

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

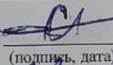
Направление подготовки: 09.03.01 «Информатика и вычислительная техника»
Профиль подготовки: «Программное обеспечение средств вычислительной техники и автоматизированных систем»

БАКАЛАВРСКАЯ РАБОТА

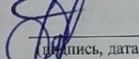
на тему

**ИГРОВОЕ ПРИЛОЖЕНИЕ ПОД ОС ANDROID В
ЖАНРЕ «ПЛАТФОРМЕР»**

Студент

 Сидоров Н.Р.
(подпись, дата) 16.06.2023

Руководитель

 Федюнин Р.Н.
(подпись, дата) 19.06.2023

Нормоконтролёр

 Бычков А.С.
(подпись, дата) 20.06.2023

Работа допущена к защите (протокол заседания кафедры от 02.06.2023 г. №14)

Заведующий кафедрой

 М.А. Митрохин
(подпись)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 20.06.23 № 23)

Секретарь ГЭК

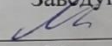
 Е.И. Гурин
(подпись)

Пенза, 2023

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой
 М.А. Митрохин
«26» апреля 2023г.

**ЗАДАНИЕ
НА БАКАЛАВРСКУЮ РАБОТУ**

1. Студент Сидоров Никита Романович гр.19ВВП1 факультета вычислительной техники.

2. Тема работы: Игровое приложение под ОС *Android* в жанре «платформер»

Тема утверждена приказом ПГУ №458/о от "26" апреля 2023г.

3. Руководитель бакалаврской работы к.т.н., доцент, каф. ВТ ПГУ Федюнин Р.Н.

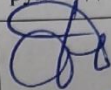
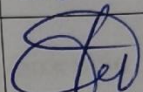
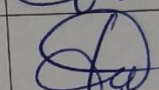
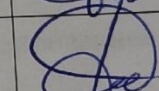
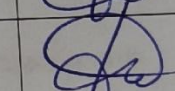
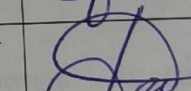
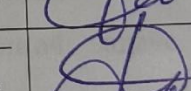
4. Задание на работу (назначение разработки, исходные данные и т.п.)

Разработать приложение на тему «Игровое приложение под ОС *Android* в жанре «платформер». Приложение должно иметь адаптивный интерфейс для работы на различных устройствах с ОС *Android*, обладать отзывчивым и интуитивно понятным управлением, качественным аудиовизуальным сопровождением и при этом быть достаточно оптимизированным.

5. Перечень подлежащих разработке вопросов.

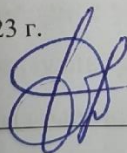
Требуется провести анализ предметных областей: изучить существующие игровые приложения в жанре «платформер» на ОС *Android*. Выбрать платформу разработки, графический редактор, язык программирования и редактор кода, наиболее подходящие для процесса разработки приложения. Определить архитектуру приложения и создать графические ресурсы. Изучить сферу разработки игровых приложений и использовать полученные знания в работе.

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Изучение предметной области	32	26.04.2023 – 30.04.2023	
2	Анализ требований	18	30.04.2023 – 05.05.2023	
3	Выбор технологий разработки	22	30.04.2023 – 05.05.2023	
4	Создание графических ресурсов	90	5.05.2023 – 26.05.2023	
5	Разработка приложения	150	5.05.2023 – 26.05.2023	
6	Тестирование приложения	24	5.05.2023 – 26.05.2023	
7	Оформление пояснительной записки	48	26.05.2023 – 31.05.2023	

Дата выдачи задания "26" апреля 2023 г.

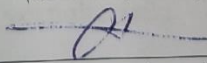
Руководитель бакалаврской работы



Федюнин Р.Н.

(подпись, дата)

Задание к исполнению принял студент


(подпись, дата)

Сидоров Н.Р.

Работу к защите допустить
Декан факультета



Фионова Л.Р.

(подпись, дата)

20.06.2023

Реферат

Бакалаврская работа содержит 116 листов, 50 рисунков, 3 таблицы, 5 источников и 3 приложения

ПЛАТФОРМЕР, МОБИЛЬНАЯ РАЗРАБОТКА, *ANDROID*, РАЗРАБОТКА ИРЫ НА ЮНИТИ, ГЕЙМДИЗАЙН, АНИМАЦИИ, ВИЗУАЛЬНЫЕ ЭФФЕКТЫ

Цель работы – разработать качественное игровое приложение под систему *Android* в жанре «платформер».

Объектами разработки являются:

- адаптивное мобильное игровое приложение, оптимизированное для различных типов устройств;
- создание графических ресурсов для использования в разрабатываемом приложении, работа с анимациями.

В результате выполненных работ было разработано игровое приложение, которое полностью соответствует поставленной задаче, а именно:

- имеет отзывчивое сенсорное управление;
- качественную аудиовизуальную составляющую;
- адаптируется под разные виды экранов и обладает достаточной производительностью на мобильных устройствах.

					ПГУ1.090301.22.001 ПЗ		
Изм.	Лист	№ докум.	Подпись	Дата	Лит.	Лист	Листов
Разраб.		Сидоров Н.Р.					
Провер.		Федюнин Р.Н.				2	116
Н. Контр		Бычков А.С.			ФВТ, каф ВТ		
Утверд.		Митрохин М.А.			гр. 19ВВП1		

Содержание

Перечень сокращений и обозначений	6
Введение	7
1 Игровое приложение под ОС <i>Android</i>	8
1.1 Анализ темы.....	8
1.2 Аналоги игрового приложения	10
1.2.1 « <i>Dead Cells</i> »	10
1.2.2 « <i>Oddmar</i> ».....	11
1.2.3 « <i>Super Meat Boy Forever</i> »	12
1.2.4 « <i>Limbo</i> ».....	13
1.2.5 « <i>Downwell</i> ».....	14
1.3 Постановка задачи	15
1.4 Выбор платформы разработки	16
1.4.1 <i>Unity</i>	16
1.4.2 <i>Unreal Engine</i>	17
1.4.3 <i>Godot Engine</i>	18
1.4.4 <i>CryEngine</i>	19
1.5 Выбор графического редактора	21
1.5.1 <i>Aseprite</i>	22
1.5.2 <i>Adobe Photoshop</i>	23
1.5.3 <i>Pro Motion NG</i>	24
1.6 Выбор языка программирования	26
1.6.1 <i>C#</i>	26
1.6.2 <i>C++</i>	27
1.6.3 <i>GDScript</i>	29
1.7 Выбор редактора кода	30
1.7.1 <i>Visual Studio</i>	30
1.7.2 <i>JetBrains IntelliJ IDEA</i>	31
1.8 Выбор инструментов разработки.....	32

1.8.1 Платформа разработки	32
1.8.2 Графический редактор.....	33
1.8.3 Редактор кода.....	34
1.8.4 Язык программирования	35
1.9 Архитектура приложения.....	35
1.9.1 <i>Component-based architecture</i>	35
1.9.2 <i>Model-View-Controller</i>	36
1.10 Разработка игрового приложения	38
1.10.1 Концепция	38
1.10.2 Графика и дизайн.....	42
1.10.3 Программирование	47
1.10.3.1 Классы используемые в проекте	49
1.10.3.2 <i>PlayerControls</i>	50
1.10.3.3 <i>PlayerCombat</i>	51
1.10.3.4 <i>Player</i>	51
1.10.3.5 <i>GameManger</i>	52
1.10.3.6 <i>SoundManager</i>	53
1.10.3.7 <i>Particles</i> скрипты	53
1.10.3.8 <i>Sound</i> скрипты.....	53
1.10.3.9 <i>Parallax</i>	53
1.10.3.10 <i>OnDestroy</i>	54
1.10.4 Звук и музыка.....	54
1.10.5 Тестирование и отладка.....	54
1.11 Описание приложения.....	55
2 Охрана труда.....	60
2.1 Негативные факторы, воздействующие на оператора ПЭВМ	60
2.1.1 Оборудование рабочего места	60
2.1.2 Нормы освещенности	63
2.1.3 Электробезопасность.....	63

2.1.4 Пожарная безопасность	64
2.1.5 Нормы труда и отдыха	64
Заключение	66
Список используемых источников.....	67
Приложение А – UML - диаграммы.....	68
Приложение Б – Листинг программы	72
Приложение В – Презентация	108

Перечень сокращений и обозначений

В настоящей работе применяют следующие сокращения и обозначения

Баг (*Bug*) – ошибка в процессе разработки, дефект программы

Геймдизайнер (*Game Designer*) – специалист разработки игрового контента, игровых механик и общего игрового опыта

Геймплей (*Gameplay*) – взаимодействие игрока с игровым миром и механиками игры

Дизайнер уровней (*Level Designer*) – специалист по созданию уровней, миссий и сценариев

Коллайдер (*Collider*) – компонент для определения столкновений объектов в виртуальном пространстве

ПК (*PC*) – персональный компьютер

Пиксель-арт (*Pixel Art*) – техника создания графики, основанная на использовании пикселей

Рывок (*Dash*) – способ быстрого перемещения в заданном направлении

Спрайт – графический элемент, используемый в разработке игр и анимации

Точка интереса (*Point of Interest*) – место или элемент, который привлекает внимание игрока

AAA-игры – высокобюджетные, крупномасштабные и высококачественные видеоигры

IDE (*Integrated Development Environment*) – интегрированная среда разработки

Рейкастинг (*Raycasting*) – инструмент для определения столкновений и обнаружения препятствий на пути луча

UI (*User Interface*) – пользовательский интерфейс

VR (*Virtual Reality*) – виртуальная реальность

Введение

В наше время технический прогресс неустанно движется вперед. Еще совсем недавно было трудно представить, что телефон может стать портативным устройством с функционалом, покрывающим потребности современного человека. Развитие технологий позволяет нам выполнять множество задач на мобильных устройствах, начиная от общения и получения информации до выполнения работы и управления делами. Однако, с ростом функциональности современных смартфонов, люди все чаще обращаются к ним в качестве источника развлечений. Мобильные игры, приложения для потокового воспроизведения музыки, социальные сети – все это стало неотъемлемой частью нашей повседневной жизни.

Игровая индустрия является одной из самых динамично растущих и прибыльных отраслей современного мира. Каждый год игры привлекают миллионы игроков со всего мира, и спрос продолжает расти. Таким образом, разработка игрового приложения может предоставить возможность создать востребованный продукт.

Также, разработка игрового приложения предоставляет возможность проявить творческие и технические умения. Работа над игровым приложением требует навыков программирования, дизайна и управления проектом, что является ценным опытом в области разработки программного обеспечения.

Android является самой распространенной операционной системой, используясь на миллионах смартфонов, планшетов и других устройств по всему миру. Широкая аудитория, удобные инструменты разработки, возможности монетизации и поддержка сообщества делают *Android* наиболее привлекательной платформой для создания и распространения игровых приложений.

					ПГУ1.090301.22.001 ПЗ	Лист
						7
Изм.	Лист	№ докум.	Подпись	Дата		

1 Игровое приложение под ОС *Android*

1.1 Анализ темы

Первые мобильные игры появились еще в конце 20 века, когда мобильные телефоны стали получать возможность отображать простые графические элементы и игровые приложения.

Одной из самых популярных первых мобильных игр были игра «*Snake*» (Змейка), которая была предустановлена на многих моделях *Nokia* в 1997 году. Игра представляла собой простую аркаду, где игрок управлял змейкой, пытаясь съесть пиксельные точки, при этом избегая столкновения со своим хвостом.

Настоящий прорыв в мобильных играх произошел с выпуском первых смартфонов. Эти устройства, оснащенные передовыми технологиями и мощными процессорами, открыли новые возможности для разработчиков игр. С появлением смартфонов игровой опыт стал намного более привлекательным и захватывающим. Благодаря своим возможностям в области визуальных эффектов, мультимедийных функций и сенсорного управления, интерактивность и разнообразие игровых механик возросли в разы.

В период с 2000 по 2003 годы мобильные устройства получили доступ к интернету, что открыло двери к онлайн-взаимодействиям и многопользовательским приложениям. С появлением интернет-доступа, смартфоны стали неотъемлемой частью повседневной жизни, предоставляя пользователю доступ к различным сервисам и контенту в любом месте и в любое время.

Одним из ключевых моментов в развитии мобильных игр стал запуск магазина приложений *App Store* компании *Apple* в 2008 году. Этот шаг предоставил разработчикам возможность легко распространять свои игры по всему миру и достичь огромной аудитории пользователей. Аналогично, в 2012 году *Google* запустила свой магазин приложений *Google Play*, что дополнило

экосистему *Android*. Оба магазина предоставили удобную платформу для распространения, продажи и обновления игровых приложений, способствуя значительному росту индустрии мобильных игр.

Мобильные игры предлагают разнообразие жанров, чтобы удовлетворить интересы и предпочтения игроков. Они обеспечивают увлекательный геймплей и возможность погрузиться в виртуальные миры, независимо от места и времени. Некоторые игры предлагают быструю и динамичную активность, где игроки могут соревноваться за победу и ощутить адреналин. Другие же предлагают интеллектуальные вызовы, где игроки могут развивать свои навыки решения головоломок и стратегического мышления. Еще одним жанром, являются игры с элементами приключения, они позволяют игрокам окунуться в увлекательную историю, испытать азарт от приключений и открытий. Независимо от того, ищет ли пользователь быстрые испытания или глубокий сюжет, мобильные игры обещают подарить уникальный игровой опыт.

Для создания мобильных игр необходимы специализированные компетенции и умения. Разработчики игр должны обладать глубокими знаниями в области программирования, дизайна и анимации. Они должны быть в состоянии создавать увлекательный геймплей, создавать интерактивные и интуитивно понятные интерфейсы. Разработчики должны обладать знаниями и навыками программирования на языках *C++*, *C#*, *GScript* и т.п. Они должны также быть знакомы с различными средами разработки, такими как *Unity*, *Godot Engine*, *Unreal Engine* и т.п.

Одной из ключевых задач при разработке мобильных игр является оптимизация производительности и адаптация под различные устройства. Разработчики должны учесть различные разрешения экранов и аппаратные возможности устройств, чтобы графический и игровой контент был оптимизирован и обеспечивал плавную работу на всех современных устройствах.

1.2 Аналоги игрового приложения

Фрагменты геймплея представлены на рисунках 1.1, 1.2, 1.3, 1.4, 1.5.

1.2.1 «Dead Cells»

«Dead Cells» – это независимая видеоигра в жанре рогалик-платформера, разработанная *Motion Twin* и выпущенная в 2018 году, позже была портирована на системы *iOS* и *Android*. Игра сочетает в себе элементы метроидвании и рогалика, предлагая игроку исследование различных уровней, борьбу с врагами и сбор улучшений для своего персонажа.

В «Dead Cells» игрок управляет безымянным бессмертным героем, который исследует лабиринты, наполненные опасными врагами и ловушками. Персонаж может использовать разнообразное оружие, умения и магические способности для сражения с противниками. При каждой смерти персонаж возвращается в начало игры, но сохраняет некоторые улучшения и открытые пути, что позволяет прогрессировать дальше с каждым прохождением.

«Dead Cells» отличается от других рогаликов своей быстрой и плавной игровой механикой, а также красивой пиксельной графикой. Она также предлагает разнообразие предметов, оружия и способностей, что позволяет игрокам менять свой стиль игры и экспериментировать с различными комбинациями. Фрагмент геймплея «Dead Cells» изображен на рисунке 1.1.

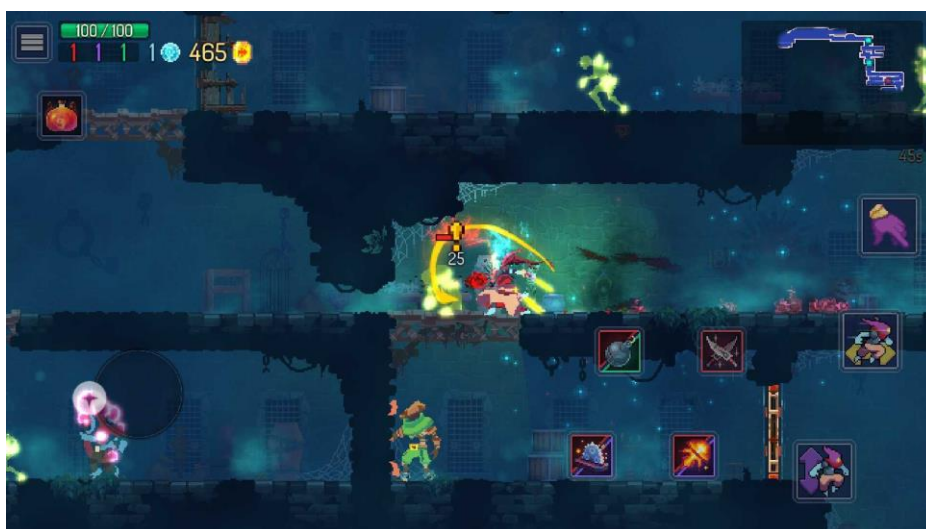


Рисунок 1.1 – Фрагмент геймплея «Dead Cells»

1.2.2 «Oddmar»

«Oddmar» – это мобильная игра, разработанная и выпущенная студией *Mobge Ltd*. Она доступна для устройств на базе операционной системы *iOS* и *Android*.

«Oddmar» – это платформер с элементами головоломок и приключений. Игрок управляет персонажем по имени *Oddmar*, викингом, который стремится найти свое место в мире и доказать свою ценность. В игре представлены разнообразные уровни с препятствиями, головоломками и битвами с врагами.

Графика «Oddmar» выполнена в красочном и подробно проработанном стиле, а музыкальное сопровождение создает атмосферу викингской саги. Игра предлагает различные навыки и способности, которые можно улучшать по мере прохождения.

«Oddmar» получила положительные отзывы за отличную графику, увлекательный геймплей и захватывающую историю. Она обеспечивает приятный игровой опыт на мобильных устройствах и пользуется популярностью среди любителей платформеров и приключенческих игр. Фрагмент геймплея «Oddmar» изображен на рисунке 1.2.

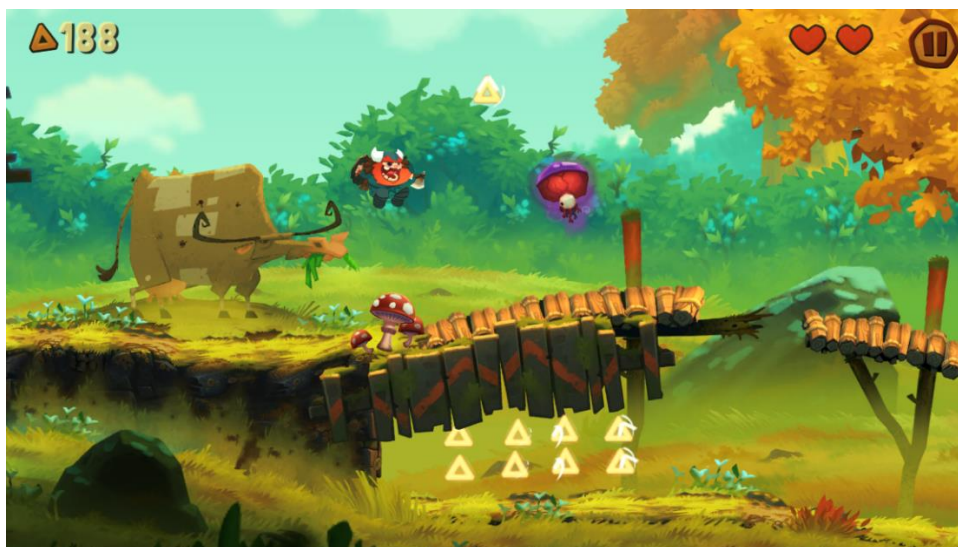


Рисунок 1.2 – Фрагмент геймплея «Oddmar»

1.2.3 «*Super Meat Boy Forever*»

«*Super Meat Boy Forever*» – это видеоигра, являющаяся прямым продолжением оригинальной *Super Meat Boy*, разработанной *Team Meat*. Игра была выпущена в 2020 году и доступна на различных платформах, включая *Nintendo Switch*, *PlayStation 4*, *Xbox One*, и *PC*, в 2023 году была портирована на платформы *iOS* и *Android*.

«*Super Meat Boy Forever*» является платформером, в котором игрок управляет персонажем по имени *Meat Boy* или его партнером *Bandage Girl*, которые пытаются спасти своего похищенного ребенка. Геймплей игры характеризуется быстрыми и интенсивными уровнями, где игрок должен преодолевать различные препятствия, ловушки и врагов, чтобы достичь конечной цели.

В отличие от оригинала, «*Super Meat Boy Forever*» предлагает процедурно генерируемые уровни, это означает, что каждый проход игры будет уникальным. Также в игре добавлен элемент автоматического бега, что придает еще больше динамики и скорости игровому процессу.

«*Super Meat Boy Forever*» продолжает традицию оригинальной игры, предлагая высокий уровень сложности и вызовов для игроков, однако также внедряет новые механики и усовершенствования. Она была хорошо принята критиками и фанатами серии, предлагая увлекательное и захватывающее платформерное испытание. Фрагмент геймплея «*Super Meat Boy Forever*» изображен на рисунке 1.3.

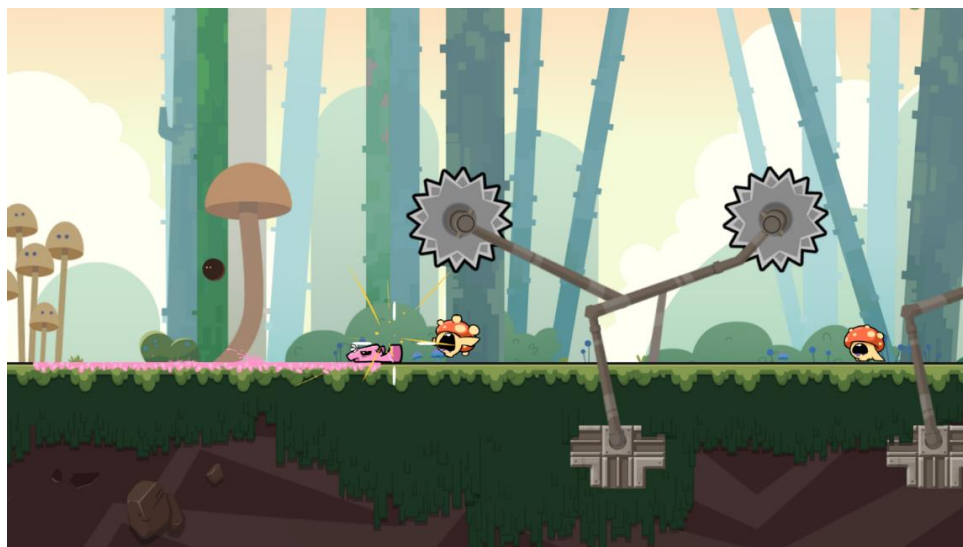


Рисунок 1.3 – Фрагмент геймплея «*Super Meat Boy Forever*»

1.2.4 «*Limbo*»

«*Limbo*» – это платформер-головоломка, разработанная датской компанией Playdead. Игра была выпущена в 2010 году и доступна на различных платформах, включая *Xbox*, *PlayStation*, *PC*, *iOS* и *Android*.

В «*Limbo*» игрок управляет безымянным мальчиком, который оказывается в темном и загадочном мире. Геймплей игры основан на решении головоломок и преодолении препятствий, таких как ловушки и враги. Игра в основном работает с черно-белой палитрой и создает атмосферу пугающей и мрачной сказки.

Одна из особенностей «*Limbo*» – это ее атмосфера и визуальный стиль. Игра использует минималистичную графику и шедевральные эффекты освещения, создавая атмосферу таинственности и напряжения. Музыкальное сопровождение также способствует усилению атмосферы и эмоциональной глубины игры.

«*Limbo*» была хорошо принята критиками и игроками благодаря своей уникальной атмосфере, головоломкам и истории. Она стала коммерческим успехом и получила несколько наград и номинаций в различных категориях, включая искусство, геймплей и звуковое оформление. Фрагмент геймплея «*Limbo*» изображен на рисунке 1.4.



