МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

ДОПУСТИТЬ К ЗАЩИТЕ

Заведующий кафедрой, д.ф.-м.н.,
профессор

__________ Л.Б. Соколинский

«___»___________ 2024 г.

Разработка компьютерной игры в жанре «Платформер» с процедурной генерацией игрового мира на платформе Unity

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 09.03.04.2024.308-355.ВКР

Научный руководитель,
ст. преподаватель кафедры СП
__________ П.Г. Верман

Автор работы,
студент группы КЭ-404
__________ Е.М. Блинова

Ученый секретарь
(нормоконтролер)
_____________ И.Д. Володченко
«___»___________ 2024 г.

Челябинск, 2024 г.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

УТВЕРЖДАЮ
Зав. кафедрой СП
__________ Л.Б. Соколинский
29.01.2024 г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студентке группы КЭ-404
Блиновой Елене Михайловне,
обучающейся по направлению
09.03.04 «Программная инженерия»

1. **Тема работы** (утверждена приказом ректора от 22.04.2024 г. № 764-13/12)

Разработка компьютерной игры в жанре «Платформер» с процедурной генерацией игрового мира на платформе Unity.

2. **Срок сдачи студентом законченной работы:** 03.06.2024 г.
3. **Исходные данные к работе**

3.1. Роллингз Э., Моррис Д. Проектирование и архитектура игр. 2 изд.: Пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 1040 с.

3.2. Unity Documentation. [Электронный ресурс] URL: https://docs.unity.com (дата обращения: 29.01.2024 г.).

3.3. Шорт Т.Х., Адамс Т. Процедурная генерация в гейм-дизайне. // М.: ДМК Пресс, 2020. – 344 с.

4. **Перечень подлежащих разработке вопросов**

4.1. Выполнить анализ предметной области.

4.2. Выполнить обзор алгоритмов процедурной генерации.

4.3. Спроектировать игровое приложение.

4.4. Реализовать игровое приложение.

4.5. Выполнить тестирование приложения.

5. Дата выдачи задания: 29.01.2024 г.

Научный руководитель,
ст. преподаватель кафедры СП П.Г. Верман

Задание принял к исполнению Е.М. Блинова

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 6
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 8
1.1. Обзор игр-платформеров 8
1.2. Обзор игр с механикой сбора предметов 11
1.3. Обзор игр с процедурной генерацией контента 13
1.4. Сравнительный анализ игровых механик 15
2. ПРОЦЕДУРНАЯ ГЕНЕРАЦИЯ 17
2.1. Классификация методов процедурной генерации 17
2.2. Обзор алгоритмов процедурной генерации 19
3. ПРОЕКТИРОВАНИЕ 29
3.1. Требования к игровому приложению 29
3.2. Концепция 30
3.3. Варианты использования игрового приложения 34
3.4. Проектирование игрового генератора 36
3.5. Архитектура игрового приложения 38
3.6. Макеты пользовательского интерфейса 40
4. РЕАЛИЗАЦИЯ 44
4.1. Средства реализации 44
4.2. Структура реализации 45
4.3. Реализация компонента генератора и игрового менеджера 46
4.4. Реализация компонентов шины и комнат 55
4.5. Реализация компонента персонажа 57
4.6. Реализация компонента кристаллов и инвентаря 60
4.7. Реализация компонента платформ 61
4.8. Реализация компонента открываемых объектов 63
4.9. Реализация компонентов звука и фона 64
4.10. Результаты реализации 66
5. ТЕСТИРОВАНИЕ 74
5.1. Тестирование требований 74

5.2. Юзабилити-тестирование 76
5.3. Изменение баланса игры 77
ЗАКЛЮЧЕНИЕ 79
ЛИТЕРАТУРА 80

ВВЕДЕНИЕ

Актуальность

Игровая индустрия является неотъемлемой частью современной культуры. По данным исследования [40], 60% россиян играют в видеоигры регулярно или эпизодически. Игры предоставляют возможности для отдыха и социального взаимодействия, обеспечивая разнообразные формы развлечения. Люди играют в игры, чтобы преодолевать препятствия, выполнять задачи для достижения целей и получить ощущение прогресса и успеха.

Компьютерные игры воплощают в себе не только развлекательный аспект, но и являются полем для творчества. За последние 6 лет (2019-2024 гг.) на платформе Global Game Jam [15] в России было создано 479 игровых проекта, из которых 155 выпущены в Челябинске на базе Южно-Уральского государственного университета.

Актуальным для разработчиков и игроков остается жанр «Платформер». Игры-платформеры имеют низкий порог вхождения для широкой аудитории – в них играют люди разных возрастов, поскольку для этого не требуется специальных навыков. Такие игры рассчитаны как на небольшое количество проведенного времени в игре, так и на долгое прохождение.

Игры позволяют пользователям погрузиться в приключения с новыми сюжетами, мирами. В связи с этим одной из существующих проблем в разработке игр становится достижение реиграбельности – способности взывать желание и интерес у игрока к повторному прохождению игры после первого завершения. Использование процедурной генерации в играх предоставляет разработчикам возможность автоматического создания неограниченного количества уровней с сохранением эффекта неожиданности, что продлевает взаимодействие с игровым приложением.

Среда разработки Unity [33] предоставляет разработчикам возможность создавать игры под различные платформы и операционные системы,

что способствует достижению широкого охвата аудитории, а также создавать различные визуальные эффекты для реалистичной анимации, которая делает игры более привлекательными для пользователей.

Постановка задачи

Целью выпускной квалификационной работы является разработка компьютерной игры в жанре «Платформер» с процедурной генерацией игрового мира на платформе Unity.

Для достижения поставленной цели необходимо решить следующие задачи:

1) провести анализ предметной области;
2) выполнить обзор алгоритмов процедурной генерации;
3) спроектировать игровое приложение;
4) реализовать игровое приложение;
5) протестировать игровое приложение.

Структура и содержание работы

Работа состоит из введения, пяти глав, заключения и списка литературы. Объем работы составляет 85 страниц, объем списка литературы – 58 источников. В первой главе описывается предметная область проекта, анализируются существующие игровые приложения, приводится обзор на средства разработки. Вторая глава посвящена обзору методов и алгоритмов процедурной генерации контента. В третьей главе приводится проектирование игрового приложения, включая требования, игровую концепцию, варианты использования, а также архитектуру. В четвертой главе описывается реализация компонентов архитектуры игрового приложения. Пятая глава посвящена тестированию игрового приложения.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Обзор игр-платформеров

Платформер – это жанр компьютерных игр, в которых игровой процесс основан на перемещении по платформам и лестницам. Большинство подобных игр предназначены для одиночной игры в рамках одного или нескольких уровней. Платформеры относятся к играм с графической перспективой «side-scrolling», где камера следует за персонажем сбоку, по мере его движения по уровню «сверху-вниз» и «справа-налево». Основной группой механик в таких играх является совокупность умений и навыков взаимодействия персонажа с игровым миром. В качестве примеров можно привести: UnEpic, Braid, Celeste, Hollow Knight, Fez, Super Mario Bros.

В Braid [52] игрок перемещается по локациям, используя для этого лестницы, платформы, канаты и другие предметы, расположенные на уровнях (рисунок 1).

Рисунок 1 – Игровой процесс Braid

В Celeste [46] игроку предстоит пройти цепочку локаций, состоящих из последовательности 700 уникальных экранов с различными тайниками. Каждый игровой экран имеет определенный набор особенностей в зависи-

мости от типа локации. Для прохождения игры персонаж применяет комбинацию различных трюков. Геометрия локации состоит из групп тайлов (синие, голубые, зеленые объекты прямоугольной формы на рисунке 2), которые вместе создают структуру игрового экрана и образуют платформы (в том числе и стены) для передвижения. Платформы делятся на доступные и недоступные, столкновение с недоступными является проигрышем и возвращает игрока к стартовой точке экрана.

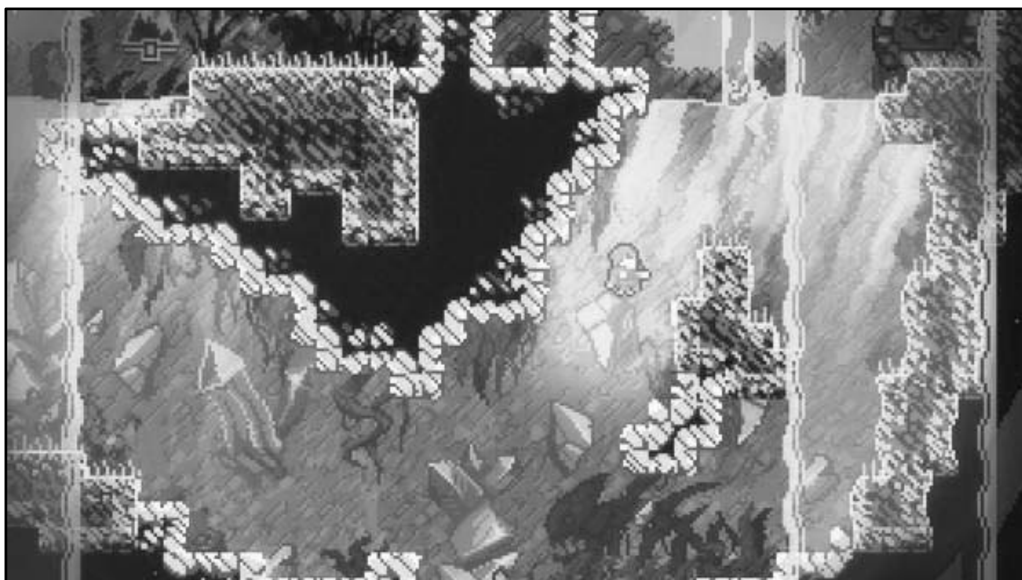

Рисунок 2 – Игровой экран Celeste

В UnEpic [51] главный герой попадает в средневековый замок, в котором ему предстоит сразиться с гоблинами, волшебниками и духами. Во время перемещения по комнатам игрок сталкивается с ловушками и противниками. Игровой процесс построен следующим образом: игрок находит различные артефакты, осваивает новую способность, побеждает босса и получает доступ к следующей части замка, перемещаясь между комнат с помощью дверей.

В Hollow Knight [53] особенностью игры является большой единый игровой мир. Игра содержит элементы платформ (на рисунке 3 показано перемещение персонажа). В верхней части игрового экрана расположен пользовательский интерфейс: показатель здоровья, энергии и счетчик очков.

Рисунок 3 – Элементы платформ в Hollow Knight

В Super Mario Bros [57] персонаж ходит, прыгает по платформам, избегает вражеских перемещающихся грибов, собирает монеты и усилители: супер-грибы, огненные цветы и звезды. Платформы могут разрушаться, отнимать жизни, являться тайниками, в которых спрятаны предметы. В начале игры у персонажа есть одна жизнь. В пользовательском интерфейсе отображается пройденное с момента начала игры время, счетчик очков и монет, номер текущего уровня (рисунок 4), а также количество жизней. Прохождение уровней является последовательным.

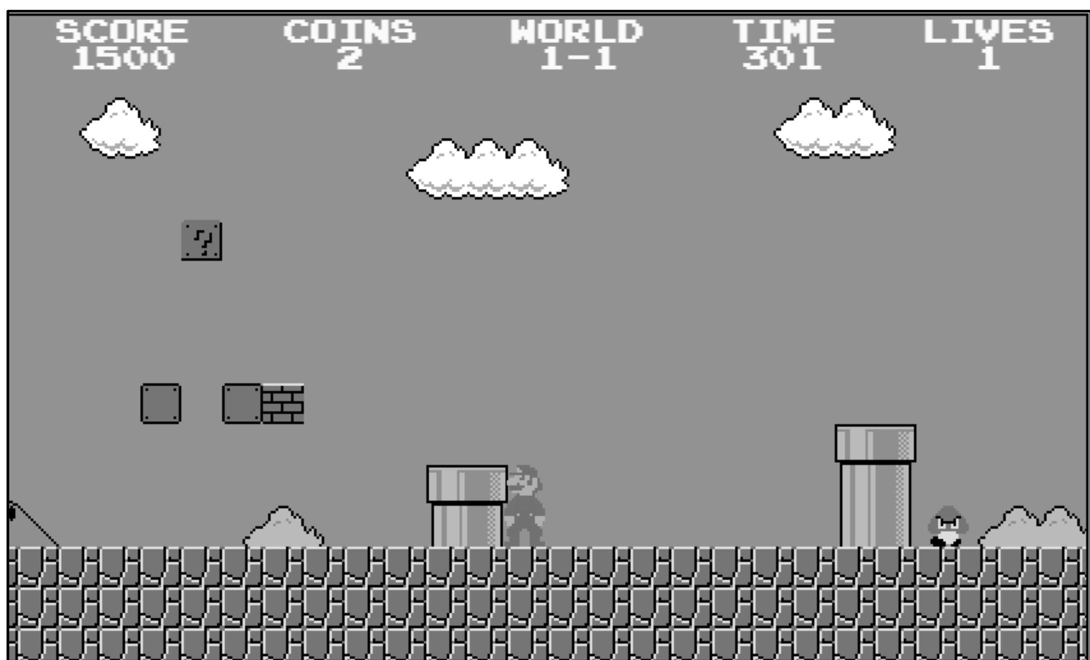

Рисунок 4 – Игровой процесс в Super Mario Bros

Особенностью игры Fez [48] является представление по четырем 2D видам 3D мира с помощью механики вращения, которая позволяет просматривать уровень и соединять таким способом недоступные пути. Игровой мир (рисунок 5) включает в себя: неигровых персонажей, квесты, предметы для сбора, статичные и подвижные платформы, лестницы, двери.

Рисунок 5 – Игровой процесс в Fez

1.2. Обзор игр с механикой сбора предметов

Помимо ранее рассмотренных механик перемещения персонажа, особенностей представления игровых миров и взаимодействия с ними игрока, существует группа механик, связанных со сбором предметов. Данные механики способствуют заинтересованности игрока в изучении мира, являются основой различных квестов, определяют скорость прохождения. Для реализации данной группы механик применяется инвентарь – элемент пользовательского интерфейса. В качестве примеров можно привести такие игры-платформеры, как Fez, Banjo Panda, Shovel Knight.

В Banjo Panda [56] персонаж собирает ананасы. Игроки строят платформы и мосты, чтобы помочь пандам добраться до ананасов, которые используются для обмена на новые предметы усиления, обретения новых способностей или изменения внешности персонажа.

В Fez одной из основных игровых целей является сбор объектов «Золотой куб». Сбор золотых кубов отображается в виде заполнения слотов на отдельном экране (рисунок 6). Для этого игроку необходимо собирать антикубы с помощью решения головоломок, находить артефакты, для открытия достижений, собирать «письменные кубы» для прохождения игровой истории и продолжения сюжета.

Рисунок 6 – Интерфейс счетчика золотых кубов в игре Fez

Shovel Knight [50] представляет собой 2D платформер с восьмибитной графикой. Игрок собирает золото и реликвии: золото используется для улучшения брони и оружия, а реликвии дают уникальные способности. На рисунке 7 приведен скриншот игрового экрана и инвентаря. В верхней части пользовательского интерфейса расположена статистика игры: количество собранного золота (1706), реликвий (140), количество оставшихся жизней представлено красными кружками, а также количество боссов, которых осталось пройти. В центральной части меню инвентаря расположены по секциям реликвии (Relics) и снаряжения (Gear). Секция реликвии заполнена предметами, представленными иконками. В нижней части расположено описание выбранной реликвии и ее численное значение.

Рисунок 7 – Инвентарь из игры Shovel Knight

1.3. Обзор игр с процедурной генерацией контента

Процедурная генерация контента представляет собой механику автоматического создания элементов игрового мира, повышая таким образом уровень интереса игрока. В качестве примера рассмотрены следующие игры: The Binding of Isaac, Darkest Dungeon, Rogue Legacy.

В The Binding of Isaac [54] при каждом новом запуске игры меняется расположение подземелий. На уровне есть комнаты с торговцем, сокровищем и боссом (пример подземелья изображен на рисунке 8). Особенностью игры является случайное добавление секретных комнат и комнат трех уровней сложности, которые сочетаются с двумя типами подземелий: простые и средние комнаты, средние и сложные комнаты.

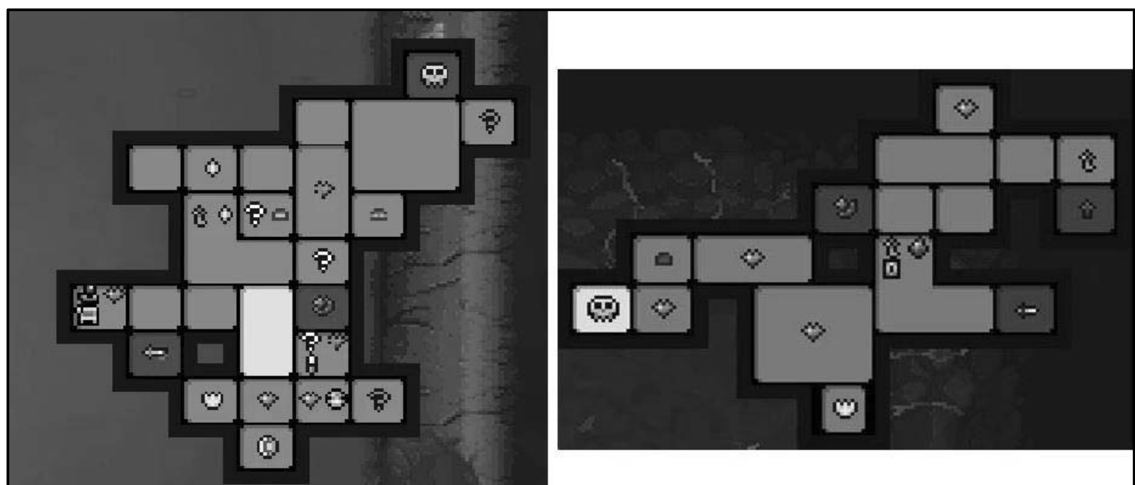

Рисунок 8 – Подземелья в The Binding of Isaac

В Darkest Dungeon [47] процедурная генерация используется для создания подземелий. В начале создается карта, состоящая из квадратов, являющихся комнатами. В комнате генерируется одно из событий: пустая комната, сундук с наградой, сражение, сражение с наградой. В дальних комнатах генерируются обычные и особые боссы. Следующим этапом создаются переходы между комнатами. В переходах с определенной вероятностью создается событие: сбор игровых объектов, ловушки или сражения. На рисунке 9 приведена игровая карта.

Рисунок 9 – Карта подземелий Darkest Dungeon

В Rogue Legacy [49] процедурная генерация работает следующим образом. Осуществляется генерация комнат: случайным образом происходит выбор точек на карте для размещения заранее спроектированных комнат. Каждая спроектированная комната имеет заданное количество врагов и их начальное размещение. Затем осуществляется создание переходов: гарантируется, что каждая комната будет иметь соединение хотя бы с одной другой комнатой. На рисунке 10 приведена карта подземелья.

Рисунок 10 – Карта подземелья в Rogue Legacy

1.4. Сравнительный анализ игровых механик

Для создания игры в жанре платформер были рассмотрены следующие группы механик: перемещения персонажа и его взаимодействия с игровым миром, сбора предметов, их хранение и применение, процедурной генерации контента.

Большая часть игр в жанре платформер, связанных с механикой сбора предметов, содержит в пользовательском интерфейсе инвентарь или информацию о текущем состоянии игры. При этом взаимодействие с игровым миром преимущественно происходит либо через элементы сражений, либо через решение головоломок и прохождения препятствий.

Представление игрового мира в рассмотренных проектах является либо закрытым, либо полуоткрытым. Закрытый игровой мир подразумевает собой последовательную цепочку уровней, при этом перемещение на предыдущие уровни либо невозможно, либо не имеет смысла. Полуоткрытый игровой мир представляет собой сеть из доступных и недоступных областей для свободного перемещения между уровнями, локациями, этажами.

Процедурная генерация представляет возможность для автоматического создания различных элементов игрового мира. В том числе для размещения объектов внутри игрового мира.

На этапе проектирования приложения игровая концепция для игры в жанре платформер будет состоять из следующих основных частей:

1) полуоткрытого мира, в котором игрок сможет исследовать игровое пространство;

2) создания платформ с различными механиками поведения, оказывающих влияние на способ перемещения персонажа;

3) создания нескольких типов локаций, каждая процедурно сгенерированная локация будет являться отдельной комнатой;

4) предметов для сбора, которые будут храниться в инвентаре и использоваться для перемещения между комнатами.

Также было принято решение не включать в концепцию вспомогательные квесты и элементы сражения с целью сосредоточить внимание игрока на задаче распределения ресурсов для перемещения между комнатами. Таким образом игрок сможет сосредоточиться на основных механиках жанра платформер: прыжках, перемещении и навигации по комнатам. Отсутствие вспомогательных квестов и боевых элементов упрощает игровой процесс, делая игру более доступной для всех возрастов, а также поддерживает быстрый темп игры.

Выводы по первой главе

Был проведен обзор существующих решений по трем категориям проектов. Был выполнен сравнительный анализ игровых приложений по основным механикам и особенностям, а также приведены основные элементы для проектирования игровой концепции.

2. ПРОЦЕДУРНАЯ ГЕНЕРАЦИЯ

Из-за обширного применения способов и возможностей процедурной генерации контента (ПГК) возникает проблема проектирования компонентов и назначения применения генераторов. В работе [55] выделяется два подхода в постановке задачи процедурной генерации контента.

Первый заключается в том, чтобы создать генератор, определив: область его действия, функции, области допустимых и недопустимых результатов генерации. Данный подход позволяет определить все объекты генерации, а также определить качества «идеального» результата генерации.

Второй подход заключается в том, чтобы использовать определенный метод ПГК до тех пор, пока не будет получен объект генерации, приближенный к желаемому результату. Для этого за основу берутся ранее созданные проекты, дизайнер определяет комбинацию генерируемых объектов, после чего происходит пересмотр существующих решений на основе полученных результатов.

2.1. Классификация методов процедурной генерации

В работе [55] была предложена следующая классификация видов ПГК: методы распределения, параметрические методы, методы на основе плиток, методы на основе формальных грамматик, использование решателей ограничений, методы на основе агентов и моделирования.

Методы распределения позволяют распределять элементы объекта генерации в области пространства или времени. Эти методы основаны на конкретных правилах и параметрах, которые устанавливаются исходя из игровой концепции и баланса игры.

Методы параметризации позволяют изменять как до начала, так и во время игрового процесса параметры и характеристики существующих объектов, повышая тем самым адаптивность игрового мира исходя из действий игрока.

Методы на основе плиток представляют результат в виде случайных или упорядоченных наборов, а также позволяют получать различные выборки сочетаний существующих объектов в определенные группы. Объекты представляют собой модульные элементы одинакового размера (характеристик), из которых состоит результат генерации.

