Geekbrains

**Создание 2D RPG игры «Wizardo» на языке Python с использованием средств библиотеки Pygame**

Программа: Python-разработчик. Специалист. Смешанный формат

Специализация: Разработчик — Программист

ФИО: Рязанцев Иван Александрович

г.Азов

2025 г.

**Оглавление**

[Введение. 4](#_Toc187070742)

[2. Основы создания компьютерных игр на Python. 6](#_Toc187070743)

[2.1. Обзор существующих игр на Python. 6](#_Toc187070744)

[2.2. Технологии разработки игр на Python. 8](#_Toc187070745)

[2.3. Анализ библиотеки Pygame для создания игр. 10](#_Toc187070746)

[3. Проектирование игрового приложения. 13](#_Toc187070747)

[3.1. Постановка задач проекта 13](#_Toc187070748)

[3.2. Определение жанра игры 16](#_Toc187070749)

[3.3. Разработка игровой механики 19](#_Toc187070750)

[3.4. . Создание концепции персонажей и сюжета 22](#_Toc187070751)

[4. Реализация игры. 25](#_Toc187070752)

[4.1. Разработка пользовательского интерфейса. 25](#_Toc187070753)

[4.2. Создание игровых уровней и окружения. 30](#_Toc187070754)

[4.3. Написание кода на Python 34](#_Toc187070755)

[5. Тестирование и отладка. 40](#_Toc187070756)

[5.1. Методология тестирования. 40](#_Toc187070757)

[5.2. Проведение тестов и анализ результатов. 41](#_Toc187070758)

[Заключение 43](#_Toc187070759)

[Список используемой литературы 45](#_Toc187070760)

[Приложения 46](#_Toc187070761)

# Введение.

Тема «Создание 2D-игры в жанре ролевой игры с помощью Python и Pygame» затрагивает всё более актуальный аспект современных развлечений — разработку игр. В последние годы игры стали неотъемлемой частью досуга и отдыха миллионов людей по всему миру. Среди различных жанров особое место занимают ролевые игры (RPG), предоставляющие игрокам возможность погрузиться в захватывающие миры, развивать персонажей и принимать решения, которые существенно влияют на сюжетную линию. Создание 2D-игры в жанре ролевой игры — захватывающая задача, требующая как технических, так и творческих навыков, а также глубокого понимания игровой механики.

В этой работе мы рассмотрим библиотеку Pygame, которая предлагает мощные инструменты для разработки игр на Python. Pygame позволяет создавать интерактивные и визуально привлекательные игры благодаря своим обширным функциям, таким как обработка событий, управление графикой и звуком, а также анимация. Использование Pygame упрощает процесс разработки, предоставляя разработчикам удобный интерфейс для взаимодействия с игровыми объектами.

Основная цель этой дипломной работы — создать 2D-ролевую игру, в которой игроки смогут управлять персонажем, сражаться с врагами и использовать различные способности и предметы. В процессе разработки будут рассмотрены такие аспекты, как дизайн уровней, создание персонажей, боевая механика и взаимодействие с неигровыми персонажами (NPC). Кроме того, будут проанализированы методологии, использованные при разработке, и принципы игрового дизайна, что будет способствовать созданию высококачественной игры.

В результате этой работы целью является не только разработка функционирующей игры, но и создание структурированного кода, который может служить образовательным ресурсом для студентов, интересующихся разработкой игр с использованием Python и Pygame 9.

Таким образом, это исследование улучшит навыки автора в программировании и разработке игр, а также станет ценным ресурсом для сообщества разработчиков, заинтересованных в создании 2D-ролевых игр на Python.

**Инструменты:** PyCharm Community Edition 2022.2.1, Pygame,Git, Draw.io.

**Состав команды**: Рязанцев Иван Александрович (разработчик).

# 2. Основы создания компьютерных игр на Python.

# 2.1. Обзор существующих игр на Python.

Python зарекомендовал себя как мощный и универсальный язык программирования в сфере разработки игр. С помощью Python было создано множество успешных игр, демонстрирующих его возможности и гибкость. В этом разделе мы рассмотрим несколько известных игр, разработанных на Python, и познакомимся с различными жанрами и применением этого языка при создании игр.

Одной из самых известных игр, использующих Python, является Civilization IV — пошаговая стратегическая игра, разработанная Firaxis Games. В этой игре игроки управляют цивилизацией, начиная с античных времён и до наших дней, принимая стратегические решения для расширения своей империи. Python использовался в основном для интерфейса модификации игры, позволяя игрокам настраивать и создавать собственный контент, тем самым повышая долговечность и реиграбельность игры.

Ещё одна важная игра — Battlefield 2, шутер от первого лица (FPS), разработанный компанией Digital Illusions CE. Сценарии на Python управляют различными игровыми механиками, включая игровые режимы и системы подсчёта очков. Использование Python в этом контексте демонстрирует его эффективность для серверной логики в сложных многопользовательских средах, способствуя привлечению игроков.

EVE Online, массовая многопользовательская онлайн-игра (MMO), действие которой разворачивается в научно-фантастической вселенной, также широко использует Python. В игре используется вариант Python, известный как Stackless Python, который облегчает одновременную работу огромного количества игроков без ограничений традиционной многопоточности. Это инновационное использование Python позволило EVE Online поддерживать сложный, динамичный мир с высоким уровнем взаимодействия игроков.

В сфере инди-игр Frets on Fire — популярная ритм-игра, разработанная с помощью Python. Этот проект с открытым исходным кодом позволяет игрокам имитировать игру на музыкальных инструментах, следуя подсказкам на экране. Его разработка с помощью Pygame показывает, насколько Python может быть доступным для начинающих разработчиков игр, позволяя им создавать увлекательные и креативные игры.

Наконец, Darkest Dungeon 2, продолжение известной ролевой игры в жанре «рогалик», использует Python для различных аспектов разработки, включая игровую механику и написание сценариев. Эта игра — пример того, как Python можно использовать для реализации сложных игровых систем, сохраняя при этом элегантную и управляемую кодовую базу.

Разнообразие этих игр иллюстрирует способность Python адаптироваться к разным жанрам и его потенциал как инструмента для разработки игр. Благодаря широкому спектру библиотек и фреймворков, таких как Pygame, разработчики могут создавать увлекательные игры, отвечающие ожиданиям современных игроков. Поскольку Python продолжает развиваться, он остаётся предпочтительным выбором как для начинающих, так и для опытных разработчиков игр, стремящихся воплотить свои идеи в жизнь.

# 2.2. Технологии разработки игр на Python.

Разработка компьютерных игр на Python поддерживается различными технологиями, фреймворками и библиотеками, которые упрощают создание увлекательных игр. В этом разделе представлены некоторые из наиболее популярных инструментов для разработки игр на Python, выделив их особенности и области применения.

Pygame: Pygame, пожалуй, самая известная библиотека для разработки игр на Python. Она предоставляет разработчикам доступ к графике, звуку и обработке ввода. Pygame предназначен для создания 2D-игр и имеет простой API, что делает его доступным для новичков и в то же время достаточно гибким для опытных разработчиков. Его сильные стороны — простота использования, обширная документация и активное сообщество, которое предоставляет множество обучающих материалов и ресурсов.

Pyglet: Pyglet — ещё одна надёжная библиотека для разработки игр, способная работать как с 2D-, так и с простой 3D-графикой. Она особенно хорошо подходит для создания визуально насыщенных приложений, поскольку включает поддержку OpenGL — API для рендеринга графики, широко используемого в разработке игр. Pyglet обеспечивает бесперебойную работу с мультимедиа, позволяя разработчикам встраивать в свои игры аудио, видео и различные форматы изображений без дополнительных зависимостей.

Panda3D: Panda3D — это комплексный движок для разработки 3D-игр, который предлагает широкий спектр функций, включая физику, обнаружение столкновений и графический рендеринг. Он позволяет разработчикам создавать захватывающие среды, используя Python для написания скриптов. Архитектура Panda3D поддерживает быстрое создание прототипов и производство 3D-приложений, что делает его популярным среди независимых разработчиков и образовательных учреждений.

Kivy: Kivy — это библиотека Python с открытым исходным кодом, предназначенная для разработки кроссплатформенных приложений, в том числе игр. Основное внимание в ней уделяется поддержке мультитач, созданию пользовательских интерфейсов и простоте настройки. Kivy подходит для разработки мобильных и настольных приложений, а его гибкость в создании графических пользовательских интерфейсов делает его мощным инструментом для разработчиков, желающих создавать визуально привлекательные игры с сенсорным управлением и жестами.

Ren’Py: Ren’Py — это специализированный движок для создания визуальных новелл и сюжетных игр. Созданный на Python, он позволяет разработчикам создавать разветвленные сюжетные линии, интегрируя изображения и звук. Его простой язык сценариев упрощает процесс создания сложных сюжетных линий, что делает его любимым среди разработчиков, ориентированных на повествовательный опыт.

PyOpenGL: PyOpenGL обеспечивает привязку к OpenGL, позволяя разработчикам Python получить доступ к одной из самых мощных доступных графических библиотек. Эта технология необходима для создания высокопроизводительных 3D-игр и симуляторов. С помощью PyOpenGL разработчики могут использовать передовые методы рендеринга, создавая богатые визуальные эффекты и сложные среды.

Каждая из этих технологий предлагает уникальные возможности, которые подходят для разных аспектов разработки игр. Выбор подходящих инструментов часто зависит от конкретных требований проекта, таких как целевая аудитория, игровая механика и желаемый визуальный стиль. Поскольку популярность Python в игровом сообществе продолжает расти, эти технологии будут играть ключевую роль в формировании будущего разработки игр, позволяя как начинающим, так и опытным разработчикам воплощать свои творческие идеи в жизнь.

# 2.3. Анализ библиотеки Pygame для создания игр.

Pygame зарекомендовала себя как одна из ведущих библиотек для разработки игр на Python, особенно для 2D-проектов. В этом разделе мы проведём анализ Pygame, уделив особое внимание её функциям, сильным и слабым сторонам, а также общей пригодности для разработки различных типов игр.

Основные функции: Pygame предлагает широкий набор функций, отвечающих потребностям разработчиков игр. Он предоставляет функциональные возможности для рендеринга графики, воспроизведения звука и обработки входных данных, которые необходимы для создания интерактивных игр. Ключевыми компонентами Pygame являются:

Графика: Pygame поддерживает как 2D-изображения, так и анимацию спрайтов, что позволяет разработчикам создавать визуально динамичные игры. Он использует простой DirectMedia Layer (SDL) для эффективной обработки графики.

Звук: Библиотека поддерживает различные звуковые форматы, что позволяет разработчикам добавлять музыку и звуковые эффекты для улучшения игрового процесса.

Обработка событий: Pygame использует архитектуру, управляемую событиями, которая позволяет эффективно управлять вводом данных, включая события с клавиатуры и мыши, что облегчает адаптивный игровой процесс.

Сильные стороны: Pygame обладает несколькими преимуществами, которые делают ее привлекательным выбором для разработки игр:

Простота использования: простой API Pygame удобен для начинающих, что делает его доступным для новичков в программировании и разработке игр. Эта простота позволяет быстро создавать прототипы и сокращает время обучения.

Активное сообщество: Pygame может похвастаться активным сообществом разработчиков, которые активно публикуют учебные пособия, документацию и примеры проектов. Такая поддержка сообщества бесценна для новичков, которым нужны рекомендации и ресурсы.

Кроссплатформенная совместимость: Pygame является кроссплатформенной программой, позволяющей запускать игры на Windows, macOS и Linux без значительных изменений в коде. Благодаря такой универсальности разработчикам проще охватить более широкую аудиторию.

Недостатки: несмотря на свои преимущества, Pygame имеет некоторые ограничения:

Производительность: хотя Pygame подходит для 2D-игр, он может работать не так эффективно, как другие движки, когда дело касается более ресурсоёмких приложений. Разработчики могут столкнуться с проблемами производительности в сложных играх с большим количеством спрайтов или продвинутой графикой.

Ограниченная поддержка 3D: Pygame в первую очередь ориентирован на разработку 2D-игр и не имеет встроенных инструментов для 3D-графики. Хотя можно реализовать 3D-функционал с помощью сторонних библиотек или интеграций, эти решения могут быть не такими простыми, как использование специализированных 3D-движков, таких как Panda3D или Unity.

Подходят для разработки игр: Pygame особенно хорошо подходит для разработки аркадных игр, платформеров и образовательных проектов. Его удобный интерфейс способствует экспериментам, что делает его отличным инструментом для любителей и новичков в разработке игр. Кроме того, многие образовательные учреждения используют Pygame для обучения концепциям программирования с помощью игровых методов обучения.

В заключение отметим, что Pygame является мощной и доступной библиотекой для разработки 2D-игр на Python. Несмотря на то, что у неё могут быть определённые ограничения в производительности и 3D-возможностях, простота использования, поддержка сообщества и широкая совместимость с платформами делают её предпочтительным выбором для многих разработчиков, стремящихся создавать увлекательные игры. Поскольку Python продолжает развиваться и набирать популярность в игровой индустрии, Pygame останется ценным ресурсом как для начинающих, так и для опытных создателей игр, стремящихся воплотить свои идеи в жизнь.

# 3. Проектирование игрового приложения.

# 3.1. Постановка задач проекта

Успешная разработка игрового приложения во многом зависит от чёткого определения задач проекта. В этом разделе описан процесс постановки целей и разбивки проекта на выполнимые задачи, которые определяют ход разработки игры от концепции до завершения.

Определение концепции игры: первым шагом в определении задач проекта является определение основной концепции игры. Это включает в себя определение жанра (например, платформер, головоломка, ролевая игра), целевой аудитории и уникальных особенностей. Определение концепции игры поможет оптимизировать последующие задачи и решения на протяжении всего процесса разработки.

Постановка целей: Как только концепция игры разработана, важно установить конкретные, измеримые, достижимые, актуальные и ограниченные по времени (SMART) цели. Например, цели могут включать:

Завершение создания игрового прототипа в течение трех месяцев.

Интеграция ключевых игровых механик, таких как перемещение персонажа или повышение уровня, к определенному сроку.

Достижение определенного эстетического стиля или игрового баланса в соответствии с отзывами игроков во время тестирования.

Создание структуры задач: Разбивка проекта на более мелкие, управляемые задачи имеет решающее значение для эффективного управления проектом. Это может быть достигнуто путем создания структуры разбивки работ (WBS). Некоторые распространенные категории задач включают в себя:

Задачи по дизайну: разработка визуальных и звуковых элементов игры, таких как дизайн персонажей, фон и звуковые эффекты. Кроме того, в эту категорию входит создание игровой механики и макетов пользовательского интерфейса (UI).

Задачи по разработке: написание кода для реализации игровых функций, интеграция ресурсов и разработка архитектуры системы. Сюда также может входить разработка искусственного интеллекта (ИИ) и моделирование физики.

Задачи по тестированию: разработка протоколов тестирования для обеспечения бесперебойной работы игры и отсутствия ошибок. Сюда входит модульное тестирование, интеграционное тестирование и пользовательское тестирование для сбора отзывов о игровом процессе и общем пользовательском опыте.

