СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Понятие Игрового Контента [1] можно отнести ко всем элементам игры, которые так или иначе влияют на геймплей. Это определение включает в себя такие аспекты как ландшафт, карты, уровни, диалоги, квесты, персонажей, наборы правил, музыку, оружие и прочее. С момента зарождения игровой индустрии, большая часть данного контента создается вручную художниками, и так как технологии не стоят на месте, то с каждым годом качество такого контента повышается и приближается к фильмографическому.

Однако, с повышением качества растет и стоимость создания чего-либо, как денежная, так и в отношении времени. А учитывая, что художники зачастую работают с инструментами отдельными от сред игровой разработки, то нередки случаи возникновения цикла создание-конвертация-тестирование, чтобы добиться требуемого воздействия на ощущения пользователя внутри игры. Поэтому, чтобы снизить затраты на разработку игрового контента, все чаще применяется Процедурная Генерация Контента.

Процедурная Генерация Контента для Игр (Procedural Content Generation for Games, PCG-G) подразумевает алгоритмическое создание игрового контента, отбор интересных экземпляров из всех сгенерированных, и выбор тех, что могут предоставить необходимый уровень качества для игроков. Такой подход позволяет создавать контент автоматически, тем самым снижая нагрузку на художников, но он тоже имеет свои сложности: генерация контента требует не только вычислительной мощности, но и способа для оценки технической и культурной ценности экземпляров. Поэтому, не удивительно, что за три десятилетия исследований и разработки, игровое сообщество все еще не изобрело генератора общего назначения. Следовательно, хотя некоторые методы генерации контента и становятся обычной практикой в игровой индустрии, но по большей части они специализируются на определенном контексте и игровых элементах. Например, SpeedTree становится стандартным средством

процедурной генерации деревьев, что видно на примере таких игр, как Grand Theft Auto IV (RockStar Games), Batman: Arkham Asylum (Eidos Interactive).

Не стали исключением из этой ситуации и подземелья. Будучи одним из самых распространенных элементов популярных жанров игр, подземелья прошли весь путь от их зарождения в виде текстовых форматов и двумерной репрезентации до полноценных трехмерных уровней, для создания которых крупные игровые компании нанимают профессионалов [2]. Однако, несмотря на свою популярность, трехмерные подземелья, в отличие от двумерных, не так часто рассматриваются со стороны процедурной генерации. Согласно исследованию [3], за последние 10 лет лишь несколько статей затронули генерацию не только двумерных, но и трехмерных подземелий. Стоит также заметить, что [4] и [5] подходы не генерируют трехмерные подземелья как таковые, а просто создают каркас, на который в последующем применяются модели комнат и коридоров, созданные заранее дизайнером. Да и в статье [6], автор тоже генерирует двумерный набросок, из которого выдавливается трехмерное пространство. В итоге, из всех генераторов, только один [7] создает реальное трехмерное подземелье и у него есть свои ограничения. А именно, данное подземелье не поддерживает ни разграничения комнат, ни каких-либо маркеров для расстановки игровых элементов.

Таким образом, учитывая что технологии не стоят на месте, а эпоха двумерных подземелий постепенно уходит, возникает необходимость в решении, которое будет не только генерировать подземелья в трех плоскостях, но и учитывать маркеры в пространстве, позволяя дизайнерам иметь больше контроля над результатом. Для разработки такого решения, в данной работе сначала будут рассмотрены процедурная генерация контента и подземелья по-отдельности, затем алгоритмы, которые могут связать два этих понятия в единое целое. После чего будет предложен концепт метода процедурной генерации многоуровневых подземелий с его последующей реализацией.

1 ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Процедурно Генерируемый контент

Игровой контент, который поддается процедурной генерации, был разделен [8] на шесть основных категорий:

1) Игровые биты – элементарные единицы игрового контента. Они могут быть обязательными, такие как текстурные различия между игроком и противниками, и опциональными, например фоновая музыка. Но ни те, ни другие, будучи рассматриваемыми вне контекста, обычно не подвергнуты прямому воздействию со стороны игрока. Основными игровыми элементами, которые можно отнести к данному типу являются текстуры, растительность, постройки, звуковые и видео эффекты.
2) Игровое пространство – место или локация, где происходит игровое событие, заполненная игровыми битами, среди которых ориентируется игрок. Данная категория включает в себя закрытые помещения, открытые пространства и водные тела.
3) Игровые системы – симуляции более сложных игровых сред, таких как экосистемы, дорожные сети, населённые местности и взаимодействие различных элементов, которые делают игровой мир более живым.
4) Игровые сценарии – способы организации ранее перечисленных категорий для описания последовательности развития игровых событий. Они включают в себя пазлы, истории, задания и уровни.
5) Игровой дизайн – набор правил, целей и механик игры.
6) Дополнительный контент – неигровой контент, сопутствующий игровому миру. Например, таблицы очков и оповещающие сообщения.

На основе этих категорий можно выделить определенные группы, к

которым тот или иной контент может быть определен [9].

1.1.1 Обязательный и опциональный контент

Первый признак по которому можно различать контент – является он необходимым или нет. Обязательный контент необходим игроку для дальнейшего прогресса. Например, подземелья, которые надо преодолеть, монстры, которых надо одолеть, важные игровые правила и т.д. В то же время опциональный контент – это тот, который игрок может выборочно пропустить, например, доступные оружия или здания которые можно проигнорировать. Ключевая разница в том, что обязательный контент всегда должен быть валидным, а именно ни при каких обстоятельствах не мешать игроку прогрессировать на уровне. С другой стороны, если оружие или здание не влияют значительно на возможности игрока, то допускаются ошибки в их генерации.

1.1.1 Онлайн и оффлайн контент

Другое различие контента основано на том, выполняется ли его генерация в реальном игровом времени или же оффлайн во время разработки. Примером первого может служить генерация внутренности здания когда игрок открывает дверь и входит в него, в противном случае алгоритм генерирует расстановку мебели заранее и художник потом ее редактирует и выпускает вместе с игрой. Также, возможен и усредненный вариант, когда алгоритм, запущенный на сервере стратегии в реальном времени, советует игрокам набор новых карт, основываясь на статистике их прошлых игр.

1.2 Методы процедурной генерации контента

В 1978 году Дон Д. Ворс использовал простой алгоритм для создания данжей в своей игре Beneath Apple Manor. А в 1980 году была выпущена игра Rogue, которая положила начало жанру под названием Rogue-like, одной из главных особенностей которого и стала процедурная генерация уровней. С тех

пор генерация контента начала свое развитие в игровой индустрии и обросла определенными характеристиками, присущими тем, или иным алгоритмам.

Например, генерационные алгоритм могут просто основываться на рандомном числе, которое полностью определяет результат, а могут предоставлять более высокий уровень контроля со стороны пользователя. Алгоритмы могут быть как линейными, так и подразумевать цикличные генерации. Линейный алгоритм создает контент единожды, и этот контент вынужден всегда быть корректным или же как минимум быть на приемлемом уровне, например, генерация ландшафта методом фракталов. Циклический подход подразумевает использование сразу двух механизмов: генерации и теста. После того, как экземпляр сгенерирован, проводятся тесты согласно определенным критериям, и если экземпляр не удовлетворяет требованиям, проводится новая генерация. Также, генерация может быть полностью автоматической или же позволять (а иногда и требовать) вмешательство со стороны пользователя.

На основе этих характеристик можно выделить три основных типа алгоритмов [10]: линейные, основанные на поиске и основанные на машинном обучении. Конструктивные методы включают в себя генераторы псевдослучайных чисел, генеративные грамматики, пространственные алгоритмы, клеточные автоматы и прочее. Алгоритмы основанные на поиске охватывают использование искусственного интеллекта и методы поиска, такие как эвристический и локальный поиск. На данный момент, алгоритмы, использующие искусственный интеллект, предоставляют больше контроля пользователю и позволяют добиться более реалистичных результатов, хоть и за счет увеличения времени работы и снижения гарантии на получение ожидаемого контента.

1.2.1 Линейные методы

Линейные методы подразумевают под собой алгоритмы, которые работают за фиксированное, зачастую малое, количество времени и не оценивают конечный результат [11]. Большинство из этих методов довольно легки в реализации, однако они предоставляют довольно ограниченный контроль над результатом и самим процессом.

Самым распространенным примером линейных методов является генератор на основе псевдослучайных чисел. Данный генератор широко используется для создания подземелий и лабиринтов [12]. Кроме того, так как результаты всегда одинаковы для одного и того же набора значений, данные генераторы используются для сжатия данных.

Другим примером линейного метода являются алгоритмы, использующие фракталы или грамматики. Первые зачастую применяются в симуляции процессов близких к природным, в то время как грамматики получили популярность в области генерации растительности и городов [13].

1.2.2 Методы на основе поиска

Методы на основе поиска являются частным случаем алгоритмов, работающих по циклической модели, но со следующими оговорками:

1) Тестирующая функция не просто одобряет или отклоняет результат, но также и оценивает его, используя одно или множество значений.
2) Генерация нового экземпляра учитывает оценки данные прошлым кандидатам. Таким образом, целью является создание контента с наивысшей оценкой.

Данный тип генераторов часто применяется для создания набора правил для относительно простых игр, а также для генерации уровней платформеров и трасс гоночных игры. Кроме того, такие генераторы позволяют вносить изменения, основываясь на внешних данных, если использовать последние для

определения оценки. К примеру, можно использовать внутриигровую статистику для определения популярности и эффективности того или иного предмета. [9]

1.2.3 Методы на основе машинного обучения

Процедурная генерация контента с помощью машинного обучения обычно подразумевает создание контента на основе моделей, натренированных на уже существующем контенте. В отличие от поисковых методов, данные алгоритмы не требуют дополнительной настройки и опираются только на существующие артефакты. Все техники генерации с помощью машинного обучения могут быть разделены на 3 группы: последовательность, сетка и граф. Так, карта из платформера может быть представлена последовательностью Y координат платформ, а двухмерный данж в виде сетки комнат. [14]

1.3 Подземелье

Оригинальный термин “Dungeon (Подземелье)” обозначает собой тюрьму, либо же хранилище, расположенное под замком или же иным строением [15]. В компьютерных играх же, в качестве подземелий подразумевают лабиринтообразное окружение, где путешественник входит в одну часть подземелья, находит сокровища, избегает или уничтожает монстров, спасает людей, попадает в ловушки, и, в конце концов, выходит из другой. Вероятно, данный концепт зародился и получил распространение благодаря настольной игре Dungeons and Dragons, после чего стал уникальной чертой почти каждой компьютерной ролевой игры, включая the Legend of Zelda и the Final Fantasy. Успех подобных тайтлов, в сумме с огромными возможностями для воображения, сделали подземелья идеальной целью для процедурной генерации.

1.3.1 Компоненты подземелий

Опыт исследователей в процедурной генерации показывает, что любой объект может быть разделен на определенные шаблоны дизайна [16], так

называемые единицы проектирования. Например, атрибуты игрока могут быть выражены через максимальный показатель здоровья и маны, их скорость восстановления, урон, защиту, скорость и другие. Подземелья также не являются исключением и их структуру тоже можно разбить на основные составляющие:

1) Блок – основная единица пространства в подземелье. Отдельно взятые блоки имеют два логических атрибута с ними связанные: Проходимость и Просматриваемость.
2) Уровень – пространство, состоящее из блоков.
3) Блок стены – базовый Блок в подземелье. Данные блоки не проходимы и не просматриваемы. Базовым наполнение таких блоков зачастую выступает простой камень.
4) Блок пола – блоки, которые являются проходимыми и просматриваемыми.
5) Предмет – игровой объект, который может находится в Блоке, но также может быть подобран и перемещен в другой Блок.
6) Актер – игровой объект, который может перемещаться по Блокам и выполнять различные действия. Например, игрок и монстры.
7) Прямая видимость – логическая функция, возвращающая истинное значение, если есть линия просматриваемых блоков из точки А в точку Б.
8) Проходимость – логическая функция, возвращающая истинное значение, если есть проход из точки А в точку Б на Уровне.
9) Очередность – логическая функция, возвращающая истинное значение, если точка А должна быть пройдена до точки Б.

На основе этих элементарных компонентов можно дать определения более масштабным понятиям:

1) Пространство – набор соединенных блоков с одинаковыми атрибутами.
2) Комната – пространство из проходимых блоков, окруженное пространством из непроходимых блоков.
3) Коридор – комната, чей размер часто сильно меньше по сравнению с обычными комнатами.

1.3.2 Классификация подземелий

Подземелья в играх можно разбить на четыре группы:

1) Соединенные комнаты – тип подземелий, наиболее часто встречающийся в текстовых приключенческих играх. Игрок перемещается между локаций без явных коридоров, проходов и туннелей. Такие подземелья удобно представлять в виде графов.
2) Комнаты и коридоры – подземелья, заполненные комнатами, которые соединены разветвляющимися коридорами. Как и комнаты, коридоры тоже являются частью игрового пространства и там могут происходить различные действия. Обычно, для таких подземелий в качестве представления используют 2D матрицы, но графы тоже подходят.
3) Лабиринты. Могут быть как уникурсальные, т.е. имеющие только один вход и выход, так и мультикурсальные, имеющие сразу несколько входов и выходов. Чаще всего в таких подземельях пространство, занимаемое коридорами, значительно больше пространства комнат. Также как и для второго типа, для их представления используются двухмерные матрицы.
4) Подземелья с открытой местностью – подземелья, состоящие из больших открытых пространств с препятствиями, которые мешают свободному перемещению. В таких подземельях предпочтение

отдается тактическим маневрам. Тоже могут быть представлены матрицей.

1.4 Процедурная генерация подземелий

В большинстве приключенческих и ролевых игр, подземелья обычно состоят из нескольких комнат, соединенных коридорами. В качестве комнат и коридоров могут выступать как обычные построенные помещения, так и пещеры, главное чтобы само подземелье можно было представить в виде геометрии и топологии. Таким образом, генерация подземелий будет состоять из трех основных элементов:

1) Репрезентативная модель: абстрактное, упрощенное представление подземелья, предоставляющее общий вид финальной структуры.
2) Метод для конструирования репрезентативной модели.
3) Метод для создания итоговой геометрии подземелья из репрезентативной модели.

Так как процедурная генерация, по сути своей, является автоматизированным дизайном, то и к генерации подземелий подразумеваются все те же требования, что и к их ручному созданию. Подземелья уникальны в том плане, что связаны как с дизайном геймплея, так и с самим игровым пространством. В отличие от, например, уровней платформеров или гоночных трасс, подземелья предпочитают свободное перемещение игрока с целью исследования, но в то же время и требуют строгого контроля над игровыми ощущениями, прогрессом и темпом. Например, игрок должен иметь возможность свободно исследовать подземелье, но не сталкиваться с ситуацией, когда дальнейший прогресс невозможен. Поэтому, генерация подземелий больше стремится достичь структурированного контроля над созданием контента, а не неожиданной вариативности самого контента.

Однако, даже данная особенность не стала ограничением для разработчиков и потому на сегодняшний день есть множество различных алгоритмов генерации подземелий, среди которых самыми популярными являются:

1) Клеточные автоматы (Cellular Automata)
2) Порождающие грамматики (Generative Grammars)
3) Генетические алгоритмы (Genetic Algorithms)

Ниже данные генераторы описаны более подробно и дана оценка тому, насколько их возможности подходят разработчикам с точки зрения игрового дизайна.