Рисунок 1.4 – Фрагмент геймплея «*Limbo*»

1.2.5 «*Downwell*»

«*Downwell*» – это пиксельный платформер с вертикальной прокруткой, разработанный и выпущенный студией Morrpin. Игра доступна на различных платформах, включая *Android*, *iOS*, *PlayStation 4*, *Nintendo Switch* и *PC*.

В «*Downwell*» игрок управляет персонажем, падающим в узкий колодец, полный врагов и опасностей. Геймплей основан на стрельбе и прыжках с использованием «пистолетных ботинок», которые позволяют персонажу выполнять прыжки и атаковать врагов. Цель игры - добраться как можно глубже в колодец, преодолевая препятствия и сражаясь с врагами на пути.

«*Downwell*» отличается своим уникальным визуальным стилем, который включает в себя пиксельную графику и ограниченную палитру цветов. Игра также привлекает внимание своим интенсивным геймплеем и нестандартными механиками.

«*Downwell*» была хорошо принята игроками и критиками, получив высокие оценки за свою реиграбельность, динамичность и стильную пиксельную графику. Фрагмент геймплея «*Downwell*» изображен на рисунке 1.5.

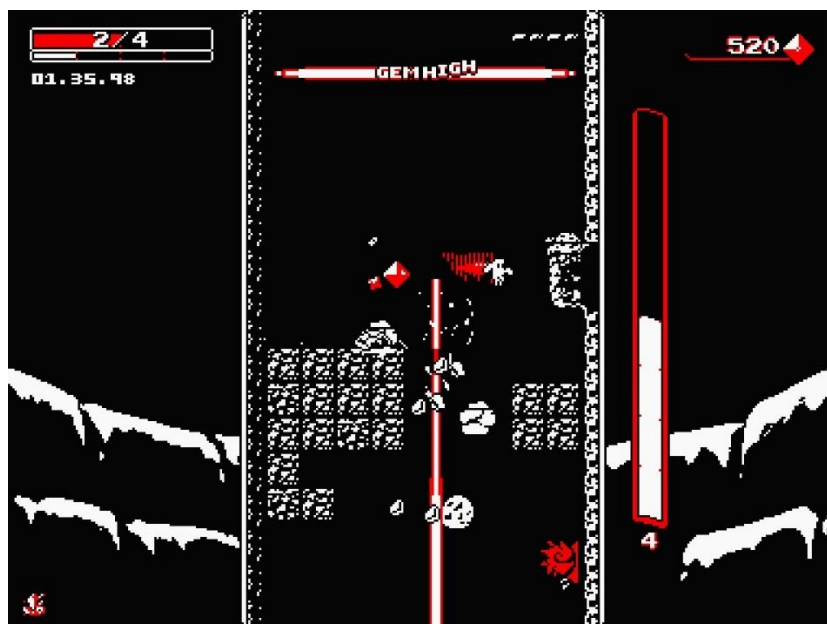


Рисунок 1.5 – Фрагмент геймплея «Downwell»

1.3 Постановка задачи

Необходимо разработать игровое приложение под ОС *Android*. Приложение должно удовлетворять следующие требования:

- интуитивно понятное и отзывчивое сенсорное управление;
- качественные графические элементы и анимации;
- качественное аудиосопровождение;
- обучающий уровень, предлагающий игроку опробовать базовые механики и возможности для прохождения игры;
- обеспечить баланс сложности игры;
- реализовать интерфейс пользователя, включающий в себя:
 - 1) главное меню с инструментами управления приложением;
 - 2) внутриигровой интерфейс с инструментами управления персонажем и приложением;
- адаптивное разрешение под разные виды экранов.

1.4 Выбор платформы разработки

На сегодняшний день существует множество различных инструментов, как платных, так и бесплатных, предназначенных для разработки игр. Они охватывают широкий диапазон возможностей и функциональности, чтобы помочь разработчикам создавать качественные игры.

1.4.1 Unity

Одним из самых популярных инструментов является *Unity*, который предоставляет широкий спектр функций и возможностей разработки игр для различных платформ, включая *Android*.

Unity (Unity3D) имеет ряд особенностей:

- кроссплатформенность: *Unity* позволяет разработчикам создавать игры, которые могут быть запущены на различных платформах с минимальными изменениями. Это экономит время и усилия разработчиков и позволяет быстро адаптировать игры под разные устройства;
- интуитивный интерфейс: *Unity* имеет дружелюбный пользовательский интерфейс. Интерфейс позволяет легко создавать и организовывать ресурсы игры, настраивать компоненты и скрипты, а также управлять сценами и анимацией;
- расширяемость: *Unity* имеет активное сообщество разработчиков и огромную библиотеку пакетов и расширений. Разработчики могут использовать готовые решения, созданные другими разработчиками, или создавать собственные пакеты для расширения возможностей;
- мощный игровой движок: *Unity* обладает мощным игровым движком, который предоставляет широкий набор инструментов для создания игровых механик, управления физикой, реализации искусственного интеллекта, создания визуальных эффектов и многое другое;
- графика и анимация: *Unity* поддерживает создание как 2D, так и 3D графики. Он предлагает мощные инструменты для создания реалистичных

визуальных эффектов, освещения, текстур и анимаций. *Unity* также поддерживает использование шейдеров и пост-обработки для создания высококачественной графики.

1.4.2 *Unreal Engine*

Еще одним из распространенных инструментов является *Unreal Engine*, который предоставляет высококачественные графические возможности и мощный игровой движок. *Unreal Engine* также поддерживает разработку игр для различных платформ, включая *Android*.

Unreal Engine имеет ряд особенностей:

- мощный игровой движок: *Unreal Engine* является одним из наиболее мощных и передовых игровых движков на рынке. Он предлагает широкий спектр функций и возможностей для создания высококачественных игр;
- фотореалистичная графика: *Unreal Engine* обладает впечатляющими графическими возможностями, включая фотореалистичную графику, динамическое освещение, физическую симуляцию и визуальные эффекты. Это позволяет создавать визуально потрясающие игры с высоким уровнем реализма;
- инструменты для разработки AAA-игр: *Unreal Engine* предлагает богатый набор инструментов и функций, которые помогают разработчикам создавать крупномасштабные и высококачественные игры. Включает в себя редактор миров, систему анимации, физическую симуляцию, систему искусственного интеллекта и многое другое;
- кроссплатформенная поддержка: *Unreal Engine* поддерживает разработку игр для различных платформ, включая мобильные устройства. Это позволяет достичь широкой аудитории пользователей и максимизировать охват рынка;
- большое сообщество и ресурсы: *Unreal Engine* имеет активное и обширное сообщество разработчиков, что обеспечивает доступ к различным ресурсам, обучающему материалу, документации и поддержке. Это делает

процесс разработки более поддерживаемым и помогает в решении проблем и вызовов.

1.4.3 Godot Engine

Godot Engine представляет собой гибкий игровой движок с открытым игровым кодом. Преимуществом *Godot Engine* является легковесность, возможность работы с любых устройств, даже через браузер. Он поддерживает разработку игр для различных платформ, включая *Android*.

Godot Engine имеет ряд особенностей:

- активное сообщество: *Godot Engine* имеет сообщество разработчиков, которые активно поддерживают и развивают движок. Существует множество ресурсов, форумов, tutorиалов и расширений, которые помогают новичкам и опытным разработчикам находить поддержку и делиться своими знаниями и опытом;

- бесплатность и открытый исходный код: *Godot Engine* является бесплатным для использования и распространения, что делает его доступным для всех разработчиков. Кроме того, он имеет открытый исходный код, что позволяет разработчикам изучать его, вносить изменения и улучшения, а также делиться своими наработками с сообществом;

- скриптинг на различных языках: *Godot Engine* поддерживает скриптинг на нескольких языках программирования, включая *GDScript* (язык, специально разработанный для *Godot*), *C#*, *Python* и другие. Это позволяет разработчикам выбирать наиболее подходящий язык для реализации логики игры и взаимодействия с объектами;

- мощный и гибкий игровой движок: *Godot Engine* предлагает мощный и гибкий игровой движок, способный обрабатывать разнообразные игровые жанры и стили. Он поддерживает 2D и 3D графику, анимацию, физику, звук, и многое другое, что позволяет создавать разнообразные игровые проекты;

- удобный редактор сцен: *Godot Engine* обладает удобным визуальным редактором сцен, который позволяет создавать и настраивать игровые

объекты, компоненты и их взаимодействие. Разработчики могут создавать сложные сцены путем перетаскивания и настройки компонентов, что упрощает процесс создания игровых уровней и мира;

- кроссплатформенность: *Godot Engine* позволяет создавать игры для различных платформ, включая *Windows, macOS, Linux, Android, iOS* и другие. Это делает его удобным инструментом для разработки игр, которые могут быть запущены на разных устройствах и операционных системах.

1.4.4 *CryEngine*

CryEngine – это игровой движок, разработанный компанией *Crytek*. Он представляет собой мощный инструмент для создания высококачественных компьютерных игр с продвинутой графикой, физикой и другими технологическими возможностями.

CryEngine имеет ряд особенностей:

- визуальные возможности: *CryEngine* славится своими продвинутыми графическими возможностями. Он поддерживает фотореалистичное отображение света и теней, динамическое освещение, реалистичные отражения и преломления, высококачественные текстуры и детализацию окружающей среды. Это позволяет создавать впечатляющие и реалистичные визуальные эффекты;

- физическая симуляция: *CryEngine* обладает продвинутой физической симуляцией, которая позволяет разработчикам создавать реалистичные физические взаимодействия с объектами в игровом мире. Это включает реалистичную анимацию персонажей, динамическую деформацию объектов и взаимодействие с жидкостями;

- инструменты для создания игрового контента: *CryEngine* предоставляет разработчикам широкий набор инструментов для создания игрового контента. Это включает редактор уровней, редактор материалов, редактор анимаций и другие инструменты для создания и редактирования игровых объектов и сцен;

- мультиплатформенная поддержка: *CryEngine* обладает возможностью разработки игр для различных платформ, включая *PC*, консоли и мобильные устройства. Это позволяет разработчикам достичь широкой аудитории и выпустить игру на разных платформах;

- встроенная поддержка виртуальной реальности: *CryEngine* имеет встроенную поддержку виртуальной реальности (*VR*), что позволяет разработчикам создавать игры, специально адаптированные для *VR*-платформ;

- комплексное решение: *CryEngine* предоставляет полноценное решение для разработки игр, включая графику, физику, звук, искусственный интеллект, анимацию и другие аспекты игрового процесса. Это упрощает и ускоряет процесс разработки и позволяет разработчикам сосредоточиться на создании уникального геймплея и контента.

В таблице 1.1 можно увидеть сравнение всех вышеописанных платформ для разработки игр.

Таблица 1.1 – Сравнение игровых движков

Редактор	Достоинства	Недостатки
<i>Unity</i>	Кроссплатформенность, большое сообщество и экосистема, большое количество обучающего материала, интуитивный интерфейс	Ресурсоемкость в работе, большой размер игровых приложений, сложность освоения для начинающих

Редактор	Достоинства	Недостатки
<i>Unreal Engine</i>	Мощный графический движок, весь необходимый функционал для разработки AAA-игр, кроссплатформенность	Высокие системные требования, сложность изучения инструментария, большой размер игровых приложений, возможность низкой производительности на мобильных устройствах
<i>Godot Engine</i>	Бесплатность и открытый исходный код, легковесность и быстрота работы, интуитивный визуальный редактор	Сложность разработки для ряда платформ, малочисленное сообщество, недостаток документации
<i>CryEngine</i>	Продвинутые графические возможности работы с графикой, кроссплатформенная поддержка, обширный инструментарий	Требуется покупки лицензии, сложность в освоении, высокие системные требования игровых приложений

1.5 Выбор графического редактора

Графические редакторы, доступные сегодня, предоставляют широкий спектр возможностей и функциональности, которые помогают разработчикам создавать качественные игры. Они обладают различными инструментами для

создания и редактирования графических элементов, включая текстуры, спрайты, модели и другие ресурсы игры.

Независимо от того, нужно ли создать реалистичную 3D-графику, пиксельную графику или векторные изображения, существуют редакторы, специально разработанные для этих целей. Таким образом, разработчики могут выбрать наиболее подходящий графический редактор в соответствии с требованиями своего проекта и своими навыками.

1.5.1 Aseprite

Aseprite – это специализированный пиксельный редактор, который широко используется для создания и редактирования пиксельной графики и спрайтов. Он предлагает множество инструментов и функций, специально разработанных для работы с пиксельной графикой.

Aseprite имеет ряд особенностей:

- пиксельный редактор: *Aseprite* специально разработан для работы с пиксельной графикой. Он предоставляет интуитивный интерфейс и инструменты для создания и редактирования пиксельных изображений;
- анимация: *Aseprite* обладает мощными инструментами анимации, которые позволяют создавать плавные и детализированные анимации. Аниматоры могут создавать и редактировать кадры, настраивать скорость воспроизведения и добавлять эффекты;
- импорт и экспорт спрайтов: *Aseprite* поддерживает различные форматы файлов для импорта и экспорта спрайтов, включая *GIF*, *PNG*, *JSON* и другие. Это упрощает работу с другими инструментами и игровыми движками;
- удобный пользовательский интерфейс: *Aseprite* предлагает простой и интуитивно понятный интерфейс, который облегчает работу с инструментами и настройками. Художники могут настраивать раскладку, горячие клавиши и интерфейс в соответствии со своими предпочтениями;
- кисти и эффекты: *Aseprite* предлагает различные кисти для рисования и создания эффектов, таких как размытие, освещение и тени. Также есть

возможность создавать собственные кисти и эффекты, чтобы придать уникальный стиль своим работам.

1.5.2 Adobe Photoshop

Adobe Photoshop является одним из самых популярных и мощных графических редакторов, который широко используется в различных областях, включая разработку игр.

Adobe Photoshop имеет ряд особенностей:

- редактирование изображений: *Photoshop* предоставляет широкий набор инструментов и возможностей для редактирования и обработки фотографий и других изображений. Художники могут корректировать цвета, ретушировать дефекты, улучшать качество изображения и многое другое;

- работа с слоями: *Photoshop* поддерживает работу с слоями, что позволяет вам создавать сложные композиции из различных элементов. Пользователь может настраивать прозрачность слоев, применять эффекты, маскировать части изображения и т.д.;

- текст и дизайн: *Photoshop* предлагает богатые возможности для работы с текстом, включая выбор шрифтов, настройку размера, цвета и эффектов. Дизайнеры также могут создавать дизайны, логотипы, иллюстрации и другие элементы графики;

- работа с 3D: *Photoshop* имеет инструменты для работы с 3D-графикой, позволяющие создавать, редактировать и текстурировать 3D-модели. Художники могут применять освещение, материалы и другие эффекты для создания реалистичных 3D-изображений;

- анимация: *Photoshop* предлагает функции для создания простых анимаций и GIF-изображений. Аниматоры могут создавать последовательности кадров, настраивать временные интервалы, добавлять эффекты перехода и многое другое.

1.5.3 Pro Motion NG

Pro Motion NG (также известный как *Pro Motion*) – это графический редактор, специально разработанный для создания пиксельной графики и анимации.

Pro Motion NG имеет ряд особенностей:

- пиксельное рисование: *Pro Motion NG* предоставляет широкий набор инструментов и функций для рисования пиксельной графики. Художники могут создавать и редактировать изображения с высокой степенью детализации и точности;

- анимация: редактор включает инструменты для создания анимации, позволяя вам создавать плавные и детализированные анимированные спрайты. Аниматоры могут настраивать кадры, временные интервалы, петли и другие параметры анимации;

- тайловая карта: *Pro Motion NG* поддерживает создание тайловых карт, которые используются в играх для создания фоновых изображений и уровней. Художники могут легко создавать и редактировать тайлы, а затем использовать их для построения игровых миров;

- импорт и экспорт: программа поддерживает различные форматы файлов для импорта и экспорта изображений, включая *PNG*, *GIF*, *BMP* и другие. Это позволяет легко интегрировать созданные графические элементы в проект;

- режимы редактирования: *Pro Motion NG* предлагает несколько режимов редактирования, включая слой, кадр и анимационный режимы. Это обеспечивает удобную и гибкую среду для работы с различными аспектами графики и анимации.

В таблице 1.2 можно увидеть сравнение всех вышеописанных графических редакторов.

Таблица 1.2 – Сравнение графических редакторов

Редактор	Достоинства	Недостатки
<i>Aseprite</i>	Специализируется на работе с пиксельной графикой, удобный и интуитивно понятный интерфейс, встроенный редактор для создания анимаций, поддержка сообщества, удобный импорт палитр	Ограниченный набор инструментов в сравнении с другими редакторами, отсутствие поддержки векторной графики, платная лицензия
<i>Adobe Photoshop</i>	Мощные инструменты для обработки и редактирования изображений, профессиональные инструменты для дизайна, широкий спектр поддерживаемых файлов	Высокая стоимость лицензии, ориентированность на растровую графику, высокие требования к ресурсам
<i>Pro Motion NG</i>	Специализируется на работе с пиксельной графикой, удобная настройка интерфейса, мощные функции для создания анимаций	Отсутствие специализированных функций для создания высококачественных изображений, отсутствие поддержки векторной графики, не подходит для проектов с высоким требованием к производительности

1.6 Выбор языка программирования

Для создания мобильных игр существует множество языков программирования, каждый из которых имеет свои особенности и применения. Выбор языка программирования зависит от различных факторов, таких как тип игры, платформа, целевая аудитория, опыт разработчика и предпочтения команды разработки.

1.6.1 C#

C# является рекомендуемым языком программирования для работы с *Unity*. Он предоставляет широкий набор инструментов и функций, специально разработанных для создания игр. C# используется для разработки игровой логики, управления объектами, обработки пользовательского ввода и многое другое. Он также позволяет разрабатывать сценарии (скрипты), которые могут быть присоединены к игровым объектам для определения их поведения.

C# имеет ряд преимуществ:

- поддержка *Unity*: *Unity* является одним из наиболее популярных игровых движков, который предоставляет интеграцию с C#. Разработчик может использовать C# для создания игровой логики, управления объектами, обработки пользовательского ввода, работы с анимациями и других аспектов разработки игры в *Unity*;
- широкая база знаний: C# имеет активное сообщество разработчиков и обширную базу ресурсов, включая документацию, учебники, форумы и онлайн-курсы. Это облегчает изучение и получение поддержки при разработке мобильных игр на C#;
- возможности: C# является объектно-ориентированным языком программирования с широким спектром функциональных возможностей. Он предоставляет средства для управления памятью, обработки событий, работы с графикой и аудио, а также разработки многопоточных приложений. Это

позволяет разработчикам создавать сложные и высокопроизводительные мобильные игры.

C# имеет ряд недостатков:

- ограничение платформы: C# сильно связан с игровым движком *Unity*, что может ограничить выбор платформы для вашей игры. Если вам нужно создать игру для платформы, не поддерживаемой *Unity*, вам может потребоваться использовать другой язык программирования или игровой движок;

- изначальная конфигурация: при использовании C# с *Unity* вам может потребоваться некоторое время на начальную настройку и настройку окружения разработки. Вам также потребуется установить *Unity* и настроить его для работы с C#;

- потребление ресурсов: Игры, разработанные на C# с использованием *Unity*, могут потреблять больше системных ресурсов в сравнении с некоторыми другими языками программирования. Это может оказывать влияние на производительность и потребление энергии мобильных устройств.

1.6.2 C++

C++ – это высокоуровневый язык программирования, широко применяемый в разработке игр, операционных систем, мобильных и других приложений, где требуется высокая производительность и низкоуровневый доступ к ресурсам компьютера.

C++ имеет ряд преимуществ:

- производительность: C++ предлагает высокую производительность и эффективное использование ресурсов компьютера. Он позволяет непосредственно управлять памятью и оптимизировать выполнение кода, что делает его особенно подходящим для разработки игр и других приложений, где важна скорость работы;

- возможности: C++ обладает широким спектром возможностей, включая поддержку объектно-ориентированного программирования (ООП),

шаблоны, многопоточность, низкоуровневую работу с памятью и другие функции. Это позволяет разработчикам создавать сложные и масштабируемые системы с высокой степенью контроля;

- **портатбельность:** C++ является кроссплатформенным языком программирования, что означает, что код, написанный на C++, может быть компилирован и работать на различных операционных системах и архитектурах. Это упрощает разработку приложений, которые могут быть запущены на разных платформах без необходимости переписывания большей части кода.

C++ имеет ряд недостатков:

- **сложность:** C++ является достаточно сложным языком программирования, который требует хорошего понимания основных концепций и языковых возможностей. Его функции и гибкость могут привести к более сложному коду и большому количеству ошибок, особенно для новичков;

- **управление памятью:** в C++ разработчики должны самостоятельно управлять памятью, выделять и освобождать ресурсы, что может привести к ошибкам, таким как утечки памяти или использование неверных указателей. Некорректное использование памяти может привести к нестабильности и ошибкам в работе программы;

- **большой объем кода:** из-за своей близости к низкоуровневому программированию и поддержке мощных функций, код на C++ может быть более громоздким и объемным по сравнению с некоторыми другими языками программирования. Это может затруднить разработку и поддержку проекта, особенно для небольших команд или начинающих программистов.

1.6.3 GDScript

GDScript – это динамический язык программирования, специально разработанный для использования в игровом движке *Godot*.

GDScript имеет ряд преимуществ:

- простота использования: *GDScript* был разработан с упором на простоту и понятность. Он имеет простой синтаксис и легко читается, что делает его доступным для начинающих разработчиков и тех, кто не имеет опыта в программировании;

- интеграция с *Godot*: *GDScript* является официальным языком программирования для движка *Godot*. Он имеет глубокую интеграцию с функциями и системами *Godot*, что облегчает разработку игр и взаимодействие с игровым движком;

- эффективность работы: *GDScript* является интерпретируемым языком, что означает, что он обеспечивает быструю разработку и мгновенное тестирование изменений в коде без необходимости компиляции. Это ускоряет процесс итеративной разработки игр.

GDScript имеет ряд недостатков:

- отсутствие широкой поддержки: *GDScript* является специфичным языком, разработанным для использования только в движке *Godot*. Это означает, что он не настолько распространен и популярен, как некоторые другие языки программирования. Меньшее сообщество и отсутствие обширной документации могут ограничить доступность ресурсов и поддержку;

- ограниченная интеграция с другими языками: в отличие от некоторых других игровых движков, *Godot* и *GDScript* имеют ограниченную поддержку для других языков программирования. Это может быть проблематично, если вам требуется использовать сторонние библиотеки или инструменты, написанные на других языках;

- ограниченные возможности оптимизации: *GDScript* является интерпретируемым языком, что может сказываться на его производительности в сравнении с некоторыми компилируемыми языками. В некоторых случаях это может быть проблемой для игр с высокими требованиями к производительности.

1.7 Выбор редактора кода

1.7.1 Microsoft Visual Studio

Visual Studio (полное название - *Microsoft Visual Studio*) – это интегрированная среда разработки (IDE), предоставляемая компанией *Microsoft*. Она широко используется разработчиками для создания различных типов приложений, включая игры.

Visual Studio имеет ряд преимуществ:

- гибкий инструментарий: *Visual Studio* предоставляет широкий набор инструментов и функций для разработки приложений. Включает в себя редактор кода с подсветкой синтаксиса, автодополнением и интегрированной отладкой, а также инструменты для версионирования кода, управления проектами и развертывания приложений;

- широкая поддержка языков программирования: *Visual Studio* поддерживает несколько языков программирования, включая C++, C#, *Visual Basic*, F#, JavaScript, Python и многие другие. Это позволяет разработчикам выбирать наиболее подходящий язык для своего проекта;

- интеграция с другими инструментами и сервисами: *Visual Studio* интегрируется с различными сервисами и платформами, такими как Azure, GitHub, Docker и другими. Это облегчает работу с различными инструментами разработки, системами контроля версий и облачными сервисами;

- расширяемость: *Visual Studio* имеет мощную систему плагинов, которая позволяет расширять его функциональность и интегрировать дополнительные инструменты и ресурсы. Разработчики могут создавать свои собственные расширения или использовать готовые плагины, чтобы улучшить процесс разработки и повысить продуктивность.

