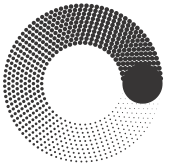
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ**

**УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**



**МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

***Факультет Информационных технологий***

***Кафедра Информатики и информационных технологий***

**направление подготовки**

**09.03.02 «Информационные системы и технологии»**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**БАКАЛАВРСКАЯ РАБОТА**

**Тема:** Разработка системы процедурной генерации уровней в игровом проекте

**Исполнитель** Погодин Михаил Денисович

(Фамилия И.О.)(Подпись)

**Руководитель** Алпатова Марианна Валерьевна

(Фамилия И.О., степень, звание)(Подпись)

**Нормоконтроль**

(Фамилия И.О., степень, звание)(Подпись)

**Антиплагиат**

(Фамилия И.О., степень, звание)(Подпись)

**«ДОПУЩЕНО К ЗАЩИТЕ»**

**Зав. кафедрой ИиИТ: Е.В. Булатников**

(Подпись)

**Москва**

**2022**

**ОГЛАВЛЕНИЕ**

[**ВВЕДЕНИЕ** 3](#_Toc105063064)

[**1. АНАЛИТИЧЕСКАЯ ЧАСТЬ** 4](#_Toc105063065)

[**1.1 Описание предметной области** 4](#_Toc105063066)

[**1.2 Понятие пошаговой стратегии** 7](#_Toc105063067)

[**Понятие процедурной генерации** 8](#_Toc105063068)

[**1 Общие определения** 8](#_Toc105063069)

[**2 Критерии** 8](#_Toc105063070)

[**Аппаратное обеспечение** 9](#_Toc105063071)

[**Инструментарий, язык программирования** 9](#_Toc105063072)

[**1 Unreal Engine** 9](#_Toc105063073)

[**2 Rider for Unreal Engine** 9](#_Toc105063074)

[**1.3. Цель проекта, основные задачи.** 10](#_Toc105063075)

[**Выводы к 1 главе** 11](#_Toc105063076)

[**2 ПРОЕКТНО-КОНСТРУКТОРСКАЯ ЧАСТЬ** 12](#_Toc105063077)

[**2.1 Задачи** 12](#_Toc105063078)

[**Выбор метода** 12](#_Toc105063079)

[**2.1.2 Описание метода** 14](#_Toc105063080)

[**2.2 Процесс разработки** 16](#_Toc105063081)

[**Сетка шестигранников** 19](#_Toc105063082)

[**Создание коридоров** 21](#_Toc105063083)

[**Поиск маршрута** 21](#_Toc105063084)

[**Поиск клеток в радиусе** 21](#_Toc105063085)

[**Обработка пользовательского ввода** 21](#_Toc105063086)

[**2.2 Создание новых типов клеток** 22](#_Toc105063087)

[**2.2.1 Класс TileBase и его производные** 22](#_Toc105063088)

[**2.3 Создание новых типов комнат** 24](#_Toc105063089)

[**2.3.1 Определение параметров комнаты** 24](#_Toc105063090)

[**2.3.2 Функция GenerateRoom** 24](#_Toc105063091)

[**Выводы ко 2 главе** 25](#_Toc105063092)

[**3 ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ** 26](#_Toc105063093)

[**3.1 Описание реализации** 26](#_Toc105063094)

[**3.2 Классы и методы** 26](#_Toc105063095)

[**3.2.1 Класс TileBase** 26](#_Toc105063096)

[**3.2.2 Методы класса TileBase** 27](#_Toc105063097)

[**3.2.3 Класс TileMap** 27](#_Toc105063098)

[**3.2.4 Методы класса TileMap** 28](#_Toc105063099)

[**3.3 Результаты выполнения программы** 32](#_Toc105063100)

[**Пример интеграции** 33](#_Toc105063101)

[**Работа генератора** 33](#_Toc105063102)

[**User Interface** 33](#_Toc105063103)

[**Управление** 33](#_Toc105063104)

[**Выводы к 3 главе** 33](#_Toc105063105)

[**Заключение** 34](#_Toc105063106)

[**Список литературы** 35](#_Toc105063107)

[**ПРИЛОЖЕНИЕ В** 36](#_Toc105063108)

# **ВВЕДЕНИЕ**

Главной целью данной дипломной работы является исследование способов процедурной генерации уровней и реализация собственного алгоритма на основе выбранного метода. Результатом проделанной работы станет программа, способная строить систему комнат на двумерной сетке шестигранников и параметризировать наполнение комнат, их количество и наличие определенных типов. В дальнейшем предполагается развитие и расширение системы с интеграцией в компьютерную игру, в которой она будет выполнять функцию генератора уровней.

На текущий момент, на самой популярной платформе цифровой дистрибьюции компьютерных игр Steam игр с меткой “Рогалик” более 1900, что делает тему создания инструмента для игр подобного жанра актуальной. Однако, если добавить в фильтр метки “Процедурная генерация” и “Пошаговая стратегия” игр останется чуть меньше 100 штук, что означает, что данный жанр не успел надоесть игроку и данная ниша свободна.

В первой главе рассматривается предметная область и обуславливается выбор технологий для реализации проекта. Ставятся цели и задачи, реализация которых описана в последующих главах. В проектно-конструкторской части рассматриваются некоторые методы генерации, описывается выбранный алгоритм и способы реализации имеющихся задач. В практической части описывается интеграция системы в игровой проект, представлен функционал обработки пользовательского ввода и интерфейс приложения. Содержится последовательность работы для расширения базы генерируемого контента и документация.

# **1. АНАЛИТИЧЕСКАЯ ЧАСТЬ**

## **1.1 Описание предметной области**

Конечный результатом проделанной работы является приложение, демонстрирующее систему процедурной генерации уровней на двумерной сетке шестигранников. Предметной областью дипломного проекта становится система процедурной генерации уровней. **Система процедурной генерации уровней** – это программа, следующая определенным алгоритмам для создания бесконечного количества случайных уровней, соответствующих заранее определенным параметрам.

Построение уровней с помощью алгоритмов процедурной генерации – распространенная задача, с которой сталкиваются в сфере компьютерных игр. Этими алгоритмами принято пользоваться, если существует необходимость быстро создавать большое количество игровых уровней с уникальными параметрами, которое недостижимо при разработке в реальные сроки и при разумных трудозатратах обычным способом построения уровней вручную.

При создании уровней руками, левел-дизайнер проходит длительный процесс расстановки контента сначала на блоках без текстур, отвечающих за геймплей, затем происходит множество итераций этого уровня для поиска наиболее интересной и понятной игроку конфигурации.

Данный подход имеет серьезный недостаток в контексте инди-разработки при коротких сроках – невозможность быстрого итерирования контента и отсутствие параметров уровня. При процедурной генерации же наполнение уровня расставляется с помощью алгоритма, что позволяет пропустить длительный процесс итерирования в поисках непривычного решения.

Итак, главными преимуществами процедурной генерации являются:

* **Скорость** генерации в соответствии с нуждами игрового процесса. Как уже описано выше, создание множества уровней с помощью человека займет гораздо больше времени.
* **Надежность** системы. Генерируемый контент соответствует предварительно заданным параметрам, из чего вытекают следующие два преимущества.
* **Регулируемость** параметров для изменения создаваемого контента в зависимости от ситуации. Нам может понадобиться уровень без ветвлений на 10 комнат, а может потребоваться зацикленный лабиринт, содержащий уже 100 комнат.
* **Разнообразие** генерируемого контента, что выражается в доступной для расширения системе. Добавление новых типов наполнения, будь то отдельные клетки или целые комнаты с их наполнением, не является проблемой.
* **Правдоподобность** генерируемого контента, которая не позволяет игроку задуматься о том, что уровни создавал не человек, хотя для игроков жанра рог-лайк это не будет новостью, однако это не должно бросаться в глаза.

В контексте темы диплома процедурная генерация должна выполнять следующие задачи:

* Создание игровых комнат в соответствии с заданным типом
* Создание коридоров между комнатами
* Наполнение игрового уровня ячейками-клетками для передвижения по ним игрового персонажа
* Хранение и обработка массива ячеек для выполнения задач взаимодействия между ячейками и управляемым героем

Появление систем процедурной генерации уровней непосредственно связано с историей игр жанра рог-лайк. В 1980 году студенты Калифорнийского университета в Санта-Крузе Гленн Вичман и Майкл Той объединились для создания игры, которая бы случайным образом изменяла схему лабиринта и расположение монстров/предметов в Colossal Cave Adventure. При помощи программной библиотеки Unix под названием curses, которая была своего рода инструментом для рисования в текстовом режиме, пара разработчиков создала приключение в подземном мире, в котором нужно было путешествовать с уровня на уровень, каждый из которых состоял из девяти или менее комнат, соединённых коридорами, в поисках Амулета Йендора. На самом деле, они не искали Амулет, он был всего лишь приманкой, наративной причиной спуска в опасное подземелье. При запуске новой игры уровни, свитки, оружие и расположение объектов изменялись случайным образом. Предназначение предметов было неизвестно (если только у вас не находилось свитка идентификации), пока игрок не осмелится его использовать, рискуя нанести себе непреднамеренный урон. Разработчики распространяли исполняемые версии своей игры Rogue по всем кампусам Калифорнийского университета. В ней было 26 типов монстров — по одному на каждую букву алфавита (заглавную), а персонаж игрока обозначался символом "@", который мог перемещаться не быстрее и не медленнее, чем монстры. Совершая любое из действий — перемещаясь в одном из направлений, читая свиток, ища ловушки или подбирая предмет — игрок тратил один ход. Действия были привязаны к различным клавишам: «w» — использовать оружие, «W» — надеть броню, «t» — бросить оружие, h, j, k и l — переместиться, и так далее. И смерть, которая запросто настигала даже опытных игроков, была перманентной — пользователь должен был сосредоточиться на освоении механик, а не на дизайне уровней игры.