1.4.1 Генерация подземелий методом Клеточных автоматов

Данная генерация основана на многомерной сетке, состоящей из клеток. Каждая клетка имеет ссылку на набор клеток, которые являются для нее соседними, и начальное состояние в начальный момент исполнения алгоритма (t = 0). Чтобы высчитать состояние клетки в следующем поколении (t + 1), к ней и ее соседям применяется набор правил. После генераций, в сетке формируется паттерн, который зависит от начальных состояний клеток и правил. Данный паттерн и является репрезентативной моделью Клеточных автоматов. Например, состояние клеток может быть { стена, пол, камень }, обозначая может ли игрок пройти по определенной клетке. Результат такого алгоритма представлен на рисунке 1.1.

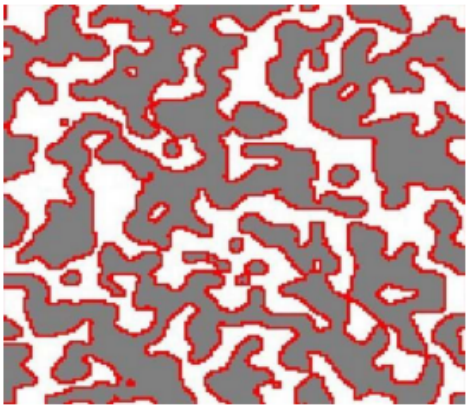

Рисунок 1.1 – Карта, сгенерированная Клеточными Автоматами

Однако, у данного подхода есть свой ряд минусов [17]. Такой генератор может образовывать большие пустые пространства, пещеры, отделенные от основной, и блоки стен, рандомно расположенные посреди пещер и коридоров.

В своей работе, Джонсон [18] использует возможности клеточных автоматов для генерации уровней пещеры и позволяет пользователю регулировать следующие параметры:

1) Процент клеток, заполненных камнем
2) Число поколений
3) Число соседей, помеченных как {камень}
4) Число соседних клеток

К сожалению, данные параметры не имеют ничего общего с игровыми показателями, а потому любая связь между параметром и игровой характеристикой может быть установлена только методом проб и ошибок.

1.4.2 Генерация подземелий методом Порождающих грамматик

Изначально порождающие грамматики использовались для описания набора лингвистических фраз. Метод создает фразы путем ограниченного выбора из списка рекурсивных трансформационных или производных правил, которые используют слова в качестве терминальных символов. Позже, на основе порождающих грамматик были разработаны графовые и форменные грамматики, которые позволяют вместо слов использовать узлы и формы.

В статье [4] Van der Linden использует порождающие грамматики для создания подземелий. Причем, была реализована собственная система геймплейных грамматик, после чего автор использовал смесь грамматик для достижения большего уровня контроля:

1. Графовые и форменные грамматики для создания общего вида уровня
2. Геймплейные грамматики для связи комнат с квестами

Репрезентативная модель и конечный результат представлены на рисунках 1.2 и 1.3.

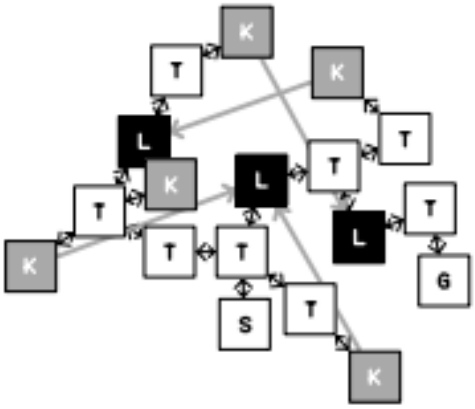

Рисунок 1.2 – Граф, сгенерированный Генеративными грамматиками

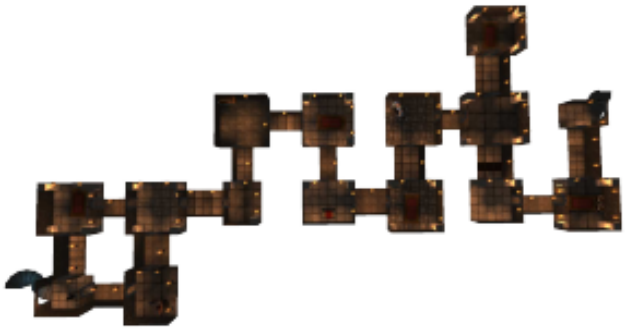

Рисунок 1.3 – Подземелье, построенное на основе графа

Касательно контроля со стороны игрового дизайна, данный подход вынуждает дизайнеров полностью уточнить генеративные грамматики еще на стадии реализации, однако он дает возможность обработать любой

реализованный игровой сценарий, а также указать дополнительные параметры для принятия алгоритмом решений.

1.4.3 Генерация подземелий с помощью Генетических алгоритмов

Эволюционные алгоритмы – эволюционные поисковые алгоритмы, которые пытаются найти оптимальное решение путем комбинирования результатов [19]. Для данных алгоритмов требуется генетическое представление и функция оценки. Генетическое представление требуется для кодирования возможных решений в виде строк, которые называют генами или хромосомами. Функция оценки же может измерить качество этих решений.

В статье [20] авторы используют эволюционные алгоритмы для создания уровней подземелий. Они используют древовидную структуру для представления уровней, и ее же для генетического представления. Узлы в дереве являются комнатами, и ребра представляют собой коридоры. Пример работы их генератора представлен на рисунке 1.4.

Также, для оценки они вынесли такие цели, как:

1) Максимизация длины пути от входа до выхода
2) Максимизация доступности комнат.

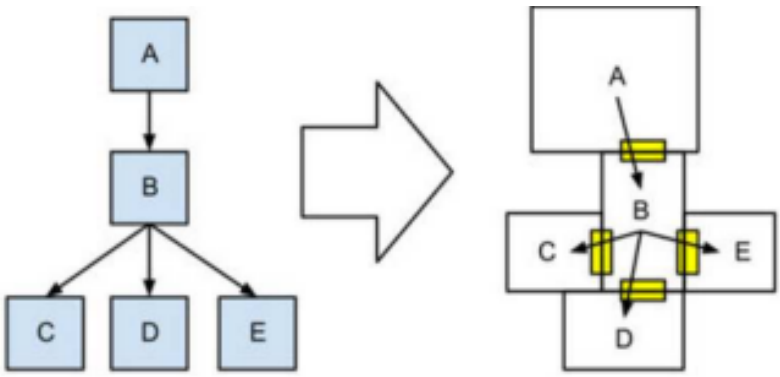

Рисунок 1.4 – Граф, сгенерированный Эволюционным алгоритмом

Однако, алгоритм описанный авторами не поддерживает создание каких-либо геймплейных связей.

2 ПОСТАНОВКА ЗАДАЧИ И РАЗРАБОТКА АЛГОРИТМА

2.1 Требования и ограничения

Целью данной главы является подробное определение функциональных и нефункциональных требований к системе генерации многоуровневых подземелий с учетом маркеров в пространстве. Функциональные требования определяют, как система должна взаимодействовать с пользователями и какие функции и возможности она должна предоставить, то есть такие требования можно описать вопросом “Что система должна делать?” [21]. В свою очередь, нефункциональные требования определяют качество и характеристики системы, такие как производительность, безопасность и уровень сложности, в упрощенной формулировке – “Как система должна это делать?” [22].

Данная глава служит важным ориентиром для разработчиков, геймдизайнеров и других будущих пользователей, а также предоставляет четкий и обоснованный набор требований, который будет использоваться при создании системы генерации подземелий. В процессе разработки, каждое требование будет тщательно рассмотрено и воплощено в функциональности или характеристиках системы, чтобы обеспечить успешную реализацию поставленных задач.

Эта глава также предоставит важную информацию об ограничениях системы, которые будут соблюдаться в процессе ее разработки и эксплуатации. Ограничения помогут определить рамки функциональности и производительности, что в свою очередь позволит точно настроить и реализовать требования проекта.

В следующих разделах этой главы будут подробно рассмотрены функциональные и нефункциональные требования, а также ограничения, которые сформулированы на основе анализа потребностей пользователей и особенностей проекта. Соблюдение этих требований будет гарантировать качество и эффективность системы генерации подземелий.

2.1.1 Функциональные требования

Для процедурного генератора подземелий в рамках данной научно-исследовательской работы были выделены следующие функциональные требования:

1) Генерация подземелья:
    a) Функция генерации подземелья: Система должна предоставлять функцию для создания нового подземелья. Это может быть вызвано программно или через интерфейс.
    b) Настройка сложности уровня: Пользователи должны иметь возможность задавать параметры сложности для каждого подземелья, такие как размер, количество комнат и пути.
    c) Начальное и конечное местоположение: Каждое подземелье должно иметь определенное начальное и конечное местоположение, между которыми игроки будут перемещаться.
2) Размещение маркеров:
    a) Размещение маркеров в пространстве: Система должна способствовать размещению маркеров (ловушек, сокровищ, специальных событий и других объектов) внутри подземелья.
    b) Учет разнообразия маркеров: Система должна обеспечивать разнообразие маркеров, чтобы создавать интересные игровые сценарии.
    c) Проверка конфликтов: Система должна предотвращать конфликты маркеров между собой или с другими элементами уровня.
3) Размещение комнат:
    a) Генерация комнат: Система должна создавать комнаты в подземелье с учетом его размеров, формы и структуры.

b) Связь с маркерами: Размещение комнат должно учитывать расположение маркеров.

c) Разнообразие комнат: Система должна обеспечивать разнообразие комнат с различными характеристиками и функциями.

4) Соединение комнат:

a) Создание проходов в 3х плоскостях: Система должна создавать не только горизонтальные проходы, но и поддерживать вертикальные и наклонные пути между комнат на разных высотах.

b) Обеспечение проходимости: Проходы должны быть доступными и обеспечивать проходимость всего подземелья.

c) Учет маркеров и комнат: При создании проходов необходимо учитывать существующие маркеры и комнаты.

Эти функциональные требования обеспечивают основную функциональность системы генерации многоуровневых подземелий с учетом маркеров в пространстве. Они определяют, как система будет генерировать подземелья, учитывая сложность, разнообразие и размещение маркеров для обогащения игрового опыта.

2.1.2 Нефункциональные требования

Для обеспечения стабильной и эффективной работы системы были сформированы следующие нефункциональные требования:

1) Производительность:

a) Время генерации: Система должна генерировать подземелья достаточно быстро, чтобы геймдизайнеры не испытывали задержек при разработке. Время генерации не должно превышать определенного порога, например, не более 1-2 минут.

b) Эффективное использование ресурсов: Система должна оптимально использовать ресурсы, такие как процессорное время и память, чтобы обеспечить высокую производительность.

2) Уникальность:
   a) Разнообразие генераций: Система должна обеспечивать достаточное разнообразие в создаваемых подземельях, чтобы предотвратить монотонность игры.
3) Гибкость:
   a) Настройка параметров генерации: Система должна предоставлять средства для настройки параметров генерации подземелий, чтобы разработчики игр могли адаптировать их под свои нужды.
   b) Модульность: Компоненты системы (генератор уровней, размещение маркеров и т. д.) должны быть модульными и легко заменяемыми для обеспечения гибкости и расширяемости.
4) Баланс игры:
   a) Сбалансированный геймплей: Генерируемые подземелья должны обеспечивать сбалансированный игровой процесс, где ни один тип маркера не доминирует, и игрокам предоставляется адекватный вызов.
5) Графика и интерфейс:
   a) Визуальное качество: Графическое представление подземелий должно позволять геймдизайнерам явно определить тот или иной участок подземелья.
   b) Интуитивность интерфейса: Если система предоставляет пользовательский интерфейс для настройки параметров

генерации, он должен быть интуитивно понятным и удобным в использовании.

c) Поддержка разных платформ: Пользовательский интерфейс должен быть адаптирован к разным платформам.

2.2 Моделирование системы

Так как разрабатываемая система является довольно сложной и состоит из нескольких механизмов, то будет логично выделить данные механизмы в отдельные модули, что позволит не только сделать их легко модифицируемыми, но и даст в будущем больше опций для оценки эффективности всей системы. Таким образом, были определены подсистемы описанные ниже.

2.2.1 Модуль размещения маркеров

Модуль размещения маркеров функционирует как независимая система и отвечает исключительно за размещение заранее определенных маркеров внутри сгенерированного уровня. В качестве входных данных модуль может принимать типы маркеров, а также их заранее определенные местоположения. При этом, типы маркеров не должны быть фиксированными константами, позволяя разработчикам редактировать их на ходу.

Одним из важных аспектов при размещении маркеров является учет ограничений, связанных с каждым маркером. Это обеспечивает соблюдение правил игры и гарантирует, что размещение маркеров не нарушит баланс игры. Вот какие могут быть ограничения:

1) Минимальные и максимальные расстояния: Некоторые маркеры могут иметь ограничения на минимальное и максимальное расстояние между собой. Модуль должен учитывать эти ограничения, чтобы избежать слишком сгущенного или разреженного размещения.

2) Взаимодействие маркеров: Некоторые маркеры могут требовать определенных условий или взаимодействий, связанных с игровым процессом. Модуль должен обрабатывать эти взаимосвязи.
3) Обеспечение достижимости: Важно, чтобы все маркеры были достижимы игроком. Модуль должен гарантировать, что нет изолированных маркеров, к которым невозможно добраться.

2.2.2 Модуль генерации комнат

Этот модуль предназначен для генерации комнат с учетом маркеров, расставленных в пространстве. Он осуществляет размещение комнат таким образом, чтобы удовлетворить заданным маркерам и соблюсти ограничения. Это может быть обеспечено следующим функционалом:

1) Анализ маркеров: Модуль получает информацию о маркерах, определенных геймдизайнером. Каждый маркер может иметь свой тип, характеристики и правила размещения.
2) Планирование комнат: Модуль начинает с планирования расположения комнат, принимая во внимание заранее размещенные маркеры. Комнаты планируются так, чтобы соответствовать типам и ограничениям маркеров.
3) Соблюдение ограничений: Модуль обеспечивает, чтобы сгенерированные комнаты и их расположение соответствовали ограничениям, установленным разработчиками. Среди ограничений могут быть размеры и количество комнат.
4) Адаптация к пространству: Модуль учитывает размеры и форму предоставленного для заполнения пространства, а также его характеристики при планировании и размещении комнат.

2.2.3 Модуль генерации коридоров

Задача данной подсистемы – обеспечить логичное и интересное соединение комнат с учетом контекста маркеров и ограничений. Возможности, необходимые данному модулю, следующие:

1) Анализ маркеров и комнат: Модулю передается информация о маркерах и сгенерированных комнатах. Эта информация служит основой для размещения коридоров.
2) Планирование соединений: Модуль решает, какие комнаты должны быть соединены коридорами, и какие маркеры и ограничения следует учитывать при этом.
3) Алгоритмы поиска пути: Модуль использует алгоритмы поиска пути, такие как A*, для нахождения оптимальных маршрутов между комнатами, учитывая маркеры.
4) Создание коридоров: Коридоры генерируются вдоль найденных маршрутов. Они могут иметь разные формы и структуры в зависимости от контекста маркеров и ограничений.
5) Учет ограничений коридоров: Модуль обеспечивает, чтобы созданные коридоры соответствовали ограничениям, таким как минимальная и максимальная длина коридоров и их структура.

2.3 Выбор инструментов и алгоритмов

2.3.1 Генерация комнат

Так как генерация подземелий, в особенности их двумерных вариантов, концепция не новая, существуют различные способы генерации комнат. Самым простым является рандомная расстановка комнат с проверкой отсутствия их пересечения [23], но, так как результат требует оценки, полная рандомизация будет не лучшим вариантом. Потому следует рассмотреть алгоритмы с большим уровнем контроля. Один из таких алгоритмов уже был рассмотрен ранее – метод

Клеточных автоматов. С помощью клеточных автоматов получаются интересные результаты [7], однако, в таком методе очень сложно контролировать появления и размещение конкретных деталей.