Visual Studio имеет ряд недостатков:

- тяжеловесность: *Visual Studio* является мощной интегрированной средой разработки, и это может приводить к высокому потреблению ресурсов

компьютера. Он может быть медленным и требовательным к памяти, особенно при работе с большими проектами, что может сказываться на производительности;

- сложность и избыточность: возможности *Visual Studio* могут быть избыточными для простых проектов или новичков в программировании. Интерфейс может быть перегружен функциями, что может создавать путаницу или трудности в освоении.

1.7.2 *JetBrains IntelliJ IDEA*

JetBrains IntelliJ IDEA – это интегрированная среда разработки (*IDE*), разработанная компанией *JetBrains*, которая предоставляет широкий набор инструментов и функций для разработки приложений на различных языках программирования, включая *Java*, *Kotlin*, *JavaScript*, *Python* и другие.

JetBrains IntelliJ IDEA имеет ряд преимуществ:

- умный редактор кода: *IntelliJ IDEA* предлагает многофункциональный редактор кода с подсветкой синтаксиса, автодополнением, быстрым навигационным поиском и интегрированным отображением документации. Он также обладает возможностями статического анализа кода, автоматическими исправлениями и рефакторингом, что помогает улучшить производительность и качество кода;

- поддержка различных языков и фреймворков: *IntelliJ IDEA* обеспечивает широкую поддержку различных языков программирования, фреймворков и технологий. Он предлагает инструменты разработки для *Java*, *Kotlin*, *JavaScript*, *HTML/CSS*, *SQL* и многих других. Кроме того, *IntelliJ IDEA* предоставляет интеграцию с популярными фреймворками, такими как *Spring*, *Hibernate*, *Angular*, *React* и другими;

- мощные инструменты для разработки: *IntelliJ IDEA* предлагает набор мощных инструментов, таких как отладчик, система контроля версий, интегрированный сборщик проектов, автоматические тесты и профилировщик

производительности. Это позволяет разработчикам эффективно разрабатывать, отлаживать и тестировать свои приложения.

JetBrains IntelliJ IDEA имеет ряд недостатков:

- сложность и изначальный порог вхождения: *IntelliJ IDEA* является мощной и расширяемой *IDE*, что может создавать некоторую сложность для новичков. Ее обширный набор функций может потребовать времени и изучения, чтобы использовать все возможности эффективно;

- требовательность к ресурсам: *IntelliJ IDEA* является мощным инструментом, и его запуск и работа могут потреблять значительные ресурсы компьютера. Это может быть проблематично для слабых или устаревших систем, что может повлиять на производительность разработки.

1.8 Выбор инструментов разработки

Графические интерфейсы представлены на рисунках 1.6, 1.7, 1.8.

1.8.1 Платформа разработки

В качестве платформы разработки выбрана *Unity*, которая была рассмотрена ранее с указанием ее особенностей, преимуществ и недостатков.

Графический интерфейс *Unity* представлен на рисунке 1.6.

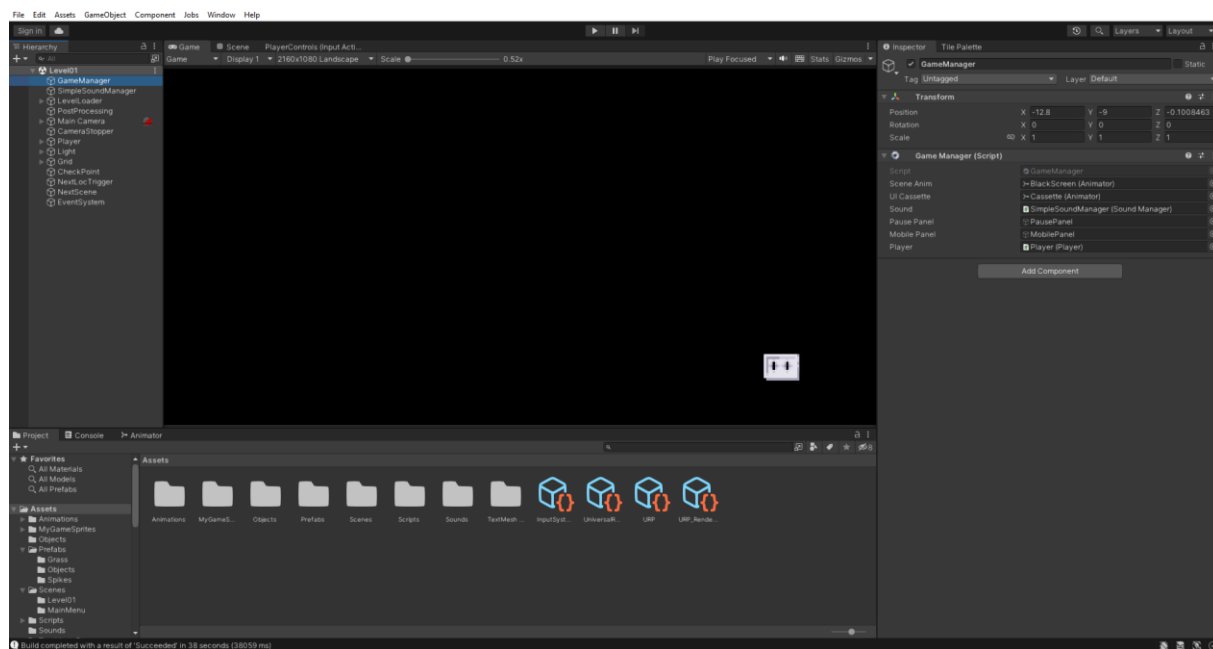


Рисунок 1.6 – Интерфейс *Unity*

Интерфейс программы разделен на несколько блоков. В нижнем левом углу находится блок «*Project*», который предназначен для работы с ресурсами проекта. Слева сверху находится окно «*Hierarchy*», которое отображает все элементы, используемые в выбранной при работе сцене. В центре расположено окно «*Game*», где отображается то, что пользователь увидит при запуске выбранной сцены. Рядом с ним находится блок «*Scene*», который позволяет работать с объектами и графическими элементами игры. В данном окне происходит основная часть работы над приложением. В правой части находится блок «*Inspector*», который отображает информацию о выбранном объекте и его свойствах. Правее расположен блок «*Input Actions*», где происходит настройка управления персонажем. Интерфейс программы предлагает гибкую настройку расположения блоков, позволяя пользователям легко изменять их положение для обеспечения более удобного рабочего процесса.

1.8.2 Графический редактор

В качестве графического редактора был выбран *Aseprite*, который ранее был рассмотрен с указанием особенностей, преимуществ и недостатков. Графический интерфейс *Aseprite* представлен на рисунке 1.7.

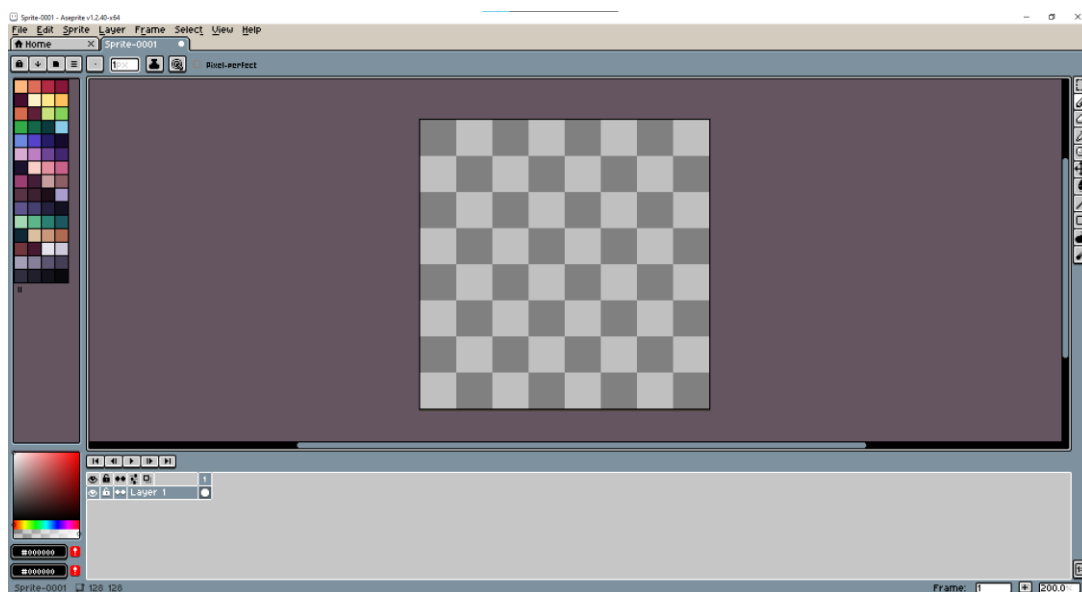


Рисунок 1.7 – Интерфейс *Aseprite*

Интерфейс программы состоит из нескольких блоков. По левую сторону расположен блок для настройки палитры с возможностью выбирать, загружать или составлять ее самостоятельно. Правее расположен блок рабочей области, где происходит процесс создания изображения. Снизу находится блок работы со слоями и анимациями. Справа находится блок инструментов. Верхняя панель предназначена для настройки выбранного инструмента. Пользователям предоставляется возможность гибко настраивать расположение разделов в интерфейсе программы, что позволяет легко адаптировать его под свои потребности и обеспечить максимально комфортное рабочее окружение.

1.8.3 Редактор кода

В качестве редактора кода был выбран *Microsoft Visual Studio 2019*, который был рассмотрен ранее с указанием преимуществ и недостатков. Графический интерфейс *Visual Studio* представлен на рисунке 1.8.

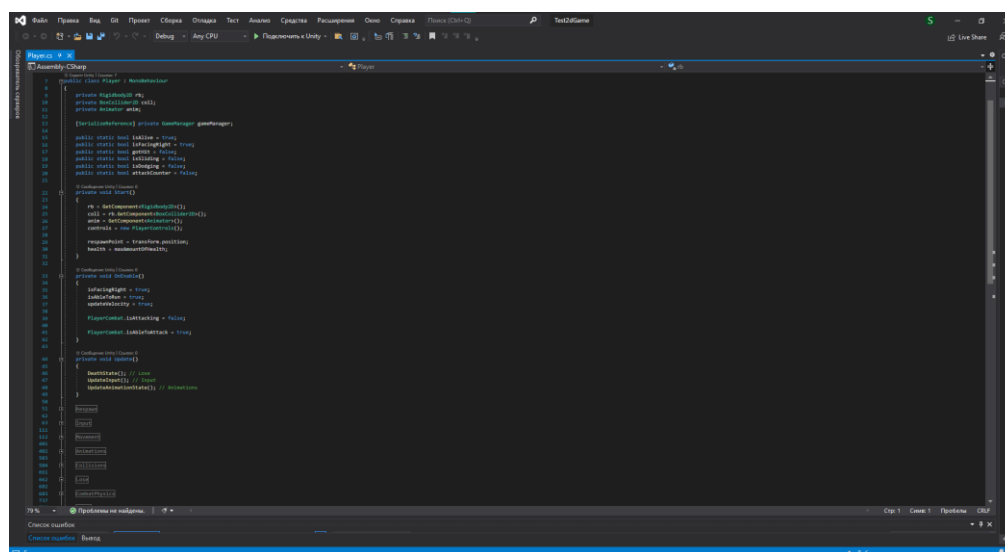


Рисунок 1.8 – Интерфейс *Visual Studio*

1.8.3 Язык программирования

В качестве языка программирования был выбран язык программирования *C#*, который был рассмотрен ранее с указанием его особенностей, преимуществ и недостатков. Выбор *C#* в качестве языка

программирования для работы с *Unity* обеспечивает удобство, эффективность и широкие возможности при создании игр и приложений.

1.9 Архитектура приложения

Схематичные изображения паттернов проектирования представлены на рисунках 1.9, 1.10.

Паттерн проектирования – это повторяемый архитектурный шаблон, который предлагает решение для типичных проблем, возникающих при проектировании программного обеспечения. Паттерны проектирования помогают структурировать и организовать код, улучшить его гибкость, повторное использование и поддерживаемость.

1.9.1 *Component-based architecture*

В данном приложении применяется паттерн проектирования, который помогает организовать код и управлять состоянием приложения. *Component-based architecture* предлагает эффективные архитектурные решения и способы взаимодействия между компонентами. Схематичное изображение паттерна *Component-based architecture* представлено на рисунке 1.9.

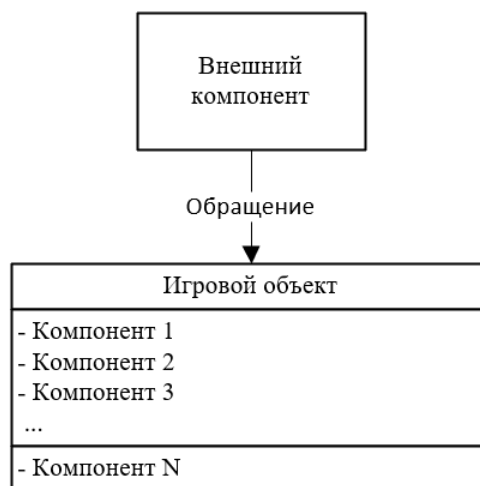


Рисунок 1.9 – Схематичное изображение паттерна *Component-based architecture*

"Игровой объект" представляет отдельный объект в игре, такой как персонаж, препятствие или предмет.

Компоненты (*Component*) отображаются внутри игрового объекта и представляют его функциональность и поведение. Они могут быть представлены в виде списков или отдельных блоков, в зависимости от их количества.

«- Компонент 1» и «- Компонент 2» обозначают конкретные компоненты, которые присутствуют в игровом объекте.

«...», «- Компонент N» указывают на то, что в игровом объекте может быть любое количество компонентов.

В контексте данного игрового приложения внешним компонентом служат скрипты или классы.

1.9.2 *Model-View-Controller*

Также в данном игровом приложении используются принципы паттерна *MVC* (*Model-View-Controller*). В рамках *MVC* различные компоненты приложения разделены на три основных блока:

- модель (*Model*): отвечает за представление данных и логику приложения. Модель содержит информацию, методы для работы с этой информацией и логику приложения, которая оперирует этими данными. В игровом контексте, модель представляет игровые объекты, их свойства, поведение и системы управления;

- представление (*View*): отображает данные модели пользователю и отвечает за визуальное представление интерфейса. В *Unity* представление может быть реализовано с помощью компонентов *Unity UI* или собственных пользовательских компонентов. Представление отображает данные модели и реагирует на пользовательский ввод;

- контроллер (*Controller*): обрабатывает пользовательский ввод и взаимодействует с моделью и представлением. Контроллер получает события от пользовательского ввода (например, нажатия клавиш или жесты на экране)

и передает соответствующие команды модели или представлению для выполнения требуемых действий.

Схематичное изображение паттерна *Model-View-Controller* представлено на рисунке 1.10.

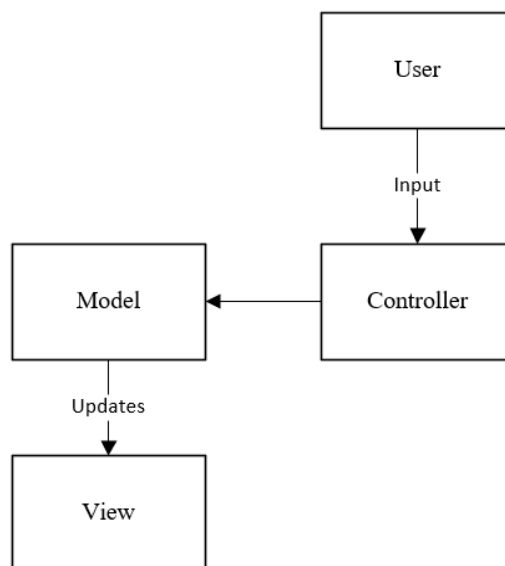


Рисунок 1.10 – Схематичное изображение паттерна *Model-View-Controller*

1.10 Разработка игрового приложения

При разработке игрового приложения можно выделить несколько этапов:

- концепция: этап, на котором определяются основные идеи, направленность игры, игровые механики и другие ключевые аспекты;
- графика и дизайн: создание графических ресурсов, таких как персонажи, окружение, объекты, интерфейс и спецэффекты. Включает в себя рисование, моделирование, анимацию и создание текстур;
- программирование: написание кода для реализации игровой логики, управления объектами, взаимодействия с пользователем, анимаций и других функциональных аспектов игры. Включает использование языков программирования, таких как *C#*, *C++*, *GDScript* и других;

- звук и музыка: создание звуковых эффектов, музыки и аудиоинтерфейсов для обогащения игрового опыта;

- тестирование и отладка: проверка функциональности игры, выявление и исправление ошибок (багов), оптимизация производительности и тестирование на различных платформах и устройствах.

1.10.1 Концепция

В ходе изучения аналогов игрового приложения были обнаружены неотъемлемые элементы, которые делают проекты в жанре «платформер» по-настоящему увлекательными.

Если рассматривать геймплейную составляющую проектов, можно выделить схожие механики вертикального передвижения. Платформеры предоставляют персонажу возможность перемещаться по различным грамотно спроектированным игровым локациям или поверхностям, преодолевать интересные препятствия по мере продвижения.

Как правило простых механик платформинга для достижения этих целей недостаточно и разработчики используют разнообразные инструменты передвижения помимо прыжка. Это может включать: двойные прыжки, взаимодействия со стенами (прыжки от стен, скольжение, возможность взбираться на преграды), рывок (*Dash*), возможность толкать и тянуть игровые предметы для решения задач уровня или пространственных головоломок и т.д. Всё это невозможно без слаженной работы геймдизайнера, дизайнера уровней и программистов.

Можно выделить ряд требований к механикам передвижения при разработке платформера:

- *Jump Buffering*: установка таймера для регистрации нажатия на кнопку прыжка за мгновение до касания поверхности. Такая обработка нажатия позволяет избежать неприятного ощущения от отсутствия его регистрации при раннем вводе команды;

- *Coyote Time*: установка таймера для регистрации нажатия на кнопку прыжка, когда игровой персонаж уже сошел с поверхности, но еще не начал падение. Такая обработка ввода помогает игроку, который опоздал с нажатием на кнопку;

- *Edge Detection*: для избегания нежелательной остановки персонажа при контакте с углами поверхностей обычно используют систему рейкастинга (*Raycasting*). Такой подход позволяет регистрировать столкновения коллайдера персонажа. При разработке игрового приложения был использован метод *OverlapCircle*, который имеет схожий функционал, но более точно считывает столкновения при работе с полигональными поверхностями;

- *Jump Height*: чем больше длительность нажатия на кнопку, тем выше прыжок. Повышает ощущение контроля над персонажем;

- *Jump Hang Time* и *Faster Fall*: при удержании кнопки прыжка в воздухе увеличивается время до начала падения, при котором персонаж ускоряется. Такой подход дает игроку больше контроля над персонажем и создает ощущение весомости главного героя, что в свою очередь расширяет и улучшает игровой опыт;

- *Acceleration* и *Deceleration*: плавное ускорение и остановка персонажа, этот подход повышает уровень контроля игрока над персонажем и придает главному герою инерцию, что в конечном итоге улучшает и расширяет игровой опыт.

Можно выделить ряд требований к дизайну уровней при разработке платформера:

- играбельность: уровень должен быть интересным и увлекательным для игрока. Он должен предоставлять достаточное количество вызовов и задач, чтобы игрок мог проявить свои навыки и получить удовлетворение от преодоления препятствий;

- прогрессия: уровень должен предлагать постепенное увеличение сложности и интенсивности, чтобы игрок постоянно ощущал прогресс и достижения. Это может включать в себя введение новых механик, уровней сложности или врагов по мере продвижения игрока;

- баланс: уровень должен быть хорошо сбалансирован в терминах сложности и вознаграждения. Необходимо учесть уровень навыков и опыта игрока, чтобы предложить достаточно вызовов, но при этом не слишком обременить его.

- тематика и дизайн: уровень должен соответствовать общей тематике и стилю игры. Визуальное оформление, аудиоэффекты и музыка должны создавать атмосферу и поддерживать настроение игрока.

- возможности и вариативность: уровень должен предлагать разнообразные пути, альтернативные решения и возможности для экспериментов. Это позволит игроку исследовать и находить собственный стиль игры, повышая его вовлеченность.

В игровом приложении были учтены все вышеперечисленные требования. В качестве дополнительных инструментов перемещения был реализован прыжок от стены, скольжение, рывок (в качестве которого выступает удар персонажа).

Логика *Edge Detection* представлена на рисунке 1.11.

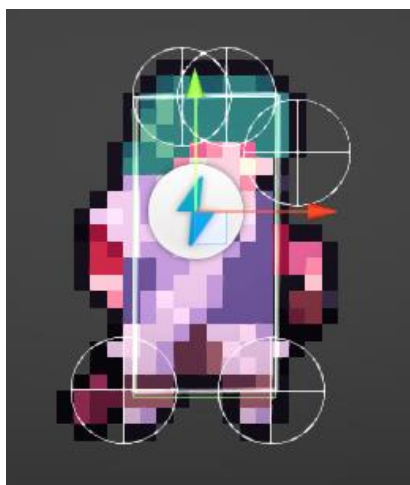


Рисунок 1.11– Логика *Edge Detection*

Две верхние окружности позволяют персонажу огибать края платформы при прыжке. Нижние помогают игроку запрыгивать на препятствия, если он ошибся с высотой или дальностью прыжка. Верхняя правая окружность регистрирует касания для прыжка от стены.

Обучающий уровень представлен на рисунке 1.12.

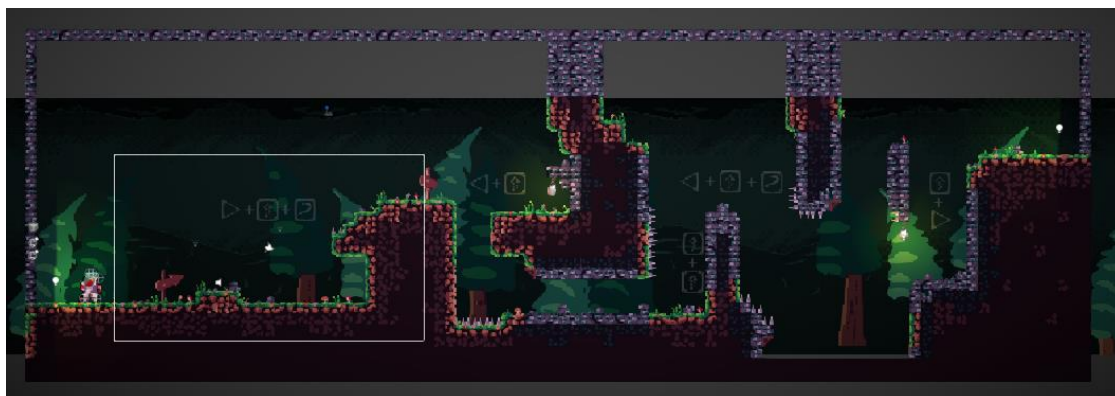


Рисунок 1.12– Обучающий уровень

Уровень соответствует всем вышеперечисленным требованиям, также имеет интуитивно понятные подсказки, что позволяет игроку быстро освоить базовые механики игры.

1.10.2 Графика и дизайн

В игре использован пиксель-арт и спрайтовые покадровые анимации, чтобы придать ей особый визуальный стиль и атмосферу. Использование пиксельной графики в игровом приложении может быть хорошей идеей по нескольким причинам:

- простота и доступность: создание пиксельной графики может быть относительно проще и доступнее для многих разработчиков, особенно для небольших команд или инди-разработчиков. Она не требует высоких ресурсов и специализированного оборудования, как другие типы графики.

- малый размер файлов: пиксельная графика обычно имеет меньший размер файлов по сравнению с более сложными форматами графики, такими как векторная графика или трехмерные модели. Это может быть полезно при разработке мобильных игр или игр для онлайн-платформ, где размер файла

игры имеет значение для быстрой загрузки и оптимизации производительности.