На сегодняшний день игры жанра рог-лайк продолжают набирать популярность и обрастать новыми механизмами генерации внутри жанра, однако в данной работе было решено создать систему, заточенную под игры в жанрах пошаговая стратегия и рог-лайк. Данные игровые проекты подразумевают геймплейную систему, основанную на тактических решениях и дискретности действий игрока и искусственного интеллекта.

## **1.2 Понятие пошаговой стратегии**

Пошаговая стратегия – жанр компьютерных игр, игровой процесс которых состоит из последовательности ходов (шагов), фиксированных во времени.

Основными критериями пошаговой стратегии является дискретность во времени и наличие тактической составляющей. Игровой процесс таких игр зачастую более длительный и не требующий мгновенных решений. Во время своего хода игрок раздает указания своим игровым персонажам (далее – юнитам) в соответствии с текущим положением на игровом поле.

Примерами пошаговых стратегий являются:

* Серия игр Civilization
* Серия игр Heroes of Might and Magic
* Серия игр X-COM
* Final Fantasy Tactics



Рисунок 1 Игровой процесс пошаговых стратегий может отличаться в зависимости от фазы геймплея

Во многих пошаговых стратегиях реализована более комплексная логика игрового окружения и боевых единиц. Таким образом, в играх появляется возможность использования рельефа, дальности видимости, укрытий и т.д. для достижения преимущества над соперником. Также юниты могут иметь индивидуальную систему развития, по мере прохождения и получения опыта, кроме того, индивидуализация юнитов может до настройки внешности и личностных качеств, что формирует у игрока эмоциональную связь с созданными персонажами и подталкивает к более бережному геймплею за более развитых юнитов, нежели за новобранцев в отряде.



Рисунок 2 Пример внешнего вида персонифицированных юнитов отряда.

В контексте данного проекта игровой процесс должен представлять из себя камерные боевые действия, в которых активно используются тактические решения, так как управляемые игроком персонажи не всегда сильнее своих противников. Сами же сражения происходят в отдельных комнатах процедурно-генерируемого подземелья. Пример того, как может выглядеть подземелье показан на изображении ниже.

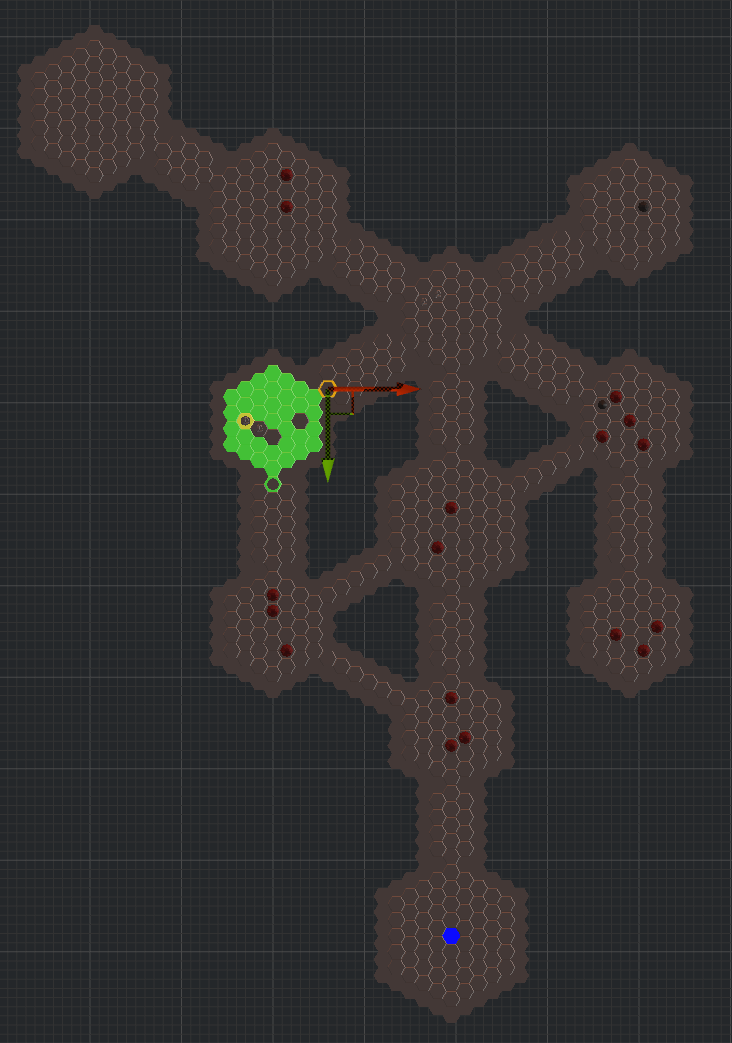
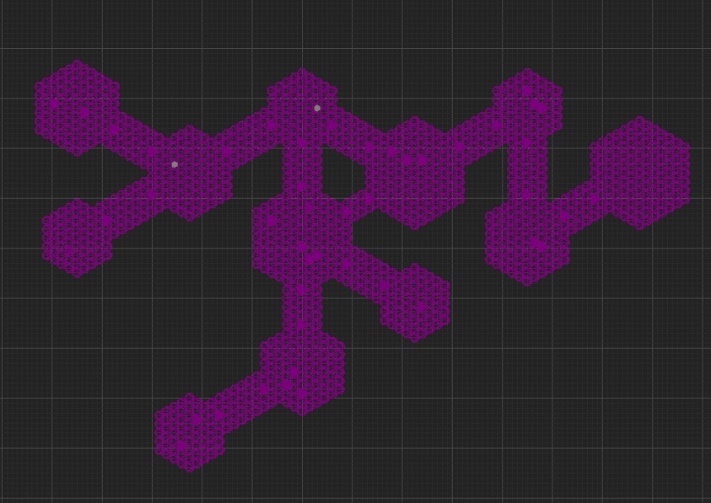


Рисунок 3 Примеры генерируемых карт с помощью разработанного в контексте проекта алгоритма

## **Понятие процедурной генерации**

### **1 Общие определения**

### **2 Критерии**

## **Аппаратное обеспечение**

## **Инструментарий, язык программирования**

В данном разделе описывается выбранное для разработки программное обеспечение. В контексте игрового движка выбор пал на Unreal Engine, а разработка программной части велась в IDE Rider. Изначально система генерации уровней строилась на этом решении, поэтому стоит перечислить преимущества данного выбора.

### **1 Unreal Engine**

Unreal Engine — игровой движок, разрабатываемый и поддерживаемый компанией Epic Games. Первой игрой на этом движке был шутер от первого лица Unreal, выпущенный в 1998 году. Хотя движок первоначально был предназначен для разработки шутеров от первого лица, его последующие версии успешно применялись в играх самых различных жанров, в том числе стелс-играх, файтингах и многопользовательских ролевых онлайн-играх. В прошлом движок распространялся на условиях оплаты ежемесячной подписки; с 2015 года Unreal Engine бесплатен, но разработчики использующих его приложений обязаны перечислять 5% роялти от общемирового дохода с некоторыми условиями.

Написанный на языке C++, движок позволяет создавать игры для большинства операционных систем и платформ: Microsoft Windows, Linux, Mac OS и Mac OS X; консолей Xbox, Xbox 360, Xbox One, PlayStation 2, PlayStation 3, PlayStation 4, PSP, PS Vita, Wii, Dreamcast, GameCube и др., а также на различных портативных устройствах, например, устройствах Apple (iPad, iPhone), управляемых системой iOS и прочих. (Впервые работа с iOS была представлена в 2009 году, в 2010 году продемонстрирована работа движка на устройстве с системой webOS).

### **2 Rider for Unreal Engine**

Rider — быстрая и мощная IDE для Unreal Engine и C++-разработки под Windows. Она помогает анализировать файлы Blueprints, использовать механизм рефлексии, следить за стилем и точностью кода. Инструмент доступен бесплатно в рамках программы раннего доступа.

Rider обладает нативной поддержкой C++, способен обрабатывать файлы Blueprints в проекте Unreal Engine и помогает ускорить процесс разработки с помощью макросов рефлексии Unreal Engine.

Файлы Blueprints представляют собой данные в бинарном формате, редактирование которых происходит как правило в визуальном редакторе внутри Unreal Editor. Объекты в этих файлах наследуются от классов на C++, переопределяют параметры из C++ части игры. И Rider for Unreal Engine является той уникальной средой разработки, которая зачитывает все необходимые файлы Blueprints и показывает эти связи в редакторе кода на C++.

Рефлексия в Unreal Engine реализована с помощью специальных макросов (UCLASS, UFUNCTION, UPROPERTY и др). Rider знает, что параметры таких макросов — это не просто текст. Парсер языка C++ в ReSharper C++ и Rider умеет действительно “понимать” значение этих макросов, даже не запуская Unreal Header Build tool.

## **1.3. Цель проекта, основные задачи.**

Основной целью проекта является разработка системы процедурной генерации уровней для игрового проекта. Система процедурной генерации предназначена для упрощения создания игрового наполнения, и должна обладать соответствующими характеристиками. Процесс разработки описан в данной главе с точки зрения алгоритмов и методов реализации, с помощью блок схем и подробного описания шагов.

Завершив работу над системой генерации, что подразумевает удовлетворительные результаты по критериям скорости, надежности и расширяемости системы, целью становится создание слаженной системы взаимодействия пользователя с создаваемым игровым пространством, то есть возможность передвигаться, ориентироваться в игровом мир и интерфейсе, взаимодействовать с необходимыми компонентами. Данный этап находится на стадии активной разработки.

Также существует необходимость создать систему, связующую разрабатываемый генератор с файлами конфигураций для обеспечения большей гибкости в настройках и удобства в хранении данных об уровнях.