Другой популярный способ размещения комнат называется “слепой копатель”, являющийся частным случаем алгоритмов с агентом. Заключается он в использовании некоего абстрактного агента, который рандомно перемещается по выделенному пространству “шагами” и с определенным шансом после шага может разместить комнату [24]. Данный алгоритм можно применить для разрабатываемого генератора искусственно подталкивая агента в направлении маркеров. Тем не менее, данный способ все еще остается довольно рандомизированным и не гарантирует качественного покрытия пространства, к тому же могут быть образованы пересекающиеся комнаты [11] с различными маркерами, как изображено на рисунке 2.1.

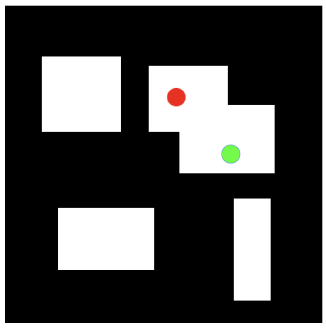

Рисунок 2.1 – Пересекающиеся комнаты в результате работы агента

Еще одним алгоритмом разделения подземелья и выявления комнат является метод “Двоичного разбиения пространства”, который подразумевает разделение двух- или трехмерного пространства на разъединенные подмножества такие, что любая точка этого пространства, будет лежать только в одном из подмножеств [6]. Таким образом получается довольно структурированная система из подпространств, в которых можно впоследствии разместить комнаты как представлено на рисунке 2.2. Единственное, что надо

будет учитывать, это возможность одномоментного наличия различных маркеров в одном подпространстве.

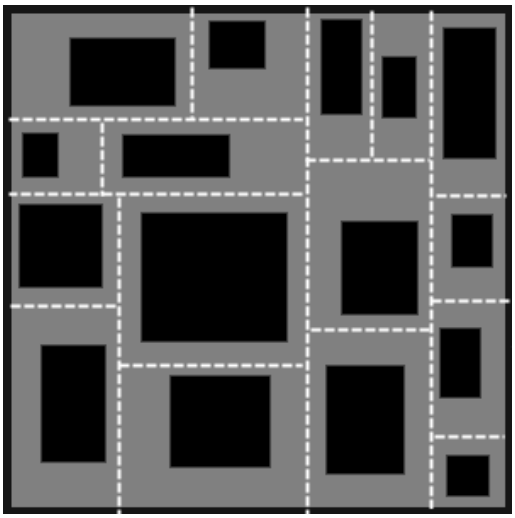

Рисунок 2.2 – Размещение комнат в разбитом пространстве [25]

Последним методом, что стоит рассмотреть, является не совсем алгоритм генерации подземелий, а способ разбиения помещения на функциональные комнаты [26]. Данный метод предлагает сначала размещать определенные точки, обозначающие относительное расположение комнат, а потом постепенно наращивать их размер. С этим алгоритмом появляется возможность создавать комнаты не только прямоугольной формы. Действие алгоритма видно на рисунке 2.3.

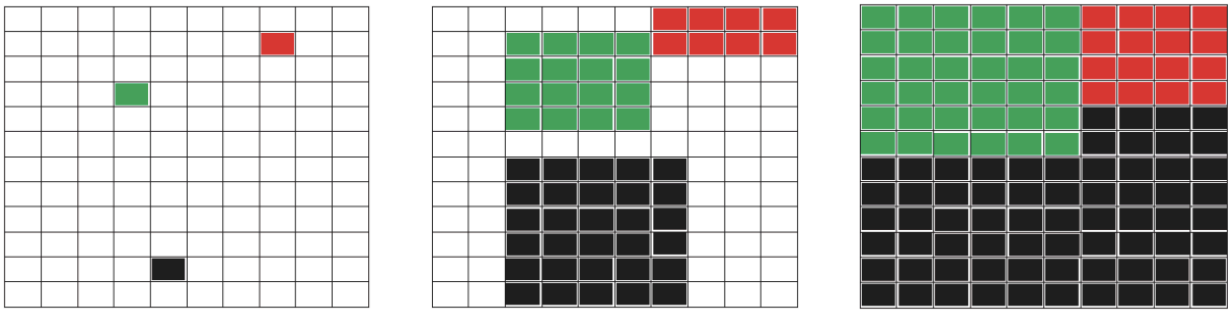

Рисунок 2.3 – Постепенное заполнение пространства комнатами

После рассмотрения всех алгоритмов выше, было решено использовать модифицированный последний алгоритм. Сначала, на основе маркеров, будут выбираться центры обязательных комнат. После чего, если необходимо, будут

выбраны центры дополнительных комнат, причем из нескольких генераций будет выбрана та, где комнаты более равномерно распределены и предоставляют больше игровой зоны.

2.3.2 Генерация коридоров

Также как и комнаты, коридоры можно генерировать несколькими способами. Большинство из них почти полностью повторяют перечисленные алгоритмы генерации комнат и не могут быть использованы в данной работе из-за несовместимости рандомизации с наличием маркеров и необходимости создания гарантированных путей между ними. Поэтому, надо было найти другой способ. Для упрощения задачи я разделил генерацию коридоров на 2 раздельных этапа: определение связей между комнатами и построение реальных маршрутов на основе этих связей.

Чтобы выстроить связи типа комната-коридор-комната было решено прибегнуть к теории графов, где центры комнаты это узлы, а коридоры – ребра. Так как расположение комнат нам уже известно, то надо на их основе построить ребра. Для этого можно воспользоваться алгоритмами на графах, а именно Триангуляцией Делоне. Данный алгоритм позволяет объединить все узлы ребрами в геометрические треугольники (рисунок 2.4) и обладает следующими полезными для генератора свойствами [27]:

1) Триангуляция Делоне максимизирует минимальный угол среди всех углов всех построенных треугольников, тем самым избегаются «тонкие» треугольники.
2) Триангуляция Делоне максимизирует сумму радиусов вписанных окружностей.
3) Триангуляция Делоне минимизирует максимальный радиус минимального объемлющего шара.

4) Триангуляция Делоне на плоскости обладает минимальной суммой радиусов окружностей, описанных около треугольников, среди всех возможных триангуляций.

То есть, данная триангуляция позволит еще на моменте создания ребер минимизировать количество прилегающих друг к другу и пересекающихся коридоров. Кроме того, определенные вариации данного алгоритма также поддерживают трехмерное пространство, используя сферу вместо окружности.

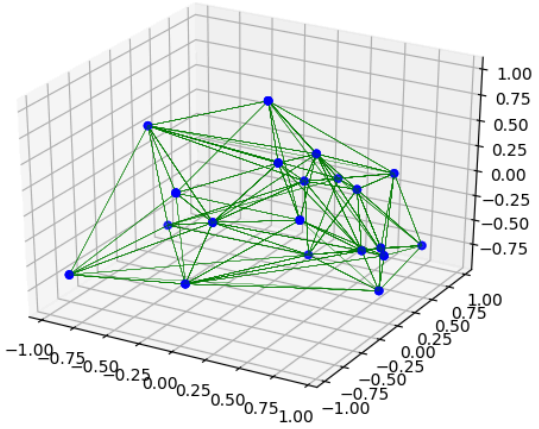

Рисунок 2.4 – Триангуляция Делоне в трехмерном пространстве

Далее, после получения основного разбиения, можно из созданных ребер уже выбирать необходимые на основе правил, установленных маркерами и строить по ним коридоры. Для этого можно воспользоваться алгоритмами поиска пути, популярными в разработке видеоигр. Например, можно использовать A* алгоритм, который позволяет строить путь не только на основании расстояния и препятствий, но также разрешает указывать “цену” прохождения отдельных участков пространства, что способствуют большей настраиваемости генератора.

2.3.3 Выбор инструментов

Так как целевой аудиторией итоговой системы являются разработчики игр, то и программное обеспечение, через которое должен осуществляться доступ к генератору, должно быть им знакомым. Таковым может являться игровой

движок. Однако, в наше время ассортимент игровых движков довольно широкий, из-за чего очень сложно написать генератор, который бы работал везде, потому следует выбрать тот, что подходит для системы лучше всего. Для этого есть два основных фактора: возможность реализации всего описанного выше функционала и удобство работы разработчиков с будущим генератором.

В то время как модель генерации комнат можно назвать простой в реализации, Триангуляция Делоне не является столь же тривиальной, а необходимость в использовании ее трехмерной версии подталкивает к использованию уже готовых решений, чье количество в открытом доступе ограничено. К счастью, библиотека SciPy, доступная на языке программирования Python предоставляет такой функционал. Соответственно, движок разработки должен поддерживать возможность исполнения скриптов написанных на Python. Таковых известных автору два: Unreal Engine и Unity. Оба движка поддерживают Python посредством внешних плагинов.

Если же сравнивать Unreal Engine и Unity со стороны удобства использования и настройки генератора, то автор предпочтет Unity. Тому есть две причины:

1) У Unity удобная реализация создания пользовательских инструментов редактирования на основе EditorWindow.
2) Unity позволяет довольно наглядно отрисовывать неигровые объекты на сцене через графический интерфейс называемый Gizmo.

3 ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ

3.1 Программная архитектура

На предыдущем этапе исследования были выделены основные компоненты системы, которые, в процессе разработки архитектуры, были распределены по функциональным модулям. Для большей ясности, сами модули и связи между ними изображены диаграммой пакетов на рисунке 3.1 и более подробно описаны ниже.

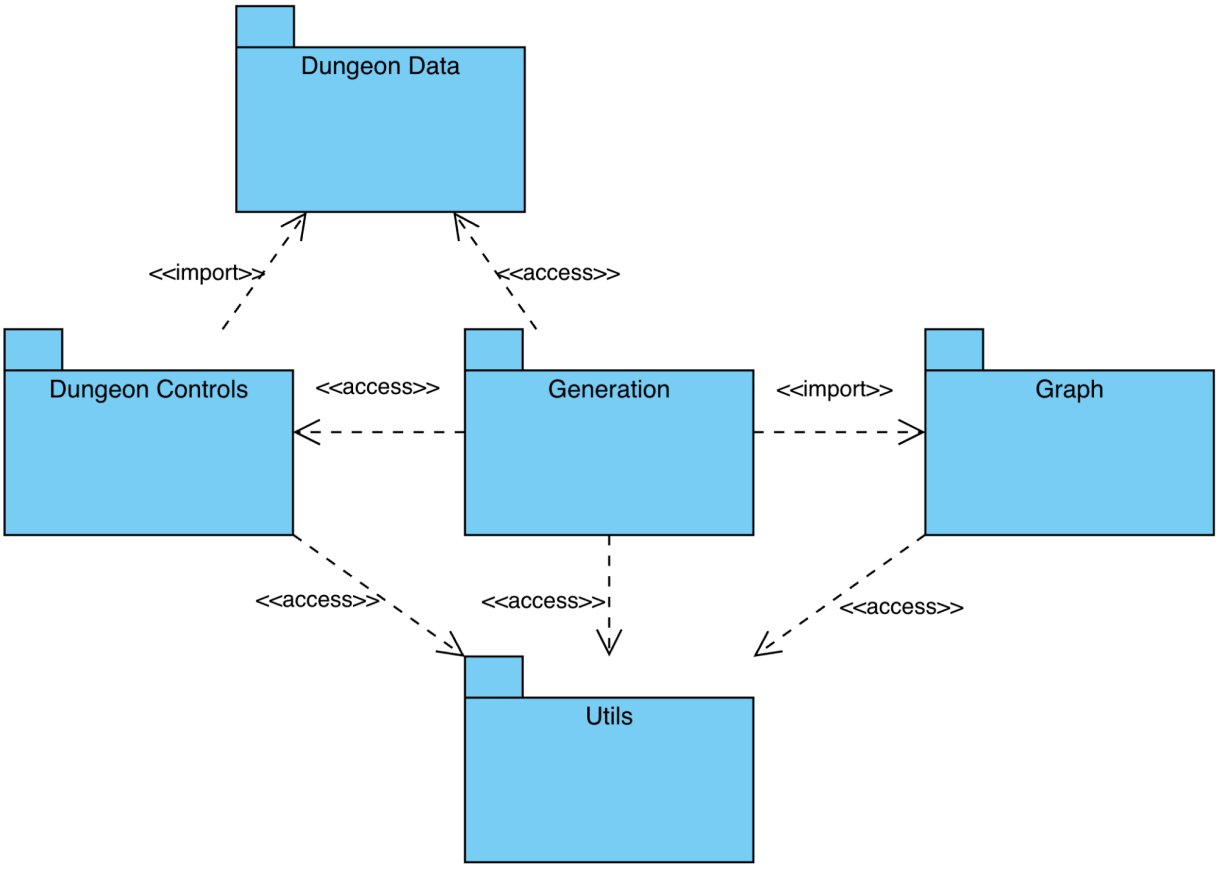

Рисунок 3.1 – Диаграмма пакетов реализуемой системы

Пакет DungeonData отвечает за хранение данных генерируемого подземелья в виде трехмерной сетки, где каждая ячейка охарактеризована координатами в пространстве и типом куска подземелья, к примеру ячейка комнаты или пустая ячейка. Также, в данном пакете реализован основной класс комнаты, а также ее базовые подтипы.

Управление генерацией со стороны игровых дизайнеров осуществляется благодаря пакету DungeonControls, который позволяет не только размещать сами комнаты в виде маркеров в пространстве, но также предоставляет функционал установки валидаторов, запускающихся как до генерации, так и после нее.

Так как в предыдущей главе для генерации связей между комнатами было решено использовать теорию графов, то был введен отдельный пакет для работы с ними – Graph. В нем находятся реализации вершин, ребер, а также некоторых алгоритмов, которые будут описаны позднее.

Основным же компонентом системы является пакет Generation, в котором описана генерация подземелья на основе уже перечисленных модулей и их контекст синхронизации – GenerationContext, где хранятся промежуточные и финальные результаты. А пакет Utils реализует функционал, упрощающий связь между модулями.

3.2 Размещение маркеров

Для реализации маркеров в пространстве был выбран метод, основанный на заранее созданных игровых объектах, так называемых префабов, с навешенным на них скриптом. Этот подход позволяет геймдизайнеру управлять созданием комнат в месте нахождения объекта маркера. Для этой цели был разработан класс RoomNode, который обеспечивает необходимую функциональность и может быть использован как самодостаточный компонент игрового объекта. Визуальное представление связи класса RoomNode с классом комнаты параметров отображено на рисунке 3.2.

