ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Департамент программной инженерии

УТВЕРЖДАЮ
Академический руководитель
образовательной программы
«Программная инженерия»,
профессор департамента программной
инженерии, канд. техн. наук

_____ В.В. Шилов
«___» _____ 2018 г

Выпускная квалификационная работа

на тему **«Игра “Dark Lab”»**

по направлению подготовки 09.03.04 «Программная инженерия»

Научный руководитель:
доцент департамента
программной инженерии факультета
компьютерных наук,
канд. техн. наук

Ахметсафина Р.З.

_____
Подпись, Дата

Выполнил:
студент группы БПИ141
4 курса бакалавриата
образовательной программы
«Программная инженерия»

Фазли Д.М.К.

_____
Подпись, Дата

Москва 2018

Реферат

Генерация карт в видеоиграх давно стала мощным инструментом в руках разработчиков. Еще со времен *Beneath Apple Manor* эта техника эффективно использовалась для создания неповторимых локаций для исследования игроками без необходимости в тщательном дизайне каждого отдельного уровня.

Однако обычно в играх генерация карты происходит либо еще перед началом нового уровня, либо уже в процессе игры, но на большом расстоянии от игрока. Разработчики не реализует системы, в которых карта менялась бы постоянно прямо вокруг игроков, но оставляла бы процесс этих изменений незаметным для них.

В данной работе представлена игра *Dark Lab* в жанре survival horror, в которой центральное место занимает подобная механика. В игре персонаж просыпается в темном помещении, исследует его, использует различные предметы и ищет спасительный выход, в то время как за ним охотится уязвимый к свету монстр – «тьма». Но главное то, что все это время, куда бы игрок ни шел, карта постоянно меняется в темноте буквально на краю поля зрения игрока, что создает ощущения тревоги и потерянности, которые хороши для этого жанра игр.

В работе подробно разобран собственный разработанный алгоритм динамической случайной генерации карты, проанализированы другие существующие алгоритмы, представлены детали архитектуры приложения и иные особенности программной реализации, описаны полученные результаты. Также представлен обзор других игр того же жанра.

Работа содержит 67 страниц, 3 главы, 82 изображения, 57 источников, 4 приложения.

Ключевые слова*: разработка игр, survival horror, генерация карт, Unreal Engine 4, C++, ПК.*

Abstract

Map generation in video games has long since become a powerful instrument in the hand of developers. Ever since *Beneath Apple Manor* this technique has been used effectively to create unique locations for players to explore without the necessity for thorough design of each individual level.

However, map generation in games usually happens either before the start of a new level or during the game, but a great distance away from the player. Developers do not implement systems, that could have map continuously morphing right around the players while leaving the process of these changes hidden from them.

This paper presents the game *Dark Lab* in the survival horror genre, in which the central place is given to a similar mechanic. In this game the character wakes up in a dark facility, which he then explores using various objects and looking for a saving exit, while a vulnerable to light monster – the “darkness” – is trying to hunt him down. But, most importantly, during all this time wherever the player goes, the map continuously changes in the darkness on the edge of the player’s field of view, which creates the senses of anxiety and disorientation that are good for this genre of games.

The developed dynamic random map generation algorithm is explained in this paper at length, some other existing algorithms are analyzed, application’s architecture’s details and other peculiarities of program implementation are presented, the results are described. Also, other games of the same genre are reviewed.

This paper contains 67 pages, 3 chapters, 82 illustrations, 57 bibliography items, 4 appendices.

Keywords: ***game development, survival horror, map generation, Unreal Engine 4, C++, PC.***

Содержание

Определения и термины

Survival horror – это жанр видеоигр, в которых протагонист должен спастись из пугающей и/или жестокой среды. Игры жанра стремятся создать атмосферу похожую на фильмы ужасов [42].

Блупринты (blueprints) – это продвинутая технология визуального скриптинга, встроенная в *Unreal Engine* и используемая для прототипирования или для полноценной разработки [6].

Геймплей (gameplay) – это игровой процесс, набор игровых правил, целей и сложностей [4], компонент игры, завязанный на взаимодействии игры и игрока.

Динамическая генерация – процедурная генерация, которая происходит во время игры и на которую могут повлиять действия игрока в отличие от статической генерации, запускаемой перед загрузкой уровня [13].

Игровая механика – это абстракция, состоящая из методов и правил, созданных для того, чтобы игрок с ними взаимодействовал и являющихся частью геймплея игры [4].

Игровой движок (game engine) – это коллекция библиотек и инструментов, собранных вместе для создания игр, также обладающая графом объектов сцены и редактором мира/уровня [14].

Инди-игра (indie (independent) game) – это независимо разработанная игра, обычно созданная без финансовой поддержки издателя. Инди игры часто фокусируются на инновационных идеях.

Кастомизация – это индивидуализация, настройка чего-то под конкретные нужды.

Контроллер (controller) – управляющий класс, контролирующий другой класс (в *Unreal Engine*).

Крафт (craft) – это механика создания внутриигровых предметов из добываемых в игре ресурсов или из других предметов. Обычно осуществляется посредством комбинации.

Кулдаун (cooldown) – это время ожидание после использования способности в игре, во время которого эту способность невозможно применить повторно [7].

Лаг (lag) – это задержка между двумя событиями. В играх лагом часто называют задержку между вводом пользователя и изменением картинки (может возникать по разным причинам).

Подземелье – в процедурной генерации это структура карты состоящая из набора комнат (зачастую прямоугольных) и соединяющих их коридоров.

Процедурная генерация – это процесс создания контента с использованием какого-то алгоритма. При вызове одного и того же алгоритма с одинаковыми параметрами выдается одинаковый результат [21].

Пулинг (pooling, object pool) – это паттерн проектирование, направленный на переиспользование часто создаваемых и удаляемых объектов с целью повышения производительности и избегания фрагментации памяти [26].

Рейкаст (raycast) – это общий метод и единичное применение проверки пересечении выпускаемого из какой-нибудь точки пространства луча с объектами игрового мира [51].

Респаун (respawn) – это повторное создание объекта в игровом мире. Обычно после уничтожения или схожего эффекта.

Саунд-дизайн (sound design) – это искусство и процесс создания различных звуковых элементов. Используется в кинематографе, создании видеоигр, театре, радио и т.д.

Случайная генерация – это процедурная генерация с элементом случайности. Например, когда алгоритмам процедурной генерации на вход поступают случайные элементы. Обычно, когда говорят, что что-то сгенерировано процедурно, имеется в виду именно случайная генерация [21].

Спаун (spawn) – это создание объекта в игровом мире.

Фреймворк (framework) – это коллекция библиотек и инструментов, собранных вместе для решения какой-либо задачи [14].

Введение

К настоящему времени видеоигры стали частью повседневной жизни. То, что раньше было хобби для обособленного сообщества, стало обычным развлечением для мужчин и женщин всех возрастов [24]. Люди играют у себя дома, в игровых клубах, по дороге на работу, загородом и в принципе везде, где могут найти время и подходящие для игр устройства.

Говоря об устройствах, стоит признать, что вперед вышли игры на мобильных платформах, однако персональные компьютеры до сих пор занимают весомую нишу среди игровых платформ (28%), оставаясь в предпочтении многих геймеров, но, что самое главное, индустрия как целое продолжает развиваться, показав рост в 10.7% за 2017-й год [45]. Кроме того, постоянно проводятся игровые конференции, и разработка игра приводит к появлению новых алгоритмов и технологий. Однако, несмотря на широкое распространение видеоигр, ситуация среди разных жанров кардинально отличается. Мы остановимся более подробно на одном из них – survival horror.

Survival horror игры стремятся испугать игрока, окунуть его в атмосферу отчаяния и тревоги, ведь страх – очень сильная эмоция, понятная любому человеку. «Survival» составляющая таких игр обычно заключается в ограниченности ресурсов и возможностей игрока, скованности его действий и стремлении спастись от ужаса. В основном сюжеты таких игр строятся вокруг попыток игрока сбежать от различных монстров и других противников, иногда эти сюжеты приводят к множественным концовкам. Идеологически survival horror игры прежде всего ставят перед собой задачу довести игроков до состояния страха. К этому осторожно подходят, выстраивая пугающую атмосферу игры, прежде чем показать монстров и другие главные причины испуга.

Этот жанр отличается преданной, но ограниченной аудиторией: она никогда не достигнет размахов современных игр-блокбастеров. В связи с этим в прошлом horror игры начали заполнять элементами из других игр, которые полностью разрушали создаваемую иллюзию, противореча концепциям жанра, но этот тренд в итоге спал. При этом survival horror игры требуют эксперимента, новых решений. Нежелание идти на риски с новаторскими идеями, а также ограниченность целевой аудитории останавливают издателей от вложений в игры этого жанра. Однако именно это создало нишу для инди-игр. **Инди-играми** называют проекты, разработанные независимыми группами без финансовой поддержки издателей (обычно). У таких проектов больше возможностей создавать более короткие, но сфокусированные на чем-то игры, большее пространство для экспериментов и меньшее финансовое давление [44]. Другими плюсами именно для инди сцены стала доступность мощных средств разработки и краудфандинговые платформы [1].

Инди игры в целом выпускаются все в большем и большем количестве [25]. Среди них немало и survival horror игр. Только за 2017-й год усилиями независимых разработчиков были выпущены такие успешные проекты как *Observer* [27], *Detention* [10] и *Darkwood* [9], что показывает потенциал рынка инди-игр этого жанра.

Данная работа посвящена разработке подобной игры для ПК – **Dark Lab**. Во многом она следует законам жанра: персонаж, контролируемый игроком, оказывается заперт в темном, пугающем помещении – лаборатории – и ищет способы выбраться оттуда. По сюжету персонаж не знает, где оказался и надеется найти какие-то подсказки и путь из лаборатории. Соответственно, нахождение выхода является главной целью игры и условием победы игрока. Это положительная концовка игры.

В процессе исследования лаборатории за персонажем охотится монстр – **«тьма»**. Временами она отступает и оставляет игрока в покое, но, если окажется, что тот близок к тому, чтобы покинуть лабораторию, «тьма» не прекратит свою охоту. Если «тьма» настигает игрока, тот погибает (считается «потерянным во тьме»), и игра завершается поражением игрока (отрицательная концовка).

Кроме следования канонам survival horror игра было необходимо провести эксперимент и реализовать идею, которая не присутствовала бы в других играх. В данном проекте этим экспериментом и основной особенностью игры является сама лаборатория: она постоянно изменяется в темноте и там, где раньше игрок натыкался на стену, он может позже обнаружить новый коридор, а где раньше была комната с выходом, может оказаться зал с «тьмой». Подобные пространственные изменения можно встретить в таких играх как *SCP: Containment Breach* [35] и *Stanley Parable* [39], но в обоих случаях они реализованы лишь как заскриптованные сцены, а не полноценные механики, проходящие стержнем через всю игру как в *Dark Lab*.

Среди других игровых механик – наборов игровых методов и правил – и прочих особенностей *Dark Lab* можно выделить: вид сверху; влияние света на «тьму» замедлением ее передвижения и даже временным испугом; способность «тьмы» со временем становиться невосприимчивой к свету; использование различных предметов, таких как осветительные приборы и карты от дверей; активация объектов в лаборатории вроде и вышеупомянутых дверей; звуки шагов и приближения «тьмы» и т.д.

Причины, по которым многие из этих и некоторые другие элементы были добавлены в *Dark Lab* описаны в первой главе.

Таким образом, **целью** данной работы является создание игры в реальном времени для ПК в жанре survival horror, в которой игрок пытается найти выход из помещения-лаборатории, в то время как за ним охотится монстр, а сама лаборатория постоянно меняется в темноте незаметно для игрока.

Для достижения данной цели были поставлены следующие **задачи**:

1. Разработать основной концепт, сценарий и правила игры;
2. Изучить и проанализировать существующие survival horror игры и выделить основные черты жанра;
3. Изучить и проанализировать существующие алгоритмы случайной генерации карт;
4. Разработать функциональные требования к программе;
5. Разработать структуры данных, используемые в программе;
6. Разработать собственный алгоритм случайной генерации для динамического изменения карты в процессе игры;
7. Выбрать средства и инструменты разработки;
8. Изучить и использовать в разработке *Unreal Engine 4* [48];
9. Разработать архитектуру программы;
10. Разработать программу для ПК с полным функционалом игры;
11. Провести тестирование программы.
12. Разработать техническую документацию.

