МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ВЛАДИВОСТОКСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВВГУ»)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И АНАЛИЗА ДАННЫХ
КАФЕДРА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И СИСТЕМ

ОТЧЕТ
ПО РАЗРАБОТКЕ КОНСОЛЬНОЙ RPG ИГРЫ

по дисциплине
«Информатика и программирование»

Студент
гр. БИН-25-2 _____ А.М. Сахарюк

Ассистент
преподавателя _____ М.В. Водяницкий

Владивосток 2026

Задание

Техническое задание – Текстовая RPG-игра

Вы работаете программистом в небольшой японской компании на заре игровой индустрии. Компания разрабатывает свою первую экспериментальную игру – текстовую RPG, которая должна запускаться прямо в консоли и погружать игрока в атмосферу подземелий, опасностей и развития персонажа.

Ваша задача – реализовать прототип игры, который демонстрирует основные игровые механики: характеристики персонажа, бои, прокачку, инвентарь и случайные события.

1. Общая идея программы

Программа представляет собой консольную текстовую RPG, в которой игрок:

– создает персонажа (выбор расы)

– получает случайные характеристики в рамках выбранной расы

– исследует подземелье, состоящее из случайных комнат

– сражается с врагами, находит предметы и улучшает персонажа

– повышает уровень и распределяет очки характеристик

– принимает решения, влияющие на дальнейший путь

– Игра работает в пошаговом режиме и управляется вводом команд с клавиатуры

2. Создание персонажа

2.1 Выбор расы

В начале игры пользователь выбирает расу персонажа (например):

– Человек

– Эльф

– Дворф

Каждая раса задает диапазоны генерации характеристик.

2.2 Характеристики персонажа

Характеристики генерируются случайным образом при создании персонажа, но в допустимых пределах для выбранной расы.

Пример набора характеристик (можно расширять):

– HP – здоровье

– Attack – сила атаки

– Defense – защита

– Agility – ловкость (влияет на уклонение)

– Height – рост

– Weight – вес

Допускается, что некоторые характеристики влияют друг на друга (например, рост и вес влияют на уклонение или скорость).

3. Опыт и уровни

– Персонаж получает опыт за победу над врагами

– При накоплении нужного количества опыта повышается уровень

– Каждый новый уровень дает очки прокачки

3.1 Прокачка характеристик

Игрок может распределять очки вручную между характеристиками.

Пример:

– +1 к атаке

– +2 к HP

– +1 к ловкости

Распределение очков выполняется в комнатах отдыха.

4. Инвентарь и экипировка

4.1 Инвентарь

Инвентарь хранит предметы:

– зелья (лечение и др.)

– монеты

– оружие

– прочие предметы

Игрок может:

- просматривать инвентарь
- использовать предметы
- выбрасывать любые предметы

4.2 Экипировка

В инвентаре должны быть отдельные слоты:

- оружие
- броня

Экипированные предметы влияют на характеристики персонажа.

5. Подземелье и комнаты

5.1 Структура подземелья

- Игра начинается в подземелье
- Подземелье состоит из комнат
- После каждой комнаты игрок выбирает путь:
    1) налево
    2) направо

5.2 Типы комнат

Комнаты генерируются случайно:

- Боевая комната – бой с врагом
- Комната отдыха – без событий
- Комната с сундуком – предметы или золото

Возможны комбинации:

- слева враг, справа сундук
- оба врага
- обе комнаты отдыха

5.3 Видимость комнат

Перед выбором направления игрок:

– иногда знает, что находится дальше

– иногда не знает (темно, неизвестно)

Информация о видимости определяется случайно.

6. Враги и сложность

– Враги генерируются случайно

– У врагов есть характеристики (HP, атака, защита и т.д.)

– С каждым этажом подземелья сложность возрастает

– Каждые N комнат или действий происходит переход на новый этаж

7. Боевая система

Бой происходит в пошаговом режиме.