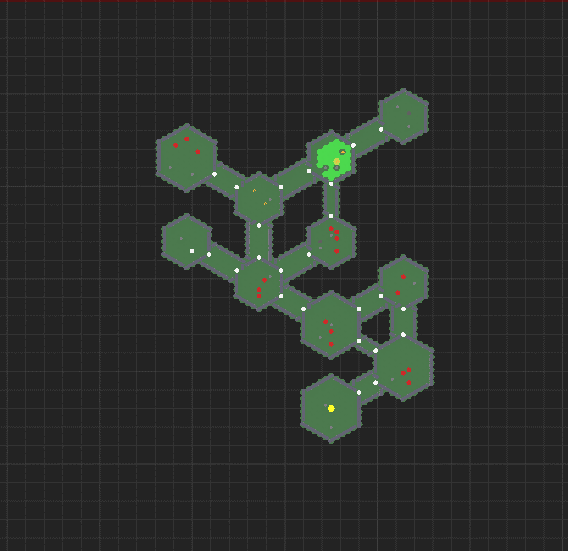


Рисунок 4 Вид сверху на результат работы генератора.

## **Выводы к 1 главе**

# **2 ПРОЕКТНО-КОНСТРУКТОРСКАЯ ЧАСТЬ**

## **2.1 Задачи**

Первоначально для реализации системы процедурной генерации уровней становится необходимо определиться с методом процедурной генерации, избранном в конкретном проекте. Для данного проекта было рассмотрено несколько подходов, из которых был выбран один и в дальнейшем доработан для нужд проекта. Далее будут описаны некоторые методы процедурной генерации уровней.

## **Выбор метода**

**Метод двоичного разделения пространства** подразумевает последовательное разделение некоторой, заранее определенной области, надвое, несколько раз. Деления производятся либо по горизонтали, либо по вертикали, и после выполнения всех операциях деления, в каждом из полученных участков необходимо создать меньшие области, которые станут комнатами. После этого в соответствии с неким алгоритмом, эти комнаты соединяются между собой.

Из плюсов можно выделить отсутствие пересечения комнат и ощущение иерархии в построенных системах комнат. Однако минусами данной системы становятся отсутствие цикличности лабиринтов и отсутствие удобства создания клеточных уровней для пошаговых игровых проектов

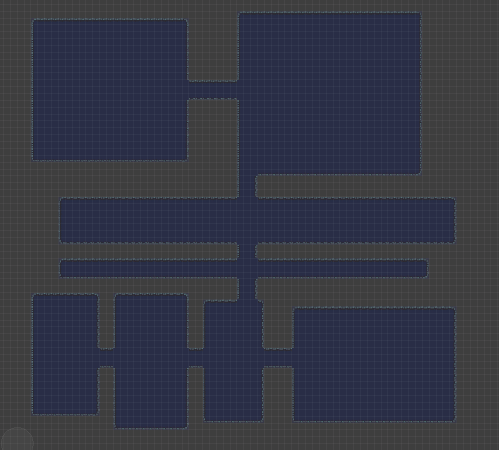


Рисунок 5 Пример работы BSP-генератора

Идея **метода клеточных автоматов** состоит в том, чтобы заполнить первоначальный массив ячеек случайным образом, а затем итерировать полученный массив с помощью следующего правила: плитка становится стеной, если область вокруг нее содержит не менее 5 стен. После нескольких подобных итераций количество шума уменьшается и появляется подобный пещере узор.

Минусами данной системы является отсутствие возможности создавать отдельные комнаты и соответственно их наполнять.

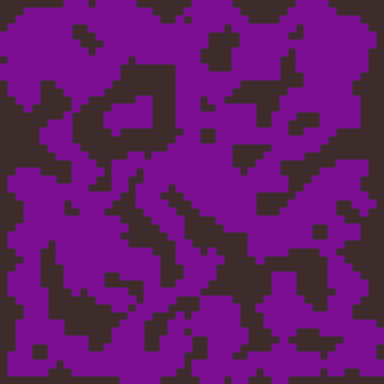


Рисунок 6 Пример работы алгоритма создания пещер

Последним из рассматриваемых методов стал **марширующий метод**, который подразумевает направленное создание комнат в подземелье, образуя подобие кривой, с возможностью соединить соседствующие комнаты в дальнейшем. Такой подход исключает пересечение комнат, однако позволяет склеивать ближайшие при желании. Также данный подход предполагает возможность ветвления, что добавляет полученной кривой сложности.

Данный подход подходит клеточной системе и позволяет создавать более интересные конструкции, чем в вышеописанных примерах, в связи с чем было принято решение взять за основу марширующий метод генерации на сетке шестигранников.

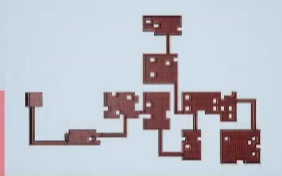


Рисунок 7 Пример работы марширующего алгоритма

### **2.1.2 Описание метода**

За основу генерации взят метод а основе сетки. Это простой метод создания карт, содержащих комнаты, которые гарантированно вписываются в пространство подземелья. Разделив пространство подземелья на набор ячеек, комнаты можно разместить в пространстве без необходимости проверять, перекрывает ли каждая комната другую комнату или выходит за пределы пространства подземелья.

В большинстве игр жанра роглайк, подземелье представляет собой массив определенного типа данных, где каждая ячейка в массиве определяет плитку (местоположение) в подземелье. Параметры плитки могут быть простым типом данных, таким как целое число, или более сложным составным типом, таким как структура. Особенности рельефа подземелья, то есть стены, полы, двери и так далее, обычно являются лишь частью общего описания данных подземелья.

После выбора метода появилась необходимость формализации алгоритма в виде блок-схем в соответствии с задумкой проекта жанра пошаговая стратегия с элементами рог-лайк. Блок-схема основного алгоритма, без раскрытия основных подпроцессов представлена ниже на Рисунке 5. Подробное раскрытие блок-схем подпроцессов представлены в Приложении Б.

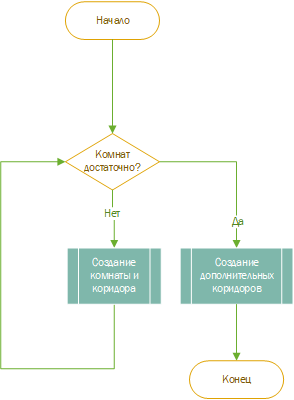


Рисунок 8 Алгоритм работы генератора

После утверждения алгоритма появляется необходимость определиться с обрабатываемыми и хранимыми данными о создаваемых ячейках. Так, при наполнении комнат, генератор задает клеткам различный параметр их типа, также система должна знать координаты клеток, их содержимое и состояние.

Взаимодействие игровых персонажей с клетками происходит с помощью передачи сигнала от объекта-персонажа к объекту-клетке о попытке взаимодействия и дальнейшего определенного функционала в зависимости от типа клетки.

## **2.2 Процесс разработки**

Первоначальной задачей в реализации алгоритма построения подземелий на сетке шестигранников стало понимание правил создания комнат, поиска соседних клеток по индексу.

Расстановка комнат без пересечений реализуется достаточно просто с помощью бэкапов двумерных массивов и предварительных проверок. Пример результата расстановок можно видеть на изображении ниже.

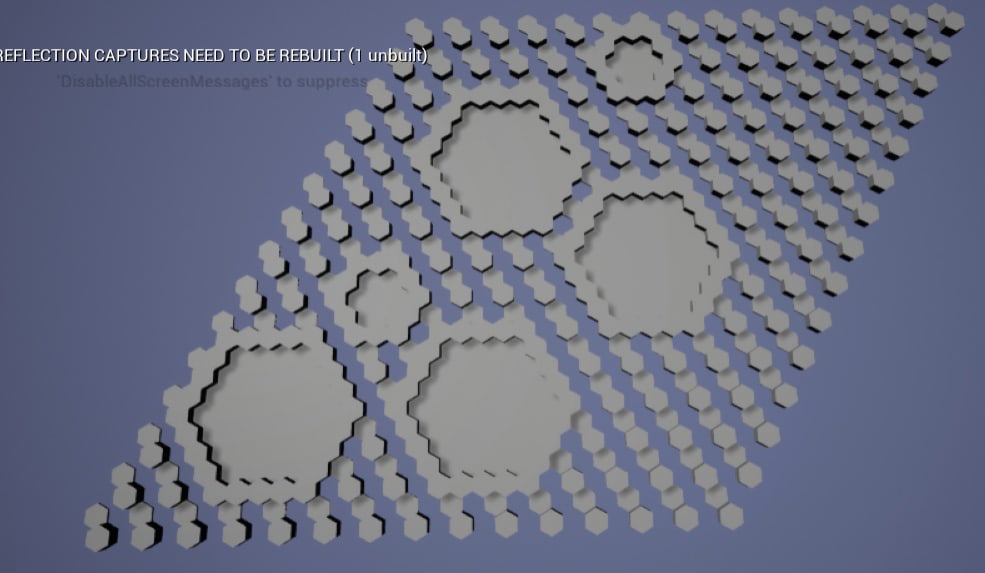


Рисунок 9 Этап создания комнат

Следующей задачей стало создание коридоров между комнатами, и это вызвало определенные трудности, ведь комнаты могут не пересекаться, а Г-образный коридор на сетке шестигранников труднопостроим.

Было принято решение создавать коридоры лишь из углов комнат, что внесло изменения в расстановку комнат, теперь для определения совместимости комнат достаточно знать координаты их центров, ведь если они равны по одной из осей, то коридор придет из угла Х одной комнаты в угол Х-3 второй комнаты.