Задачи по маркетингу: планирование продвижения игры, включая создание маркетинговой стратегии, разработку веб-сайта и подготовку к запуску. Это также может включать вовлечение сообщества через социальные сети и публичные демонстрации.

Установление сроков и контрольных точек: установление временных рамок для каждой задачи помогает поддерживать проект в рабочем состоянии. Установление четких контрольных точек позволяет команде разработчиков отслеживать прогресс и отмечать достижения. Например, важной вехой может быть завершение первого уровня игры или выпуск бета-версии для тестирования игроками.

Распределение ролей и обязанностей: Важно распределять роли и обязанности между членами команды на основе их навыков и опыта. Это гарантирует, что каждым аспектом процесса разработки игры будет заниматься квалифицированный специалист, что может повысить качество конечного продукта. Общие роли могут включать в себя:

Менеджер проекта: Курирует весь проект, обеспечивая выполнение задач в срок и в рамках объема.

Геймдизайнер: занимается механикой игрового процесса, дизайном уровней и общим опытом работы с пользователем.

Программист: отвечает за написание кода и интеграцию всех игровых систем.

Художник: Создает визуальные элементы, включая персонажей, окружение и компоненты пользовательского интерфейса.

Звуковой дизайнер: разрабатывает аудиоэлементы, такие как музыка и звуковые эффекты.

Постоянный контроль и корректировка: регулярный контроль хода проекта и корректировка задач по мере необходимости крайне важны для соблюдения сроков. Это может включать в себя уточнение целей на основе отзывов о тестировании, перераспределение ресурсов или адаптацию сроков к непредвиденным трудностям.

Таким образом, этап определения задач при разработке проекта является основополагающим для успеха игрового приложения. Определив концепцию игры, поставив чёткие цели, разбив проект на выполнимые задачи, распределив роли и установив сроки, разработчики могут создать структурированный подход к разработке игры, который повысит эффективность и производительность. Четко сформулированный план проекта не только направляет команду разработчиков, но и подготавливает почву для успешного запуска игры.

# 3.2. Определение жанра игры

Определение жанра игры — важнейший этап в процессе разработки, поскольку он влияет на многие аспекты, включая механику игрового процесса, художественный стиль, структуру повествования и маркетинговую стратегию. В этом разделе рассматривается важность определения жанра и объясняется, как эффективно классифицировать игру в рамках этой концепции.

Понимание игровых жанров: игровые жанры — это классификации, которые объединяют игры в группы на основе схожих механик игрового процесса, тем и целей. К распространённым жанрам относятся экшн, приключения, ролевые игры (RPG), симуляторы, стратегии, спортивные игры и головоломки. У каждого жанра есть свои особенности и ожидания, которые могут помочь в процессе разработки игры и определить целевую аудиторию.

Важность определения жанра: определение жанра игры в начале работы над проектом служит нескольким ключевым целям:

Определение основных элементов дизайна: жанр определяет основные игровые механики. Например, экшн-игра будет делать упор на быстрый игровой процесс и рефлексы, в то время как логическая игра будет сосредоточена на решении проблем и стратегии.

Ориентация на аудиторию: разные жанры рассчитаны на разные демографические группы. Определяя жанр, разработчики могут адаптировать свои маркетинговые усилия и игровой опыт для привлечения желаемой аудитории игроков.

Определение ожиданий индустрии: к играм в рамках устоявшихся жанров предъявляются определенные требования. Понимание этих норм позволяет разработчикам либо соответствовать, либо стратегически подрывать эти ожидания, создавая инновационные возможности.

Исследование и анализ: Чтобы эффективно определить игровой жанр, разработчики должны провести тщательное исследование и анализ:

Маркетинговые исследования: проанализировать успешные игры в рамках предполагаемого жанра, чтобы понять популярные тенденции, механику и эстетику дизайна. Это может дать представление о том, что находит отклик у игроков и какие новые жанры могут предоставить возможности.

Конкурентный анализ: Изучите игры конкурентов, чтобы выявить их сильные и слабые стороны. Этот анализ может помочь отличить новую игру от других игр того же жанра, что потенциально может привести к инновационному подходу.

Гибридные жанры: Во многих современных играх сочетаются элементы разных жанров, создавая гибридные впечатления. При определении жанра важно понимать, подходит ли игра к одной категории или включает в себя аспекты из других. Например, игра, сочетающая элементы экшена и RPG, может быть сосредоточена на боевой механике, а также на развитии персонажа и глубине повествования. Четкое определение этих элементов может помочь разработчикам правильно позиционировать игру на рынке и точно определять ожидания игроков.

Итеративный процесс: Определение жанра - это не разовое решение, оно должно пересматриваться и уточняться на протяжении всего процесса разработки. По мере развития игры разработчики могут обнаружить, что их первоначальная жанровая классификация нуждается в корректировке в соответствии с игровым процессом или отзывами игроков. Если сохранять непредвзятость и готовность вносить изменения в жанровые определения, конечный продукт может получиться более качественным.

В заключение отметим, что определение жанра игры — это фундаментальный аспект игрового дизайна, который формирует проект от замысла до завершения. Понимая нюансы жанровой классификации, проводя тщательные исследования, признавая наличие гибридных элементов и сохраняя гибкость на протяжении всего процесса разработки, разработчики могут создать чёткое представление о своей игре, которое найдёт отклик у игроков и выделит её на конкурентном рынке. Такое тщательное определение жанра в конечном счёте способствует успеху игры и удовлетворению её аудитории.

# 3.3. Разработка игровой механики

Разработка игровой механики является важнейшим компонентом игрового дизайна, поскольку эта механика определяет, как игроки взаимодействуют с игровым миром и что составляет основу игрового процесса. В этом разделе описываются процессы, связанные с созданием эффективной игровой механики, важность баланса и итераций, а также методы тестирования и доработки этой механики.

Определение основной механики: первым шагом в разработке игровой механики является определение основных действий и взаимодействий, которые будут определять игровой процесс. Основные механики — это фундаментальные элементы, которые обеспечивают взаимодействие игроков и влияют на общее впечатление от игры. К распространённым примерам относятся:

Передвижение: то, как игроки перемещаются по игровому миру, включая прыжки, бег и использование транспортных средств.

Бой: системы, с помощью которых игроки взаимодействуют с врагами или решают задачи, включая схемы атак, механизмы защиты и системы здоровья.

Управление ресурсами: то, как игроки собирают, используют и управляют ресурсами, такими как здоровье, боеприпасы или внутриигровая валюта. Определение этих ключевых механик в начале процесса проектирования закладывает основу для дальнейшей разработки и помогает формировать другие аспекты игры.

Балансировка механики: Балансировка механики имеет решающее значение для обеспечения того, чтобы игровой процесс оставался приятным и сложным, не вызывая разочарования. Это включает в себя:

Настройка сложности: Механика должна быть тщательно настроена, чтобы поддерживать соответствующий уровень сложности на протяжении всей игры. Этого можно достичь с помощью таких методов, как увеличение сложности игры с противником или постепенное внедрение новых механик по мере продвижения игроков.

Обратная связь с игроками: сбор и анализ отзывов игроков во время тестирования помогает выявить механики, которые могут быть слишком слабыми или слишком сильными, что позволяет разработчикам вносить обоснованные изменения.

Разнообразные стратегии игроков: создание возможностей для разных стилей игры побуждает игроков взаимодействовать с механиками по-разному. Это может сделать игровой процесс более глубоким и повысить реиграбельность.

Итеративное развитие: игровые механики следует разрабатывать итеративно, часто тестируя и совершенствуя их. Такой подход позволяет разработчикам оценивать эффективность механик в режиме реального времени и вносить необходимые изменения. Ключевые методы итеративной разработки включают в себя:

Прототипирование: создание быстрых прототипов для тестирования механик перед их полной реализацией. Прототипирование позволяет быстро экспериментировать и определять, является ли механика увлекательной и интересной.

Игровое тестирование: вовлечение игроков в тестовые сессии для получения прямой обратной связи по поводу механик. Наблюдение за игроками во время их взаимодействия с игрой может выявить проблемы, которые не очевидны для команды разработчиков, что приводит к более эффективным доработкам.

Циклы обратной связи: внедрение систем, в которых действия игроков приводят к немедленной обратной связи, может способствовать повышению вовлеченности. Например, вознаграждение игроков за выполнение заданий или достижение целей может создать приятную атмосферу, способствующую продолжению игры.

Интеграция повествования и механики: взаимодействие между игровой механикой и повествованием имеет жизненно важное значение для создания захватывающего впечатления. Механика должна быть разработана таким образом, чтобы улучшить историю и развитие персонажа. Например, стелс-механика в игре, основанной на повествовании, может добавить напряженности и способствовать сопереживанию персонажам. Обеспечение того, чтобы механика поддерживала сюжет, приведет к созданию целостного впечатления, которое найдет отклик у игроков.

Документация и коммуникация: в процессе разработки важно чётко документировать игровую механику. Это помогает убедиться, что все члены команды понимают, как работает каждая механика и как она вписывается в общий дизайн игры. Использование таких инструментов, как документы по дизайну и блок-схемы, может облегчить коммуникацию и повысить эффективность совместной работы программистов, художников и дизайнеров.

В заключение отметим, что разработка игровой механики — это сложный процесс, который включает в себя определение основных действий, балансировку игрового процесса, создание прототипов и тестирование, интеграцию сюжетных элементов и ведение чёткой документации. Сосредоточившись на создании увлекательной и сбалансированной механики, разработчики могут создать игровой процесс, который увлечёт игроков и будет способствовать общему успеху игры. Продуманный подход к механике не только повышает вовлечённость игроков, но и усиливает заложенные в игру темы и сюжеты.

# 3.4. . Создание концепции персонажей и сюжета

Создание концепций персонажей и сюжета является основополагающим аспектом игрового дизайна, поскольку оно существенно влияет на вовлечённость игроков и общее впечатление от повествования. В этом разделе описывается процесс создания привлекательных персонажей и их вплетение в значимые сюжеты, которые обогащают игровой мир.

Определение архетипов персонажей: первым шагом в создании персонажа является определение архетипов, которые определяют черты характера, роли и отношения в игре. К распространённым архетипам относятся:

Протагонист: Главный персонаж, с помощью которого игроки знакомятся с историей, обычно сталкивающийся с трудностями и продвигающий сюжет вперед.

Антагонист: основной источник конфликта для главного героя, часто воплощающий противоположные ценности или цели.

Второстепенные персонажи: Эти персонажи могут быть союзниками, наставниками или соперниками, каждый из которых добавляет глубины повествованию и улучшает путешествие главного героя. Выявление архетипов помогает создать полноценных персонажей, которые находят отклик у игроков и выполняют важные функции в истории.

Разработка предыстории персонажа: подробная предыстория обогащает персонажей, предоставляя контекст для их мотивов, поведения и отношений. При разработке предыстории учитывайте следующее:

Происхождение: откуда родом персонаж? Какой опыт сформировал его личность?

Цели и мотивы: что движет персонажем? Личные амбиции, желания и страхи должны быть чётко определены, чтобы влиять на его действия на протяжении всей игры.

Отношения: как связи персонажа влияют на его развитие? Учитывайте семейные отношения, дружбу и соперничество, которые усложняют характер персонажа. Эти элементы создают основу для развития персонажа, помогая игрокам эмоционально сблизиться с героями по мере развития сюжета.

Создание сюжетной структуры: сюжетная структура обеспечивает структуру повествования, описывая последовательность событий и отношения между персонажами. Ключевые компоненты включают:

Структуру сюжета: использование узнаваемой структуры, такой как трёхактная модель или путь героя, помогает создать целостную сюжетную линию. Сюда входят завязка, конфликт, кульминация и развязка.

Конфликт: создание явных конфликтов — внутренних (внутри персонажа) или внешних (между персонажами или силами) — движет сюжетом и способствует развитию персонажей.

Темы: определение основных тем сюжета, таких как искупление, дружба или предательство, добавляет глубины повествованию и влияет на решения и результаты персонажей.

Интеграция игрового процесса с повествованием: взаимодействие между игровой механикой и повествованием имеет решающее значение для создания захватывающего впечатления. Персонажи и их сюжетные линии должны быть переплетены с элементами игрового процесса, позволяя игрокам влиять на ход истории посредством своего выбора. Считать:

Разветвление повествования: Разработка вариантов, влияющих на развитие персонажа и исход истории, поощряет волю игрока и повышает реиграбельность.

Игровой процесс, основанный на персонажах: механика, основанная на чертах характера, таких как особые способности или уникальные взаимодействия, должна соответствовать сюжету, чтобы создать единое впечатление. Обеспечение того, чтобы игровой процесс и повествование поддерживали друг друга, повышает общую вовлеченность и удовлетворенность игроков.

Итеративная разработка и обратная связь: Создание персонажей и сюжетных концепций должно быть итеративным процессом. Регулярные игровые тесты и сеансы обратной связи позволяют команде разработчиков оценить, насколько хорошо персонажи находят отклик у игроков и насколько эффективно развивается история. Важные соображения включают:

Связь с игроком: Сбор отзывов игроков о взаимоотношениях с персонажами и эмоциональных вложениях может помочь внести необходимые коррективы в дизайн персонажа и ход повествования.

Корректировка элементов повествования. На основе отзывов разработчиков может потребоваться доработка мотивов персонажей, конфликтов или сюжетных поворотов для повышения вовлечённости и ясности.

В заключение отметим, что создание концепций персонажей и сюжета — важная часть разработки игры, которая включает в себя создание привлекательных архетипов персонажей, обогащение предыстории, создание последовательной структуры повествования, интеграцию игрового процесса с элементами повествования и использование отзывов для последовательного улучшения. Уделяя внимание этим аспектам, разработчики могут создавать запоминающихся персонажей и захватывающие истории, которые увлекают игроков, способствуют установлению эмоциональных связей и улучшают общее впечатление от игры. Хорошо продуманный сюжет не только способствует развитию персонажей, но и усиливает темы и механики игры, создавая единый и захватывающий интерактивный мир.

# 4. Реализация игры.

# 4.1. Разработка пользовательского интерфейса.

В этом разделе рассматривается процесс разработки пользовательского интерфейса, включая принципы проектирования, инструменты и технологии, соображения удобства использования и итеративное тестирование для оптимизации.