Работа структурирована следующим образом: в первой главе приведен обзор игр жанра survival horror, а также кратко описаны некоторые существующие алгоритмы случайной генерации карт и причины, по которым они не подходят для проекта *Dark Lab* в чистом виде; во второй главе описан разработанный автором алгоритм генерации карты и применяемая для этого модель, а также логика поведения «тьмы»; в третьей главе изложены выбор средств разработки, технические подробности разработанной программы и особенности реализации; в заключении перечислены основные полученные результаты и пути дальнейшей работы; в приложениях А-Г приведена документация к программе, включающая техническое задание, программу и методику испытаний, руководство оператора, а также текст программы.

Глава 1. Исследование рынка survival horror игр и методов генерации карт

В этой главе приводится обзор существующих игр в жанре survival horror и из них выделен функционал, необходимый в *Dark Lab*. После этого показаны ограничения генерации карт, дан краткий анализ различных существующих алгоритмов генерации и освещены их недостатки.

1.1. Обзор survival horror игр

Как уже было отмечено, survival horror игры, несмотря на ограниченность аудитории, получили широкое распространение благодаря инди разработчикам. По статистике сервиса SteamSpy среди игр в Steam [40] – самой популярной платформе распространения игр – представлены сотни игр в жанре survival horror [43]. Естественно, сравнить *Dark Lab* с каждой из них в рамках данной работы не представляется возможным и, более того, подобный анализ нецелесообразен.

Среди жанра horror в целом, как в кино, так и в играх, распространен такой принцип как «less is more» или «чем меньше, тем больше», заключающийся в том, что игрок или зритель должен быть ограничен («less») как в информации, так и в возможностях, чтобы быть больше («more») подверженным страху [23]. Например, образ монстра представляется тем более пугающим, чем меньше его видит наблюдатель, пока его собственное воображение достраивает детали. Аналогичное верно и для игровых механик: чем больше дать игроку возможностей, тем меньше он будет испуган, отвлекаясь на ненужные элементы и теряя погружение [44]. Именно поэтому анализ и сравнение игр с точки зрения наличия тех или иных особенностей геймплея и составление построенной на этом таблицы достоинств и недостатков не имеет смысла. Однако можно рассмотреть различные успешные survival horror проекты и выделить из них аспекты, которые подходят к основной концепции *Dark Lab*.

1.1.1. Darkwood

Первым рассматриваемым проектом является игра 2017-го года *Darkwood* [9]. В игре главный герой исследует лес и различные локации в нем, общается с местными обитателями и прячется от монстров по ночам, стараясь выжить.

Среди особенностей *Darkwood* можно выделить:

- Множество локаций, предметов;
- Неповторимую атмосферу;
- Продвинутую систему крафта – создания предметов в игре из доступных ресурсов;
- Разветвленную сюжетную составляющую;
- Процедурную генерацию (создание алгоритмами) мира в каждой новой игре;

- Всевозможные способности персонажа.

Атмосфера как одно из основных достоинств встречается почти у всех успешных survival horror игр, потому как именно от нее зависит, насколько игра пугающая, поэтому на этом пункте останавливаться не нужно, его необходимо реализовать уже потому, что *Dark Lab* это survival horror игра. Обычно атмосфера создается за счет темных тонов палитры, особенностей графического стиля игры, общей обстановки, музыки и звуков (о чем подробнее дальше) и даже за счет пользовательского интерфейса (или его отсутствия).

Процедурная генерация карты также изначально входит в концепт *Dark Lab,* хоть и в другом виде. Все остальное из этого списка не было добавлено в *Dark Lab*.

Рисунок 1. Darkwood

Что же было взято из *Darkwood* так это **вид сверху**, как на рис.1. Дело в том, что обычно survival horror игры делают от первого или от третьего лица, то есть с видом глазами персонажа или у него из-за спины. Это сближает игрока с управляемым персонажем, позволяя добиться большего эмоционального погружения, а значит и большего испуга. Однако *Darkwood* доказывает, что игра может быть столь же страшной и в совершенно иной проекции.

Кроме того, в *Dark Lab* должны изменяться области не вне поля зрения игрока, а вне освещения, но игрок не поймет, что какая-то комната с лампой остается неизменной, если он даже не видит ее. Именно поэтому для данного проекта вид сверху подходит как нельзя лучше.

Отдельно отметим, что, как и *Darkwood, Dark Lab* могла бы быть 2D игрой, однако было принято решение использовать **3D**, чтобы не тратить время на разработку собственной двухмерной системы освещения, а сосредоточиться на других аспектах проекта.

1.1.2. SCP: Containment Breach

Следующая на рассмотрении – *SCP: Containment Breach* (далее *SCP:CB*) [35] – старая бесплатная инди-игра, основанная на статьях и мифах из SCP Foundation [34], рассказывающих о секретной организации, скрывающей таинственные аномальные объекты. В игре игроку предстоит выбраться из зоны задержания с множеством подобных объектов. Она уже упоминалась при описании *Dark Lab* в связи с одной аномальной областью в игре, в которой зациклены коридоры. Во многом *Dark Lab* базируется именно на этой игре.

Что касается особенностей игры, то, кроме естественно пугающей атмосферы (рис.2), множества вариантов окончания игры и различных используемых предметов стоит выделить то, как реализованы монстры, а также механику моргания персонажа.

Рисунок 2. SCP: Containment Breach

Во-первых, в игре нет возможности просто победить монстров. Это, конечно, часто бывает в survival horror играх, ведь бессилие игрока порождает страх. Но в *SCP:CB*, в отличие от многих других игр, игроку дают различные правила, благодаря которым он хоть и слаб по отношению к монстрам, но все еще может что-то сделать, чтобы помочь себе спастись хотя бы на время. Например, один из основных противников в игре – SCP-173 – не может двигаться, пока на него смотрят, но именно поэтому введена механика моргания. Подобная механика противостояния «тьме» реализована и в *Dark Lab,* но об этом подробнее в разборе следующей игры.

Во-вторых, важно то, как в игре ведут себя монстры, когда они появляются. В *SCP:CB* это наполовину автоматическая, но наполовину заскриптованная система. Так, монстры преследуют игрока в процессе игры, действуя в соответствии со своим поведением, своими ограничениями и сильными сторонами, но при этом, если игроку удалось сбежать от монстра, тот перестает просто бесцельно ходить по миру, и становится подконтролен игровой системе. Можно сказать, что он

далее следует сценарию и может появиться в специальных местах, но его поведение не становится слишком заскриптованным и предсказуемым как, например, в *Outlast* [29].

Такая же система реализована в *Dark Lab*. Основной монстр – «тьма» – преследует игрока, пытается ему навредить, но, когда игроку удается отогнать монстра, тот отступает на время, но появится вновь, например, когда игрок найдет ключ к выходу из лаборатории.

Также «тьма», как и SCP-173, **моментально убивает** игрока, когда настигает его, без каких-либо полосок здоровья, которые так часто реализуют в других играх.

1.1.3. Alan Wake

Выпущенная в 2010-м году для Xbox 360 и в 2012-м для Windows, игра *Alan Wake* [2] несколько выбивается из обозреваемой категории игр. Сами разработчиками *Alan Wake* определяется как психологический экшен-триллер, однако ей все же часто присуждают жанр survival horror-а. Чем бы она ни являлась, игра обладает рядом особенностей, одна из которых нашла свое место и в *Dark Lab*.

Сюжет игры вертится вокруг писателя, находящегося в поиске своей пропавшей жены и столкнувшегося с Темной Сущностью, захватывающей город по ночам. Он исследует локации, сражается с монстрами и общается с местными жителями. Игру хвалят за сюжет, музыку, экшен-составляющие и общую кинематографичность.

Рисунок 3. Alan Wake

Однако самой важной особенностью, повлиявшей на *Dark Lab,* является **эффект света на монстров.** В игре освещенные области – это уголки безопасности для персонажа, а свет можно использовать как оружие (рис.3), ослабляя или даже уничтожая любого противника. Но, конечно, это не единственная игра с подобной механикой. Нечто похожее было реализовано, например, в *Metro: Last Light* [56], где пауки боялись света от фонаря.

Так или иначе, эта механика отлично подходит для *Dark Lab*, где свет и тьма уже используются, как важные составляющие геймплея. В нашем проекте свет замедляет «тьму», останавливает ее и может даже отпугнуть и заставить скрыться. Также, если персонаж игрока слишком долго остается на свету, охотящаяся за ним «тьма» отступит до лучших времен. Но было бы слишком просто, если бы любой фонарь спокойно останавливал «тьму». Поэтому «тьма» имеет **сопротивление свету**, которое она со временем увеличивает, поглощая свет рядом. Чем выше сопротивление, тем меньше свет досаждает «тьме». Набрав достаточно много, «тьма» будет игнорировать не только фонарь игрока, но даже лампы в лаборатории.

1.1.4. Sound Design

Этот пункт несколько отличается от предыдущих. Здесь речь пойдет не об особенности какого-то отдельного проекта, а о важном элементе любой игры жанра – о саунд-дизайне – процессе создания звуковой составляющей любого проекта.

Саунд-дизайн – необходимая составляющая для horror игры [54]. Большая часть впечатлений игрока достигается за счет атмосферы игры, а атмосферу составляют данные, получаемые человеком во время игры: звук и визуальная составляющая. С этой точки зрения звуки важны для любой игры, однако horror игры зачастую ограничивают поле зрения игрока, помещают его в темные помещения и лабиринты. В связи с этим игрок получает не так много визуальной информации, и этот пробел могут заполнить звуки, их важность в таких играх значительно выше.

Звуки создают в голове у игрока различные образы, воображение само достраивает картину, недоступную зрению. Если человек видит монстра, он боится его, если же он видит пустую комнату, страх обычно не возникает. С другой стороны, в правильной атмосфере игрока одинаково охватывает испуг, когда он слышит шипение монстра или шорох занавесок от ветра.

Саунд-дизайн является важной частью horror игр еще с ранних представителей жанра. Так, культовой серии *Silent Hill* [37] (рис.4) уже почти 20 лет и эта игра известна за мастерское использование в ней аудио составляющей, а ее саунд-дизайнер Акира Ямаока считается легендой в сообществе фанатов horror игр [55].

Разработчики до сих пор продолжают экспериментировать с саунд-дизайном, стараясь привнести что-то новое в свои игры и создать неповторимую атмосферу. А иногда звуковая составляющая даже становится частью игровой механики, как, например, в находящейся в разработке игре *Stifled* [41], где игрок может ориентироваться только на звуки и их визуальное отображение (рис.5), но они же привлекают и монстров.

В *Dark Lab* подобные механики со звуками добавлены не были, было решено следовать уже упомянутому принципу «less is more» и сосредоточиться на геймплейных составляющих, связанных со светом. Но при этом было уделено внимание основным звукам в игре: добавлены шаги персонажа, создающий пугающую атмосферу эмбиент – ненавязчивая обволакивающая фоновая музыка – и звуки «тьмы».

Рисунок 4. Silent Hill

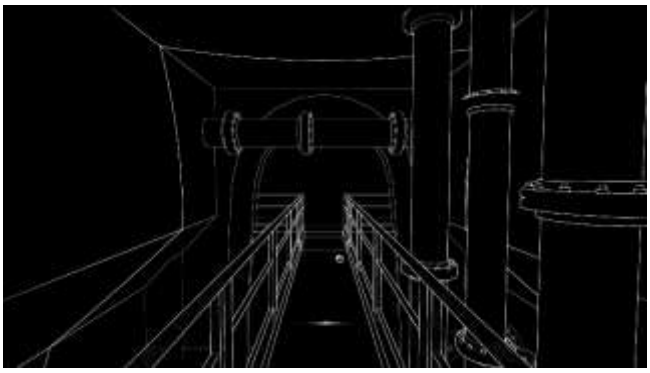

Рисунок 5. Stifled

1.2. Алгоритмы и ограничения генерации карт

Процедурная генерация — это широко используемая техника создания всевозможного контента из кода, а не посредством дизайна вручную. Это мощный инструмент в руках разработчиков игр. Считается, что впервые процедурная генерация была применена в игре *Beneath Apple Manor* [5] (рис.6), выпущенной в 1978-м году, положив начало экспериментам с генерацией контента в играх. Еще через два года, в 1980-м на свет появилась игра *Rogue* [32] (рис.7), ставшая культовой классикой и породившая целый жанр roguelike игр во многом именно благодаря процедурной генерации.