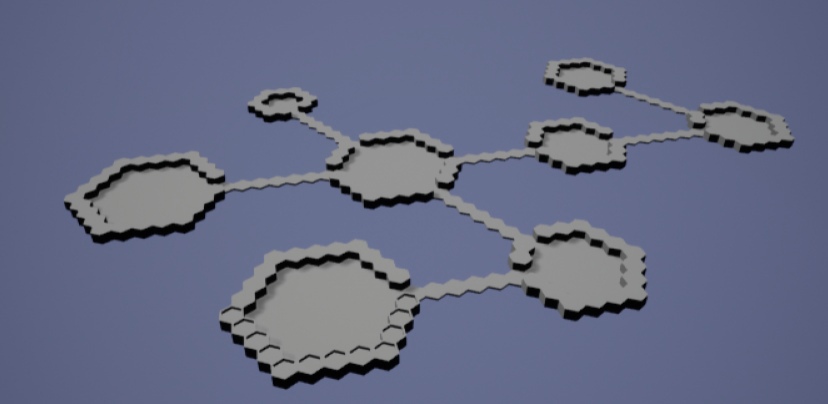


Рисунок 10 Этап создания коридоров

После добавления стен в коридорах возникла необходимость в создании зацикленности в существующих подземельях. Решением стало создание фиксированного расстояния между центрами комнат, что позволило добавить коридоры после основной генерации уровня. Таким образом, сначала создается ветвистая система комнат, а затем с определенной вероятностью некоторые комнаты соединяются между собой, образуя цикличные лабиринты.

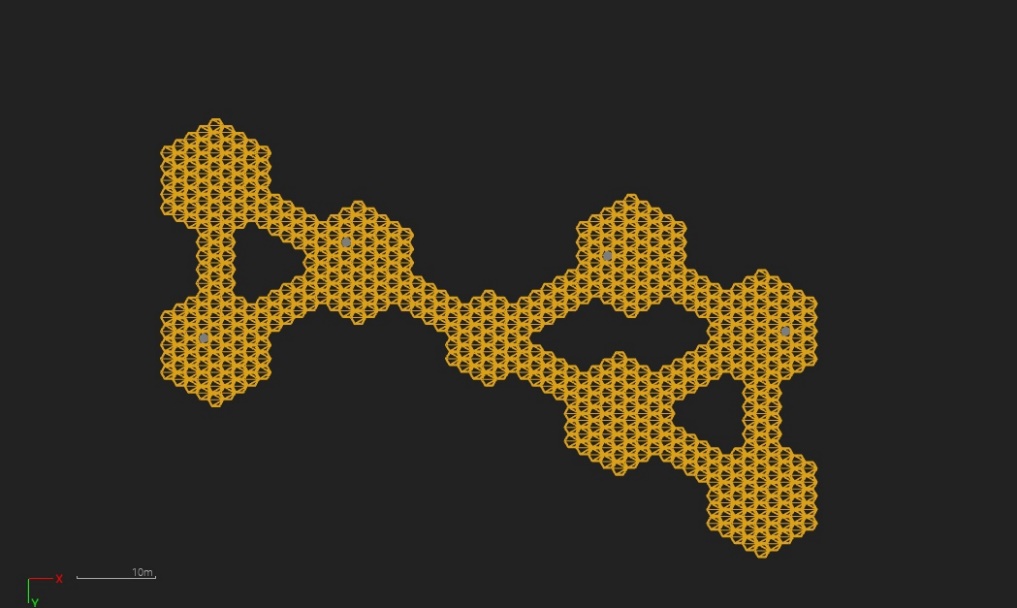


Рисунок 11 Один из промежуточных результатов построения уровня.

Затем в систему были добавлены типы комнат, что позволило определять возможность ветвления и соединения с определенными типами комнат. Так, например, перед комнатой, в которой находится выход из уровня, обязательно находятся две комнаты, имеющие только один вход и выход. Также один лишь вход имеют комнаты с сундуком, или магазин.

Таким образом, была построена система, генерирующая неповторяющиеся подземелья на гексагональной сетке с возможностью наполнения комнат в зависимости от их типов.

Данная система также подразумевает наличие типов клеток, имеющих отличающуюся логику в зависимости от их типа.

Итоговый результат работы генератора можно увидеть на изображении ниже.

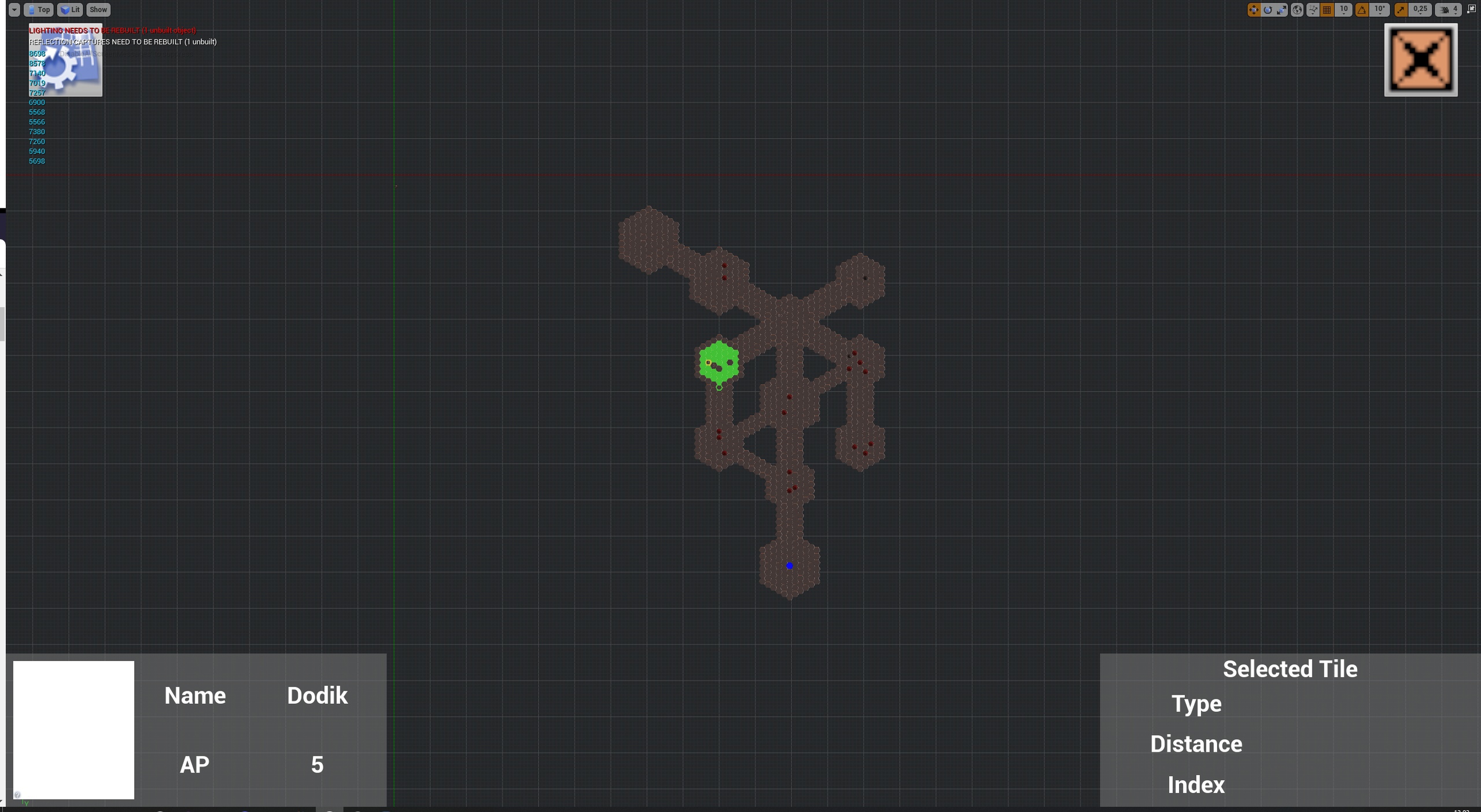
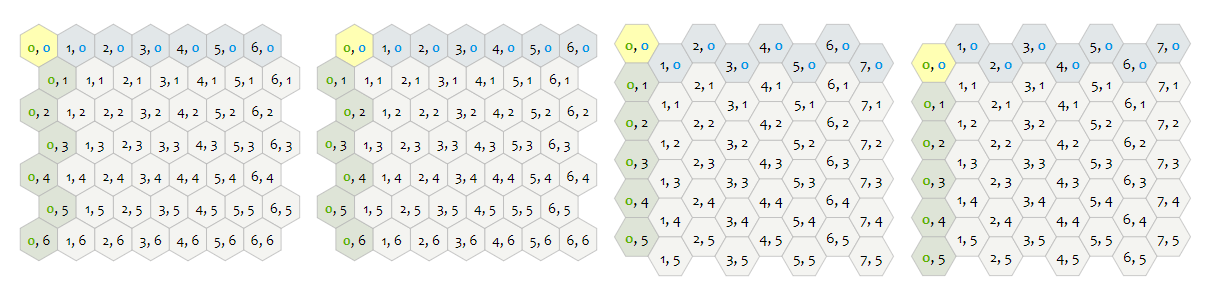


Рисунок 12 Результат с наполнением комнат.

## **Сетка шестигранников**

Для размещения клеток на уровне, системе нужно знать, куда и какие клетки ставить. С этой целью были определены система координат и правила размещения клеток на этой сетке. Существует три способа рассмотрения координат на сетке шестигранников:

**Координаты сдвига** подразумевает сдвига каждого последующего столбца (обозначается **q**) или ряда (обозначается **r**). Мы можем получить несколько вариаций итоговой сетки за счёт сдвигов только четных или нечетных колонок или рядов. Таким образом уже имеется четыре варианта построения сетки на выбор.