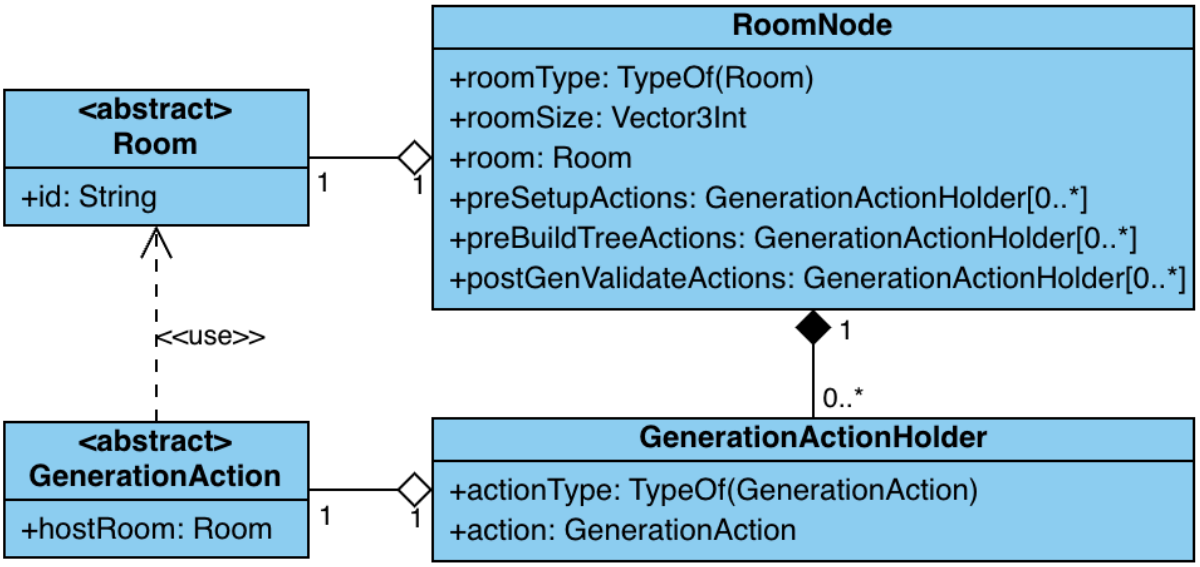

Рисунок 3.2 – Диаграмма классов для зависимостей RoomNode

Таким образом, объект маркера содержит два основных настраиваемых параметра:

1) Позиция: позволяет задавать точное местоположение маркера в пространстве сцены.
2) Тип комнаты: задается через RoomNode и указывает на класс комнаты, которая будет создана в этом месте.
3) Размер комнаты: желаемый размер комнаты, созданной маркером

Значение RoomType в компоненте RoomNode не просто элемент перечисления, а указывает на конкретный класс комнаты, который будет создан в этом месте. Это позволяет программистам добавлять новые виды комнат с уникальным функционалом без изменения основного кода. Данную информацию можно отображать визуально на самих макрерах, что представлено на рисунке 3.3.

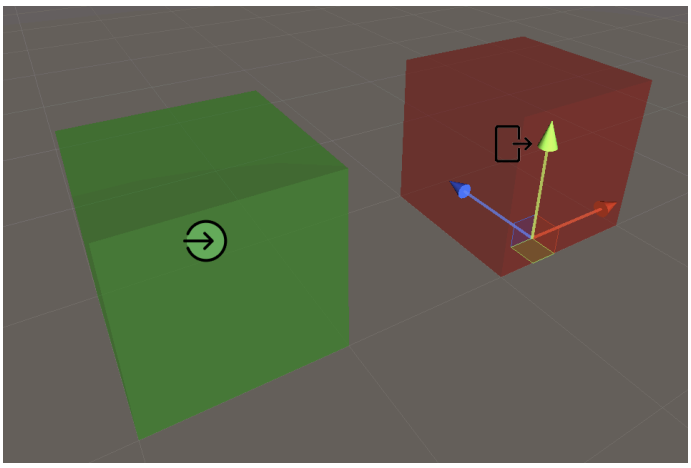

Рисунок 3.3 – Маркеры комнат входа и выхода

Для контроля процесса обработки маркеров реализована система "действий", выполняемых на разных этапах генерации. Это позволяет добавлять различные действия, необходимые для валидации корректности подземелий. Например, можно использовать предшествующее генерации условие наличия комнаты определенного типа с конкретным идентификатором.

Этот подход обеспечивает гибкость и расширяемость системы, позволяя геймдизайнерам и программистам эффективно сотрудничать при создании игровых уровней. Преимущества такого подхода включают удобство визуального редактирования, гибкость в добавлении новых типов комнат и простоту в контроле процесса генерации маркеров.

В качестве базовых разработано пять типов маркеров, представленных на рисунке 3.4.

Рисунок 3.4 – Слева направо: пустая комната, комната входа, комната выхода, комната босса, комната с ключом, закрытая комната

3.3 Генерация комнат

За создание комнат в модуле генератора отвечает класс RoomSpawner, чью работу можно разделить на два этапа: расстановка ядер комнат и разрастание комнат по трем осям.

Сначала рассмотрим этап расстановки. Перед описанием самого алгоритма стоит выделить настраиваемые параметры, на основе которых он работает. Эти параметры отображены в таблице 3.1.

Таблица 3.1 – Используемые параметры для размещения ядер комнат

Название параметра	Тип данных	Описание
ExtraRoomCount	int	Количество комнат, сгенерированных дополнительно к обозначенным геймдизайнерами
MinSpawnRadius	int	Минимальный радиус желаемой пустоты вокруг генерируемых ядер
RadiusOffsetMultiplier	float	Множитель минимального расстояния от ядер комнат до краев рабочего пространства
RadiusIntersectMultiplier	float	Множитель минимального расстояния между ядер комнат
MaxSpawnTries	int	Максимальное количество попыток размещения очередной дополнительной комнаты

Теперь, когда параметры определены, стоит описать основу самого алгоритма размещения – сетку “занятости”. Перед исполнением алгоритма формируется трехмерная сетка соразмерная той, на которой генерируется подземелье. Когда ядро комнаты добавляется в сетку, то занятой отмечается не

только ячейка с самим ядром, но и все ячейки, которые находятся в определенном радиусе вокруг него. Данный радиус рассчитывается путем перемножения MinSpawnRadius и RadiusIntersectMultiplier.

Таким образом, в начале самого алгоритма происходит размещение ядер всех преднастроенных геймдизайнерами комнат и “закрашивание” занятых ячеек. После чего, начинаются попытки генерации дополнительных комнат, в количестве определяемым переменной ExtraRoomCount. При размещении ядер дополнительных комнат, помимо такой же отметки заполненности сетки, выполняется еще две проверки:

1. Минимальное отдаление от других ядер: высчитывается благодаря занятости той или иной ячейки сетки.
2. Минимальное отдаление от границы: определяется перемножением параметров MinSpawnRadius и RadiusOffsetMultiplier.

Упрощенное представление в двумерном пространстве можно увидеть на рисунке 3.5. В данном примере минимальный радиус равен 6 единицам, отступ от границы - 3 единицы, а радиус пересечения - 9 единиц. Допустимая область размещения представлена темной сеткой.

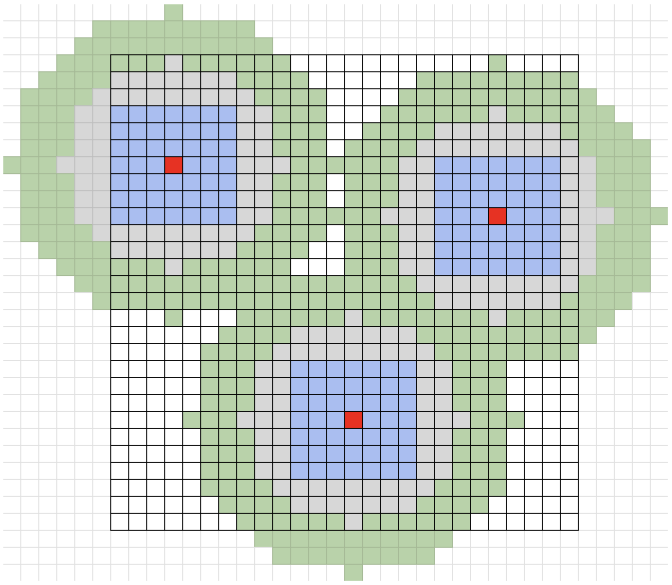

Рисунок 3.5 – размещение ядер комнат в двумерном пространстве

После того, как все ядра оказываются на своих местах, алгоритм переходит на второй этап – увеличение размера комнат. Во второй главе работы было принято решение использовать для этого довольно простой механизм, который расширяет объем комнат во всех трех направлениях. При этом оставляется некоторое пространство между ними для коридоров. Таким образом, даже после увеличения размеров комнат, остается достаточно места для прохода между ними. Сам процесс разрастания контролируется параметрами, указанными в таблице 3.2.

Таблица 3.2 – Используемые параметры для увеличения размера комнат

Название параметра	Тип данных	Описание
RoomMinSize	Vector3Int	Минимальный размер созданной комнаты
RoomMaxSize	Vector3Int	Максимальный размер созданной комнаты
InteriorSpace	Vector3Int	Размер желаемого межкомнатного пространства
GrowthSteps	int	Количество попыток разрастания комнат

Упрощенная схема алгоритма, а также результат его работы, указаны на рисунках 3.6 и 3.7 соответственно.

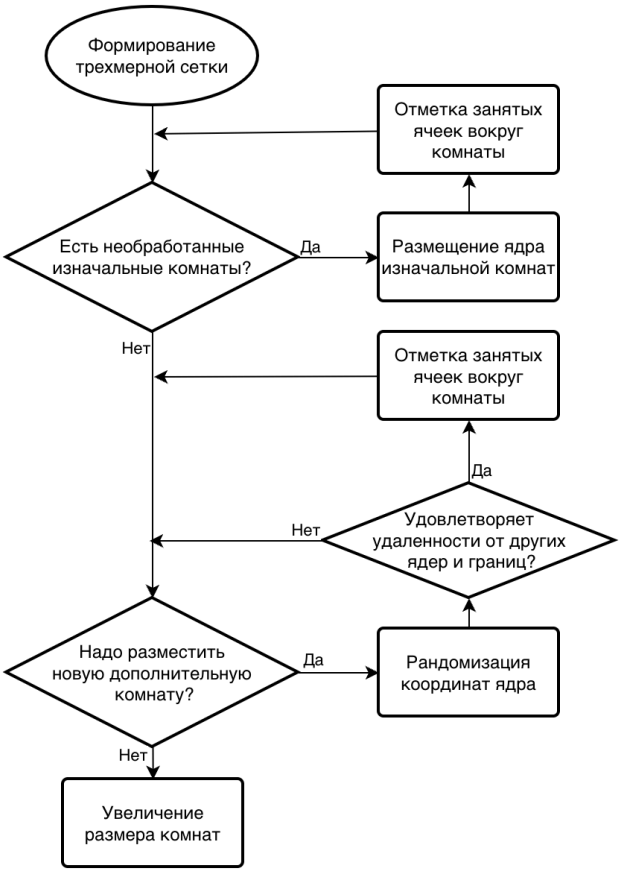

Рисунок 3.6 – Упрощенный алгоритм генерации комнат

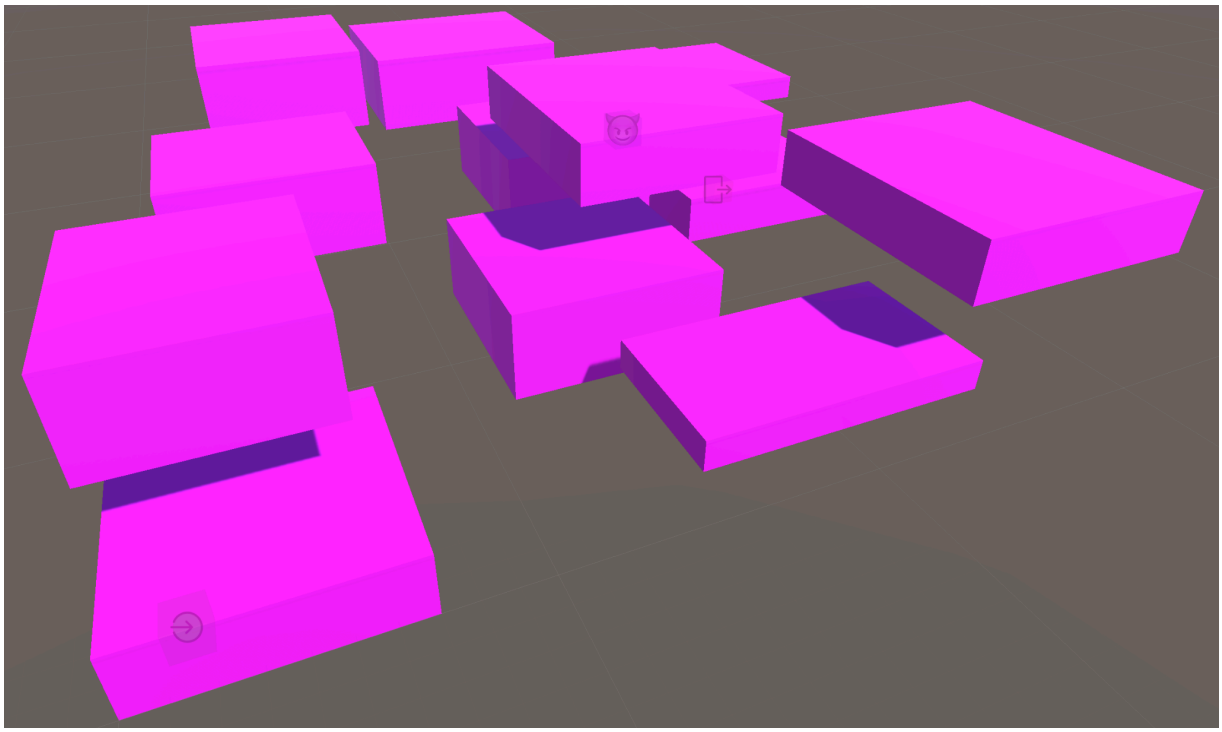

Рисунок 3.7 – Результат работы алгоритма генерации комнат

3.4 Генерация коридоров

Функционал генерации коридоров является более сложным как концептуально, так и фактически. Как и с генерацией комнат, для реализации процедурно генерируемый коридоров пришлось разбить весь алгоритм на два основных шага, каждый из которых стоит рассмотреть отдельно.

3.4.1 Построение начального графа

Самым простым решением было бы использовать полный граф, но в первую очередь следует понять, что результирующий граф должен быть построен с учетом того, что каждое ребро в нем будет означать наличие коридора между комнат. И если таких ребер будет слишком много, то подземелье не просто станет чрезмерно запутанным, но и в целом неиграбельным. Кроме того, огромное количество ребер сильно увеличит время, требуемое алгоритму для выполнения. Поэтому для создания относительно разреженного графа лучше воспользоваться алгоритмом создания подразбиения Делоне.

Одним из случаев подразбиения Делоне, также известным как триангуляция Делоне, является метод разбиения множества точек на множество треугольников, которые обладают рядом важных свойств. К примеру, триангуляция стремится к максимизации наименьшего угла и минимизации разброса длин ребер. Также, это разбиение подразумевает такие триангуляции, где ни одна точка не находится внутри описанной окружности любого треугольника.

Для трехмерного пространства разбиение работает аналогично, но учитывает еще и третье измерение, что модифицирует некоторые ограничения. К примеру, это означает, что никакая точка не должна находиться внутри сферы, определенной четырьмя вершинами тетраэдра. При этом важно учитывать правило, что ни одна грань не должна пересекаться с другой. Это обычно

осуществляется путем удаления и добавления ребер и вершин и такой подход обеспечивает более равномерное распределение итоговых фигур.

В настоящее время известно значительное количество различных алгоритмов построения триангуляции Делоне. Одним из них является инкрементный алгоритм Боуэра-Ватсона. В 1981 году, Адриан Боуэр [28] и Дэвид Ватсон [29] разработали независимо друг от друга и в одно и то же время опубликовали в журнале The Computer Journal. Данный алгоритм работает для любого количества измерений и имеет алгоритмическую сложность $O(n^{(2d-1)/d})$, где d – размерность пространства [30].