Прежде чем приступить к процессу проектирования, важно определить цели пользовательского интерфейса. Пользовательский интерфейс должен быть направлен на улучшение игрового процесса за счет:

1. Повышение удобства использования. обеспечение того, чтобы игроки могли легко перемещаться по меню, получать доступ к инвентарю и понимать игровую механику, имеет первостепенное значение.
2. Улучшение погружения. Пользовательский интерфейс должен органично вписываться в визуальный стиль игры и повествование, усиливая тематические элементы игрового мира.
3. Упрощение обратной связи. Интерфейс должен обеспечивать четкую и немедленную обратную связь с действиями игрока, например, визуальные подсказки при выполнении действия или изменении статуса игры.
4. Принципы дизайна. Хорошо продуманный пользовательский интерфейс основан на нескольких ключевых принципах, обеспечивающих эффективность и удовлетворенность пользователей.
5. Согласованность. Поддержание единообразия в цветах, шрифтах и стилях элементов на протяжении всей игры помогает игрокам лучше освоиться с интерфейсом.
6. Наглядность. Элементы пользовательского интерфейса должны быть легко понятны, с использованием интуитивно понятных значков и надписей, которые точно отображают их функции.
7. Визуальная иерархия. Создание четкой визуальной иерархии привлекает внимание игроков к важным элементам, таким как цели, показатели работоспособности и интерактивные кнопки, повышая их способность принимать обоснованные решения во время игры.
8. Инструменты и технологии. Выбор правильных инструментов для разработки пользовательского интерфейса имеет важное значение для бесперебойного процесса внедрения. К числу популярных инструментов и технологий относятся:
9. Игровые движки: Многие игровые движки, такие как Unity и Unreal Engine, оснащены надежными инструментами для создания пользовательского интерфейса. Например, в Unity есть системы Unity UI и UI Toolkit для создания динамических интерфейсов.
10. Программное обеспечение для дизайна: такие программы, как Adobe XD, Sketch и Figma, позволяют дизайнерам создавать прототипы макетов пользовательских интерфейсов, визуальные ресурсы и эффективно сотрудничать с другими членами команды.
11. Фреймворки для программирования: для создания пользовательских функций пользовательского интерфейса может потребоваться программирование с использованием таких фреймворков, как HTML/CSS для веб-игр или C# и C++ для более сложных взаимодействий в игровых движках.

Тестирование удобства использования: итеративное тестирование необходимо для того, чтобы пользовательский интерфейс соответствовал потребностям и ожиданиям игроков. Этот процесс включает в себя:

Тестирование игрового процесса: проведение сеансов тестирования игрового процесса для оценки того, как игроки взаимодействуют с пользовательским интерфейсом, помогает выявить проблемы с удобством использования и области, требующие улучшения. Наблюдение за игроками в режиме реального времени позволяет выявить элементы, которые могут сбивать с толку или быть неинтуитивно понятными.

Сбор отзывов: сбор отзывов с помощью опросов или интервью после игровых сессий позволяет получить ценные сведения о том, как игроки взаимодействуют с пользовательским интерфейсом, что может помочь в его доработке.

Непрерывная итерация: разработка пользовательского интерфейса — это непрерывный процесс. Регулярное обновление и корректировка дизайна на основе отзывов игроков гарантируют, что интерфейс останется эффективным и увлекательным на протяжении всего жизненного цикла игры.

Доступность и инклюзивность: создание инклюзивного пользовательского интерфейса необходимо для того, чтобы все игроки могли наслаждаться игрой. Функции доступности могут включать:

настраиваемые параметры: возможность для игроков изменять элементы управления, размер шрифта и цветовые схемы может улучшить игровой процесс для людей с ограниченными возможностями или особыми предпочтениями.

Поддержка вспомогательных технологий: обеспечение совместимости со считывателями с экрана, настройками цветового контраста для игроков с дальтонизмом и функцией преобразования текста в речь расширяет аудиторию и способствует созданию более инклюзивной игровой среды.

В заключение отметим, что разработка пользовательского интерфейса — это фундаментальный аспект реализации игры, который требует тщательного рассмотрения принципов дизайна, выбора подходящих инструментов, тестирования удобства использования и функций доступности. Сосредоточившись на создании интуитивно понятного и привлекательного пользовательского интерфейса, разработчики могут улучшить взаимодействие с игроками, повысить качество игрового процесса в целом и сделать так, чтобы игра понравилась широкой аудитории. Хорошо продуманный пользовательский интерфейс не только поддерживает игровую механику, но и обогащает сюжет и погружает игроков в игровой мир, что делает его важнейшим элементом успешного игрового дизайна.

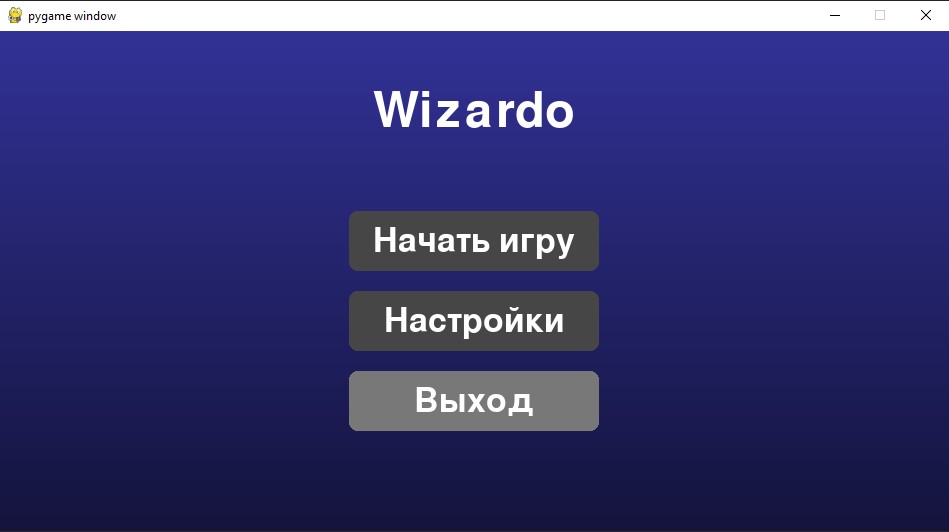
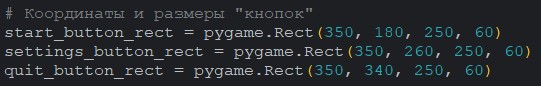
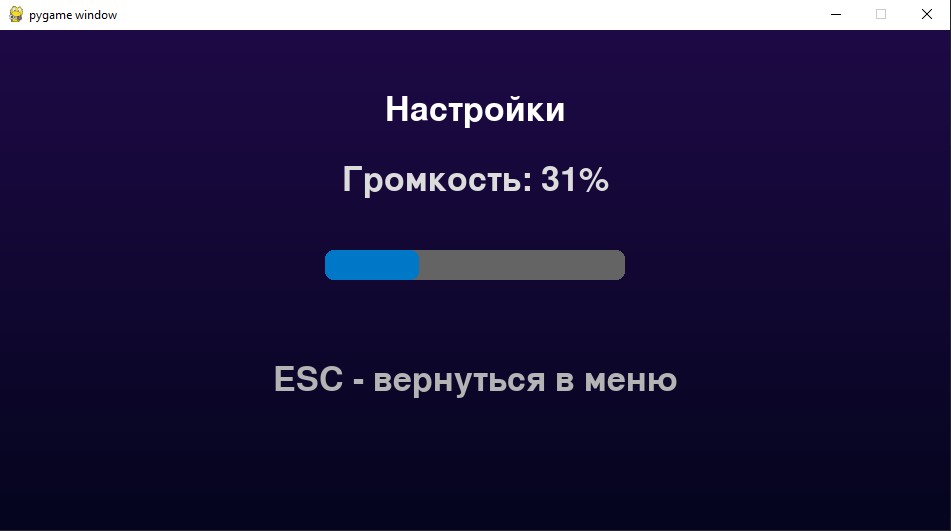
Ниже, на рис.1, представлен начальный экран игры, разработанный с учетом перечисленных принципов проектирования интерфейса и требований к его созданию.

Рис.1. Начальный экран игры.

В коде размер и координаты кнопок определяются переменными:



На рис.2 представлено меню «Настройки», где можно настроить громкость музыки в игре.

Рис.3. Меню «Настройки».

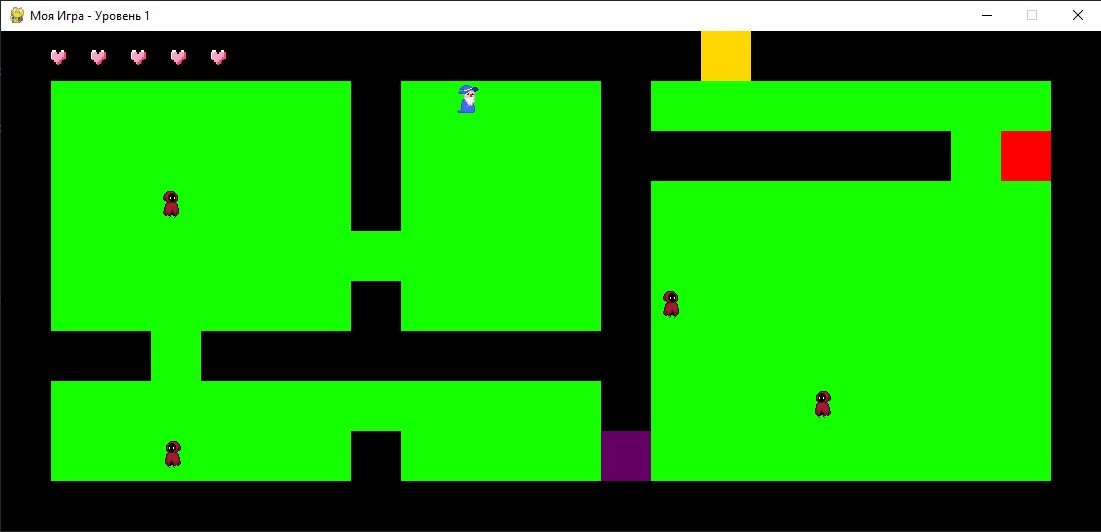
На рис.4 показан экран игры, в который мы переходим при нажатии на кнопку «Начать игру».

Рис.4. Экран игры «Уровень 1».

Более подробный разбор создания уровней, персонаже и условий игры будут рассмотрены в следующем подразделе «Создание игровых уровней и окружения.».

# 4.2. Создание игровых уровней и окружения.

Уровни не только служат фоном для игрового процесса, но и существенно влияют на впечатления игроков, их вовлечённость и общее повествование. В этом разделе описаны основные элементы и средства при разработке уровней игры и персонажей.

Концептуализация дизайна уровней:

- первый этап создания уровней включает в себя концептуализацию общей структуры и хода каждого уровня.

Ключевые факторы включают:

1. тематика и эстетика: Определение четкой тематики помогает создать единый визуальный стиль для каждого уровня. Тематика может варьироваться от фэнтези и научной фантастики до исторических событий и должна отражать элементы дизайна, цвета, материалы и повествование об окружающей среде.
2. Прогресс игрока: Разработка уровней должна способствовать ощущению прогресса, добавляя новые механики, задачи и сюжетные элементы по мере прохождения игры. Уровни должны быть построены таким образом, чтобы постепенно увеличивать сложность, обеспечивая удовлетворительную кривую обучения.
3. Разнообразие игрового процесса: в том числе разнообразные игровые возможности на уровнях, такие как исследование, решение головоломок, сражения и скрытность, привлекают игроков и побуждают их адаптировать свои стратегии.
4. Планировка и последовательность действий: Создание хорошо продуманной планировки имеет решающее значение для навигации и взаимодействия игроков. Ключевые элементы дизайна макета включают в себя:
5. Маршруты и навигацию: создание интуитивно понятных маршрутов, которые помогут игрокам проходить уровни, не вызывая разочарований. Включение ориентиров и визуальных подсказок может помочь игрокам понять, куда идти дальше.
6. Расположение врагов и испытаний: стратегическое расположение врагов, препятствий и испытаний необходимо для балансировки игрового процесса. Расположение должно основываться на темпе игрового процесса и введении новых механик, чтобы столкновения оставались сложными, но справедливыми.

Игра создавалась по принципам ООП – полиморфизм, наследственность и инкапсуляция. Программа состоит из нескольких файлов:

- main.py – основной исполнительный файл игры;

- boss.py – файл, в котором расположен класс «Boss», финального и самого сложного врага;

- settings.py – файл, в котором находятся настройки размера и цветов интерфейса игры, а также размер шрифтов;

- player.py – файл, в котором расположен класс игрока «Player»;

- level.py - файл, в котором расположен класс игрока «Level» и расписаны свойства и характеристики уровней;

- enemy.py – файл, в котором расположен класс врагов «Enemy» и производится загрузка и обработка свойств врагов.

В файле main.py полностью описан процесс игры и взаимодействия между собой приведенных выше классов.

Как указывалось ранее уровни создаются с помощью наследования от класса «Level» и представляет собой список уровней с разметкой и расположением окружения, как показано на рис.5. «Создание списка уровней в файле main.py».

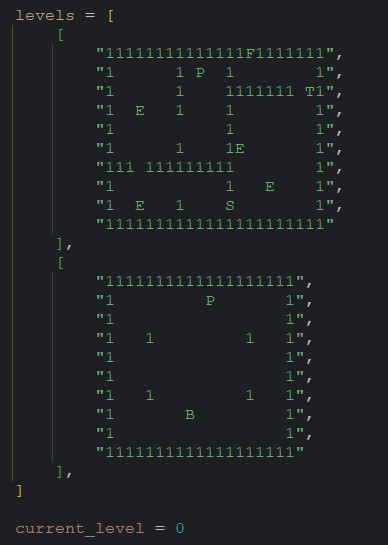


Рис.5. Создание списка уровней в файле main.py.

Здесь:

P – начальное положение игрока в текущем уровне;

E – начальное положение врагов в текущем уровне;

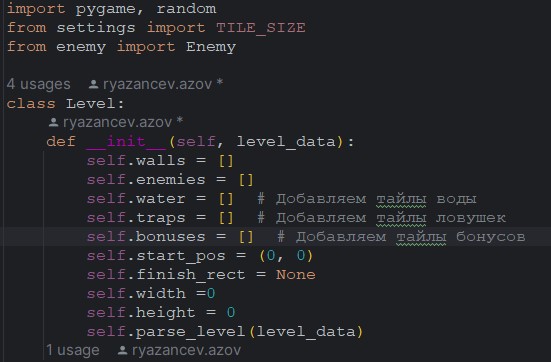
B – начальное положение босса;

T – положение ловушки, попадая в которую, игрок возвращается в начальное положение;

S – бонус, прибавляет игроку дополнительную жизнь – сердце;

1 – границы игрового поля и препятствия в уровне;

При создании класса «Level» использовались (импортировались) стандартные библиотеки pygame и random, а также переменные из файла «settings.py» и целиком класс «Enemy» из файла «enemy.py», как показано ниже на рис.6. «Импорт библиотек, переменных и классов для создания класса Level.».

Рис.6. Импорт библиотек, переменных и классов для создания класса Level.

После импорта необходимых элементов мы создаем непосредственно сам класс «Level», производим инициализацию элементов уровней, таких как: стены, враги, бонусы, начальные позиции и т.д.

Также в классе прописываются функции «random\_bool», «parse\_level» и «draw», с помощью которых будут строиться и рисоваться уровни в игре, а также располагаться и взаимодействовать персонажи.

Полный текст программного кода класса «Level» представлен в приложении №9.5.