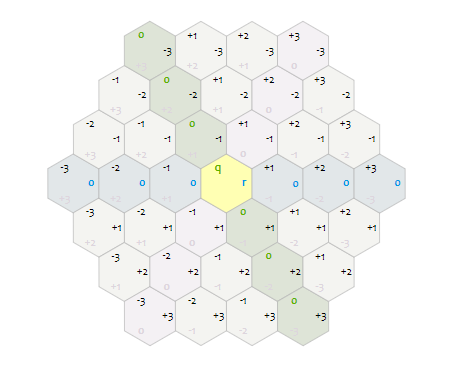
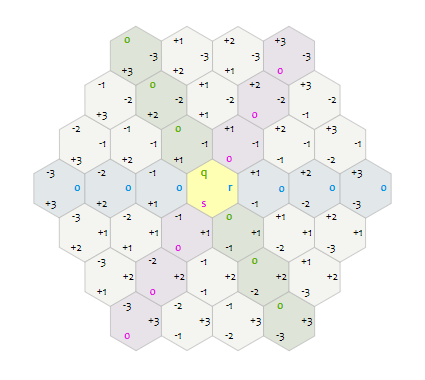


*Рисунок 13 Вариации сдвигов четных или нечетных столбцов или рядов*

Также возможно производить сдвиг столбец за столбцом или ряд за рядом, что добавляет еще два варианта построения. Однако если первые четыре образуют нечто, отдаленно напоминающее прямоугольник, то оставшиеся будут создавать ромбическую конструкцию. Этот подход больше походит на другой вид рассмотрения координат, который будет рассмотрен в дальнейшем.

**Кубические координаты** имеют три оси, в отличие от сеток квадратов. Основным законом этой сетки является уравнение вида:

Каждое направление сетки шестигранников становится результатом сложения двух других координат клетки, умноженных на минус единицу. Это зачастую применяется во время определения и поиска соседних клеток.

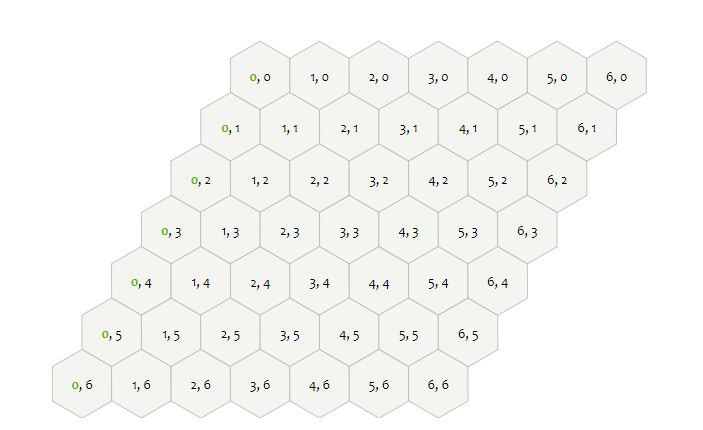
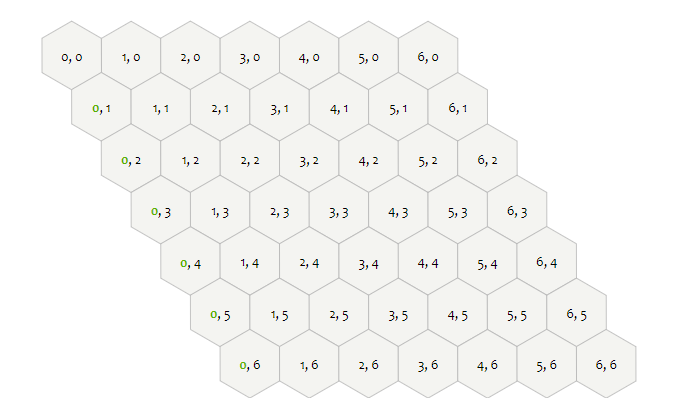


*Рисунок 14 Кубические и осевые координаты.*

Третьим видом координат являются **осевые координаты**, которые также называются "наклонные" или "скошенные". Они работают по тому же принципу, что и кубические за тем исключением, что мы не храним координату s. В соответствии с уравнением выше мы всегда можем посчитать s как:

Кубические и осевые координаты позволяют нам производить математические вычисления, в отличие от координат сдвига. Поэтому реализация алгоритмов упрощается в кубических или осевых координатах.

Для данного проекта и дальнейшего удобства в расчетах было решено использовать осевые координаты с измененным наклоном одной из осей. Таким образом не происходит увеличения координат через клетку, как на рисунках выше. То, как сетка выглядела до и после наклона оси, изображено ниже.



*Рисунок 15 До и после наклона сетки.*

Итоговая сетка представлена в виде массива типов клеток. Для произведения вычислений были созданы функции для получения координаты по индексу в массиве и наоборот.

*Листинг 1 Функции перевода координат в индекс массива и обратно.*

FVector2D ATileMap::IndexToCoord(int32 index)

{

return FVector2D(index/worldSize, index%worldSize);

}

int32 ATileMap::CoordToIndex(FVector2D coord)

{

return (coord.X\*worldSize+coord.Y);

}

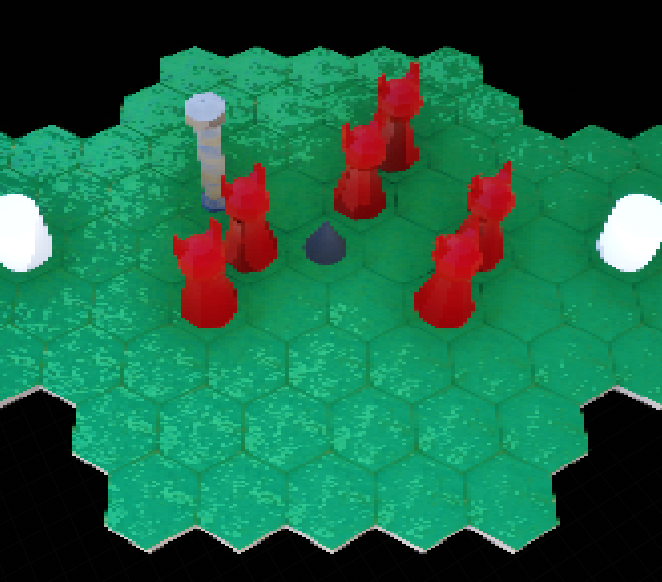
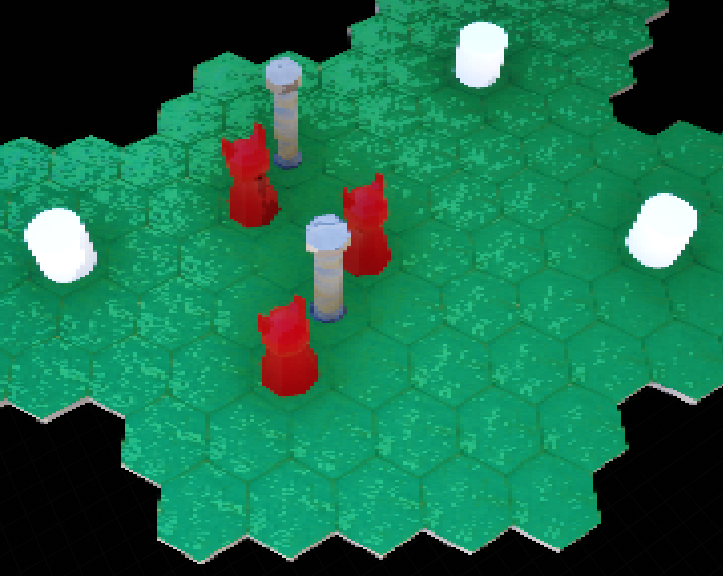
## **Создание комнат**

При первичном создании комнаты системе необходимо знать ее размер и координаты точки начала построения (верхний левый угол комнаты), после чего система считает количество рядов, заранее определяет координаты углов комнат (для дальнейшего удобства при создании коридоров). Полная функция представлена в Приложении В. После определения всех переменных мы с помощью вложенного цикла по рядам задаем клеткам тип.

*Листинг 2 Циклы первоначальной расстановки клеток в комнатах.*

for(int32 j = start; j < end; j++)  
{  
 tileIndex = worldSize\*i+j;  
 if (TileTypesTempBackup[tileIndex]!=ETT\_None&&TileTypesTempBackup[tileIndex]!=ETT\_Wall)  
 {  
 success=false;  
 break;  
 }  
 if((j==start||j==end-1)||(i==startX||i==startX+size\*2-2))  
 {  
 TileTypesTemp[tileIndex]=ETT\_Wall;  
 }  
 else if((j>start&&j<end-1)&&(i>startX&&i<startX+size\*2-2))  
 {  
 TileTypesTemp[tileIndex]=ETT\_Room;  
 if((j>start+1&&j<end-2)&&(i>startX+1&&i<startX+size\*2-3)) TempRoom.WayTilesArray.Add(tileIndex);  
 }  
  
 if (size==minRoomSizeDefault) minRoomSize++;  
 if (!success) break;  
}

После первичного определения границ комнаты, функция, в зависимости от типа создаваемой комнаты, задаются различные блоки наполнения. Таким образом, в обычной комнате расположены несколько врагов, для которых мы заранее определяем верхний и нижний предел. Однако в комнате с ключом система расставляет врагов на несколько единиц больше.



*Рисунок 16 Обычная комната и комната с ключом (в виде конуса).*

Также для некоторых типов комнат, таких как ранее упомянутая комната с ключом, задаются специальные параметры расстановки. В данном случае, количество таких комнат зависит от количества комнат с сундуком. В начале цикла генерации комнат-коридоров определяется количество комнат типа "Сундук" и такое же значение присваивается переменной, отвечающей за количество комнат-"ключей".