- возможность создания ретро-стилизированных игр: пиксельная графика позволяет создавать игры в ретро-стиле, которые могут быть популярными среди определенной аудитории. Это может быть особенно привлекательно для любителей ретро-игр или для тех, кто хочет воссоздать атмосферу классических игр.

- гибкость в анимации и дизайне: пиксельная графика может быть легко анимирована и изменена в соответствии с потребностями игры. Она предлагает большую гибкость в создании и настройке дизайна персонажей, уровней и эффектов, может быть адаптирована к различным стилям игры.

Визуал и анимации игры формируют огромную часть впечатлений от игрового процесса и играют роль в формировании опыта игрока. Качественный визуальный стиль помогает погрузить пользователя в игровой мир и создать атмосферу, которая соответствует задумке разработчиков.

Визуальные эффекты и анимации помогают игрокам лучше понять, как работают игровые механики и взаимодействия. Они могут служить визуальными подсказками, помогая игрокам ориентироваться в игровом пространстве, распознавать ключевые элементы и принимать обоснованные решения в игре. Это важно для обеспечения понятности игрового процесса, передачи информации о состоянии объектов, действиях персонажей и изменениях в игровой среде.

Всё это помогает игре выделиться среди других проектов и создать собственную идентичность. Привлекает внимание игроков, а также делает игру запоминающейся и узнаваемой в игровом сообществе.

Для игрового приложения было отрисовано более 150 кадров анимации, составлен и нарисован тайлсет (*TileSet*), все элементы интерфейса, игровые объекты, частицы и логотипы игры.

					ПГУ1.090301.22.001 ПЗ	Лист
						42
Изм.	Лист	№ докум.	Подпись	Дата		

Tileset – это графический набор изображений, который используется для создания игрового уровня или карты в играх, особенно в жанрах платформеров, *RPG* и стратегий в стиле «пиксель-арт». *Tileset* состоит из множества небольших изображений, называемых тайлами, которые представляют собой квадратные или прямоугольные фрагменты графики.

Работа над анимациями показана на рисунках 1.12, 1.13 [1].

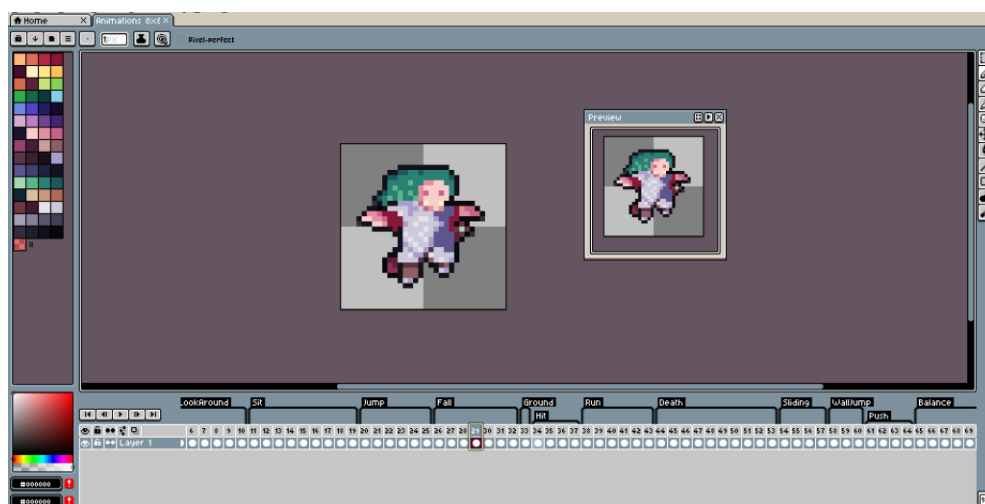


Рисунок 1.12 – Работа над анимациями



Рисунок 1.13 – Работа над анимациями

Работа над *TileSet* и элементами интерфейса показана на рисунках 1.14, 1.15.

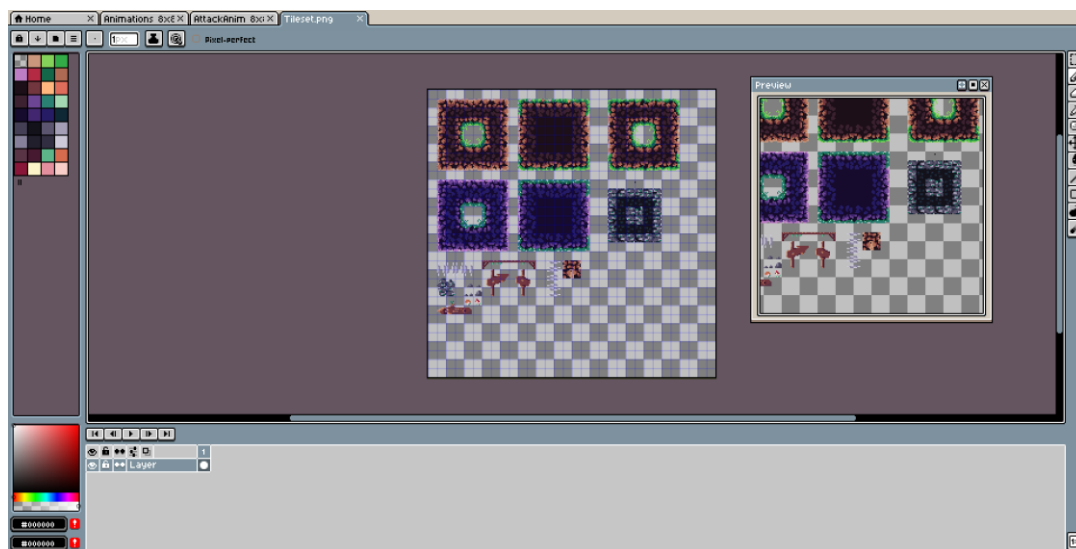


Рисунок 1.14 – Работа над *TileSet* и элементами интерфейса

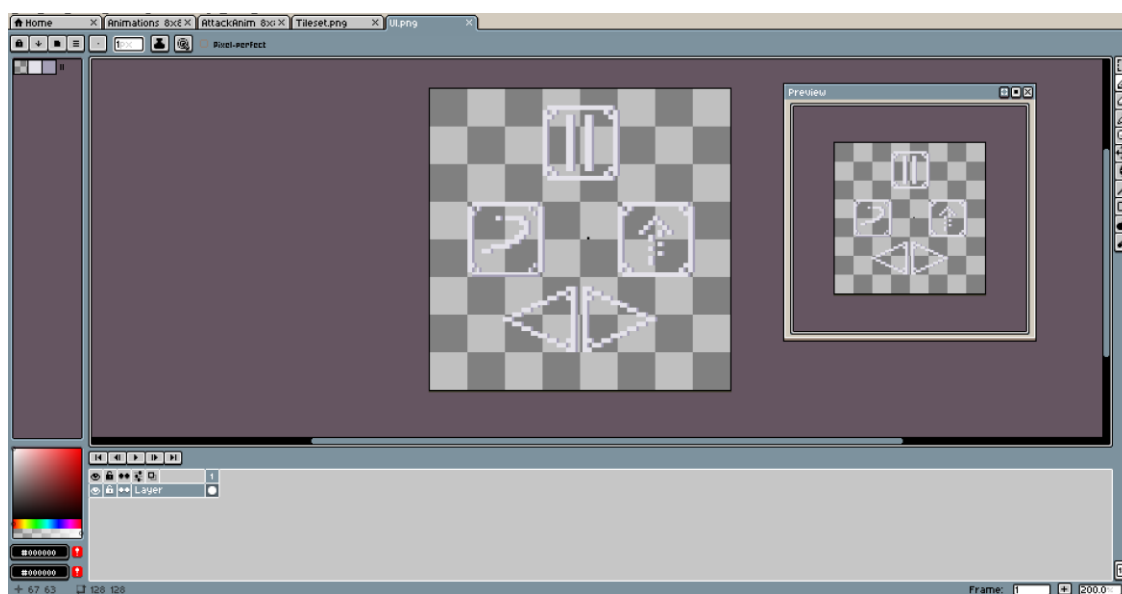


Рисунок 1.15 – Работа над *TileSet* и элементами интерфейса

Анимации были использованы в игре с помощью встроенного аниматора *Unity*. *Unity* позволяет создавать ключевую анимацию путем установки ключевых кадров и интерполяции между ними. Все анимации персонажа и окружения были синхронизированы со звуковыми эффектами. *Animation* и дерево анимаций проекта представлены на рисунках 1.16, 1.17 [2].

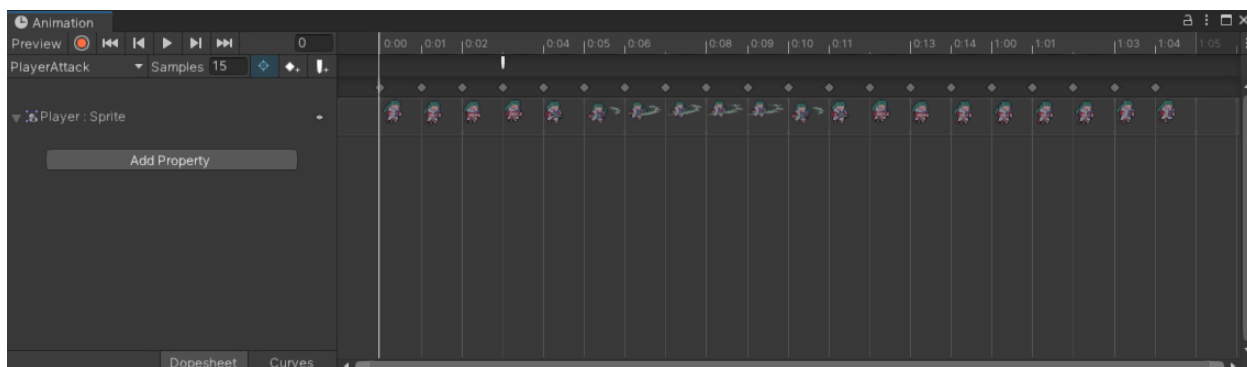


Рисунок 1.16 – *Animation* и дерево анимаций

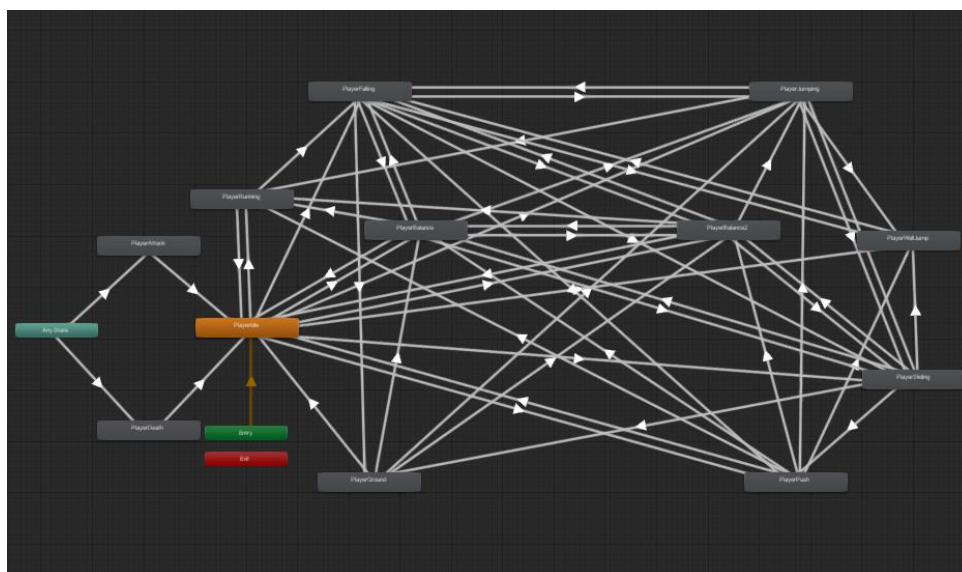


Рисунок 1.17 – *Animation* и дерево анимаций

Было уделено особое внимание подбору и настройке освещения и частиц с целью создания атмосферы спокойствия и умиротворения в игре. Инструментарий *Unity* для работы с частицами – *Particle System* представлен на рисунке 1.18.

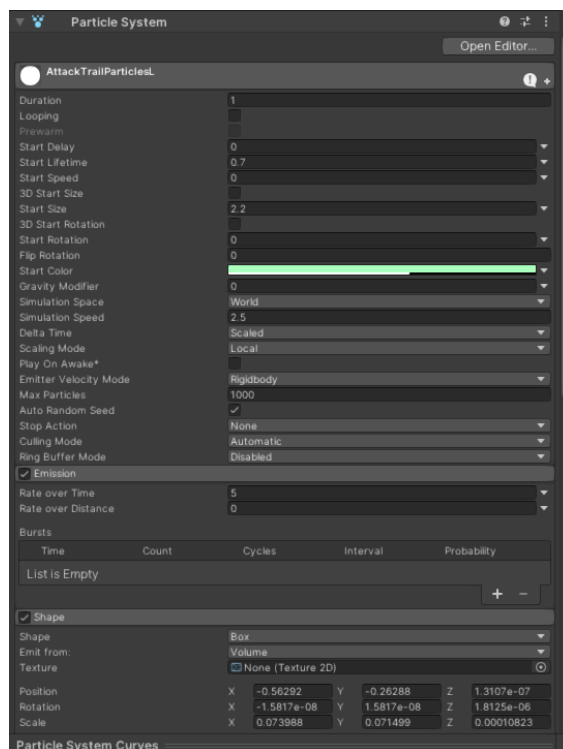


Рисунок 1.18 – *Particle System*

Для создания дополнительной динамики и глубины картинки игры задний фон бы поделен на части по слоям и использовался для создания эффекта параллакса.

Параллакс (*Parallax*) – это визуальный эффект, который создается путем движения объектов на разных плоскостях с разной скоростью. Это дает ощущение глубины и перспективы в 2D или 3D сценах. Эффект параллакса представлен на рисунке 1.19.

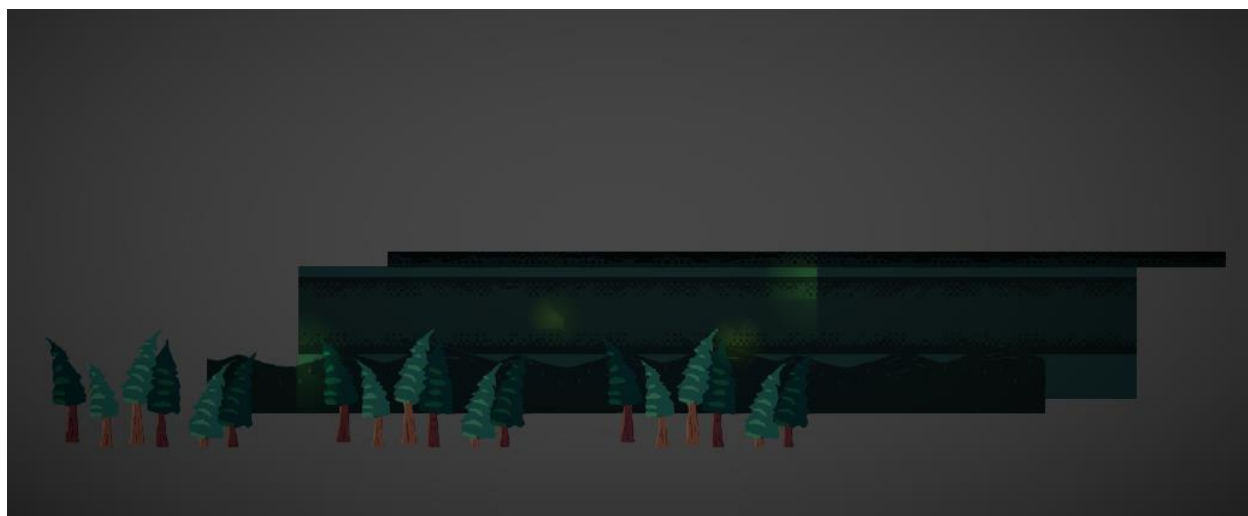


Рисунок 1.19 – Эффект параллакса

1.10.3 Программирование

Игровое приложение состоит из трех сцен (*Scene*), каждая сцена содержит в себе ряд игровых объектов такие как [3]:

- *GameManager*: служит менеджером (контроллером) сцены, устанавливает целевой фреймрейт всего приложения при запуске, работает с загрузочными анимациями через *LevelLoader*, воспроизведением звуков помощью *SoundManager*, дает возможность игроку ставить сцену на паузу и переходить на сцену главного меню через *PausePanel*, управляет сценами в зависимости от действий и состояний игрового персонажа с помощью скрипта *Player*;

- *SoundManager*: служит менеджером (контроллером) звукового сопровождения в игре содержит в себе все компоненты *Audio Source*, которые позволяют вызывать звуковые эффекты из файлов игры и гибко их настраивать, также к объекту подключен компонент *Player Animator*, для удобной синхронизации звуков и анимаций персонажа;

- *LevelLoader*: объект, который хранит в себе все объекты интерфейса: *BlackScreen* – анимация затухания экрана при загрузке, *UICassette* – анимация спрайта кассеты как индикатора загрузки сцены, *MobilePanel* – объект, который содержит все *UI* кнопки, *PausePanel* – объект, который содержит кнопки меню паузы;

- *PostProcessing*: объект, хранящий компоненты настройки постобработки;

- *MainCamera*: объект, содержащий компонент камеры и ее настройки, а также компонент *CinemachineVirtualCamera* для расширения функционала родительского объекта;

- *CameraStopper*: объект, который содержит компонент *PolygonCollider2D* который связан с *CinemachineVirtualCamera* и устанавливает ограничение на ее передвижение;

- *Player*: объект, который содержит компоненты настройки физики персонажа *RigidBody 2D*, его коллайдер *Box Collider 2D*, аниматор *PlayerAnimator*, *Shadow Caster 2D* для взаимодействия с источниками освещения, *Player Input* для управления персонажем через *New Unity Input System*. Скрипт *Player* связан с *GameManager*, объектами для регистрации коллизий, *SoundManager*, объектами частиц, и реализует логику передвижения и состояний персонажа. Скрипт *LightController* изменяет свечение персонажа в зависимости от его состояния;

- *Light*: объект, который хранит объекты с компонентом *Light 2D*. Дочерние объекты обеспечивают глобальное и локальное освещение сцен;

- *Grid*: объект, который хранит объекты тайловых карт (поверхностей *Terrain*, объектов заднего фона *BackGroundObjects*), объекты наносящие урон персонажу «шипы» *Spikes*, изображения заднего фона *BackGround*, анимированные объекты травы, цветов *Grass*;

- *LocTriggers*: объект, который хранит объекты с компонентами *Box Collider 2D*, триггеры для перехода между сценами и нанесения урона игроку в случае падения в «пропасть»;

- *Particles*: объект, который содержит объекты с компонентами *Particle System* для гибкой настройки частиц окружения;

- *ShadowCasters*: объект, который содержит объекты с компонентами *Shadow Casters 2D* для реакции поверхностей на локальное освещение.

- *EventSystem*: объект с компонентами *Event System* и *Input System UI Input Module* для обработки ввода пользователя через компонент *Player Controls*.

New Unity Input System (новая система ввода *Unity*) – это обновленная система для обработки пользовательского ввода в игровых приложениях, представленная в *Unity* начиная с версии 2019.1. Она предоставляет улучшенные возможности для работы с различными устройствами ввода, такими как клавиатура, мышь, геймпады, сенсорные экраны и другими.

1.10.3.1 Классы используемые в проекте

В *Unity* все классы наследуются от базового класса *UnityEngine.Object*. Класс *UnityEngine.Object* является фундаментальным классом в *Unity*, от которого наследуются все объекты, создаваемые в игре, включая игровые объекты (*Game Objects*), компоненты (*Components*), ресурсы (*Resources*) и другие [5].

UnityEngine.Object предоставляет базовые функциональные возможности, такие как управление жизненным циклом объекта, доступ к идентификатору объекта (*instance ID*), проверка на существование объекта и другие операции.

При создании пользовательских классов в *Unity*, таких как скрипты компонентов, эти классы также наследуются от *UnityEngine.Object* (или других специализированных классов, таких как *MonoBehaviour*, если это скрипт компонента). Это позволяет использовать множество встроенных функций *Unity* для работы с объектами, а также связывать их с игровым движком и редактором *Unity*.

Все скрипты компонентов наследуются от базового класса *UnityEngine.MonoBehaviour*. Класс *MonoBehaviour* предоставляет функциональность, необходимую для создания и управления поведением компонентов в *Unity*.

На рисунке 1.20 представлена *UML* диаграмма классов проекта [4].

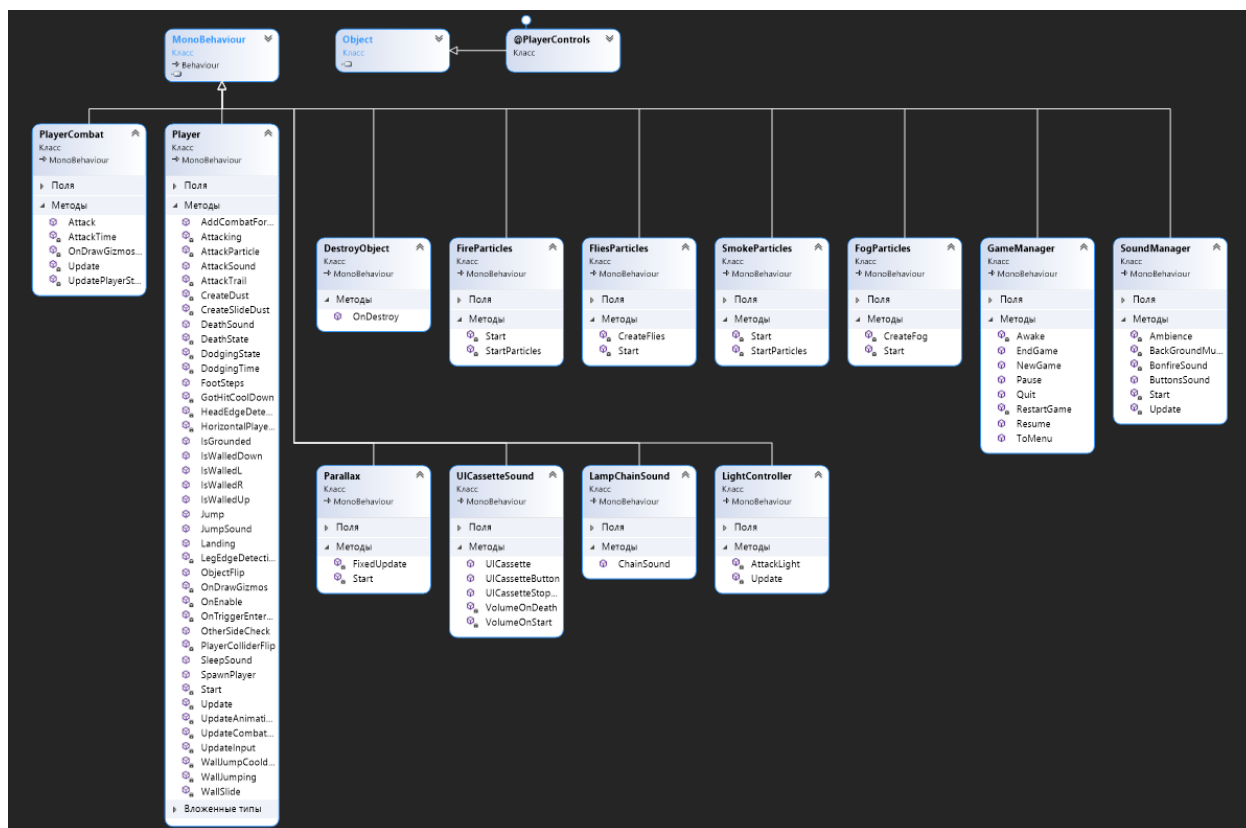


Рисунок 1.20 – UML диаграмма классов проекта

Рассмотрим классы более подробно.

1.10.3.2 *PlayerControls*

Автоматически генерируемый класс при работе с новой системой ввода *Unity*, содержит в себе все необходимые методы для регистрации нажатий, назначения клавиш. Является основой управления вводом и может быть расширен и настроен для обработки различных игровых событий и действий.