# 4.3. Написание кода на Python

Написание кода на Python является основополагающим аспектом реализации игры, поскольку позволяет разработчикам создавать логику, механику и функциональность, которые управляют игрой. В этом разделе описываются лучшие практики, основные концепции и ключевые стратегии для эффективного написания игры на Python, особенно с использованием таких библиотек, как Pygame.

Использование интегрированной среды разработки (IDE): можно выбрать IDE или текстовый редактор, с помощью которого можно написать код программы. Среди популярных вариантов для разработки на Python — PyCharm, Visual Studio Code и Atom, которые предлагают такие функции, как подсветка синтаксиса и инструменты отладки.

Для написания кода данного проекта был использован PyCharm Community Edition 2022.2.1, интерпретатор Python 3.10, а для учета версий программы был создан репозиторий на ресурсе GitHub (ссылка на проект: www.https://github.com/IvanRyazancev/Diplom). В качестве библиотеки использована стандартная библиотека Pygame. Эта библиотека предоставляет функции для работы с графикой, звуками и пользовательским вводом.

Типичная структура проекта включает в себя:

а) Основной цикл. Центральным компонентом любой игры является основной цикл, работающий непрерывно и управляющий игровыми событиями. Этот цикл обновляет состояние игры и отображает графику. Базовая структура представлена ниже, на рис.7. Базовая структура игры:

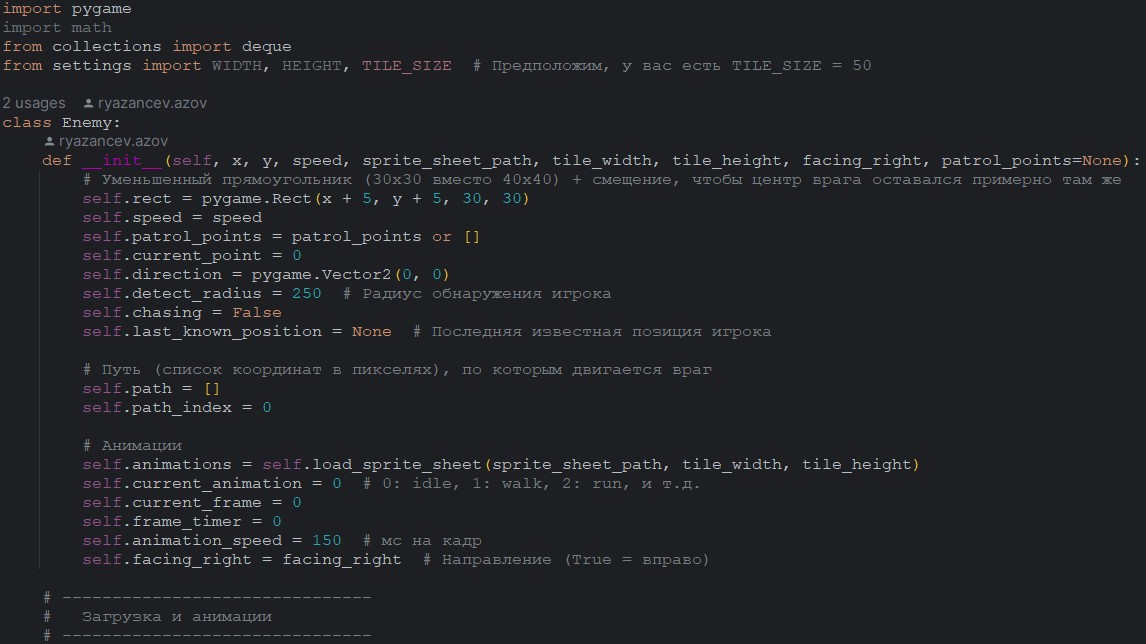
Рис.7. Базовая структура игры.

На данном рисунке представлена сокращенная версия кода, более подробно данный код представлен в приложении №9.1.

б) Состояния игры. Управление различными состояниями игры (например, титульный экран, воспроизведение, игра окончена) может осуществляться с помощью структуры конечного автомата, которая изменяет поведение и рендеринг в зависимости от текущего состояния.

в) Реализация игровой логики. Разработка игровой логики включает в себя определение того, как объекты взаимодействуют и как игра реагирует на действия игрока. Основные соображения включают в себя:

1. Создание классов и объектов. Использование объектно-ориентированного программирования (ООП) упрощает обслуживание и организацию кода. В данной работе были определены классы для игровых сущностей, таких как игроки, враги и предметы, инкапсулируя свойства и поведение, относящиеся к этим сущностям. Например, создание класса «Enemy», код класса приведен в приложении №9.6, часть данного класса представлена ниже, на рис.8. Создание класса «Enemy».

Рис.8. Создание класса «Enemy».

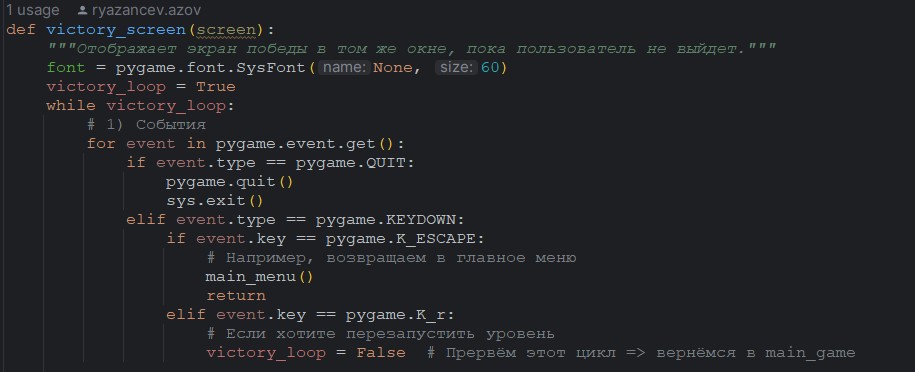
1. Обработка пользовательского ввода: реализация обработки пользовательского ввода с помощью событий позволяет игрокам взаимодействовать с игрой. Pygame предоставляет функции для захвата событий клавиатуры и мыши, что позволяет разработчикам реагировать соответствующим образом. Пример приведен на рис.9. Обработка события, вывод экрана победы, пока пользователь не выйдет.

Рис.9. Обработка события, вывод экрана победы, пока пользователь не выйдет.

1. Отображение графики: отображение графики является основным компонентом разработки игр. Pygame упрощает этот процесс с помощью различных методов.

- Загрузка и отображение изображений. В моем проекте для изображения главного героя, врагов, босса и анимирования этих персонажей были использованы спрайты - небольшая графическая картинка или изображение,

которое можно анимировать. Для обработки спрайтов была написана функция, пример, которой представлен ниже, на рис.10. Функция для обработки спрайтов.

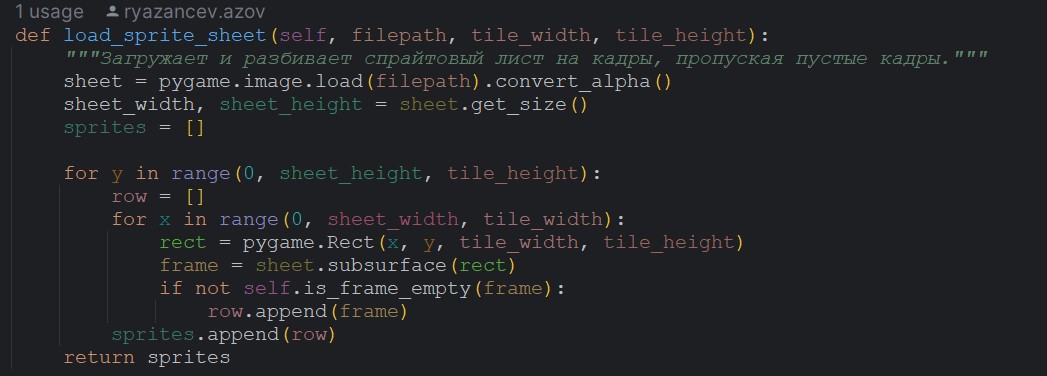


Рис.10. Функция обработки спрайтов.

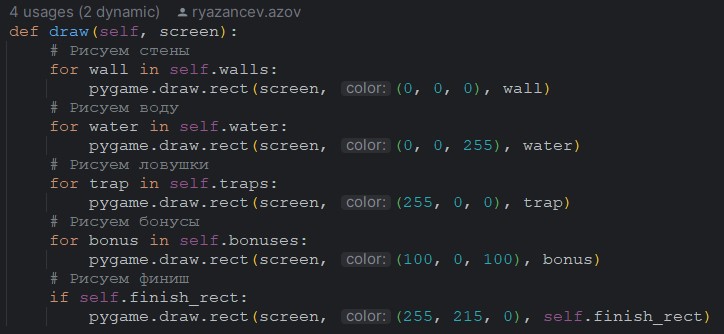
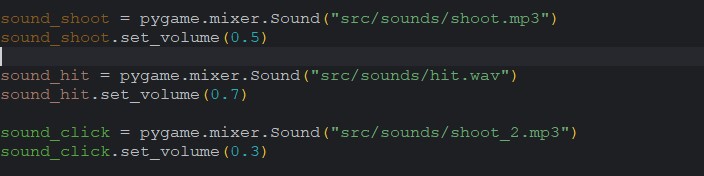
 - Рисование фигур. Pygame предоставляет функции для рисования базовых фигур, таких как прямоугольники, линии и круги, которые можно использовать для игровых элементов, например для фона или наложений пользовательского интерфейса. На рис.11 представлен пример использования данной возможности в моем проекте.

Рис.11. Рисование элементов игры.

Интеграция звука и музыки: добавление аудиоэлементов улучшает игровой процесс. На рис.12 представлена реализация интеграции музыки и звуков.

Рис.12. Интеграция музыки и звуков в игру.

Контроль версий. Для контроля версий использовалась система контроля версий Git, которая позволяет разработчикам отслеживать изменения, эффективно сотрудничать и управлять различными версиями игрового кода. Регулярное внесение изменений и создание ответвлений для новых функций помогает поддерживать организованный рабочий процесс разработки.

# 5. Тестирование и отладка.

# 5.1. Методология тестирования.

Тестирование — важнейший этап разработки 2D-ролевой игры «Wizardo», обеспечивающий функциональность и удобство использования конечного продукта. В данном дипломном проекте была использовано игровое тестирование, позволяющее оценить производительность игры и удобство использования.

- Игровое тестирование: игровое тестирование предполагает, что реальные пользователи играют в игру, чтобы оставить отзыв о механике игрового процесса, уровнях сложности и общем удовольствии от игры. Этот метод качественного тестирования позволяет получить информацию, которую невозможно получить с помощью автоматизированных тестов. Отзывы игроков-тестировщиков бесценны для выявления областей, требующих улучшения, таких как неясные инструкции, несбалансированные враги или раздражающая игровая механика.

# 5.2. Проведение тестов и анализ результатов.

В ходе тестирования игры "Wizardo" с участием 10 пользователей выявлены как сильные стороны, так и недостатки. В целом первый уровень игры показался тестировщикам простым, но достаточным для освоения базовой механики. Среднее время прохождения первого уровня составило 3 минуты 12 секунд, при этом только 3 из 10 тестировщиков погибли хотя бы раз на этом этапе, а общее количество смертей на уровне составило 7. Пользователи отметили интересную механику отскакивающих пуль, которая позволяет убивать врагов из-за углов. Однако для новичков это вызвало трудности: 4 из 10 игроков жаловались, что попадать по врагам из-за рикошета сложно, особенно в узких коридорах.

Враги на первом уровне, хотя и не представляют серьёзной угрозы, долго преследуют игрока, что добавляет напряжения. 6 из 10 тестировщиков указали, что преследование врагами затрудняет исследование лабиринта. Также отмечено, что иногда поведение врагов делает их предсказуемыми, и опытные игроки легко справлялись с ними, используя рикошет.

На втором уровне сражение с боссом оказалось чрезвычайно сложным для большинства участников. Среднее время прохождения уровня составило 6 минут 48 секунд, но только 4 из 10 игроков смогли завершить бой, остальные сдались после многочисленных попыток. Среднее количество смертей на игрока на этом этапе достигло 4,1, а общий показатель смертей за все тестирование составил 41.

На первой фазе босса, где он активно двигается и ловко уворачивается от пуль, пользователи испытывали значительные трудности: 7 из 10 игроков отметили, что попасть в босса практически невозможно из-за его маневренности. В среднем только 1 из 10 выстрелов попадал в цель, что вызывало у игроков ощущение беспомощности.

Вторая фаза, буллет-хелл, оказалась критической для большинства участников: 8 из 10 тестировщиков теряли почти все жизни именно на этом этапе. В среднем игроки выдерживали не более 7 секунд активного обстрела, прежде чем погибнуть. Более половины пользователей (6 из 10) предложили уменьшить плотность атак в этой фазе, поскольку увернуться от такого количества снарядов кажется практически невозможным.

Третья фаза, где босс начинает двигаться сквозь препятствия и хаотично стрелять, была названа одновременно напряжённой и захватывающей. Однако её хаотичность вызвала неоднозначные отзывы: 5 из 10 тестировщиков посчитали эту часть несправедливой из-за невозможности предсказать поведение босса, а 4 из них заявили, что фаза ощущается как «рандом», где успех зависит больше от удачи, чем от навыков.

Несмотря на высокую сложность, 3 из 10 тестировщиков — опытные игроки — оценили игру положительно, отметив, что она представляет собой достойный вызов для любителей хардкорных механик. Однако 7 из 10 пользователей, включая новичков, посчитали, что уровень сложности требует корректировки, особенно в плане точности попадания по боссу и уменьшения плотности обстрела на второй фазе.

Игра получила смешанные отзывы: с одной стороны, игроки отметили интересные механики, особенно отскок пуль и вариативность фаз босса, с другой — критически отнеслись к чрезмерной сложности и трудности адаптации к поведению босса.

# Заключение

В заключение хотелось бы отметить, что при разработке 2D RPG игры «Wizardo» с использованием Python и средств библиотеки Pygame удалось применить знания, полученные при прохождении курсов переподготовки по специализации «Разработчик — Программист». В данном дипломе были применены три основных принципа ООП:

- Инкапсуляция;

- Наследование;

- Полиморфизм.

Также в ходе реализации проекта были изучены средства библиотеки Pygame, в ходе работы над проектом были изучены различные аспекты игрового дизайна, включая создание уровней и развитие персонажей. Использование Python позволило быстро создавать прототипы и эффективно писать код, а Pygame обеспечил необходимую основу для работы с графикой, звуками и пользовательским вводом. Гибкость Python и Pygame позволила реализовать широкий спектр игровых механик и функций, обогатив общий игровой опыт. Кроме того, этот проект стал ценной возможностью для обучения, повышения навыков программирования и углубления понимания принципов разработки игр. Также пришлось решить проблему оптимизации производительности. Таким образом, можно сделать вывод о том, что цели, поставленные в этом дипломе достигнуты.