Методы на основе концепции формальной грамматики подразумевают использование созданной системы правил для определения, как могут формироваться или расширяться структуры (результат генерации), состоящие из дискретных единиц (строк символов, графов, двумерных массивов). Каждый символ имеет набор подсимволов, из которых он может быть составлен. Во время генерации происходит рекурсивный выбор символов и подсимволов, пока не будет получена конечная строка, представляющая определенный набор элементов и свойств результирующего объекта генерации.

Методы, использующие решатели ограничений, осуществляют поиск решения из большого набора ограничений и правил с условиями. Задача описывается в формате «переменная-условие», а решатель ограничений подбирает решение, которое будет подходить под все требования.

Методы на основе агентов и моделирования используют имитацию естественных процессов с помощью заранее спроектированных абстрактных сущностей и правил их взаимодействия.

По применению методы ориентированы на определенный стиль графической перспективы, а также могут быть разделены по назначению генерации: создание сценария и игровых правил, создание подземелий разного вида, создание карт ландшафта, создание персонажей, размещение игровых элементов (предметы, противники, ресурсы и т.д.). При этом, алгоритмы из одной категории могут быть применимы и для других, поскольку окончательное решение в выборе алгоритма принимаются исходя из особенностей поставленной для реализации задачи. Так, например, комнаты с 2D видом

сверху и коридоры между ними при изменении на вид сбоку становятся помещениями, соединенными лестницами или иными переходами. Кроме того, алгоритмы могут относится к нескольким видам методов и могут комбинироваться при решении задачи ПГК.

2.2. Обзор алгоритмов процедурной генерации

Рассматриваемые в обзоре алгоритмы разбиты по следующим категориям: для игр с видом сверху и с видом сбоку. При этом, алгоритмы из одной категории могут быть применимы и для других, поскольку окончательное решение в выборе алгоритма принимается исходя из особенностей поставленной для реализации задачи.

Игры с графической перспективой «Top-Down»

Представляют игроку обзор отображения персонажей и объектов с высоты, перемещение игрока осуществляется по направлениям «вперед-назад-влево-вправо» Для создания игровой территории применяются такие алгоритмы, как алгоритмы создания лабиринтов туннелирования на основе агентов и с помощью случайного блуждания [12].

Алгоритм Эйлера позволяет создать игровое пространство путем обхода каждой ячейки игрового поля, определяя ее назначение, при этом гарантируется наличие минимум одного пути от начала до конца, то есть создание лабиринта. Алгоритм поиска в глубину (Depth First Search, DFS) [8] является еще одним способом генерации лабиринта.

Алгоритм туннелирования [17] является алгоритмом процедурной генерации, представляющий собой процесс автоматического создания коридоров и комнат. Существует комбинирование нескольких алгоритмов на разных этапах генерации. В работе [9] создание подземелья осуществляется следующим образом. Первым шагом в подземелье предварительно размещается произвольное количество готовых единиц пространства: стены, открытые области, комнаты, в которых могут быть размещены неигровые пер-

сонажи для создания квестов и сюжетов. Затем популяции агентов «Строителей» строят подземелья вокруг заранее размещенных объектов. Далее агенты «Строителей» подкласса «Ползуны» строят стены на открытом фоне и порождают себе подобных. На закрытом фоне работают агенты «Тунелеры», которые порождают агентов для создания комнат. При этом каждый строитель относится к своему поколению. Поколение потомка начинает работу только после вымирания поколения родителя. Строитель умирает по двум причинам: закончилось пространство или они достигли максимального количества шагов, за которое возможно построить стену или тоннель, изменить направление или создать одного или двух потомков.

Пример реализации алгоритма для создания игровой карты приведен на рисунке 11. Яркие цвета показывают, где расположены большие сокровища и скопления неигровых персонажей.

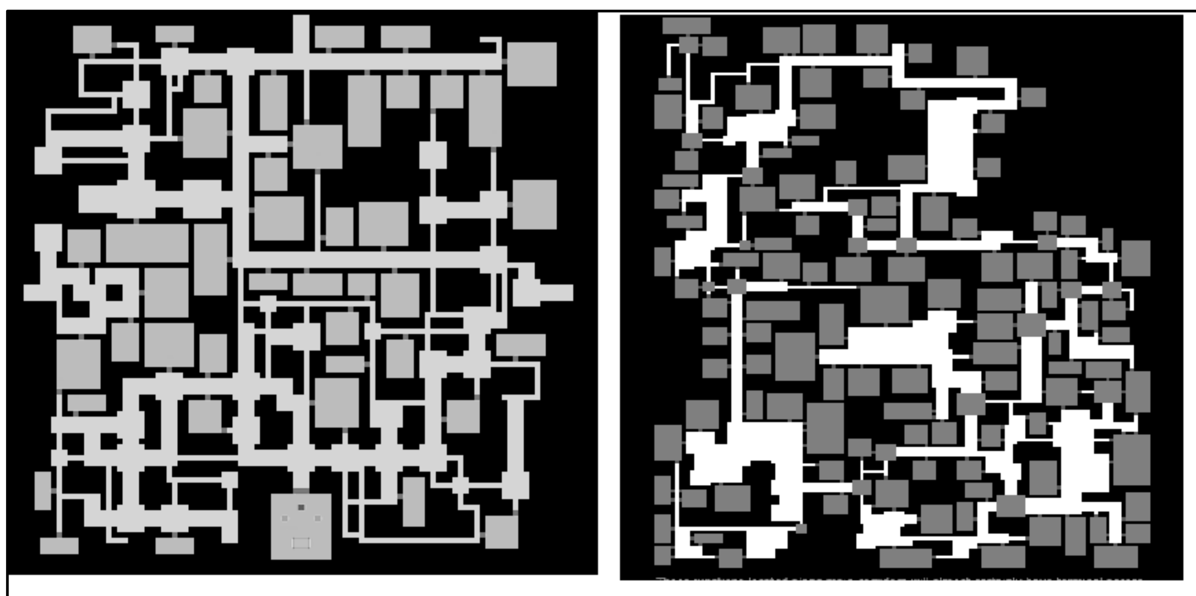

Рисунок 11 – Реализация карты комбинацией алгоритмов туннелирования и агентов

Двоичное разбиение пространства (Binary Space Partitioning) [14] является рекурсивным алгоритмом процедурной генерации методом разделения области на меньшие. Алгоритм описывается следующим образом. Со-

здается шаблон, представляющий собой прямоугольник с дополнительными параметрами, каждый прямоугольник хранит информацию о коридорах. Создается начальный прямоугольник – область для разбиения. Пока есть возможность, производится рекурсивное разбиение области по вертикальному или горизонтальному направлению разбиения. Внутри каждой области, которая более не подлежит разбиению, создается комната произвольного размера в пределах области. Производится соединение комнат коридорами для каждой области с двумя дочерними комнатами.

Итоговая структура представляет готовый уровень, включающий в себя несколько областей. Пример изображен на рисунке 12. Параметры размеры комнат, разбиения являются настраиваемыми для получения разнообразных результатов, а также достижения необходимых уровней сложности. При этом, комнаты с видом сверху могут стать регионами для заполнения с видом сбоку.

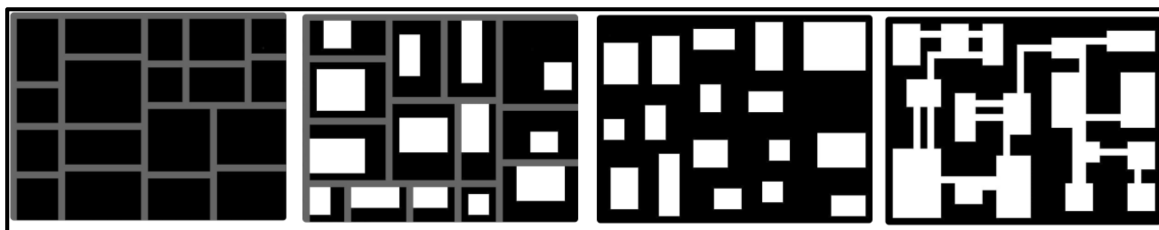

Рисунок 12 – Реализация алгоритма двоичного разделения пространства

Алгоритм случайного блуждания (Random Walk Algorithm) используется для моделирования случайного процесса, в котором объект перемещается на каждом шаге в случайном направлении. Алгоритм позволяет создавать пещеры и ландшафты для двумерных карт. Суть работы алгоритма заключается в следующем. Создается сетка заданного размера, в которой каждая ячейка начинает с нулевым значением, обозначающим не исследованное состояние. Выбирается случайная начальная позиция на сетке. На каждом шаге перемещения выбирается случайное направление для перемещения текущей позиции влево, вправо, вверх или вниз. Осуществляется проверка,

находится ли новая позиция в пределах граница сетки. Новая позиция становится текущей, ячейка обозначается исследованным состоянием. Процесс повторяется заданное количество итераций.

Пример реализации алгоритма приведен на рисунке 13. Слева представлен вариант с 65535 итерациями, слева с 524280 количеством итераций.

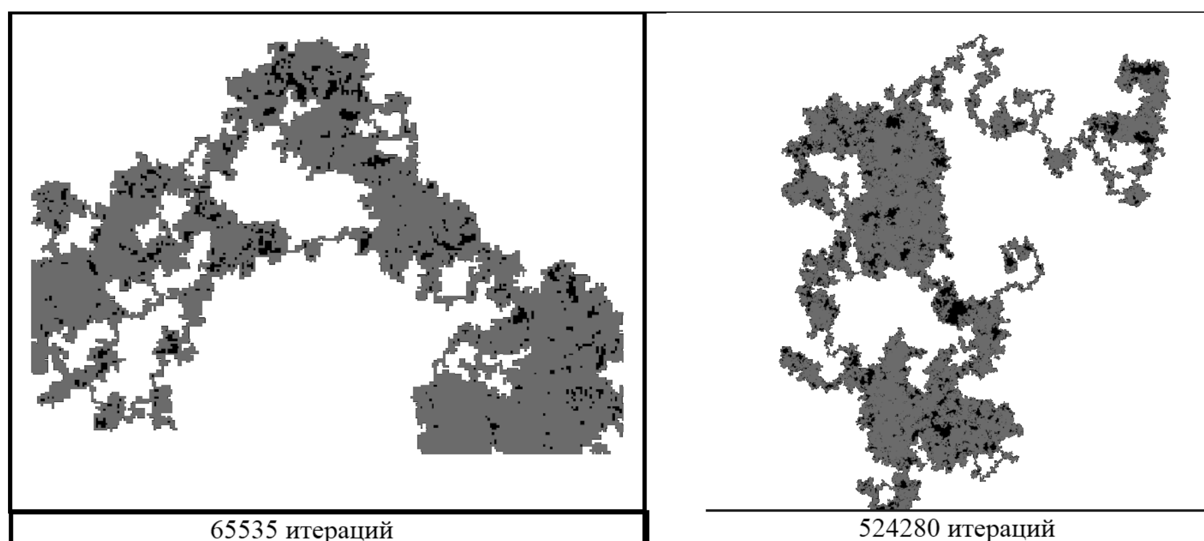

Рисунок 13 – Результат работы алгоритма случайного блуждания

Похожим на алгоритм случайного блуждания является реализация подземелья в игре The Binding of Isaac (рисунок 8), которое генерируется следующим образом [41]. Осуществляется инициализация пространства с заданными параметрами. Создается пустая сетка размером 9 на 8, каждая ячейка получает номер ее координаты («01», «02», и т.д.). Происходит размещение начальной комнаты в ячейке «35», ячейка добавляется в очередь. Осуществляется рост подземелья в ширину. Для каждой ячейки происходит циклический обход в 4-х направлениях для определения соседней ячейки. Если ячейка является пустой и соответствует критериям, то она добавляется в очередь, иначе она добавляется в список конечных комнат. Осуществляется проверка количества комнат на уровне, а также проверка на то, чтобы комната с боссом не находилась рядом со стартовой. Комната босса размещается в последней ячейке списка конечных комнат. Секретная комната раз-

мещается в соответствии с заданными критериями. Обычные комнаты выбираются в соответствии с уровнем сложности и текущим игровым этапом. Происходит заполнение комнат ловушками, монстрами, декорациями. Преимуществом данного алгоритма является регулировка уровня сложности игры при сравнительно несложной реализации.

Игры с графической перспективой «Side-Scrolling»

Представляют игроку обзор отображения персонажей и объектов с учетом продвижения игрока слева направо и снизу вверх. Для создания территории используются такие алгоритмы, как клеточные автоматы [1], алгоритм размещения меток на основе сетки [36]. Универсальным решением является ПГК с помощью коллапса волновой функции [35].

Клеточные автоматы (Cellular Automata) используются для генерации ландшафта с помощью сетки клеток. Состояние каждой клетки обновляется по заданным правилам на основе состояний ее соседей. Создается сетка, представляющая игровой уровень. Определяется набор правил для управления состояниями клеток, условие остановки алгоритма или завершающее число итераций. Сетка заполняется случайным образом различными типами состояний клеток (полы, стены). Применяются заданные правила, по которым формируются новые состояния для всех клеток. На рисунке 14 приведен пример того, какие пространства можно получить данным алгоритмом, меняя входные параметры.

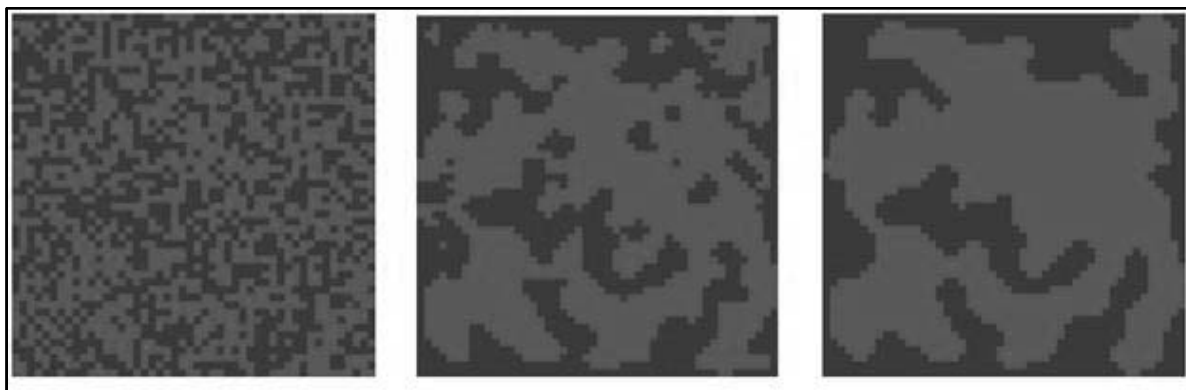

Рисунок 14 – Результат работы клеточного автомата

В статье приведен пример использования алгоритма клеточных автоматов для генерации с использованием дополнительных итераций для создания переходов, чтобы карта представляла собой единую локацию.

Алгоритм размещения меток на основе сетки позволяет выполнять размещение платформ и других объектов на заданной территории. Принцип работы алгоритма заключается в следующем. Задается начальное количество узлов. Вычисляются потенциальные позиции меток путем разбиения игрового поля на сетку клеток заданного размера. Выполняется ранжирование меток в соответствии с расстоянием до объектов. При необходимости выполняется оценка расстояния от каждой метки до ближайших объектов и сортировка оценок по возрастанию расстояния. Выполнение ограничений границ, в следствии чего каждая метка не должна выходить за пределы игрового мира. На рисунке 15 приведены конечные результаты работы алгоритма в зависимости от заданных количества узлов: от 25 до 500.

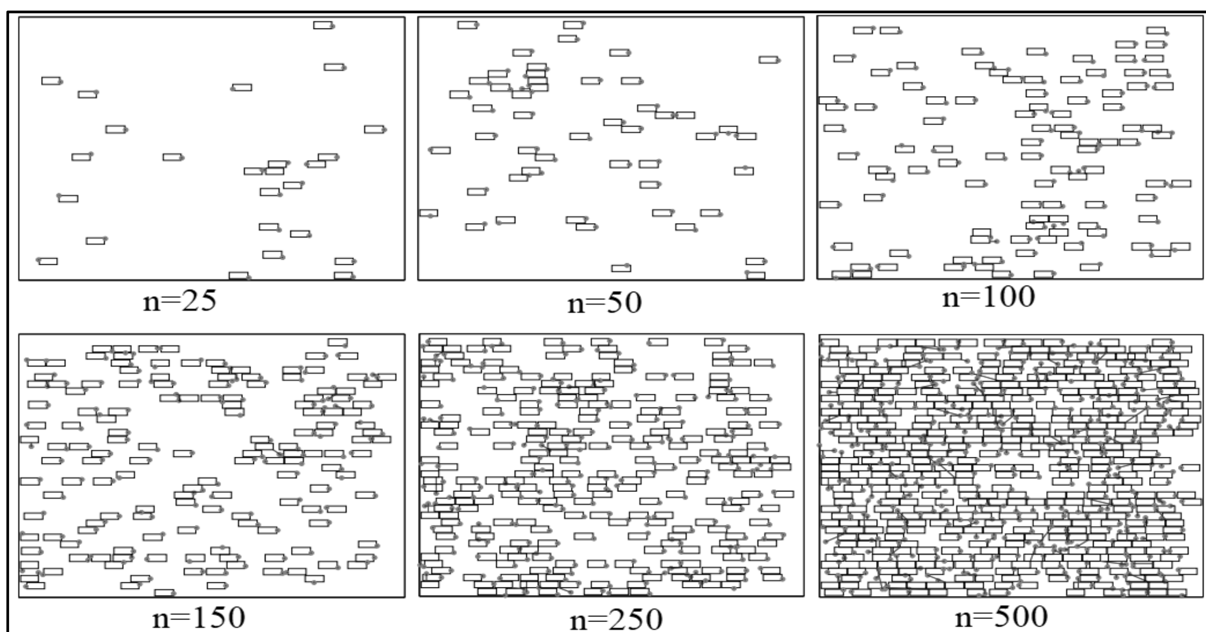

Рисунок 15 – Размещение заданного количества меток внутри сетки

Дерево поведения (Behavior Trees) [26] является алгоритмом для управления процедурной генерацией с помощью дерева структурированных узлов. Дерево поведения состоит из: корневого узла, с которого начинается

выполнение алгоритма, внутренних узлов, определяющих направление поведения, листьев, определяющих действия для выполнения, тактов (сигналов), определяющих передачу управления от узлов.

Суть использования деревьев поведений в процедурной генерации состоит в описании логики генерации уровня и конкретных действий для размещения сегментов уровня. Пример генерации уровня с помощью данного алгоритма приведен на рисунке 16.

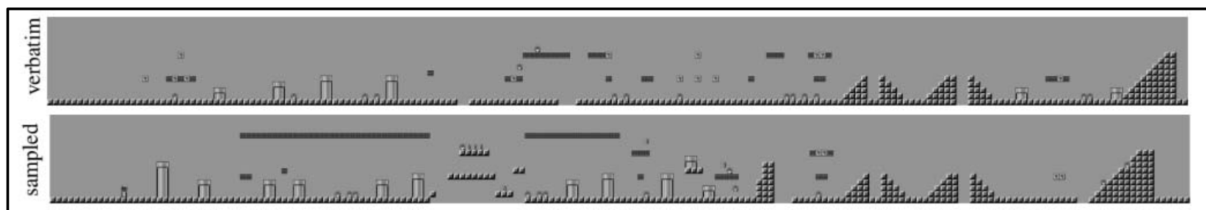

Рисунок 16 – Процедурная генерация уровня деревом поведений

В размещение объектов и ресурсов с учетом игровой логики применяется алгоритм дискретизации диска Пуассона [23]. Это метод генерации точек в пространстве, который обеспечивает равномерное распределение с минимальным расстоянием между ними. Принцип генерации основан на вычислении нормализованного радиуса дисков Пуассона. Создается случайное распределение точек на плоскости. Для точек вычисляется оптимальный радиус дисков на основе их распределения в пространстве. Пространство делится на ячейки, для каждой ячейки вычисляется максимально вписанный круг вокруг центра ячейки.

Рисунок 17 иллюстрирует последовательные улучшения, достигнутые итерациями в работе [23]: красные точки обозначают текущее положение, а черные – показывают центр дисков.

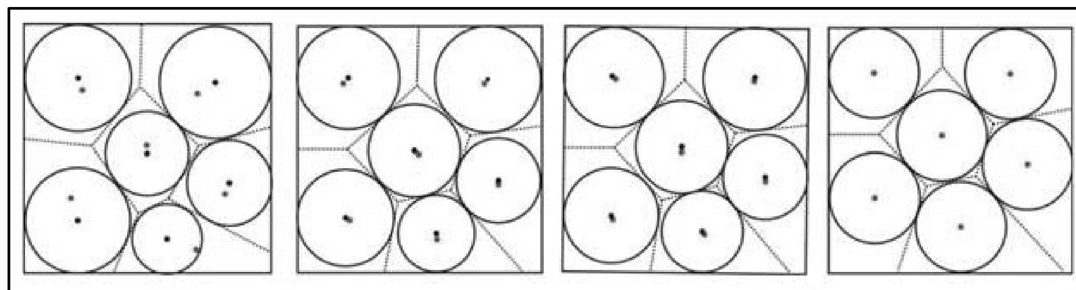

Рисунок 17 – Процесс упаковки дисков Пуассона

Процедурная генерация с помощью коллапса волновой функции (Wave Function Collapse) позволяет генерировать текстуры, изображения, уровни или другие элементы игрового мира на основе образцов. Принцип работы заключается в следующем. Инициализируется пространство для заполнения и количество итераций. Задаются входные образцы в виде небольших участков данных. Применяется коллапс волновой функции к исходным образцам для создания новых данных. Осуществляется распространение волны по пространству с выбором наиболее вероятного образца. Осуществляется обновление соседних образцов и их вероятностей. Получается новый набор данных по исчерпанию количества итераций. Пример итераций приведен на рисунке 18.

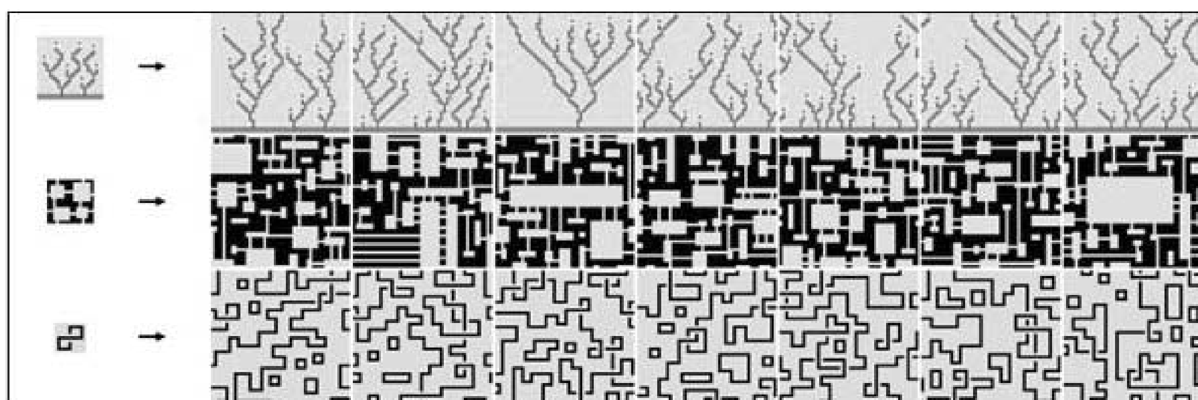

Рисунок 18 – Пример сгенерированных изображений волновой функцией

Алгоритм коллапса волновой функции применяется также для создания плиточных полей с помощью байесовского априорного распределения вероятностей [37].

