()...

    public void HoldSpecial()...

    public void ReleaseSpecial()...
}
```

Рисунок 28 – Реализация оружейной модели

На рисунке 28 показана частичная реализация оружейной модели игрока. Большинство событий о вводе, получаемые с контроллера, передаются в само оружие, которое обрабатывает его самостоятельно, после чего оружие отправляет события изменения своего состояния в модель.

3.3 Разработка соединения между комнатами

Комнаты задаются классом RoomInfo. У каждой комнаты есть свои свои выходы, которые задаются классом RoomExitInfo.

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

namespace Dreamrogue.Worldgen
{
    public class RoomInfo : MonoBehaviour
    {
        public Tilemap Grass => _grass;
        public Tilemap Wall => _wall;
        public IReadOnlyList<RoomExitInfo> Exits => _exits.AsReadOnly();
        public BoundsInt Bounds => _wall.cellBounds;

        [SerializeField]
        private Tilemap _grass;
        [SerializeField]
        private Tilemap _wall;
        [SerializeField]
        private List<RoomExitInfo> _exits;

        public void CompressBounds()
        {
            _grass.CompressBounds();
            _wall.CompressBounds();
        }
    }

    [Serializable]
    public struct RoomExitInfo
    {
        [SerializeField]
        private BoundsInt bounds;
        [SerializeField]
        private Orientation orientation;

        public BoundsInt Bounds => bounds;
        public Orientation Orientation => orientation;

        public RoomExitInfo(BoundsInt bounds, Orientation orientation)
        {
            this.bounds = bounds;
            this.orientation = orientation;
        }
    }
}
```

Рисунок 29 – Реализация классов RoomInfo и RoomExitInfo

На рисунке 29 изображена реализация классов RoomInfo и RoomExitInfo. RoomInfo хранит в себе два основных тайлмапа – стены и трава. Это область границ и область ходьбы соответственно. Границами комнаты (объект BoundsInt) является границей тайлмапа стен. Этот объект хранит в себе 4 величины – ширина и высота, формирующие размер, а также X и Y, формирующие положение границ.

Каждый RoomExitInfo также хранит в себе границы. Помимо границ данная структура также содержит в себе ориентацию выхода. Это используется для определения направления соединения комнат.

```
private void ConnectRooms(RoomInfo room1, RoomInfo room2, RoomExitInfo exit1, RoomExitInfo exit2)
{
    // room 1 is always upmost/leftmost
    // swap if that's not true
    var perp = OrientationUtil.PerpendicularAxis(exit1.Orientation);
    if (RoomUtil.WorldCompareCells(room1, room2, exit1.Bounds, exit2.Bounds, Extreme.Min, perp) == -1)
    {
        (room1, room2) = (room2, room1);
        (exit1, exit2) = (exit2, exit1);
    }

    RoomInfo[] turns = worldgenPieces.GetTurnPieces(exit1.Orientation, exit2.Orientation);
    var o1 = exit1.Orientation;
    var o2 = exit2.Orientation;

    PlaceExit(room1, exit1);
    PlaceExit(room2, exit2);

    if (o1 == o2)
        ConnectSameFacing(room1, room2, exit1, exit2, turns);
    else if (OrientationUtil.ParallelAxis(o1) == OrientationUtil.ParallelAxis(o2))
        ConnectOppositeFacing(room1, room2, exit1, exit2, turns);
    else
        ConnectDifferentAxes(room1, room2, exit1, exit2, turns);
}
```