В перспективе существует множество возможностей для расширения «Wizardo», включая добавление нового контента, улучшение графики и игрового процесса. Этот проект не только закладывает прочный фундамент для дальнейшего развития, но и подчёркивает важность непрерывного обучения и адаптации в постоянно развивающейся сфере игрового дизайна. Таким образом, «Wizardo» — это свидетельство творческого подхода и потенциала начинающих разработчиков игр, демонстрирующее, как доступные инструменты, такие как Python и Pygame, могут воплотить в жизнь творческие концепции.

# Список используемой литературы

* Книга: Эрик Мэтиз. «Изучаем Python» 3-е издание. Санкт-Петербург: «Питер», 2020 г.
* Книга: Н.Прохоренок, В.Дронов. «Python 3 и PyQt5. Разработка приложений» 2-е издание. Санкт-Петербург: «БХВ-Петербург», 2019 г.
* Книга: Анатолий Постолит. «Python, Django и PyCharm для начинающих». Санкт-Петербург: «БХВ-Петербург», 2021 г.

# Приложения

**Приложение №9.1. Основной файл игры main.py.**

import pygame  
import sys  
from settings import *FPS*, *WHITE*, *BLUE*, *GREEN*from player import Player  
from level import Level  
  
# pygame.mixer.pre\_init(44100, -16, 2, 256)  
pygame.init()  
pygame.mixer.init()  
  
*sound\_shoot* = pygame.mixer.Sound("src/sounds/shoot.mp3")  
*sound\_shoot*.set\_volume(0.5)  
  
*sound\_hit* = pygame.mixer.Sound("src/sounds/hit.wav")  
*sound\_hit*.set\_volume(0.7)  
  
*sound\_click* = pygame.mixer.Sound("src/sounds/shoot\_2.mp3")  
*sound\_click*.set\_volume(0.3)  
  
# --------------------------------------------  
# Глобальные переменные (пример)  
# --------------------------------------------  
*WIDTH*, *HEIGHT* = 950, 500  
*screen* = pygame.display.set\_mode((*WIDTH*, *HEIGHT*))  
*clock* = pygame.time.Clock()  
  
  
# --------------------------------------------  
# Вспомогательные функции  
# --------------------------------------------  
  
def draw\_gradient\_background(*surf*, *top\_color*, *bottom\_color*):  
 *"""  
 Рисует простой вертикальный градиент от top\_color к bottom\_color.  
 """  
 rect* = *surf*.get\_rect()  
 # Высота экрана  
 *height* = *rect*.height  
  
 for *y* in range(*height*):  
 # Вычисляем пропорцию между top\_color и bottom\_color  
 *ratio* = *y* / *height  
 r* = int(*top\_color*[0] + (*bottom\_color*[0] - *top\_color*[0]) \* *ratio*)  
 *g* = int(*top\_color*[1] + (*bottom\_color*[1] - *top\_color*[1]) \* *ratio*)  
 *b* = int(*top\_color*[2] + (*bottom\_color*[2] - *top\_color*[2]) \* *ratio*)  
 pygame.draw.line(*surf*, (*r*, *g*, *b*), (0, *y*), (*rect*.width, *y*))  
  
  
def draw\_button(*surf*, *rect*, *text*, *font*, *mouse\_pos*):  
 *"""  
 Рисует кнопку (прямоугольник + текст).  
 Подсвечивает кнопку, если курсор мыши внутри rect.  
 """* (*mx*, *my*) = *mouse\_pos  
 color\_normal* = (70, 70, 70)  
 *color\_hovered* = (120, 120, 120)  
 *text\_color* = (255, 255, 255)  
  
 # Если курсор в пределах кнопки — подсветить  
 if *rect*.collidepoint(*mx*, *my*):  
 pygame.draw.rect(*surf*, *color\_hovered*, *rect*, border\_radius=8)  
 else:  
 pygame.draw.rect(*surf*, *color\_normal*, *rect*, border\_radius=8)  
  
 # Рисуем текст по центру кнопки  
 *text\_surf* = *font*.render(*text*, True, *text\_color*)  
 *text\_rect* = *text\_surf*.get\_rect(center=*rect*.center)  
 *surf*.blit(*text\_surf*, *text\_rect*)  
  
  
# --------------------------------------------  
# Меню настроек (регулировка громкости)  
# --------------------------------------------  
def settings\_menu():  
 *font* = pygame.font.SysFont(None, 50)  
  
 # Берём текущую громкость (0.0 - 1.0)  
 *current\_volume* = pygame.mixer.music.get\_volume()  
  
 # Параметры ползунка  
 *slider\_width* = 300  
 *slider\_height* = 30  
 *slider\_x* = (*WIDTH* - *slider\_width*) // 2  
 *slider\_y* = 220  
  
 # Для отслеживания, «таскает» ли пользователь ползунок  
 *dragging* = False  
  
 *running* = True  
 while *running*:  
 *mouse\_pos* = pygame.mouse.get\_pos()  
 for *event* in pygame.event.get():  
 if *event*.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()  
  
 elif *event*.type == pygame.KEYDOWN:  
 if *event*.key == pygame.K\_ESCAPE:  
 *running* = False # Возвращаемся в главное меню  
  
 elif *event*.type == pygame.MOUSEBUTTONDOWN:  
 if *event*.button == 1:  
 # Проверяем, кликнул ли пользователь по ползунку  
 # Считаем, что ползунком считается прямоугольник [slider\_x..slider\_x+slider\_width]  
 # и по высоте [slider\_y..slider\_y+slider\_height].  
 *slider\_rect* = pygame.Rect(*slider\_x*, *slider\_y*, *slider\_width*, *slider\_height*)  
 if *slider\_rect*.collidepoint(*mouse\_pos*):  
 *dragging* = True  
 # Обновляем громкость сразу  
 *current\_volume* = (*mouse\_pos*[0] - *slider\_x*) / *slider\_width  
 current\_volume* = max(0.0, min(1.0, *current\_volume*))  
 pygame.mixer.music.set\_volume(*current\_volume*)  
  
 elif *event*.type == pygame.MOUSEBUTTONUP:  
 if *event*.button == 1:  
 *dragging* = False  
  
 elif *event*.type == pygame.MOUSEMOTION:  
 # Если пользователь «тащит» ползунок  
 if *dragging*:  
 *current\_volume* = (*mouse\_pos*[0] - *slider\_x*) / *slider\_width  
 current\_volume* = max(0.0, min(1.0, *current\_volume*))  
 pygame.mixer.music.set\_volume(*current\_volume*)  
  
 # Рисуем фон (градиент для красоты)  
 draw\_gradient\_background(*screen*, (30, 10, 70), (5, 5, 30))  
  
 # Заголовок  
 *title* = *font*.render("Настройки", True, (255, 255, 255))  
 *title\_rect* = *title*.get\_rect(center=(*WIDTH* // 2, 80))  
 *screen*.blit(*title*, *title\_rect*)  
  
 # Текст громкости  
 *volume\_text* = *font*.render(f"Громкость: {int(*current\_volume* \* 100)}%", True, (220, 220, 220))  
 *volume\_rect* = *volume\_text*.get\_rect(center=(*WIDTH* // 2, 150))  
 *screen*.blit(*volume\_text*, *volume\_rect*)  
  
 # Отрисовка "полоски" (background) для громкости  
 pygame.draw.rect(*screen*, (100, 100, 100), (*slider\_x*, *slider\_y*, *slider\_width*, *slider\_height*), border\_radius=8)  
  
 # Отрисовка "залитой части" (синий цвет, например)  
 *fill\_width* = int(*current\_volume* \* *slider\_width*)  
 pygame.draw.rect(*screen*, (0, 120, 200), (*slider\_x*, *slider\_y*, *fill\_width*, *slider\_height*), border\_radius=8)  
  
 # Подпись о возврате  
 *tip* = *font*.render("ESC - вернуться в меню", True, (180, 180, 180))  
 *tip\_rect* = *tip*.get\_rect(center=(*WIDTH* // 2, 350))  
 *screen*.blit(*tip*, *tip\_rect*)  
  
 pygame.display.flip()  
 *clock*.tick(*FPS*)  
  
  
# --------------------------------------------  
# Главное меню  
# --------------------------------------------  
def main\_menu():  
 pygame.mixer.music.stop()  
 pygame.mixer.music.load("src/music/back\_music.mp3")  
 pygame.mixer.music.set\_volume(0.1) # Стартовая громкость  
 pygame.mixer.music.play(-1) # -1 => играть в цикле  
  
 *font\_big* = pygame.font.SysFont(None, 72)  
 *font\_small* = pygame.font.SysFont(None, 50)  
  
 # Координаты и размеры "кнопок"  
 *start\_button\_rect* = pygame.Rect(350, 180, 250, 60)  
 *settings\_button\_rect* = pygame.Rect(350, 260, 250, 60)  
 *quit\_button\_rect* = pygame.Rect(350, 340, 250, 60)  
  
 *in\_menu* = True  
 while *in\_menu*:  
 *mouse\_pos* = pygame.mouse.get\_pos()  
  
 for *event* in pygame.event.get():  
 if *event*.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()  
  
 elif *event*.type == pygame.MOUSEBUTTONDOWN:  
  
 *channel* = pygame.mixer.find\_channel(True) # Найти свободный канал (или создать)  
 *channel*.play(*sound\_click*)  
  
 if *event*.button == 1: # Левая кнопка мыши  
 if *start\_button\_rect*.collidepoint(*event*.pos):  
 # Начать игру  
 *in\_menu* = False  
 elif *settings\_button\_rect*.collidepoint(*event*.pos):  
 # Переход в меню настроек  
 settings\_menu()  
 elif *quit\_button\_rect*.collidepoint(*event*.pos):  
 pygame.quit()  
 sys.exit()  
  
 # Рисуем фон (градиент)  
 draw\_gradient\_background(*screen*, (50, 50, 150), (20, 20, 60))  
  
 # Заголовок  
 *title\_surface* = *font\_big*.render("Wizardo", True, (255, 255, 255))  
 *title\_rect* = *title\_surface*.get\_rect(center=(*WIDTH* // 2, 80))  
 *screen*.blit(*title\_surface*, *title\_rect*)  
  
 # Кнопки  
 draw\_button(*screen*, *start\_button\_rect*, "Начать игру", *font\_small*, *mouse\_pos*)  
 draw\_button(*screen*, *settings\_button\_rect*, "Настройки", *font\_small*, *mouse\_pos*)  
 draw\_button(*screen*, *quit\_button\_rect*, "Выход", *font\_small*, *mouse\_pos*)  
  
 pygame.display.flip()  
 *clock*.tick(*FPS*)  
  
  
# Определение уровней  
  
def victory\_screen(*screen*):  
 *"""Отображает экран победы в том же окне, пока пользователь не выйдет."""  
 font* = pygame.font.SysFont(None, 60)  
 *victory\_loop* = True  
 while *victory\_loop*:  
 # 1) События  
 for *event* in pygame.event.get():  
 if *event*.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()  
 elif *event*.type == pygame.KEYDOWN:  
 if *event*.key == pygame.K\_ESCAPE:  
 # Например, возвращаем в главное меню  
 main\_menu()  
 return  
 elif *event*.key == pygame.K\_r:  
 # Если хотите перезапустить уровень  
 *victory\_loop* = False # Прервём этот цикл => вернёмся в main\_game  
  
 # 2) Рисуем фон, текст  
 *screen*.fill((30, 30, 30))  
 # Надпись "ПОБЕДА!"  
 *victory\_text* = *font*.render("ПОБЕДА!", True, (255, 215, 0))  
 *victory\_rect* = *victory\_text*.get\_rect(center=(*WIDTH* // 2, *HEIGHT* // 2 - 50))  
 *screen*.blit(*victory\_text*, *victory\_rect*)  
  
 # Пример счёта (заглушка). Может быть, count\_killed\_enemies, etc.  
 *score\_text* = *font*.render("Очки: 999", True, (255, 255, 255))  
 *score\_rect* = *score\_text*.get\_rect(center=(*WIDTH* // 2, *HEIGHT* // 2 + 10))  
 *screen*.blit(*score\_text*, *score\_rect*)  
  
 *tip\_text* = *font*.render("ESC - меню, R - перезапуск", True, (200, 200, 200))  
 *tip\_rect* = *tip\_text*.get\_rect(center=(*WIDTH* // 2, *HEIGHT* // 2 + 80))  
 *screen*.blit(*tip\_text*, *tip\_rect*)  
  
 # 3) Обновляем экран  
 pygame.display.flip()  
 *clock*.tick(*FPS*)  
  
  
def main\_game():  
 pygame.mixer.music.stop()  
 pygame.mixer.music.load("src/music/for\_battle.mp3")  
 pygame.mixer.music.set\_volume(0.1) # Стартовая громкость  
 pygame.mixer.music.play(-1) # -1 => играть в цикле  
  
 *levels* = [  
 [  
 "11111111111111F1111111",  
 "1 1 P 1 1",  
 "1 1 1111111 T1",  
 "1 E 1 1 1",  
 "1 1 1",  
 "1 1 1E 1",  
 "111 111111111 1",  
 "1 1 E 1",  
 "1 E 1 S 1",  
 "1111111111111111111111"  
 ],  
 [  
 "1111111111111111111",  
 "1 P 1",  
 "1 1",  
 "1 1 1 1",  
 "1 1",  
 "1 1",  
 "1 1 1 1",  
 "1 B 1",  
 "1 1",  
 "1111111111111111111"  
 ],  
 ]  
  
 *current\_level* = 0  
  
 screen = pygame.display.set\_mode((950, 500))  
 *level* = Level(*levels*[*current\_level*])  
  
 *screen* = pygame.display.set\_mode((*level*.width, *level*.height))  
 # Загружаем изображение сердечка  
 *heart\_image* = pygame.image.load("src/sprites/sheart.png").convert\_alpha()  
 *heart\_image* = pygame.transform.scale(*heart\_image*, (32, 32)) # Масштабируем сердечко  
  
 def draw\_lives(*screen*, *lives*):  
 *"""Рисует сердечки в правом верхнем углу."""* for *i* in range(*lives*):  
 *screen*.blit(*heart\_image*, (200 - *i* \* 40, 10)) # Отступ между сердечками  
  
 pygame.display.set\_caption(f"Моя Игра - Уровень {*current\_level* + 1}")  
  
 # Создаем игрока  
 *player* = Player(*level*.start\_pos[0], *level*.start\_pos[1], "src/sprites/wizard\_tiles.png", 32,  
 32, *sound\_shoot*)  
 *enemies* = *level*.enemies  
  
 # Шрифт для «Game Over»  
 *font\_game\_over* = pygame.font.SysFont(None, 60)  
  
 *running* = True  
 while *running*:  
 *delta\_time* = *clock*.tick(*FPS*)  
  
 # 1) Считываем события  
 for *event* in pygame.event.get():  
 if *event*.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()  
  
 if *event*.type == pygame.KEYDOWN:  
 if *event*.key == pygame.K\_SPACE:  
 *player*.shoot()  
  
 # 2) Обновляем логику, если игрок ещё не умер (либо анимация смерти уже идёт, но не завершилась)  
 # Важно продолжать вызывать update, чтобы кадры смерти переключались!  
  