Алгоритмы создания игровых уровней

Алгоритм контекстно свободной грамматики [10] применим в рамках проверки правил для сочетания существующих событий, а также для генерации квестов и сюжетных линий.

Алгоритмы генерации игровых уровней и окружения на основе алгоритмов Прима (Prim's Algorithm) [25] и Краксала (Kruskal's Algorithm) [16],

а также алгоритмов на основе случайного графа позволяют создать связанное игровое пространство. Были рассмотрены следующие алгоритмы создания графов.

1. Алгоритм создания графа (Random Graph [31]), в котором каждая пара вершин соединена ребром с определенной вероятностью.
2. Алгоритм создания графа на основе моделей малого мира (Small-World Graph) [30]. В данном графе большое количество вершин связаны с близкими соседями по наименьшему пути между вершинами.
3. Алгоритм создания графа на основе модели Барабаши-Альберта (Barabasi-Albert Model) [3]. В данном методе генерируется граф, в котором новые вершины присоединяются к существующим вершинам с вероятностью, пропорциональной их степени связи.
4. Алгоритм создания графа на основе регулярной решетки (Grid-Based Graph) [13].
5. Способ создания графа на основе минимального дерева (Minimum Spanning Tree) [19]. Метод подразумевает поиск подграфа, содержащий все вершины, но не содержащий циклов и имеющий минимальную сумму весов ребер.

Алгоритмы создания ландшафтов

Одним из способов использования распределения вероятностей является алгоритм на основе Шума Перлина (Perlin Noise) [58]. Алгоритм создает непрерывные формы, поэтому подходит для создания естественных элементов, таких как горы, холмы, долины, и другие непрерывные формы. Принцип работы алгоритма шума Перлина для генерации ландшафта заключается в следующем. Создается заданных размеров карта ландшафта. Осуществляется настройка параметров частоты, амплитуды, октавы и смещения. Для каждой точки карты применяется шумовая функция Перлина для задания значения уровня шума. Для каждой точки ландшафта с помощью значения шумовой функции присваиваются высоты с помощью установки пороговых значений для определения размещения разных элементов ландшафта,

такие как горы, холмы, равнины. Осуществляется визуализация полученных результатов.

На рисунке 19 приведен пример процедурной генерации ландшафта с помощью функции Перлина. Автором работы было создано 8 регионов.

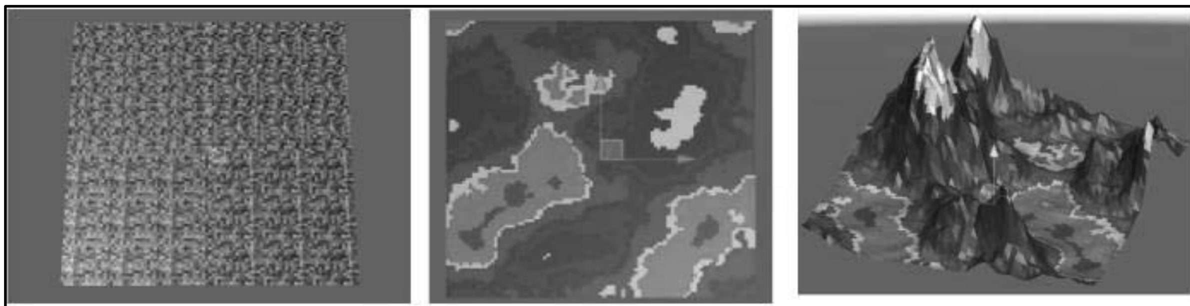

Рисунок 19 – Реализация генерации с помощью шума Перлина

Выводы по второй главе

Были рассмотрены 6 видов классификации методов процедурной генерации контента. Далее были рассмотрены алгоритмы ПГК для игр с видом сверху, сбоку, применение алгоритмов создания графов для генерации связанного игрового окружения, а также способы размещения объектов на игровой территории.

Было принято решение, что для создания генератора будет использоваться комбинация следующих алгоритмов:

1) построения графа по модели Барабаши-Альберта, поскольку он прост в реализации;
2) дискретизации диска Пуассона для размещения ресурсов;
3) алгоритм размещения меток на основе сетки;
4) алгоритм поиска в глубину для создания лабиринта;
5) двоичного разбиения пространства.

Таким образом, для реализации желаемого генератора будут использованы 5 алгоритмов, которые соответствуют основным элементам, определенными в выводе первой главы для создания игровой концепции.

3. ПРОЕКТИРОВАНИЕ

3.1. Требования к игровому приложению

В ходе проектирования были определены следующие функциональные требования [38].

1. Игровое приложение должно иметь стартовое меню, которое предоставляет пользователю возможности: начать игру, изменить пользовательские настройки, выйти из игры.

2. Игровое приложение должно процедурно генерировать: локации, переходы между локациями, размещение объектов на локации.

3. Игровое приложение должно предоставлять игроку три уровня сложности.

4. Игровое приложение должно предоставлять игроку возможность управления персонажем с клавиатуры.

5. Игровое приложение должно воспроизводить фоновую музыку и звуковые эффекты при действиях персонажа и взаимодействии с объектами.

6. Игровое приложение предоставляет следующие пользовательские настройки: изменение громкости фоновой музыки и звуковых эффектов, а также изменение размеров экрана.

7. Игровое приложение должно предоставлять игроку при запуске игрового процесса справку с информацией об управлении и конечной игровой цели.

В ходе проектирования были определены следующие нефункциональные требования [38].

1. Игровое приложение должно функционировать на компьютерах операционной системы Windows 10 и Linux.

2. Игровое приложение должно поддерживать частоту кадров в 60 кадров в секунду (fps) для обеспечения плавности и качества графики.

3. Игровое приложение должно использовать уровень совместимости API Net Standard 2.1.

3.2. Концепция

«Возвращение цвета» – это 2D платформер, действие которого разворачивается в черно-белом мире, лишенном цвета. Игроки будут исследовать процедурно генерируемые локации и находить переходы между экранами, при этом восстанавливая цвет мира.

Сюжет

Игрок видит мир в черно-белых тонах, пока не восстановит цвет. По мере прохождения игры, элементы игрового мира становятся цветными. Игрок исследует процедурно генерируемые локации, каждая из которых имеет визуальные и игровые особенности. Игрок собирает кристаллы разных цветов, открывает двери в новые комнаты.

Атмосфера игры

Основной экран игры представляет собой двухмерное изображение темного мира. Каждая комната принадлежит одному из четырех стилей оформления: зеленый лес, огненная земля, морские глубины и воздушный замок. Персонаж является спрайтом – двумерным изображением, представленным в виде цветной фигуры.

Интерфейс

Игрок управляет персонажем, перемещая его по платформам в комнате. Перемещение персонажа, сбор кристаллов, активаций дверей осуществляется с помощью клавиш управления.

В игре имеется инвентарь, который также управляется с помощью клавиатуры. В инвентаре отображаются собранные игроком кристаллы и особые предметы для открытия сундука, а также текущий процент прохождения уровня. Вся игровая область представлена на главном экране. По мере перемещения персонажа по уровню, камера плавно следует за ним, показывая игроку новые области.

Когда персонаж попадает в область двери или сундука, интерфейс отображает пользователю состояние этого объекта: если объект был открыт,

то пользователь видит подсказку, нажатие на какую кнопку приведет к использованию объекта, иначе игрок видит количество кристаллов, необходимых для открытия. У дверей есть дополнительная подпись, в номер какой комнаты она ведет.

Начало игры

Изначально все двери, фон, платформы являются серыми. Чтобы открыть дверь и получить особый предмет, необходимо собрать указанное в подсказке количество кристаллов. По умолчанию в начале игры в инвентаре выбран красный кристалл. Игрок начинает игру с комнаты с индексом «0», в которой всегда расположен сундук. В начале игры игрок может подойти к сундуку и посмотреть, сколько особых предметов ему нужно собрать, чтобы открыть его и пройти игру.

Цели игры

Цель игры может меняться в зависимости от предпочтений игрока. В общем случае цель игры представляет собой открытие сундука в стартовой комнате. Открытие сундука ведет к завершению игры. Выделяется следующая комбинация общих целей: максимизация набираемых очков со всех комнат, достижение полного восстановления цвета во всех посещенных комнатах. По завершении игры пользователь видит результаты по всем посещенным и не посещенным комнатам.

Объекты игры

Объектами игры являются: персонаж, платформы, кристаллы, двери, особый предмет и сундук. Также к элементам игрового мира относятся: фон, музыка и инвентарь.

Персонаж – главный герой игры, которым управляет игрок. Он может перемещаться по платформам, собирать кристаллы и использовать их для открытия дверей.

Платформы – основные элементы игрового мира, по которым перемещается персонаж. Есть различные типы платформ: статичные и не-статичные.

Двери – переходы между уровнями игры. Двери открываются с помощью кристаллов. Тип локации следующего уровня, на который можно попасть, зависит от типа выбранной двери. При этом дверь открывается кристаллом типа текущей локации, либо той, в которую ведет дверь.

Кристаллы – цветные объекты, которые персонаж собирает. Всего 4 типа кристаллов соответствующие своим локациям.

Двери, фон и платформы находятся в одном из двух состояний: цветное (активное) и бесцветное (неактивное).

Сундук – всегда цветной предмет, находящийся в игровом мире в одном экземпляре в стартовой комнате.

Игровые возможности

Сбор кристаллов: кристаллы являются основным ресурсом в игре. Игрок может собирать кристаллы разных цветов в зависимости от типа той локации, которой принадлежит игровой уровень, при этом все типы кристаллов находятся в инвентаре. Собранные кристаллы приносят игроку очки и увеличивают процент прохождения текущей комнаты.

Правила

Основным игровым правилом является процент прохождения. Каждый уровень имеет процент прохождения (от 0 до 100%). Этот процент повышается, когда игрок собирает кристаллы. Платформы и двери становятся активными, когда достигнут определенный порог прохождения уровня. Для платформ он распределен равномерно от 0 до 100.

Игрок может проиграть только на локации «Облачное пространство», если упадет за нижнюю границу платформ. После этого игра завершается без результатов прохождения игры.

Локации

Всего есть 4 типа локаций: огненные земли, зеленый лес, облачное пространство, морские глубины. Каждая локация содержит красные, зеленые, светло-голубые и синие кристаллы и двери соответственно. Препят-

ствиями на каждом типе локаций являются не-статичные платформы: изменяющие направление гравитации персонажа, подвижные, исчезающие, меняющие направление движения персонажа.

Комнаты

Комнаты в игре представляют собой отдельные процедурно сгенерированные локации. Некоторые уровни могут быть более открытыми, предоставляя игроку возможность свободного перемещения и сбора кристаллов, в то время как другие могут быть более сложными для прохождения. В зависимости от выбранного уровня сложности, требования к количеству собираемых кристаллов, необходимых для открытия дверей, а также количество не-статичных платформ могут варьироваться.

Процедурная генерация

Сперва игроком выбирается уровень сложности от 1 до 3, где 3 – самый сложный. Затем в зависимости от уровня сложности генерируется общее представление игрового мира, которое представляет собой определенное количество комнат, которые относятся к одному из четырех типов локации. Далее генерируется количество объектов особого предмета для открытия сундука.

Следующим шагом происходит генерация комнаты по заданным параметрам, включая уровень сложности: определяется размер комнаты, количество кристаллов и координаты их расположения, с определенной вероятностью выбирается тип заполнения комнаты – на основе алгоритма создания лабиринта или на основе разбиения комнаты на регионы. В соответствии с выбранным типом происходит размещение в комнате платформ, не-статичные платформы выбираются произвольным образом из статичных, после происходит расположение граничных платформ в комнате. Далее происходит расчет количества кристаллов, которые необходимы для открытия двери. Затем размещаются двери, которые ведут в другие комнаты в соответствии с исходным представлением игрового мира.

Первоначальные значения всех показателей от уровня сложности занесены в таблицу 1 и могут меняться в ходе тестирования.

Таблица 1 – Параметры генерации для разных уровней сложности

Параметр	Уровень сложности 1	Уровень сложности 2	Уровень сложности 3
Количество комнат	5	10	15
Количество очков за кристалл	1	2	3
Количество кристаллов в комнате	(20-30)*1,1	(30-50)*1,2	(50-55)*1,3
Процент количества кристаллов от общего в комнате для открытия двери	20%	30%	50%
Процент количества не-статичных платформ	10%	20%	30%
Количество особых предметов для открытия сундука	3	7	10

3.3. Варианты использования игрового приложения

Для описания способов взаимодействия с игровым приложением была спроектирована диаграмма вариантов использования [38], приведенная на рисунке 20.

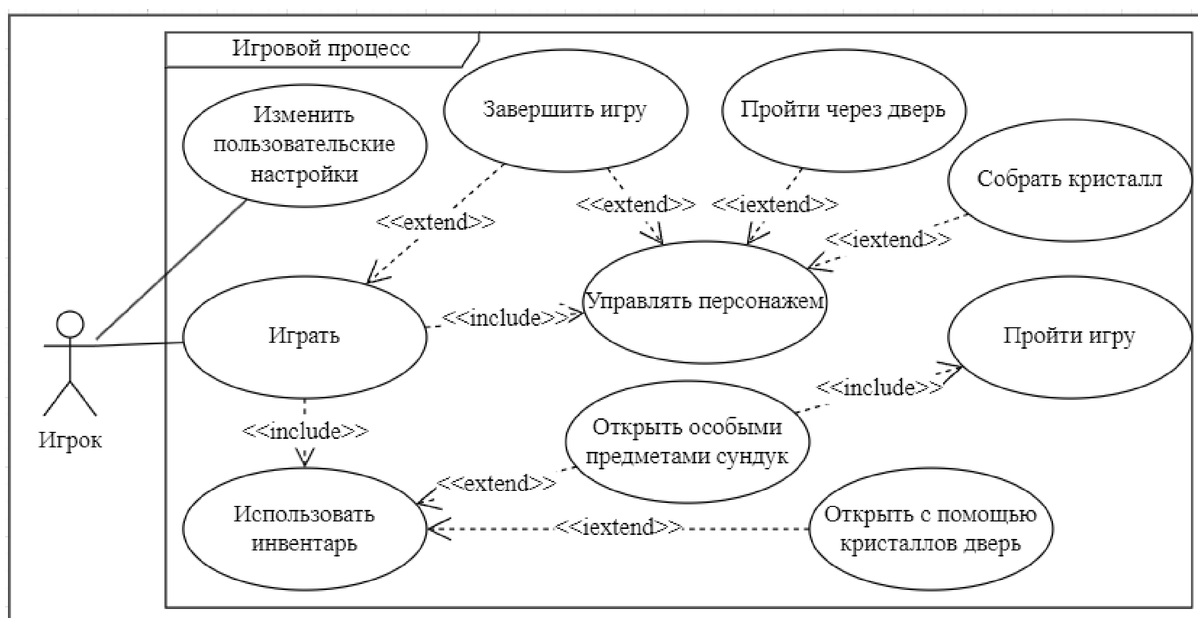

Рисунок 20 – Диаграмма вариантов использования игрового приложения

Актер «Игрок» представляет собой роль, которую выполняет пользователь при взаимодействии с игровым приложением.

«Игрок» может начать новую игру. «Игрок» выбирает этот вариант, чтобы начать игровую сессию с выбранным уровнем сложности. Когда «Игрок» выбрал этот вариант использования, игровое приложение показывает ему справку по управлению и конечную игровую цель. Затем начинается игровой процесс внутри сгенерированного мира.

«Игрок» может изменить пользовательские настройки: он может редактировать параметры громкости фоновой музыки и звуковых эффектов при взаимодействии с игровыми объектами. Также он может изменить размер окна игрового приложения.

После того, как «Игрок» запустил игровой процесс, он может управлять персонажем, перемещая его с помощью клавиатуры по платформам.

Во время перемещения «Игрок» может пройти через дверь. Этот вариант использования означает загрузку новой сгенерированной комнаты, если комната с указанным на двери номером не была посещена ранее, или активацию комнаты, если она была сгенерирована ранее. Текущая комната становится неактивной.

Во время перемещения «Игрок» может собрать кристалл. Этот вариант использования позволяет «Игроку» увеличивать общий счет очков, увеличивает процент прохождения комнаты. Игровое приложение увеличивает количество кристаллов собранного типа на 1.

Во время игры «Игрок» может использовать инвентарь. Для этого он нажимает клавишу на клавиатуре для активации желаемого объекта. Игровое приложение подсвечивает цветом активированную ячейку инвентаря и деактивирует остальные.

Во время взаимодействия с инвентарем «Игрок» может открыть с помощью кристаллов дверь. Для этого он нажимает на клавишу клавиатуры, используя выбранный элемент инвентаря. Если тип выбранного кристалла соответствует типу двери или типу текущей локации и «Игроку» хватает ко-

личества, то игровое приложение уменьшает количество выбранного кристалла в инвентаре, увеличивает число объектов особого предмета и делает дверь доступной для перемещения, то есть цветной.

Во время взаимодействия с инвентарем «Игрок» может открыть особыми предметами сундук. Для этого он нажимает клавишу клавиатуры. Если количества особых предметов достаточно для открытия сундука, игровое приложение уменьшает количество выбранного предмета.

Открытие сундука позволяет «Игроку» пройти игру. Игровое приложение останавливает игровой процесс, после чего показывает «Игроку» результаты: проценты прохождения всех посещенных, не-посещенные комнаты, а также общее количество очков, которое набрал «Игрок».

Во время игры «Игрок» может досрочно завершить игру двумя способами: завершить игровой процесс с помощью кнопки выхода или проиграть игру, если во время управления на локации «Облачное пространство» он упадет за пределы границы комнаты.

3.4. Проектирование игрового генератора

Проектируемый генератор должен выполнять следующие действия:

1) генерировать связи между локациями;
2) генерировать размер комнаты;
3) расположение дверей в комнате;
4) расположение игровых объектов в комнате;
5) количество кристаллов в комнате;
6) количество очков за кристалл.

Хорошими качествами результатов генерации, являющимися основой игрового процесса, являются:

1) точки переходов между комнатами расположены не ближе, чем на две платформы по каждой из сторон;

2) игровые объекты распределены равномерно по игровому пространству;

3) количество препятствий определяется уровнем сложности.

Недопустимыми результатов генерации являются:

1) наличие недостижимых предметов;

2) недостижимые платформы.

Для реализации данного генератора используются выбранные в выводе второй главы алгоритмы: граф по модели Барабаши-Альберта, дискретизация диска Пуассона, создание лабиринта, размещение меток на основе сетки и метод двоичного разбиения пространства (BSP).

Игровой генератор состоит из следующих компонентов: модели, графа, локации, размещения кристаллов, а также размещения платформ на основе сетки и лабиринта (рисунок 21).

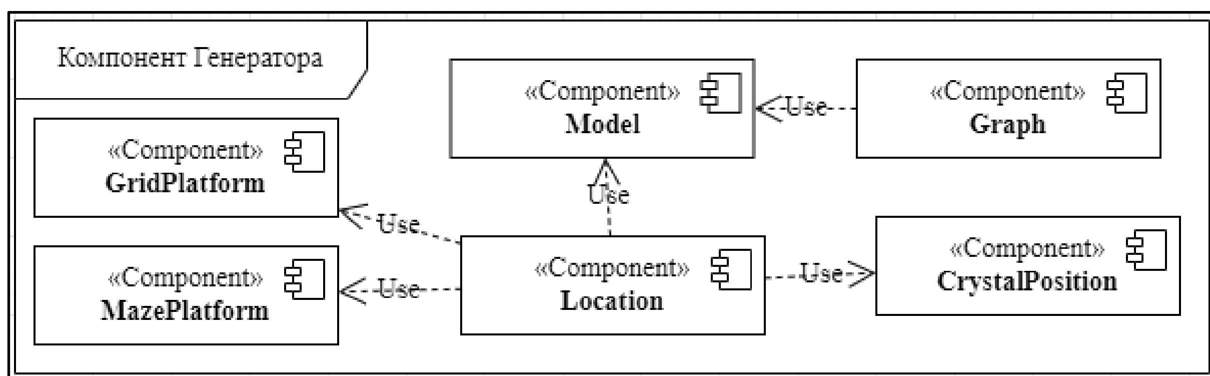

Рисунок 21 – Диаграмма компонентов Генератора

В компоненте Model хранятся все параметры генерации (таблица 1).

Компонент Graph отвечает за генерацию графа и присваивание сгенерированным вершинам типы локаций, а ребрам присваивает двери. Использует компонент модели для определения количества комнат.

Компонент CrystalPosition отвечает за способ генерации кристаллов.

Компонент GridPlatform отвечает за способ генерации платформ с помощью алгоритмов BSP и размещения платформ внутри регионов по сетке.

Компонент MazePlatform отвечает за способ генерации платформ с помощью алгоритма создания лабиринта.

Компонент Location отвечает за представление результата генерации комнаты. Использует компоненты GridPlatform и MazePlatform для выбора

стратегии генерации новой комнаты. Использует компонент Model для установки параметров генерации, и компонент CrystalPosition для размещения объектов кристаллов.

3.5. Архитектура игрового приложения

На рисунке 22 приведена диаграмма компонентов основной игровой сцены. На диаграмме отображены взаимодействия между компонентами. Архитектура основной игровой сцены состоит из компонентов элементов игрового мира, а также компонентов, управляющих игровым процессом.

Группа компонентов «Элементы игрового мира» состоит из компонентов: платформ, кристаллов, звука, игрового фона, открывающихся объектов, инвентаря и персонажа.

Группа компонентов «Игровой процесс» состоит из компонентов: генератора, комнаты, игрового менеджера, а также шины.