*Листинг 3 Код наполнения* комнаты на примере комнаты с ключом

**case ERT\_KeyRoom:**  
 selIndex = TempRoom.WayTilesArray[RandomOdd(0,TempRoom.WayTilesArray.Num()-1)];  
 TileTypesTemp[selIndex]=ETT\_Key;  
 TempRoom.WayTilesArray.Remove(selIndex);  
 for (int32 i = 0; i < TempRoom.Enemies; i++)  
 {  
 selIndex = TempRoom.WayTilesArray[RandomOdd(0,TempRoom.WayTilesArray.Num()-1)];  
 TileTypesTemp[selIndex] = ETT\_EnemySpawn;  
 TempRoom.WayTilesArray.Remove(selIndex);  
 }  
 keyCount++;  
 CurRoomType = ERT\_Base;  
 break;

## **Создание коридоров**

После успешного создания и заполнения комнаты система выбирает случайный из доступных углов для дальнейшего прокладывания коридора к следующей комнате. При создании коридора учитываются такие параметры как:

* Размер предыдущей комнаты
* Наличие ветвления
* Шанс ветвления
* Направление коридора

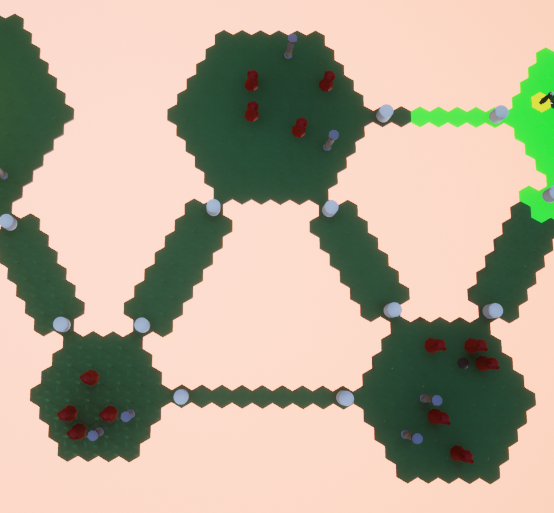


Рисунок Различия в длинах коридоров между комнатами размеров 4, 5 и 6

Длина коридоров, хоть и варьируется, но, если считать расстояние между центрами соединенных комнат, оно всегда будет одинаковым. Различия в длине коридоров объясняется различиями в размерах комнат. Также, прежде чем начать создавать коридор, системе необходимо знать, с каких сторон от клетки начала построения находятся стены. Готовые коридоры замыкаются клетками-дверями.

Кроме основных, широких коридоров, существуют дополнительные. Этот вид коридоров создается по завершении основного цикла генерации и учитывает только точки начала и конца маршрута, а также направление. Определяется оно в функции FindOptionalCorridors(), которая сравнивает все комнаты на предмет параллельности одной из осей сетки и наличия дверей в необходимых для построения углах.

## **Поиск маршрута**

Поиск пути между двумя клетками осуществляется с помощью алгоритма A\*. Так как на данный момент в системе клеток не предусмотрена система стоимости шага, было принято решение модифицировать алгоритм поиска в ширину, а не алгоритм Дейкстры, так как на текущий момент разницы между этими подходами нет.



Рисунок Поиск пути между двумя точками.

Приоритезация желаемой клетки производится путем сортировки массива доступных клеток по возрастанию значений параметров F и G, отвечающих за суммарное расстояние между клетками и расстояние между текущей и исходной клеткой соответственно. После сортировки происходит выбор первого элемента массива и его удаление, что позволяет избежать повторного выбора клетки.

После нахождения желаемой клетки происходит дополнительный цикл проверок, позволяющий убрать лишние клетки из маршрута, если таковые имеются.

Листинг Дополнительная проверка после нахождения целевой клетки

Path.Add(CurIndex);  
for (int32 i = FindTileByIndex(B)->G-1; i >= 0; i--){  
 for (FVector2D dir : NeighsIndexes) {  
 int32 adTile = CoordToIndex(IndexToCoord(CurIndex) + dir);  
 const ATileBase\* CheckTile = FindTileByIndex(adTile);  
 if (CheckTile->G == i && closedPath.Contains(adTile)){  
 CurIndex = adTile;  
 Path.Add(CurIndex);  
 break; } } }

## **Поиск клеток в радиусе**

Данный поиск клеток в своей основе имеет обычный поиск в ширину и применяется для выделения клеток в определенной области. На данный момент существует две вариации данного функционала: FindTilesReachable() и FindTilesInRange(). Отличаются данные функции тем, что FindTilesReachable производит дополнительную проверку клетки на доступность.

Листинг Часть функции FindTilesInRange()

for (FVector2D N : NeighsIndexes) {

int32 curInd = CoordToIndex(IndexToCoord(i)+N);

ATileBase\* CheckTile = FindTileByIndex(curInd);

if (CheckTile && !Visited.Contains(curInd)

&& CheckTile->TilesStruct.Available) {

CheckTile->Distance=k;

CurRun.Add(curInd);

Visited.Add(curInd);

Result.Add(CheckTile);

}

}

В проекте эти функции применяются на данный момент только для выделения доступных для передвижения клеток, однако в дальнейшем функцию FindTilesInRange() возможно будет применять в выделении зоны поражения специфического оружия или приема.



Рисунок Зона передвижения радиусом 6 клеток.

## **Обработка пользовательского ввода**

Пользовательские сигналы о взаимодействии обрабатывает виртуальная функция Char\_Interaction(), в ней определяется реакция клетки. Рассмотрим на примере тайлов с двумя различными реакциями на взаимодействие:

* Клетка-дверь изначально доступна для интеракции, однако после подачи сигнала пользователем, дверь открывается и становится доступна только для движения
* Клетка-сундук никогда не доступна для движения, но всегда доступна для взаимодействия, которое подразумевает открытие дополнительного пользовательского интерфейса, отражающего содержимое карманов управляемого персонажа и контейнера на клетке.

Реакции на пользовательские сигналы описываются индивидуально и могут подразумевать взаимодействие с другими клетками или объектами на уровне.

## **2.2 Создание новых типов клеток**

Кроме разработки самого алгоритма генерации, появляется необходимость формализовать правила создания новых типов клеток для дальнейшего появления их на уровне и пользовательского взаимодействия с ними.

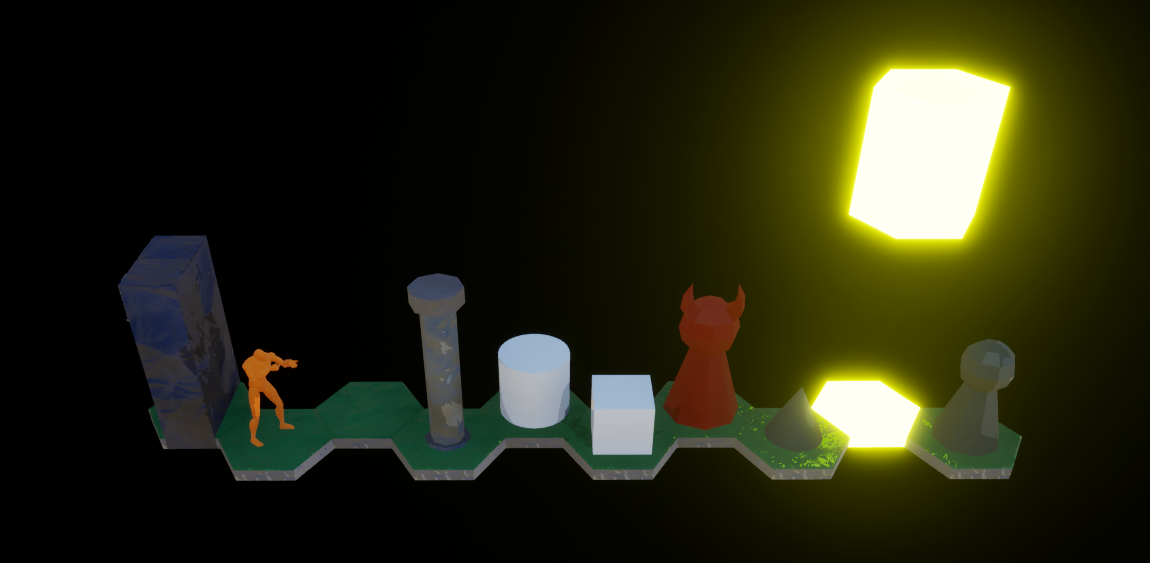


Рисунок 20 Все типы клеток на данный момент

### **2.2.1 Класс TileBase и его производные**

Для начала нужно определиться, что входит в структуру клетки, с какими параметрами и методами взаимодействует игрок. Таким образом, основными параметрами клетки являются:

* Координаты клетки
* Параметр доступности клетки
* Параметр доступности для взаимодействия
* Тип клетки
* Массив ссылок на соседствующие клетки
* Массив содержимого клетки, необходимый для системы инвентаря и сбора предметов.

Ниже приведен блок кода, который содержит в себе обозначения всех компонентов в классе TileBase.

И если данные о координатах клетка любого типа получает одинаково, то остальные параметры управляющий системой может настроить в наследуемых классах. Типы клеток присваиваются и создаются в соответствии с наследуемым классом.