Рисунок 30 – Основная функция соединения двух комнат

На рисунке 30 изображена функция для соединения двух комнат по двум выходам. Функция изменяет их порядок, если он неверен (считается, что комната 1 – это всегда самая верхняя и самая левая комната). Затем функция получает все нужные повороты, которые передаются далее. Функция берёт на себя обязанность проставления выходов, так как все комнаты поначалу являются закрытыми. После этого определяется тип соединения и производится само соединение.

```
private void ConnectSameFacing(RoomInfo room1, RoomInfo room2, RoomExitInfo exit1, RoomExitInfo exit2, RoomInfo[] turns)
{
    RoomInfo turn1 = turns[0];
    RoomInfo turn2 = turns[1];

    Tuple<Extreme, Axis> ea = OrientationUtil.ExtremeAxis(exit1.Orientation);
    Vector3 furthestCell = RoomUtil.WorldExtremeCell(ea.Item1, room1, room2, room1.Bounds, room2.Bounds, ea.Item1, ea.Item2, turn2.Bounds.size.y);
    BoundsInt turnBounds1 = MakeTurnBounds(room1, exit1, furthestCell, turn1.Bounds.size);
    BoundsInt turnBounds2 = MakeTurnBounds(room2, exit2, furthestCell, turn2.Bounds.size);

    // place turns
    PlaceRoom(room1, turn1, turn1.Bounds, turnBounds1);
    PlaceRoom(room2, turn2, turn2.Bounds, turnBounds2);

    // place the corridors from exits to turns
    PlaceCorridor(room1, exit1.Bounds, turnBounds1, exit1.Orientation);
    PlaceCorridor(room2, exit2.Bounds, turnBounds2, exit2.Orientation);

    // place the corridor from turn to turn
    turnBounds2 = ConvertBoundsToDifferentRoomSpace(room2, room1, turnBounds2);
    PlaceCorridor(room1, turnBounds1, turnBounds2, OrientationUtil.ConvertToDifferentAxis(exit1.Orientation));
}
```

Рисунок 31 – Реализация соединения комнат, смотрящих в одну сторону.

На рисунке 31 изображён код для соединения двух комнат, если они смотрят в одну сторону. Большое количество функциональности вынесено в функции-помощники для упрощения реализации, так как комбинаций комнат, выходы которых ориентированы одинаково, всего 4 (восток-восток, север-север и т.д.), и требуется учесть каждую.

Для соединения комнат, смотрящих в одну сторону, требуется два поворота. Предполагается, что комнаты не могут быть расположены друг за другом, что потребовало бы три поворота. Алгоритм можно поделить на следующие шаги:

1. Найти границы поворотов
2. Расположить повороты
3. Соединить коридором каждую комнату с нужным поворотом
4. Соединить коридором два поворота

```
private void ConnectDifferentAxes(RoomInfo room1, RoomInfo room2, RoomExitInfo exit1, RoomExitInfo exit2, RoomInfo[] turns)
{
    RoomInfo turn = turns[0];
    var cell = RoomUtil.GetWorldCell(room2, exit2.Bounds, Extreme.Min);
    var turnBounds1 = MakeTurnBounds(room1, exit1, cell, turn.Bounds.size);
    var turnBounds2 = ConvertBoundsToDifferentRoomSpace(room1, room2, turnBounds1);

    PlaceRoom(room1, turn, turn.Bounds, turnBounds1);
    PlaceCorridor(room1, exit1.Bounds, turnBounds1, exit1.Orientation);
    PlaceCorridor(room2, exit2.Bounds, turnBounds2, exit2.Orientation);
}
```