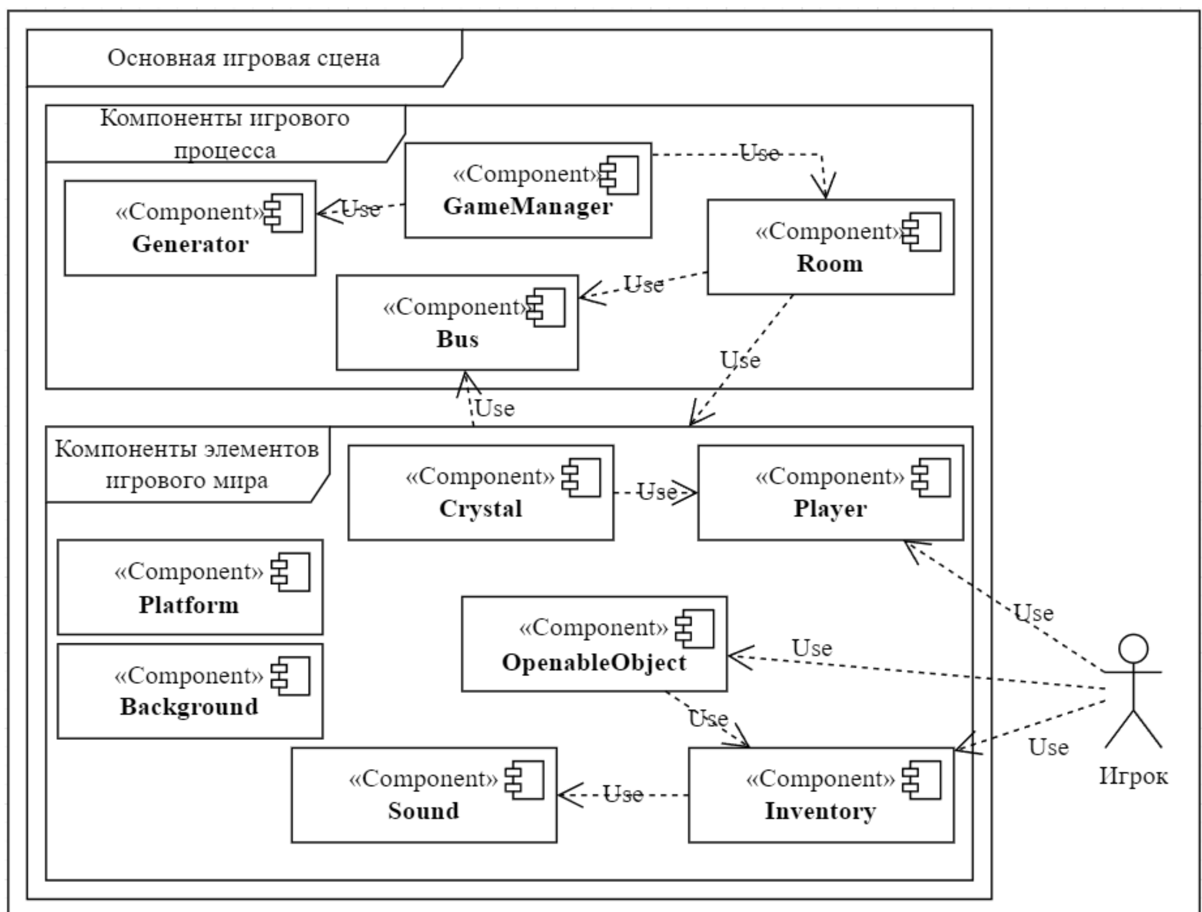

Рисунок 22 – Диаграмма компонентов основной игровой сцены

Компоненты игрового процесса

Компонент Generator отвечает за процедурную генерацию в игровом мире (рисунок 21). Он содержит алгоритмы и параметры, которые определяют, когда и как создаются новые игровые объекты.

Компонент Bus (шина) представляет шаблон проектирования Event Bus [11]. Он используется для обеспечения коммуникации между компонентами игрового приложения и слабой связности между компонентами, поскольку не взаимодействуют напрямую. Основой этого компонента является метод «публикация-подписка». Компонент «публикует» событие в шину, а другие компоненты, имеющие «подписку», реагируют на эти события.

Компонент GameManager отвечает за управление игровыми комнатами, переключение между ними. Этот компонент взаимодействует с компонентом генератора для запуска процедурной генерации комнат. Компонент получает данные о текущей комнате от Room и управляет переходами между комнатами через Bus.

Компонент Room отвечает за управление локацией. В нем хранится информация о комнате, описывается визуализация локации, обрабатывается обновление счета «Игрока». Данный компонент взаимодействует с компонентами элементов игрового мира, а также компонентом шины.

Компоненты элементов игрового мира

В игровой сцене «Игрок» взаимодействует с игровой сценой через компоненты персонажа, инвентаря и открываемых объектов. Компоненты элементов игрового мира получают обновления состояний от компонентов Room и Bus.

«Игрок» использует компонент Player (персонаж), который включает: данные и состояние игрока, отображение состояния игрока, взаимодействие с пользователем. В игровой сцене находится всегда один экземпляр данного компонента. Компонент получает данные об окружении через компоненты Crystal и Bus.

Компонент Crystal отвечает за управление и визуализацию кристаллов в игре. Взаимодействует с компонентом Player напрямую, когда происходит обработка взаимодействия экземпляра кристалла с персонажем.

Компонент Platform представляет собой основные объекты игрового мира, по которым перемещается персонаж. Этот компонент отвечает за реализацию механик платформ: статичных, исчезающих, движущихся, инвертирующих движение и гравитацию персонажа.

Компонент Background отвечает за управление игровым фоном и состоит из трех основных частей: хранения данных фона, отображения фона и управления фоном в зависимости от изменения состояния уровня.

Компонент Sound отвечает за звуковые эффекты, воспроизводимые в ответ на действия игрока, а также за громкость эффектов и проигрываемой фоновой музыки.

Компонент Inventory отвечает за отображение в пользовательском интерфейсе состояние о собранных игроком объектов и реализует логику их использования. Использует компонент Sound для воспроизведения звуков.

Компонент OpenableObject представляет собой объекты, которые можно открывать – двери и сундук. Взаимодействует с компонентом Inventory, проверяя количество необходимых предметов. При изменении состояния передает данные в Room через Bus.

3.6. Макеты пользовательского интерфейса

Всего было спроектировано 5 макетов пользовательского интерфейса.

На рисунке 23 приведен StartScreen – главный экран стартовой сцены игрового приложения, который отображается при запуске. На экране расположены две кнопки: «Играть» и «Выйти». Фон стартового экрана состоит из четырех картинок, представляющих собой фоны каждой локации. В верхней части расположено название игры на однотонном фоне. Ниже находится область с выбором уровня сложности игры, где «1» – самый легкий, а «3» – самый сложный. Выбранная игроком сложность становится подсвеченной.

Область изменения настроек громкости включает в себя два слайдера для установки параметров громкости музыки и игровых звуков. Также на главном экране расположен игровой персонаж.

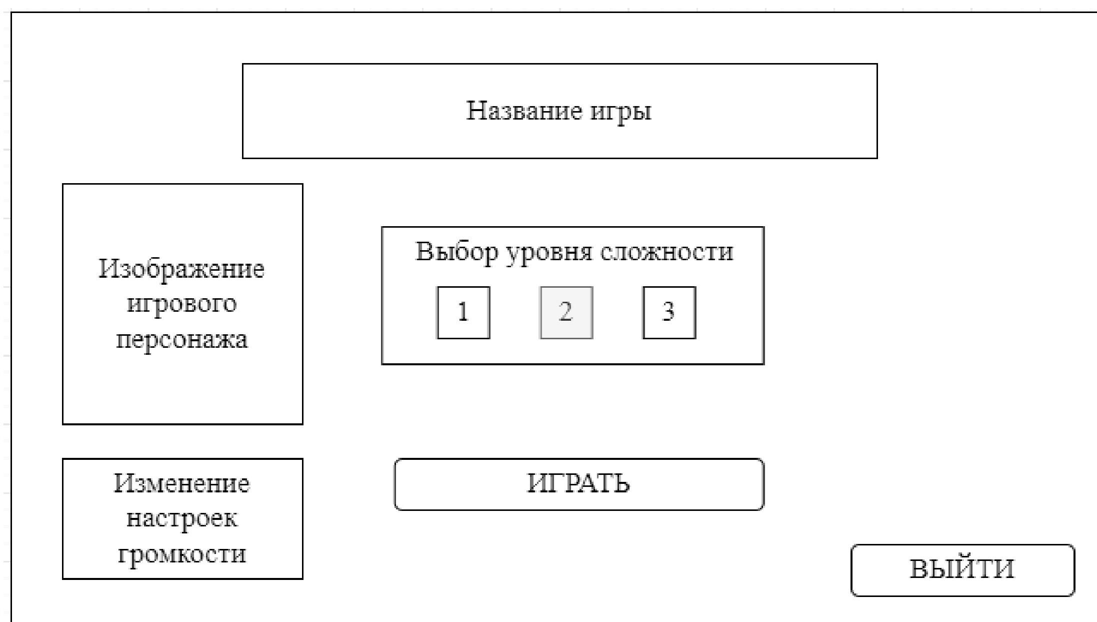

Рисунок 23 – Макет экрана StartScreen

Когда «Игрок» нажимает на кнопку «Играть» открывается окно (рисунок 24 с игровой справкой, содержащей: информацию о клавишах управления персонажем, инвентарем, достижение главной игровой цели, а также правила, в которых расположена информация о взаимодействии с дверями, сундуком, условием прохождения игры, а также описание ситуации проигрыша. Информация представлена в виде текстовых полей и изображений.

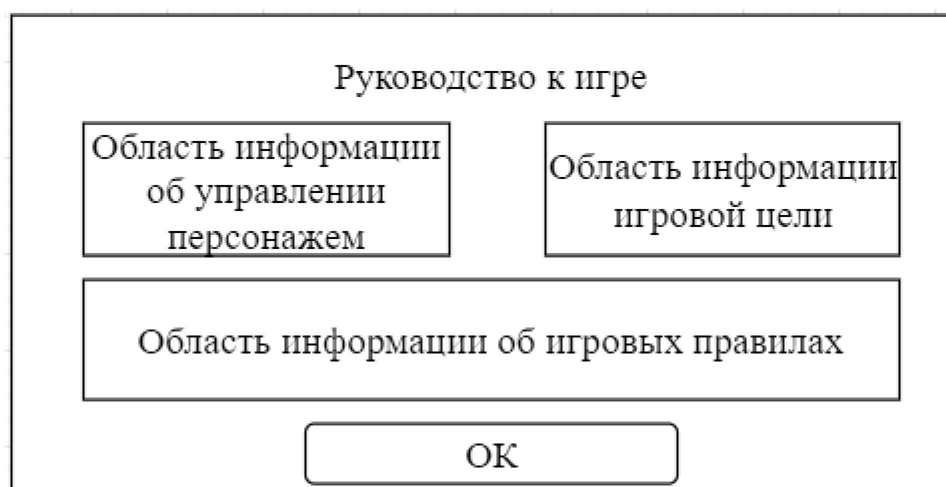

Рисунок 24 – Макет окна ManualScreen

Когда «Игрок» нажимает на кнопку «ОК», загружается основная игровая сцена. На рисунке 25 приведен GameScreen – экран основной игровой сцены, который отображается во время игрового процесса. В центре экрана всегда находится персонаж. Внизу расположена область инвентаря. Инвентарь представляет собой 5 ячеек: 4 для кристаллов и 1 для особых предметов. Каждая ячейка инвентаря содержит информацию о находящемся в ней предмете: изображение предмета, количество, а также название клавиши клавиатуры для выбора предмета – «1», «2», «3», «4», «5». Ячейка активного для использования предмета подсвечивается цветом. Если предмет доступен для применения в игровом мире, то над персонажем появляется надпись «Подсказка», указывающая на то, какое действие доступно «Игроку» в данный момент. Если таковых нет – подпись исчезает с экрана.

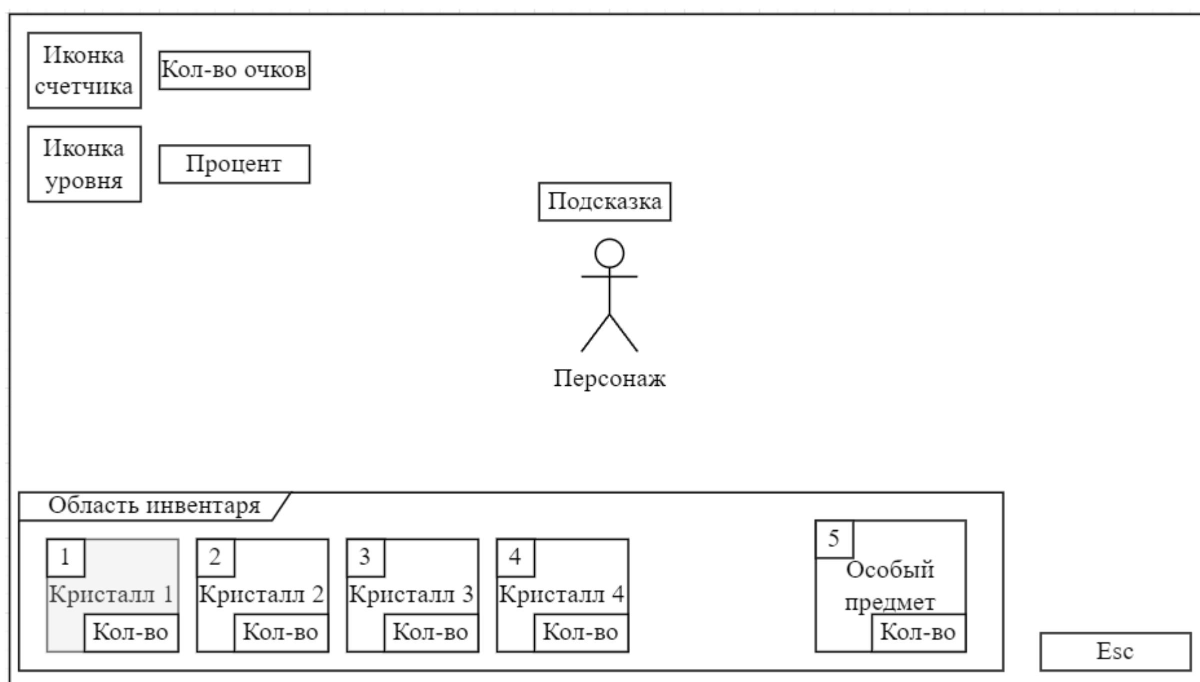

Рисунок 25 – Макет экрана GameScreen

Подпись «Esc», указывает на то, какую клавишу клавиатуры нужно нажать, чтобы вызвать экран игрового меню MenuScreen (рисунок 26). После этого поверх игрового экрана откроется окно с выбором одного из действий соответствующих кнопок: «Вернуться в игру» и «Выйти». Также на данном окне есть область изменения настроек громкости в виде слайдеров.

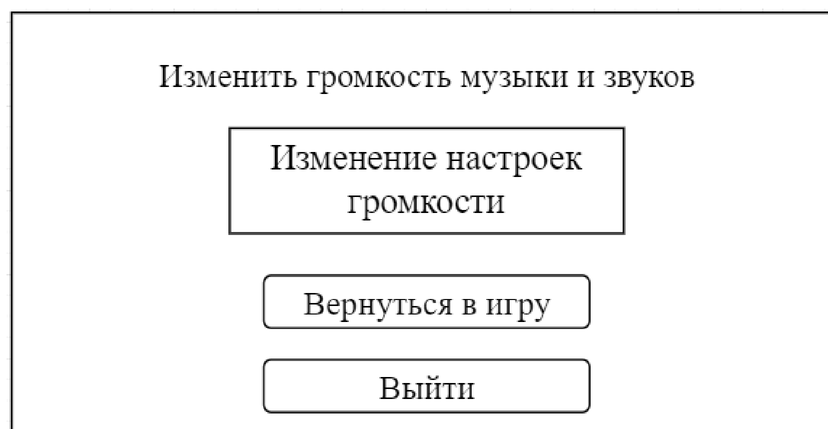

Рисунок 26 – Макет окна MenuScreen

Когда «Игрок» проходит игру, открывается окно GameCompleted (рисунок 27). На экране содержится информация о том, что игра была пройдена. Присутствует кнопка «ОК», нажатие на которую ведет «Игрока» на стартовую сцену с экраном StartScreen. В центре экрана располагается область вывода результатов прохождения игры – текстовое поле, в которое игровое приложение помещает данные.

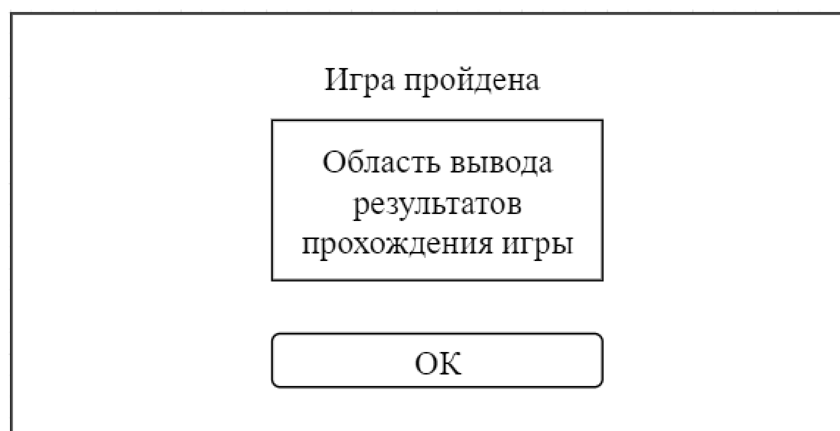

Рисунок 27 – Макет окна GameCompleted

Выводы по третьей главе

В данной главе были определены функциональные и нефункциональные требования, спроектирована концепция игры, приведены варианты использования приложения. Был описан процесс проектирования игрового генератора. Созданы компонентная архитектура приложения и макеты пользовательского интерфейса.

4. РЕАЛИЗАЦИЯ

4.1. Средства реализации

Игровой движок Unity позволяет создать следующие типы проектов для компьютерных игр: 2D, 3D, 2D (UPR), 3D (UPR), 3D (HDRP), 3D Sample Scenes (UPR), 3D Sample Scene (HDRP).

2D-проекты разработаны с использование двумерной графики и механик игрового процесса. В таких проектах игровой мир представлен в виде плоскости, по которой перемещаются персонажи и объекты. Примерами таких игр является игры: вид сверху «Top-Down» и вид сбоку «Side-scrolling».

3D-проекты разработаны с использованием трехмерной графики. В таких проектах игровой мир представлен в трех измерениях (горизонталь, вертикаль и глубина), по которым могут перемещаться объекты и персонажи.

UPR-проекты создаются с использованием Universal Render Pipeline (UPR) [34]. Universal Render Pipeline (UPR) – инструмент для создания визуальных эффектов, который обеспечивает более гибкие настройки для процесса рендеринга.

3D HDRP (High Definition Render Pipeline) проекты также поддерживают расширенные возможности для работы с графикой, физикой, анимацией.

Были выбраны следующие средства реализации:

1) версия Unity 2022.3.19f1 [43] и тип проекта 2D;

2) UnityHub [44] – приложение для управления версиями и проектами Unity;

3) текстовый редактор кода Visual Studio Code [45], поддерживающий подсветку синтаксиса, автоматического дополнения кода, отладку и интеграцию с системой контроля версий Github [42];

4) библиотека QuikGraph 2.5.0 [27] для работы с графами;

5) нейронная сеть для генерации изображений объектов игрового мира [22].

Также были использованы следующие готовые наборы из официального магазина AssetStore Unity [32]:

1) изображение персонажа взято из готового набора [28];

2) элементы пользовательского интерфейса взяты из пакета графического интерфейса [5].

4.2. Структура реализации

Реализация игрового приложения состоит из трех частей:

1) реализации генерации игрового окружения в соответствии с диаграммой компонентов генератора;

2) реализации компонентов, отвечающих за управление игровым процессом;

3) реализации компонентов, представляющих элементы игрового мира.

Реализация основана на паттерне MVC [4]. Представлением является объект на сцене – собранный префаб [24]. Префабом является компонент настроенного игрового объекта, который наследуются от компонента `MonoBehaviour` [20], хранит информацию о координатах, изображении, имеет коллайдер и дополнительные параметры. В контроллерах реализуется игровая логика, поведение объектов, а в моделях хранятся численные параметры, которые были получены при генерации. На рисунке 28 приведены все реализованные префабы, используемые на основной игровой сцене.

Рисунок 28 – Префабы проекта

В ходе реализации были реализованы 48 классов, 2 перечисления, которые отвечают за состояния объектов: принадлежности объекта к определенной реализации локации, а также состоянию персонажа.

4.3. Реализация компонента генератора и игрового менеджера

Реализация компонента игрового менеджера

Компонент игрового менеджера состоит реализован в классе `GameManager`, наследуемый от `MonoBehaviour`. В методе `Start` инициализируются словари для хранения префабов локаций, дверей и созданных комнат. Происходит подписка на события загрузке новой комнаты и метод передачи информации о прохождении игры. На рисунке 29 приведена диаграмма деятельности, которая описывает процесс переключения комнат. Метод `StartNewLocation` приведен в листинге 1.