Рисунок 6. Beneath Apple Manor

Рисунок 7. Rogue

Процедурной генерацией можно создавать текстуры, геймплей, предметы и даже другие генераторы. В проекте *Dark Lab* должна быть **генерация лаборатории**, то есть карты. Однако карта может быть представлена очень разными объектами в разных играх. Это может быть лабиринт с узкими разветвленными дорожками, множество длинных платформ, системы пещер или же набор комнат, соединенных коридорами. Последнее подходит для лаборатории как нельзя

лучше и подобные структуры обычно называют подземельями (dungeons), так как в старых играх они представляли из себя лабиринты с тюремными клетками [30].

Комнаты тоже могут быть разных форм и размеров, разные по расположению относительно друг друга. Мы остановимся на прямоугольных комнатах, параллельных друг другу, как наиболее классических среди алгоритмов генерации подземелий, и будем рассматривать по большей части создающие именно их техники.

Рассмотрим теперь элементы стандартного генератора [30]:

1. Прежде всего репрезентативная модель – абстрактное упрощенное представление подземелья, отражающее основную структуру финальной карты.
2. Далее идет метод конструирования этой репрезентативной модели. Можно сказать, что именно этот элемент отвечает за основную генерацию.
3. Наконец, метод создания геометрии подземелья из его репрезентативной модели.

Мы разберем подробнее именно второй пункт. Но сначала ограничения генерации.

1.2.1. Ограничения генерации

Прежде чем рассматривать конкретные техники процедурной генерации подземелий, необходимо определить ограничения (требования), которые стоит наложить на генерацию и на ее результат. Позже во второй главе показано, как выбранный алгоритм удовлетворяет всем этим ограничениям. Сначала обратим внимание на некоторые стандартные ограничения на примере тех параметров генератора, на которых фокусировался разработчик игры *Hauberk* [33]:

1. Генератор должен быть эффективным.

   Обычно это не столь строгое ограничение, когда уровень создается только на старте игры, но для *Dark Lab* скорость работы генератора критична.
2. Подземелье должно быть связанным.

   Для стандартных алгоритмов это означает, что из любой комнаты можно достичь любую другую. Это также автоматически спасает от траты времени на генерацию частей карты, которую игрок даже не увидит. Для *Dark Lab* это ограничение принимает несколько иную форму, потому как лаборатория даже не должна генерироваться сразу вся и меняется в темноте. Так, большую часть времени на карте даже не существует выход, несмотря на то что игрок должен его достичь. Именно поэтому для *Dark Lab* правильным ограничением будет следующее:

2*. Из любой сгенерированной области должен быть путь в несгенерированную.

3. Подземелье должно быть неидеальным.

   «Идеальным» считаются подземелья и лабиринты, в которых между любыми двумя точками есть только один путь. Другими словами, в них нет циклов. Это приводит к

неинтересному опыту, множественным тупикам и необходимости возвращаться по своим стопам. Неидеальные подземелья решают эту проблему.

4. Открытые комнаты.

   В лаборатории должно быть множество комнат.

5. Коридоры.

   Комнаты должны соединяться системой коридоров. Это не столь строгое требование потому как коридорами могут быть просто узкие комнаты.

6. Все части генератора должны быть настраиваемы.

   Это очень важная часть процедурной генерации. Через всевозможные настройки можно достигать совершенно разных результатов, а именно это и является основной целью генерации контента.

В *Dark Lab* используется **динамическая случайная генерация карты**. Случайной ее делают элементы рандомизации, а динамической – использование генерации, когда уже идет игра, а не строго перед запуском уровня. Эта генерация должна случаться в процессе игры вокруг игрока, когда он заходит в новые комнаты, покидает старые, когда выключается где-то свет и т.д. Это накладывает дополнительные ограничения на алгоритмы генерации:

7. Генерация карты не затрагивает «закрепленные» комнаты.

   Это необходимо, чтобы игрок мог «закреплять» какие-то комнаты светом, гарантируя, что они не будут меняться, пока свет горит. Обычно это включает и комнату игрока.

8. Генерация только части карты.

   Большая часть классических алгоритмов создает карту только перед игрой и алгоритмы рассчитаны на генерацию сразу всей карты, что не подходит для *Dark Lab*.

9. «Бесконечные» коридоры и циклы.

   Подобные элементы не встречаются в обычных играх, но должны быть нормой в *Dark Lab*. При правильной генерации игрок будет попадать в подобные ситуации автоматически время от времени.

10. Сложные для прохождения уровни.

    Учитывая постоянную перегенерацию алгоритм должен гарантировать, что выход и другие важные элементы карты не появятся прямо рядом со стартом и потребуют какого-то поиска.

Для проекта были изучены некоторые виды техник, описанные далее.

Как и с обзором игр, приводится не полный перечень всех плюсов и минусов, а базовые описания алгоритмов. Также выделены аспекты методов, противоречащие описанным выше ограничениям на генерацию, не позволяющие использовать эти методы в *Dark Lab*.

1.2.2. Search-based approach

Первый на рассмотрении – подход, основанный на поиске контента с необходимыми свойствами – **search-based approach** [30]. На самом деле это не конкретная техника или алгоритм создания карт, а лишь способ генерации любого контента, когда определяются какие-то свойства генерируемого объекта и необходимая планка этих свойств, после чего итеративно генерируется контент, пока не будет получен подходящий. Этот подход можно было бы применить к любому описанному дальше типу алгоритмов.

Подход, основанный на поиске, требует нескольких центральных элементов: алгоритм поиска, ищущий «хорошие» объекты, репрезентация контента, создаваемого в процессе, и функция оценки свойств или качества созданного объекта.

Обычно в качестве алгоритма поиска используют эволюционные методы, такие как **генетические алгоритмы** [18], в которых сначала алгоритмом генерируется первое поколение создаваемых объектов, затем циклично определяются искомые свойства, «убиваются слабые» объекты и создаются мутации «сильных». Иногда, однако, применяют и обычные переборы.

Главная проблема данного подхода в том, как много времени он занимает. Он нарушает первое обозначенное ограничение на алгоритмы генерации для *Dark Lab* и поэтому не подходит.

1.2.3. Agent-based algorithms

Основанные на агентах алгоритмы [30] обычно используют одного агента для создания подземелья. Этот **агент «копает» коридоры и комнаты** в изначально заполненном материей пространстве (рис.8). В итоге получаются органичные и довольно хаотичные подземелья. При этом сложно предсказать, как настройки искусственного интеллекта (ИИ) агента повлияют на итоговую карту.

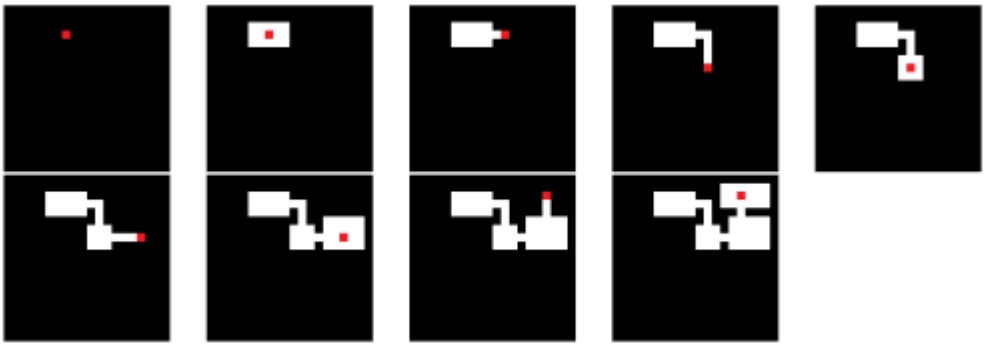

Рисунок 8. Пример работы агентного алгоритма

Один из классических примеров агентного алгоритма это drunkard walk [12] – алгоритм, в котором агент перемещается на каждом шаге в случайном направлении, роет тоннели и останавливается только когда выроет заданное количество клеток.

Несмотря на то, что непредсказуемость агентных алгоритмов затрудняет работу с ними, эта техника не противоречит ни одному из заданных ранее ограничений и, хоть и с рядом изменений, но может быть использования для процедурной генерации лаборатории в *Dark Lab*.

1.2.4. Space partioning

Главная идея следующего вида алгоритмов в том, чтобы **разделять пространство** на части, называемые клетками (cells), а затем соединять эти части, создавая подземелья [30]. Обычно разделение ведется рекурсивно иерархично и в процессе создается space-partioning tree – дерево разделений пространства. Корень дерева представляет всю карту. Разделение продолжается до достижения минимального размера комнаты.

Этот подход ведет к созданию строгих, организованных уровней. Очень важно то свойство, что комнаты гарантированно не пересекаются.

Самый популярный метод – **binary space partioning** (BSP), в котором на каждом шагу разделение происходит ровно на две части и создается BSP tree. Пример показан на рис.9 [3].

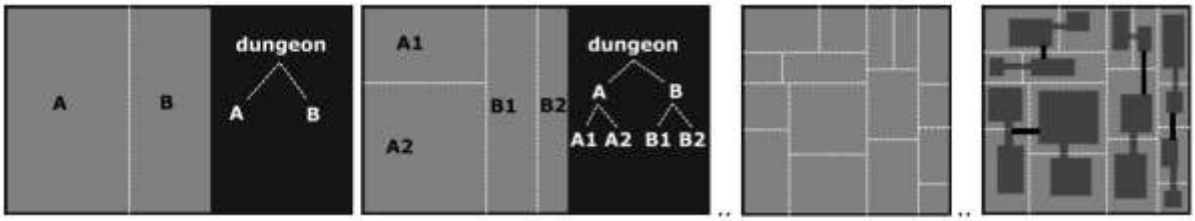

Рисунок 9. Пример работы алгоритма разделения пространства

Несмотря на положительные свойства этого типа алгоритмов, они не подходят для проекта *Dark Lab*. Эти алгоритмы рассматривают подземелье как целое, создавая его с нуля, потому они вступают в конфликт с седьмым и восьмым из поставленных ограничений. Однако идею разделения карты иерархично можно использовать для установления типов различных областей. Например, область ближе к выходу из лаборатории может быть более сложной.

1.2.5. Cellular automata

Клеточные автоматы – известная дискретная модель, изучаемая в компьютерных науках, в физике, математике и биологии и применяемая в том числе и для построения карт [30]. В этой модели пространство является решеткой из ячеек, каждая ячейка находится в одном из предопределенных состояний, и каждая ячейка имеет определенную окрестность – набор соседних ячеек. Далее на каждой итерации по определенным правилам и на основе своей окрестности каждая клетка меняет свое состояние.

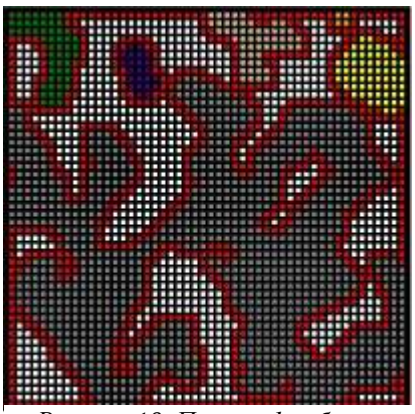

Рисунок 10. Пример 1 работы клеточных автоматов

На клеточных автоматах построены многие алгоритмы, они просты и быстро работают, однако в области генерации карт они обычно создают не подземелья с коридорами и комнатами, а **системы пещер**, как на рис.10 и рис.11 [17].

Рисунок 11. Пример 2 работы клеточных автоматов

Подобные пещеры не подходят для *Dark Lab*. К тому же из других недостатков можно выделить непредсказуемость этих алгоритмов и то, что они оперируют на всей карте, как и space-partioning алгоритмы, нарушая восьмое из поставленных на генерацию ограничений.

В связи с пещерами отдельно стоит упомянуть методы, строящие карты на основе **диаграммы Вороного** [53] (рис.12) Такая диаграмма строится на наборе случайных точек расчетом областей, в которых до каждой конкретной из этих точек минимальное возможное расстояние.

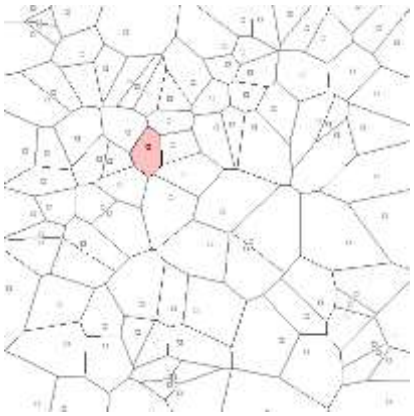

Рисунок 12. Диаграмма Вороного

Из-за получаемой формы такие диаграммы используются как раз для пещер, или же для областей стран, для земли и моря, для территорий фракций. Но использующие их алгоритмы, как и клеточные автоматы, тоже не подходят для *Dark Lab*.