Рисунок 32 – Реализация соединения комнат на разных осях

На рисунке 32 изображена реализация соединения комнат с выходами, ориентации которых находятся на разных осях, например, юг-восток, север-запад и т.д. Для соединения двух комнат таким образом требуется всего один поворот. Предполагается, что комнаты не могут быть расположены друг за другом, что потребовало бы три поворота.

```
private void ConnectOppositeFacing(RoomInfo room1, RoomInfo room2, RoomExitInfo exit1, RoomExitInfo exit2, RoomInfo[] turns)
{
    RoomInfo turn1 = turns[0];
    RoomInfo turn2 = turns[1];

    var cell1 = RoomUtil.GetWorldCell(room1, exit1.Bounds, Extreme.Min);
    var cell2 = RoomUtil.GetWorldCell(room2, exit2.Bounds, Extreme.Min);

    Vector3 worldMiddle = new((cell1.x + cell2.x) / 2, (cell1.y + cell2.y) / 2);

    var turnBounds1 = MakeTurnBounds(room1, exit1, worldMiddle, turn1.Bounds.size);
    var turnBounds2 = MakeTurnBounds(room2, exit2, worldMiddle, turn2.Bounds.size);

    // place turns
    PlaceRoom(room1, turn1, turn1.Bounds, turnBounds1);
    PlaceRoom(room2, turn2, turn2.Bounds, turnBounds2);

    // place the corridors from exits to turns
    PlaceCorridor(room1, exit1.Bounds, turnBounds1, exit1.Orientation);
    PlaceCorridor(room2, exit2.Bounds, turnBounds2, exit2.Orientation);

    // place the corridor from turn to turn
    turnBounds2 = ConvertBoundsToDifferentRoomSpace(room2, room1, turnBounds2);
    PlaceCorridor(room1, turnBounds1, turnBounds2, OrientationUtil.ConvertToDifferentAxis(exit1.Orientation));
}
```

Рисунок 33 – Реализация соединения комнат с противоположными выходами

На рисунке 33 представлены реализация соединения комнат, выходы которых смотрят друг друга, то есть, противоположны друг другу. Это могут быть такие сочетания как восток-запад, юг-север и т.д. Не допускается параллельное расположение комнат по противоположной оси, так как в этом случае количество поворотов увеличивается до четырёх.

```
private void PlaceExit(RoomInfo target, RoomExitInfo exitInfo)
{
    var exitRoom = worldgenPieces.GetExit(exitInfo.Orientation);
    var exitBounds = new BoundsInt(Vector3Int.zero, exitInfo.Bounds.size);
    PlaceRoom(target, exitRoom, exitBounds, exitInfo.Bounds);
}

private void PlaceCorridor(RoomInfo target, BoundsInt from, BoundsInt to, Orientation orientation)
{
    var path = OrientationUtil.IsVertical(orientation) ?
        worldgenPieces.VerticalPath :
        worldgenPieces.HorizontalPath;

    foreach(var placementBounds in OrientationUtil.EnumerateBounds(from, to, path.Bounds.size, orientation))
        PlaceRoom(target, path, path.Bounds, placementBounds);
}

private void PlaceRoom(RoomInfo target, RoomInfo source, BoundsInt sourceBounds, BoundsInt placementBounds)
{
    var wall = source.Wall.GetTilesBlock(sourceBounds);
    var grass = source.Grass.GetTilesBlock(sourceBounds);

    int i = 0;
    var positions = new Vector3Int[placementBounds.size.x * placementBounds.size.y];
    foreach (var pos in placementBounds.allPositionsWithin)
        positions[i++] = pos;

    target.Grass.SetTiles(positions, grass);
    target.Wall.SetTiles(positions, wall);
}
```

Рисунок 34 – Функции расставления объектов

На рисунке 34 показаны различные функции, использующиеся для расставления объектов соединений по сцене. Каждый выход, поворот и фрагмент коридора считается отдельной комнатой. Метод PlaceRoom позволяет расставить сразу и стены, и траву комнаты. Методы PlaceExit и PlaceCorridor работают на основе ранее упомянутого метода, чтобы расставлять повороты и коридоры.

```
// always goes from left to right, from bottom to top
// bounds have to be on the same x or y (if enumerating on y or x accordingly)
public static IEnumerable<BoundsInt> EnumerateBounds(BoundsInt b1, BoundsInt b2, Vector3Int size, Orientation orientation)
{
    if (IsVertical(orientation))
    {
        if (b1.y > b2.y)
            (b1, b2) = (b2, b1);
        for (int i = b1.yMax; i < b2.yMin; i++)
            yield return new BoundsInt(new Vector3Int(b1.x, i), size);
    }
    else
    {
        if (b1.x > b2.x)
            (b1, b2) = (b2, b1);
        for (int i = b1.xMax; i < b2.xMin; i++)
            yield return new BoundsInt(new Vector3Int(i, b1.y), size);
    }
}
```