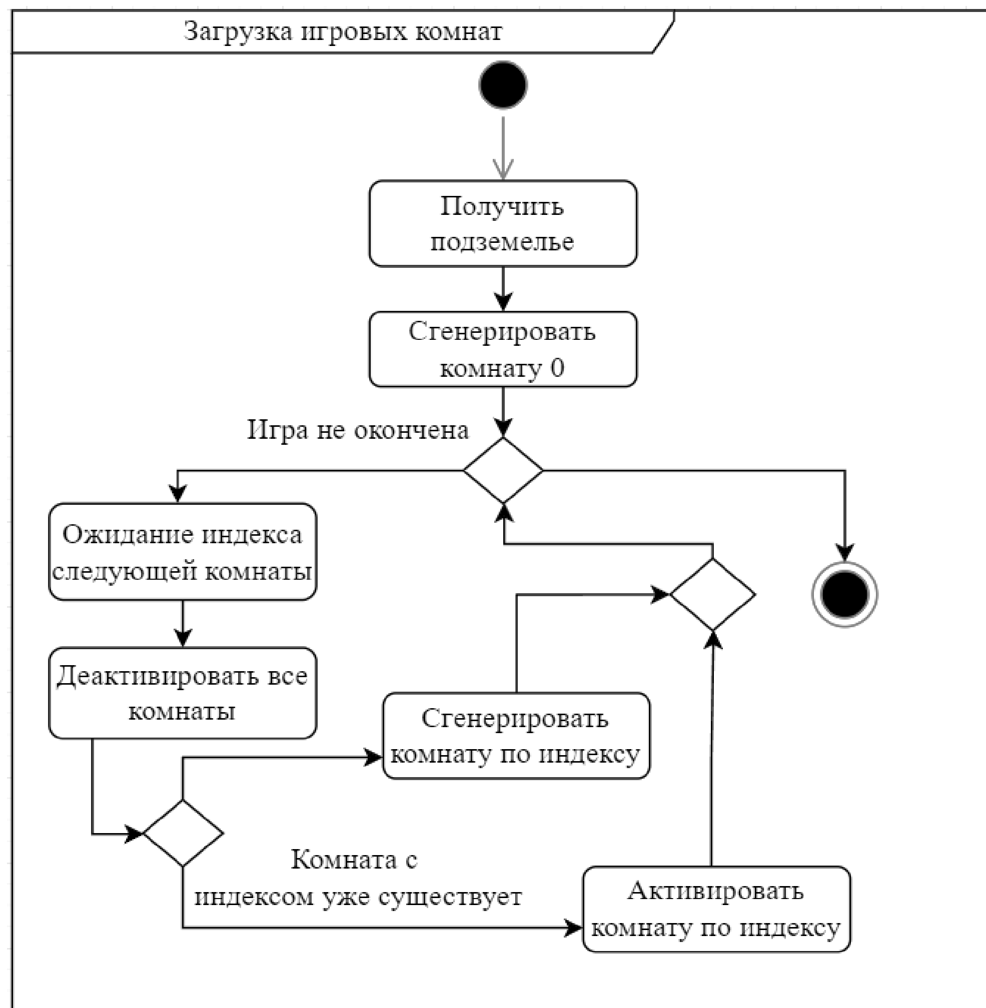

Рисунок 29 – Диаграмма деятельностей создания и переключения комнат

Листинг 1 – Метод `StartNewLocation` класса `GameManager`

```
private void StartNewLocation(int index){
    indexCurrentLocation = index;
    var doorInNewModel = GetDoorsForIndexLocation(index);
    LocationType = _locationNetwork.Rooms[index];
    LevelView levelPrefab = levelPrefabs[locationType];
    LevelModel newModel;
    int difficulty = PlayerPrefs.GetInt("Difficulty");
    if (index == 0){
        newModel = generatorLocations.GenerateNewLocation(2,
            index, doorInNewModel,
            GeneratorModel.GetCountForChest(difficulty));
    }
    else {
        newModel = generatorLocations.GenerateNewLocation(2, index,
            doorInNewModel);
    }
    LevelView levelInstance = Instantiate(levelPrefab);
    createdLevels[index] = levelInstance;
    levelInstance.model = newModel;
    levelInstance.SetModel();
    levelInstance.gameObject.SetActive(true);
    PlayerController.Instance.CurrentType = levelInstance.crystalType;
}
```

Этот метод устанавливает текущий номер комнаты равным переданному индексу. Затем метод `GetDoorsForIndexLocation(index)` возвращает все двери, связанные с текущей комнатой. На основе графа комнат и переходов выбирается префаб комнаты, соответствующий переданному типу локации. Далее осуществляется генерация новой модели комнаты. Если индекс комнаты равен 0, то генерируется модель комнаты с сундуком, иначе – обычная. Создается экземпляр комнаты, добавляемый в словарь созданных комнат. После этого происходит активация созданной комнаты на игровой сцене.

Реализация компонента генератора

Компонент генератора состоит из 6 классов: `GeneratorGraph`, `GeneratorLocation`, `GeneratorMazePlatform`, `GeneratorGridPlatform`, `GeneratorCrystalPosition`, и `GeneratorModel`.

Класс `GeneratorGraph` создает граф для генерации заданного количества комнат, создает все двери в каждую комнату, а также назначает комнате один из типов локации. Класс `GeneratorModel` содержит все параметры для генерации, приведенные в таблице 1.

Класс `GeneratorLocation` отвечает за создание локации, используя экземпляры классов `GeneratorMazePlatform`, `GeneratorGridPlatform` и `GeneratorCrystalPosition`.

Реализация алгоритма создания графа

На рисунке 30 приведена диаграмма деятельности создания на основе графа комнат и переходов между ними.

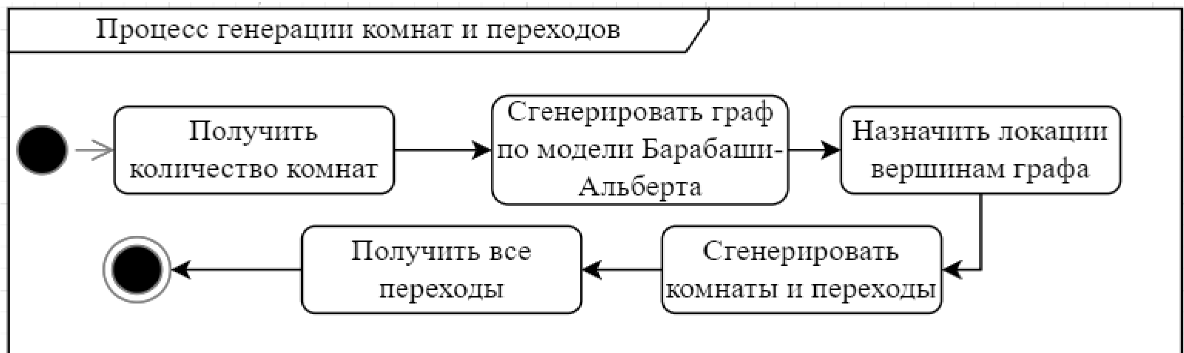

Рисунок 30 – Диаграмма деятельности создания комнат и переходов

Метод `BarabasiAlbertGraph(int n, int m)` реализует алгоритм Барабаши-Альберта для генерации случайного графа. Создается пустой граф G и список целей `targetList`. Далее осуществляется создание полного графа с m начальными узлами: для каждого узла от 0 до m добавляется вершина в граф G и ребро между каждой парой вершин. Затем каждый узел добавляется в `targetList` m раз. После этого происходит добавление оставшихся узлов: для каждого узла от m до n добавляется вершина в граф G, после выбираются m целей из `targetList` в случайном порядке. Для каждой цели добавляется ребро между текущим узлом и целью, если такого ребра еще нет. Затем каждая цель и текущий узел добавляются в `targetList` m раз. Код метода приведен в листинге 2. Временная сложность алгоритма генерации графа составляет $O(n \times m + m^2)$, где: n – количество комнат, а m – количество начальных узлов, m^2 – создание полного графа с m начальными узлами, $n \times m$ – добавление оставшихся узлов. Пространственная сложность алгоритма составляет $O(n \times m)$.

Листинг 2 – Метод создания графа в классе GeneratorGraph

```
private static UndirectedGraph<int, Edge<int>> BarabasiAlbertGraph(int n,
int m)
{
    var G = new UndirectedGraph<int, Edge<int>>();
    var targetList = new List<int>();
    var random = new System.Random();
    for (int i = 0; i < m; i++)
    {G.AddVertex(i);
        for (int j = 0; j < i; j++)
        {G.AddEdge(new Edge<int>(i, j)); }
        targetList.AddRange(Enumerable.Repeat(i, m)); }
        for (int i = m; i < n; i++)
        {G.AddVertex(i);
    var targets = targetList.OrderBy(x => radom.Next()).Take(m).ToList();
    foreach (var target in targets.Take(m))
    {var newEdge = new Edge<int>(i, target);
    if (!G.ContainsEdge(newEdge))
    {G.AddEdge(newEdge); }}
    targetList.AddRange(targets.Take(m));
    targetList.AddRange(Enumerable.Repeat(i, m)); }
    return G;
}
```

Затем в момент загрузки новой комнаты GeneratorLocation возвращает модель. На рисунке 31 приведена диаграмма деятельности генерации комнаты. Параметр m – число из генератора, которое задает порог вероятности для появления комнаты, сгенерированной по регионам, n – случайное число, полученное при запуске генерации комнаты.

В листинге 3 приведен лог генерации игрового мира с назначением вершинам графа комнат, создание переходов.

Листинг 3 – Лог генерации игрового мира

```
 Выбран уровень сложности: 1
[Generator] [GeneratorGraph] Vertices: 0, 1, 2, 3, 4
[Generator] [GeneratorGraph] Edges: (1, 0), (2, 0), (3, 1), (4, 3)
[Generator] [GeneratorGraph] Levels: Level 0: Sky, Level 1: Red, Level 2:
Blue, Level 3: Green, Level 4: Red,
    Prexod:
    Level 0: Sky -> Level 1: Red, Level 2: Blue
    Level 1: Red -> Level 0: Sky, Level 3: Green
    Level 2: Blue -> Level 0: Sky
    Level 3: Green -> Level 1: Red, Level 4: Red
    Level 4: Red -> Level 3: Green
[GameManager]Door Index:0, Current Location:0, Sky, Next Location:1,Red
[GameManager]Door Index:1, Current Location:0, Sky, Next Location:2,Blue
[GameManager]Door Index:2, Current Location:1, Red, Next Location:0,Sky
[GameManager]Door Index:3, Current Location:1, Red, Next Location:3,Green
[GameManager]Door Index:4, Current Location:2, Blue, Next Location:0,Sky
[GameManager]Door Index:5, Current Location:3, Green, Next Location:1,Red
[GameManager]Door Index:6, Current Location:3, Green, Next Location:4,Red
[GameManager]Door Index:7, Current Location:4, Red, Next Location:3,Green
```

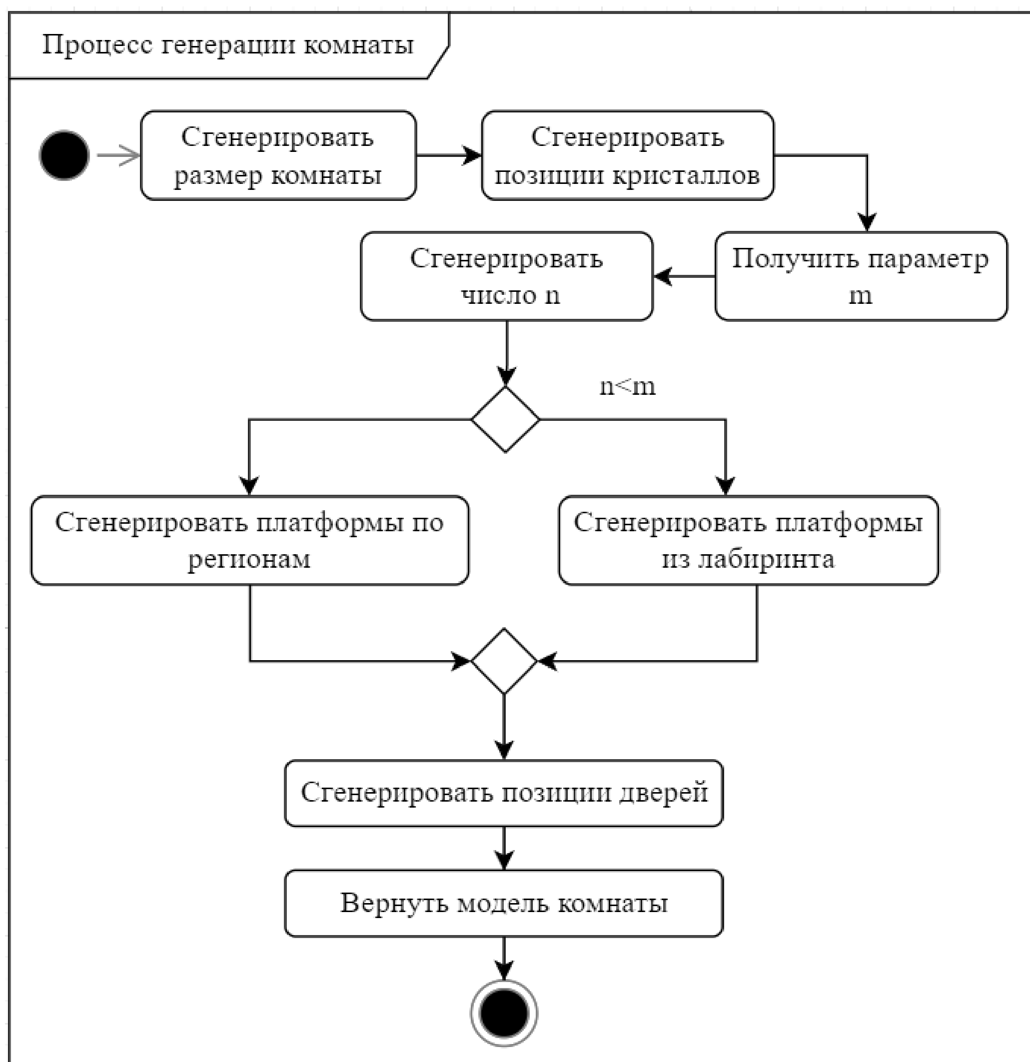

Рисунок 31 – Диаграмма деятельности генерации модели комнаты

Реализация алгоритма создания платформ из лабиринта

Процесс генерации платформ на основе алгоритма создания лабиринта реализован в классе `GeneratorMazePlatform`. Конструктор класса инициализирует размер платформы, который будет использоваться при генерации. Далее происходит генерация лабиринта на основе стека. При этом осуществляется удаление некоторых платформ с заданной вероятностью, дабы создать «прорези» для более свободного перемещения персонажа. После этого происходит создание платформ на основе сгенерированного лабиринта – добавление их в список платформ, который присваивается модели комнаты.

Алгоритм генерации лабиринта на основе стека (алгоритм поиска в глубину) реализован в методе `GenerateMaze(int width, int height, float removalProbability)` и работает следующим образом: создается пустой лабиринт и стек, в который добавляется начальная позиция (0,0). Пока стек не пуст, извлекается текущая позиция. Для каждой соседней позиции проверяется, не была ли она уже посещена `(maze[neighbor.y, neighbor.x] == 0)`. Если соседняя позиция не была посещена, она добавляется в стек, и между текущей и соседней позициями создается проход. При этом с вероятностью `removalProbability` текущая позиция удаляется из лабиринта. Далее происходит удаление из стека: если для текущей позиции не найдено не-посещенных соседей, она удаляется из стека. В листинге 4 приведен код создания лабиринта на основе стека. Временная и пространственная сложности алгоритма составляют $O(width \times height)$ – соответствует количеству клеток в поле.

Листинг 4 – Код метода `GenerateMaze` класса `GeneratorMazePlatform`

```
public void GenerateMaze(int width, int height, float removalProbability)
{
    maze = new int[height, width];
    Stack<Vector2Int> stack = new Stack<Vector2Int>();
    stack.Push(new Vector2Int(0, 0));
while (stack.Count > 0)
{
    Vector2Int current = stack.Peek();
    maze[current.y, current.x] = 1;
    List<Vector2Int> neighbors = new List<Vector2Int>();
    if (current.x > 1) neighbors.Add(new Vector2Int(current.x - 2, cur-
rent.y));
    if (current.x < width - 2) neighbors.Add(new Vector2Int(current.x + 2,
current.y));
    if (current.y > 1) neighbors.Add(new Vector2Int(current.x, current.y -
2));
    if (current.y < height - 2)
    neighbors.Add(new Vector2Int(current.x, current.y + 2));
    neighbors.Shuffle();
    bool found = false;
    foreach (var neighbor in neighbors)
    {if (maze[neighbor.y, neighbor.x] == 0){
        stack.Push(neighbor);
    maze[(current.y + neighbor.y) / 2, (current.x + neighbor.x) / 2] = 1;
    if (Random.value < removalProbability)
    {maze[current.y, current.x] = 0;}
        found = true;
        break; }}
   if (!found) {stack.Pop();}}
}
```

Далее метод `GeneratePlatforms(Rect region)` генерирует платформы на основе сгенерированного лабиринта. Происходит инициализация пустого списка платформ, осуществляется вычисление размеров каждой ячейки лабиринта. Для каждой ячейки, если она не была посещена, создается платформа. Позиция платформы вычисляется на основе позиции ячейки в лабиринте и размеров ячейки. Созданная платформа добавляется в список платформ. После этого метод возвращает результирующий список всех сгенерированных координат для размещения платформ. Код метода приведен в листинге 5.

Листинг 5 – Код метода `GeneratePlatforms`

```
public List<Vector3> GeneratePlatforms(Rect region)
{
    var platforms = new List<Vector3>();
    var cellWidth = region.width / maze.GetLength(1);
    var cellHeight = region.height / maze.GetLength(0);
    for (int y = 0; y < maze.GetLength(0); y++)
    {for (int x = 0; x < maze.GetLength(1); x++)
        {if (maze[y, x] == 0)
         {var platformX = region.x + x * cellWidth + cellWidth / 2 - label-
Size.x / 2;
var platformY = region.y + y * cellHeight + cellHeight / 2 - labelSize.y /
2;
        platforms.Add(new Vector3(platformX, platformY, 0));}}
    }
    return platforms;
}
```

Реализация алгоритма создания платформ по регионам

Процесс генерации платформ на основе алгоритма создания лабиринта реализован в классе `GeneratorGridPlatform`. Конструктор инициализирует размер метки и размер сетки, которые используются при генерации платформ. Далее происходит позиционирование статичных платформ: в методе `PositionStaticPlatforms()` генерируются регионы с помощью рекурсивного алгоритма двоичного разбиения пространства (BSP), затем осуществляется размещение платформ в каждом регионе.

Алгоритм BSP реализован в методе `RecursiveBinarySpacePartition(Rect area, int depth, bool horizontal)`. Сперва происходит проверка глубины рекурсии: если глубина рекурсии равна нулю, то текущая

область добавляется в список регионов и рекурсия завершается. Далее осуществляется разбиение пространства: если глубина рекурсии не равна нулю, то текущая область разбивается на две части. Разбиение происходит горизонтально или вертикально в зависимости от значения параметра. Точка разбиения выбирается случайно в диапазоне от 0,4 до 0,8 от ширины или высоты области. Для каждой из полученных областей вызывается рекурсивный вызов метода, при этом глубина рекурсии уменьшается на единицу, а направление разбиения меняется на противоположное. Код приведен в листинге 6. Временная и пространственная сложности алгоритма составляют $O(2^{depth})$.

Листинг 6 – Реализация метода `RecursiveBinarySpacePartition` класса `GeneratorGridPlatform`

```
private List<Rect> RecursiveBinarySpacePartition(Rect area, int depth, bool
horizontal)
{var regions = new List<Rect>();
    if (depth == 0) {regions.Add(area); }
    else{var split = UnityEngine.Random.Range(0.4f, 0.8f);
    Rect first, second;
    if (horizontal) {first = new Rect(area.x, area.y, area.width * split,
area.height);
        second = new Rect(area.x + area.width * split, area.y, area.width *
(1 - split), area.height); }
    else{
        first = new Rect(area.x, area.y, area.width, area.height * split);
        second = new Rect(area.x, area.y + area.height * split, area.width,
area.height * (1 - split)); }
        regions.AddRange(RecursiveBinarySpacePartition(first, depth - 1,
!horizontal));
    regions.AddRange(RecursiveBinarySpacePartition(second, depth - 1, !hor-
izontal)); }
    return regions;
}
```

Далее вызывается алгоритм размещения платформ в регионе реализован в методе `PlacePlatformsInRegion(Rect region)`. Сперва происходит инициализация пустого списка с координатами платформ. Затем добавляются точки для входа и выхода в регион – вход добавляется в левый нижний угол региона, выход – в правый верхний. Далее осуществляется соединение входа и выхода путем вычисления минимального расстояния между двумя точками, а вдоль этого расстояния создаются платформы. После происходит процесс размещения дополнительных платформ в пустых областях. Затем

удаляются излишние платформы для создания более свободных областей для перемещения персонажа – после этого остается некоторый процент от исходного количества платформ.

Алгоритм размещения меток на основе сетки реализован в методе `FillEmptyAreaWithPlatforms(Rect region, List<Vector3> existing-Platforms)`, который создает сетку для отслеживания пустых областей и добавляет платформы. Для регионов создается сетка для отслеживания пустых областей как двумерный массив. Для каждой существующей платформы вычисляются ее координаты внутри сетки, и соответствующая ячейка отмечается как занятая. Далее для каждой ячейки в сетке, если она свободна, создается новая платформа с координатами, соответствующими этой ячейке. В конце выполнения метод возвращает список новых платформ.

После этого метод `RemoveExcessPlatforms(List<Vector3> plat-forms, float targetPlatformRatio)` реализует алгоритм удаления излишних платформ следующим образом. Создается пустой список для хранения платформ, которые будут сохранены. Затем исходя из заданного процента платформ, которые должны остаться, вычисляется целевое количество платформ для сохранения. Если количество больше нуля, то выбирается случайная стартовая платформа для сохранения. Пока количество сохраненных платформ меньше целевого, алгоритм продолжает искать ближайшую платформу к последней сохраненной платформе, вычисляя для каждой платформы расстояние до последней сохраненной. Платформа с наименьшим расстоянием сохраняется и удаляется из исходного списка. После того, как все ближайшие платформы были найдены и сохранены, метод возвращает итоговый список платформ.

Реализация алгоритма размещения кристаллов

Реализация размещения кристаллов основана на алгоритме дискретизации диска Пуассона. Процесс генерации реализован в классе в классе `GeneratorCrystalPosition`. Сперва осуществляется инициализация случайных образцов в методе `GeneratePoissonDiskSamples` (листинг 7), в

этот список добавляются случайные точки в пределах заданной ширины и высоты. Под образцами подразумеваются генерируемые координаты кристаллов. Затем выполняется релаксация Ллойда по заданному количеству шагов: на каждом шаге вычисляются центроиды ячеек, после чего к ним перемещаются образцы. В методе `ComputeCentroids` вычисляется центроид для каждой ячейки, создается сетка для хранения количества точек и их совокупных позиций в каждой ячейке. В методе `GenerateCrystalPosition` генерируются позиции кристаллов с учетом проверки границ исходного поля. Временная сложность алгоритма генерации точек составляет $O(n \times r)$, пространственная сложность составляет $O(n)$, где: n – количество образцов (кристаллов), r – количество шагов релаксации. Пространственная и временная сложности вычисления центроидов составляет $O(n + w \times h)$, где: w – ширина области, h – высота области.

Листинг 7 – Код метода `GeneratePoissonDiskSamples`

```
public List<Vector3> GeneratePoissonDiskSamples(int width, int height, int
numSamples, int relaxationSteps = 5){
    float radius = 3.0f;
    List<Vector3> samples = new List<Vector3>();
    System.Random rand = new System.Random();
    for (int i = 0; i < numSamples; i++){ samples.Add(new Vec-
tor3((float)rand.NextDouble() * width, (float)rand.NextDouble() * height,
0)); }
    for (int step = 0; step < relaxationSteps; step++){
    List<Vector3> centroids = ComputeCentroids(samples, width, height);
    for (int i = 0; i < samples.Count; i++)
        {Vector3 centroid = centroids[i];
        Vector3 direction = centroid - samples[i];
        direction.Normalize();
        samples[i] += direction * radius;}
    }
    return samples;
}
```

4.4. Реализация компонентов шины и комнат

Класс `Bus` реализует компонент Bus. Реализация основана на использовании паттерна Singleton [29], чтобы в любой момент времени существовал только один экземпляр данного класса. Класс содержит события, которые вызываются для отправки данных экземплярами других компонентов.