1.3. Выводы по первой главе

Проведен краткий обзор существующих игр в жанре survival horror. Особое внимание уделено функциональным решениям, подходящим для *Dark Lab*. Эти решения в числе основного функционала необходимо реализовать в данном проекте:

- 3D с видом сверху;
- «Тьма» действует наполовину автоматически и наполовину подчиняется «сценарию»;
- «Тьма» моментально действует на персонажа (убивает его);
- Свет опасен «тьме», в нем она замедлена, и она его сторонится;
- Фоновая музыка и звуковые эффекты.

В приложении А «Техническое задание» приведены более подробно функциональные требования к программе. Помимо этого, были выделены ограничения на процедурную генерацию лаборатории и с их учетом разобраны популярные типы алгоритмов создания карт. Несмотря на существование множества мощных техник, подходящих для разных ситуаций, ни одна из них не может быть использовать в *Dark Lab* в чистом виде и лишь основанные на **агентах** методы могут быть имплементированы с некоторыми изменениями. Об этом и говорится в следующей главе.

Глава 2. Описание алгоритмов и структур данных

В этой главе сначала описывается общая структура проекта, выделяются самые важные его компоненты, после этого приводятся особенности этих компонент, используемые ими структуры данных и применяемые в работе алгоритмы.

2.1. Общая структура проекта

Проект *Dark Lab* включает множество зависимых друг от друга компонент (рис.13). Центральное место занимают управляемый персонаж, лаборатория (карта), по которой персонаж передвигается, объекты и предметы на ней, модуль генерации этой карты и «тьма», охотящаяся на игрока. Персонаж и «тьма» подчиняются своим контроллерам – управляющим модулям.

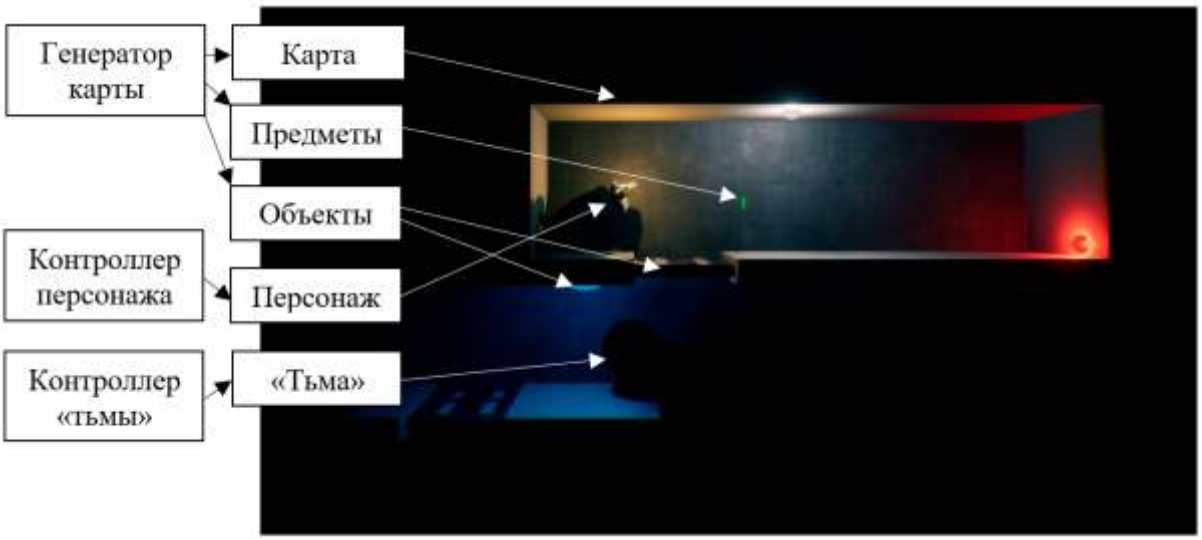

Рисунок 13. Основные компоненты Dark Lab

Подробнее о разных классах и их реализации рассказано в следующей главе, а здесь разобраны только представляющие интерес алгоритмически части программы: игровое пространство и «тьма», но перед этим также описаны ключевые возможности персонажа, потому как, хоть его способности и не включают никаких особых алгоритмов, именно персонаж исследует карту и именно за персонажем охотится монстр в процессе игры, а потому персонаж косвенно влияет на алгоритмы.

«Тьма», наоборот, никоим образом не влияет на генерацию карты и даже не знает о ее существовании (за исключением источников освещения), ведь, в отличие от персонажа, «тьма» может летать через стены. При этом фокус проекта был не на интеллекте монстра, а на генерации карты, потому поведение «тьмы» достаточно примитивное. Оно кратко описано в конце главы.

Наконец, самую весомую часть как всего проекта, так и этой главы составляет **создание игрового пространства**. Именно оно вызвало наибольшие трудности в разработке. В пункте 2.3. подробно разобрана структура карты и все стадии работы алгоритма ее генерации.

2.2. Возможности персонажа

Главная возможность персонажа – это, конечно, его **способность передвигаться** по карте. Игрок проводит большую часть игры в движении, ведь цель – достичь выхода. Соответственно, карта предоставляет пространство для этого движения, а также ограничивает его, когда это необходимо. Генератору нельзя запереть игрока, ведь тот не может ходить через стены или телепортироваться как «тьма».

Игрок может двигаться в любом направлении на плоскости, а также смотреть в любую сторону вокруг, однако это не нужно специально учитывать в алгоритмах генерации карты, ведь они зависят от освещения, а не от взгляда игрока (хотя вместе со взглядом игрок может поворачивать включенный фонарик и это генератор уже будет учитывать).

«Тьма» также постоянно отслеживает передвижения игрока.

Следующая возможность игрока – это **использование объектов** на уровне: игрок может открывать двери (рис.14), подбирать предметы в свой инвентарь. Генератор, в свою очередь, должен расставлять эти объекты по карте в доступных игроку локациях. «Тьме» же объекты карты безразличны, если это не осветительные приборы.

Рисунок 14. Персонаж с зажигалкой рядом с открываемой дверью

Наконец, игрок может **экипировать** некоторые из подобранных предметов и **использовать** их. На генерацию карты и поведение тьмы это не влияет (опять же, если это не что-то дающее свет вроде зажигалки).

2.3. Генерация карты

Еще со времен задумки проекта было понятно, что генерация карты станет самым сложным компонентом программы, требующим разработки нового алгоритма и структур данных. В первой главе были разобраны существующие алгоритмы и решено, что единственный подходящий их тип – это **агентные**. В итоге разработанный метод опирается на одну из основных идей агентных алгоритмов: генерация происходит не извне, а изнутри карты методом **расширения**. На рис.15 промежуточный результат.

Рисунок 15. Пример генерации части карты, вид из редактора

Далее разобраны разработанные структура карты и основной цикл ее генерации, а также указано, как были удовлетворены поставленные в первой главе ограничения на генерацию.

2.3.1. Структура карты

Стояла задача разработать такую структуру, которая позволяла бы создавать карту, привязанную к **клеточной сетке**, но при этом не ограничивала бы такой сеткой передвижения игрока. В связи с необходимостью генерировать карту расширением нужно было также удобно отслеживать все ходы между комнатами.

В итоге двумя ключевыми единицами карты являются комната и проход. Свойства любой комнаты включает ее расположение на координатной сетке и ее размер, а также указатели на все проходы, ведущие **в** комнату или **из** комнаты. Считается, что любая комната окружена стеной шириной в одну клетку по всему периметру, кроме мест расположения проходов.

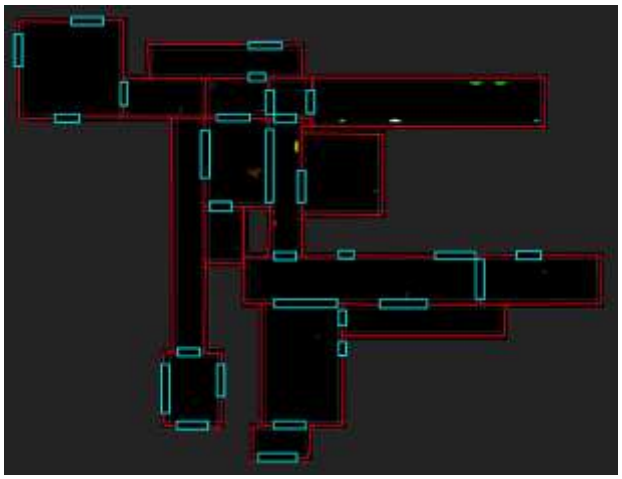

Рисунок 16. Пример связи комнат (красные линии) через проходы (бирюзовые линии). Стены комнат схематично показаны тонкими, но на самом деле имеют единичную ширину (более бледные голубые линии) и могут пересекаться друг с другом

Свойства прохода в свою очередь включают его положение на карте, ширину прохода, одно из четырех направлений (вверх, вниз, вправо, влево), факт того, является ли проход дверью, и цвет прохода (важен только если это дверь). Проход также хранит указатели на две комнаты, которые этот проход и соединяет (одна «в» и одна «из»), причем, если одна из комнат перестала существовать, проход остается стоять и к нему может в будущем быть подсоединена новая комната, однако, если проход теряет обе комнаты, а значит больше не ведет ни «в», ни «из», он удаляется.

Такая структура (рис.16) позволяет не только удобно создавать новые комнаты лаборатории и отслеживать их соединения, но также хорошо справляется именно с поставленной задачей перестраивания карты в процессе игры, так как проходы создают удобную **прослойку** между комнатами карты. Эта структура также удовлетворяет 3-му и 4-му ограничениям генерации из первой главы: она позволяет описывать комнаты и коридоры, где коридоры на деле являются просто длинными комнатами.

Можно заметить, что в таком виде структура не позволяет отслеживать некоторые параметры, например, объекты в комнатах, вид самих комнат, высоту стен и др. И это будет верное замечание. Дело в том, что в *Dark Lab* есть разделение между основной моделью и тем, что можно назвать воплощением или физической моделью лаборатории. В пункте 1.2. первой главы была рассмотрена работа стандартного генератора и представлено классическое разделение между репрезентативной моделью и геометрией подземелья. Именно такое

разделение присутствует и в *Dark Lab,* но при этом не у всех объектов есть отдельная репрезентативная модель.

Физическая модель (рис.17), существующая в пространстве, так же важна для генерации, как и репрезентативная, хранящая базовые данные о комнатах. В то время как именно вторая генерируется (прежде чем быть переведенной в первую), именно в первой, к примеру, проверяется освещенность той или иной комнаты или прохода (а также «тьмы», но на генерацию карты это не влияет).

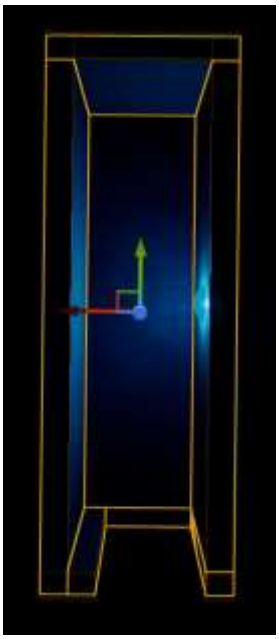

Рисунок 17. Объекты комнаты с проходом

В итоге хоть и можно было абсолютно всем физическим объектам дать отдельное описание в репрезентативной модели, это оказалось нецелесообразно в реалиях задачи *Dark Lab.*

Подробнее про структуры данных, связанные с объектами лаборатории, будет рассказано в следующей главе, потому как они не представляют интереса алгоритмически и участвуют только в последних двух фазах цикла генерации, где мы их также немного затронем.

2.3.2. Основной цикл генерации

Каждая игра в *Dark Lab* начинается с генерации карты (рис.18). Сначала создается стартовая комната (подробнее про это в 2.3.2.3.), в нее помещается персонаж и затем генератор запускает один проход цикла генерации карты. Цикл генерации карты состоит из четырех последовательных фаз: изменение темноты (reshape darkness, или просто reshape), расширение карты (expand), создание или спаун комнат (spawn) и заполнение их предметами (fill). Первая фаза иногда опускается. Например, как раз на старте игры, потому как еще нечего изменять.

Следуя принципам, схожим с агентными алгоритмами, каждая из этих фаз, кроме первой, запускается из определенной начальной комнаты на карте. Далее алгоритм в каждой из этих трех фаз действует рекурсивно, о чем подробнее в соответствующих подпунктах. Изначально в процессе разработке фаза reshape действовала так же.