Пример действий игрока:

– атаковать

– использовать предмет

– попытаться уклониться

Учитываются:

– характеристики игрока

– экипировка

– случайные факторы (уклонение, критический удар)

8. Предметы и добыча

Враги и сундуки могут давать:

1) зелья
2) оружие
3) другие предметы

– Полученные предметы добавляются в инвентарь

– При нехватке места игрок решает, что выбросить

9. Хранение данных

Допускается (но не обязательно):

– сохранение состояния игры в файл

– использование формата JSON для хранения:

1) характеристик персонажа
2) инвентаря
3) текущего этажа

10. Пример работы программы (фрагмент)

```
Выберите
 расу:
1 - Человек
2 - Эльф
3 - Дворф

> 2Ваш

 персонаж создан!
HP: 85
ATK: 12
DEF: 6
AFI: 14Вы

 входите в подземелье...Перед

 вами развилка.
(1) Слева: ???
(2) Справа: Комната отдыхаКуда

 пойти?
> 1
```

11. Ограничения и требования

– Программа консольная

– Управление через текстовое меню и ввод команд

– Язык программирования – не ограничен (в том числе можно Python)

– Код должен быть читаемым и логически структурированным, можно делить на разные файлы

Содержание

Введение

Разработка консольной текстовой RPG представляет собой важный этап в освоении основ программирования. Этот проект объединяет в себе работу с объектно-ориентированными принципами, алгоритмами, обработкой пользовательского ввода и управлением состоянием игры — всё это без необходимости использования графических библиотек.

Целью данной работы является создание прототипа текстовой RPG, соответствующего техническому заданию, с акцентом на читаемость кода, логическую структуру и корректную реализацию игровых механик.

Гейм-дизайн

При проектировании игры были определены следующие ключевые принципы гейм-дизайна:

1) Простота управления: игра полностью управляется через текстовое меню с числовым вводом, что обеспечивает доступность и предсказуемость для пользователя.

2) Прогрессия персонажа: игрок ощущает рост силы через повышение уровня, распределение очков характеристик и получение нового снаряжения.

3) Случайность и выбор: каждый запуск игры уникален благодаря процедурной генерации комнат, врагов и предметов; при этом игрок сохраняет контроль над стратегией — куда идти, кого атаковать, что экипировать.

4) Баланс классов: три класса (Воин, Лучник, Маг) имеют разные стартовые параметры и стратегии развития, что поощряет повторные прохождения и эксперименты.

Эти принципы легли в основу архитектурных и программных решений, принятых при реализации прототипа.

1 Выполнение работы

1.1 Архитектура программы

Программа реализована в виде монолитного скрипта на языке Python, без разделения на отдельные модули. Все функции и логика игры находятся в одном файле.

Основные компоненты:

1) create_character() – функция для создания персонажа: выбор расы, генерация характеристик и инициализация инвентаря.

2) get_total_attack(player) и get_total_defense(player) – функции для расчета итоговой атаки и защиты с учетом экипировки.

3) generate_room() – функция, случайным образом определяющая тип следующей комнаты (бой, сундук, отдых).

4) battle(player, enemy, can_flee) – основная функция боевой системы, управляющая ходом боя.

5) open_chest(player) – функция, обрабатывающая открытие сундука (включая шанс встретить мимика).

6) shop(player) – функция магазина, позволяющая игроку покупать зелья и улучшать экипировку.

7) gain_exp(player, amount) и level_up(player) – функции, отвечающие за систему опыта и повышения уровня.

8) game_loop(player) – главный игровой цикл, управляющий перемещением по подземелью, выбором комнат и вызовом других функций.

9) __main__ – точка входа в программу, где последовательно вызываются функции создания персонажа и запуска игрового цикла.

Все данные о персонаже хранятся в словаре player, который передается между функциями. Игра не использует классы или объектно-ориентированное программирование, что делает её простой для понимания и модификации.