Каждое событие имеет соответствующий метод, который вызывает событие с переданными данными.

Компонент Room реализован классами `LevelController`, `LevelModel` и `LevelView`, который наследуется от `MonoBehaviour`.

Класс `LevelController` отвечает за управление событиями и игровой логикой внутри экземпляра комнаты, созданной компонентом `GameManager`. Класс содержит модель данных и представления комнаты, методы для создания: кристаллов, фона, платформ, дверей, сундука, а также метод для размещения персонажа при генерации комнаты. В классе осуществляется подписка `Bus.Instance.SendScore` методом `HandleScoreUpdate` на событие отправки обновления счетчика очков в комнате для каждого экземпляра платформы. Метод `SpawnCrystals()` реализует создание кристаллов на сцене: метод проходит по каждому объекту в списке `model.Crystal`. Для каждого объекта создается новый экземпляр кристалла `CrystalView` на позиции `gameObjectModel.Position` с использованием префаба `gameObjectModel.Prefab.UnityEngine.Object.Instantiate` создает новый экземпляр объекта в игровом мире, принимая три аргумента: префаб для создания, позицию и вращение для размещения нового объекта. Созданный кристалл добавляется в иерархию объектов под объектом `LevelView`. В конце созданная модель кристалла передается в представление кристалла.

Класс `LevelModel` представляет модель данных для комнаты игры. Он содержит информацию о префабах для кристаллов, платформ и фона, позициях для размещения кристаллов и платформ, общем счете, количестве кристаллов и текущем счете. Класс также предоставляет методы для управления моделью текущего счета: увеличения и уменьшения текущего счета.

Класс `LevelView` используется как префаб для сборки визуальной части комнаты на игровой сцене, в нем хранятся префабы: кристалла, статичных платформ, не-статичных платформ, платформ границ уровня, игрового фона, всех типов дверей и сундука. Всего реализовано 4 префаба типа

LevelView, каждый из которых представляет конкретный тип локации. На рисунке 32 приведен префаб для локации типа «Облачное пространство».

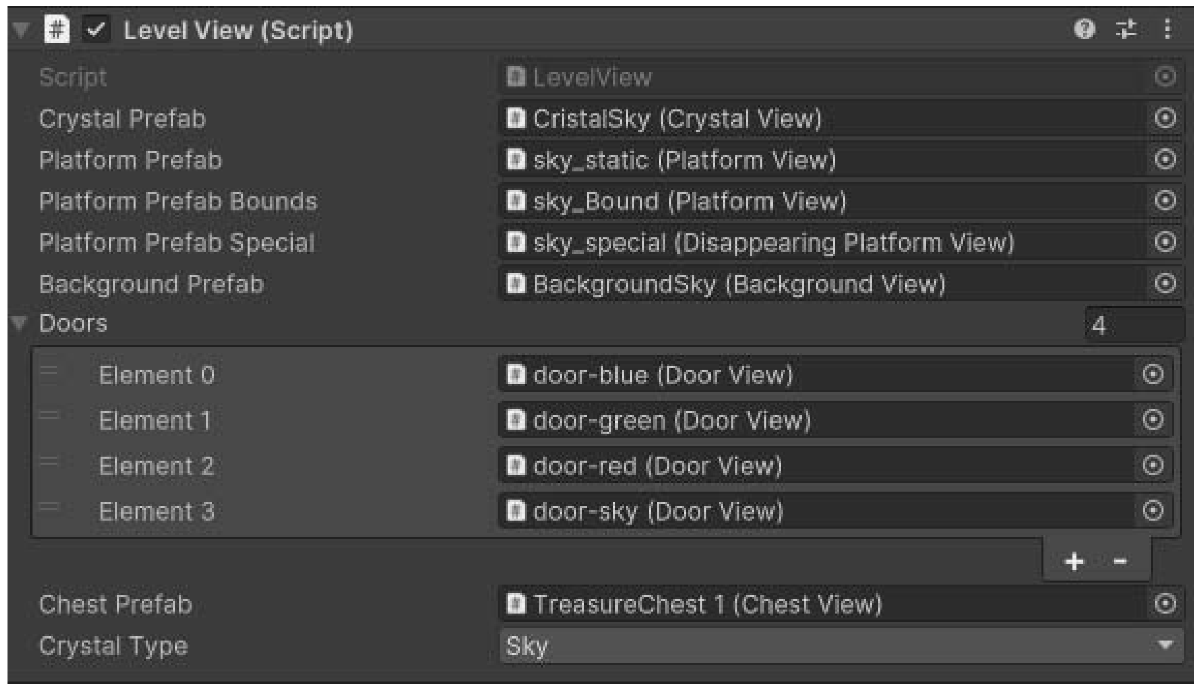

Рисунок 32 – Интерфейс в инспекторе Unity компонента LevelView

4.5. Реализация компонента персонажа

Реализация управления персонажа

Компонент персонажа состоит из следующих классов: PlayerController, PlayerModel, а также CameraManager и PlayerView, которые наследуются от MonoBehaviour. На рисунке 33 приведена диаграмма классов компонента Player.

Контроллер управляет поведением персонажа и содержит ссылки на модель и представление, управляя их взаимодействием. Контроллер обрабатывает пользовательский ввод в методе Update().

Класс CameraManager управляет камерой в игре, чтобы она следовала за «Игроком». Класс обновляет свою позицию каждый кадр, чтобы персонаж всегда оставался в центре экрана. Также класс имеет текстовое поле со всплывающей подсказкой.

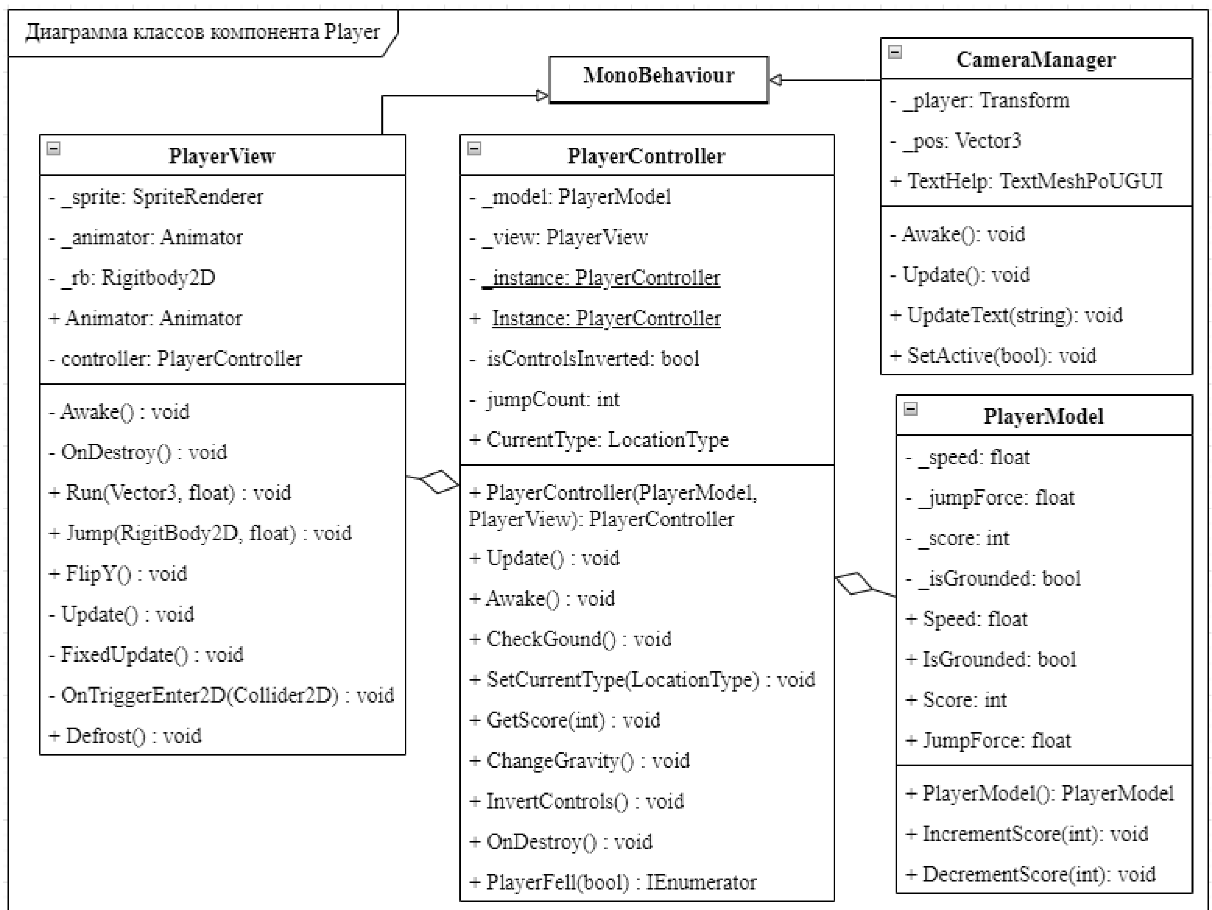

Рисунок 33 – Диаграмма классов компонента Player

Класс `PlayerModel` представляет модель данных игрока, он содержит в себе параметры скорости перемещения, силы прыжка, переменную для проверки, находится ли персонаж на платформе, а также счетчик очков.

Класс `PlayerView` управляет визуальным отображением персонажа. Он содержит ссылку на контроллер и методы для взаимодействия с контроллером, управляющие состоянием персонажа. Персонаж находится в одном из состояний: идет, прыгает или находится в покое. Метод `Run` (листинг 8) отвечает за перемещение персонажа и управляет его анимацией.

Листинг 8 – Код обработки перемещения персонажа

```
 public class PlayerView : MonoBehaviour
{...public void Run(Vector3 direction, float speed)
    {
     transform.position = Vector3.MoveTowards(transform.position, trans-
form.position + direction, speed * Time.deltaTime);
 _     sprite.flipX = direction.x < 0.0f; }
...
}
```

Реализация анимации персонажа

Реализация анимации персонажа в Unity происходила покадрово. Во время игры изображения проигрываются последовательно, создавая движущееся изображение. Когда состояние персонажа меняется (например, переходит из состояния ходьбы в состояние прыжка), Unity автоматически переключает набор изображений. Это происходит с помощью компонента Animator. На рисунке 34 представлена раскадровка персонажа для создания анимации.

Рисунок 34 – Нарезка на спрайты изображения персонажа

У персонажа есть 4 состояния анимации.

1. Hero_idle – это состояние, в котором персонаж находится в покое. Отсюда персонаж может перейти в любое другое состояние.
2. Any State – это специальное состояние, которое позволяет переходить в любое другое состояние независимо от текущего состояния. Из Any State можно перейти в состояния Hero_go и Hero_jump.
3. Hero_go – это состояние, в котором персонаж перемещается.
4. Hero_jump – это состояние, в котором персонаж прыгает.

На рисунке 35 приведена диаграмма состояний анимации для персонажа в Unity Animator [2]. Entry является начальным состоянием, откуда начинается вся анимация. Entry автоматически переходит в состояние Hero_idle. Стрелки между состояниями указывают на возможные переходы между ними.

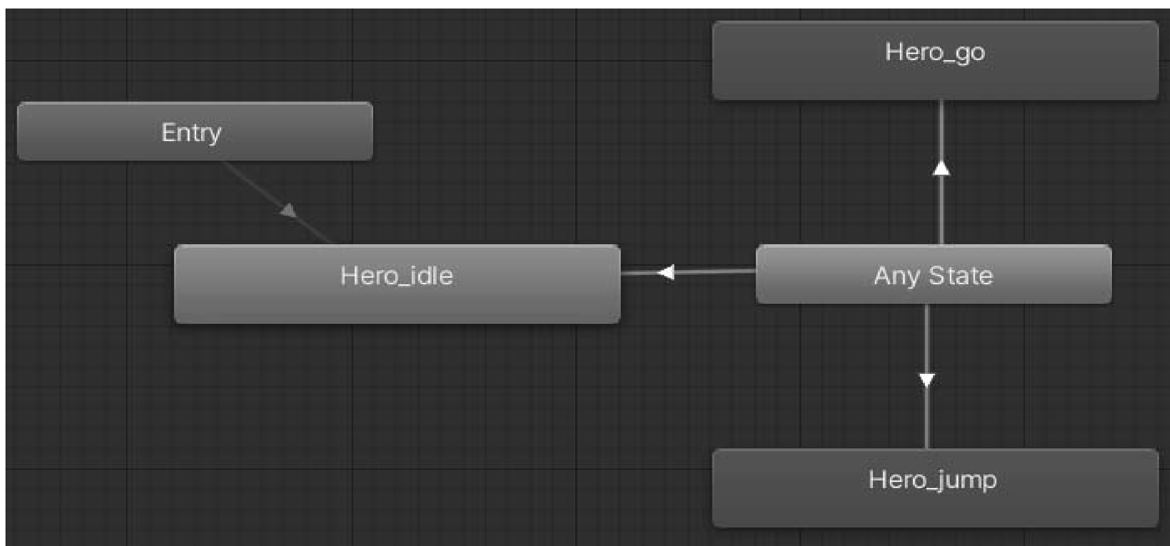

Рисунок 35 – Состояния анимации

4.6. Реализация компонента кристаллов и инвентаря

Компонент Crystal реализован классами `CrystalController`, `CrystalModel`, `CrystalView`, который наследуется от `MonoBehaviour`. Модель хранит данные кристалла, содержит свойства `Score` и `Type`, которые определяют основные характеристики объекта. Представление содержит ссылку на контроллер, а также методы для взаимодействия с контроллером. Контроллер в свою очередь управляет поведением кристалла в игре, содержит ссылки на представление и модель, управляет их взаимодействием. Контроллер обрабатывает события входа в триггер персонажем и уничтожает объект после этого, а значение счетчика в ячейке инвентаря увеличивается на 1. Все коллайдеры объектов кристаллов на сцене являются триггерами [6].

Компонент Inventory реализован классами `InventoryView` и `InventorySlot`. Класс `InventoryView` отвечает за визуальное представление инвентаря в игре. Данный класс отображает изменения о количества каждого типа предмета в инвентаре, а также управляет изменением слотов, предоставляя другим классам методы для увеличения и уменьшения количества предметов определенного типа в инвентаре. Он реализует паттерн Singleton для обеспечения единственного экземпляра этого класса в игре –

это достигается с помощью статического свойства `Instance` и проверки в методе `Awake()`. Ячейки инвентаря являются массивом, который хранит все слоты. Каждый слот содержит определенный тип предмета и количество этих предметов (листинг 9).

Листинг 9 – Код метода `UpdateText` класса `InventoryView`

```
public static InventoryView Instance {get; private set;}
 public InventorySlot[] slots;
...
 private void UpdateText()
 {foreach (var slot in slots) {
    scoreTexts[slot.locationType].text = slot.Count.ToString();}}
```

4.7. Реализация компонента платформ

Компонент состоит из классов: `PlatformController`, `PlatformView`, `PlatformModel`. Каждый тип платформы (исчезающая, движущаяся, меняющая гравитацию, направление движения персонажа, обычная) имеет свои контроллер и представление. Класс `ChangeableState` является базовым для всех игровых объектов, которые имеют два состояния отображения на сцене, и наследуется от `MonoBehaviour`. Код класса приведен в листинге 10.

Листинг 10 – Класс `ChangeableState`

```
public class ChangeableState: MonoBehaviour    {
  [SerializeField] protected SpriteRenderer stateColorless;
  [SerializeField] protected SpriteRenderer stateColor;
  protected virtual void Awake()
  {
  stateColorless = transform.GetChil d(0).GetComponent<SpriteRenderer>();
  stateColor = transform.GetChild(1).GetComponent<SpriteRenderer>();
  stateColor.gameObject.SetActive(false);}
  public virtual void ChangeState()
  { stateColor.gameObject.SetActive(true);
     stateColorless.gameObject.SetActive(false); }
  }
}
```

В этом классе `stateColorless` и `stateColor` это поля, которые хранят ссылки на компоненты `SpriteRenderer`. Эти компоненты управляют отображением спрайтов для объектов игры. Метод `ChangeState()` меняет состояние объекта. На рисунке 36 представлена диаграмма классов компонента.

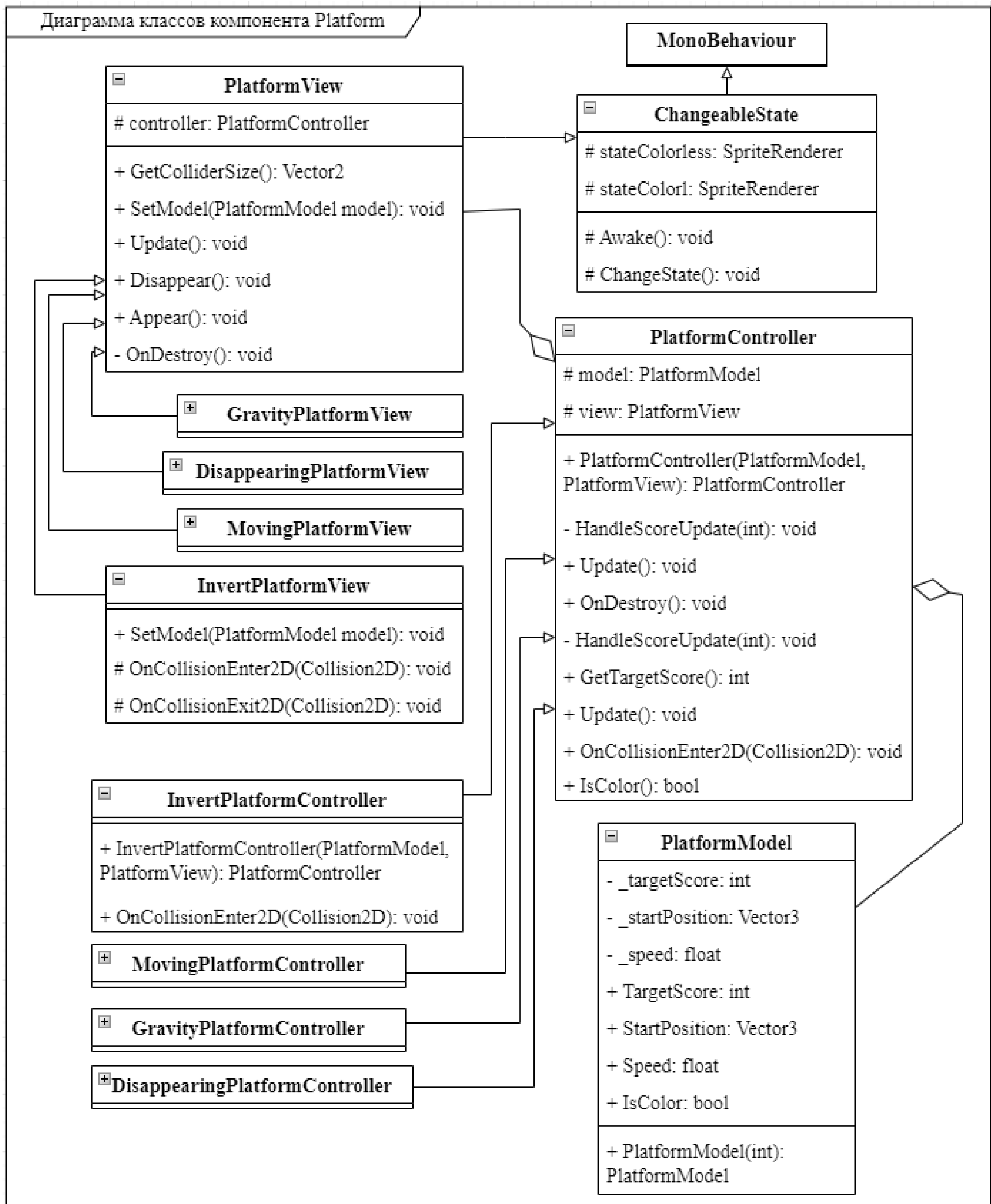

Рисунок 36 – Диаграмм классов компонента Платформ

Класс `PlatformController` управляет поведением платформы в игре. Он принимает модель и представление платформы в качестве параметров конструктора, подписывается на событие обновления счета и обрабатывает это событие, изменяя состояние платформы на «цветное» если счет достиг целевого значения.

Подвижные и исчезающие платформы реализованы с использованием корутин (Coroutines) [7]. Метод `Update()`, приведенный в листинге 11, обновляется каждый кадр и управляет движением платформы.

Листинг 11 – Метод `Update` класса `MovingPlatformController`

```
public override void Update()
{
    if (!isMoving) {view.StartCoroutine(StartMoving());}
    else{
        float verticalDistance = 4f; Vector3 targetPosition =
model.StartPosition + Vector3.up * verticalDistance * _direction;
     if(
        view.transform.position.y >= model.StartPosition.y + verticalDis-
tance && _direction > 0 ||
        view.transform.position.y <= model.StartPosition.y && _direction <
0) {_direction *= -1f;}
       view.transform.position = Vector3.MoveTowards(view.transform.posi-
tion, targetPosition, model.Speed * Time.deltaTime);
    }
}
```

Если платформа еще не движется, начинается корутина `StartMoving` (листинг 12) класса `MovingPlatformController`, если платформа уже движется, происходит перемещение платформы на заданное расстояние и направление.

Листинг 12 – Корутина `StartMoving()`

```
private IEnumerator StartMoving()
{
float delay = Random.Range(0f, 10f);
    yield return new WaitForSeconds(delay);
    isMoving = true;
}
```

4.8. Реализация компонента открываемых объектов

Компонент `OpenableObject` состоит из двух игровых объектов на сцене: сундука и дверей. Сундук реализован классами `ChestController`, `ChestModel`, `ChestView`, который наследуется от `MonoBehaviour`. Двери реализованы классами `DoorController`, `DoorModel`, `DoorView`, который наследуется от класса `ChangeableState`, описанном в предыдущем разделе.

Класс `ChestModel` – это модель данных, описывающая состояние сундука. Она содержит такие свойства, как целевой счет (`TargetScore`) для от-

крытия сундука, начальную позицию (`StartPosition`) и флаг, указывающий, открыт ли сундук (`IsOpen`). Класс `ChestView` – это представление сундука, отвечающее за визуализацию и взаимодействие с пользователем. Оно содержит ссылку на контроллер (`ChestController`), который обрабатывает логику взаимодействия с сундуком.

Класс `DoorModel` – это модель данных, описывающая состояние двери. Она содержит такие свойства, как целевой счет (`TargetScore`) для открытия двери, начальную позицию (`StartPosition`), флаг, указывающий, открыта ли дверь (`IsOpen`), а также информацию о текущей и следующей локации (`CurrentLocation` и `NextLocation`). Класс `DoorView` – это представление двери, отвечающее за визуализацию и взаимодействие с пользователем. Оно содержит ссылку на контроллер (`DoorController`), который обрабатывает логику взаимодействия с дверью.