1.10.3.3 *PlayerCombat*

Скрип реализует механику атаки персонажа, в контексте игрового приложения используется для добавления силы к объекту персонажа по направлению движения.

UpdatePlayerState – осуществляет связь со скриптом *Player* и проверяет состояния персонажа.

Attack – реализация атаки, ограничивает количество атак (в контексте приложения – рывков). Использует метод *AttackTime*.

AttackTime – осуществляет выход из состояния атаки и устанавливает задержку на ее повторное использование.

1.10.3.4 *Player*

В скрипте описана вся логика передвижения, обработка столкновений, работа с физикой и состояниями персонажа.

DeathState – устанавливает состояние персонажа на проигрыш и вызывает метод *EndGame* из *GameManager*.

UpdateInput – реализует обновление передвижения персонажа во встроенном *Update* методе. Связан со всеми методами. Влияющими на передвижение.

UpdateAnimationState – получает информацию о состояниях персонажа и устанавливает необходимые значения в дереве анимаций. Служит основным инструментом связи с аниматором и позволяет удобно добавлять и редактировать поведение анимаций.

SpawnPlayer – устанавливает положение персонажа после поражения, вызывается с помощью *GameManager*.

ObjectFlip – поворачивает объект по горизонтали, вызывается с помощью *PlayerColliderFlip*.

PlayerColliderFlip – реализует поворот персонажа в зависимости от горизонтального ввода в *HorizontalPlayerMovementController*.

HorizontalPlayerMovementController – реализует логику горизонтального передвижения.

Jump – реализует логику вертикального перемещения, в том числе использует метод *WallJumping* для связи с логикой прыжка от стены.

WallJumpCooldown – используется для выхода из состояния прыжка от стены и устанавливает задержку на повторное использование механики.

WallSlide – содержит логику скольжения по стенам.

OnTriggerEnter2D – встроенный метод для обработки коллизий объекта, реализует логику получения урона и работы с *LocTriggers*.

IsWalled и *OtherSideCheck* – используются для обработки касаний с коллайдерами поверхностей.

LegEdgeDetection – осуществляет помощь с запрыгиванием на препятствия путем перемещения объекта персонажа.

HeadEdgeDetection – осуществляет помощь с огибанием препятствий над коллайдером персонажа путем смещения объекта в сторону по направлению движения.

IsGrounded – проверяет касается ли персонаж твердой поверхности.

UpdateCombatState – связан с *PlayerCombat*, получает информацию о состоянии атаки.

AddCombatForce – вызывается с помощью *PlayerCombat* и добавляет горизонтальную силу по направлению движения.

OnDrawGizmoz – встроенный метод *Unity*, позволяет отображать физические методы *OverlapCircle* в редакторе сцен.

Также скрипт *Player* имеет связь с методами *SoundManager* для удобной синхронизации звуков и анимаций, содержит ряд методов для активации эффектов частиц, привязанных к объекту персонажа.

1.10.3.5 *GameManager*

Работает с состояниями сцен, анимациями их загрузки и звуковыми эффектами. Устанавливает целевой фреймрейт во встроенном методе *Awake*.

EndGame – вызывается при поражении игрока, вызывает анимацию загрузки и метод *RestartGame*.

RestartGame – вызывает метод *SpawnPlayer* и запускает анимацию сцены.

NewGame – вызывается по кнопке из сцены меню, запускает сцену первого уровня.

Quit – запускается из сцены меню и реализует закрытие приложения.

Pause – вызывается из любой сцены кроме меню по кнопке паузы, останавливает течение времени в приложении, вызывает *PausePanel* и отключает *MobilePanel*. Выключает звук приложения через *SoundManager*.

Resume – вызывается из любой сцены кроме меню и выполняет выход из состояния *Pause* с восстановлением всех значений.

ToMenu – вызывается по кнопке в *PausePanel* и переключает текущую сцену на сцену меню.

1.10.3.6 *SoundManager*

Скрипт для работы со звуками из компонентов *Audio Source* родительского объекта *SimpleSoundManager*. Обеспечивает возможность отключения звука в игре по кнопке паузы и содержит ряд методов для активации звукового сопровождения.

1.10.3.7 *Particles* скрипты

Ряд схожих скриптов, обеспечивающих вызов эффектов частиц на сценах, включая эффекты «дыма», «светлячков», «огня» и «тумана».

1.10.3.8 *Sound* скрипты

Скрипты для активации 3D источников звука, привязанных к конкретным объектам, но всё еще контролируемых *SoundManager*.

1.10.3.9 *Parallax*

Простой скрипт, привязанный к объекту камеры. Устанавливает скорость движения слоев заднего фона в зависимости от положения камеры.

1.10.3.10 *OnDestroy*

Отдельный скрипт для удаления объектов со сцены, удобно использовать по завершению анимации через *Animation Events*.

1.10.4 Звук и музыка

В приложении использовалась музыка и звуки из бесплатных источников без авторских прав. В игре присутствуют 3 источника с настроенным позиционным звуком. Для создания атмосферы используются зацикленные звуки и музыка. Для передвижения и состояний игрового

персонажа используются звуки в ретро стилистике для поддержания стилистики приложения.

Звуковое пространство может помочь игрокам ориентироваться в игровом мире. Звуки из разных направлений или их изменение при приближении или удалении от объектов могут помочь игрокам определить расстояние и направление к объектам или событиям в игре.

Звуки могут служить обратной связью для игроков, указывая на успешное выполнение действия, получение урона, активацию способности и т.д. Такая обратная связь помогает игрокам оценивать свои действия и принимать решения в игре. Музыка может вызывать яркие эмоциональные реакции. Интенсивная в боевых сценах, спокойная музыка в мирных местах или напряженная в хоррор-играх, каждая из них усиливает воздействие на игрока.

1.10.5 Тестирование и отладка

В ходе тестирования было выявлено несколько багов, в том числе и критических, что заняло большое количество времени на их устранение. Тестирование помогло успешно настроить баланс игры, исправить недочеты в дизайне уровня, работе физики и логике передвижения. Всё это помогло привести геймплей к состоянию, которое можно назвать комфортным.

Тестирование на мобильных устройствах способствовало исправлению многих недочетов и внесению важных для игрового процесса изменений. Оно проводилось на 6 различных устройствах на платформе *Android*, благодаря чему производительность можно назвать достаточно стабильной, а приложение оптимизированным под мобильную систему.

1.11 Описание приложения

При запуске игрового приложения проигрывается вступительная заставка с логотипами игры и платформы разработки. Вступительная заставка показана на рисунке 1.21.



Рисунок 1.21 – Вступительная заставка

Далее пользователь попадает на первую сцену «*MainMenu*», она является полноценной *Unity* сценой, а не цикличным изображением или видеороликом. Данная сцена спроектирована с минимальным количеством активных объектов, чтобы не влиять на производительность приложения. Здесь пользователю предоставляется возможность начать игру, используя кнопку «*NEW GAME*», и выйти из приложения, используя кнопку «*QUIT*». Главное меню представлено на рисунке 1.22.

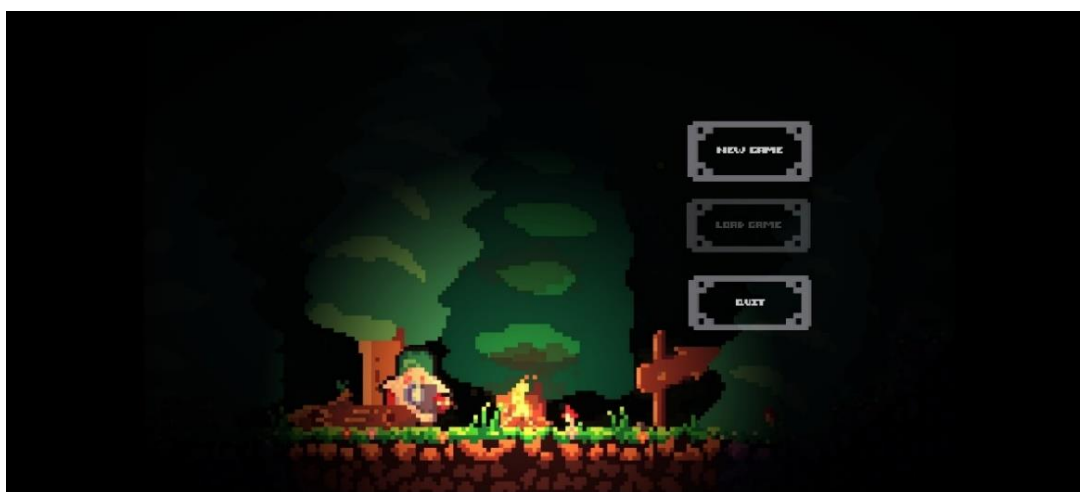


Рисунок 1.22 – Главное меню

По нажатию кнопки «*NEW GAME*» проигрывается анимация со звуковым сопровождением, которая является экраном загрузки в приложении. Экран загрузки представлен на рисунке 1.23.

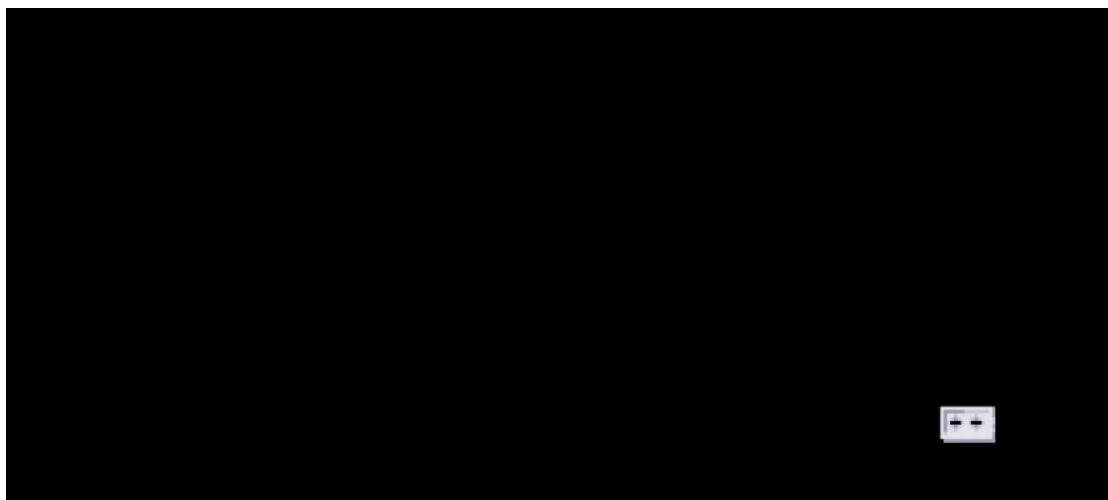


Рисунок 1.23 – Экран загрузки

Затем игрок попадает на первую сцену «*Level01*», которая служит плавным переходом от визуального оформления меню к игровому процессу. Сцена также является логическим продолжением главного меню. Звуковое сопровождение меняется на более подходящее ко времени наступления «рассвета». В первую очередь игрок замечает объект «потухшего костра» и визуальный ориентир, указывающий направление движения. В конце локации (за пределами камеры) находится светящийся объект, который также указывает, что игрок движется в правильном направлении. Первая сцена «*Level01*» изображена на рисунке 1.24.

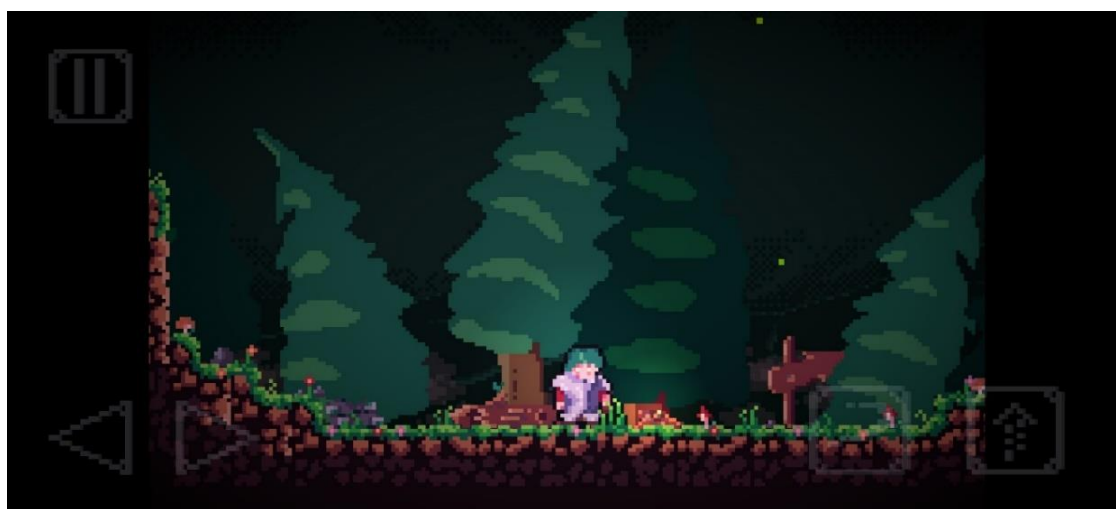


Рисунок 1.24 – Первая сцена «*Level01*»

Далее, после экрана загрузки, игрок попадает на сцену «*Level02*», которая является обучающим уровнем. На сцене расположены визуальные

подсказки, которые помогают игроку быстрее разобраться с базовыми механиками игры. Преодолев первое препятствие, игрок понимает, как использовать механику прыжка и прыжка от стены в комбинации с атакой, интерфейс позволяет выполнить это действие без особых усилий. Первая точка интереса представлена на рисунке 1.25.

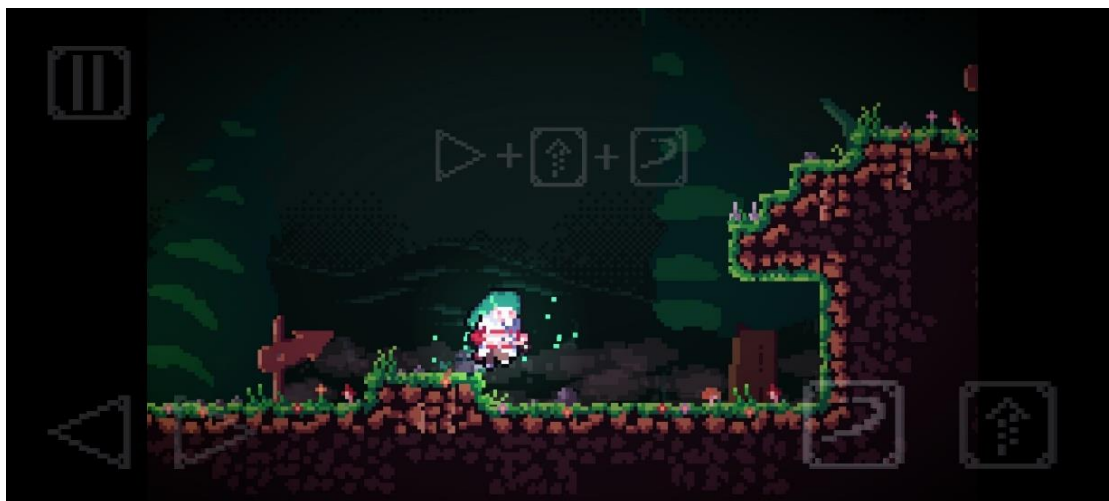


Рисунок 1.25 – Первая точка интереса

Второй точкой интереса является «сломанный указатель» и объект «фонарь», которые побуждают игрока остановиться и обратить внимание на подсказку. Здесь пользователь знакомится со второй механикой «скольжение». Вторая точка интереса представлена на рисунке 1.26.

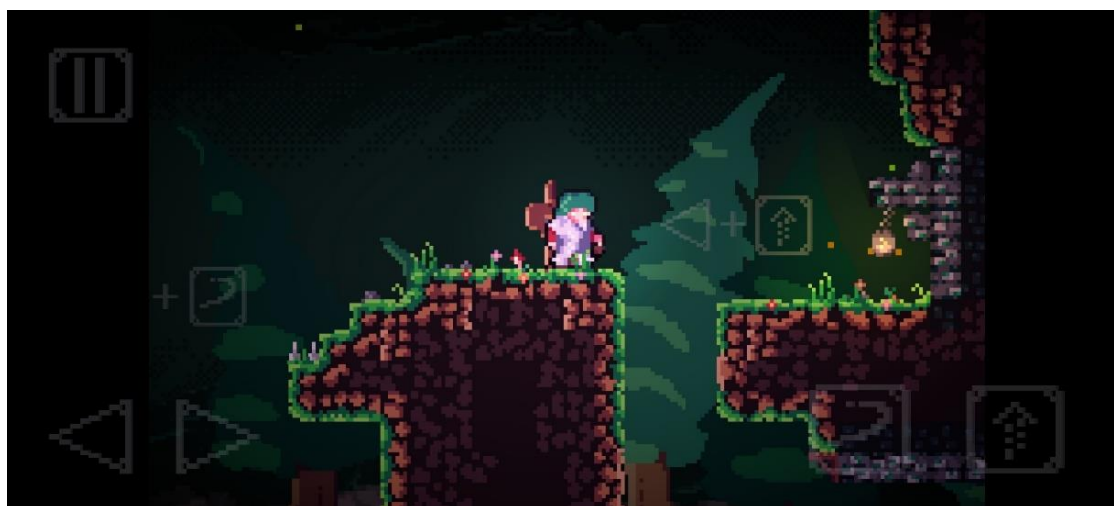


Рисунок 1.26 – Вторая точка интереса

Затем, при спуске на нижний уровень локации перед игроком возникает проблема, механика прыжка от стены не работает на данном участке. Так

происходит знакомство с механикой вариативного прыжка (прыжок от стены в комбинации с кнопкой движения дает персонажу больше горизонтальной скорости, обычный же дает вертикальную). Третья точка интереса представлена на рисунке 1.27.

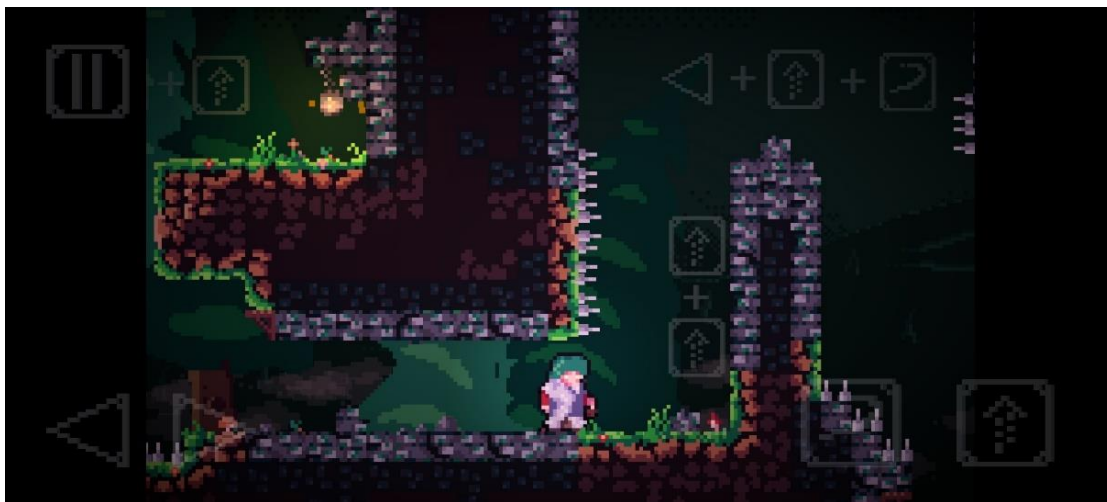


Рисунок 1.27 – Третья точка интереса

После чего перед игроком появляется «пропасть», для преодоления которой придется применить все ранее изученные механики. Четвертая точка интереса представлена на рисунках 1.28, 1.29.

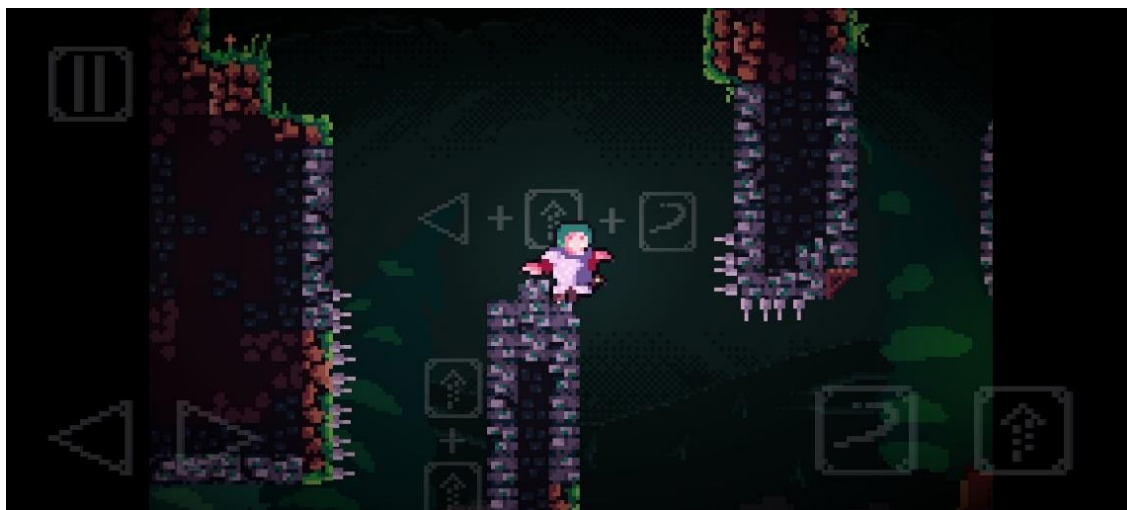


Рисунок 1.28 – Четвертая точка интереса



Рисунок 1.29 – Четвертая точка интереса

Так игрок попадает на завершающий этап обучения, он может преодолеть препятствие используя вертикальный вариант прыжка от стены, однако игра предлагает использовать более оптимизированное по времени решение – использование второй стены для комбинации прыжков. Пятая точка интереса представлена на рисунке 1.30.

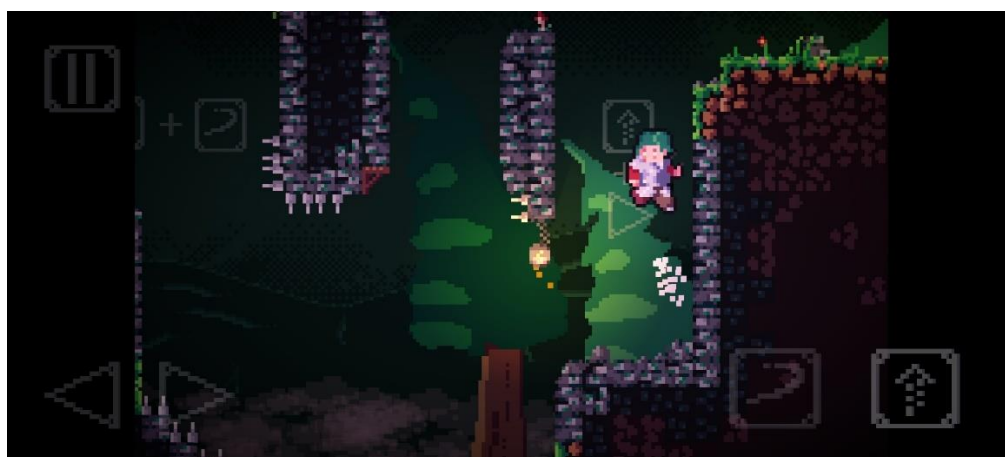


Рисунок 1.30 – Пятая точка интереса

Конец уровня представляет собой переход на сцену главного меню. Стоит заметить, что на протяжении всего уровня игрок сохраняется в контрольных точках, что позволяет опробовать все возможные решения для преодоления преград без страха потери прогресса. Финальный переход представлен на рисунке 1.31.