 *current\_time* = pygame.time.get\_ticks()  
  
 *player*.update(*delta\_time*, *level*.walls)  
  
 for *trap* in *level*.traps:  
 if (*player*.rect.colliderect(*trap*)):  
 *player*.rect.x, *player*.rect.y = *level*.start\_pos  
 for *bonus* in *level*.bonuses:  
 if (*player*.rect.colliderect(*bonus*)):  
 *player*.lives += 1  
 *level*.bonuses.remove(*bonus*)  
  
 for *enemy* in *enemies*:  
 *enemy*.update(*player*, *level*.walls, *delta\_time*, *current\_time*)  
  
 for *enemy* in *enemies*:  
 if hasattr(*enemy*, 'is\_dead'):  
 if (*enemy*.is\_dead):  
 victory\_screen(*screen*)  
 # 3) Проверяем столкновения с врагами (уменьшаем жизни и вызываем die(), если <= 0)  
  
 for *enemy* in *enemies*:  
  
 if *player*.rect.colliderect(*enemy*.rect):  
 if not *player*.invincible: # Если неуязвимость не активна  
 *player*.lives -= 1  
 *sound\_hit*.play()  
  
 # Включаем i-frames  
 *player*.invincible = True  
 *player*.invincible\_timer = *player*.invincible\_time  
  
 if *player*.lives <= 0:  
 *player*.die()  
 # Противника не удаляем!  
 break  
  
 # 4) Проверяем финиш  
 if *level*.finish\_rect and *player*.rect.colliderect(*level*.finish\_rect):  
 *current\_level* += 1  
 if *current\_level* >= len(*levels*):  
 print("Игра пройдена!")  
 pygame.quit()  
 sys.exit()  
 else:  
 *level* = Level(*levels*[*current\_level*])  
 *screen* = pygame.display.set\_mode((*level*.width, *level*.height))  
 pygame.display.set\_caption(f"Моя Игра - Уровень {*current\_level* + 1}")  
 *player* = Player(*level*.start\_pos[0], *level*.start\_pos[1], "src/sprites/wizard\_tiles.png",  
 32, 32, *sound\_shoot*)  
 *enemies* = *level*.enemies  
  
 # 5) Обновляем снаряды  
 *player*.update\_bullets(*level*.walls, *enemies*)  
  
 # 6) Отрисовка текущего кадра  
 *screen*.fill(*GREEN*)  
 *level*.draw(*screen*)  
  
 # Отрисовываем игрока и врагов  
 *player*.draw(*screen*)  
 *player*.draw\_bullets(*screen*)  
 draw\_lives(*screen*, *player*.lives)  
 for *enemy* in *enemies*:  
 *enemy*.draw(*screen*)  
  
 pygame.display.flip()  
  
 # 7) Теперь проверяем: если жизнь <= 0 и анимация смерти уже достигла последнего кадра — показываем «Game Over»  
  
 if *player*.lives <= 0 and *player*.is\_dead:  
 # Проверим, достиг ли игрок конца анимации смерти:  
 *current\_anim* = *player*.animations[*player*.current\_animation]  
 # Если мы на последнем кадре (или превысили его)  
 if *player*.current\_frame >= len(*current\_anim*) - 1:  
 # Запускаем под-цикл «Game Over»  
 *game\_over\_loop* = True  
 while *game\_over\_loop*:  
 # Рисуем тот же кадр (окружение, фон) — чтобы всё оставалось на экране  
 *screen*.fill(*GREEN*)  
 *level*.draw(*screen*)  
  
 for *e* in *enemies*:  
 *e*.draw(*screen*)  
  
 # Рисуем игрока на последнем кадре смерти  
 *player*.draw(*screen*)  
  
 # Выводим текст  
 *go\_text* = *font\_game\_over*.render("GAME OVER", True, (255, 0, 0))  
 *go\_rect* = *go\_text*.get\_rect(center=(*screen*.get\_width() // 2, 100))  
 *screen*.blit(*go\_text*, *go\_rect*)  
  
 *info\_text* = *font\_game\_over*.render("ESC - меню, R - перезапуск", True, (255, 255, 255))  
 *info\_rect* = *info\_text*.get\_rect(center=(*screen*.get\_width() // 2, 200))  
 *screen*.blit(*info\_text*, *info\_rect*)  
  
 pygame.display.flip()  
 *clock*.tick(*FPS*)  
  
 # События внутри экрана «Game Over»  
 for *ev* in pygame.event.get():  
 if *ev*.type == pygame.QUIT:  
 pygame.quit()  
 sys.exit()  
 elif *ev*.type == pygame.KEYDOWN:  
 if *ev*.key == pygame.K\_ESCAPE:  
 # Выходим в главное меню  
 main\_menu()  
 main\_game()  
 return # Полностью выходим из main\_game, т.к. меню перезапустит/закроет игру  
  
 elif *ev*.key == pygame.K\_r:  
 # Перезапуск текущего уровня  
 *level* = Level(*levels*[*current\_level*])  
 *player* = Player(*level*.start\_pos[0], *level*.start\_pos[1],  
 "src/sprites/wizard\_tiles.png", 32, 32, *sound\_shoot*)  
 *enemies* = *level*.enemies  
 *player*.lives = 3 # Восстанавливаем жизни  
  
 *game\_over\_loop* = False # выходим из «Game Over»  
 # Когда вышли из под-цикла (нажали R) — игрок/уровень пересозданы, игра продолжается  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main\_menu()  
 main\_game()

**Приложение №9.2. Файл класса «Boss».**

import pygame  
import math  
import random  
  
class Boss:  
 STATE\_NORMAL = 0  
 STATE\_FINAL\_BARRAGE = 1  
 STATE\_FINAL\_CHAOS = 2  
  
 def \_\_init\_\_(self, *x*, *y*, *tile\_size*, *arena\_width*=950, *arena\_height*=500):  
 *width* = *tile\_size* \* 2  
 *height* = *tile\_size* \* 2  
 self.rect = pygame.Rect(*x*, *y*, *width*, *height*)  
  
 self.arena\_width = *arena\_width* self.arena\_height = *arena\_height* # Здоровье и «обычные» параметры  
 self.max\_health = 14  
 self.health = self.max\_health  
 self.is\_dead = False  
  
 self.bullets = []  
 self.shoot\_cooldown = 1000  
 self.last\_shot\_time = 0  
 self.detect\_radius = 300  
 self.shots\_left = 20 # Кол-во выстрелов в NORMAL  
  
 # Параметры движения  
 self.speed = 2.0  
 *angle* = random.random() \* 2 \* math.pi  
 self.velocity = pygame.Vector2(math.cos(*angle*), math.sin(*angle*)) \* self.speed  
  
 self.change\_dir\_cooldown = 3000  
 self.last\_change\_dir\_time = 0  
  
 # Уклонение  
 self.dodge\_radius = 100  
 self.dodge\_strength = 1.0  
  
 # Три стадии  
 self.state = self.STATE\_NORMAL  
  
 # ------- FINAL\_BARRAGE -------  
 self.barrage\_shoot\_interval = 50 # мс между пулями в финальной/хаотичной стадии  
 self.last\_barrage\_shot\_time = 0  
 self.center\_target = (*arena\_width* // 2, *arena\_height* // 2)  
  
 # Сколько длится стадия FINAL\_BARRAGE (10 секунд)  
 self.final\_barrage\_duration = 5000 # мс  
 self.final\_barrage\_start\_time = 0 # когда мы перешли в финальную стадию  
  
 # ------- FINAL\_CHAOS -------  
 # Тут босс игнорирует стены и двигается хаотично, продолжая стрелять  
 # Можно завести флаги/таймеры, но пока оставим бесконечно.  
  
 def update(self, *player*, *walls*, *delta\_time*, *current\_time*):  
 *"""Главная логика босса, вызывается каждый кадр."""* if self.is\_dead:  
  
 return  
  
 # Если здоровье < 50% или патроны кончились -> FINAL\_BARRAGE (если ещё не началось)  
 if (self.shots\_left <= 0 or self.health <= self.max\_health // 2) and self.state == self.STATE\_NORMAL:  
 self.state = self.STATE\_FINAL\_BARRAGE  
 self.final\_barrage\_start\_time = *current\_time* # Зафиксировали время начала  
  
 # Если мы уже в FINAL\_BARRAGE более 10 секунд -> переходим в FINAL\_CHAOS  
 if self.state == self.STATE\_FINAL\_BARRAGE:  
 if (*current\_time* - self.final\_barrage\_start\_time) > self.final\_barrage\_duration:  
 self.state = self.STATE\_FINAL\_CHAOS  
  
 # Вызываем соответствующий метод обновления  
 if self.state == self.STATE\_NORMAL:  
 self.update\_normal(*player*, *walls*, *delta\_time*, *current\_time*)  
 elif self.state == self.STATE\_FINAL\_BARRAGE:  
 self.update\_final\_barrage(*player*, *delta\_time*, *current\_time*)  
 elif self.state == self.STATE\_FINAL\_CHAOS:  
 self.update\_final\_chaos(*player*, *delta\_time*, *current\_time*)  
  
 # Обновляем пули  
 self.update\_bullets(*walls*, *player*)  
  
 # -----------------------------  
 # 1) NORMAL  
 # -----------------------------  
 def update\_normal(self, *player*, *walls*, *delta\_time*, *current\_time*):  
 # Двигаемся и уклоняемся  
 self.move\_around(*walls*, *delta\_time*, *current\_time*, *player*)  
 # Стреляем в игрока, если видим  
 *distance* = pygame.Vector2(self.rect.center).distance\_to(*player*.rect.center)  
 if *distance* < self.detect\_radius and self.shots\_left > 0:  
 if *current\_time* - self.last\_shot\_time >= self.shoot\_cooldown:  
 self.shoot\_toward\_player(*player*.rect.center)  
 self.last\_shot\_time = *current\_time* self.shots\_left -= 1  
  
  
# -----------------------------  
 # 2) FINAL\_BARRAGE (10 секунд)  
 # -----------------------------  
 def update\_final\_barrage(self, player, *delta\_time*, *current\_time*):  
 *"""  
 Босс идёт в центр, стоит на месте и спамит пули в случайных направлениях.  
 Игнорируем уклонение, не трогаем стены.  
 """  
 center\_pos* = pygame.Vector2(self.center\_target)  
 *to\_center* = *center\_pos* - pygame.Vector2(self.rect.center)  
 *dist* = *to\_center*.length()  
  
 # Идём в центр, если ещё не там  
 if *dist* > 5:  
 *move\_speed* = 4.5  
 *direction* = *to\_center*.normalize()  
 *movement* = *direction* \* *move\_speed* \* (*delta\_time* / 16.67)  
 self.rect.x += *movement*.x  
 self.rect.y += *movement*.y  
 else:  
 # На центре. Спамим пули  
 if *current\_time* - self.last\_barrage\_shot\_time >= self.barrage\_shoot\_interval:  
 self.shoot\_random\_bullet()  
 self.last\_barrage\_shot\_time = *current\_time* # -----------------------------  
 # 3) FINAL\_CHAOS (хаотичное движение, игнор стен)  
 # -----------------------------  
 def update\_final\_chaos(self, player, *delta\_time*, *current\_time*):  
 *"""  
 Босс двигается хаотично, НЕ уклоняется, игнорирует стены  
 (можно просто не вызывать check\_collision\_x / check\_collision\_y),  
 продолжает выпускать случайные пули (поток).  
 """* # 1) Хаотично меняем направление время от времени  
 if *current\_time* - self.last\_change\_dir\_time > self.change\_dir\_cooldown:  
 self.change\_direction()  
 self.last\_change\_dir\_time = *current\_time* # 2) Двигаем rect (никаких коллизий со стенами)  
 *movement* = self.velocity \* (*delta\_time* / 16.67)  
 self.rect.x += *movement*.x  
 self.rect.y += *movement*.y  
  
 # Если вышли за арену — либо телепортируемся, либо отражаем  
 # или просто оставим, пусть улетает, на ваш выбор.  
 # Допустим, чтобы не вылетел далеко, сделаем «отражение» по краям экрана:  
 if self.rect.left < 0 or self.rect.right > self.arena\_width:  
 self.velocity.x \*= -1  
 if self.rect.top < 0 or self.rect.bottom > self.arena\_height:  
 self.velocity.y \*= -1  
  
 # 3) Спам пуль  
 if *current\_time* - self.last\_barrage\_shot\_time >= self.barrage\_shoot\_interval:  
 self.shoot\_random\_bullet()  
 self.last\_barrage\_shot\_time = *current\_time* # -----------------------------  
 # Общие методы стрельбы  
 # -----------------------------  
 def shoot\_toward\_player(self, *target\_pos*):  
 *direction* = pygame.Vector2(*target\_pos*) - pygame.Vector2(self.rect.center)  
 if *direction*.length\_squared() > 0:  
 *direction* = *direction*.normalize()  
 *bullet\_speed* = 7  
  
 *bullet\_rect* = pygame.Rect(self.rect.centerx, self.rect.centery, 10, 10)  
 *bullet* = {  
 'rect': *bullet\_rect*,  
 'direction': *direction*,  
 'speed': *bullet\_speed* }  
 self.bullets.append(*bullet*)  
  
 def shoot\_random\_bullet(self):  
 *"""Стреляет пулей в случайную сторону из центра босса."""  
 angle* = random.uniform(0, 2 \* math.pi)  
 *direction* = pygame.Vector2(math.cos(*angle*), math.sin(*angle*))  
 *bullet\_speed* = 6  
  
 *bullet\_rect* = pygame.Rect(self.rect.centerx, self.rect.centery, 10, 10)  
 *bullet* = {  
 'rect': *bullet\_rect*,  
 'direction': *direction*,  
 'speed': *bullet\_speed* }  
 self.bullets.append(*bullet*)  
  
 # -----------------------------  
 # Обновление пуль  
 # -----------------------------  
 def update\_bullets(self, *walls*, *player*):  
 *bullets\_to\_remove* = []  
 for *bullet* in self.bullets:  
 # Движение пули  
 *bullet*['rect'].x += *bullet*['direction'].x \* *bullet*['speed']  
 *bullet*['rect'].y += *bullet*['direction'].y \* *bullet*['speed']  
  
  
# Столкновение со стенами (кроме финального хаоса —  
 # но тут пули всё равно можно уничтожать о стены)  
 for *wall* in *walls*:  
 if *bullet*['rect'].colliderect(*wall*):  
 *bullets\_to\_remove*.append(*bullet*)  
 break  
  
 # Попадание в игрока  
 if *bullet*['rect'].colliderect(*player*.rect):  
 if not *player*.invincible:  
 *player*.lives -= 1  
 *player*.invincible = True  
 *player*.invincible\_timer = *player*.invincible\_time  
 if *player*.lives <= 0:  
 *player*.die()  
 *bullets\_to\_remove*.append(*bullet*)  
 continue  
  