Классы `ChestController` и `DoorController` – это контроллеры, отвечающие за обработку логики взаимодействия с соответствующими объектами (сундуком и дверью). Контроллеры обрабатывают события, такие как вход и выход «Игрока» из области взаимодействия, проверяют активный слот инвентаря и выполняют соответствующие действия (открытие сундука/двери, переход на следующий уровень).

Контроллеры также взаимодействуют напрямую с классами `CameraManager` и `InventoryView`, для обновления текста на экране и управления инвентарем «Игрока».

4.9. Реализация компонентов звука и фона

Реализация управления звуком

Компонент управления игровыми звуками `Sound` состоит из двух классов, реализующих паттерн проектирования Singleton: `SoundManager` и `SoundSettings`.

`SoundSettings` отвечает за настройку громкости музыки и звуковых эффектов в игре. Он использует слайдеры, размещенные на стартовом

экране, для изменения громкости и сохраняет эти настройки. Настройки применяются при изменении громкости и сохраняются при следующем запуске игры. Для сохранения настроек используется PlayerPrefs [21]. В листинге 13 приведен код сохранения значения громкости музыки под ключом «MusicVolume», если на событие OnMusicVolumeChanged подписаны какие-либо методы, то они вызываются с новым значением громкости в качестве аргумента.

Листинг 13 – Код метода SetMusicVolume класса SoundSettings

```
public void SetMusicVolume(float volume)
    {MusicVolume = volume;
    PlayerPrefs.SetFloat("MusicVolume", volume);
     OnMusicVolumeChanged?.Invoke(volume);
}
```

Класс SoundManager отвечает за воспроизведение и остановку звуков в игре. Он содержит массив ссылок на все звуковые компоненты в игре и применяет настройки громкости. Этот класс используется в обеих сценах: стартовой и сцене с игровым процессом. Класс InventorySlot использует SoundManager для воспроизведения звука из массива по индексу, когда количество объектов в ячейке инвентаря уменьшается или увеличивается.

Реализация компонента фона

Компонент Background состоит из трех классов: BackgroundView, BackgroundModel и BackgroundController.

Класс представления отвечает за визуальное отображение фона в игре. Он содержит ссылку на компонент Image, а также текущий цвет фона и прогресс в достижении целевого цвета. Модель хранит данные для фона – процент осветления за каждый собранный кристалл и количество очков для достижения полной окраски цветом. Контроллер управляет поведением фона в игре.

В листинге 14 приведен код метода HandleScoreUpdate, который обрабатывает обновление счета, вычисляет прогресс в достижении целевого

счета, вычисляет новый цвет фона с помощью линейной интерполяции (метод `Math.Lerp` [18]) на основе прогресса и применяет этот новый цвет к представлению фона.

Листинг 14 – Код `HandleScoreUpdate` класса `BackgroundController`

```
public void HandleScoreUpdate(int score)
{float progress = (float)score /model.TargetScore; if (progress >= 1f)
{progress = 1f; } view.T = progress; Color newColor = new Color(
    Mathf.Lerp(view.CurrentColor.r, model.TargetColor.r, view.T),
    Mathf.Lerp(view.CurrentColor.g, model.TargetColor.g, view.T),
    Mathf.Lerp(view.CurrentColor.b, model.TargetColor.b, view.T),
    view.CurrentColor.a); view.ChangeColor(newColor);}
```

4.10. Результаты реализации

Оценка качества генерации

На рисунке 37 приведено изображение локации, сгенерированной методом лабиринта.

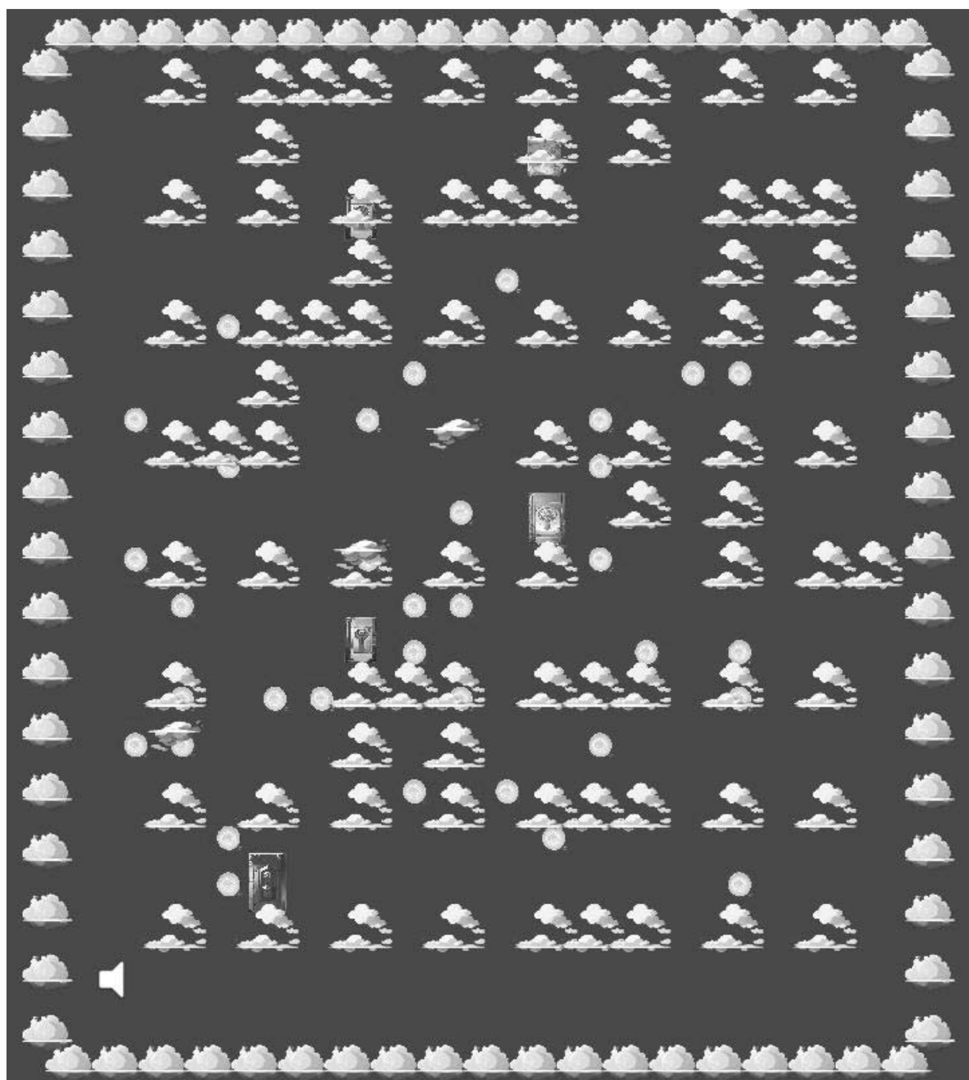

Рисунок 37 – Локация «Облачное пространство» методом лабиринта

В генераторе игры были установлены следующие параметры: случайное число для выбора типа платформы было равно 2, что привело к генерации платформы-лабиринта. Размер метки для лабиринта составил (1,00; 1,30). Ширина и высота лабиринта составили 17 единиц. Вероятность удаления платформы во время генерации лабиринта составила 0,9. Каждый кристалл в игре оценивается в 2 очка. Всего в уровне игры было сгенерировано 39 кристаллов. Для открытия двери в игровом уровне необходимо собрать 7 кристаллов.

На рисунке 38 приведено изображение локации, сгенерированной методом регионов. В генераторе игры были установлены следующие параметры: случайное число для выбора типа платформы было равно 9, что привело к генерации случайной сетки платформ. Размер метки для сетки платформ составил (2,00; 3,00). Размеры сетки составили 17 единиц в ширину и 12 единиц в высоту. Было сгенерировано 32 региона. Каждый кристалл в игре оценивается в 3 очка. Всего в уровне игры было сгенерировано 37 кристаллов. Для открытия двери в игровом уровне необходимо собрать 6 кристаллов.

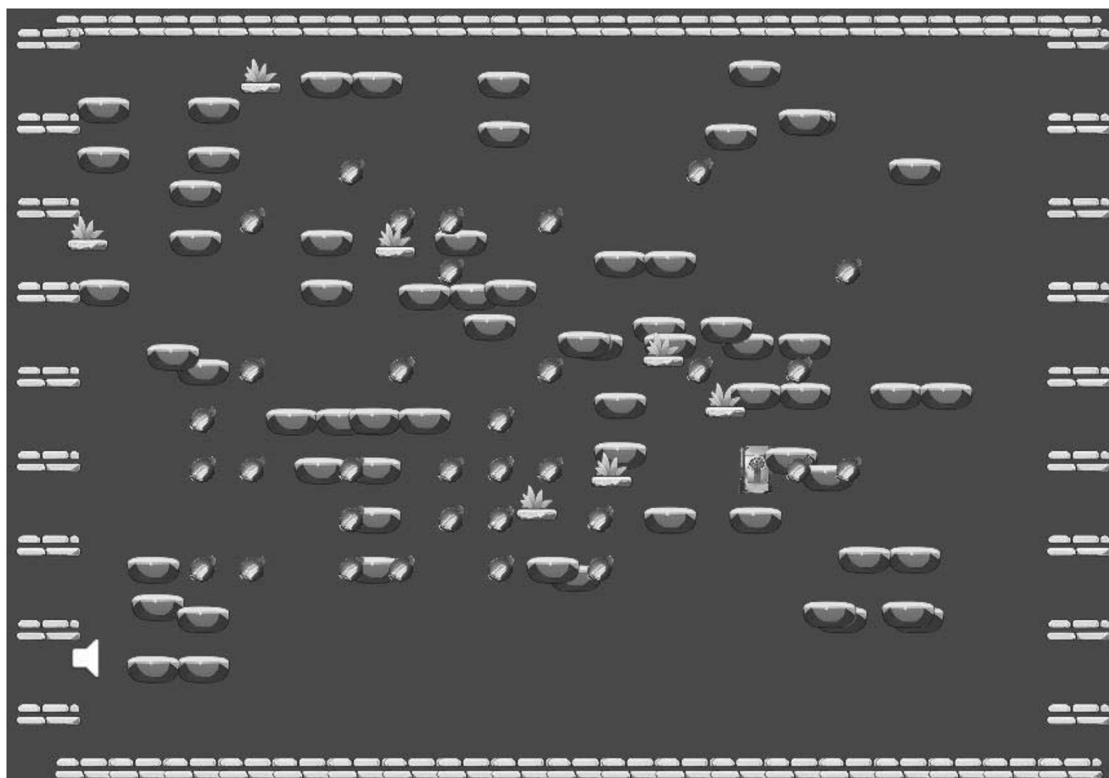

Рисунок 38 – Локация «Зеленый лес» методом регионов

Из приведенных изображений видно, что наиболее равномерно распределены платформы на локациях, сгенерированных алгоритмом создания лабиринта. Локации, сгенерированные вторым методом, имеют более открытые и более закрытые области для перемещения персонажа. Также сгенерированный игровой мир может частично скрывать некоторые кристаллы за платформами. Исходя из требований к проектируемому генератору, были выполнены следующие условия:

1) генератор выполняет все действия, определенные на этапе проектирования;

2) всеми хорошими качествами обладает генерация локации на основе лабиринта;

3) генерация на основе размещения меток не соответствует хорошему требованию о равномерном расположении платформ на локации, однако данный результат является допустимым, поскольку создает для «Игрока» дополнительный уровень сложности прохождения;

4) отсутствуют недопустимые результаты генерации.

Результат реализации плавного изменения цвета локации

В инвентаре выбранная ячейка с кристаллами является активной и подсвечивается цветом, соответствующим типу кристалла. На рисунке 39 приведены все 5 состояний строки инвентаря в интерфейсе в зависимости от выбранного типа кристаллов.

Рисунок 39 – Инвентарь в зависимости от активного типа кристалла

На рисунке 40 приведены изображения изменения игрового мира в зависимости от текущего процента прохождения комнаты.

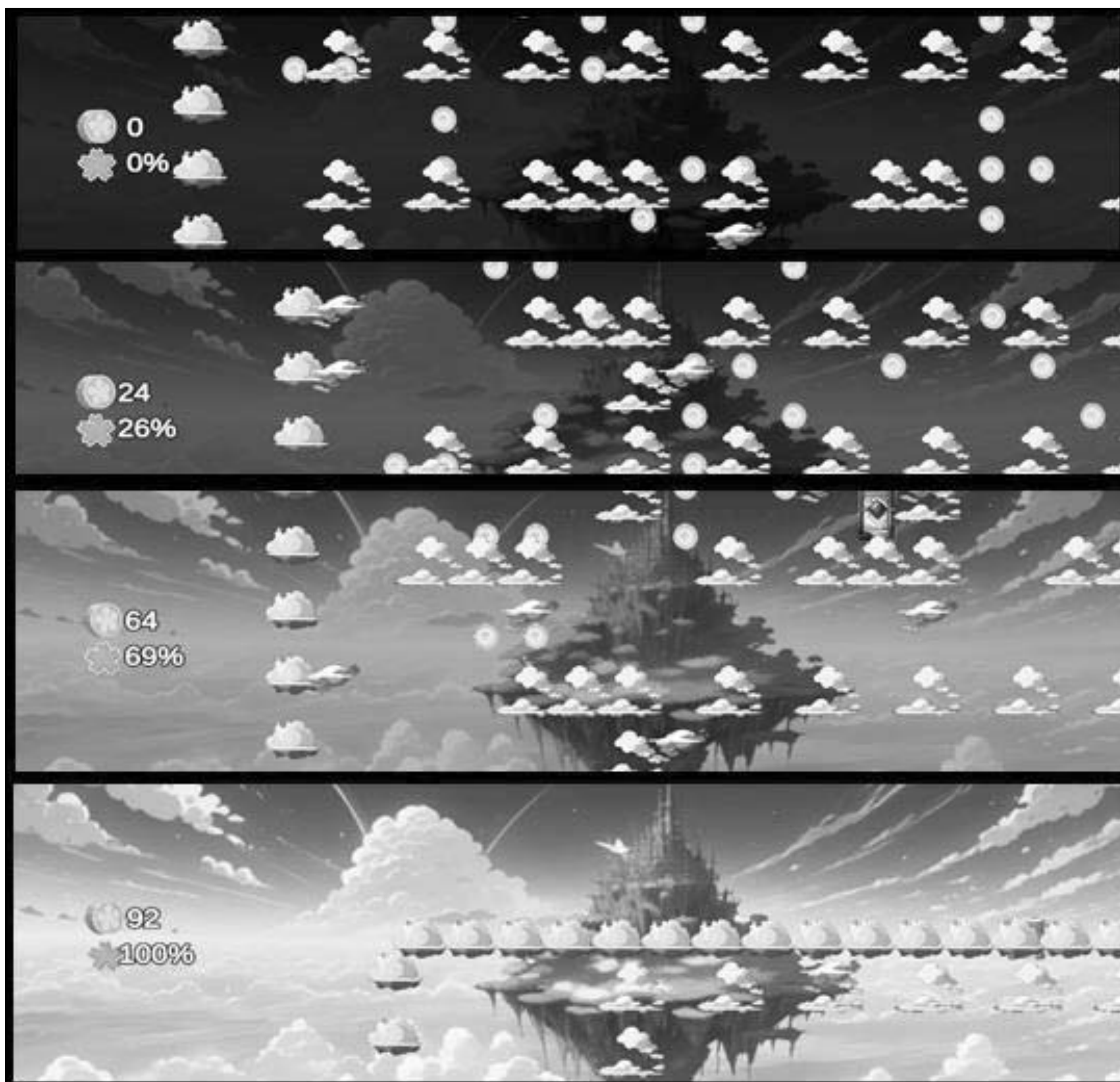

Рисунок 40 – Процесс изменения игрового мира от 0% до 100%

Реализация игровых сцен

На рисунке 41 приведена реализация стартового экрана игрового приложения. Когда «Игрок» выбирает уровень сложности, ячейка с цифрой становится отмечена галочкой.

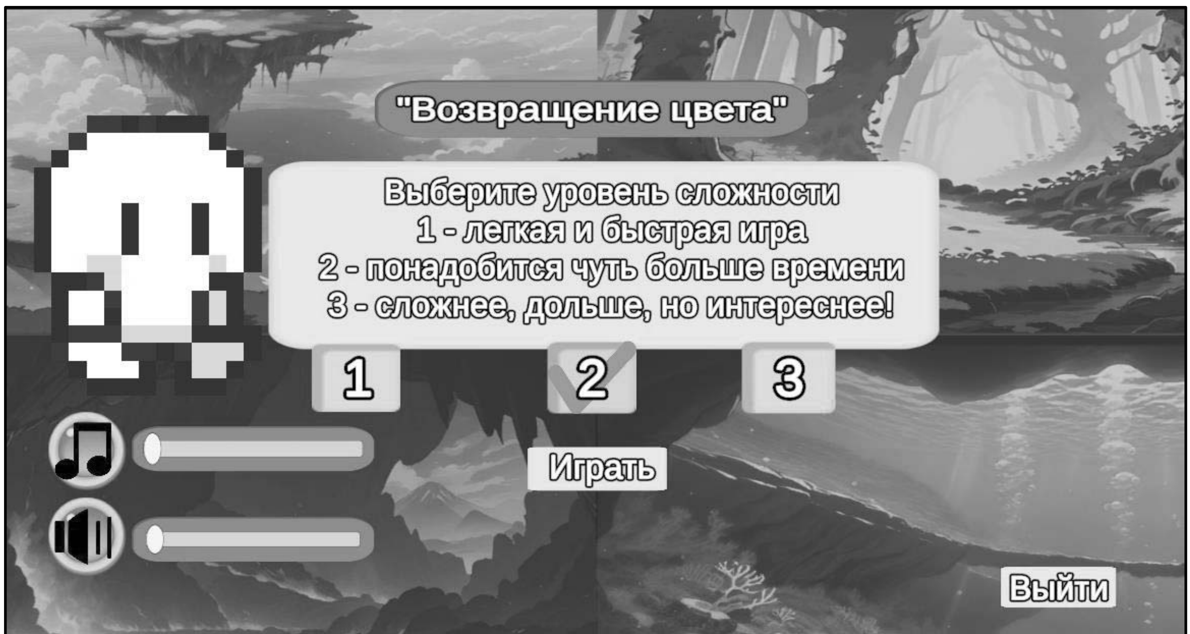

Рисунок 41 – Стартовый игровой экран

На рисунке 42 приведена реализация окна справки игрока, которое «Игрок» видит после того, как выберет уровень сложности и нажмет кнопку «Играть». На данном окне показана игровая цель, два состояния каждой двери. Порядок дверей соответствует порядку изображения кристаллов.

Рисунок 42 – Окно справки игрока

На рисунке 43 изображено 4 типа возможных состояний подсказок для «Игрока», когда персонаж осуществляет взаимодействие с объектом двери в ситуациях, когда:

1) выбран неверный слот инвентаря;

2) выбранный слот инвентаря может открыть дверь, однако не хватает кристаллов;

3) выбран верный слот инвентаря, кристаллов достаточно и дверь ранее не была открыта;

4) состояние, когда дверь была ранее открыта (она является цветной и персонаж может использовать ее для перемещения в другую комнату).

Рисунок 43 – Реализация всплывающих подсказок для игрока

На рисунке 44 приведена реализация основного игрового экрана.

Рисунок 44 – Основной игровой экран

На рисунке 45 приведена реализация окна меню, в которое «Игрок» попадает, когда нажимает кнопку «Esc» на клавиатуре. В данном окне «Игрок» может с помощью предложенных слайдеров изменить громкость музыки и звука, после чего вернуться к игровому процессу или завершить его. При изменении громкости музыки и звука происходит сохранение текущих параметров. При повторном запуске игрового приложения, параметры сохраняют свои значения.

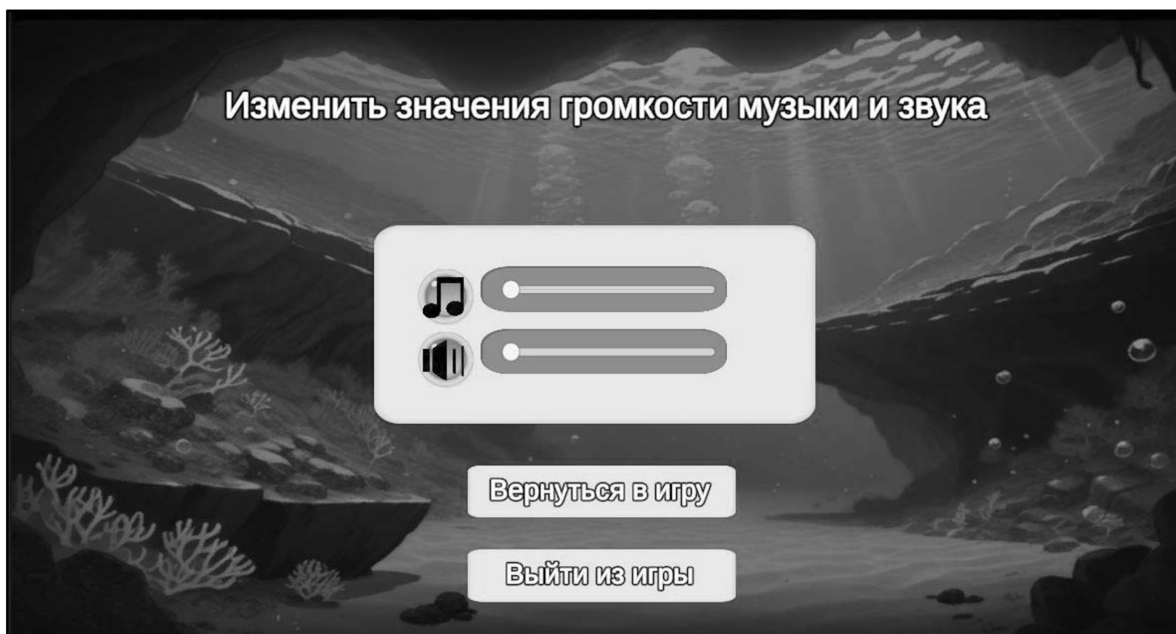

Рисунок 45 – Окно меню во время игрового процесса

На рисунке 46 приведено окно с выводом результатов о прохождении игры, когда «Игрок» откроет сундук и нажмет Enter для выхода. Окно показывает информацию обо всех игровых комнатах. Информация содержится в виде списка комнат с указанием индекса и процента прохождения в ней, а также суммарное количество очков, которое «Игрок» собрал за время игры. Для каждой комнаты отображен процент как количество собранных очков от общего количества очков. После того, как «Игрок» нажимает кнопку «ОК», игровое приложение загружает стартовую сцену, при этом до закрытия основной игровой сцены происходит отписка от всех событий внутри каждого компонента для освобождения ресурсов системы.