Таким образом, для того чтобы алгоритм работал, ему нужна какая-то **начальная комната**. Учитывая, что алгоритм расширяет и изменяет карту в процессе, самым логичным выбором оказалась комната, в которой и находится игрок при запуске алгоритма. Так каждый раз, когда меняется карта, можно быть уверенным, что игрок это заметит и изменения не будут сделаны впустую.

Еще одной важной частью алгоритма является **момент запуска очередного прохода**. Пусть это и *цикл* генерации, он не срабатывает на каждом кадре, это было бы слишком ресурсозатратно. В этом плане существует два сценария. Первый – игрок куда-то двигается, и

карта должна меняться из-за этого в процессе и не отставать от игрока. Второй – игрок посмотрел за дверь, потом закрыл эту дверь и ждет: карта со временем так же должна поменяться, и за дверью может оказаться что-то новое. Второе это всего лишь пример, но таких ситуаций может быть много.

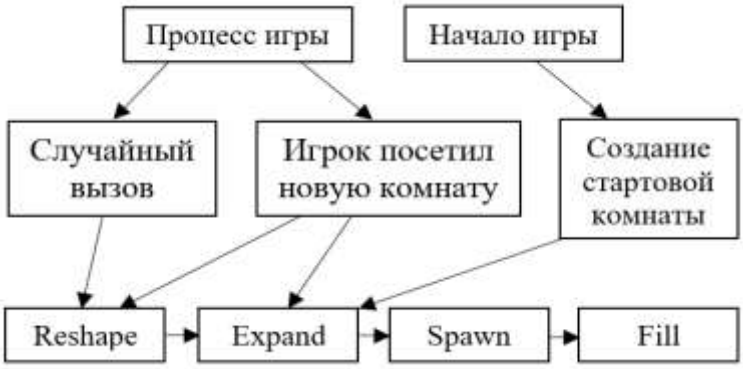

Рисунок 18. Схема вызова генерации карты

В итоге проход цикла запускается при двух возможных условиях:

1) Игрок зашел в новую комнату. В этом случае фаза reshape с определенной вероятностью будет опущена, ведь лаборатория не обязана меняться после *каждой* посещенной игроком комнаты. Остальные же фазы работают в обязательном порядке. Что интересно, если никогда не запускать reshape (и если второе условие также отключено и не случается), алгоритм будет просто бесконечно создавать карту, куда бы ни шел игрок, но созданные части карты будут оставаться статичными.

2) Случайный вызов перегенерации. Проверка на случайный вызов осуществляется каждые несколько секунд с заданной вероятностью. Если такой вызов произошел, обязательно срабатывают все фазы цикла генерации, потому как обычно это случается в комнатах, где фазы expand, spawn и fill уже были проведены ранее и запускать их снова, не сделав перед этим reshape не имеет смысла.

Говоря о заданных вероятностях и других константах, стоит отметить, что генератор удовлетворяет 6-му из поставленных ранее ограничений: численные характеристики генератора **настраиваемы** (рис.19) и могут быть легко изменены разработчиком, если нужно настроить генерацию карты определенным образом.

```
int ExpandDepth = 5;
int SpawnFillDepth = 4;
int ReshapeDarknessDepth = 3;
int MaxFixDepth = 4;
int MinRoomSize = 5;
int MaxRoomSize = 35;
int MinRoomArea = 25;
int MaxRoomArea = 250;
int MinRoomNumOfPassages = 1;
int MaxRoomNumOfPassages = 8;
```

Рисунок 19. Некоторые параметры генерации карты

Перейдем к разбору каждой из фаз цикла генерации.

2.3.2.1. Reshape

Первой фазой цикла генерации является **изменение темноты**. Это осуществляется в двух подфазах. Сначала алгоритм проходится во всем заспауненным (созданным в физической модели) комнатам и удаляет те из них, что находятся полностью в темноте (на самом деле не удаляет полностью, а

Рисунок 20. Пример рейкастов между осветительными приборами и "тьмой"

добавляет в пул (**Pool Darkness**), но об этом позже). Такая проверка освещенности осуществляется посредством рейкастов (raycast, «бросание луча», проверка пересечения с чем-то по прямой) между близкими источниками освещения и заданными точками в комнатах и в проходах. Тот же принцип используется и при проверке освещенности «тьмы» (рис.20) только для нее есть дополнительные вычисления.

Кроме неосвещенных комнат удаляют также незаспауненные. Комнату игрока и ряд других исключений, наоборот, не удаляют никогда. После всех удалений остается набор комнат, многие из которых имеют «сломанные» проходы – проходы, соединенные лишь с одной комнатой. Соответственно, «сломанными» комнатами называют комнаты, имеющие хотя бы один «сломанный» проход. Если в этот момент перейти к следующей фазе цикла генерации, возникнет множество ошибок. Поэтому во время прохода подфазы удаления темноты освещенные комнаты и комнаты-исключения не удаляют, а добавляют в список на «починку».

Когда все удаления закончены, комнаты из списка начинают «чинить» (**Fix**), первой выбирая наиболее приоритетную – комнату игрока. Приоритет важен, потому как в процессе «починки» некоторые проходы из неудаленных комнат могут быть удалены, а иногда удаляются даже целые комнаты, которые не удается «починить». Из-за этого фазу reshape можно назвать самой ненадежной, но она все еще работает стабильно, выполняя задачу изменения темноты (рис.21-25).

«Починка» одной «сломанной» комнаты происходит следующим образом:

1) Производится цикл по всем ее проходам.

2) Находятся «сломанные» проходы.

3) Определяется критичность необходимости «починить» этот проход – для проходов из комнаты игрока и других исключений.

Рисунок 21. Reshape: Начало

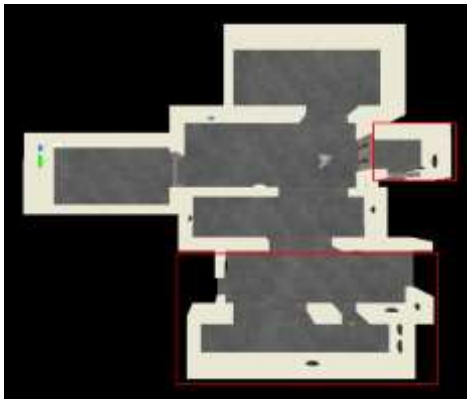

Рисунок 22. Reshape: Эти в темноте

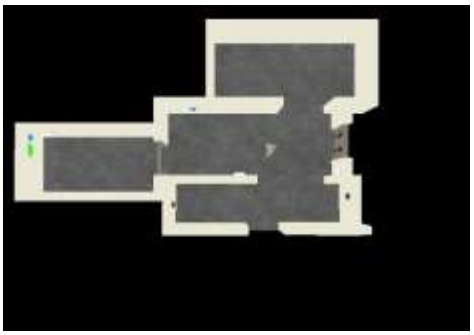

Рисунок 23. Reshape: Pool Darkness

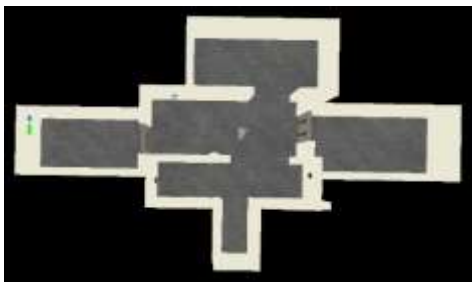

Рисунок 24. Reshape: Fix

Рисунок 25. Reshape: Конец. На первый взгляд игрока ничего и не изменилось, но есть новые комнаты – успех. На деле игрок не видит результат reshape, ведь после еще три фазы

4) Определяется общая необходимость «починить» этот проход – необходимо, если это уже определено как критичное условие, либо если проход освещен.

5) Если общей необходимости «починить» нет, с определенной вероятностью решается «починить» в любом случае.

6) Если до этого момента решается, что «чинить» не нужно, (и то же самое дальше по алгоритму, когда все-таки решается не «чинить» проход) проход удаляется, а если одна из его комнат заспаунена (имеет физическое воплощение на экране), то на месте прохода спаунится новая стенка (чтобы не респаунить (создавать заново) всю комнату).

7) Наконец, если проход необходимо «чинить», это происходит почти так же, как создание/поиск новой комнаты для прохода в фазе expand, что описано подробнее далее. Вкратце, находится минимальное пространство для комнаты и производится попытка либо создать новую комнату, либо присоединиться к уже существующей. Обычно, если ни то, ни то не выходит, проход решают все же не «чинить». Однако в ситуациях, когда «починка» критична, проводится дополнительная проверка тех комнат, которые мешают «починке» и, если они не приоритетны, уже их удаляют и происходит дополнительный рекурсивный вызов «починки» для комнат, соседних с новыми удаленными. Рекурсия ограничена глубиной.

По окончании фазы reshape гарантируется **отсутствие «сломанных» комнат**.

Из ограничений генерации из первой главы эта фаза гарантирует удовлетворение двух. Во-первых, «закрепленные» светом комнаты после фазы reshape остаются на месте. При этом можно сразу отметить, что в других фазах в принципе комнаты никогда не удаляются. Это удовлетворяет 7-му ограничению. Во-вторых, за счет комбинации удалений этой фазы и расширений фазы expand создаются «бесконечные» циклы на карте – определенные изменяющиеся системы комнат и проходов, которые создают иллюзию замкнутости там, где ее быть не должно. Тут же укажем, что бесконечные коридоры существуют за счет фазы expand, о которой речь пойдет далее, и вместе коридоры и циклы удовлетворяют 9-му ограничению.

2.3.2.2. Expand

Фаза expand отвечает за **расширение карты** начиная с какой-то комнаты, то есть как раз в этой фазе происходит основная генерация лаборатории: создаются новые комнаты и проходы. Одним из важнейших условий, поставленных в первой главе, было «из любой сгенерированной области должен быть путь в несгенерированную». Так как генерацию было принято осуществлять из комнаты игрока, это условие преобразовалось в **«игрок всегда имеет путь в несгенерированную область»**. Это способ гарантировать способность игрока продолжать исследовать лабораторию. И это достигается на верхнем уровне фазы expand.

Считается, что у игрока есть путь из комнаты А в комнату Б, если:

1) Комнаты А и Б соседние и игрок может пройти, то есть их соединяет пустой проход или проход с дверью, для которой у игрока есть карта.

2) Или если есть путь из комнаты А в другую комнату, из которой есть путь в Б.

На деле это означает, что игрок может пройти по проходам из одной комнаты до другой используя свои ключ-карты. Через белые двери игрок может проходить всегда (рис.26).

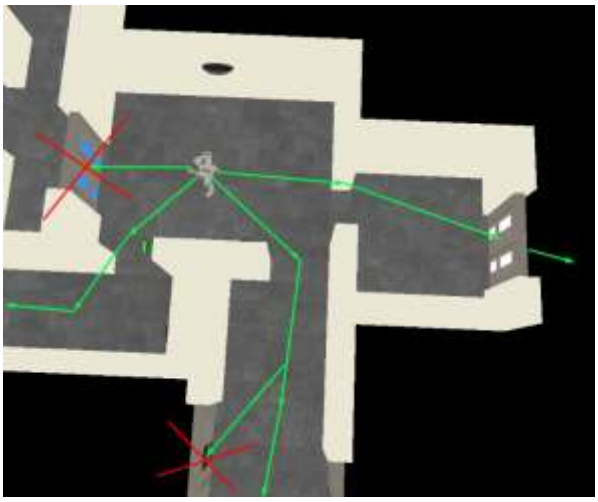

Рисунок 26. Пример наличия и отсутствия пути

Сгенерированными частями карты считаются расширенные (expanded) комнаты. Поэтому необходимо, чтобы из комнаты игрока всегда был путь в нерасширенные комнаты (хотя бы в одну такую). В итоге, **расширение на верхнем уровне** можно представить так:

1) Пробуем сделать расширение в глубину (позже подробнее, что это такое).

2) Если удалось и есть путь в нерасширенные, это конец фазы.

3) Если нет, то сначала пытаемся поменять цвет некоторых незаспауненных дверей на белый, потому как это может мешать нахождению пути. Конечно, стоит ограничение, чтобы дверь выхода никогда не поменяла так свой цвет. Путь есть? Если да, конец фазы.

4) Если пути все еще нет, повторяем фазу reshape, эффективно очищая карту от того, что игрок не видит, а затем снова расширяем карту и снова проверяем получилось ли на этот раз.