Алгоритм стартует с создания одного дополнительного тетраэдра, который содержит в себе все начальные точки из набора S, после чего точки поочередно добавляются в триангуляцию. На каждом шагу, все тетраэдры, чьи описывающие сферы содержат в себе новую точку P, обозначаются конфликтующими и убираются из разбиения. Данное действие создает полость внутри триангуляции – многоугольник, который затем повторно подвергается триангуляции с учетом того, что каждый новый треугольник имеет точку P в качестве своей вершины [31]. На рисунке 3.8 представлен процесс разбиения двух тетраэдров точкой в шесть новых путем удаления смежной грани и добавления новых ребер.

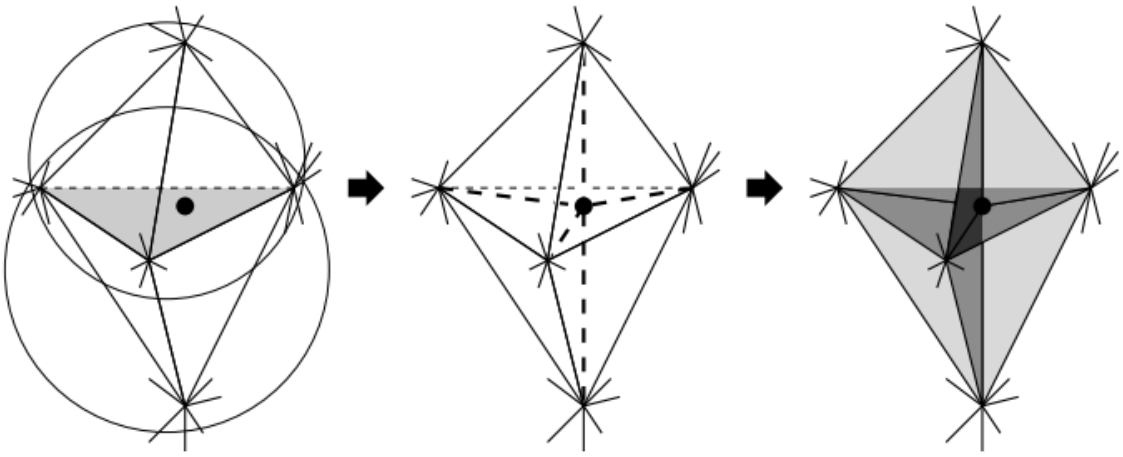

Рисунок 3.8 – Пример работы алгоритма Боуэра-Ватсона в трех измерениях

Согласно статье [32], уравнение такой сферы может быть представлено в виде матрицы на рисунке 3.9.

$$\begin{vmatrix} x^2+y^2+z^2 & x & y & z & 1 \\ x_1^2+y_1^2+z_1^2 & x_1 & y_1 & z_1 & 1 \\ x_2^2+y_2^2+z_2^2 & x_2 & y_2 & z_2 & 1 \\ x_3^2+y_3^2+z_3^2 & x_3 & y_3 & z_3 & 1 \\ x_4^2+y_4^2+z_4^2 & x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0.$$

Рисунок 3.9 – Уравнение для вершин (x_i, y_i, z_i), где $i = 1, \ldots, 4$

В результате решения данного уравнения можно получить формулы для центра и радиуса описывающей сферы, изображенные на рисунке 3.10.

$$x_0 = \frac{D_x}{2a} \qquad y_0 = \frac{D_y}{2a} \qquad z_0 = \frac{D_z}{2a} \qquad r = \frac{\sqrt{D_x^2 + D_y^2 + D_z^2 - 4ac}}{2|a|}.$$

Рисунок 3.10 – Формулы для центра и радиуса сферы

Таким образом, представив центры комнат в виде вершин и используя алгоритм Боуэра-Ватсона, можно составить тетраэдрализованный граф, в котором длина коридоров будет минимизирована, тем самым уменьшая относительное время путешествия игрока из комнаты в комнату. Пример полученного разбиения можно увидеть на рисунке 3.11, где черными линиями отмечены ребра итогового разбиения.

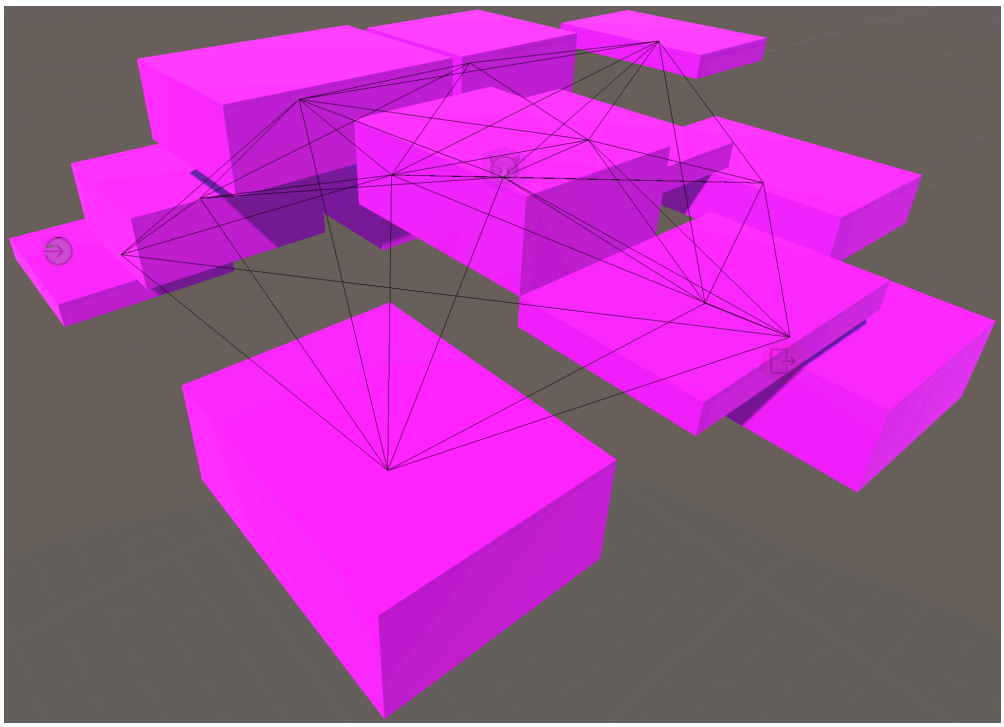

Рисунок 3.11 – Пример триангуляции на основе положения комнат

Также, было разработано действие EnsureConnection для маркеров, которое позволяет модифицировать полученный граф. Он позволяет не только убедиться в наличии связи одного маркера с определенным другим, но и удалить все остальные ребра у начального маркера. Это может быть полезно для создания игровых связей по типу “комната босса - выход”. В дальнейшей обработке такие ребра считаются не модифицируемыми.

3.4.1 Получение подграфа

Как видно из примера, данное разбиение все еще выглядит нагруженным, если на основе каждого ребра создавать новый коридор. Поэтому, было решено из полученного графа выделить подграф, который бы все еще покрывал все комнаты, но имел бы гораздо меньшее количество ребер. Первым что приходит в таком случае на ум – использовать алгоритмы нахождения остовного дерева. Одним из распространенных и довольно простых алгоритмов является алгоритм Прима, который находит такое остовное дерево, сумма длин ребер которого минимально. Однако, после реализации такого решения стало понятно, что это

лишает нас опции создания различных графов из которых можно было бы выбрать наиболее подходящий вариант, из-за чего взор сместился на более рандомизированный алгоритм, который при этом включает в себя возможность промежуточной оценки результатов – генетический алгоритм.

Основой любого генетического алгоритма являются четыре операции: кодирование хромосом, отбор (селекция), скрещивание и мутация [33]. Для большинства задач форма кодирования генов играет большую роль и может различаться своим представлением. Так, одним из популярных является бинарная кодировка, где каждый ген в хромосоме может быть записан в виде 0 или 1. Такой подход позволяет быстрее производить операции скрещивания и мутации. Именно он и был выбран для нашего решения, в результате чего каждому ребру соответствует единичный ген, обозначающий его наличие или отсутствие в конечной выборке.

Для размножения результатов было решено использовать операцию скрещивания по единичной точке, которой является центральный ген, а именно, половина первой хромосомы-родителя дополняется второй половиной другой хромосомы-родителя. В качестве мутаций каждый ген может поменять свое значение на противоположное. Визуальное представление операций можно увидеть на рисунке 3.12

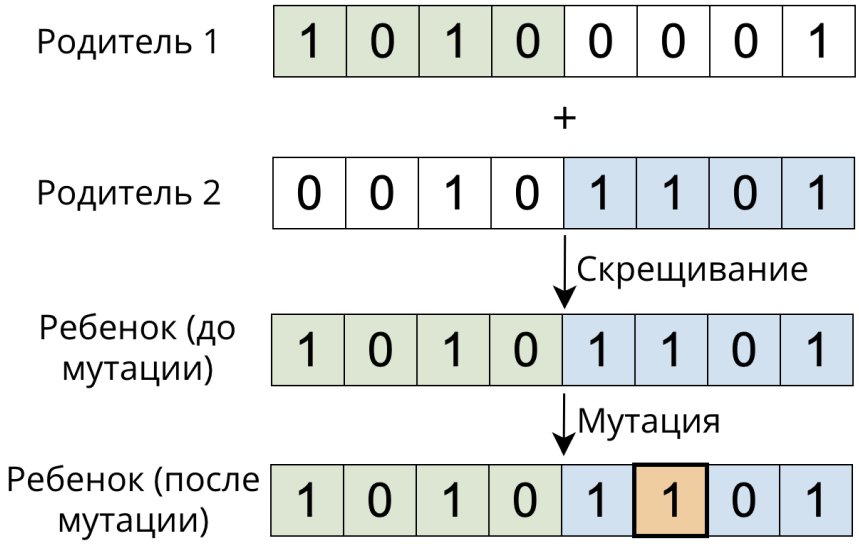

Рисунок 3.12 – Генетические операции над хромосомами

Так как в данном случае сложно оценить результат каким-то одним единственным параметром, было принято решение разработать функционал, позволяющий комбинировать различные функции оценки. При этом, некоторые из них могут полностью отбраковывать предоставленные решения. Например:

1) Количество компонент связности графа – так как нам требуется поддерживать целостность всего подземелья, а именно обеспечить доступность всех комнат, все графы с более чем одной компонентой считаются неподходящими.
2) Доступность закрытых комнат – чтобы избежать невозможности дальнейшего продвижения по подземелью, должна быть возможность открыть все закрытые комнаты путем предварительного посещения комнаты с ключом.

Остальные функции предоставляют более гибкую настройку, так как имеют по два дополнительных параметра - вес функции относительно других и целевое значение оцениваемого параметра. Среди них находятся функции оценки:

1) Количество коридоров – желаемый процент сохраненных ребер, помимо минимального количества, которое всегда на единицу меньше количества комнат
2) Разветвленность – характеристика подземелья рассчитываемая по формуле

$$L = k/(n - m),$$

где n – количество комнат в подземелье; m – количество комнат с всего 1 коридором, но не больше 2х, так называемые вход и выход подземелья; k – количество комнат, имеющих больше 2х присоединенных коридоров. То есть, чем больше итоговое значение, тем более древовидным является топология подземелья.

3) Наклон коридоров – желаемый относительный уровень наклона коридоров в подземелье, определяемый среднеквадратичной процентной ошибкой

4) Длина коридоров – желаемая относительная длина коридоров, также основанная на среднеквадратичной процентной ошибке

Итоговая функция приспособленности может быть представлена формулой

$$F(x) = F_{\text{компоненты}}(x) + F_{\text{ключ/замок}}(x) + w_{\text{ребра}} F_{\text{ребра}}(x) + w_{\text{ветви}} F_{\text{ветви}}(x) + w_{\text{наклон}} F_{\text{наколон}}(x) + w_{\text{длина}} F_{\text{длина}}(x),$$

Такая комбинация оценщиков позволила получать деревья, которые не только сохраняют геймплейные характеристики, но и стремятся привести топологию подземелья к желаемой геймдизайнерами. Примеры таких генераций можно наблюдать на рисунках 3.13 и 3.14.

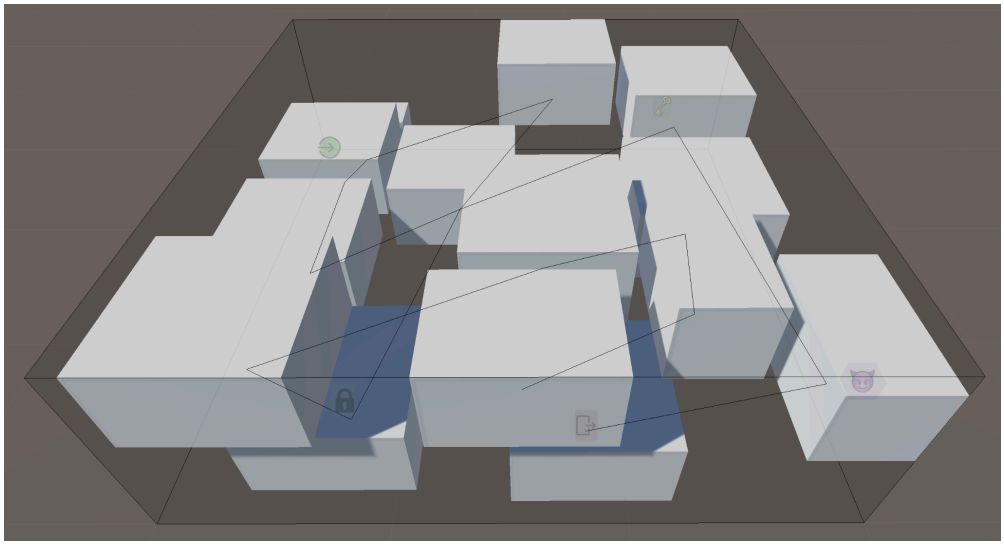

Рисунок 3.13 – Дерево с минимальным количеством ребер и минимизацией ветвления

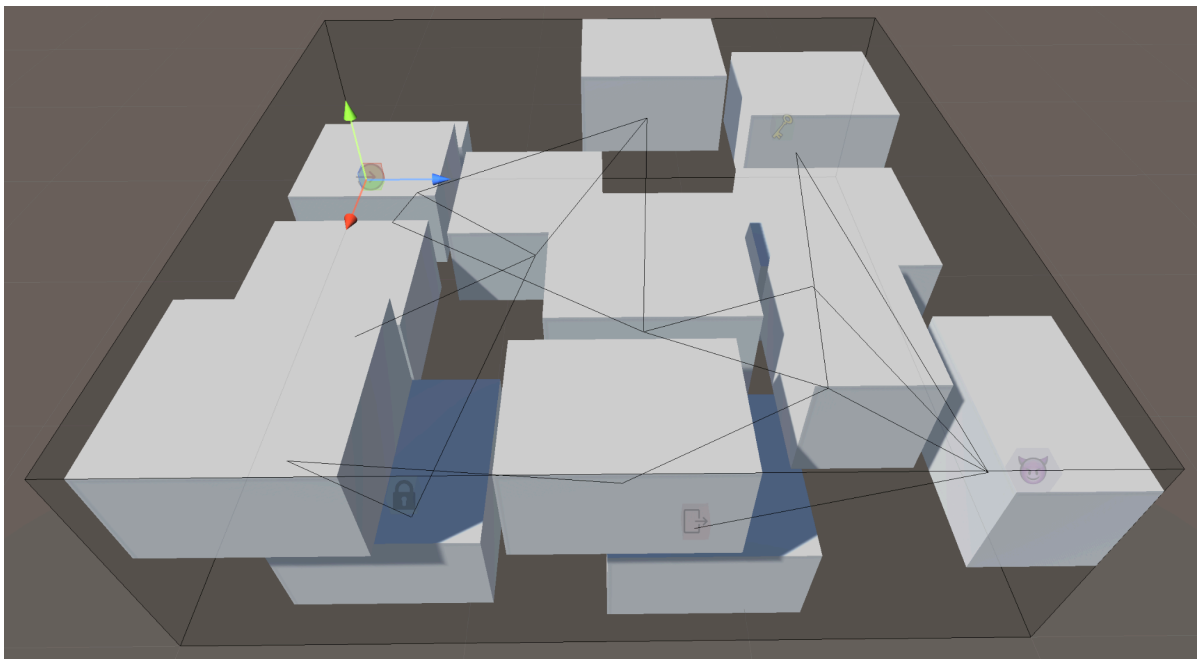

Рисунок 3.14 – Дерево со средней разветвленностью и 20 процентами дополнительных ребер

3.4.3 Построение путей

Теперь, когда у нас есть связи между комнат, мы можем перейти к непосредственному построению путей на сетке генератора. Для этого используется уже упоминавшийся во второй главе алгоритм A*. Данный алгоритм позволяет находить кратчайший путь от начальной точки к конечной, учитывая препятствия и сложность пути в трех измерениях. Чтобы оценить эту саму сложность, определенным “шагам” алгоритма присваивается оценка, которая влияет на перемещение по той или иной ячейке и высчитывается она для ячейки v по формуле $f(v) = g(v) + h(v)$, где $g(v)$ – наименьшая стоимость пути в v из начальной вершины, а $h(v)$ – эвристическое приближение стоимости пути от v до конечной цели [34].