 # Если вышли за границы  
 if (*bullet*['rect'].x < -50 or *bullet*['rect'].x > 2000 or  
 *bullet*['rect'].y < -50 or *bullet*['rect'].y > 2000):  
 *bullets\_to\_remove*.append(*bullet*)  
  
 for *b* in *bullets\_to\_remove*:  
 if *b* in self.bullets:  
 self.bullets.remove(*b*)  
  
 # -----------------------------  
 # ДВИЖЕНИЕ + УВОРОТ (NORMAL)  
 # -----------------------------  
 def move\_around(self, *walls*, *delta\_time*, *current\_time*, *player*):  
 # Меняем направление раз в 3 секунды  
 if *current\_time* - self.last\_change\_dir\_time > self.change\_dir\_cooldown:  
 self.change\_direction()  
 self.last\_change\_dir\_time = *current\_time* # Уклоняемся от пуль  
 self.dodge\_bullets(*player*)  
  
 # Перемещение с учётом коллизии  
 *movement* = self.velocity \* (*delta\_time* / 16.67)  
 *old\_rect* = self.rect.copy()  
  
 self.rect.x += *movement*.x  
 self.check\_collision\_x(*walls*, *old\_rect*)  
 self.rect.y += *movement*.y  
 self.check\_collision\_y(*walls*, *old\_rect*)  
  
 def change\_direction(self):  
 *angle* = random.random() \* 2 \* math.pi  
 self.velocity = pygame.Vector2(math.cos(*angle*), math.sin(*angle*)) \* self.speed  
  
 def dodge\_bullets(self, *player*):  
 if not hasattr(*player*, 'bullets'):  
 return  
 for *b* in *player*.bullets:  
 *dist* = pygame.Vector2(self.rect.center).distance\_to(*b*['rect'].center)  
 if *dist* < self.dodge\_radius:  
 *away* = pygame.Vector2(self.rect.center) - pygame.Vector2(*b*['rect'].center)  
 if *away*.length\_squared() > 0:  
 *away* = *away*.normalize() \* self.dodge\_strength  
 self.velocity += *away* \* 1.5  
 if self.velocity.length() > self.speed \* 2.5:  
 self.velocity = self.velocity.normalize() \* (self.speed \* 2.5)  
  
 def check\_collision\_x(self, *walls*, *old\_rect*):  
 for *wall* in *walls*:  
 if self.rect.colliderect(*wall*):  
 self.rect.x = *old\_rect*.x  
 self.velocity.x \*= -1  
 break  
  
 def check\_collision\_y(self, *walls*, *old\_rect*):  
 for *wall* in *walls*:  
 if self.rect.colliderect(*wall*):  
 self.rect.y = *old\_rect*.y  
 self.velocity.y \*= -1  
 break  
  
 # -----------------------------  
 # ЗДОРОВЬЕ  
 # -----------------------------  
 def take\_damage(self, *amount*):  
 self.health -= *amount* if self.health <= 0:  
  
 self.is\_dead = True  
  
 # -----------------------------  
 # РИСОВАНИЕ  
 # -----------------------------  
 def draw(self, *screen*):  
 if not self.is\_dead:  
 pygame.draw.rect(*screen*, (200, 0, 0), self.rect)  
 self.draw\_health\_bar(*screen*)  
 for *bullet* in self.bullets:  
 pygame.draw.rect(*screen*, (0, 0, 0), *bullet*['rect'])  
  
 def draw\_health\_bar(self, *screen*):  
 *bar\_width* = self.rect.width  
 *bar\_height* = 8  
 *bar\_x* = self.rect.x  
 *bar\_y* = self.rect.y - *bar\_height* - 5  
 pygame.draw.rect(*screen*, (0, 0, 0), (*bar\_x*, *bar\_y*, *bar\_width*, *bar\_height*))  
 *health\_ratio* = self.health / self.max\_health  
 *current\_width* = int(*bar\_width* \* *health\_ratio*)  
 pygame.draw.rect(*screen*, (0, 200, 0), (*bar\_x*, *bar\_y*, *current\_width*, *bar\_height*))

**Приложение №9.3. Файл настроек «settings.py»**

*WIDTH* = 900  
*HEIGHT* = 600  
*FPS* = 60  
*WHITE* = (255, 255, 255)  
*BLUE* = (0, 0, 255)  
*GREEN* = (20, 255, 0)  
  
*TILE\_SIZE* = 50 # Размер тайла (используется для сетки уровня)

**Приложение №9.4. Файл класса «Player»**

import pygame  
  
from boss import Boss  
from settings import *WIDTH*, *HEIGHT*class Player:  
 def \_\_init\_\_(self, *x*, *y*, *sprite\_sheet\_path*, *tile\_width*, *tile\_height*, *sound\_shoot*):  
 self.invincible = False # Флаг неуязвимости  
 self.invincible\_time = 2000 # (мс) сколько длится неуязвимость  
 self.invincible\_timer = 0 # Счётчик оставшейся неуязвимости  
 self.rect = pygame.Rect(*x*, *y*, 25, 31)  
 self.speed = 4  
 self.sound\_shoot = *sound\_shoot* self.bullets = []  
 self.direction = pygame.Vector2(0, -1) # Начальное направление (вверх)  
 self.facing\_right = True # Направление взгляда (True = вправо)  
 self.is\_shooting = False # Флаг анимации стрельбы  
 self.is\_dead = False # Флаг анимации смерти  
 self.lives = 5  
  
 # Индексы анимаций:  
 # 0 - idle, 1 - walk, 6 - death, 7 - shoot (по вашему коду)  
 self.animations = self.load\_sprite\_sheet(*sprite\_sheet\_path*, *tile\_width*, *tile\_height*)  
 self.current\_animation = 0  
 self.current\_frame = 0  
 self.frame\_timer = 0  
 self.animation\_speed = 150 # Мс на кадр  
  
 def load\_sprite\_sheet(self, *filepath*, *tile\_width*, *tile\_height*):  
 *"""Загружает и разбивает спрайтовый лист на кадры, пропуская пустые кадры."""  
 sheet* = pygame.image.load(*filepath*).convert\_alpha()  
 *sheet\_width*, *sheet\_height* = *sheet*.get\_size()  
 *sprites* = []  
  
 for *y* in range(0, *sheet\_height*, *tile\_height*):  
 *row* = []  
 for *x* in range(0, *sheet\_width*, *tile\_width*):  
 *rect* = pygame.Rect(*x*, *y*, *tile\_width*, *tile\_height*)  
 *frame* = *sheet*.subsurface(*rect*)  
 if not self.is\_frame\_empty(*frame*):  
 *row*.append(*frame*)  
 *sprites*.append(*row*)  
 return *sprites* def is\_frame\_empty(self, *frame*):  
 *"""Проверяет, является ли кадр пустым (все пиксели прозрачные)."""* for *x* in range(*frame*.get\_width()):  
 for *y* in range(*frame*.get\_height()):  
 # Альфа-канал > 0 означает, что пиксель не полностью прозрачный  
 if *frame*.get\_at((*x*, *y*))[3] > 0:  
 return False  
 return True  
  
 def move(self, *walls*):  
 *"""Двигается, если жив. Стрельба (is\_shooting) — не блокирует движение."""* if self.is\_dead:  
 return # Не двигаемся, если мертвы  
  
 *keys* = pygame.key.get\_pressed()  
 *movement* = pygame.Vector2(0, 0)  
  
 if *keys*[pygame.K\_w] or *keys*[pygame.K\_UP]:  
 *movement*.y -= 1  
 if *keys*[pygame.K\_s] or *keys*[pygame.K\_DOWN]:  
 *movement*.y += 1  
 if *keys*[pygame.K\_a] or *keys*[pygame.K\_LEFT]:  
 *movement*.x -= 1  
 self.facing\_right = False  
 if *keys*[pygame.K\_d] or *keys*[pygame.K\_RIGHT]:  
 *movement*.x += 1  
 self.facing\_right = True  
  
 # Если есть движение — переключаемся на walk, иначе idle  
 if *movement*.length\_squared() > 0:  
 *movement* = *movement*.normalize() \* self.speed  
 self.direction = *movement*.normalize()  
 # Если не мертвы и не стреляем, то ставим анимацию walk  
 if not self.is\_dead and not self.is\_shooting:  
 self.set\_animation(1) # 1: walk  
 else:  
 if not self.is\_dead and not self.is\_shooting:  
 self.set\_animation(0) # 0: idle  
  
 # Коллизии (x, y)  
 *old\_rect* = self.rect.inflate(-1, -1)  
 self.rect.x += *movement*.x  
 self.collide(*walls*, 'x', *old\_rect*)  
 self.rect.y += *movement*.y  
 self.collide(*walls*, 'y', *old\_rect*)  
  
 def set\_animation(self, *animation\_index*):  
 *"""Меняем анимацию. Сбрасываем кадр в 0, если это новая анимация."""* # Если уже умерли и кадр дошёл до конца, не переключаемся  
 if self.is\_dead and self.current\_animation == 6:  
 # Анимация смерти (индекс 6) — не перезаписываем  
 return  
  
 if self.current\_animation != *animation\_index*:  
 self.current\_animation = *animation\_index* self.current\_frame = 0  
 self.frame\_timer = 0  
  
 def update\_animation(self, *delta\_time*):  
 *"""Обновляем текущий кадр анимации."""* self.frame\_timer += *delta\_time* if self.frame\_timer >= self.animation\_speed:  
 self.frame\_timer = 0  
 self.current\_frame += 1  
  
 # Если дошли до конца текущей анимации  
 if self.current\_frame >= len(self.animations[self.current\_animation]):  
 # Если анимация смерти — останавливаемся на последнем кадре  
 if self.is\_dead and self.current\_animation == 6:  
 self.current\_frame = len(self.animations[6]) - 1  
 # Если анимация стрельбы  
 elif self.is\_shooting and self.current\_animation == 7:  
 # Закончили стрельбу, возвращаемся в idle  
 self.is\_shooting = False  
 self.set\_animation(0) # idle  
 else:  
 # Для walk / idle — зацикливаем  
 self.current\_frame = 0  
  
 def collide(self, *walls*, *direction*, *old\_rect*):  
 *"""Не даём игроку пройти сквозь стены."""* for *wall* in *walls*:  
 if self.rect.colliderect(*wall*):  
 if *direction* == 'x':  
 self.rect.x = *old\_rect*.x  
 elif *direction* == 'y':  
 self.rect.y = *old\_rect*.y  
  
 def shoot(self):  
 *"""Стреляем (если не мертвы и не в процессе стрельбы). Пуля создаётся сразу."""* if self.is\_dead:  
 return  
 self.sound\_shoot.play()  
 self.is\_shooting = True  
 # 1) Создаём пулю сразу же  
 self.create\_bullet()  
 # 2) Ставим анимацию стрельбы  
 self.set\_animation(7) # 7: shoot  
  
 def die(self):  
 *"""Запускаем анимацию смерти (индекс 6)."""* if self.is\_dead:  
 return # Уже мёртв  
 self.is\_dead = True  
 self.set\_animation(6) # 6: death  
  
 def create\_bullet(self):  
 *"""Создаёт снаряд (пуля) в направлении self.direction."""  
 offset* = self.direction \* (self.rect.width // 2 + 5)  
 *bullet\_pos* = pygame.Vector2(self.rect.center) + *offset  
  
 bullet* = {  
 'bounced': False,  
 'rect': pygame.Rect(*bullet\_pos*.x - 5, *bullet\_pos*.y - 5, 10, 10),  
 'direction': self.direction.copy()  
 }  
 self.bullets.append(*bullet*)  
  
 def update\_bullets(self, *walls*, *enemies*):  
 *"""Обновляем полёт пуль, проверяем столкновения со стенами/врагами/игроком."""* if self.is\_dead:  
 return # Если игрок мертв, не двигаем его пули  
  
 *bullets\_to\_remove* = []  
 for *bullet* in self.bullets:  
 # Движение пули  
 *bullet*['rect'].x += *bullet*['direction'].x \* 10  
 *bullet*['rect'].y += *bullet*['direction'].y \* 10  
  
 # Столкновение с врагами  
 for *enemy* in *enemies*:  
 if *bullet*['rect'].colliderect(*enemy*.rect):  
 *bullets\_to\_remove*.append(*bullet*) # пуля исчезает  
  
 if isinstance(*enemy*, Boss):  
 # Если это босс, уменьшаем здоровье  
 *enemy*.take\_damage(1)  
 # Если босс погиб после урона, только тогда удаляем  
  
 else:  
 # Обычный враг умирает мгновенно  
 if *enemy* in *enemies*:  
 *enemies*.remove(*enemy*)  
  
 # Выходим из цикла, т.к. пуля уже потрачена  
 break  
  
 # Столкновение со стенами (отскок один раз)  
 for *wall* in *walls*:  
 if *bullet*['rect'].colliderect(*wall*):  
 if *bullet*['bounced']:  
 *bullets\_to\_remove*.append(*bullet*)  
 else:  
 *dx* = *bullet*['rect'].centerx - *wall*.centerx  
 *dy* = *bullet*['rect'].centery - *wall*.centery  
 if abs(*dx*) > abs(*dy*): # Горизонтальное столкновение  
 *bullet*['direction'].x \*= -1  
 else: # Вертикальное столкновение  
 *bullet*['direction'].y \*= -1  
 *bullet*['bounced'] = True  
 break  
  
 # Попадание в самого себя  
 # if bullet['rect'].colliderect(self.rect):  
 # bullets\_to\_remove.append(bullet)  
 # if not self.invincible: # Если неуязвимость не активна  
 # self.lives -= 1  
  
 # Включаем i-frames  
 # self.invincible = True  
 # self.invincible\_timer = self.invincible\_time  
  
 # if self.lives <= 0:  
 # self.die()  
  
 # break  
  
 # Проверяем выход за границы  
 if (*bullet*['rect'].x < 0 or *bullet*['rect'].x > *WIDTH* or  
 *bullet*['rect'].y < 0 or *bullet*['rect'].y > *HEIGHT*):  
 *bullets\_to\_remove*.append(*bullet*)  
  
 # Удаляем нужные пули  
 for *bullet* in *bullets\_to\_remove*:  
 if *bullet* in self.bullets:  
 self.bullets.remove(*bullet*)  
  
 def draw\_bullets(self, *screen*):  
 *"""Рисуем все пули (простой прямоугольник)."""* for *bullet* in self.bullets:  
 pygame.draw.rect(*screen*, (0, 0, 255), *bullet*['rect'])  
  
 def draw(self, *screen*):  
 if self.invincible:  
 # Мигаем с некоторой частотой  
 # (Например, раз в 200 мс скрываем спрайт)  
 *t* = pygame.time.get\_ticks() // 200  
 if *t* % 2 == 0:  
 # Рисуем обычный спрайт  
 *frame* = self.animations[self.current\_animation][self.current\_frame]  
 *screen*.blit(*frame*, self.rect.topleft)  
 else:  
 # Пропускаем кадр, не рисуем (эффект мигания)  
 pass  
 else:  
 # Обычная отрисовка  
 *frame* = self.animations[self.current\_animation][self.current\_frame]  
 if not self.facing\_right:  
 *frame* = pygame.transform.flip(*frame*, True, False)  
 *screen*.blit(*frame*, self.rect.topleft)  
 """Рисуем нужный кадр анимации (учитывая направление)."""  
  