5) Повторяем пункты 3-4 до тех пор, пока expand не удастся (или же пока не случится одно из предусмотренных исключений). На всякий случай количество таких повторов ограничено (рис.27).

По итогам за одну или несколько итераций (иногда с reshape в процессе) создается такая карта, которая удовлетворяет поставленному условию и не запирает нигде игрока.

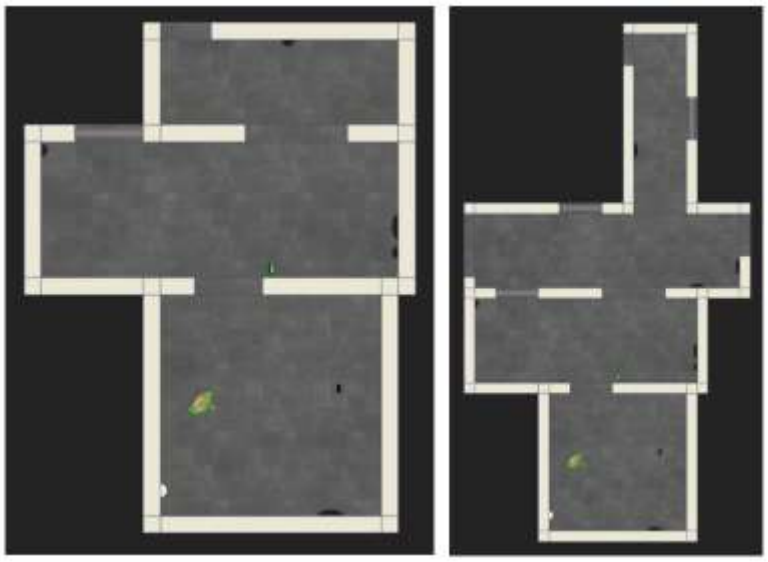

Рисунок 27. Пример совместной работы Reshape и Expand. Слева – до прохода генерации, справа – после

Теперь перейдем к **расширению в глубину**. Это процесс расширения многих комнат вокруг определенной стартовой точки рекурсивно.

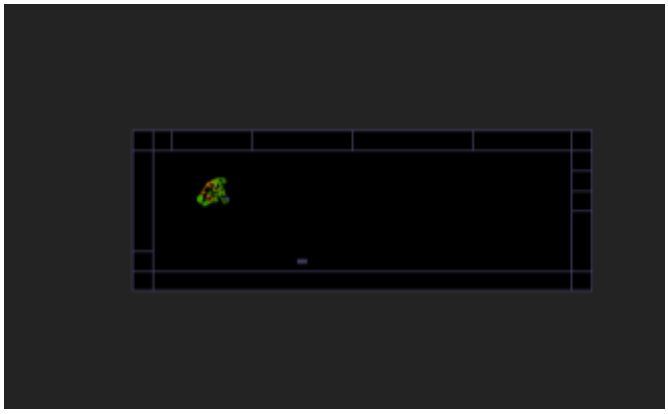

Рисунок 28. Expand: Начало

1) Первым делом выбирается комната начала применения алгоритма (это обычно комната игрока).

2) Если эта комната еще не расширена и не заспаунена (обычно все заспауненные расширены, но об этом в пункте 2.3.2.3.), мы расширяем эту комнату. Процесс расширения одной комнаты описан дальше.

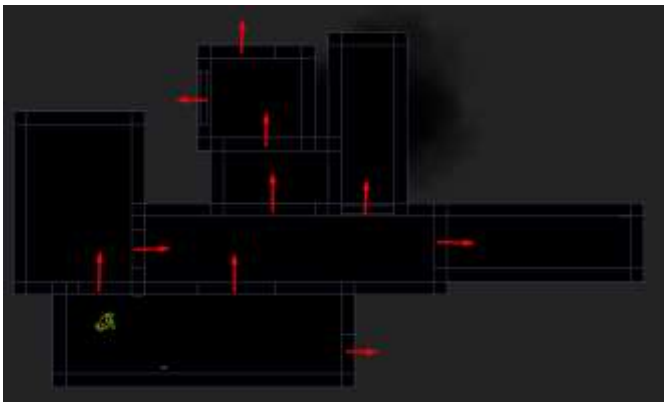

Рисунок 29. Expand: Конец. Стрелками указаны направления расширения

3) Если глубина рекурсии слишком большая, останавливаемся.

4) Если еще не остановились, находим все соседние комнаты и повторяем расширение в глубину рекурсивно с ними, передавая им новое значение глубины.

5) Так до тех пор, пока все ветви рекурсии не завершат работу.

По результатам такой простой рекурсии все комнаты на определенном расстоянии от игрока оказываются расширенными (expanded) и готовыми к спауну и заполнению (рис.28-29). Подход создания карты таким образом изнутри, как уже говорилось, позаимствован у агентных алгоритмов и благодаря ему создается не вся карта, а лишь ее часть, что удовлетворяет 8-му из поставленных ограничений.

Наконец, самый низкий уровень – **расширение одной комнаты**. Во многом это совпадает с «починкой» комнаты на фазе reshape, только нет «сломанных» проходов, которые необходимо «починить», а создаются новые проходы из комнаты и уже для них пытаются создать новые нерасширенные комнаты или найти возможность присоединиться к существующим комнатам.

1) Сначала проверяются некоторые исключения, такие как комната выхода, которую мы не хотим расширять.

2) Далее определяется желаемое количество проходов в комнате и максимальное количество попыток создания этих проходов, которое нельзя превышать.

3) После для каждой попытки сначала ищется случайное место для прохода в стенах комнаты (проводится еще проверка, не занято ли это место), случайно же определяется, должен

ли быть этот проход дверью. Так же определяют, будет ли этот проход выходом. Предусмотрено, чтобы выход появлялся только через какое-то время после блуждания игрока по лаборатории, что удовлетворяет 10-му ограничению из первой главы.

4) Далее, уже имея представление, какой проход мы пытаемся «пробурить» в стене, мы создаем минимальное пространство для комнаты за стеной, после чего проверяем, не пересекается ли это пространство с одной из заспауненных комнат.

5) Если такое пересечение есть, то обычно мы не хотим трогать комнату, с которой пересеклись (и отбрасываем попытку как провалившуюся). Однако в случае если та комната полностью в темноте и не является комнатой игрока, мы удаляем ее с карты, но оставляем в репрезентативной модели, чтобы заспаунить ее заново потом, перед этим соединив нашу расширяемую комнату с ней через созданный проход. Такое соединение возможно только если пересеченная комната полностью покрывает минимальный размер комнаты прохода.

6) Если пересечения с заспауненными нет, мы проверяем, не пересекается ли то же минимальное пространство прохода с одной из созданных, но еще не заспауненных комнат.

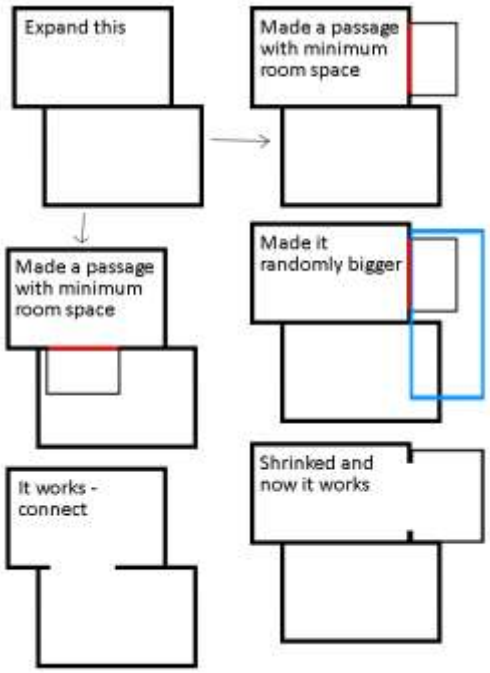

Рисунок 30. Два основных варианта успешного расширения комнаты

7) Если такое пересечение есть, мы проверяем, чтобы пересекшаяся комната полностью покрывала минимальное пространство прохода и тогда к ней можно присоединить нашу расширяемую комнату через созданный проход (рис.30). Если покрывает неполностью, попытка создания прохода считается провальной. Именно за счет соединения с уже созданными комнатами достигается неидеальность карты, то есть то, что в ней как в графе есть обычные циклы, что удовлетворяет 3-му ограничению на генерацию карты из первой главы.

8) Если до этого момента не было ни провалов, ни успешных присоединений, потому как не было пересечений с другими комнаты, то принимается решение о создание новой случайной комнаты. Так как уже определено и проверено минимальное место для комнаты, то провал уже случиться не может, ведь новую комнату базируют именно на этом минимальном пространстве. Первым делом это *пространство случайным образом увеличивают*. Это необходимо для создания больших комнат, но, естественно, приводит к возникновению новых пересечений с существующими комнатами. Однако на этот раз не нужно проверять, можно ли с кем-то соединиться. При нахождении пересечения *область просто уменьшают* минимально (как можно меньшее уменьшение, shrink) в сторону минимального проверенного пространства комнаты. В

итоге это не только позволяет найти комнате место на карте, но и делает комнаты более близкими друг к другу, что в будущем повышает шанс присоединения одной комнаты к другой при пересечении минимальной комнаты нового прохода с существующей комнатой (пункты 5 или 7).

9) Наконец, с созданными параметрами размера и расположения создается новая комната и как ей, так и расширяемой комнате добавляется созданный проход.

Таким образом, расширение карты осуществляется на **трех уровнях** и именно эта фаза решает, по каким комнатам будет ходить игрок. А вот как они будут выглядеть решается в следующей фазе – spawn.

2.3.2.3. Spawn

В этой фазе генерации карты абстрактные модели комнат и проходов преображаются в набор видимых игроком стен, поверхностей пола и дверей в трехмерном пространстве (рис.31). Как и расширение в глубину, спаун происходит **рекурсивно из начальной комнаты** до заданной глубины, причем параллельно с ним для этих комнат сразу происходит и наполнение (fill). Можно было бы сначала заспаунить нужные комнаты и только потом заполнять, однако такой подход не изменит результат, но несколько увеличит количество выполняемых операций.

Рисунок 31. Основные составляющие комнаты

Надо отметить, что операция создания нового объекта на сцене является одной из самых затратных во всем алгоритме, а потому был применен ряд оптимизаций (уменьшений затрат времени и памяти), направленных на ускорение работы алгоритма в этой фазе. Основные из них это, во-первых, переиспользование объектов карты вместо постоянного их создания и удаления (pooling) и, во-вторых, «умный» выбор комнат, которые необходимо спаунить (не просто в глубину как в expand, а с ограничениями). Подробнее об этом рассказано в пункте 3.3.1., но главное, что за счет этих улучшений работы алгоритма удовлетворяется первое поставленное в первой главе ограничение на генерацию – генератор работает эффективнее.

2.3.2.4. Fill

Наконец, последняя фаза генерации, происходящая параллельно со спауном комнат – это **заполнение комнат объектами**. Эта часть алгоритма работает достаточно прямолинейно: для разных типов объектов определяется, нужны ли они в комнате и сколько нужно и максимальное количество попыток создать эти объекты. Попытки могут быть неудачные, потому как каждый созданный объект занимает в комнате место, на которое другой объект быть заспаунен уже не может. К тому же, проходы в стенах так же занимают место внутри комнаты, чтобы, например, настенные лампы не могли появляться в двери или в воздухе в проходе.

Кроме полностью случайной генерации предусмотрены также два исключительных случая. Во-первых, выходная комната. При обнаружении такой в ней создается объект для выхода. Во-вторых, комнаты в которые ведет только один проход и это цветная (не белая) дверь. В таких комнатах очень высокая вероятность обнаружить ключ-карту следующего «уровня».

Упомянутый выше и описанный подробнее в третьей главе pooling относится не только к стенам и дверям на карте, но и к интерактивным объектам лаборатории.

2.4. Поведение «тьмы»

«Тьма» – это чувствительный к свету летающий сгусток черного «дыма», враждебно настроенный против игрока (рис.33). Одно это предложение определяет основные особенности «тьмы»: она может летать (на плоскости), она преследует игрока, и она боится источников освещения. Также в определенных условиях «тьма» может телепортироваться в темноте. Для управления поведением «тьмы», ей было присвоено одно из трех **состояний**:

1) Пассивное – ничего не делает, безобидно по отношению к игроку.
2) Охота – стремится настигнуть игрока и убить его.
3) Отступление – старается уйти подальше во тьму.