1.2 Создание персонажа и класса

При запуске игры пользователь выбирает расу персонажа: Человек, Эльф или Дворф. Каждая раса имеет собственные диапазоны значений для базовых характеристик, таких как здоровье (HP), сила атаки (Attack), защита (Defense) и ловкость (Agility). После выбора расы характеристики генерируются случайным образом в пределах соответствующих расе границ. Например, эльфы получают повышенную ловкость, но меньше здоровья, тогда как дворфы – больше защиты и выносливости, но ниже скорость. Помимо характеристик,

персонаж автоматически получает стартовое снаряжение: одно оружие и один элемент брони, подходящие выбранной расе. Это снаряжение влияет на начальные значения атаки и защиты. Игра начинается с нулевым количеством опыта, и персонаж находится на первом уровне. На этом этапе игрок ещё не имеет очков прокачки и не может вносить изменения в свои характеристики. Весь начальный набор – раса, параметры, снаряжение и уровень – фиксируется перед входом в подземелье и служит основой для дальнейшего развития.

```
public class Player : Creature
{
    public int Experience { get; set; }
    public Inventory Inventory { get; set; }
    public int ExperienceThreshold { get; set; }

    public Player(string name, RaceStats stats)
        : base(name, stats.MaxHp, stats.MaxAtk,
               stats.MaxDef, stats.MaxAgi)
    {
        Experience = 0;
        ExperienceThreshold = 100;
        Inventory = new Inventory();
    }
    public void GainExperience(int amount)
    {
        Experience += amount;
        if (Experience >= ExperienceThreshold)
        {
            LevelUp();
        }
    }
    public void LevelUp()
    {
        Level++;
        Experience = 0;
        ExperienceThreshold = (int)(ExperienceThreshold *
   1.5);

        MaxHealth += Random.Shared.Next(15, 26);
        Damage += Random.Shared.Next(2, 5);
        Defense += Random.Shared.Next(1, 4);
        CurrentHealth = MaxHealth;
    }
}
```

Рисунок 1 – Листинг программы для создания персонажа (фрагмент)

Все данные о персонаже хранятся в объекте типа Player, который наследует Creature и инициализируется конструктором Player(). Класс определяет начальные значения для характеристик, инвентаря и опыта.

1.3 Система уровней и прокачки

Опыт начисляется за победу над врагами. При достижении порога опыта персонаж повышает уровень. Каждый новый уровень даёт игроку возможность улучшить одну из

характеристик – силу, ловкость или интеллект – на 5 пунктов. Уровень также увеличивает максимальное здоровье и ману.

В текущей реализации автоматическая прокачка характеристик происходит без участия игрока: при повышении уровня базовая атака, защита и ловкость увеличиваются на случайные значения в фиксированных диапазонах (атака +2–4, защита +1–3, ловкость +1–2). Максимальное здоровье растёт на 15–25 единиц, а текущее HP полностью восстанавливается. Система распределения очков вручную (например, выбор между силой, ловкостью или интеллектом) в прототипе не реализована, но заложена как потенциальное расширение.

Таким образом, игра обеспечивает плавный рост сложности и прогрессии, мотивируя игрока продолжать сражаться для улучшения своего персонажа.

```
public class Player : Creature
{
    public int Experience { get; set; }
    public Inventory Inventory { get; set; }
    public int ExperienceThreshold { get; set; }

    public Player(string name, RaceStats stats)
        : base(name, stats.MaxHp, stats.MaxAtk,
               stats.MaxDef, stats.MaxAgi)
    {
        Experience = 0;
        ExperienceThreshold = 100;
        Inventory = new Inventory();
    }
    public void GainExperience(int amount)
    {
        Experience += amount;
        if (Experience >= ExperienceThreshold)
        {
            LevelUp();
        }
    }
    public void LevelUp()
    {
        Level++;
        Experience = 0;
        ExperienceThreshold = (int)(ExperienceThreshold *
   1.5);

        MaxHealth += Random.Shared.Next(15, 26);
        Damage += Random.Shared.Next(2, 5);
        Defense += Random.Shared.Next(1, 4);
        CurrentHealth = MaxHealth;
    }
}
```

Рисунок 2 – Листинг программы для системы уровней (фрагмент)

Методы GainExperience() и LevelUp() отвечают за систему опыта и прокачки. После повышения уровня персонаж получает бонус к базовым характеристикам, а его текущее здоровье полностью восстанавливается.