Относительно нашей задачи помимо расстояний от и до комнат, для оценки доступны такие параметры как направление и тип ячейки. Таким образом были введены параметры указанные в таблице 3.3.

Таблица 3.3 – Используемые параметры для A* эвристики

Название параметра	Тип данных	Описание
EmptyCost	int	Стоимость прохождения по пустой ячейке
HallwayCost	int	Стоимость прохождения по уже проложенному коридору
RoomCost	int	Стоимость прохождения по ячейке комнаты
InnerRoomCost	int	Стоимость прохождения по ячейке комнаты, являющейся стартовой или конечной
YCost	int	Стоимость изменения координаты Y, то есть стоимость перемещения по вертикали

В результате, имея перемещение по трем осям и используя приведенные выше параметры получаются коридоры как на рисунке 3.15.

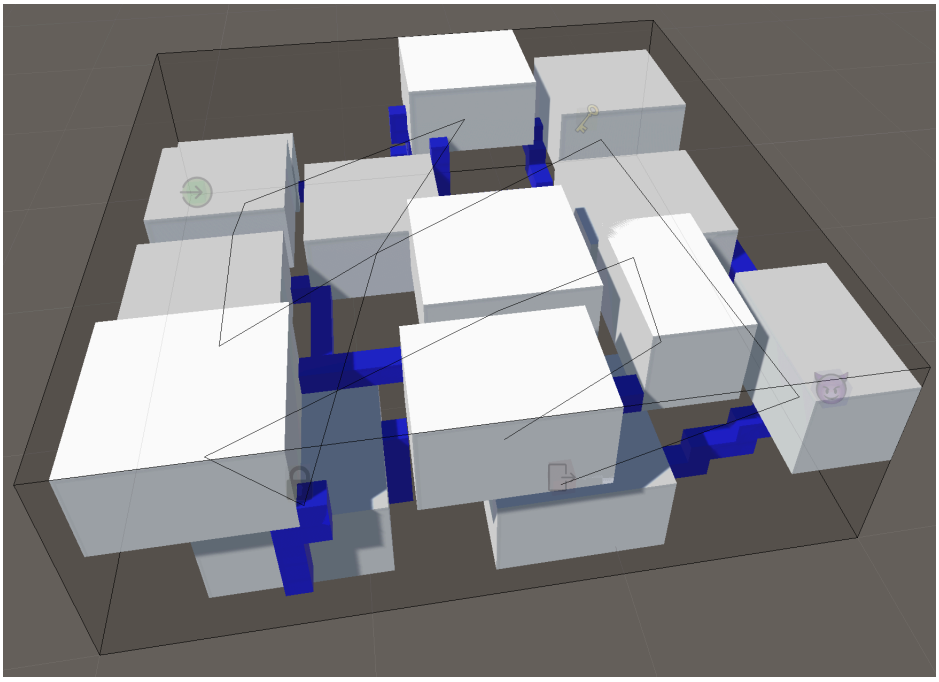

Рисунок 3.15 – Коридоры, построенные в результате работы алгоритма A*

4 СБОР И АНАЛИЗ МЕТРИК

4.1 Методология

Основной целью исследования является реализация процедурного генератора многоуровневых подземелий, которые могли бы быть в дальнейшем использованы геймдизайнерами как во время разработки, так и уже в изданной игре. Чтобы корректно определить возможность применения разработанного алгоритма на практике, требуется ввести объективные метрики оценки сгенерированных подземелий и провести эксперименты для их сбора. После чего можно будет провести анализ полученных результатов и сформулировать выводы.

Несколько базовых метрик для оценивания были основаны на шаблонах подземелья предложенных Дальскогом [35] и Чингом [36], а также на научной работе по сравнению двух разных генераторов двумерных подземелий [37]. В результате был получен следующий список характеристик для анализа:

1) Количество комнат
2) Отклонение размера комнат
3) Достижимость комнат
4) Проходимость закрытых комнат
5) Время выполнения генерации (миллисекунды)

Помимо этого, были добавлены еще 4 метрики, оценивающие насколько успешно система справляется с достижением значений, указанных в качестве целевых метрик для генетического алгоритма:

1) Количество коридоров
2) Разветвленность (линейность)
3) Наклон коридоров
4) Длина коридоров

4.1.1 Представление данных

Для анализа было решено использовать подземелье с шириной и глубиной в 30 ячеек, а высотой в 10 ячеек, что дает суммарный объем в 9.000 ячеек. Для сравнения, в работе по сравнению двумерных генераторов использовались подземелья размерностью 100x100, т.е. 10.000 клеток[3]. Также, для генетического алгоритма было установлено ограничение в 1000 генераций.

Так как сам генератор имеет обширную возможность кастомизации, протестировать все комбинации параметров является невозможным. Поэтому, было решено разделить тестирование на две части: базового и генетического функционала. Для первой части было решено зафиксировать почти все параметры, в том числе и настройки генетического алгоритма. Единственным изменяющимся параметром является количество дополнительных комнат, и целью данного тестирования является оценить справляется ли вообще генератор с задачей процедурного создания подземелья с учетом маркеров и их настроек. Таким образом, получается три набора исходных данных, которые представлены в таблицах 4.1.1, 4.1.2 и 4.1.3 ниже.

Таблица 4.1.1 – Параметры генерации комнат

Название параметра	Тип данных	Значение
ExtraRoomCount	int	5 / 10 / 15
MinSpawnRadius	int	5
RadiusOffsetMultiplier	float	0.5
RadiusIntersectMultiplier	float	1.25
MaxSpawnTries	int	5
RoomMinSize	Vector3Int	(6, 3, 6)
RoomMaxSize	Vector3Int	(10, 4, 10)
InteriorSpace	Vector3Int	(2, 0, 2)
GrowthSteps	int	20

Для теста вручную были размещены пять изначальных комнат – входа, выхода, босса, комната с ключом и, соответствующая ей, закрытая комната.

Таблица 4.1.2 – Параметры маркеров комнат

Тип комнаты	Название параметра	Значение
Вход	Position	(0, 0, 0)
	Size	(4, 3, 4)
	Generation actions	-
Выход	Position	(26, 0, 19)
	Size	(4, 3, 4)
	Generation actions	EnsureConnectionAction{ target = Boss, Remove others = true}
Босс	Position	(24, 6, 25)
	Size	(7, 3, 7)
	Generation actions	-
Ключ	Position	(5, 6, 24)
	Size	(0, 0, 0) - рандомизируется
	Generation actions	-
Замок	Position	(25, 2, 7)
	Size	(3, 2, 3)
	Generation actions	-

Таблица 4.1.3 – Параметры функций приспособленности

Тип функции	Название параметра	Значение
Количество коридоров	Weight	10
	Edges filling	0
Разветвленность	Weight	10
	Branching rate	0
Наклон коридоров	Weight	0
	Slope rate	0
Длина коридоров	Weight	0
	Length rate	0

Для проверки генетического алгоритма будет использоваться первый набор параметров генерации комнат (5 комнат) и те же маркеры, но для каждой функции приспособленности будут проведены персональные тесты. Для этого для всех, кроме тестируемой функции, веса будут установлены на значении 0. Тестируемая же функция будет проверена для 3 значений целевого параметра - 25, 50 и 75 процентов. Целью данного тестирования было оценить насколько хорошо генератор достигает показателей, установленных геймдизайнером.

Также, на первом этапе проведения экспериментов, будут выполнены 100.000 базовых генераций для 5 дополнительных комнат, после чего будет проведено сравнение базовых показателей с меньшей выборкой из 10.000 генераций, чтобы определить достаточно ли точными являются результаты при меньшем количестве генераций. Если набор из 10.000 окажется недостаточным, количество генераций будет постепенно увеличиваться, пока их результаты не будут близки к базовым. После определения удовлетворяющего количества генераций, будут проведены тесты для оставшихся двух наборов исходных данных и характеристик генетического алгоритма.

Все результаты для каждого набора учитываются при окончательной

оценке и сохраняются в файл CSV. Далее, во внешнем программном обеспечении, файл CSV преобразуется в списки значений. Каждый список сортируется, и данные представляются в виде диаграммы, чаще всего диаграммы размаха. Она показывает максимальное значение, минимальное значение, верхний квартиль, медиану и нижний квартиль, что иллюстрируется на рисунке 4.1 для ясности. Такие диаграммы очень хорошо визуализируют распределение всех данных и предоставляют информацию о стабильности данных. Также, аналогичные данные группируются для лучшей наглядности различий и возможных корреляций.

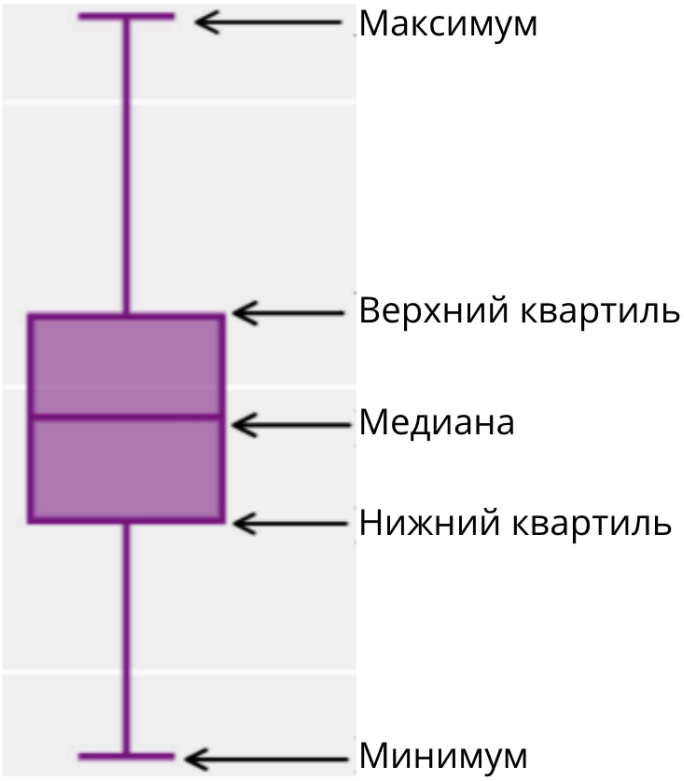

Рисунок 4.1 – Диаграмма пакетов реализуемой системы

4.2 Результаты экспериментов

Обычные оценки и исследования генераторов подземелий редко учитывают игровые метрики и часто сосредотачиваются только на оптимизации производительности. Однако, целью данных экспериментов также была и

оценка “играбельности” сгенерированных результатов. Результаты разделены на три секции в зависимости от количества тестов и их целей. Сначала результаты были собраны базовые метрики для 5 дополнительных комнат с использованием 100.000 итераций, однако это количество генераций оказалось непрактичным из-за длительного времени, необходимого для генерации и оценки такого большого числа уровней подземелий. Поэтому количество оценок было сокращено до 10.000, что оказалось оправдано, так как сравнение показало незначительную разницу между результатами для 10.000 и 1000.000, но при этом значительно сократилось требуемое время. В таблице 4.2 иллюстрируется разница между оцененными уровнями для 10.000 и 100.000 генераций, перечисляя средние значения базовых метрик.

Таблица 4.2 – Сравнение средних значений метрик для 100.000 и 10.000 генераций. Использовался набор данных с 5 дополнительными комнатами.

Метрика	100.000 генераций	10.000 генераций
Количество комнат	9.952	9.943
Отклонение размера комнат (%)	3.029	3.197
Достижимость комнат (%)	100	100
Проходимость закрытых комнат (%)	100	100
Время генерации (мс)	318.97	331.053

4.2.1 Базовые результаты 100.000 генераций

Для получения точных результатов в данном исследовании была выполнена генерация и оценка 100.000 подземелий. В следующий таблицах и графиках отображены результаты данных генераций. Набор данных был таким же, как и первый набор для оценки 10.000 генераций.

Как видно из таблицы 4.2.1, почти всегда генератору удавалось добавить все 5 дополнительных комнат, однако, были и редкие моменты когда последняя

комната не была создана. Также, хорошо показал себя алгоритм разрастания комнат, благодаря которому среднее отклонение составило около 3%. Максимальное же значение отклонения может быть объяснено комнатами, которые изначально имели маленький целевой размер и для них даже одно неудачное увеличение могло составлять значимую часть. Стоит упомянуть, что для расчетов отклонения размера комнат была использована средняя абсолютная процентная ошибка, которая рассчитывается по формуле

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{V_{\text{факт}} - V_{\text{цель}}}{V_{\text{цель}}}\right|,$$

где n – количество комнат в подземелье; V факт – фактический объем комнаты в результате работы алгоритма; V цель – целевой объем комнаты, указанный на маркере или полученный случайно.

Таблица 4.2.1 – Количество комнат и отклонение их размера

	Количество комнат	Отклонение размера комнат (%)
Медиана	10	2.222
Среднее значение	9.952	3.029
Минимум	9	0
Максимум	10	12.963
Нижний квартиль	10	0
Верхний квартиль	10	4.444

Согласно данным на таблице 4.2.2, алгоритм всегда успешно справлялся с достижением двух обязательных характеристик подземелья. При всех генерациях удавалось не просто соединить все комнаты, но и удостовериться в том, что закрытую комнату возможно будет открыть ключом.

Таблица 4.2.2 – Достижимость комнат и проходимость закрытых комнат

	Достижимость комнат (%)	Проходимость закрытых комнат (%)
Медиана	100	100
Среднее значение	100	100
Минимум	100	100
Максимум	100	100
Нижний квартиль	100	100
Верхний квартиль	100	100

Таблица 4.2.3 демонстрирует, что время генерации довольно стабильно держалось в районе 250-350 миллисекунд, иногда отклоняясь от данного интервала.