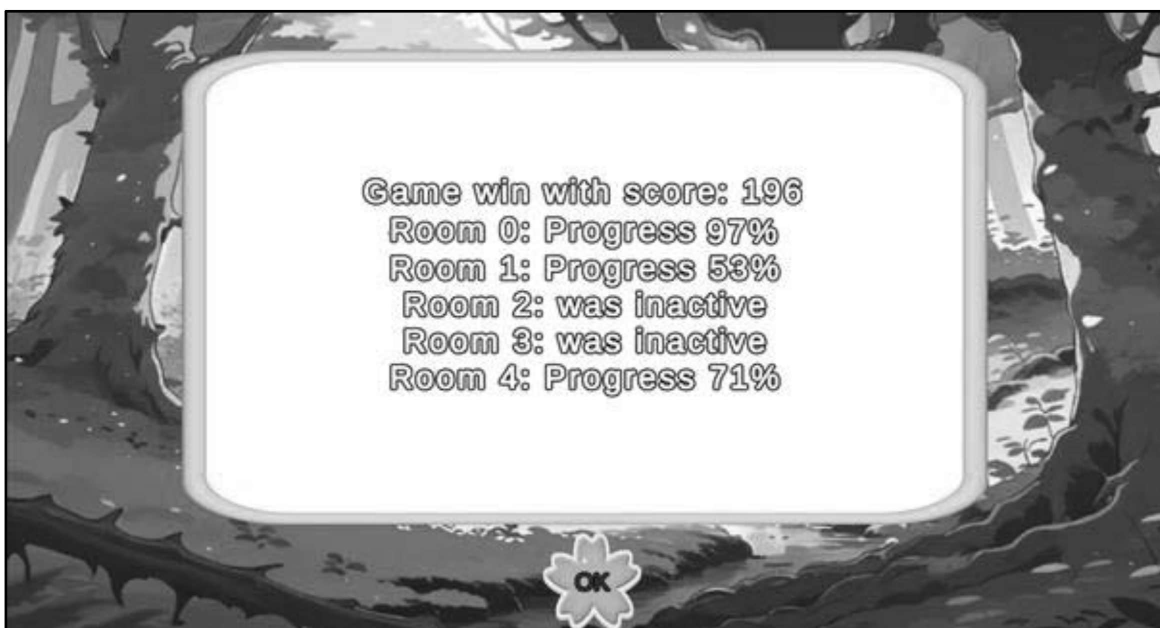

Рисунок 46 – Окно результатов прохождения игры

Выводы по четвертой главе

В данной главе были приведены используемые средства реализации, библиотеки, используемые готовые наборы изображения игровых объектов. Приведена реализация основных алгоритмов генератора, рассчитаны временная и пространственная сложности. Также приведена реализация всех компонентов, описанных на этапе проектирования. Была произведена оценка результатов генерации и изображения основных механик и элементов игрового мира. Приведены изображения реализации пользовательского интерфейса.

5. ТЕСТИРОВАНИЕ

5.1. Тестирование требований

В данном разделе приводится тестирование функциональных требований. Проводится проверка нефункциональных требований.

В таблице 2 приведено тестирование главного экрана стартовой игровой сцены.

Таблица 2 – Тестирование управления стартовой сцены

№ теста	Описание теста	Результат	Тест пройден?
1.	Игрок нажал кнопку «Играть» и не выбрал уровень сложности.	Игровой процесс не запустился	Да
2.	Пользователь выбрал уровень сложности.	Выбранный уровень сложности был визуально выделен. При наличии, предыдущий выбор перестал быть выделенным.	Да
3.	Игрок нажал на кнопку «Играть», выбран уровень сложности.	Открылось окно со справкой игрока.	Да
4.	Игрок нажал на кнопку «Выйти».	Игровое приложение завершило работу.	Да
5.	Игрок изменил положение слайдера.	Изменен уровень громкости проигрываемой музыки или звуковых эффектов.	Да
6.	Игрок нажал на кнопку «ОК» в окне «Справка игрока»	Приложение загрузило основную игровую сцену с первой процедурно сгенерированной комнатой.	Да

В таблице 3 приведено тестирование управления персонажем и инвентарем, а также изменением настроек внутри игры.

Таблица 3 – Тестирование управления игровой сцены

№ теста	Описание теста	Результат	Тест пройден?
1.	Игрок нажал стрелку вправо или клавишу A, не находясь на платформе, инвертирующей движение.	Персонаж был перемещен вправо	Да
2.	Игрок нажал стрелку вправо или клавишу A, находясь на платформе, инвертирующей движение.	Персонаж был перемещен влево	Да
3.	Игрок нажал стрелку влево или клавишу D, не находясь на платформе, инвертирующей движение.	Персонаж был перемещен влево	Да

Окончание таблицы 3

№ теста	Описание теста	Результат	Тест пройден?
4.	Игрок нажал стрелку влево или клавишу D, находясь на платформе, инвертирующей движение.	Персонаж был перемещен вправо	Да
5.	Игрок нажал пробел.	Персонаж был перемещен вверх	Да
6.	Игрок нажал два пробела подряд.	Персонаж совершил двойной прыжок.	Да
7.	Игрок вошел в зону действия кристалла.	Объект был уничтожен со сцены, был воспроизведен звуковой эффект, увеличился счет игрока и процент прохождения уровня.	Да
8.	Игрок нажал на клавиши 1, 2, 3, 4, 5	Был активирован соответствующий слот инвентаря. Остальные слоты стали неактивными.	Да
9.	Игрок нажал F и были выполнены все условия для использования предмета инвентаря.	Счетчик активной ячейки инвентаря уменьшился. Был воспроизведен звуковой эффект.	Да
10.	Игрок нажал F, но не были выполнены все условия для использования предмета инвентаря.	Счетчик выбранной ячейки инвентаря не изменился. Изменения в игровом мире не произошли.	Да
11.	Игрок нажал Enter, и были выполнены все условия для использования предмета инвентаря.	Приложение загрузило соответствующую игровую комнату или вывела для игрока информацию с результатами прохождения игры.	Да
12.	Игрок нажал Enter, но не были выполнены все условия для использования предмета инвентаря.	Изменения в игровом мире не произошли.	Да
13.	Игрок нажал на Esc на клавиатуре, окно меню не было открыто	Игровое приложение открыло окно меню игрового процесса	Да
14.	Игрок нажал на Esc на клавиатуре, окно меню было открыто	Игровое приложение закрыло окно меню игрового процесса	Да
15.	Игрок изменил положение слайдера в окне меню.	Изменен уровень громкости проигрываемой музыки или звуковых эффектов.	Да

Таблица 4 содержит информацию о соответствии игрового приложения поставленным нефункциональным требованиям.

Таблица 4 – Проверка нефункциональных требований

№ теста	Описание теста	Результат	Тест пройден?
1.	Игровое приложение скомпилировано для ОС Windows и запущено на Windows 10.	Приложение запущено без ошибок	Да
2.	Игровое приложение скомпилировано для ОС Linux и запущено на Arch Linux.	Приложение запущено без ошибок	Да
3.	Игровое приложение скомпилировано для ОС Linux и запущено на Linux Ubuntu.	Приложение запущено без ошибок	Да
4.	Игровое приложение скомпилировано с поддержкой частоты кадров 60 fps.	Приложение обеспечивает плавный переход кадров.	Да
5.	Использование библиотеки QuikGraph для NetStandart 2.0 и выше. Приложение скомпилировано с использованием API Net Standard 2.1.	Приложение использует только API Net Standard 2.1 и успешно компилируется.	Да

5.2. Юзабилити-тестирование

В юзабилити-тестировании [39] принимали участие 4 группы пользователей по 2 человека, таким образом, тестирование происходило в 4 этапа, где каждый последующий этап был основан на изменениях в предыдущем.

При тестировании перед игроками ставились следующие задачи:

1) пройти игру по одному разу на каждом из доступных уровней сложности;

2) посетить максимально возможное количество комнат;

3) визуально проверить, соблюдается ли расстояние не менее одной платформы между сундук и дверями;

4) измерить время прохождения игры на каждом уровне сложности.

В ходе первого этапа тестирования был изменен способ обработки прыжка персонажа. Была добавлена нижняя граница игрового мира для обработки ситуации падения, при котором игровой процесс завершался сразу проигрышем.

В ходе второго этапа тестирования было принято решение добавить справку игрока, содержащей информацию об управлении, правилах и конечной игровой цели.

В ходе третьего этапа тестирования проводилась настройка количественных параметров генератора (изначальные параметры приведены в таблице 1 в главе проектирования). Были внесены изменения в пользовательский интерфейс, чтобы сделать его визуально более приятным и понятным.

После внесения изменений, последней новой группе пользователей было предложено пройти игру на каждом из трех уровней сложности. Среднее время прохождения составило: 10, 20 и 25 минут при первом знакомстве с приложением на каждом уровне сложности. Повторное прохождение игры происходило быстрее, что обусловлено накоплением у пользователей игрового опыта.

На 3 уровне сложности возникла ситуация, при которой «Игрок» «заблудился» между комнатами, а количество кристаллов оказалось недостаточным, чтобы вернуться в первую комнату для открытия сундука.

Общее субъективное мнение пользователей оценивает игру как расслабляющую, увлекательную для прохождения. Было отмечено, что отсутствие элементов сражения и противников позволяет сосредоточиться на посещение новых комнат, сборе предметов и исследовании игрового мира.

5.3. Изменение баланса игры

В данном разделе приводится перечень изменений после юзабилити-тестирования. Приводится методика изменения баланса игры после проведения нескольких игр для достижения оптимального уровня сложности.

В зависимости от выбранного уровня сложности, в комнате находится N дверей разного типа, при этом всего есть K комнат, а общее количество дверей составляет $2K$, так как в каждую комнату есть вход и выход. Чтобы открыть одну дверь, необходимо m кристаллов. На одной локации можно

собрать M кристаллов. Значение m выбирается как определенный процент (приведено в таблице 1) от M.

Игра является проходимой по всем комнатам на первом уровне сложности. На втором уровне сложности было принято решение понизить процент от M с 30 до 25, чтобы из комнаты гарантированно можно было открыть от трех до четырех дверей.

На третьем уровне сложности было принято решение понизить процент от M с 50 до 30, чтобы можно было открыть 3 двери и при этом сохранить часть кристаллов для накопления и открытия дополнительной двери. При этом не все комнаты могут быть посещены, если «Игрок» не оптимально распределит кристаллы.

Таким образом, «Игрок» должен стремиться к оптимальному распределению кристаллов, чтобы максимизировать количество открытых дверей. Это может быть достигнуто путем выбора дверей, которые ведут к комнатам с большим количеством кристаллов, или путем сохранения кристаллов для открытия дверей в комнатах, где все кристаллы уже были собраны.

Выводы по пятой главе

Было проведено юзабилити-тестирование игрового приложения в соответствии с функциональными и нефункциональными требованиями. Все тесты были успешно пройдены. Также было проведено изменение баланса игры после тестирования.

ЗАКЛЮЧЕНИЕ

В рамках данной работы была разработана игра в жанре «Платформер» с процедурной генерацией игрового мира на платформе Unity. В ходе работы были выполнены следующие задачи:

1) проведен анализ предметной области;
2) выполнен обзор алгоритмов процедурной генерации;
3) спроектировано игровое приложение;
4) реализовано игровое приложение;
5) проведено тестирование игрового приложения.

Таким образом, было разработано игровое приложение с внедрением пяти алгоритмов процедурной генерации игрового контента, все поставленные задачи были выполнены.

В дальнейшем планируется внедрение для третьего уровня сложности системы противников, которые будут состоять из врагов и боссов, которые будут появляться в определенных комнатах. Также возможно добавление системы генерации квестов и сценариев на основе применения контекстно-свободных грамматик и адаптивности квестов на действия игрока. Для разнообразия игровой территории комнат возможно добавление процедурно сгенерированных структур и зданий с внутриигровыми событиями.

Дополнительно рассматривается возможность внедрения многопользовательского режима, который позволит игрокам осуществлять совместное прохождение игровых комнат. Предполагается разработка системы достижений и наград.

ЛИТЕРАТУРА

1. An Exploration of Procedural Content Generation for Top-Down Level Design | Chalmers Publication Library. [Электронный ресурс] URL: https://publications.lib.chalmers.se/records/fulltext/256132/256132.pdf (дата обращения: 05.02.2024 г.).
2. Animator | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/ScriptReference/Animator.html (дата обращения: 07.04.2024 г.).
3. Barabasi Albert Graph (for Scale Free Models) | Geeksforgeeks. [Электронный ресурс] URL: https://www.geeksforgeeks.org/barabasi-albert-graph-scale-free-models/ (дата обращения: 06.02.2024 г.).
4. Benefits of using design patterns | Unity. [Электронный ресурс] URL: https://unity.com/how-to/build-modular-codebase-mvc-and-mvp-programming-patterns (дата обращения: 20.03.2024 г.).
5. Cherry blossom GUI pack | Unity Asset Store. [Электронный ресурс] URL: https://assetstore.unity.com/packages/2d/gui/icons/cherry-blossom-gui-pack-147391 (дата обращения: 07.04.2024 г.).
6. Collider.isTrigger | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/ScriptReference/Collider-isTrigger.html (дата обращения: 07.04.2024 г.).
7. Coroutines | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/Manual/Coroutines.html (дата обращения: 01.04.2024 г.).
8. Depth First Search or DFS for a Graph | Geeksforgeeks. [Электронный ресурс] URL: https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/ (дата обращения: 06.02.2024 г.).
9. DungeonMaker. [Электронный ресурс] URL: https://dungeon maker.sourceforge.net/manual/index.html (дата обращения: 08.04.2024 г.).

10. Eklavya Sharma. Context-free Grammars | Github. [Электронный ресурс] URL: https://sharmaeklavya2.github.io/notes/theory-of-computing/context-free-grammars.pdf (дата обращения: 06.02.2024 г.).

11. Event Bus pattern | Github. [Электронный ресурс] URL: https://medium.com/elixirlabs/event-bus-implementation-s-d2854a9fafd5 (дата обращения: 01.04.2024 г.).

12. Generating a 2D map using the Random Walk algorithm | Noveltech. [Электронный ресурс] URL: https://www.noveltech.dev/procgen-random-walk (дата обращения: 06.02.2024 г.).

13. Grid-based Graph Drawing with ILP/SAT Modeling | Algorithms and Complexity Group. [Электронный ресурс] URL: https://www.ac.tuwien.ac.at/wp/wp-content/uploads/Martin-Nöllenburg-ilpsat.pdf (дата обращения: 06.02.2024 г.).

14. How to Use BSP Trees to Generate Game Maps | Tutsplus. [Электронный ресурс] URL: https://gamedevelopment.tutsplus.com/how-to-use-bsp-trees-to-generate-game-maps--gamedev-12268t (дата обращения: 05.02.2024 г.).

15. Jam Sites by Year | GLOBAL GAME JAM. [Электронный ресурс] URL: https://globalgamejam.org/jam-sites (дата обращения: 01.02.2024 г.).

16. Kruskal's Minimum Spanning Tree (MST) Algorithm | Geeksforgeeks. [Электронный ресурс] URL: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/ (дата обращения: 06.02.2024 г.).

17. Mapgen: Tunneling Algorithm | Gridsagegames. [Электронный ресурс] URL: https://www.gridsagegames.com/blog/2014/06/mapgen-tunneling-algorithm/ (дата обращения: 06.02.2024 г.).

18. Mathf.Lerp | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/ScriptReference/Mathf.Lerp.html (дата обращения: 08.04.2024 г.).

19. Minimum Spanning Trees | Algorithms, 4th Edition. [Электронный ресурс] URL: https://algs4.cs.princeton.edu/lectures/keynote/ 43MinimumSpanningTrees.pdf (дата обращения: 06.02.2024 г.).

20. MonoBehaviour | Unity Documentation. [Электронный ресурс]: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html (дата обращения: 21.03.2024 г.).

21. PlayerPrefs | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/ScriptReference/PlayerPrefs.html (дата обращения: 08.04.2024 г.).

22. Playground AI. [Электронный ресурс] URL: https://playground.com (дата обращения: 21.03.2024 г.).

23. Poisson disk sampling through disk packing / G. Liang, L. Lu, Zh. Chen, Ch. Yang. // Computational Visual Media, 2015. – Vol. 1, No. 1. – 17–26 pp.

24. Prefabs | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/Manual/ Prefabs.html (дата обращения: 01.04.2024 г.).

25. Prim's Algorithm for Minimum Spanning Tree (MST) | Geeksforgeeks. [Электронный ресурс] URL: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/ (дата обращения: 06.02.2024 г.).

26. Procedural Content Generation using Behavior Trees (PCGBT) | Github. [Электронный ресурс] URL: https://riffsircar.github.io/assets/pdfs/sarkar2021pcgbt.pdf (дата обращения: 10.02.2024 г.).

27. QuikGraph | NuGet Gallery. [Электронный ресурс] URL: https://www.nuget.org/packages/QuikGraph#supportedframeworks-body-tab (дата обращения: 21.03.2024 г.).

28. Simple 2D Platformer Assets Pack | Unity Asset Store. [Электронный ресурс] URL: https://assetstore.unity.com/packages/2d/characters/simple-2d-platformer-assets-pack-188518 (дата обращения: 07.04.2024 г.).

29. Singletons in Unity | Gamedevbeginner. [Электронный ресурс] URL: https://gamedevbeginner.com/singletons-in-unity-the-right-way/ (дата обращения: 07.04.2024 г.).

30. Small-World Graphs: characterization and alternative constructions | Rama CONT: Research Publications. [Электронный ресурс] URL: http://rama.cont.perso.math.cnrs.fr/pdf/ContTanimura.pdf (дата обращения: 06.02.2024 г.).

31. The Random Graph | Cornell University. [Электронный ресурс] URL: https://arxiv.org/pdf/1301.7544.pdf (дата обращения: 06.02.2024 г.).

32. Unity Asset Store. [Электронный ресурс] URL: https://assetstore.unity.com (дата обращения: 21.03.2024 г.).

33. Unity Documentation. [Электронный ресурс] URL: https://docs.unity.com (дата обращения: 29.01.2024 г.).

34. Using the Universal Render Pipeline | Unity Documentation. [Электронный ресурс] URL: https://docs.unity3d.com/Manual/universal-render-pipeline.html (дата обращения: 14.02.2024 г.).

35. WaveFunctionCollapse | Github. [Электронный ресурс] URL: https://github.com/mxgmn/WaveFunctionCollapse (дата обращения: 06.02.2024 г.).

36. Yasemin Ozkan Aydın, Kemal Leblebicioglu. A fast and practical grid-based algorithm for point-feature label placement problem | Cornell University. [Электронный ресурс] URL: https://arxiv.org/pdf/1712.05936.pdf (дата обращения: 06.02.2024 г.).

37. Адаптивная процедурная генерация при помощи алгоритма WaveFunctionCollapse и априорного распределения вероятностей. // Pvsm. [Электронный ресурс] URL: https://www.pvsm.ru/algoritmy/342804 (дата обращения: 06.02.2024 г.).

38. Арлоу Д., Нейштадт И. UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование, 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2007. – 624 с.

39. Барнум, К. М. Основы юзабилити-тестирования / К. М. Барнум; перевод с английского Д. А. Беликова. – Москва: ДМК Пресс, 2022. – 408 с.

40. Гейминг в России – 2022. Социальные и экономические эффекты. | Аналитический центр НАФИ. [Электронный ресурс] URL: https://nafi.ru/projects/it-i-telekom/geyming-v-rossii-2022-sotsialnye-i-ekonomicheskie-effekty/ (дата обращения: 13.02.2024 г.).

41. Генерация подземелий в Binding of Isaac. [Электронный ресурс] URL: https://habr.com/ru/articles/519658/ (дата обращения 12.02.2024 г.).

42. Официальная страница Github. [Электронный ресурс] URL: https://github.com (дата обращения: 20.03.2024 г.).

43. Официальная страница Unity 2022.3.19 | Unity. [Электронный ресурс] URL: https://unity.com/releases/editor/whats-new/2022.3.19 (дата обращения: 20.03.2024 г.).

44. Официальная страница Unity Hub. [Электронный ресурс] URL: https://unity.com/unity-hub (дата обращения: 20.03.2024 г.).

45. Официальная страница Visual Studio Code. [Электронный ресурс] URL: https://code.visualstudio.com (дата обращения: 20.03.2024 г.).

46. Официальная страница игры Celeste на платформе Steam. [Электронный ресурс] URL: https://store.steampowered.com/app/504230/Celeste/ (дата обращения 12.02.2024 г.).

47. Официальная страница игры Darkest Dungeon на платформе Steam. [Электронный ресурс] URL: https://store.steampowered.com/app/262060/Darkest_Dungeon/ (дата обращения 12.02.2024 г.).

48. Официальная страница игры Fez на платформе Steam. [Электронный ресурс] URL: https://store.steampowered.com/app/224760/FEZ/ (дата обращения 12.02.2024 г.).

49. Официальная страница игры Rogue Legacy на платформе Steam. [Электронный ресурс] URL: https://store.steampowered.com/app/241600/Rogue_Legacy/ (дата обращения 12.02.2024 г.).

50. Официальная страница игры Shovel Knight: Treasure Trove на платформе Steam. [Электронный ресурс] URL: https://store.steampowered .com/app/250760/Shovel_Knight_Treasure_Trove/ (дата обращения 10.02.2024 г.).

51. Официальная страница игры UnEpic на платформе Steam. [Электронный ресурс] URL: https://store.steampowered.com/app/233980/UnEpic/ (дата обращения 12.02.2024 г.).

52. Официальный сайт проекта Braid. [Электронный ресурс] URL: https://braid-game.com/ (дата обращения 12.02.2024 г.).

53. Официальный сайт проекта Hollow Knight. [Электронный ресурс] URL: https://www.hollowknight.com/ (дата обращения 12.02.2024 г.).

54. Официальный сайт проекта The Binding of Isaac. [Электронный ресурс] URL: https://bindingofisaac.com/ (дата обращения 12.02.2024 г.).

55. Шорт Т.Х., Адамс Т. Процедурная генерация в гейм-дизайне. // М.: ДМК Пресс, 2020. – 344 с.

56. Сайт игры Banjo Panda. [Электронный ресурс] URL: https://eddieone.itch.io/banjo-panda (дата обращения: 01.04.2024 г.).

57. Сайт игры Super Mario Bros. [Электронный ресурс] URL: https://supermarioplay.com (дата обращения: 01.04.2024 г.).

58. Чеботов Н.А. Генерация ландшафта с применением алгоритма шума Перлина в среде разработки Unity. // Виртуальное моделирование, прототипирование и промышленный дизайн: Материалы VII Международной научно-практической конференции, Тамбов, 12–14 октября 2021 года. Том Выпуск 7. – Тамбов: Издательский центр ФГБОУ ВО «Тамбовский государственный технический университет», 2021. – С. 82–87.