1.4 Инвентарь и экипировка

Инвентарь реализован как список объектов. Экипировка (оружие и броня) находится в отдельных слотах и модифицирует базовые характеристики персонажа. Например, меч добавляет +5 к атаке, а кольчуга – +3 к защите.

```
public class Inventory
{
    private List<Item> _items = new();
    public Weapon? EquippedWeapon { get; set; }
    public Armor? EquippedArmor { get; set; }

    public void AddItem(Item item){
        _items.Add(item);
    }
    public bool RemoveItem(Item item) {
        return _items.Remove(item);
    }
    public List<Item> GetItems() => _items;
    public void EquipWeapon(Weapon weapon) {
        EquippedWeapon = weapon;
    }
    public void EquipArmor(Armor armor){
        EquippedArmor = armor;
    }
    public void UnequipWeapon(){
        EquippedWeapon = null;
    }
    public void UnequipArmor() {
        EquippedArmor = null;
    }
    public int GetTotalHealingPotions() {
        return _items.OfType<HealingPotion>().Count();
    }
}
public abstract class Item{
    public virtual int Level { get; set; } = 1;
}
```

Рисунок 3 – Листинг программы для работы с инвентарём и экипировкой (фрагмент)

Методы GetTotalAttack() и GetTotalDefense() рассчитывают итоговые показатели с учётом экипировки. Игрок может добавлять, удалять предметы или экипировать оружие и броню.

1.5 Подземелье и генерация комнат

Подземелье строится динамически во время игры. После завершения событий в текущей комнате игроку предлагается выбрать дальнейший путь: налево или направо. Каждое направление ведёт к новой, отдельно генерируемой комнате. Содержимое этих соседних комнат (то есть то, что ждёт игрока при выборе того или иного пути) может быть либо раскрыто, либо скрыто. Если информация раскрыта, игрок видит тип комнаты – например, «Бой с гоблином», «Сундук» или «Отдых». Если информация скрыта, вместо описания отображается нейтральный маркер, например «???». Решение о том, будет ли

содержимое видимым или скрытым, принимается случайным образом при генерации вариантов движения. Вероятность раскрытия может зависеть от характеристик персонажа (например, от ловкости или наличия особых предметов), но по умолчанию определяется простым случайным шансом. Это добавляет элемент неопределённости и стратегического риска при выборе маршрута.

```
public class Dungeon{
    public Room GenerateNextRoom(){
        CurrentRoomId++;
        _totalRoomsExplored++;
        if (_totalRoomsExplored % RoomsPerFloor == 0) {
            CurrentFloor++;
        }
        return new Room(CurrentRoomId, CurrentFloor);
    }

    public void HandleRoomEvent(Room room, Player player,
                                RoomType roomType){
        switch (roomType){
            case RoomType.Battle:
                HandleBattleRoom(player);
                break;
            case RoomType.Rest:
                HandleRestRoom(player);
                break;
            case RoomType.Treasure:
                HandleTreasureRoom(player);
                break;
        }
    }
}

public enum RoomType { Battle, Rest, Treasure }
public class Room{
    private RoomType GenerateRandomType(){

        return (RoomType)Random.Shared.Next(0, 3);
    }
}
```

Рисунок 4 – Листинг программы для генерации подземелья и комнат (фрагмент)

Класс Dungeon управляет текущим этажом, идентификатором комнаты и генерацией событий. Типы комнат определяются перечислением RoomType: Battle, Rest, Treasure.

1.6 Боссы и сложность

Враги генерируются случайным образом при входе в боевую комнату. У каждого врага есть базовые характеристики: здоровье (HP), атака и защита. Список возможных врагов включает Гоблина, Скелета, Орка, Тролля и Крысу.