Рисунок 35 – Реализация вспомогательной функции EnumerateBounds

На рисунке 35 изображена функция EnumerateBounds, которая используется в методе PlaceCorridor. Она создаёт границы для каждого фрагмента коридора, чтобы можно было создавать коридоры любой длины.

```
public static Orientation ConvertToDifferentAxis(Orientation orientation)
{
    return orientation switch
    {
        Orientation.East => Orientation.North,
        Orientation.West => Orientation.South,
        Orientation.North => Orientation.West,
        Orientation.South => Orientation.East,
        _ => throw new ArgumentException("Unknown orientation.")
    };
}

public static Axis PerpendicularAxis(Orientation orientation)
{
    return orientation switch
    {
        Orientation.East or Orientation.West => Axis.Y,
        Orientation.North or Orientation.South => Axis.X,
        _ => throw new ArgumentException("Unknown orientation.")
    };
}

public static Axis ParallelAxis(Orientation orientation)
{
    return orientation switch
    {
        Orientation.East or Orientation.West => Axis.X,
        Orientation.North or Orientation.South => Axis.Y,
        _ => throw new ArgumentException("Unknown orientation.")
    };
}

public static Tuple<Extreme, Axis> ExtremeAxis(Orientation orientation)
{
    return orientation switch
    {
        Orientation.East => new Tuple<Extreme, Axis>(Extreme.Max, Axis.X),
        Orientation.West => new Tuple<Extreme, Axis>(Extreme.Min, Axis.X),
        Orientation.North => new Tuple<Extreme, Axis>(Extreme.Max, Axis.Y),
        Orientation.South => new Tuple<Extreme, Axis>(Extreme.Min, Axis.Y),
        _ => throw new ArgumentException("Unknown orientation.")
    };
}
```

Рисунок 36 – Реализация прочих вспомогательные функции

На рисунке 36 показаны реализации различных вспомогательных функций, которые используются для соединения комнат. Они позволяют получать различную информацию о какой-либо ориентации, чтобы ранее упомянутые алгоритмы могли работать в общем случае, а не для конкретных ориентаций выходов.

```
public static Vector3 WorldExtremeCell(Extreme extreme,
    RoomInfo room1, RoomInfo room2,
    BoundsInt bounds1, BoundsInt bounds2,
    Extreme boundsExtreme, Axis axis, int yOffset = 0)
{
    var cell1 = GetWorldCell(room1, bounds1, boundsExtreme);
    var cell2 = GetWorldCell(room2, bounds2, boundsExtreme);

    var coord1 = GetCellCoordinate(cell1, axis);
    var coord2 = GetCellCoordinate(cell2, axis);

    var result = extreme switch
    {
        Extreme.Min => coord1 < coord2 ? cell1 : cell2,
        Extreme.Max => coord1 > coord2 ? cell1 : cell2,
        _ => throw new ArgumentException("Invalid extreme."),
    };

    if (axis == Axis.Y)
        result.y -= yOffset;

    return result;
}
```

Рисунок 36 – Реализация функции WorldExtremeCell

На рисунке 36 показана реализация функции, позволяющей получить самую дальнюю точку комнаты по оси и типу экстремальной точки (минимум или максимум). Данная функция используется для определения позиции поворота при соединении двух комнат, выходы которых имеют идентичную ориентацию. Выход ставится напротив комнаты, которая по заданной оси и типу экстремальной точки считается дальнейшей.