UPROPERTY(BlueprintReadWrite, EditAnywhere)  
FTilesStruct TilesStruct = FTilesStruct();  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Components")  
UStaticMeshComponent\* Mesh;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Components")  
USphereComponent\* Sphere;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Components")  
USphereComponent\* SphereToCheckNeighs;  
  
UPROPERTY(BlueprintReadWrite,EditAnywhere, Category="Components")  
USceneComponent\* Scene;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Components")  
UStaticMeshComponent\* AdditionalMesh;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Tiles")  
ACharBase\* CharacterOnTile;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Tiles")  
TArray<ATileBase\*> Neighs;  
  
UPROPERTY(BlueprintReadWrite,EditAnywhere,Category="TileData")  
TArray<TEnumAsByte<ETileType>> TypeNeighs;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="Tiles")  
TArray<ATileBase\*> AvailableNeighs;  
  
UPROPERTY(BlueprintReadWrite, EditAnywhere, Category="ItemContainer")  
TArray<FItemStruct> Container;

Листинг 6 Структура клетки

Подразумевается, что клетки, наследуемые от класса TileContainer изначально имеют заполненный массив содержимого клетки, а также имеют параметр доступности для взаимодействия равный true.

## **2.3 Создание новых типов комнат**

Вместе с необходимостью создавать различные типы клеток, появляется и необходимость создавать различные типы комнат, что подразумевает различные алгоритмы наполнения и определение параметров для дальнейшего создания подземелья.

Таким образом, комнаты регулируются в следующих аспектах:

* Тип комнаты
* Размер комнаты
* Возможность создания дальнейшего подземелья из этой комнаты и ветвления из нее
* Алгоритм наполнения комнаты в соответствии с типом.

### **2.3.1 Определение параметров комнаты**

Параметры комнаты назначаются в цикле создания комнат и коридоров, проиллюстрированном в [приложении Б](#_Приложение_Б). В ходе цикла комнате присваивается случайный размер X\*X в предварительно заданных пределах и тип комнаты, исходя из количества комнат различных типов на уровне.

### **2.3.2 Функция GenerateRoom**

Данная функция находится в классе генератора уровней TileMap, принимает в себя номер комнаты, ее размер и тип, а возвращает булевое значение, отвечающее за успех создания комнаты на сетке.

В конце данной функции предполагается switch-case проверка на тип создаваемой комнаты для дальнейшего наполнения. Рассмотрим вариации логики на нескольких примерах.

* Тип chestRoom подразумевает выбор случайной клетки не у стены и последующую смену типа выбранной клетки на TileChest, описанный ранее.



Рисунок 21 Комната с сундуком.

* Тип enemyRoom запускает цикл на N итераций, где N – количество врагов в комнате. В теле цикла также случайным образом выбирается клетка не у стены и ее тип изменяется на TileEnemy.



Рисунок 22 Комната с врагами.

## **Выводы ко 2 главе**

# **3 ТЕХНОЛОГИЧЕСКАЯ ЧАСТЬ**

## **3.1 Описание реализации**

Для разработки программы использовался объектно-ориентированный язык C++. Выбор языка обусловлен требованиями выбранного игрового движка Unreal Engine.

Основным источником о языке программирования С++ послужили:

* Книга «Язык программирования C++»
* Официальная документация Unreal Engine
* Статья о сетках шестигранников «Hexagonal Grids»

Сайты использовались из-за удобства поиска необходимой информации в данный момент разработки, а также наличия тематических примеров.

Программа-генератор состоит из двух классов и двух дополнительных заголовочных файлов.

## **3.2 Классы и методы**

### **3.2.1 Класс TileBase**

Данный класс описывает объекты типа «Клетка», который составляет основную геометрию на генерируемом уровне. В качестве параметров в классе хранятся:

* Структура TilesStuct типа FTilesStruct, содержащий основные данные о координатах, типе и индексе тайла.
* Массив Neighs типа ATileBase, содержащий ссылки на соседей данного тайла.
* Целочисленные значения Distance, G, H, F, применяемые при поиске пути от клетки до клетки.

### **3.2.2 Методы класса TileBase**

* CheckNeighs(); и CheckAvailableNeighs(); содержат подобную логику, заполняющую массив Neighs, однако вторая функция проверяет соседнюю клетку на возможность на нее наступить.
* void ATileBase::CheckAvailableNeighs()  
  {  
   for (int i = 0; i < Neighs.Num();i++)  
   {  
   if(Neighs[i]->TilesStruct.Available) AvailableNeighs.Add(Neighs[i]);  
   }  
  }

### **3.2.3 Класс TileMap**

Является основным классом системы. Этот класс содержит алгоритм расстановки объектов типа TileBase внутри двумерной сетки по правилам, изложенным во второй главе. Содержит в себе параметры для настройки карты, такие как:

* Целочисленные значения minRoomSize, maxRoomSize, задающие соответственно минимальный и максимальный размер генерируемых комнат.
* Целочисленное значение seed, содержащее ключ к генерации подземелья
* Целочисленное значение worldSize, задающее размер двумерного массива
* Целочисленные значения spawnAmount, despawnAmount, chestAmount, keyAmount, shopAmount, задающие минимальное количество комнат определенных типов.
* Целочисленное значение roomAmount, задающее необходимое количество комнат в принципе
* Булевое значение bBranching, задающее возможность ветвления лабиринта
* Числовые значения с плавающей точкой branchChance и optionalChance, задающие вероятность ветвления и построения дополнительного коридора соответственно
* Также содержится массив, содержащий ссылки на все тайлы.

### **3.2.4 Методы класса TileMap**

Методы можно разделить на методы генерации и методы математики шестигранников.

**Методы генерации:**

* DunGenMain(); содержит последовательность инструкций для генерации
* void ATileMap::DunGenMain()  
  {  
   RandStreamGen();  
   prevLocation = FVector2D(worldSize/2);  
   GenerateAllIndexes();  
   TempArraysSetup();  
   RoomCorCycle();  
   FinishRoomCorners();  
   FindOptionalCorridors();  
   FinishRoomCorners();  
   SpawnEveryIndex();  
  }
* RandStreamGen(); позволяет создать случайный ключ генерации, вызывается в начале создания, если значение seed равно нулю
* void ATileMap::RandStreamGen()  
  {  
   if (seed==0)  
   {  
   RandStream.GenerateNewSeed();  
   seed=RandStream.GetCurrentSeed();  
   }  
   else RandStream = FRandomStream(seed);  
  }
* GenerateRoom(); создает комнату, параметрами функции являются номер текущей комнаты, размер комнаты и ее тип. В соответствии с типом комнаты создается наполнение комнат. В конце создания комнаты определяется дальнейшее направление, в котором будет создаваться коридор.
* GenerateCorridor(); создает коридор между комнатами А и Б.
* RoomCorCycle(); Рекурсивный метод, проверяющий наличие необходимых комнат на уровне и создающий комнату и коридор если комнат достаточно и они не пересекаются.
* FinishRoomCorners(); Проверяет наличие коридора у угла в комнате.
* void ATileMap::FinishRoomCorners()  
  {  
   for (FRoomStruct Room : TileRooms)  
   {  
   for (int32 i = 0; i < Room.cornerInd.Num();i++)  
   {  
   int32 Ind\_To\_Check=CoordToIndex(IndexToCoord(Room.cornerInd[i])+NeighsIndexes[i]);  
   if (TileTypes[Ind\_To\_Check]==ETT\_Path) TileTypes[Room.cornerInd[i]]=ETT\_Door;  
   }  
   }  
  }
* FindOptionalCorridors(); Проверяет наличие комнат, лежащих на одной оси, между которыми нет коридоров, после чего с определенной вероятностью он создается.
* void ATileMap::FindOptionalCorridors()  
  {  
   TempArraysSetup();  
   for (int32 i = 0; i < TileRooms.Num()-2; i++)  
   {  
   FRoomStruct A = TileRooms[i];  
   for (int32 j = i+1; j < TileRooms.Num()-1; j++)  
   {   
   FRoomStruct B = TileRooms[j];  
    
   FVector2D res = (B.start+(B.size-1))-(A.start+(A.size-1));  
     
   *//Directions 0 and 3 WORKS CORRECTLY* if (((A.start+(A.size-1)).X - (A.start+(A.size-1)).Y == (B.start+(B.size-1)).X - (B.start+(B.size-1)).Y))  
   {  
   int32 len = res.Size();  
     
   if (res < FVector2D().ZeroVector && TileTypes[A.cornerInd[0]]!=ETT\_Door && RandStream.GetFraction()<optionalChance)  
   {  
   DoOptionalCorridors(A.cornerInd[0],B.cornerInd[3],0);  
   }  
   if (res > FVector2D().ZeroVector && TileTypes[A.cornerInd[3]]!=ETT\_Door && RandStream.GetFraction()<optionalChance)  
   {  
   DoOptionalCorridors(A.cornerInd[3],B.cornerInd[0],3);  
   }  
   }  
   *//Directions 2 and 5 WORKS CORRECTLY* else if ((A.start+(A.size-1)).X == (B.start+(B.size-1)).X)  
   {  
   int32 len = res.Size();  
    
   if (res.Y > 0 && TileTypes[A.cornerInd[2]]!=ETT\_Door && RandStream.GetFraction()<optionalChance)  
   {  
   DoOptionalCorridors(A.cornerInd[2],B.cornerInd[5],2);  
   }  
   if (res.Y < 0 && TileTypes[A.cornerInd[5]]!=ETT\_Door && RandStream.GetFraction()<optionalChance)  
   {  
   DoOptionalCorridors(A.cornerInd[5],B.cornerInd[2],5);  
   }  
   }  
   *//Directions 1 and 4 WORKS CORRECTLY* else if ((A.start+(A.size-1)).Y == (B.start+(B.size-1)).Y)  
   {  
   int32 len = res.Size();  
    
   if (res.X < 0 && TileTypes[A.cornerInd[1]]!=ETT\_Door && RandStream.GetFraction()<optionalChance)  
   {  
   DoOptionalCorridors(A.cornerInd[1],B.cornerInd[4],1);  
   }  
   if (res.X > 0 && TileTypes[A.cornerInd[4]]!=ETT\_Door && RandStream.GetFraction()<optionalChance)  
   {  
   DoOptionalCorridors(A.cornerInd[4],B.cornerInd[1],4);  
   }  
   }  
   }  
   }  
  }
* DoOptionalCorridors(); Вызывается в функции, указанной выше, создает коридор между двумя углами комнат.
* void ATileMap::DoOptionalCorridors(int32 start, int32 end, int32 dir)  
  {  
   TArray<int32> walls;  
   for(int32 i = 1; i <= 2; i++)  
   {  
   walls.Add(CoordToIndex(IndexToCoord(start)+NeighsIndexes[(dir+2\*i)%6]));  
   }  
   int32 length = 0;  
   const FVector2D A = IndexToCoord(start);  
   const FVector2D B = IndexToCoord(end);  
     
   if ((dir % 3) == 0 ) length = abs((B.X-A.X));  
   else if ((dir % 3) == 1 ) length = abs (B.X-A.X);  
   else if ((dir % 3) == 2 ) length = abs (B.Y-A.Y);  
    