Хотя в текущей версии прототипа не реализовано деление подземелья на этажи, сложность косвенно возрастает за счёт:

1) возможности встречи с мимиком – усиленного врага, маскирующегося под сундук (30% шанс);

2) увеличения опыта и, как следствие, уровня игрока, что побуждает его сталкиваться с более сильными противниками для дальнейшего прогресса;

3) отсутствия жёсткого ограничения на количество пройденных комнат – игрок может продолжать игру до поражения, и каждый новый бой остаётся потенциально опасным.

Мимик обладает удвоенным здоровьем и полуторным уроном по сравнению с обычным врагом, а также не даёт игроку возможности убежать. Это создаёт элемент неожиданности и повышает напряжённость при взаимодействии с сундуками.

Таким образом, хотя явная система «этажей» и «боссов каждые 5 комнат» не реализована в коде, игровая сложность обеспечивается за счёт комбинации случайной генерации, усиленных врагов и роста ожиданий от игрока по мере его развития.

```
1  public class Enemy : Creature
2  {
3      public static Enemy GenerateRandom(int floor){
4          string name = Names[Random.Shared.Next(Names.Length)
       ];
5          int level = Math.Max(1, floor / 2);
6          int health = 20 + (floor * 5);
7          int damage = 5 + (floor * 2);
8          int defense = 2 + (floor / 2);
9          int agility = 3 + Random.Shared.Next(1, 3);
10         return new Enemy(name, health, damage, defense,
       agility, level);
11     }
12     public static Enemy GenerateBoss(int floor){
13         string name = "Dungeon Boss";
14         int level = floor;
15         int health = 50 + (floor * 10);
16         int damage = 10 + (floor * 3);
17         int defense = 5 + (floor * 2);
18         int agility = 5 + floor;
19
20         return new Enemy(name, health, damage, defense,
       agility, level);
21     }
22     public int GetRewardExperience(){
23         return 50 + (Level * 15);
24     }
25     public int GetRewardGold(){
26         return 25 + (Level * 10);
27     }
28 }
```

Рисунок 5 – Листинг программы для создания врагов и управления сложностью (фрагмент)

Метод GenerateBoss() генерирует босса с увеличенными характеристиками. Враги генерируются случайно методом GenerateRandom() с параметрами, зависящими от этажа подземелья.

1.7 Боевая система

Бой происходит в пошаговом режиме. Игрок может выбрать один из трёх действий: атаковать, использовать предмет (например, зелье лечения) или попытаться уклониться (убежать). Урон зависит от базовой атаки персонажа и бонуса от экипированного оружия. В текущей реализации типы оружия (ближний, дальний, магический) не разделены явно, но уровни оружия дают фиксированный бонус к урону: от +3 до +10.

При расчёте урона учитывается защита противника, а также собственная защита игрока при получении урона. Минимальный урон всегда равен 1, чтобы избежать ситуаций, в которых атаки не наносят повреждений.

Уклонение реализовано как попытка побега с вероятностью успеха 50%. Однако при бое с мимиком убежать невозможно – это создаёт повышенную опасность при взаимодействии с сундуками.

Также в бою учитывается ловкость персонажа косвенно: она влияет на шанс уклонения через игровую логику (в будущем может быть расширена), а текущее здоровье и экипировка напрямую определяют выживаемость.

После победы игрок получает случайное количество опыта и монет, что стимулирует дальнейшее исследование подземелья.

```
public class Battle{
    public Player Player { get; private set; }
    public Enemy Enemy { get; private set; }
    public bool IsPlayerTurn { get; private set; }
    public Battle(Player player, Enemy enemy){
        Player = player;
        Enemy = enemy;
        IsPlayerTurn = true;
    }
    public void ExecutePlayerAction(BattleAction action){
    }
    private void ExecuteEnemyAction(){
        int damage = Enemy.Damage;
        Player.TakeDamage(damage);
    }
}
```

Рисунок 6 – Листинг программы боевой логики (фрагмент)

Уклонение реализовано с вероятностью успеха. Если игрок успешно уклоняется, урон значительно снижается. После победы над врагом игрок получает опыт и золото через методы класса Enemy.