Но в каком бы ни была состоянии «тьма», ее первый инстинкт всегда это скрываться от слишком яркого света. Если тьма оказывается освещена настолько, что ее «сопротивление свету» ей не помогает, она начинает пятиться от света вместо того, чтобы делать что-либо еще. Даже если «тьма» не испугана светом, тот продолжает ее замедлять.

«Тьма» начинает игру в **пассивном** состоянии. Какое-то случайное время она ничего не делает, затем начинает охоту. Целью охоты естественно является игрок, который получает сообщение о начале охоты на экране (рис.32).

В процессе **охоты**, если «тьма» находится в темноте, но слишком далеко от игрока, она может мгновенно переместиться (телепортироваться) ближе к нему. У телепорта есть определенный кулдаун (cooldown – время восстановления способности). Также «тьма» просто

движется напрямую в сторону персонажа. Она не старается огибать освещение, но и не «застрянет» в нем, потому что ее сопротивление свету на ярком свету постоянно увеличивается. Охота заканчивается только в одном из двух условий. Первое – «тьма» выполнила свою задачу и убила персонажа, это конец игры. Второе – прошло максимальное время охоты, определяемое на старте этой фазы. Есть одно исключение, но об этом чуть ниже.

Рисунок 32. Сообщение о начале охоты "тьмы"

С окончанием охоты «тьма» начинает **отступать**. Она выбирает последний самый яркий источник освещения и движется в противоположную от него сторону. В большинстве ситуаций такой подход срабатывает и «тьме» удается уйти в темноту, после чего она переходит снова в пассивное состояние. Это же происходит если просто заканчивается максимальное время отступления «тьмы».

Таким образом, основа поведения «тьмы» составляет простой **цикл из трех состояний**. Но есть одно исключение, повышающее сложность и интерес игры: в момент, когда игрок подбирает в лаборатории черную ключ-карту (а значит он теперь имеет возможность открыть выход, когда найдет его), «тьма» начинает охоту из любого своего состояния и, более того, эта последняя охота не прекратится, пока «тьме» наконец не удастся настигнуть персонажа.

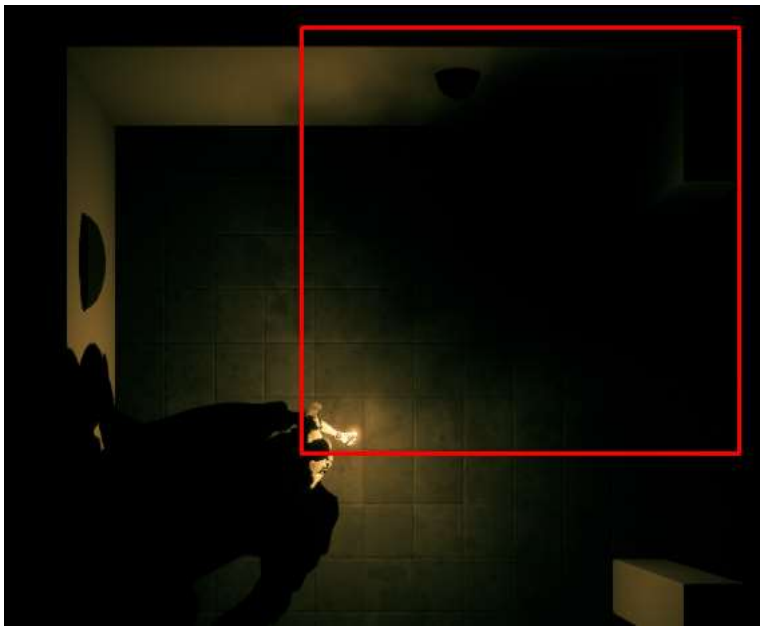

Рисунок 33. Вид "тьмы" в игре

2.5. Выводы по второй главе

Была описана общая структура проекта, для основных компонент были выделены особенности, показаны используемые структуры данных и применяемые алгоритмы. Особое внимание уделено генерации карт, как ключевой части *Dark Lab*: разобрана структура карты в лаборатории, показан проход цикла reshape – expand – spawn – fill, вызовы этой операции и детали работы каждой из фаз на нескольких уровнях абстракции.

Далее описаны детали программной реализации.

Глава 3. Программная реализация

В этой главе сначала обоснован выбор средств разработки *Dark Lab*, далее разобрана архитектура программы и некоторые особенности ее реализации и наконец представлен функционал готового приложения и его интерфейс.

3.1. Выбор средств и инструментов разработки

На данный момент для разработчиков игр доступен очень широкий набор инструментов. Всегда можно начать разработку на одной из множества медиа библиотек вроде *SFML* (*Simple and Fast Media Library*) [38], или же взять какой-то из более высокоуровневых движков, например, *GameMaker* [15], *Unity* [47] или *Unreal Engine* [48]. С другой стороны, можно, наоборот, вести разработку напрямую используя функционал *OpenGL* [28] или *DirectX* [11]. Любой выбор имеет свои плюсы и минусы.

Рисунок 34. Логотип SFML

Рисунок 35. Логотип GameMaker

Рисунок 36. Логотип Unity

Рисунок 37. Логотип Unreal Engine

Первым делом рассмотрен выбор между низкоуровневыми библиотеками или фреймворками и полноценными игровыми движками. Основное дело в фокусе разработки. В то время как низкоуровневые инструменты предоставляют более широкие возможности для детальной настройки любой компоненты под себя, движки сразу поставляют продвинутый функционал и стек внутренних инструментов, настроенных друг под друга и облегчающих процесс разработки.

Обычно движки **не** выбирают только в двух случаях:

1) Когда действительно необходимо серьезно кастомизировать (настраивать) какие-то элементы, скрытые высокоуровневой оболочкой движка;

2) Когда использование движка накладывает серьезные финансовые ограничения, ведь обычно разработчики движка получают часть прибыли с каждого использующего их продукт или иным способом обеспечивают собственную прибыль за использование их движка.

Учитывая, что фокус разработки *Dark Lab* был на процедурной генерации карты, а не на графике или на иных аспектах, для которых пришлось бы использовать более низкоуровневые функции, это не

был первый случай. Не был и второй, ведь по крайней мере на данном этапе *Dark Lab* является бесплатным продуктом и будущая монетизация остается открытым вопросом.

Рисунок 38. Логотип CryEngine

Таким образом, было **решено использовать движок**. И далее встает вопрос о его выборе.

Первым делом выбор основан на том, что *Dark Lab* должна обладать трехмерной графикой. Естественно, все двухмерные движки не подходят. Из основных популярных вариантов остаются уже упомянутые *GameMaker*, *Unity* и *Unreal Engine*, а также *CryEngine* [8] и *Godot* [20]. *GameMaker* сразу отпадает, потому как лишь в последних его версиях появилась ограниченная поддержка трехмерной графики и в основном он все же нацелен на 2D. Из оставшихся каждый обладает своими сильными и слабыми сторонами, но мы остановимся только на наиболее важных из них, определяющих итоговый выбор.

Рисунок 39. Логотип Godot

В первую очередь – сообщество пользователей. Может показаться, что небольшое сообщество – это не проблема движка и не должно повлиять на выбор инструмента разработки, однако на деле чем больше сообщество, тем лучше поддержка, меньше количество неразрешенных проблем движка, проще найти необходимую информацию и продолжать разработку программы, а не останавливаться из-за непреодолимых багов. Именно в связи с малым количеством пользователей в сравнении с конкурентами отбрасывается *Godot*, несмотря на продвинутую документацию и инструменты (чем обладают все оставшиеся варианты) и несмотря на полную бесплатность и открытость кода.

Остаются только три движка и это стандарты индустрии: *Unity*, *Unreal Engine* и *CryEngine*. Любой из них подошел бы для разработки *Dark Lab*, но все же некоторые преимущества *Unreal Engine* позволяют назвать его лучшим для разработки данного проекта.

Во-первых, лицензирование. Система оплаты *Unreal Engine* более удобна для разработчиков. Более того, существует возможность получить 5000$ от создателей движка для вывода игры в массы [57].

Во-вторых, графика. *Unreal Engine* считается одним из самых красивых игровых движков на рынке. В то время как *CryEngine* еще лучше в этом плане, оба сильно превосходят *Unity*.

В-третьих, открытый код. Опять же как и *CryEngine* *Unreal Engine* предоставляет разработчикам весь код. Доступ к коду позволяет глубже понять работу инструмента и помогает в решении множества проблем. Более того, код *Unreal Engine* можно менять, полностью настраивая его под себя, прямо как в низкоуровневых инструментах. В *Unity* доступ к коду не

был доступен на момент выбора движка и появился официально лишь в марте 2018-го, но и сейчас он доступен лишь для просмотра, а не редактирования [31].

К этому моменту может показаться, что *CryEngine* подходит почти так же хорошо как и *Unreal Engine*, однако, возвращаясь к вопросу сообщества, *CryEngine* отстает от обоих конкурентов, причем, именно *Unreal Engine* считается самым популярным движком на ПК [46].

Наконец, важно упомянуть такую особенность *Unreal Engine*, выделяющего этот движок среди аналогов, как система Blueprints для визуального скриптинга, про которую подробнее рассказано в пункте 3.3.2.

В итоге для разработки было принято **решение использовать Unreal Engine**.

Далее в плане инструментов остается вопрос выбора среды и языка программирования. И то, и то определяется *Unreal Engine*.

В качестве IDE используется **Visual Studio 2017** [52], ведь *Unreal Engine* разработан чтобы быть с ней максимально совместимым, обеспечивая удобство процесса разработки [36].

Рисунок 40. Логотип Visual Studio 2017

Программирование на движке осуществляется на **языке C++**.

Стоит отметить, что несмотря на то, что *Unreal Engine* использует обычный C++ для разработки, поверх этого языка существует множество систем, из-за которых работа программиста отличается от стандартной на том же языке вне *Unreal Engine*.

Так, например, есть инструмент Unreal Header Tool [49], добавляющий еще один шаг в процесс сборки программы. Существует набор различных макросов, которые распознаются этим инструментом, для того чтобы основываясь на них и на изначальном файле с кодом сгенерировать новый, измененный файл, который уже будет передан C++ компилятору [22].

Рисунок 41. Логотип C++

Другой важной особенностью является встроенный в *Unreal Engine* сборщик мусора (Garbage Collector), следящий за памятью, связанной с некоторыми основными типами объектов движка и требующий определенного подхода к разработке для корректной работы [16]. За памятью, выделенной другим объектам (не наследникам UObject) все еще необходимо следить вручную.

Рисунок 42. Логотип Git

Кроме перечисленных инструментов также использовался **Git** в качестве системы контроля версий [19]. Это завершает разбор инструментов, применяемых в разработке *Dark Lab*. Далее к архитектуре проекта.

3.2. Архитектура приложения

В этом разделе подробно разобрано устройство программы, показаны важнейшие используемые классы, их назначение, описаны основные свойства и функции, которыми они обладают. Но перед этим представлены некоторые важные моменты, понимание которых необходимо для дальнейшего рассмотрения архитектуры.

3.2.1. Некоторые свойства Unreal Engine

Для начала про названия классов. В *Unreal Engine* существуют несколько основных классов, наследниками которых являются почти все остальные. Главными из них являются UObject и AActor. Наследниками первого является почти любой другой класс, в том числе и AActor. Это один из центральных классов системы, определяющий многие свойства объектов, используемые внутренне движком. Второй же класс представляет объекты, которые могут существовать в пространстве мира (сцены, трехмерного окружения). Все объекты на сцене обязательно являются его наследниками. При этом к **названиям классов** к началу прибавляется “A” если они наследуют от *AActor* (а значит и от *UObject*) и “U”, если не от *AActor*, но от *UObject*. Например, класс персонажа, определенный в файле “MainCharacter.h” называется *AMainCharacter*, потому что наследует от *ACharacter*, который в свою очередь наследует от *APawn*, который является наследником *AActor*.