   *//UE\_LOG(LogTemp,Display,TEXT("Distance is %i and direction is %i"), length, dir);* if (length < maxRoomSize+1)  
   {  
   for (int32 i = 1; i <= length; i++)  
   {  
   int32 curIndex = CoordToIndex(A+NeighsIndexes[dir]\*i);  
   for(int32 j = 0; j<=1; j++) walls[j] = CoordToIndex(IndexToCoord(walls[j])+NeighsIndexes[dir]);  
   if (i<=length)  
   {  
   TileTypesTemp[curIndex]=ETT\_Path;  
   }  
   if (i<=length+1) for(int32 idWall : walls) TileTypesTemp[idWall]=ETT\_Wall;  
   }  
   TileTypes = TileTypesTemp;  
   }  
  }
* SpawnEveryIndex(); Создает и определяет положение тайла внутри игрового мира в зависимости от типа тайла.

**Методы математики шестигранников:**

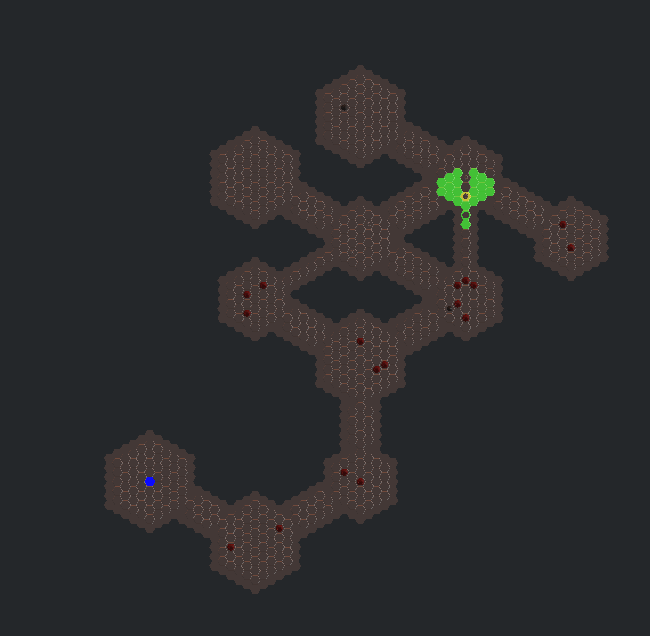
* GetNeighs(); возвращает массив типов тайлов по определенному индексу.
* TArray<TEnumAsByte<ETileType>> ATileMap::GetNeighs(int32 index)  
  {  
   FVector2D start = IndexToCoord(index);  
   TArray<TEnumAsByte<ETileType>> NeighsArray;  
   for (FVector2D dir : NeighsIndexes)  
   {  
   NeighsArray.Add(TileTypes[CoordToIndex(start+dir)]);  
   }  
   return NeighsArray;  
  }
* IndexToCoord(); переводит индекс в двумерный вектор координат.
* FVector2D ATileMap::IndexToCoord(int32 index)  
  {  
   return FVector2D(index/worldSize, index%worldSize);  
  }
* CoordToIndex(); переводит двумерный вектор в индекс тайла.
* int32 ATileMap::CoordToIndex(FVector2D coord)  
  {  
   return (coord.X\*worldSize+coord.Y);  
  }
* FindTileByIndex(); Возвращает ссылку на объект тайла исходя из его индекса.

ATileBase\* ATileMap::FindTileByIndex(int32 index)  
{  
 ATileBase\* IndexedTile = nullptr;  
 *//UE\_LOG(LogTemp,Display,TEXT("Looking for tile with index %i"), index);* for (ATileBase\* Tile : Tiles)  
 {  
 if (Tile->TilesStruct.aind == index)  
 {  
 IndexedTile = Tile;  
 break;  
 }  
 }  
 return IndexedTile;  
}

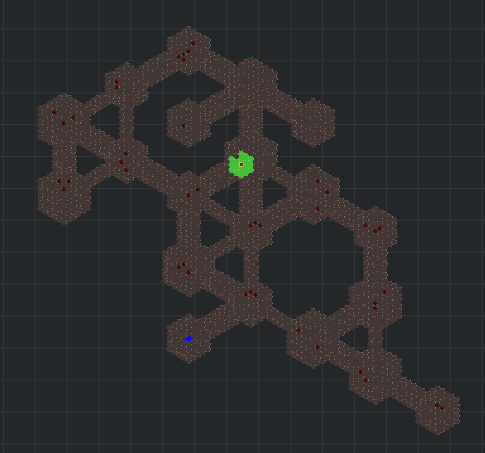
## **3.3 Результаты выполнения программы**

Результатом проделанной работы является программа-генератор, способная создавать ветвистый лабиринт из комнат, который в дальнейшем можно использовать как уровень в компьютерной игре.

На изображении ниже изображен пример карты, построенный на сетке шестиугольников 120\*120 и количеством комнат равном 11.



На рисунке ниже пример карты, построенной на сетке 250\*250 клеток и количеством комнат равном 20.



## **Пример интеграции**

### **Работа генератора**

### **User Interface**

### **Управление**

## **Выводы к 3 главе**

# **Заключение**

В курсовой работе была исследована техника процедурной генерации уровней. Были проанализированы различные методы генерации и определены основные структуры данных для генератора подземелий.

Превосходство выбранной структуры как более гибкой, универсальной и расширяемой, доказано. Также подробно описаны алгоритмы генерации, используемые для создания игровых карт.

Приведен список классов и методов, отвечающий за реализацию данных алгоритмов.

Приведены примеры карт, получаемых в результате выполнения программы.

В результате выполнения курсового проекта была разработана программа-генератор игровых подземелий на сетке шестигранников, в основе которой лежит гибкая структура данных, обеспечивающая возможность дальнейшего ее расширения и интегрирования в компьютерную игру.

В дальнейшем планируется добавить возможность управления внутриигровыми юнитами на поле, а также разработать искусственный интеллект, отвечающий за поведение существ, населяющих подземелья.

Также планируется создание интерфейса найма отрядов, создание уровней с боссами этапов, а также развитие более тонкой игровой логики.

# **Список литературы**

1. David S. Ebert, Ken Perlin, Steven Woorley, Darwyn Peachey, F.Kenton Musgrave «Texturing & Modeling: A Procedural Approach»
2. Бьерн Страуструп. «Язык программирования С++»
3. Таня X. Шорт, Тарн Адамс «Процедурная генерация в гейм-дизайне»
4. Меженин Михаил Григорьевич «Обзор систем процедурной генерации игр»
5. <https://www.redblobgames.com/grids/hexagons/> - статья, посвященная системам сеток шестигранников
6. <http://pcg.wikidot.com/pcg-algorithm:map-generation> – краткий обзор алгоритмов ПГК
7. <http://www.roguebasin.com/index.php/Grid_Based_Dungeon_Generator> - статья, посвященная генерации подземелий на основе сетки
8. <http://www.roguebasin.com/index.php/Cellular_Automata_Method_for_Generating_Random_Cave-Like_Levels> - статья, посвященная генерации подземелий на основе клеточных автоматов.

**ПРИЛОЖЕНИЕ В**

GenerateRoom(int32 roomnum, int32 size, TEnumAsByte<ERoomTypes> RoomType)

bool ATileMap::GenerateRoom(int32 roomnum, int32 size, TEnumAsByte<ERoomTypes> RoomType)  
{  
  
#pragma region VarSetup  
   
 *//TempRoomSetup* TempRoom.cornerInd.Append(cornersTemp);  
   
 for (int32 i = startX; i < (startX+(size\*2-1)); i++)  
 {  
 for(int32 j = start; j < end; j++)  
 {  
 tileIndex = worldSize\*i+j;  
 if (TileTypesTempBackup[tileIndex]!=ETT\_None&&TileTypesTempBackup[tileIndex]!=ETT\_Wall)  
 {  
 success=false;  
 break;  
 }  
 if((j==start||j==end-1)||(i==startX||i==startX+size\*2-2))  
 {  
 TileTypesTemp[tileIndex]=ETT\_Wall;  
 }  
 else if((j>start&&j<end-1)&&(i>startX&&i<startX+size\*2-2))  
 {  
 TileTypesTemp[tileIndex]=ETT\_Room;  
 if((j>start+1&&j<end-2)&&(i>startX+1&&i<startX+size\*2-3)) TempRoom.WayTilesArray.Add(tileIndex);  
 }  
  
 if (size==minRoomSizeDefault) minRoomSize++;  
 if (!success) break;  
 }  
 if (!success) break;  
 if(i<startX+size-1) end++;  
 else start++;  
 }  
 if(!success)  
 {  
 TileTypesTemp.Empty();  
 TileTypesTemp.Append(TileTypesTempBackup);  
 }  
 else  
 {  
#pragma region FillRooms   
 return success;  
}