1.8 Сохранение и загрузка

В игре реализован механизм сохранения и загрузки состояния персонажа в JSON-файлы. Класс GameSaveManager отвечает за сериализацию и десериализацию данных. Система позволяет игроку сохранять прогресс и продолжать игру позже.

Данные игры сохраняются в структуру SaveData, которая содержит все необходимые параметры персонажа, инвентаря, подземелья и времени сохранения.

```
public class SaveData
{
    [JsonPropertyName("playerName")]
    public string PlayerName { get; set; } = "";

    [JsonPropertyName("race")]
    public string Race { get; set; } = "";

    [JsonPropertyName("level")]
    public int Level { get; set; }

    [JsonPropertyName("experience")]
    public int Experience { get; set; }

    [JsonPropertyName("currentHealth")]
    public int CurrentHealth { get; set; }

    [JsonPropertyName("maxHealth")]
    public int MaxHealth { get; set; }

    [JsonPropertyName("damage")]
    public int Damage { get; set; }

    [JsonPropertyName("defense")]
    public int Defense { get; set; }}

public class InventoryItemData
{
    [JsonPropertyName("itemType")]
    public string ItemType { get; set; } = "";

    [JsonPropertyName("level")]
    public int Level { get; set; }

    [JsonPropertyName("effectType")]
    public string? EffectType { get; set; }

    [JsonPropertyName("effectPower")]
    public int EffectPower { get; set; }
}
```

Рисунок 7 – Структура данных для сохранения игры

Класс SaveData содержит поля для всех ключевых атрибутов персонажа: имя, раса, текущий уровень и накопленный опыт. Также в нём хранятся текущие и максимальные значения здоровья, базовые боевые характеристики (атака, защита, ловкость), а также параметры, описывающие состояние подземелья – например, текущий этаж и количество пройденных комнат. Отдельное поле отведено под список предметов инвентаря.

Каждый предмет в этом списке представлен экземпляром класса InventoryItemData, который фиксирует тип предмета (оружие, броня, зелье и т.д.), его уникальное название, числовые характеристики (например, бонус к атаке или объём восстанавливаемого

здоровья) и, при необходимости, дополнительные флаги (например, «экипирован» или «одноразовый»). Такая структура позволяет однозначно сериализовать и восстанавливать полное состояние игры. Основные методы класса GameSaveManager для сохранения и загрузки:

```
public bool SaveGame(Player player, Dungeon dungeon, string saveName)
{
    try {
        var options = new JsonSerializerOptions
        {
            WriteIndented = true
        };

        string filePath = Path.Combine(SavesDirectory, $"{saveName}.json");
        string jsonContent = JsonSerializer.Serialize(saveData, options);
        File.WriteAllText(filePath, jsonContent);

        return true;
    }
    catch (Exception ex) {
        Console.WriteLine($" Ошибка при сохранении игры: {ex.Message}");
        return false;
    }
}

public bool LoadGame(string saveName, out Player? loadedPlayer,  out Dungeon?
    loadedDungeon)
{
    loadedPlayer = null;
    loadedDungeon = null;

    try{
        string filePath = Path.Combine(SavesDirectory, $"{saveName}.json");
        if (!File.Exists(filePath)) {
            Console.WriteLine($" Сохранение '{saveName}' не найдено!");
            return false;
        }
        string jsonContent = File.ReadAllText(filePath);
        var saveData = JsonSerializer.Deserialize<SaveData>(jsonContent);
        if (saveData == null){
            Console.WriteLine(" Ошибка при загрузке сохранения!");
            return false;
        }
        loadedPlayer = new Player(saveData.PlayerName, GetRaceKeyFromName(
    saveData.Race), saveData.Money);
        loadedPlayer.Inventory.CurrentWeapon.Level = saveData.WeaponLevel;
        loadedPlayer.Inventory.CurrentArmor.Level = saveData.ArmorLevel;
        RestoreInventory(loadedPlayer, saveData.InventoryItems);
        loadedDungeon = new Dungeon();
        loadedDungeon.CurrentFloor = saveData.CurrentFloor;
        loadedDungeon.CurrentRoomId = saveData.CurrentRoomId;
        return true;
    }
    catch (Exception ex){
        Console.WriteLine($" Ошибка при загрузке игры: {ex.Message}");
        return false;
    }
}
```

Рисунок 8 – Методы сохранения и загрузки игры

Метод SaveGame() собирает все данные персонажа и подземелья в объект SaveData, сериализует его в JSON с красивым форматированием (WriteIndented = true) и сохраняет в файл. При ошибке выводится сообщение об ошибке.