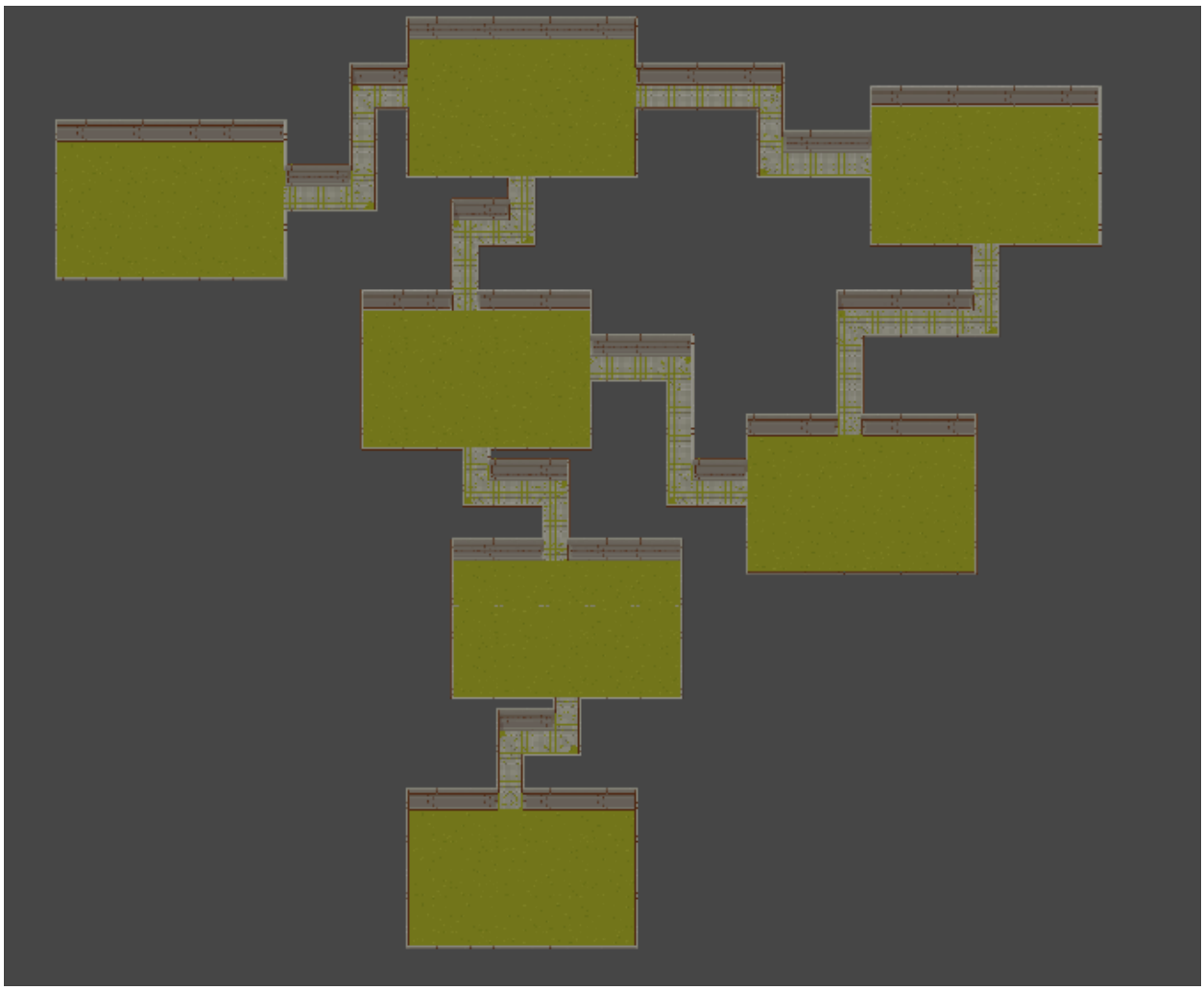


Рисунок 37 – результат соединения комнат

На рисунке 37 изображён результат работы алгоритма соединения множества комнат по разным выходам. Как можно видеть на рисунке, соединения между каждой из комнат расставились успешно, что подтверждает корректность работы алгоритма.

Заключение

Выпускная квалификационная работа посвящена разработке игрового функционала для однопользовательской 2D игры в жанрах hack’n’slash и roguelike. Цель игрока – изучение мира процедурно генерирующегося мира, улучшение своих навыков и экипировки, преодоление препятствий в виде врагов и боссов.