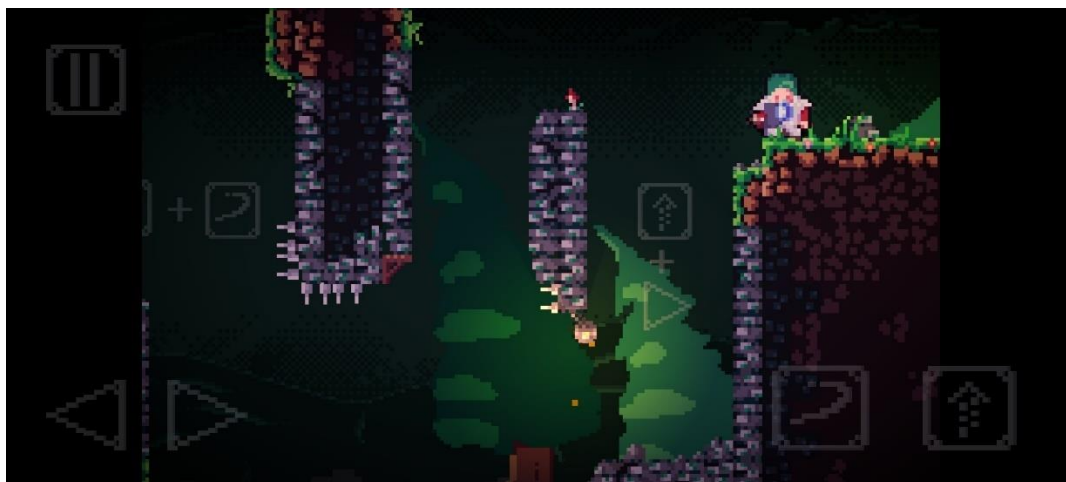


Рисунок 1.31 – Финальный переход

2 Охрана труда

2.1 Негативные факторы, воздействующие на оператора ПЭВМ

Работа с компьютерной техникой обычно связана с ограниченной подвижностью и продолжительным статическим напряжением мышц, а также может приводить к перенапряжению глаз. Эти негативные факторы могут вызывать усталость, снижение работоспособности, психологическое раздражение, физические неприятные ощущения и общую усталость.

2.1.1 Оборудование рабочего места

Для снижения негативного воздействия компьютерной техники на здоровье человека необходимо соблюдать ряд правил использования. Основные из них включают:

- расстояние от глаз до экрана должно быть примерно от 600 до 700 мм;
- монитор следует располагать ниже уровня глаз под углом примерно 10-15 градусов;
- высота рабочего стола должна быть регулируемой в пределах от 680 до 800 мм. Если такая возможность отсутствует, высота стола должна составлять 725 мм;

- клавиатуру следует размещать на поверхности стола на расстоянии от 100 до 300 мм от края, обращенного к пользователю, либо на специальной регулируемой по высоте платформе;

- рабочий стол должен предоставлять достаточное пространство для ног, включая высоту не менее 600 мм, глубину на уровне колен не менее 450 мм и на уровне вытянутых ног не менее 650 мм;

- обеспечение достаточной освещенности рабочего места;

- соблюдение правил электробезопасности и пожарной безопасности;

- регуляция режима работы и отдыха, чтобы предотвратить переутомление и снижение производительности.

На рисунке 2.1 изображена организация рабочего места, отвечающая указанным требованиям.

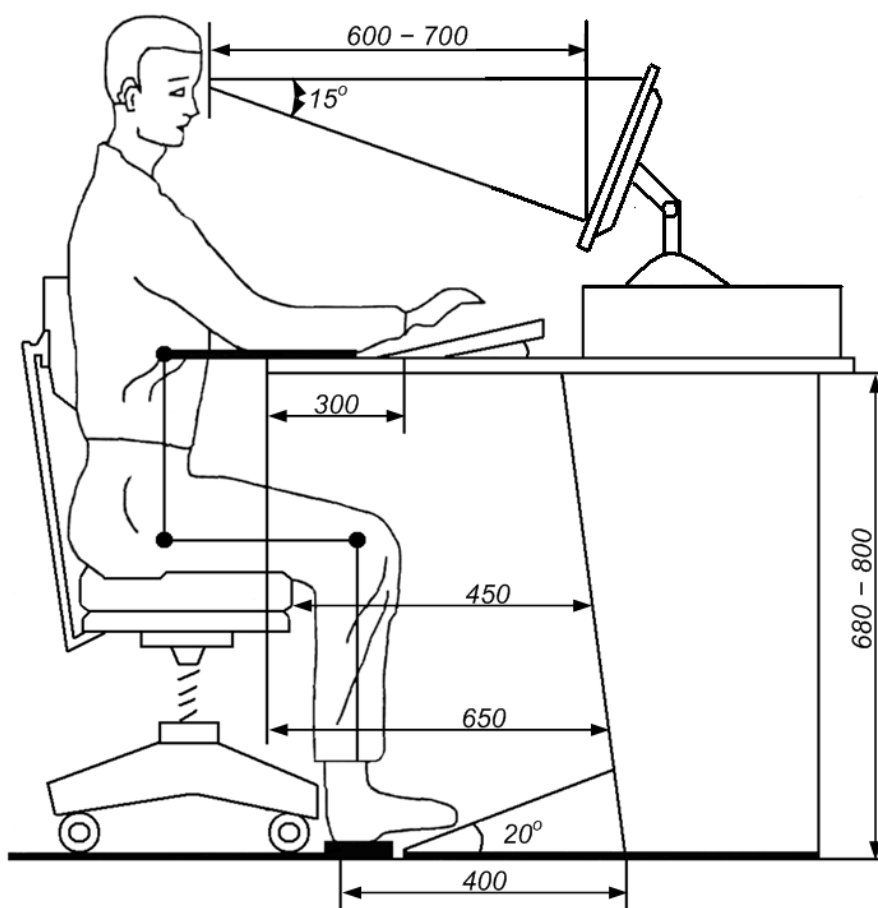


Рисунок 2.1 – Организация рабочего места

Важно, чтобы рабочее место с дисплеем обеспечивало удобство оператора в положении сидя и не создавало перегрузки костно-мышечной системы.

Организовав правильно рабочее место, также необходимо обеспечить правильную посадку пользователя ПК, правильная посадка изображена на рисунке 2.2



Рисунок 2.2 – Правильная посадка

Для достижения правильной посадки пользователя перед компьютером и обеспечения удобства выполнения задач, необходимо учитывать следующие условия:

- запястья должны быть выровнены в прямом положении;
- плечи должны быть расслаблены;
- локти должны полностью опираться на поверхность стола и образовывать угол в 90-100 градусов.

2.1.2 Нормы освещенности

Освещенность рабочего места также является важным фактором, и стандарты освещенности зависят от характера работы и места. Для компьютерных мониторов рекомендуется комбинированное освещение не менее 400 люкс и общее освещение не менее 200 люкс. Равномерность яркости на рабочем месте важна, и яркие или блестящие предметы следует исключить. Рекомендуется также включать яркие поверхности для разнообразия в поле зрения.

Для освещения часто используются потолочные или встроенные светильники с люминесцентными лампами. Чтобы избежать отражений на экране, светильники общего освещения следует установить сбоку от рабочего места и параллельно линии зрения пользователя.

2.1.3 Электробезопасность

Важным аспектом является электробезопасность, и требования к ней регламентируются соответствующими стандартами и законодательством. Для безопасной работы на компьютере рекомендуется следующее:

- все устройства должны быть подключены к одной фазе электрической сети;
- питание устройств ПК должно осуществляться через розетки типа "Европа" с заземляющими контактами;
- розетки для подключения устройств ПК должны быть маркированы по напряжению;
- номинальное напряжение сети должно быть ясно видно возле розетки.

2.1.4 Пожарная безопасность

Пожарная безопасность также требует особых мероприятий. Для предотвращения пожаров и создания безопасных условий рекомендуется:

- не загромождать проходы и запасные выходы предметами, мешающими свободному передвижению;

- обеспечить доступность средств пожаротушения;
- установить автоматическую пожарную сигнализацию и систему пожаротушения в соответствии с требованиями технического содержания установок пожарной автоматики.

2.1.5 Нормы труда и отдыха

В зависимости от продолжительности, вида и категории трудовой деятельности работник, занимающийся работой с ПК, должен соблюдать режим труда и отдыха. Существуют следующие виды трудовой деятельности, связанные с использованием персонального компьютера:

- группа А: работа по чтению информации с экрана видеодисплейного терминала с предварительным запросом;
- группа Б: работа по вводу информации;
- группа В: творческая работа в режиме диалога с ПЭВМ;
- при выполнении работ, относящихся к различным видам трудовой деятельности, основной вид деятельности с ПК определяется как тот, который занимает не менее 50% времени в течение всей рабочей смены или рабочего дня.

Для каждого вида трудовой деятельности устанавливаются 3 категории тяжести и напряженности работы с ПК:

- для группы А категория определяется суммарным числом считываемых знаков за рабочую смену (не более 60000 знаков за смену);
- для группы Б категория определяется суммарным числом считываемых или вводимых знаков за рабочую смену (не более 40000 знаков за смену);
- для группы В категория определяется суммарным временем прямой работы с ПК за рабочую смену (не более 6 часов за смену).

Таблица 2.1 указывает необходимое время на перерывы для каждой из категорий и видов работы.

Таблица 2.1 – Необходимое время на перерывы

Категория работы с ПЭВМ или ВДТ	Уровень нагрузки за рабочую смену при видах работы с ВДТ			Суммарное время регламентированных перерывов, мин	
	группа А, количество знаков	группа Б, количество знаков	группа В час	при 8- часовой смене	при 12- часовой смене
1	До 20000	До 15000	До 2	50	80
2	До 40000	До 30000	До 4	70	110
3	До 60000	До 40000	До 6	90	140

Заключение

В ходе выполнения выпускной квалификационной работы было разработано игровое приложение под ОС *Android* в жанре «платформер» с использованием игрового движка *Unity*, в качестве графического редактора использовался *Aseprite*, редактором кода был выбран *Visual Studio*, в качестве языка программирования использовался *C#*. Приложение было реализовано с использованием встроенных инструментов *Unity*.

Достоинствами разработанного приложения являются визуальное оформление и четкое следование принципам создания игры в жанре «платформер».

Приложение имеет большой потенциал для дальнейшей разработки. С добавлением большого количества контента в виде уровней, введением новых механик, качественно прописанного сюжета, игровое приложение может встать в один ряд с лучшими представителями жанра на платформе *Android* и не иметь аналогов по стилю и качеству визуального оформления.

					ПГУ1.090301.22.001 ПЗ	Лист
						66
Изм.	Лист	№ докум.	Подпись	Дата		

Список использованных источников

1. <https://www.aseprite.org> [Электронный ресурс] URL: <https://www.aseprite.org/docs/> (дата обращения: 20.04.2023)
2. docs.unity.com: [Электронный ресурс]. URL: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.html> (дата обращения: 7.05.2023)
3. <https://learn.microsoft.com>: [Электронный ресурс] URL: <https://learn.microsoft.com/ru-ru/dotnet/csharp/> (дата обращения: 10.04.2023)
4. Дж. Рамбо, М. Блаха *UML 2.0. Объектно-ориентированное моделирование и разработка*, 2-е издание. – СПб.: Питер, 2007 — 544 с.
5. Язык программирования *C# 7* и платформы *.NET* и *.NET Core*, 8-е изд. : Пер. с англ. — СПб. : ООО “Диалектика”, 2018 — 1328 с.

Приложение А

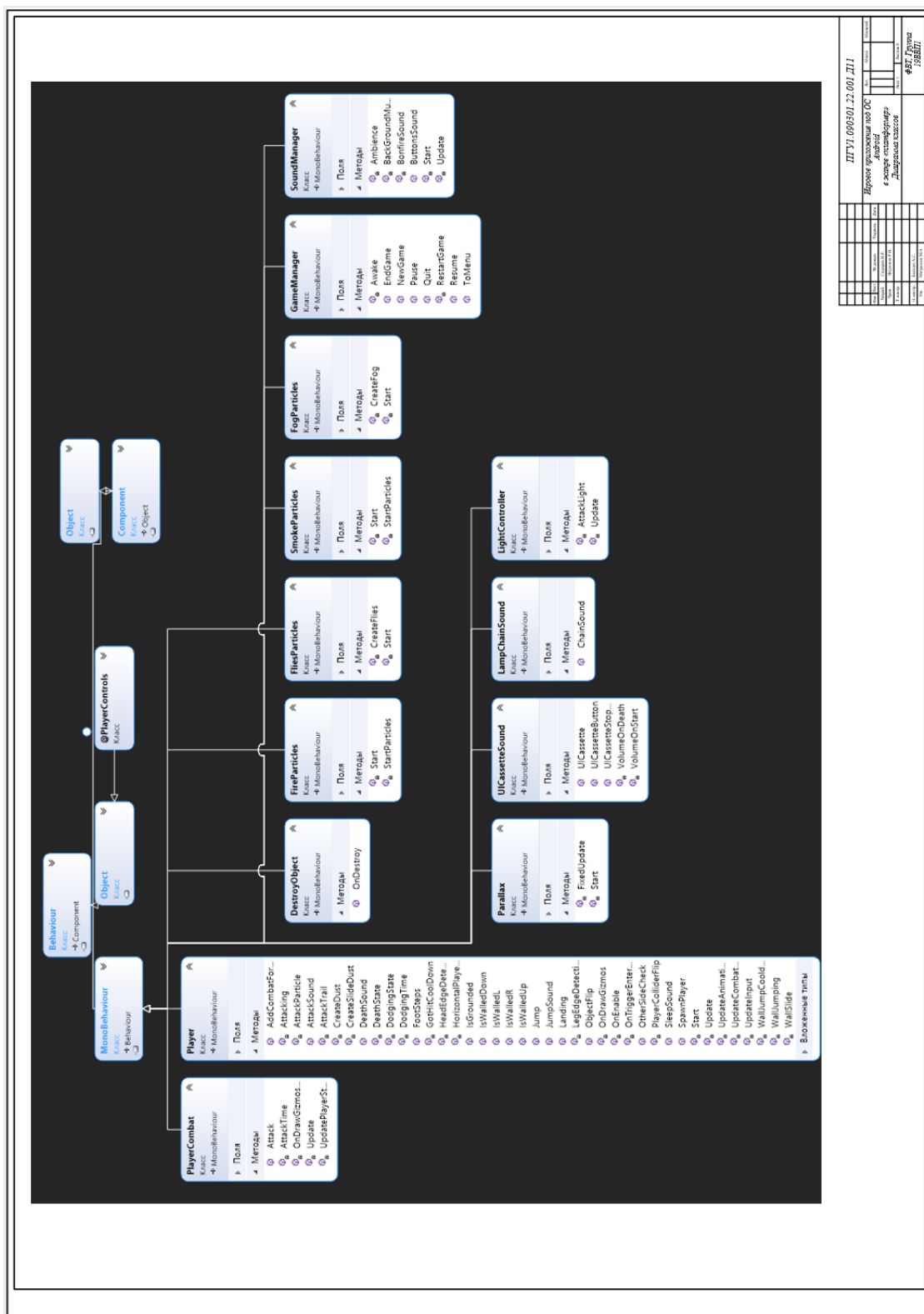
(обязательное)

Игровое приложение под ОС *Android* в жанре «платформер»

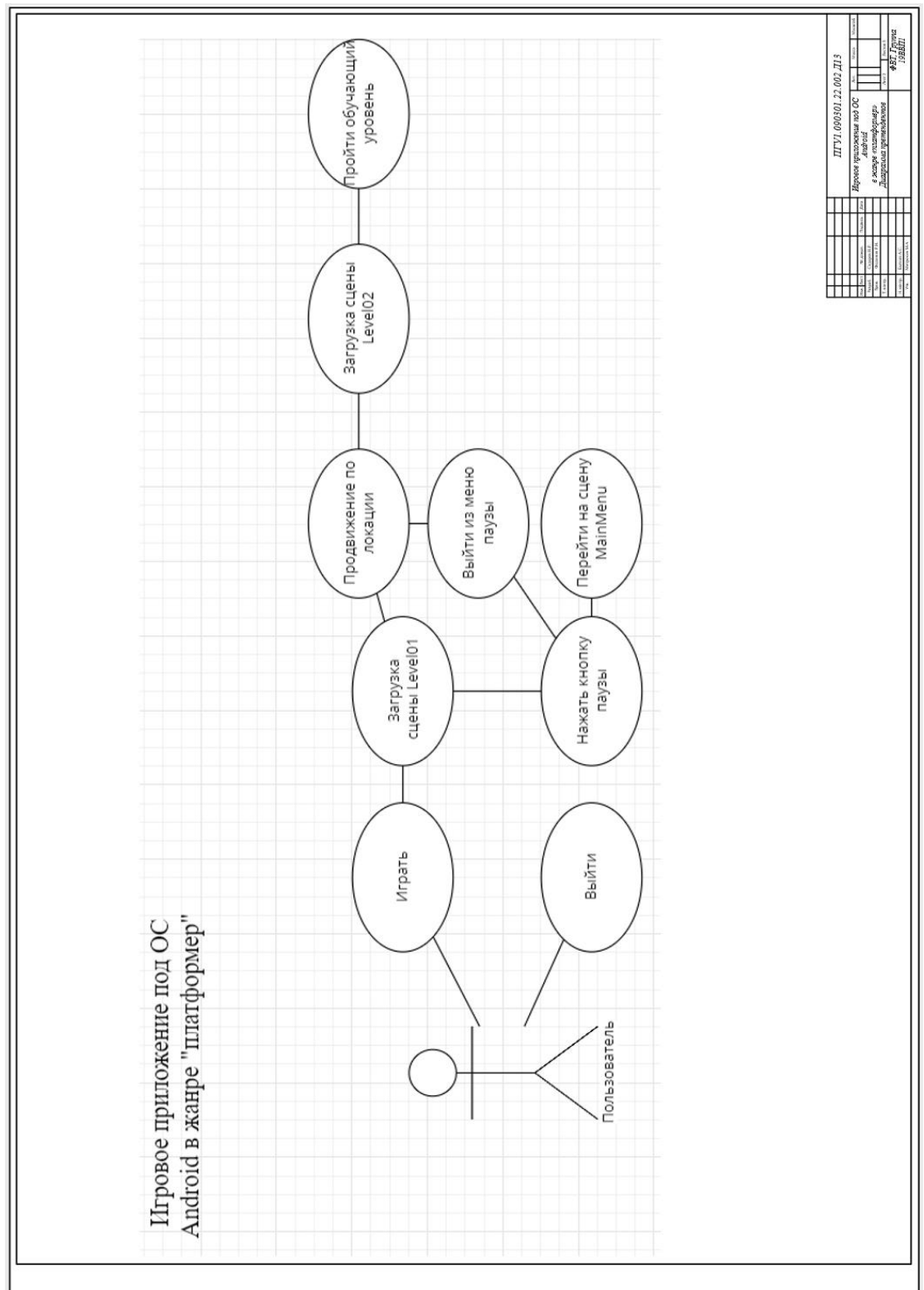
UML - диаграммы

					ПГУ1.090301.22.001 ПЗ	Лист
						68
Изм.	Лист	№ докум.	Подпись	Дата		

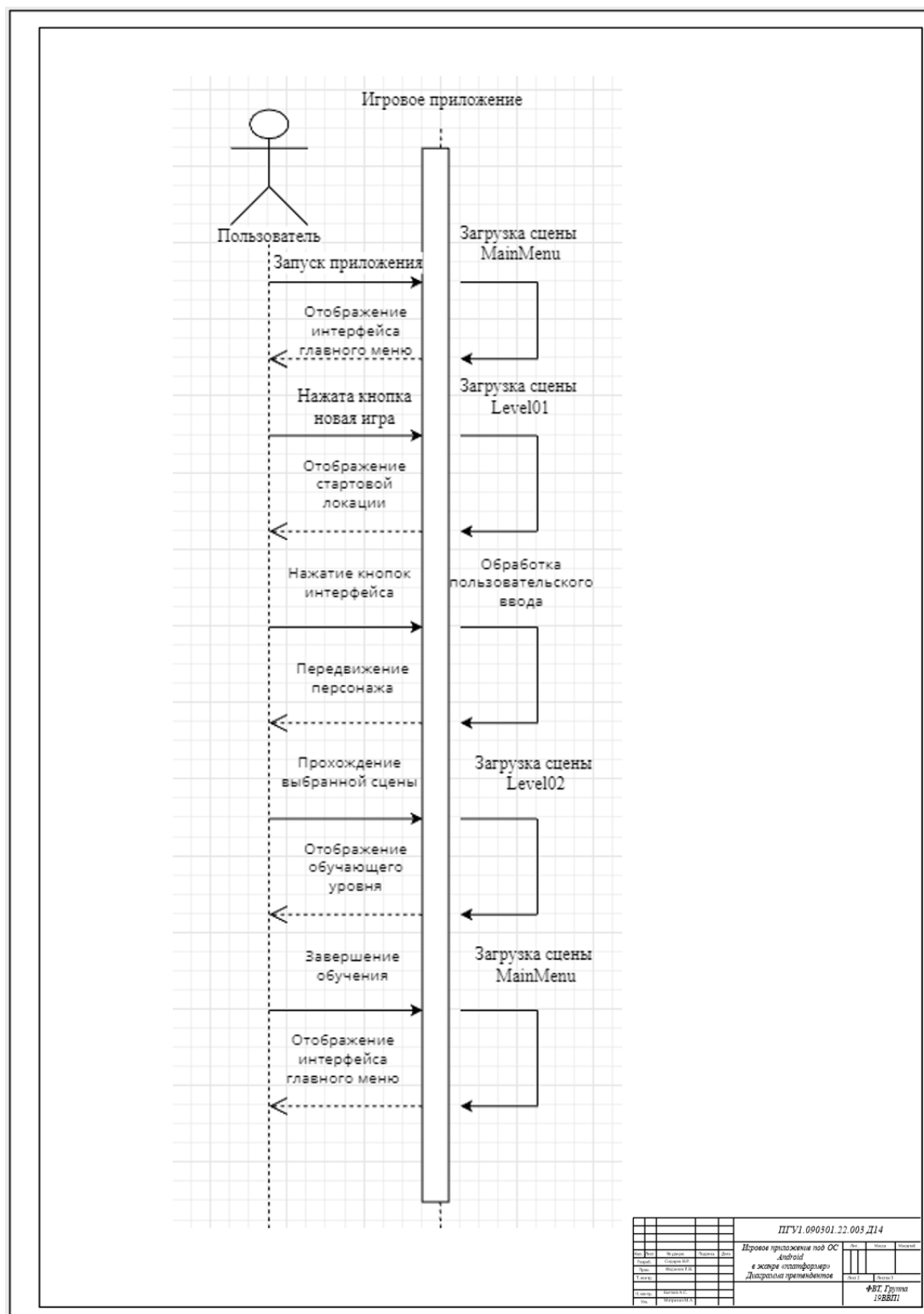
Приложение А. 1 –UML - диаграмма классов



Приложение А.3 –UML - диаграмма прецедентов



Приложение А.2 –UML - диаграмма последовательности



Приложение Б

(обязательное)

Игровое приложение под ОС *Android* в жанре «платформер»