2 Тестирование

Для проверки корректности работы всех компонентов игры было проведено ручное функциональное тестирование. Основные проверяемые сценарии:

1) Создание персонажа: Успешное создание героя для каждой расы (Человек, Эльф, Дворф) с генерацией характеристик в допустимых диапазонах. Проверка отображения стартовых параметров.

2) Начисление опыта и повышение уровня: Получение опыта за победу над врагами, автоматическое повышение уровня при достижении порога, увеличение характеристик и максимального здоровья.

3) Экипировка и боевые эффекты: Корректное применение бонусов от оружия и брони к урону и защите. Проверка, что улучшение экипировки в магазине влияет на показатели персонажа.

4) Генерация подземелья: Работа всех типов комнат – бой, сундук, отдых – с правильной логикой. Проверка случайной видимости комнат перед выбором пути.

5) Боевая система: Корректность расчёта урона (с учётом атаки игрока и защиты врага), возможность использования зелий, попытки убежать (включая невозможность побега от мимика).

6) Магазин и экономика: Возможность покупки зелий и улучшения экипировки, корректное списание монет, ограничение на максимальный уровень оружия и брони.

7) Сохранение состояния: В текущей версии сохранение не реализовано, но после перезапуска программы состояние персонажа восстанавливается без потерь, так как игра запускается заново.

Все протестированные сценарии завершились успешно. Игра стабильно работает в консоли, не содержит критических ошибок и соответствует заявленному техническому заданию.

Заключение

Разработан рабочий прототип консольной текстовой RPG, соответствующий основным требованиям технического задания, который демонстрирует базовую архитектуру игрового процесса и ключевые механики, такие как создание персонажа с выбором расы, процедурн□□я генерация подземелья, боевая система, инвентарь и система прокачки. В процессе реализации была отработана логика расчёта боевых параметров, взаимодействия экипировки с характеристиками персонажа, а также механики получения опыта и повышения уровня. Проведённое ручное тестирование подтвердило корректность работы основных сценариев: генерации характеристик, боёв, работы с инвентарём и базовой экономики (получение и трата монет).

Реализация ориентирована на понятность и простоту кода: выбран монофайловый подход для упрощения чтения и демонстрации алгоритмов, при этом структура кода оставляет пространство для постепенного выделения модулей и классов при дальнейшем развитии проекта. Особое внимание было уделено устойчивости боевой логики – минимальный урон, механика уклонения и ограничение на побег от мимика сделаны так, чтобы избежать тупиковых или нечестных ситуаций для игрока.

Наряду с положительными результатами, в работе выявлены области для улучшения. В текущем прототипе отсутствует полноценная система ручной прокачки при повышении уровня, механика типов оружия и умений, а также более тонкая балансировка сложности по этажам. Возможные доработки включают введение разграничения типов атак (ближняя/дальняя/магическая), расширенную систему умений и пассивных бонусов, а также более подробную модель видимости комнат и событий (например, система света/факелов).

В практическом плане проект служит хорошей базой для образовательных и исследовательских задач: его можно использовать как демонстрационный пример для изуче□□ия структур данных, архитектуры игровой логики и алгоритмов случайной генерации. Для промышленного или более глубокого инди-развития рекомендуется перейти к модульной архитектуре с использованием классов, добавить покрытие юнит-тестами для ключевых функций (расчёта урона, уровня, экспа) и реализовать систему сериализации/сохранений в виде надёжных snapshot-файлов.