Таблица 4.2.3 – Время генерации

	Время генерации
Медиана	317
Среднее значение	318.97
Минимум	178
Максимум	544
Нижний квартиль	275
Верхний квартиль	353

4.2.2 Базовые результаты 10.000 генераций

На рисунке 4.2.1 видно, что если на первом наборе данных система почти всегда справлялась с генерацией всех комнат, то с ростом их количества это стало труднее. Данная тенденция привела к тому, что из 10 дополнительных комнат чаще всего создавалось лишь 9, а набор, подразумевающий суммарно 20 комнат, так ни разу и не достиг целевого значения, создавая чаще всего лишь 12 дополните комнат.

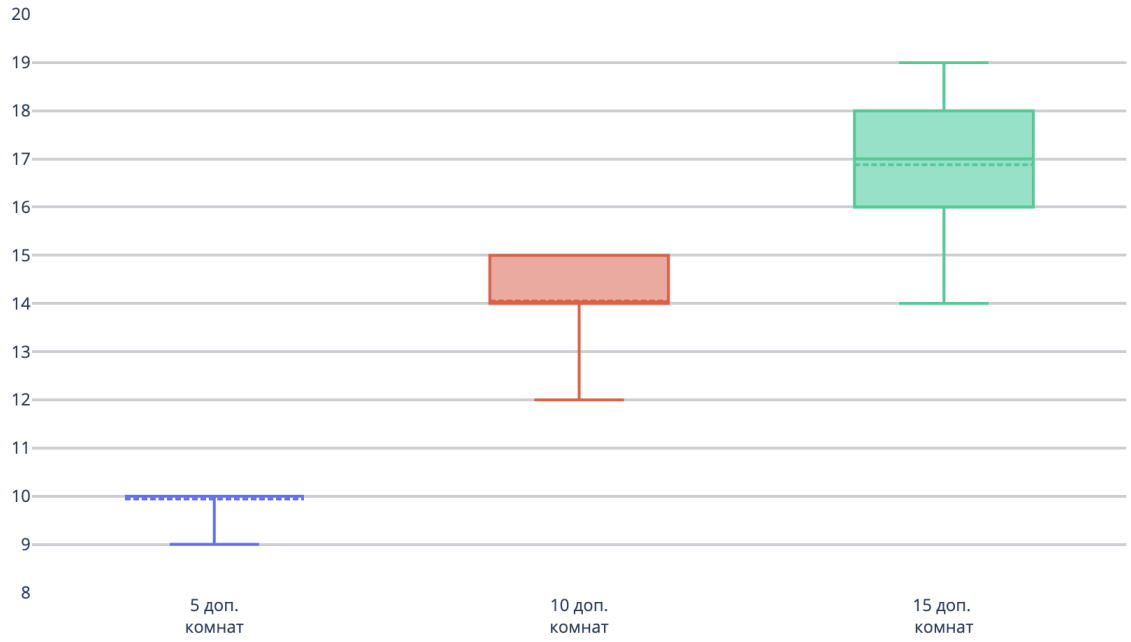

Рисунок 4.2.1 – Количество комнат

Из рисунка 4.2.2 была выявлена прямая зависимость отклонения объема комнат от их дополнительного количества. Так, например, при 15 дополнительных комнатах отклонение составило более 9%. Тем не менее, стоит заметить, что разброс отклонений всегда был примерно равен 4%, что говорит о том, что комнаты в любом случае растут согласованно, сохраняя их размеры относительно друг друга.

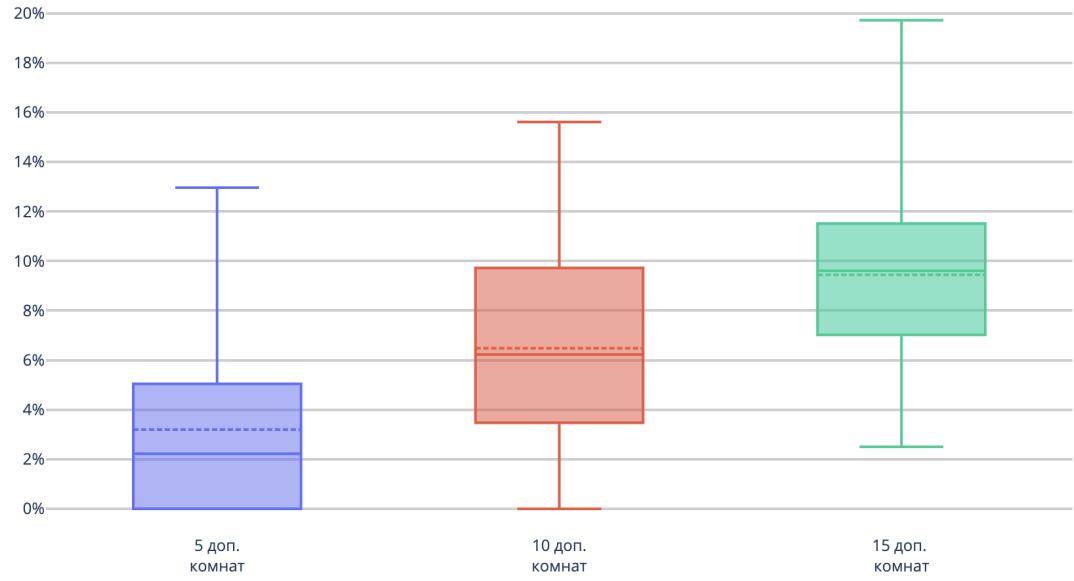

Рисунок 4.2.2 – Отклонение размера комнат

Согласно данным на рисунке 4.2.3, вне зависимости от количества комнат, алгоритм продолжил создавать полностью проходимые подземелья.

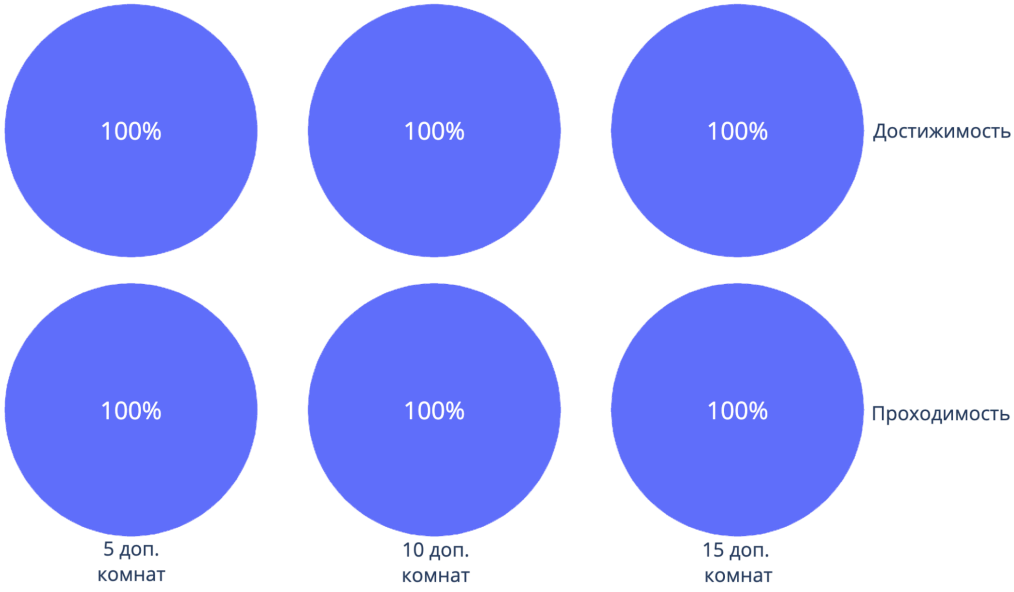

Рисунок 4.2.3 – Достижимость комнат и проходимость закрытых комнат

Последней из базовых, но не менее важной, характеристикой является время генерации подземелий. Из графиков (рисунок 4.2.4) видно, что время генерации имеет прямую зависимость от количества комнат и составило примерно 30 миллисекунд на комнату.

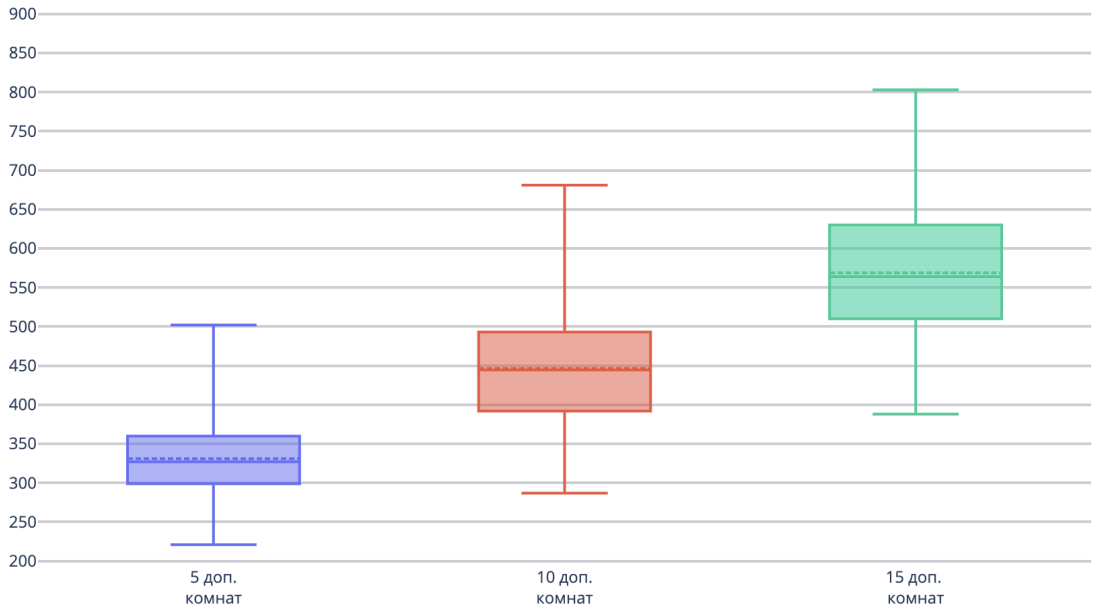

Рисунок 4.2.4 – Время генерации

4.2.3 Результаты генетической оценки 10.000 генераций

Из рисунка 4.2.5 видно, что средний процент отклонения фактического количества дополнительных коридоров от желаемого довольно низок и составляют около двух процентов вне зависимости от цели, будь то уменьшение или увеличение количества коридоров.

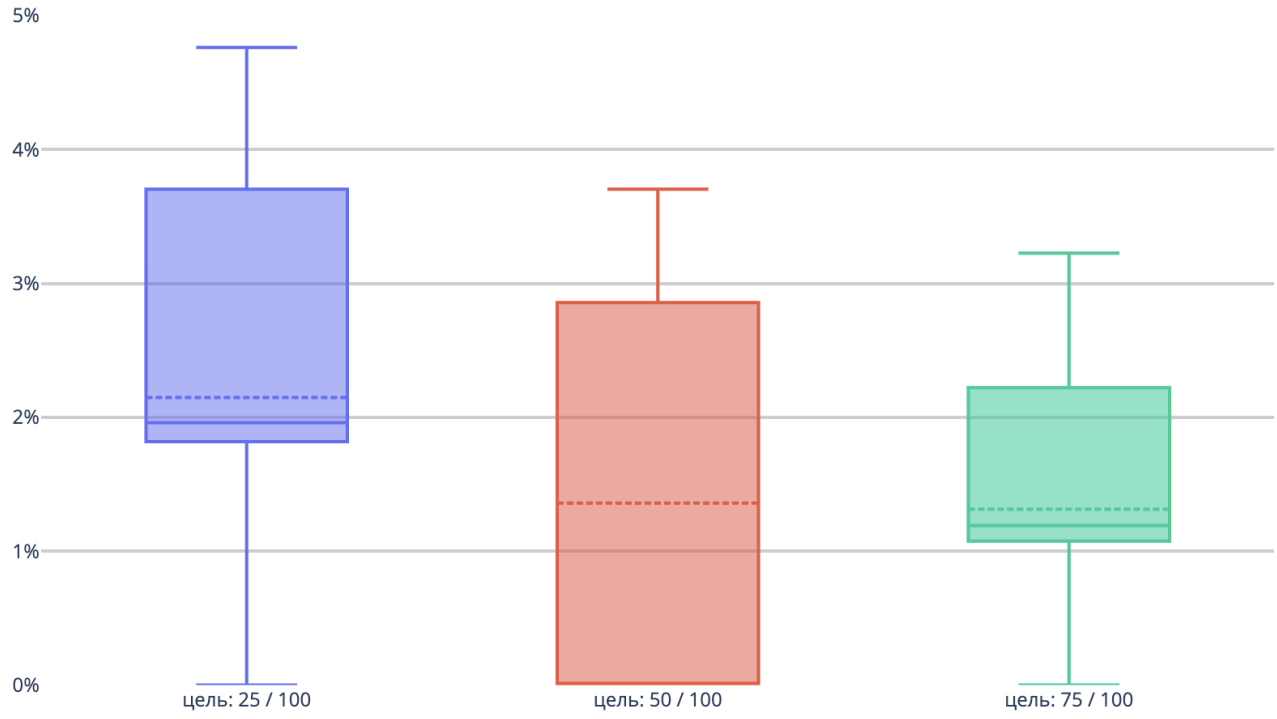

Рисунок 4.2.5 – Отклонение количества коридоров

Как упоминалось раньше, чем меньше показатель разветвленности, тем больше в подземелье комнат, имеющих ровно один вход и один выход. Согласно рисунку 4.2.6, алгоритм хорошо генерирует подземелья средней и выше разветвленности, но при попытке создавать линейные подземелья с ключами и замками иногда возникают отклонения.

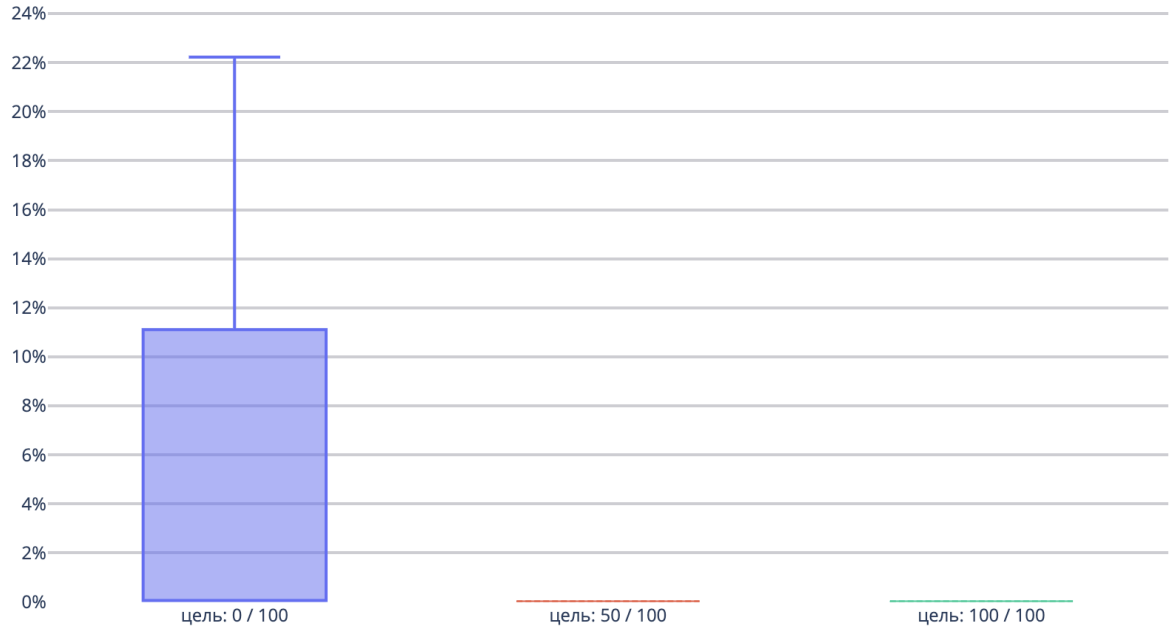

Рисунок 4.2.6 – Отклонение разветвленности подземелья

Так как все тесты проходили в объеме 30x10x30, то высота подземелья была более ограниченной по сравнению с горизонтальными характеристиками. Из-за этого генератор допускал меньше отклонений в наклоне для более пологих коридоров, а вот сделать крутые коридоры ему было уже сложно.