Целью работы является реализация игрового функционала, включающего в себя:

- передвижение игрока
- враги, которые двигаются в сторону игрока
- динамическое соединение игровых локаций

Выпускную квалификационную работу можно поделить на следующие взаимосвязанные этапы:

а) подробный теоретический анализ поставленных задач на разработку игрового функционала

б) проектирование комплексных решения с использованием различных подходов для достижения максимальной модульности и минимальной связанности

в) разработка высокопроизводительных и чистых систем, реализующих необходимый игровой функционал

В результате работы были решены такие задачи, как:

- изучены методы процедурной генерации игрового мира
- построены математические модели игрового функционала
- проведено сравнение игровых движков Unreal Engine, Godot Engine и Unity Engine
- использованы такие шаблоны и методологии как Model-View-Controller, Dependency Injection, твининг-анимации.

Разработанный игровой функционал был успешно протестирован и активно применяется в игре.

Список используемой литературы и используемых источников

1. Грегори Джейсон. Игровой движок. Программирование и внутреннее устройство. СПб.: Изд-во «Питер», 2022. 1136 с.
2. С. Тепляков. Паттерны проектирования на платформе .NET. Санкт-Петербург: Питер, 2021. 320 с.
3. Хокинг Дж. Unity в действии: Мультиплатформенная разработка на C#. Санкт-Петербург: Питер, 2016. 334 с.
4. Unity и MVC: как прокачать разработку игры [Электронный ресурс]. URL: https://habr.com/ru/articles/281783/ (дата обращения: 19.05.2024)
5. Anupama Murali, Harihara Subramanian J, Pethuru Raj Chelliah. Architectural Patterns. Packt Publishing, 2017. 468 с.
6. Dependency Injection in .NET / Mark Seemann. – Manning Publications, 2011. – 584 с.
7. DoTween Documentation [Электронный ресурс]. URL: http://dotween.demigiant.com/documentation.php (дата обращения: 19.05.2024)
8. Easing functions [Электронный ресурс]. URL: https://easings.net/ (дата обращения: 19.05.2024)
9. Events (C# Programming Guide) [Электронный ресурс]. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/ (дата обращения: 19.05.2024)
10. Game Development Patterns with Unity 2021 / David Baron. – Packt Publishing, 2024. – 246 с.
11. Godot Docs [Электронный ресурс]. URL: https://docs.godotengine.org/en/stable/ (дата обращения: 19.05.2024)
12. How Many Gamers Are There? [Электронный ресурс]. URL: https://explodingtopics.com/blog/number-of-gamers (дата обращения: 19.05.2024)

13. Introduction to Object Pooling [Электронный ресурс]. URL: https://learn.unity.com/tutorial/introduction-to-object-pooling (дата обращения: 19.05.2024)
14. The All-In-One Engine To Create Mobile Games | Unity [Электронный ресурс]. URL: https://unity.com/solutions/mobile (дата обращения: 19.05.2024)
15. Understanding Procedural Dungeon Generation in Unity [Электронный ресурс]. URL: https://gamedevacademy.org/understanding-procedural-dungeon-generation-in-unity/ (дата обращения: 19.05.2024)
16. Unity – Manual: Creating Tilemaps [Электронный ресурс]. URL: https://docs.unity3d.com/Manual/Tilemap-CreatingTilemaps.html (дата обращения: 19.05.2024)
17. Unity – Manual: Input System [Электронный ресурс]. URL: https://docs.unity3d.com/Manual/com.unity.inputsystem.html (дата обращения: 19.05.2024)
18. Unreal Engine 5 [Электронный ресурс]. URL: https://www.unrealengine.com/en-US/unreal-engine-5 (дата обращения: 19.05.2024)
19. Using Interface to damage enemies in Unity [Электронный ресурс]. URL: https://bergstrand-niklas.medium.com/using-interface-to-damage-enemies-in-unity-4a6631970811 (дата обращения: 19.05.2024)
20. What is the Tween Technique? [Электронный ресурс]. URL: https://medium.com/@satishlokhande5674/what-is-the-tween-technique-40be371ab335 (дата обращения: 19.05.2024)
21. Zenject Documentation [Электронный ресурс]. URL: https://github.com/modesttree/Zenject (дата обращения: 19.05.2024)