Листинг программы

					ПГУ1.090301.22.001 ПЗ	Лист
						72
Изм.	Лист	№ докум.	Подпись	Дата		

Файл *Player.cs*

```
using UnityEngine;
using System.Collections;
using UnityEngine.InputSystem;
using UnityEngine.SceneManagement;

public class Player : MonoBehaviour
{
    private Rigidbody2D rb;
    private BoxCollider2D coll;
    private Animator anim;

    [SerializeField] private GameManager gameManager;

    public static bool isAlive = true;
    public static bool isFacingRight = true;
    public static bool gotHit = false;
    public static bool isSliding = false;
    public static bool isDodging = false;
    public static bool attackCounter = false;

    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        coll = rb.GetComponent<BoxCollider2D>();
        anim = GetComponent<Animator>();
        controls = new PlayerControls();

        respawnPoint = transform.position;
        health = maxAmountOfHealth;
    }

    private void OnEnable()
    {
        isFacingRight = true;
        isAbleToRun = true;
        updateVelocity = true;

        PlayerCombat.isAttacking = false;

        PlayerCombat.isAbleToAttack = true;
    }

    private void Update()
    {
        DeathState(); // Lose
        UpdateInput(); // Input
        UpdateAnimationState(); // Animations
    }

    #region Respawn
```

```

private Vector3 respawnPoint;

public void SpawnPlayer()
{
    transform.position = respawnPoint;
    health = 1;
}

#endregion

#region Input
private void UpdateInput()
{
    if (isAlive == true) // Если персонаж жив
    {
        HorizontalPlayerMovementController(); //
Горизонтальное перемещение

        if (!IsGrounded() && isWallJumping == false &&
rb.velocity.y > 0f) // Обработка столкновений с углами
        {
            HeadEdgeDetection();
        }

        if (rb.velocity.x != 0f && rb.velocity.y >= 0f
&& !IsGrounded() && !IsWalledUp()) // Обработка столкновений с
углами
        {
            LegEdgeDetection();
        }

        if (IsGrounded()) // CoyoteTime
        {
            coyoteTimeCounter = coyoteTime;
        }
        else
        {
            coyoteTimeCounter -= Time.deltaTime;
        }

        WallSlide(); // Реализация скольжения по стенам

        UpdateCombatState(); // Связь со скриптом
PlayerCombat

        if (Input.GetButtonDown("Fire3")) // Реализация
уклонения
        {
            if (isDodging == false && isAbleToDodge ==
true && IsGrounded())

```

```

        {
            DodgingState();
        }
    }

    if (isAttacking == true) // Импакт от атак
    {
        AttackTrail();
        AttackParticle();
        AddCombatForce();
    }
}

#endregion

#region Movement

// Бер

PlayerControls controls;

[SerializeReference] private float runSpeed = 1f;
[SerializeReference] private float wallSlideSpeed = 1f;

private bool isAbleToRun = true;
private bool updateVelocity = true;

public float dirX = 0f;
private float acceleration = 8f;

// Прыжок

private Coroutine wallJumpCoroutine;

[SerializeReference] private float jumpForce = 2f;

private float coyoteTime = 0.1f;
private float coyoteTimeCounter;

private float jumpButtonTime = 0.2f;
private float jumpButtonCounter;

private bool isWallJumping = false;

private float fallMultiplier = 5.5f;
private float lowJumpMultiplier = 6f;

// Получение урона

```

```

[SerializeReference] private float dodgeForce = 1f;

private bool isAbleToGetHit = true;
private bool isAbleToDodge = true;
private bool onDodgeCooldown = false;

// Атака

private bool isAttacking;

public void ObjectFlip() // Поворот объекта на 180 по
горизонтали
{
    isFacingRight = !isFacingRight;
    Vector3 objectScale = transform.localScale;
    objectScale.x *= -1f;
    transform.localScale = objectScale;
}

private void PlayerColliderFlip() // Поворот персонажа
{
    if (isFacingRight == false && dirX > 0f &&
isAbleToRun == true)
    {
        ObjectFlip();
    }
    else if (isFacingRight == true && dirX < 0f &&
isAbleToRun == true)
    {
        ObjectFlip();
    }
}

private void HorizontalPlayerMovementController() //
Горизонтальное передвижение
{
    if (isAbleToRun == true)
    {
        if (gotHit == false && isDodging == false &&
isAttacking == false)
        {
            controls.Enable();

            controls.OnPlay.Move.performed += ctx =>
            {
                dirX = ctx.ReadValue<float>();
            };
        }
    }

    if (updateVelocity == true)

```

```

        {
            float targetSpeed = dirX * runSpeed;
            float smoothSpeed = Mathf.Lerp(rb.velocity.x,
targetSpeed, Time.deltaTime * acceleration);
            rb.velocity = new Vector2(smoothSpeed,
rb.velocity.y);
        }

        if (dirX != 0)
        {
            isAbleToDodge = false;
        }
        else if (dirX == 0 && onDodgeCooldown == false)
        {
            isAbleToDodge = true;
        }
    }

    public void Jump(InputAction.CallbackContext context)
    {
        if (IsWalledUp() && IsWalledDown() && !IsGrounded()
&& !isAttacking) // Реализация прыжков от стен
        {
            if (context.performed)
            {
                JumpSound();

                CreateDust();
                WallJumping();
            }
        }

        if (context.performed) // JumpBuffer
        {
            jumpButtonCounter = jumpButtonTime;
        }
        else
        {
            jumpButtonCounter -= Time.deltaTime;
        }

        if (gotHit == false && isDodging == false)
        {
            if (isWallJumping == false) // Реализация
прыжков
            {

```

```

        if (jumpButtonCounter > 0f &&
coyoteTimeCounter > 0f)
        {
            JumpSound();

            CreateDust();

            rb.velocity = new Vector2(rb.velocity.x,
jumpForce);

            if (rb.velocity.y < 0)
            {
                rb.velocity += Vector2.up *
Physics2D.gravity.y * (fallMultiplier - 5) * Time.deltaTime;
            }
            else if (rb.velocity.y > 0 &&
!context.performed)
            {
                rb.velocity += Vector2.up *
Physics2D.gravity.y * (lowJumpMultiplier - 5) * Time.deltaTime;
            }

            jumpButtonCounter = 0f;
        }

        if (context.canceled)
        {
            rb.velocity = new Vector2(rb.velocity.x,
rb.velocity.y * 0.5f);

            coyoteTimeCounter = 0f;
        }
    }
}

private void WallSlide() // Скольжение по стенам
{
    if ((IsWalledR() || IsWalledL()) && !IsGrounded() &&
rb.velocity.y < -3f && dirX != 0f)
    {
        if (IsWalledR() && isFacingRight == true)
        {
            isSliding = true;
            rb.velocity = new Vector2(rb.velocity.x,
Mathf.Clamp(rb.velocity.y, -wallSlideSpeed, float.MaxValue));
        }
        else if (IsWalledL() && isFacingRight == false)
        {
            isSliding = true;

```

```

        rb.velocity = new Vector2(rb.velocity.x,
Mathf.Clamp(rb.velocity.y, -wallSlideSpeed, float.MaxValue));
    }

    CreateSlideDust();
}
else
{
    isSliding = false;
}
}

private void WallJumping() // Прыжки от стен
{
    updateVelocity = false;
    isAbleToRun = false;
    attackCounter = true;

    if (isFacingRight == true && !IsWalledL())
    {

        if (dirX != 0 && IsWalledR())
        {
            isWallJumping = true;

            rb.velocity = new Vector2(-12f, jumpForce);

            ObjectFlip();
        }
        else if (IsWalledR())
        {
            isWallJumping = true;

            rb.velocity = new Vector2(-4f, jumpForce +
1f);

            ObjectFlip();
        }
    }
    else if (isFacingRight == false && !IsWalledR())
    {

        if (dirX != 0 && IsWalledL())
        {
            isWallJumping = true;

            rb.velocity = new Vector2(12f, jumpForce);

            ObjectFlip();
        }
        else if (IsWalledL())
        {

```

```

        isWallJumping = true;

        rb.velocity = new Vector2(4f, jumpForce +
1f);
        ObjectFlip();
    }

    if (wallJumpCoroutine != null)
    {
        StopCoroutine(wallJumpCoroutine);
    }

    wallJumpCoroutine =
StartCoroutine(WallJumpCooldown());
}

private IEnumerator WallJumpCooldown() // Кд на прыжок
от стен
{
    yield return new WaitForSeconds(0.3f);
    attackCounter = false;
    isWallJumping = false;

    if (isWallJumping)
    {
        yield break;
    }

    yield return new WaitForSeconds(0.05f);
    updateVelocity = true;
    isAbleToRun = true;

}

private void DodgingState() // Уклонение
{
    isDodging = true;
    isAbleToGetHit = false;
    isAbleToRun = false;
    onDodgeCooldown = true;
    isAbleToDodge = false;
    updateVelocity = false;

    if (isFacingRight == true)
    {
        rb.velocity = new Vector2(dodgeForce * 2,
dodgeForce);
        StartCoroutine(DodgingTime());
    }
    if (isFacingRight == false)

```

```

        {
            rb.velocity = new Vector2(-dodgeForce * 2,
dodgeForce);
            StartCoroutine(DodgingTime());
        }
    }

private IEnumerator DodgingTime() // Кд на уклонение
{
    yield return new WaitForSeconds(0.3f);
    isAbleToGetHit = true;
    isDodging = false;
    updateVelocity = true;
    ObjectFlip();

    yield return new WaitForSeconds(0.5f);
    isAbleToRun = true;

    yield return new WaitForSeconds(2.2f);
    onDodgeCooldown = false;
    isAbleToDodge = true;
}

#endregion

#region Animations

private enum MovementState { idle, jumping, falling,
ground, dead, hit, dodge, sliding, push, walljump, balance,
balance2 } // Значения переменной состояния для менеджера
анимаций
// idle 0 jumping 1 falling 2 ground 3 dead 4 hit 5
dodge 6 sliding 7 push 8 walljump 9 balance 10 balance2 11
private enum MovementRunning { running, coffeeBreak } //
Состояние coffeeBreak отключает анимацию бега
// running 0 coffeeBreak 1

private void UpdateAnimationState()
{
    MovementState state;
    MovementRunning runState;

    // running/idle

    if (dirX > 0f)
    {
        runState = MovementRunning.running;
        state = MovementState.idle;
        PlayerColliderFlip();
    }
    else if (dirX < 0f)

```

```

    {
        runState = MovementRunning.running;
        state = MovementState.idle;
        PlayerColliderFlip();
    }
else
{
    runState = MovementRunning.coffeeBreak;
    state = MovementState.idle;
}

// jumping/falling

if (rb.velocity.y > .1f)
{
    runState = MovementRunning.coffeeBreak;
    state = MovementState.jumping;
}
else if (rb.velocity.y < -.1f)
{
    runState = MovementRunning.coffeeBreak;
    state = MovementState.falling;
}

//sliding
if (isSliding == true)
{
    state = MovementState.sliding;
}

//walljumping
if (isWallJumping == true)
{
    state = MovementState.walljump;
}

//dodging
if (isDodging == true)
{
    state = MovementState.dodge;
}

//pushing/grounding
if (IsGrounded() && IsWalledUp() && IsWalledDown()
&& (IsWalledR() || IsWalledL())) && dirX != 0)
{
    runState = MovementRunning.coffeeBreak;
    state = MovementState.push;
}

```

```

0f)
    else if (IsGrounded() && !IsWalledDown() && dirX ==
    {
        state = MovementState.balance;
    }
    else if (IsGrounded() && !OtherSideCheck() && dirX
== 0f)
    {
        state = MovementState.balance2;
    }
    else if (IsGrounded())
    {
        state = MovementState.ground;
    }

    //hit
    if (gotHit == true)
    {
        state = MovementState.hit;
    }

    //death
    if (isAlive == false)
    {
        state = MovementState.dead;
    }

    anim.SetInteger("runState", (int)runState);
    anim.SetInteger("state", (int)state);
}

#endregion

#region Collisions

[SerializeField] private LayerMask jumpableGround;

[SerializeField] private Transform wallChekUp;
[SerializeField] private Transform wallChekDown;
[SerializeField] private Transform otherSide;

//[SerializeReference] private float hitForce = 2f;

private void OnTriggerEnter2D(Collider2D collision) //
Обработка все всех столкновений
{
    if (collision.gameObject.CompareTag("DamagePlayer"))
// Обработка столкновений с хитбоксами урона
{
    if (isAbleToGetHit == true)

```

```

        {
            gotHit = true;
            isAbleToRun = false;
            updateVelocity = false;
            health = 0;

            StartCoroutine(GotHitCoolDown());

            /*BoxCollider2D hitboxCollider = collision
as BoxCollider2D;
            BoxCollider2D playerCollider =
GetComponent<BoxCollider2D>();

            if (hitboxCollider != null && playerCollider
!= null)
            {
                Vector2 hitboxCenter =
hitboxCollider.bounds.center;
                Vector2 characterCenter =
playerCollider.bounds.center;

                if (hitboxCenter.x < characterCenter.x)
                {
                    if (isFacingRight == true)
                    {
                        ObjectFlip();
                    }
                    rb.velocity = new Vector2(hitForce,
hitForce);

                    StartCoroutine(GotHitCoolDown());
                }
                else if (hitboxCenter.x >
characterCenter.x)
                {
                    if (isFacingRight == false)
                    {
                        ObjectFlip();
                    }
                    rb.velocity = new Vector2(-hitForce,
hitForce);

                    StartCoroutine(GotHitCoolDown());
                }
            }
        }
    }

    if (collision.gameObject.CompareTag("CheckPoint"))
    {
        respawnPoint = transform.position;
    }
}

```

```

        if (collision.gameObject.CompareTag("End"))
        {
            SceneManager.LoadScene(0);
        }

        if (collision.gameObject.CompareTag("Next"))
        {
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex
+ 1);
        }

        if (collision.gameObject.CompareTag("Previous"))
        {
SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex
- 1);
        }
    }

    private IEnumerator GotHitCoolDown() // Кд на получение
урона
    {
        yield return new WaitForSeconds(1f);
        gotHit = false;
        updateVelocity = true;
        isAbleToRun = true;
    }

    public bool IsWalledUp() // Касается стены сверху
    {
        return Physics2D.OverlapCircle(wallChekUp.position,
0.5f, jumpableGround);
    }

    public bool IsWalledDown() // Касается стены снизу
    {
        return
Physics2D.OverlapCircle(wallChekDown.position, 0.4f,
jumpableGround);
    }

    public bool OtherSideCheck() // Касается земли сзади
    {
        return Physics2D.OverlapCircle(otherSide.position,
0.4f, jumpableGround);
    }

    public bool IsWalledR() // Касается правой стены
    {
        bool isTouchingRightWall;

```

```

        return isTouchingRightWall =
Physics2D.BoxCast(coll.bounds.center, coll.bounds.size, 0f,
Vector2.right, 0.4f, jumpableGround);
    }

    public bool IsWalledL() // Касается левой стены
    {
        bool isTouchingLeftWall;

        return isTouchingLeftWall =
Physics2D.BoxCast(coll.bounds.center, coll.bounds.size, 0f,
Vector2.left, 0.4f, jumpableGround);
    }

    private void LegEdgeDetection() // Помощь с
запрыгиванием на препятствия
    {
        float displacement = 10f;

        Collider2D legWallChek =
Physics2D.OverlapCircle(wallChekDown.position, 0.2f,
jumpableGround);

        Vector3 newPosition = transform.position;
        if (legWallChek != null)
        {
            newPosition.y += displacement * Time.deltaTime;
        }
        transform.position = newPosition;
    }

    private void HeadEdgeDetection() // Помощь в огибании
препятствий при прыжке
    {
        Vector2 colliderCenter = coll.bounds.center;
        Vector2 colliderSize = coll.size;

        float radius = 0.37f;
        float displacement = 25f;

        Vector2 checkPointLeft = new
Vector2(colliderCenter.x - colliderSize.x / 2 + 0.37f,
colliderCenter.y + colliderSize.y / 2);
        Vector2 checkPointRight = new
Vector2(colliderCenter.x + colliderSize.x / 2 - 0.37f,
colliderCenter.y + colliderSize.y / 2);

        Collider2D leftCollider =
Physics2D.OverlapCircle(checkPointLeft, radius,
LayerMask.GetMask("Ground"));
    }

```

```

        Collider2D rightCollider =
Physics2D.OverlapCircle(checkPointRight, radius,
LayerMask.GetMask("Ground"));

        Vector3 newPosition = transform.position;
        if (leftCollider != null && rightCollider == null)
        {
            newPosition.x += displacement *
Time.fixedDeltaTime;
        }
        else if (leftCollider == null && rightCollider !=
null)
        {
            newPosition.x -= displacement *
Time.fixedDeltaTime;
        }
        transform.position = newPosition;
    }

    public bool IsGrounded() // Проверка касания земли
    {
        return Physics2D.BoxCast(coll.bounds.center,
coll.bounds.size, 0f, Vector2.down, .1f, jumpableGround);
    }

    #endregion

    #region Lose

    public int health;
    [SerializeField] private int maxAmountOfHealth = 1;

    private void DeathState() //
    {
        if (health == 0)
        {
            rb.velocity = Vector2.zero;
            isAlive = false;
            gameManager.EndGame();
        }
        else
        {
            isAlive = true;
        }
    }

    #endregion

    #region CombatPhysics

```

```

        private void UpdateCombatState() // Связь со скриптом
PlayerCombat
    {
        isAttacking = PlayerCombat.isAttacking;
    }

    public void AddCombatForce() // Добавляет импакт при
атаке
    {
        Vector2 rCombatForceVector = new Vector2(25f,
rb.velocity.y * 0.1f);
        Vector2 lCombatForceVector = new Vector2(-25f,
rb.velocity.y * 0.1f);

        isAbleToRun = false;

        if (IsGrounded())
        {
            if (dirX != 0)
            {
                rCombatForceVector = new Vector2(20f,
rb.velocity.y);
                lCombatForceVector = new Vector2(-20f,
rb.velocity.y);
            }
            else
            {
                rCombatForceVector = new Vector2(10f,
rb.velocity.y);
                lCombatForceVector = new Vector2(-10f,
rb.velocity.y);
            }
        }

        if (isFacingRight == true)
        {
            rb.velocity = rCombatForceVector;

            StartCoroutine(Attacking());
        }

        if (isFacingRight == false)
        {
            rb.velocity = lCombatForceVector;

            StartCoroutine(Attacking());
        }
    }
}

```

```

        private IEnumerator Attacking() // Отключение
возможности двигаться во время атаки
        {
            yield return new WaitForSeconds(0.3f);
            isAbleToRun = true;

        }

#endregion

#region Audio

[SerializeField] private SoundManager sound;

public void JumpSound()
{
    sound.jumpSoundEffect.Play();
}

public void FootSteps()
{
    sound.runSoundEffect.Play();
}

public void Landing()
{
    sound.landingSoundEffect.Play();
}

public void AttackSound()
{
    sound.attackSoundEffect.Play();
}

public void SleepSound()
{
    sound.sleepSoundEffect.Play();
}

public void DeathSound()
{
    sound.deathSoundEffect.Play();
}

#endregion

#region Particles

[SerializeField] private ParticleSystem Dust;
[SerializeField] private ParticleSystem SlideDustRight;
[SerializeField] private ParticleSystem SlideDustLeft;

```

```

private void CreateDust()
{
    Dust.Play();
}

private void CreateSlideDust()
{
    if (isFacingRight == true)
    {
        SlideDustRight.Play();
    }
    else
    {
        SlideDustLeft.Play();
    }
}

[SerializeField] private ParticleSystem AttackParticles;
[SerializeField] private ParticleSystem
AttackTrailParticlesR;
[SerializeField] private ParticleSystem
AttackTrailParticlesL;

private void AttackParticle()
{
    AttackParticles.Play();
}

private void AttackTrail()
{
    if (isFacingRight == true)
    {
        AttackTrailParticlesL.Play();
    }
    else
    {
        AttackTrailParticlesR.Play();
    }
}

#endregion

#region Gizmos

private void OnDrawGizmos()
{
    BoxCollider2D playerCollider =
GetComponent<BoxCollider2D>();

```

```

        Vector2 colliderCenter =
playerCollider.bounds.center;
        Vector2 colliderSize = playerCollider.size;

        //Визуализация областей касания "головой" и
"ногами"

        Gizmos.DrawWireSphere(wallChekUp.position,
0.4f);
        Gizmos.DrawWireSphere(wallChekDown.position,
0.4f);
        Gizmos.DrawWireSphere(otherSide.position, 0.4f);

        // Визуализация областей столкновения "головой"

        float radius = 0.37f;

        Vector2 checkPointLeft = new
Vector2(colliderCenter.x - colliderSize.x / 2 + 0.37f,
colliderCenter.y + colliderSize.y / 2);
        Vector2 checkPointRight = new
Vector2(colliderCenter.x + colliderSize.x / 2 - 0.37f,
colliderCenter.y + colliderSize.y / 2);

        Gizmos.DrawWireSphere(checkPointLeft, radius);

        Gizmos.DrawWireSphere(checkPointRight, radius);
    }

    #endregion
}

```

Файл *PlayerCombat.cs*

```

using UnityEngine;
using System.Collections;
using UnityEngine.InputSystem;

public class PlayerCombat : MonoBehaviour
{

    [SerializeField]private Transform attackPoint;
    [SerializeField]private Animator anim;
    [SerializeField]private Player player;

    //[SerializeReference] private int attackDamage = 1;

    private int maxAttackCount = 2;
    private int attackCount;

```

```

public static bool isAttacking = false;
public static bool isAbleToAttack = true;

private bool isAlive;
private bool gotHit;
private bool isDodging;
private bool attackCounter;

void Update()
{
    UpdatePlayerState();

    if(player.IsGrounded())
    {
        attackCount = maxAttackCount;
    }
}

private void UpdatePlayerState()// Проверяет состояния
персонажа
{
    isAlive = Player.isAlive;
    gotHit = Player.gotHit;
    isDodging = Player.isDodging;
    attackCounter = Player.attackCounter;
}

public void Attack(InputAction.CallbackContext context)//
Реализация атаки и попадания
{
    if (isAlive == true)
    {
        if (gotHit == false && isDodging == false &&
attackCounter == false && !player.IsWalledR() &&
!player.IsWalledL())
        {
            if (isAbleToAttack == true && attackCount > 0)
            {
                isAttacking = true;
                isAbleToAttack = false;

                attackCount -= 1;

                anim.SetTrigger("attack");

                player.AddCombatForce();

                Collider2D[] hitEnemies =
Physics2D.OverlapCircleAll(attackPoint.position, 1f); // Создает
круг, привязанный к игроку, который собирает все хитбоксы

```

```

        foreach (Collider2D enemy in hitEnemies)//
Работает с полученными хитбоксами через теги
        {
            if (enemy.CompareTag("KillerObject"))
            {
                //enemy.GetComponent<Enemy>().TakeDamage(attackDamage);
                Debug.Log("hit");
            }
        }

        StartCoroutine(AttackTime());
    }
}

private IEnumerator AttackTime()// Осуществляет выход из
состояния атаки
{
    yield return new WaitForSeconds(0.3f);
    isAttacking = false;

    yield return new WaitForSeconds(0.6f);
    isAbleToAttack = true;
}

private void OnDrawGizmosSelected() // Позволяет увидеть
созданный круг в редакторе сцены во время атаки
{
    if (attackPoint == null)
        return;

    if (isAttacking == true)
    {
        Gizmos.DrawWireSphere(attackPoint.position, 1f);
    }
}

```

Файл *PlayerControls.cs*

```

//-----
//-----
// <auto-generated>
//      This code was auto-generated by
com.unity.inputsystem:InputActionCodeGenerator
//      version 1.5.1

```

```

//      from Assets/Scripts/PlayerControls.inputactions
//
//      Changes to this file may cause incorrect behavior and
//      will be lost if
//      the code is regenerated.
// </auto-generated>
//-----
-----

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.InputSystem;
using UnityEngine.InputSystem.Utilities;

public partial class @PlayerControls: IInputActionCollection2,
IDisposable
{
    public InputActionAsset asset { get; }
    public @PlayerControls()
    {
        asset = InputActionAsset.FromJson(@"{
""name"": ""PlayerControls"",
""maps"": [
    {
        ""name"": ""OnPlay"",
        ""id"": ""da361e45-8ac6-4c20-97e1-5c117ff1b71a"",
        ""actions"": [
            {
                ""name"": ""Move"",
                ""type"": ""Value"",
                ""id"": ""7b5417cb-1162-4058-ba0a-
d1db53bde19c"",
                ""expectedControlType"": "",
                ""processors"": "",
                ""interactions"": ""Press(behavior=2)"",
                ""initialStateCheck"": true
            },
            {
                ""name"": ""Jump"",
                ""type"": ""Button"",
                ""id"": ""9a340802-319c-499f-953a-
7b9d8e8b29d8"",
                ""expectedControlType"": ""Button"",
                ""processors"": "",
                ""interactions"": "",
                ""initialStateCheck"": false
            },
            {
                ""name"": ""Attack"",
                ""type"": ""Button"",