 def update(self, *delta\_time*, *walls*):  
 *"""Главный метод, вызывается каждый кадр для обновления логики игрока."""* # Снижаем таймер неуязвимости  
 if self.invincible:  
 self.invincible\_timer -= *delta\_time* if self.invincible\_timer <= 0:  
 self.invincible\_timer = 0  
 self.invincible = False  
  
 self.move(*walls*)  
 self.update\_animation(*delta\_time*)

**Приложение №9.5. Файл класса «Level»**

import pygame, random  
from settings import *TILE\_SIZE*from enemy import Enemy  
  
class Level:  
 def \_\_init\_\_(self, *level\_data*):  
 self.walls = []  
 self.enemies = []  
 self.water = [] # Добавляем тайлы воды  
 self.traps = [] # Добавляем тайлы ловушек  
 self.bonuses = [] # Добавляем тайлы бонусов  
 self.start\_pos = (0, 0)  
 self.finish\_rect = None  
 self.width =0  
 self.height = 0  
 self.parse\_level(*level\_data*)  
 def random\_bool(self):  
 return random.choice([True, False])  
 def parse\_level(self, *level\_data*):  
 *max\_width* = 0  
 for *y*, *row* in enumerate(*level\_data*):  
 *max\_width* = max(*max\_width*, len(*row*))  
 for *x*, *tile* in enumerate(*row*):  
 *position* = pygame.Rect(*x* \* *TILE\_SIZE*, *y* \* *TILE\_SIZE*, *TILE\_SIZE*, *TILE\_SIZE*)  
 if *tile* == '1':  
 self.walls.append(*position*)  
 elif *tile* == 'E':  
 *enemy* = Enemy(*position*.x, *position*.y, speed=4, sprite\_sheet\_path="src/sprites/enemy.png",  
 tile\_width=32, tile\_height=32, facing\_right = self.random\_bool())  
 self.enemies.append(*enemy*)  
 elif *tile* == 'B':  
 # Создать босса (x, y) - левый верхний угол,  
 # но учтите, что это 2x2 тайла => можно сместить на 0,0 и просто создать 100x100 rect.  
 from boss import Boss  
 *boss* = Boss(*position*.x, *position*.y, *TILE\_SIZE*)  
 self.enemies.append(*boss*) # добавляем в тот же список enemies  
 elif *tile* == 'P':  
 self.start\_pos = (*position*.x, *position*.y)  
 elif *tile* == 'F':  
 self.finish\_rect = *position* elif *tile* == 'W': # Вода  
 self.water.append(*position*)  
 elif *tile* == 'T': # Ловушка  
 self.traps.append(*position*)  
 elif *tile* == 'S': # Бонус  
 self.bonuses.append(*position*)  
  
 self.width = *max\_width* \* *TILE\_SIZE* self.height = len(*level\_data*) \* *TILE\_SIZE* def draw(self, *screen*):  
 # Рисуем стены  
 for *wall* in self.walls:  
 pygame.draw.rect(*screen*, (0, 0, 0), *wall*)  
 # Рисуем воду  
 for *water* in self.water:  
 pygame.draw.rect(*screen*, (0, 0, 255), *water*)  
 # Рисуем ловушки  
 for *trap* in self.traps:  
 pygame.draw.rect(*screen*, (255, 0, 0), *trap*)  
 # Рисуем бонусы  
 for *bonus* in self.bonuses:  
 pygame.draw.rect(*screen*, (100, 0, 100), *bonus*)  
 # Рисуем финиш  
 if self.finish\_rect:  
 pygame.draw.rect(*screen*, (255, 215, 0), self.finish\_rect)

**Приложение №9.6. Файл класса «Enemy»**

import pygame  
import math  
from collections import deque  
from settings import *WIDTH*, *HEIGHT*, *TILE\_SIZE* # Предположим, у вас есть TILE\_SIZE = 50  
  
class Enemy:  
 def \_\_init\_\_(self, *x*, *y*, *speed*, *sprite\_sheet\_path*, *tile\_width*, *tile\_height*, *facing\_right*, *patrol\_points*=None):  
 # Уменьшенный прямоугольник (30x30 вместо 40x40) + смещение, чтобы центр врага оставался примерно там же  
 self.rect = pygame.Rect(*x* + 5, *y* + 5, 30, 30)  
 self.speed = *speed* self.patrol\_points = *patrol\_points* or []  
 self.current\_point = 0  
 self.direction = pygame.Vector2(0, 0)  
 self.detect\_radius = 250 # Радиус обнаружения игрока  
 self.chasing = False  
 self.last\_known\_position = None # Последняя известная позиция игрока  
  
 # Путь (список координат в пикселях), по которым двигается враг  
 self.path = []  
 self.path\_index = 0  
  
 # Анимации  
 self.animations = self.load\_sprite\_sheet(*sprite\_sheet\_path*, *tile\_width*, *tile\_height*)  
 self.current\_animation = 0 # 0: idle, 1: walk, 2: run, и т.д.  
 self.current\_frame = 0  
 self.frame\_timer = 0  
 self.animation\_speed = 150 # мс на кадр  
 self.facing\_right = *facing\_right* # Направление (True = вправо)  
  
 # -------------------------------  
 # Загрузка и анимации  
 # -------------------------------  
 def load\_sprite\_sheet(self, *filepath*, *tile\_width*, *tile\_height*):  
 *sheet* = pygame.image.load(*filepath*).convert\_alpha()  
 *sheet\_width*, *sheet\_height* = *sheet*.get\_size()  
 *sprites* = []  
  
 for *y* in range(0, *sheet\_height*, *tile\_height*):  
 *row* = []  
 for *x* in range(0, *sheet\_width*, *tile\_width*):  
 *rect* = pygame.Rect(*x*, *y*, *tile\_width*, *tile\_height*)  
 *frame* = *sheet*.subsurface(*rect*)  
 if not self.is\_frame\_empty(*frame*):  
 *row*.append(*frame*)  
 *sprites*.append(*row*)  
  
 return *sprites* def is\_frame\_empty(self, *frame*):  
 for *x* in range(*frame*.get\_width()):  
 for *y* in range(*frame*.get\_height()):  
 if *frame*.get\_at((*x*, *y*))[3] > 0: # Альфа-канал > 0  
 return False  
 return True  
  
 def set\_animation(self, *animation\_index*):  
 if self.current\_animation != *animation\_index*:  
 self.current\_animation = *animation\_index* self.current\_frame = 0 # Сброс на первый кадр  
  
 def update\_animation(self, *delta\_time*):  
 self.frame\_timer += *delta\_time* if self.frame\_timer >= self.animation\_speed:  
 self.frame\_timer = 0  
 self.current\_frame += 1  
 # Зацикливаем  
 if self.current\_frame >= len(self.animations[self.current\_animation]):  
 self.current\_frame = 0  
  
 # -------------------------------  
 # Логика врага  
 # -------------------------------  
 def update(self, *player*, *walls*, *delta\_time*, current\_time=None):  
 # 1. Анимация  
 self.update\_animation(*delta\_time*)  
  
 # 2. Определяем, видим ли игрока  
 if self.can\_see\_player(*player*, *walls*):  
 self.chasing = True  
 self.last\_known\_position = *player*.rect.center  
 self.set\_animation(2) # Пусть будет "2" для преследования  
 self.go\_to\_position(self.last\_known\_position, *walls*)  
 elif self.chasing and self.last\_known\_position:  
 # Если преследуем, но игрока не видно: идём к последней известной позиции  
 self.set\_animation(2)  
 # Если уже достигли, сбрасываем  
 *dist* = pygame.Vector2(self.rect.center).distance\_to(self.last\_known\_position)  
 if *dist* < 5:  
 self.last\_known\_position = None  
 self.chasing = False  
 else:  
 self.go\_to\_position(self.last\_known\_position, *walls*)  
 else:  
 # Патрулирование  
 if self.patrol\_points:  
 self.set\_animation(1) # 1: walk (или 0: idle)  
 self.patrol(*walls*)  
 else:  
 # Если нет patrol\_points, враг просто стоит idle  
 self.set\_animation(0)  
  
  
 def go\_to\_position(self, *target\_pos*, *walls*):  
 *"""  
 Двигаемся к позиции target\_pos.  
 Используем path (если пустой или устарел — перестраиваем).  
 """* # Проверяем, есть ли путь и не достигли ли конца  
 if not self.path or self.path\_index >= len(self.path):  
 # Строим новый путь BFS  
 self.path = self.build\_path(self.rect.center, *target\_pos*, *walls*)  
 self.path\_index = 0  
  
 if self.path:  
 # Двигаемся к текущей точке из path  
 *current\_target* = self.path[self.path\_index]  
 *direction* = pygame.Vector2(*current\_target*) - pygame.Vector2(self.rect.center)  
 if *direction*.length\_squared() > 0:  
 *direction* = *direction*.normalize()  
 self.facing\_right = (*direction*.x > 0)  
  
 # Перемещаемся  
 *movement* = *direction* \* self.speed  
 *old\_rect* = self.rect.copy()  
 self.rect.x += *movement*.x  
 self.collide(*walls*, 'x', *old\_rect*)  
 self.rect.y += *movement*.y  
 self.collide(*walls*, 'y', *old\_rect*)  
  
 # Проверяем, достигли ли мы "узла"  
 if pygame.Vector2(self.rect.center).distance\_to(*current\_target*) < 5:  
 self.path\_index += 1  
  
  
  
 def can\_see\_player(self, *player*, *walls*):  
 *"""  
 Проверяет, видит ли враг игрока (без учёта сложного pathfinding).  
 Просто луч и радиус видимости.  
 """  
 distance* = pygame.Vector2(self.rect.center).distance\_to(*player*.rect.center)  
 if *distance* > self.detect\_radius:  
 return False  
  
 # Луч от врага до игрока  
 *ray* = pygame.Vector2(*player*.rect.center) - pygame.Vector2(self.rect.center)  
 *ray\_length* = *ray*.length()  
 if *ray\_length* == 0:  
 return False  
  
 *ray\_direction* = *ray*.normalize()  
 *current\_pos* = pygame.Vector2(self.rect.center)  
 *steps* = int(*ray\_length*) # По одному пикселю  
  
 for *\_* in range(*steps*):  
 *current\_pos* += *ray\_direction* # Если встречаем стену - не видим  
 if any(*wall*.collidepoint(*current\_pos*) for *wall* in *walls*):  
 return False  
 return True  
  
 # -------------------------------  
 # BFS-поиск пути по сетке  
 # -------------------------------  
 def build\_path(self, *start\_pos*, *end\_pos*, *walls*):  
 *"""  
 Простейший поиск пути (BFS) по сетке с размером тайла TILE\_SIZE.  
 Возвращает список координат (x, y) в пикселях от start\_pos до end\_pos.  
 Если путь не найден — возвращаем пустой список.  
 """* # Конвертируем пиксельные координаты в координаты тайлов  
 *start\_tile* = (int(*start\_pos*[0] // *TILE\_SIZE*), int(*start\_pos*[1] // *TILE\_SIZE*))  
 *end\_tile* = (int(*end\_pos*[0] // *TILE\_SIZE*), int(*end\_pos*[1] // *TILE\_SIZE*))  
  
 if *start\_tile* == *end\_tile*:  
 return [] # Уже там  
  
  
# Собираем множество заблокированных тайлов на основе walls  
 *blocked\_tiles* = set()  
 for *wall* in *walls*:  
 # wall.x, wall.y, wall.width, wall.height  
 # Перебираем тайлы, которые занимает стенка (примерно)  
 *x\_start* = *wall*.left // *TILE\_SIZE  
 x\_end* = (*wall*.right - 1) // *TILE\_SIZE  
 y\_start* = *wall*.top // *TILE\_SIZE  
 y\_end* = (*wall*.bottom - 1) // *TILE\_SIZE* for *tx* in range(*x\_start*, *x\_end* + 1):  
 for *ty* in range(*y\_start*, *y\_end* + 1):  
 *blocked\_tiles*.add((*tx*, *ty*))  
  
 # BFS  
 *queue* = deque([*start\_tile*])  
 *came\_from* = {*start\_tile*: None}  
 *found\_path* = False  
  
 while *queue*:  
 *current* = *queue*.popleft()  
 if *current* == *end\_tile*:  
 *found\_path* = True  
 break  
  
 for *nx*, *ny* in self.get\_neighbors(*current*):  
 if (*nx*, *ny*) not in *came\_from* and (*nx*, *ny*) not in *blocked\_tiles*:  
 *came\_from*[(*nx*, *ny*)] = *current  
 queue*.append((*nx*, *ny*))  
  
 if not *found\_path*:  
 return []  
  
 # Восстанавливаем путь по словарю came\_from  
 *path\_tiles* = []  
 *cur* = *end\_tile* while *cur* is not None:  
 *path\_tiles*.append(*cur*)  
 *cur* = *came\_from*[*cur*]  
 *path\_tiles*.reverse()  
  
 # Преобразуем список тайлов в пиксельные координаты (центр тайла)  
 *result\_path* = []  
 for (*tx*, *ty*) in *path\_tiles*:  
 *cx* = *tx* \* *TILE\_SIZE* + *TILE\_SIZE* // 2  
 *cy* = *ty* \* *TILE\_SIZE* + *TILE\_SIZE* // 2  
 *result\_path*.append((*cx*, *cy*))  
  
 return *result\_path* def get\_neighbors(self, *tile\_xy*):  
 *"""Соседние тайлы (4 направления, без диагоналей)."""* (*x*, *y*) = *tile\_xy* return [  
 (*x*+1, *y*),  
 (*x*-1, *y*),  
 (*x*, *y*+1),  
 (*x*, *y*-1),  
 ]  
  
 # -------------------------------  
 # Коллизии  
 # -------------------------------  
 def collide(self, *walls*, *axis*, *old\_rect*):  
 *"""  
 Если при перемещении по оси X или Y столкнулись со стеной,  
 откатываем позицию. Учитывая, что self.rect меньше, чем тайл.  
 """* for *wall* in *walls*:  
 if self.rect.colliderect(*wall*):  
 if *axis* == 'x':  
 self.rect.x = *old\_rect*.x  
 elif *axis* == 'y':  
 self.rect.y = *old\_rect*.y  
  
 # -------------------------------  
 # Рисование  
 # -------------------------------  
 def draw(self, *screen*):  
 *frame* = self.animations[self.current\_animation][self.current\_frame]  
 if not self.facing\_right:  
 *frame* = pygame.transform.flip(*frame*, True, False)  
 *screen*.blit(*frame*, self.rect.topleft)  
  
 #Для отладки можно нарисовать путь  
 for *i* in range(len(self.path) - 1):  
 pygame.draw.line(*screen*, (255, 0, 0), self.path[*i*], self.path[*i*+1], 2)