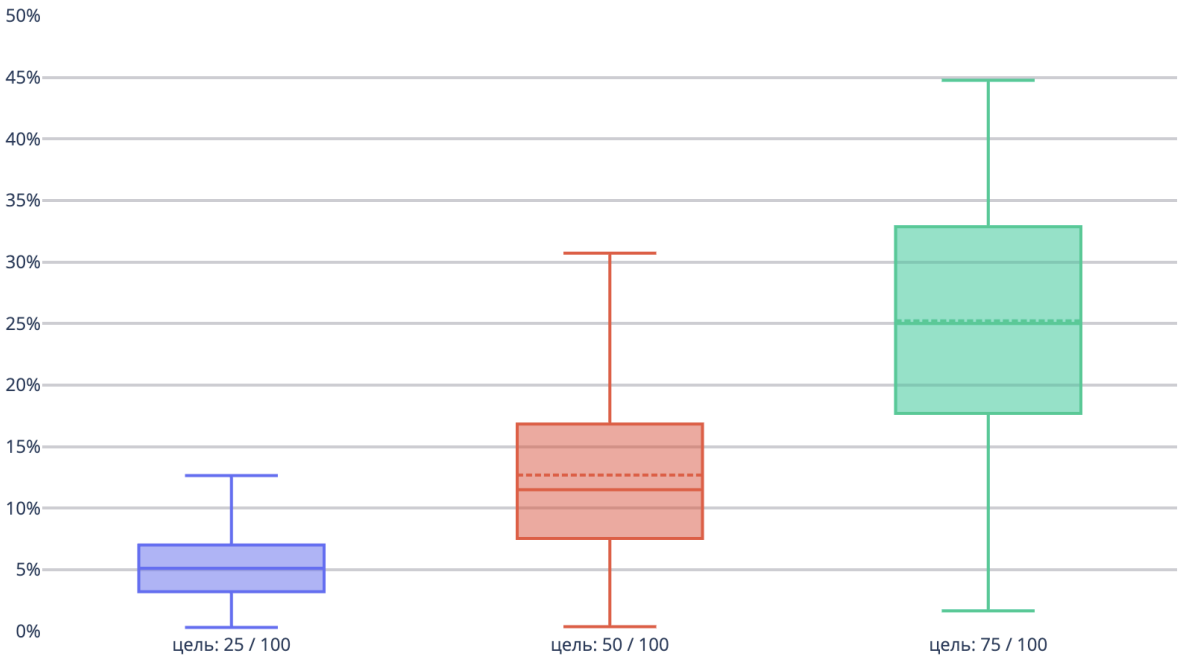

Рисунок 4.2.7 – Отклонение наклона коридоров

Из графиков (рисунок 4.2.8) видно, что лучше всего алгоритм справляется с расстановкой ребер средней длины, в то время как отклонения для коротких и длинных составило около 17%. Тем не менее, разброс находится в пределах 4%, что говорит о более менее равномерном распределении длин коридоров.

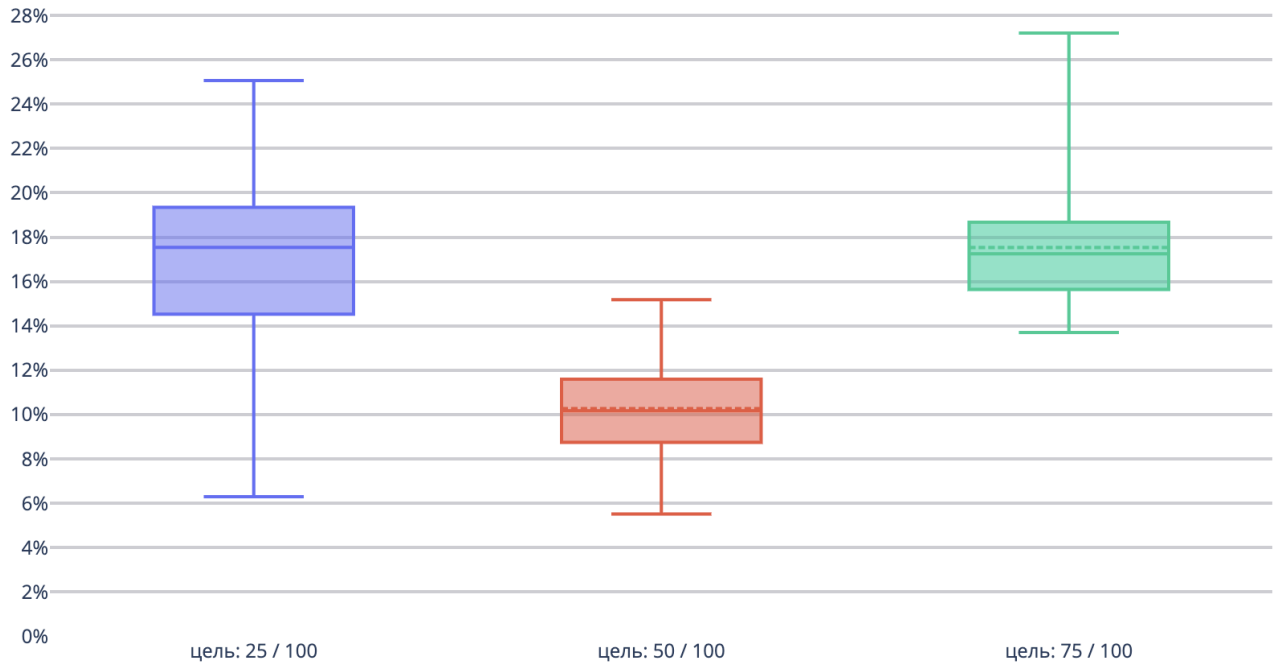

Рисунок 4.2.8 – Отклонение длины коридоров

4.3 Анализ результатов

Рассмотренные выше метрики предоставляют информацию о различных аспектах сгенерированных уровней, включая их размер, структуру, сложность и проходимость. Понимание этих метрик позволяет разработчикам игр улучшать и оптимизировать процесс генерации подземелий, обеспечивая более увлекательный и качественный игровой контент для игроков. Так, например, показатель количества комнат позволил убедиться, что генератор стремится к приближению конечного значения к заданному игровыми дизайнерами. К тому же, эксперименты показали, что алгоритм разрастания комнат оказался довольно эффективным, приводя к целевым или близким к ним размерам комнат. Также, согласно тестам, алгоритм всегда справляется с выполнением условий достижимости комнат и проходимости запертых участков.

Метрики генетического алгоритма показали, что разработанные функции оценки позволяют достигать желаемого результата, контролируя характеристики коридоров и показатель разветвленности подземелья.

Благодаря возможности заранее предсказать время генерации, разработчики могут решить стоит ли им иметь какой-то заранее сгенерированный набор или же можно использовать генерацию во время самой игры.

ЗАКЛЮЧЕНИЕ

На этапе постановки задачи были выдвинуты функциональные и нефункциональные требования. Среди первых были обозначены механизмы генерации комнат и коридоров на основе расставленных маркеров, что было достигнуто в результате комбинирования различных модулей системы. Нефункциональные требования включили в себя как легко собираемые показатели, так и более абстрактные понятия. Например, было легко измерить среднее время генерации и его зависимость от количества комнат. А вот оценить качественные характеристики было гораздо сложнее, для чего потребовалось вводить специфичные метрики, такие как проходимость и линейность подземелий. Гибкость же генератора достигнута благодаря использованию модульной системе компонентов, где каждый модуль имеет еще и свой набор настраиваемых параметров.

В итоге выполненную работу можно считать успешной, но не лишенной потенциальных доработок и улучшений. Примером таких улучшений может быть добавление новых типов комнат и “действий”, выполняемых во время и после генерации коридоров. Также, могут быть добавлены новые функции оценки генетического алгоритма и применены алгоритмы многоцелевой оптимизации. Для ускорения действия алгоритма можно использовать многопоточность. А для использования системы в непосредственной игре, можно добавить функционал постобработки, который бы на основе полученного скелета формировал готовые подземелья с противниками, ловушками и сокровищами.

Результаты данной работы, исходные коды и ресурсы опубликованы в общедоступном репозитории. Получить к ним доступ можно по ссылке: https://github.com/YahimaB/MarkerDungeonGeneration

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Hendrikx M. et al. Procedural content generation for games: A survey // ACM Transactions on Multimedia Computing, Communications, and Applications. New York, NY, USA: Association for Computing Machinery, 2013. Vol. 9, № 1. P. 1–22.
2. Blizzard. Senior Dungeon Level Designer job vacancy [Электронный ресурс]. URL: https://careers.blizzard.com/global/en/job/R018652/Senior-Dungeon-Level-Designer-World-of-Warcraft (дата обращения 10.01.2023).
3. Viana B.M.F., dos Santos S.R. A Survey of Procedural Dungeon Generation // 2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames). IEEE, 2019. P. 29–38.
4. Van der Linden R., Lopes R., Bidarra R. Designing procedurally generated levels // Proceedings of IDPv2 2013 - Workshop on Artificial Intelligence in the Game Design Process, co-located with the Ninth AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment. unknown, 2013. P. 41–47.
5. Karavolos D., Liapis A., Yannakakis G. Evolving missions for Dwarf quest dungeons // 2016 IEEE Conference on Computational Intelligence and Games (CIG). unknown, 2016. P. 1–2.
6. Baron J. Procedural Dungeon Generation Analysis and Adaptation // the SouthEast Conference. unknown, 2017. P. 168–171.
7. Santamaría-Ibirika A. et al. Procedural Playable Cave Systems Based on Voronoi Diagram and Delaunay Triangulation // 2014 International Conference on Cyberworlds. IEEE, 2014. P. 15–22.
8. Yannakakis G.N., Togelius J. Experience-Driven Procedural Content Generation // IEEE Transactions on Affective Computing. IEEE, July-Sep 2011. Vol. 2, № 3. P. 147–161.
9. Togelius J. et al. Search-Based Procedural Content Generation: A Taxonomy

and Survey // IEEE Transactions on Computational Intelligence and AI in Games. IEEE (Institute of Electrical and Electronics Engineers), 2011. Vol. 3, № 3. P. 172–186.

10. Zafar A., Mujtaba H., Beg M.O. Search-based procedural content generation for GVG-LG // Applied Soft Computing. 2020. Vol. 86. P. 105909.
11. Shaker N. et al. Constructive generation methods for dungeons and levels // Procedural Content Generation in Games. unknown, 2016. P. 31–55.
12. Van der Linden R., Lopes R., Bidarra R. Procedural Generation of Dungeons // IEEE Transactions on Computational Intelligence and AI in Games. IEEE (Institute of Electrical and Electronics Engineers), 2014. Vol. 6, № 1. P. 78–89.
13. Parish Y.I.H., Müller P. Procedural Modeling of Cities // Proceedings of the 28th annual conference on Computer graphics and interactive techniques. unknown, 2001. Vol. 2001. P. 301–308.
14. Summerville A. et al. Procedural Content Generation via Machine Learning (PCGML) // IEEE Transactions on Computational Intelligence and AI in Games. IEEE (Institute of Electrical and Electronics Engineers), 2017. Vol. PP, № 99.
15. Dungeon [Электронный ресурс] // Wiktionary. URL: https://en.wiktionary.org/wiki/dungeon (дата обращения 10.01.2023).
16. Pantaleev A. In Search of Patterns: Disrupting RPG Classes through Procedural Content Generation // Proceedings of the The third workshop on Procedural Content Generation in Games. New York, NY, USA: Association for Computing Machinery, 2012. P. 1–5.
17. Roguebasin. Cellular automata method for generating random cave-like levels [Электронный ресурс]. URL: http://www.roguebasin.com/index.php?title=Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels (дата обращения 10.01.2023).
18. Johnson L., Yannakakis G.N., Togelius J. Cellular automata for real-time

generation of infinite cave levels: Article 10 // Proceedings of the 2010 Workshop on Procedural Content Generation in Games. New York, NY, USA: Association for Computing Machinery, 2010. P. 1–4.

19. Андреев М. Генетический алгоритм. Просто о сложном. Рассказ Марка Андреева [Электронный ресурс] // Хабр. Habr, 2011. URL: https://habr.com/ru/post/128704/ (дата обращения 10.01.2023).
20. Valtchanov V., Brown J.A. Evolving dungeon crawler levels with relative placement. unknown, 2012.
21. Загиевский В.И. Информационные технологии в бизнесе: Учебное пособие. 2010.
22. Sommerville I. Software engineering,10th edition. 2015.
23. EnvavoTuts. Create a procedurally generated dungeon cave system [Электронный ресурс] // Code Envato Tuts+. Envato Tuts, 2013. URL: https://code.tutsplus.com/create-a-procedurally-generated-dungeon-cave-system--gamedev-10099t (дата обращения 15.07.2023).
24. Sheffield E.C., Shah M.D. Dungeon Digger: Apprenticeship Learning for Procedural Dungeon Building Agents // Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts. New York, NY, USA: ACM, 2018.
25. Roguebasin. Basic BSP dungeon generation [Электронный ресурс]. URL: https://www.roguebasin.com/index.php?title=Basic_BSP_Dungeon_generation (дата обращения 11.08.2023).
26. Lopes R. et al. A constrained growth method for procedural floor plan generation // Proceedings of 11th Int. Conf. Intell. Games Simul. unknown, 2010.
27. Wikipedia contributors. Триангуляция Делоне [Электронный ресурс] // Wikipedia, The Free Encyclopedia. URL: https://ru.wikipedia.org/wiki/Триангуляция_Делоне (дата обращения:

28.09.2023).
28. Bowyer A. Computing Dirichlet tessellations // Comput. J. Oxford University Press (OUP), 1981. Vol. 24, № 2. P. 162–166.
29. Watson D.F. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes // Comput. J. Oxford Academic, 1981. Vol. 24, № 2. P. 167–172.
30. Zemek M. Regular Triangulation in 3D and Its Applications. 2010.
31. Shewchuk J.R. Lecture notes on Delaunay mesh generation. people.eecs.berkeley.edu, 2012.
32. Klee V. Circumspheres and Inner Products // Math. Scand. 1960. Vol. 8. P. 363–370.
33. мKatoch S., Chauhan S.S., Kumar V. A review on genetic algorithm: past, present, and future // Multimed. Tools Appl. 2021. Vol. 80, № 5. P. 8091–8126.
34. Алгоритм A* [Электронный ресурс]. URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_A* (дата обращения 22.01.2024).
35. Dahlskog S., Togelius J., Björk S. Patterns, Dungeons and Generators // Int Conf Found Digit Game. 2015.
36. Ching F.D.K. Architecture: Form, Space, and Order. John Wiley & Sons, 2010. P. 253, 294, 428, 433, 448 p.
37. Naußed D., Sapokaite R. Evaluation of Procedural Content Generators for Two-Dimensional Top-Down Dungeon Levels. 2021.