```

```

        ""id"": ""87ef41c5-1a2b-4ce2-9bbf-
14597dc01b5d"",
        ""expectedControlType"": ""Button"",
        ""processors"": "",
        ""interactions"": "",
        ""initialStateCheck"": false
    }
],
""bindings"": [
    {
        ""name"": ""1D Axis"",
        ""id"": ""8746b6ee-34f2-4b4a-bc40-
7e2a989912b2"",
        ""path"": ""1DAxis"",
        ""interactions"": "",
        ""processors"": "",
        ""groups"": "",
        ""action"": ""Move"",
        ""isComposite"": true,
        ""isPartOfComposite"": false
    },
    {
        ""name"": ""negative"",
        ""id"": ""68db2504-26eb-4721-924f-
49cc7e436e17"",
        ""path"": ""<Keyboard>/a"",
        ""interactions"": "",
        ""processors"": "",
        ""groups"": "",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": ""positive"",
        ""id"": ""f1c14e42-090d-483e-bad6-
8483eafe51f2"",
        ""path"": ""<Keyboard>/d"",
        ""interactions"": "",
        ""processors"": "",
        ""groups"": "",
        ""action"": ""Move"",
        ""isComposite"": false,
        ""isPartOfComposite"": true
    },
    {
        ""name"": "",
        ""id"": ""f19f9920-eccb-4126-a3bf-
10bc8cb37c3e"",
        ""path"": ""<Keyboard>/space"",
        ""interactions"": "",

```

```

        ""processors"": "",
        ""groups"": "",
        ""action"": ""Jump"",
        ""isComposite"": false,
        ""isPartOfComposite"": false
    },
    {
        ""name"": "",
        ""id"": ""90631a64-f4a8-4b49-b058-
8d9b223eac68"",
        ""path"": ""<Keyboard>/shift"",
        ""interactions"": "",
        ""processors"": "",
        ""groups"": "",
        ""action"": ""Attack"",
        ""isComposite"": false,
        ""isPartOfComposite"": false
    }
]
},
""controlSchemes"": []
}");

    // OnPlay
    m_OnPlay = asset.FindActionMap("OnPlay",
throwIfNotFound: true);
    m_OnPlay_Move = m_OnPlay.FindAction("Move",
throwIfNotFound: true);
    m_OnPlay_Jump = m_OnPlay.FindAction("Jump",
throwIfNotFound: true);
    m_OnPlay_Attack = m_OnPlay.FindAction("Attack",
throwIfNotFound: true);
}

public void Dispose()
{
    UnityEngine.Object.Destroy(asset);
}

public InputBinding? bindingMask
{
    get => asset.bindingMask;
    set => asset.bindingMask = value;
}

public ReadOnlyArray<InputDevice>? devices
{
    get => asset.devices;
    set => asset.devices = value;
}

```

```

    public ReadOnlyArray<InputControlScheme> controlSchemes =>
asset.controlSchemes;

    public bool Contains(InputAction action)
    {
        return asset.Contains(action);
    }

    public IEnumerator<InputAction> GetEnumerator()
    {
        return asset.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    public void Enable()
    {
        asset.Enable();
    }

    public void Disable()
    {
        asset.Disable();
    }

    public IEnumerable<InputBinding> bindings => asset.bindings;

    public InputAction FindAction(string actionNameOrId, bool
throwIfNotFound = false)
    {
        return asset.FindAction(actionNameOrId,
throwIfNotFound);
    }

    public int FindBinding(InputBinding bindingMask, out
InputAction action)
    {
        return asset.FindBinding(bindingMask, out action);
    }

    // OnPlay
    private readonly InputActionMap m_OnPlay;
    private List<IONPlayActions>
m_OnPlayActionsCallbackInterfaces = new List<IONPlayActions>();
    private readonly InputAction m_OnPlay_Move;
    private readonly InputAction m_OnPlay_Jump;
    private readonly InputAction m_OnPlay_Attack;
    public struct OnPlayActions

```

```

{
    private @PlayerControls m_Wrapper;
    public OnPlayActions(@PlayerControls wrapper) {
m_Wrapper = wrapper; }
    public InputAction @Move => m_Wrapper.m_OnPlay_Move;
    public InputAction @Jump => m_Wrapper.m_OnPlay_Jump;
    public InputAction @Attack => m_Wrapper.m_OnPlay_Attack;
    public InputActionMap Get() { return m_Wrapper.m_OnPlay;
}

    public void Enable() { Get().Enable(); }
    public void Disable() { Get().Disable(); }
    public bool enabled => Get().enabled;
    public static implicit operator
InputActionMap(OnPlayActions set) { return set.Get(); }
    public void AddCallbacks(IONPlayActions instance)
    {
        if (instance == null ||
m_Wrapper.m_OnPlayActionsCallbackInterfaces.Contains(instance))
return;

m_Wrapper.m_OnPlayActionsCallbackInterfaces.Add(instance);
        @Move.started += instance.OnMove;
        @Move.performed += instance.OnMove;
        @Move.canceled += instance.OnMove;
        @Jump.started += instance.OnJump;
        @Jump.performed += instance.OnJump;
        @Jump.canceled += instance.OnJump;
        @Attack.started += instance.OnAttack;
        @Attack.performed += instance.OnAttack;
        @Attack.canceled += instance.OnAttack;
    }

    private void UnregisterCallbacks(IONPlayActions
instance)
    {
        @Move.started -= instance.OnMove;
        @Move.performed -= instance.OnMove;
        @Move.canceled -= instance.OnMove;
        @Jump.started -= instance.OnJump;
        @Jump.performed -= instance.OnJump;
        @Jump.canceled -= instance.OnJump;
        @Attack.started -= instance.OnAttack;
        @Attack.performed -= instance.OnAttack;
        @Attack.canceled -= instance.OnAttack;
    }

    public void RemoveCallbacks(IONPlayActions instance)
    {
        if
(m_Wrapper.m_OnPlayActionsCallbackInterfaces.Remove(instance))
            UnregisterCallbacks(instance);
    }

```

```

    }

    public void SetCallbacks(IOnPlayActions instance)
    {
        foreach (var item in
m_Wrapper.m_OnPlayActionsCallbackInterfaces)
            UnregisterCallbacks(item);
        m_Wrapper.m_OnPlayActionsCallbackInterfaces.Clear();
        AddCallbacks(instance);
    }
}

public OnPlayActions @OnPlay => new OnPlayActions(this);
public interface IOnPlayActions
{
    void OnMove(InputAction.CallbackContext context);
    void OnJump(InputAction.CallbackContext context);
    void OnAttack(InputAction.CallbackContext context);
}
}

```

Файл *GameManager.cs*

```

using UnityEngine;
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour
{
    [SerializeField] private Animator sceneAnim;
    [SerializeField] private Animator UICassette;

    [SerializeField] private SoundManager sound;

    [SerializeField] private GameObject PausePanel;
    [SerializeField] private GameObject MobilePanel;

    [SerializeField] private Player player;

    private bool isFacingRight;

    void Awake()
    {
        Application.targetFrameRate = 90;
    }

    public void NewGame()
    {
        sound.ButtonsSound();

        SceneManager.LoadScene(1);
    }

    public void Quit()

```

```

{
    sound.ButtonsSound();
    Application.Quit();
}

public void ToMenu()
{
    Time.timeScale = 1f;

    sound.ButtonsSound();

    SceneManager.LoadScene(0);
}

public void Pause()
{
    PausePanel.SetActive(true);
    MobilePanel.SetActive(false);
    Time.timeScale = 0f;
    sound.ButtonsSound();
    sound.volume = false;
}

public void Resume()
{
    PausePanel.SetActive(false);
    MobilePanel.SetActive(true);
    Time.timeScale = 1f;
    sound.ButtonsSound();
    sound.volume = true;
}

public void EndGame() // Логика завершения игры
{
    UICassette.SetTrigger("Stop");
    sceneAnim.SetInteger("Start", 0);

    Invoke("RestartGame", 1f);
}
private void RestartGame() // Перезапуск уровня
{
    player.SpawnPlayer();

    isFacingRight = Player.isFacingRight;

    UICassette.SetTrigger("Start");
    sceneAnim.SetInteger("Start", 1);

    if (!isFacingRight && player.dirX == 0f)
    {
        player.ObjectFlip();
    }
}

```

```

    }
}

}

Файл SoundManager.cs

using UnityEngine;

public class SoundManager : MonoBehaviour
{
    [SerializeField] private Animator playerAnimator;

    [SerializeField] public AudioSource jumpSoundEffect;
    [SerializeField] public AudioSource runSoundEffect;
    [SerializeField] public AudioSource landingSoundEffect;
    [SerializeField] public AudioSource attackSoundEffect;
    [SerializeField] public AudioSource sleepSoundEffect;
    [SerializeField] public AudioSource deathSoundEffect;
    [SerializeField] public AudioSource Lamp;
    [SerializeField] public AudioSource Lamp2;

    [SerializeField] public AudioSource bGMusic;
    [SerializeField] public AudioSource ambience;
    [SerializeField] public AudioSource bonfire;

    [SerializeField] public AudioSource UICassetteButton;
    [SerializeField] public AudioSource UICassetteSound;
    [SerializeField] public AudioSource UICassetteStop;
    [SerializeField] public AudioSource Buttons;

    public bool volume = true;

    private void Start()
    {
        BackGroundMusic();
        Ambience();
        BonfireSound();
    }

    private void Update()
    {
        if (volume)
        {
            bGMusic.volume = 0.05f;
            ambience.volume = 0.02f;
            bonfire.volume = 0.05f;

            if(Lamp && Lamp2 != null)
            {

```

```

        Lamp.volume = 0.05f;
        Lamp2.volume = 0.05f;
    }
}
else
{
    bGMusic.volume = 0f;
    ambience.volume = 0f;
    bonfire.volume = 0f;

    if (Lamp && Lamp2 != null)
    {
        Lamp.volume = 0f;
        Lamp2.volume = 0f;
    }
}

if (volume)
{
    jumpSoundEffect.volume = 0.3f;
    runSoundEffect.volume = 0.3f;
    landingSoundEffect.volume = 0.3f;
    attackSoundEffect.volume = 0.3f;
    sleepSoundEffect.volume = 0.1f;
}
else
{
    jumpSoundEffect.volume = 0f;
    runSoundEffect.volume = 0f;
    landingSoundEffect.volume = 0f;
    attackSoundEffect.volume = 0f;
    sleepSoundEffect.volume = 0f;
}
}

private void BackGroundMusic()
{
    bGMusic.Play();
}

private void Ambience()
{
    ambience.Play();
}

private void BonfireSound()
{
    bonfire.Play();
}

public void ButtonsSound()

```

```

    {
        Buttons.Play();
    }
}

```

Файл *LightController.cs*

```

using UnityEngine;
using System.Collections;
using UnityEngine.Rendering.Universal;

public class LightController : MonoBehaviour
{
    public Light2D PlayerLight;

    private bool isAttacking;
    private bool attackLight = false;

    Color attack = new Color32(0x60, 0xC6, 0x93, 0xFF);
    Color outAttack = new Color32(0xFF, 0xFF, 0xFF, 0xFF);

    void Update()
    {
        isAttacking = PlayerCombat.isAttacking;

        if (isAttacking)
        {
            attackLight = true;

            StartCoroutine(AttackLight());
        }
        else if (attackLight == false)
        {
            // Восстановление интенсивности света в обычное
            состояние
            PlayerLight.intensity = 0.3f;
            PlayerLight.color = outAttack;
            PlayerLight.pointLightOuterRadius = 4f;
        }
    }

    private IEnumerator AttackLight()
    {
        yield return new WaitForSeconds(0.1f);
        PlayerLight.intensity = 1f;
        PlayerLight.color = attack;

        yield return new WaitForSeconds(0.1f);
        PlayerLight.pointLightOuterRadius = 3f;

        yield return new WaitForSeconds(0.8f);
        PlayerLight.pointLightOuterRadius = 4f;
    }
}

```

```

        attackLight = false;
    }
}

```

Файл *UICassetteSound.cs*

```

using UnityEngine;
using System.Collections;

public class UICassetteSound : MonoBehaviour
{
    [SerializeField] private SoundManager sound;

    public void UICassetteButton()
    {
        sound.UICassetteButton.Play();
    }

    public void UICassette()
    {
        sound.UICassetteSound.Play();

        sound.volume = false;

        StartCoroutine(VolumeOnStart());
    }

    public void UICassetteStopSound()
    {
        sound.UICassetteStop.Play();

        sound.volume = false;

        StartCoroutine(VolumeOnDeath());
    }

    private IEnumerator VolumeOnStart()
    {
        yield return new WaitForSeconds(2f);
        sound.volume = true;
    }

    private IEnumerator VolumeOnDeath()
    {
        yield return new WaitForSeconds(2.5f);
        sound.volume = true;
    }
}

```

Файл *LampChainSound.cs*

```

using UnityEngine;

public class LampChainSound : MonoBehaviour
{
    public AudioSource chain;

    public void ChainSound()
    {
        chain.Play();
    }
}

```

Файл *DestroyObject.cs*

```

using UnityEngine;

public class DestroyObject : MonoBehaviour
{
    public void OnDestroy()
    {
        Destroy(gameObject);
    }
}

```

Файл *Parallax.cs*

```

using UnityEngine;

public class Parallax : MonoBehaviour
{
    private float length;
    private float startpos;
    public GameObject cam;
    public float parallaxEffect;

    void Start()
    {
        startpos = transform.position.x;
        length = GetComponent().bounds.size.x;
    }

    void FixedUpdate()
    {
        float temp = (cam.transform.position.x * (1 -
parallaxEffect));
        float dist = (cam.transform.position.x *
parallaxEffect);

        transform.position = new Vector3(startpos + dist,
transform.position.y, transform.position.z);
    }
}

```

```

        if (temp > startpos + length) startpos += length;
        else if (temp < startpos - length) startpos -= length;
    }
}

```

Файл *FireParticles.cs*

```

using UnityEngine;

public class FireParticles : MonoBehaviour
{
    [SerializeField] private ParticleSystem FireParticle;
    void Start()
    {
        StartParticles();
    }

    private void StartParticles()
    {
        FireParticle.Play();
    }
}

```

Файл *FliesParticles.cs*

```

using UnityEngine;

public class FliesParticles : MonoBehaviour
{
    [SerializeField] private ParticleSystem Flies;

    void Start()
    {
        CreateFlies();
    }

    private void CreateFlies()
    {
        Flies.Play();
    }
}

```

Файл *FogParticles.cs*

```

using UnityEngine;

public class FogParticles : MonoBehaviour
{
    [SerializeField] private ParticleSystem Fog;

    void Start()
    {
        CreateFog();
    }
}

```

```

    }

    private void CreateFog()
    {
        Fog.Play();
    }
}

```

Файл *SmokeParticles.cs*

```

using UnityEngine;

public class SmokeParticles : MonoBehaviour
{
    [SerializeField] private ParticleSystem SmokeParticle;
    void Start()
    {
        StartParticles();
    }

    private void StartParticles()
    {
        SmokeParticle.Play();
    }
}

```

					ПГУ1.090301.22.001 ПЗ	Лист
						107
Изм.	Лист	№ докум.	Подпись	Дата		

Приложение В

(обязательное)

Игровое приложение под ОС *Android* в жанре «платформер»

Презентация

					ПГУ1.090301.22.001 ПЗ	Лист
						108
Изм.	Лист	№ докум.	Подпись	Дата		

Игровое приложение под *OC Android* в жанре «платформер»

Автор работы:

Студент группы 19ВВП1 Сидоров Н.Р.

Научный руководитель:

к.т.н., доцент каф. ВТ Федюнин Р.Н.

Цель работы:

Разработать игровое приложение на тему «Игровое приложение под *OC Android* в жанре «платформер»».

Приложение должно иметь адаптивный интерфейс для работы на различных устройствах с *OC Android*, обладать отзывчивым и интуитивно понятным управлением, качественным аудиовизуальным сопровождением и при этом быть достаточно оптимизированным.

					ПГУ1.090301.22.001 ПЗ	Лист
						109
Изм.	Лист	№ докум.	Подпись	Дата		

Аналоги



Платформа разработки



Графический редактор



Язык программирования



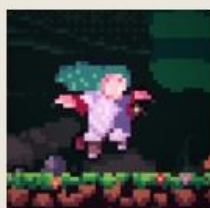
Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.22.001 ПЗ

Лист

111

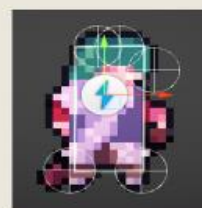
Требования к механикам игры жанра «платформер»



Jump Buffering



Coyote Time



Edge Detection

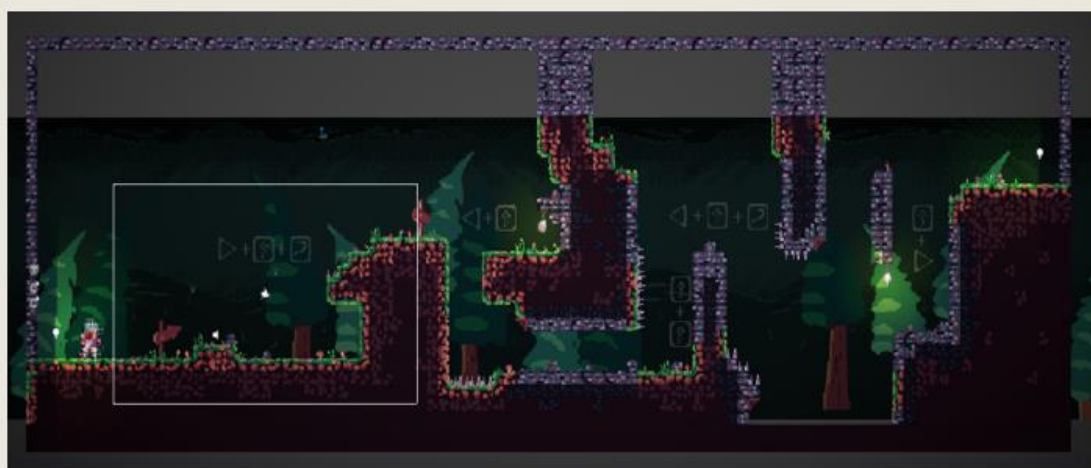


Jump Hang Time



Jump Height

Требования к дизайну уровней



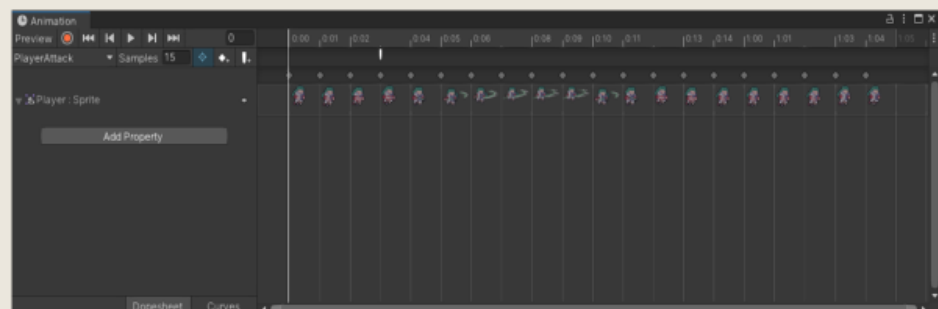
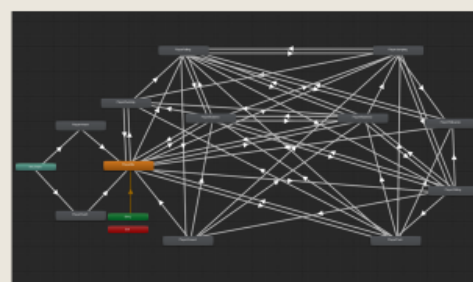
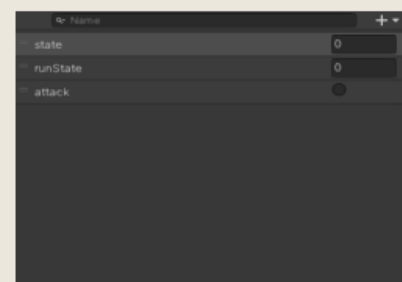
Обучающий уровень

Изм.	Лист	№ докум.	Подпись	Дата

Стилистика



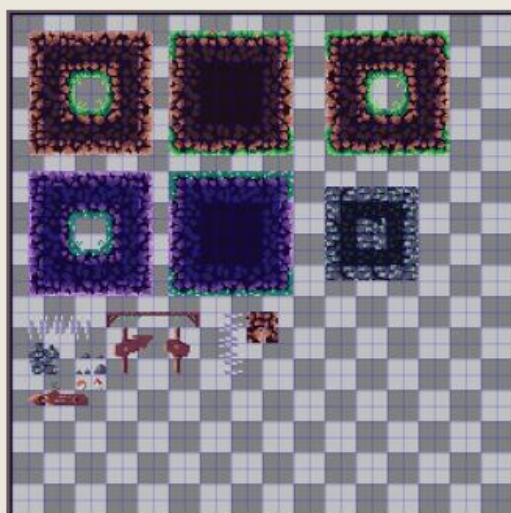
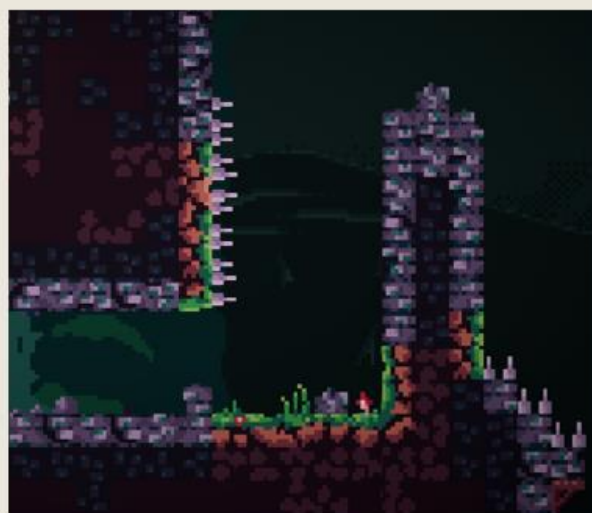
Аниматор и ключевые кадры



Анимации



Tileset



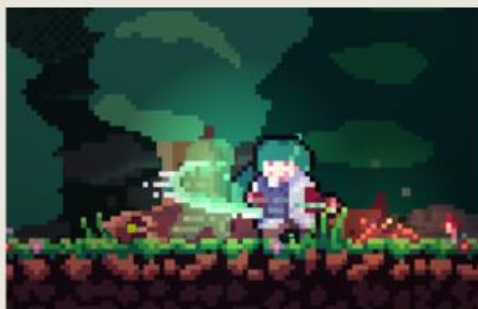
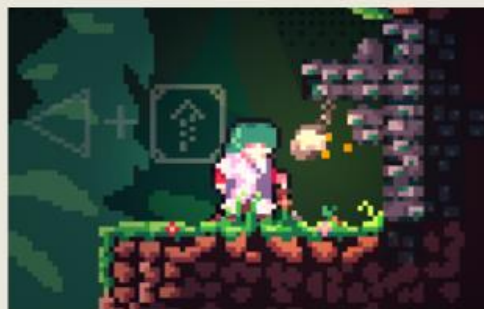
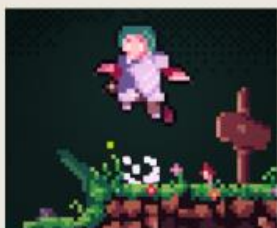
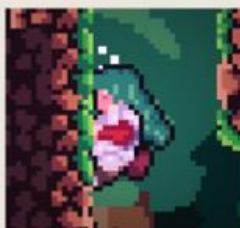
Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.22.001 ПЗ

Лист

114

Визуальные эффекты



Геймдизайн



Изм.	Лист	№ докум.	Подпись	Дата

ПГУ1.090301.22.001 ПЗ

Лист
115

Геймплей



Спасибо за внимание!

					ПГУ1.090301.22.001 ПЗ	Лист
						116
Изм.	Лист	№ докум.	Подпись	Дата		