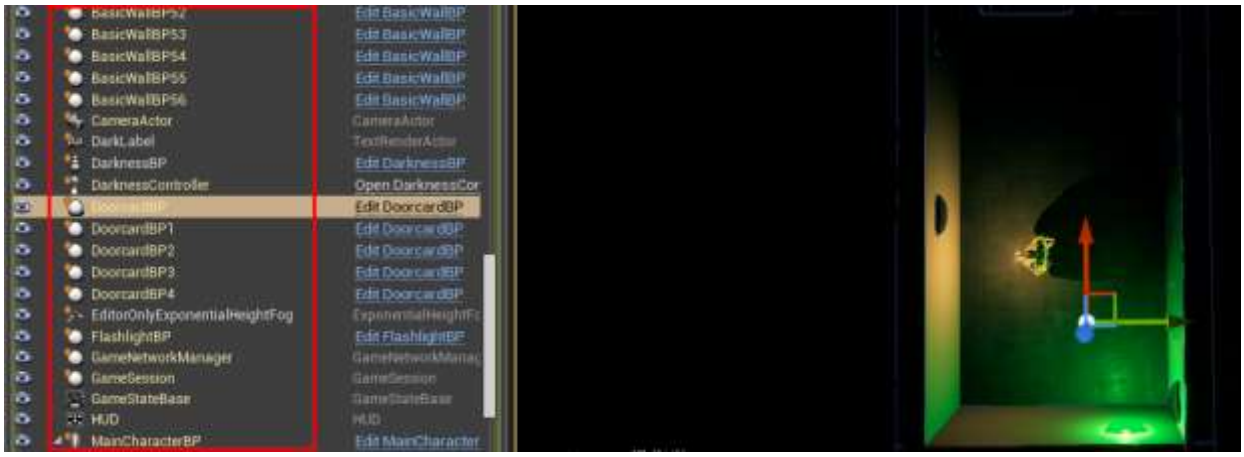

Рисунок 43. Пример AActor-ов в пространстве мира

Следующее, что важно понять, это использование в *Unreal Engine* такого ключевого паттерна как **Update Method** [50]. Он заключается в том что объекты имеют функцию, вызываемую каждый кадр, и таким образом дающую возможность применить какую-то логику к объекту перед его очередной отрисовкой на экране с учетом прошедшего с прошлого кадра времени (*deltaTime*). Это позволяет добиться постоянных изменений объекта, но избегает *лишних* изменений между кадрами, которые игрок даже не увидит. Для многих реализованных классов логика вызывается именно из таких методов.

Еще одним источником вызовов логики является, конечно же, **пользовательский ввод**. Определенные классы-контроллеры подписываются на команды ввода, задаваемые заранее в редакторе разработчиком, и таким образом отлавливают такие события как, например, «использование предмета», «открытие меню», или имеют доступ к таким параметрам как «значение по оси «движение X»». Такое абстрагирование назначения команды от конкретных клавиш позволяет проще настроить управление сразу на несколько устройств, потому в *Dark Lab* играть можно как клавиатурой с мышью, так и посредством игрового контроллера.

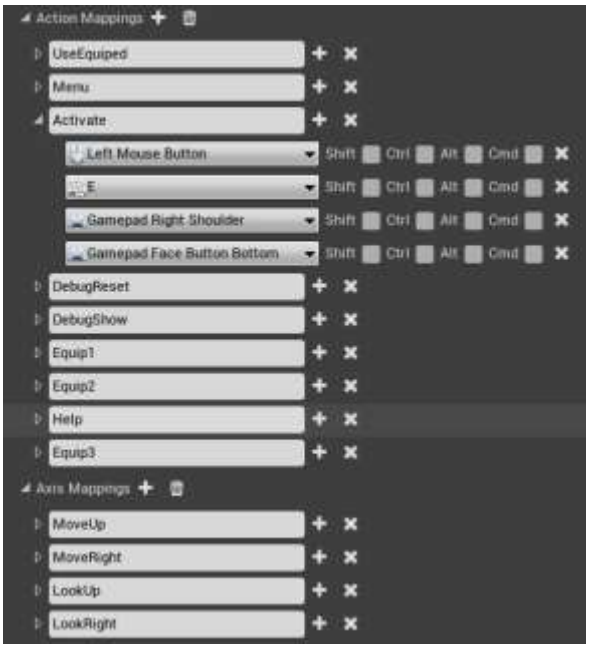

Рисунок 44. Настройка команд ввода в редакторе

3.2.2. Основные классы

Переходим к описанию классов. Все классы, структуры, перечисления и интерфейсы представлены на диаграмме классов (рис. 45).

В целом все классы можно разделить на **пять групп**:

1) Управляющие игрой классы: *AMainGameMode* и *AMenuGameMode*;
2) Две пары «контроллер»-«подконтрольный»:
   a. *AMainPlayerController* и *AMainCharacter*;
   b. *ADarknessController* и *ADarkness*;
3) Геометрия лаборатории: единицы карты *LabRoom* и *LabPassage*, перечисление *EDirectionEnum* и структура *FRectSpaceStruct*;
4) Классы пользовательского интерфейса: *UGameHUD, UMenuHUD*;
5) И самая большая группа – классы и интерфейсы связанные с объектами лаборатории, о которой речь пойдет отдельно в пункте 3.2.3.

Также есть класс *UCustromCharacterMovementComponent,* применяемый персонажем, но этот класс не требует полноценного рассмотрения, так как существует лишь для обхода внутренней ошибки в движке, а потому больше упоминаться не будет.

Каждый класс из всех групп, кроме последней, независим от всех остальных в плане наследования. Они все наследуют какой-то базовый *Unreal Engine* класс (курсивом на диаграмме классов) и дальше расширяют его функции и свойства. Внутренние наследование и реализация интерфейсов представлены среди классов пятой группы, о которой рассказано в пункте 3.2.3.

Так как *AMainGameMode* охватывает слишком много аспектов программы и связан с большой частью других классов, начнем рассмотрение не с первой, а с третьей группы, затем вернемся к первой и далее уже по порядку обратимся ко второй, четвертой и пятой (в следующей части главы). Для начала пара слов о структуре *FRectSpaceStruct* и о перечислении *Direction*.

FRectSpaceStruct – это базовая структура, представляющая прямоугольное пространство на клетчатом поле. Она обладает такими параметрами как координаты левого нижнего угла и размеры по горизонтали и по вертикали. Эта структура используется прежде всего там, где необходимо учитывать какое-то пространство на сетке без использование конкретных объектов.

EDirectionEnum – это тип-перечисление, задающий направление на клетчатой карте. Направления может быть всего четыре: вверх, вниз, влево и вправо. Этот тип используется для обозначения направления у проходов и объектов в лаборатории.

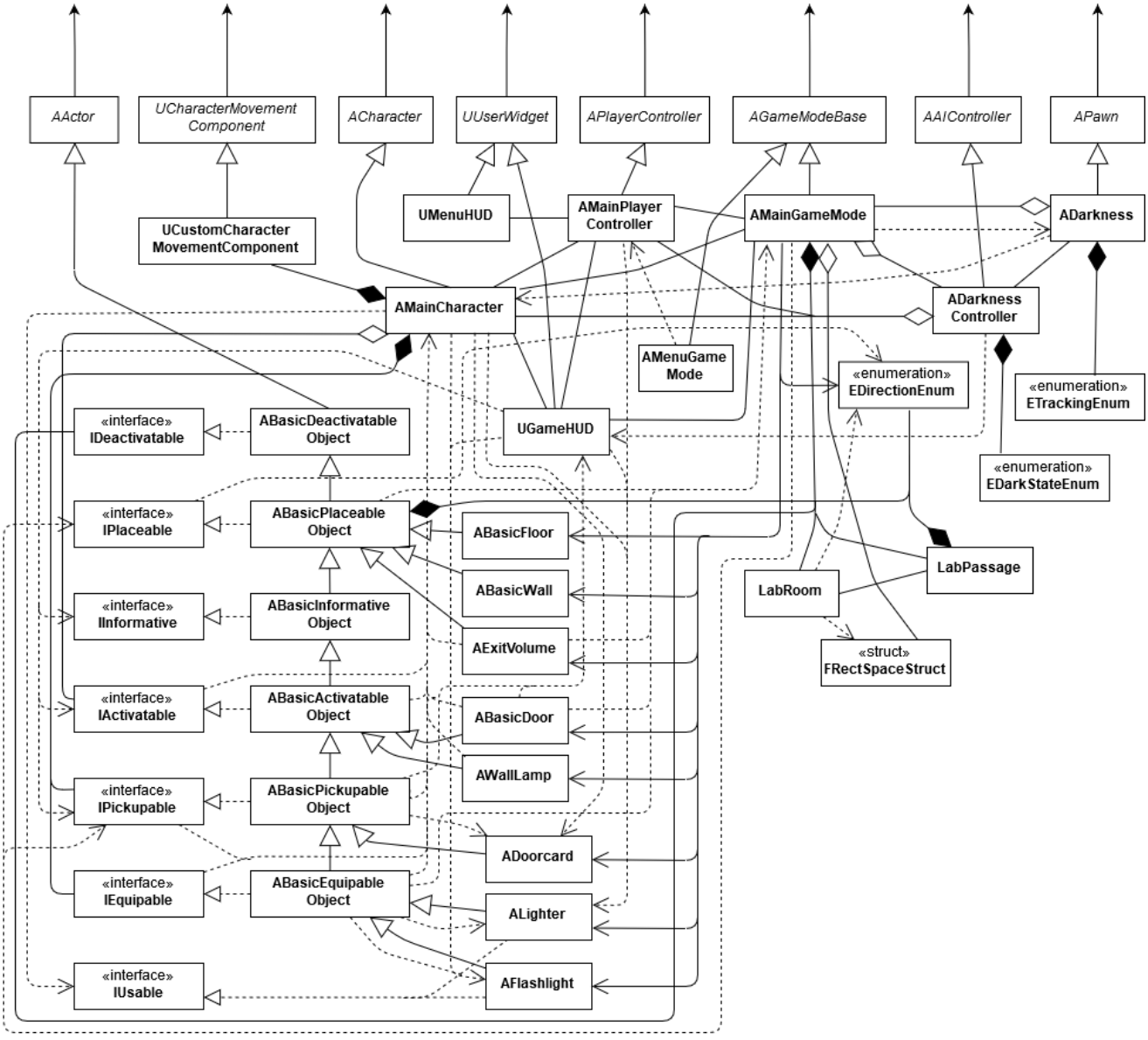

Рисунок 45. Диаграмма классов в Dark Lab. Уходящие в пустоту стрелки наверху показывают, что наследование встроенных классов Unreal Engine продолжается там

О структуре **единиц карты** уже рассказано в пункте 2.3.1 во второй главе. Реализация точно следует задуманной структуре. Карта лаборатории состоит из комнат *LabRoom* и проходов *LabPassage*. В отличие от всех остальных классов в проекте, эти не являются производными от каких-то стандартных классов *Unreal Engine*.

LabRoom, как уже объяснено, представляет базу комнаты. Она включает целые координаты комнаты на сетке карты, размеры комнаты и все проходы, ведущие в или из этой комнаты. Основные функции класса служат для добавления к комнате проходов разными способами, а деструктор следит за их удалением.

```
// Represents a room in the laboratory
class DARKLAB_API LabRoom
{
public:
    // Room's location and size
    int BotLeftX = 0;
    int BotLeftY = 0;
    int SizeX = 4;
    int SizeY = 4;

    // Passages from this room
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Room")
    TArray<LabPassage*> Passages;
```

Рисунок 46. Основные параметры LabRoom

LabPassage – это соединение между двумя комнатами. Это может быть дверь какого-то цвета или обычный проход. У прохода есть положение на карте и направление (*EDirectionEnum*). Проход существует, пока соединен хотя бы с одной комнатой (либо «из», либо «в» которую он ведет). LabPassage не обладает особыми методами.

```
// Represents a pass between two rooms in the laboratory
class DARKLAB_API LabPassage
{
public:
    // Passage's location and width
    int BotLeftX = 0;
    int BotLeftY = 0;
    int Width = 2;

    // Direction of the passage (not along its width but along player's path)
    EDirectionEnum GridDirection;

    // TODO make into a comprehensive enum instead of bool
    // True if there's a door
    bool bIsDoor = false;

    // The color of the door if it's a door. This also serves as code
    UPROPERTY(VisibleAnywhere, BlueprintReadWrite, Category = "Passage")
    FLinearColor Color = FLinearColor::White;

    // What the passage connects
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Passage")
    LabRoom* From;
    UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Passage")
    LabRoom* To;
```

Рисунок 47. Основные параметры LabPassage

Объекты и *LabRoom*, и *LabPassage* создаются, хранятся, управляются и удаляются из *AMainGameMode*, где образуют вместе карту лаборатории.

AMainGameMode отвечает за бо**л**ьшую часть процессов в игре. Это самый большой, самый сложный и самый функциональный класс, предоставляющий множество функций, используемых другими классами, и следящий за состоянием игры и ее элементов. Но самой главной задачей *AMainGameMode* является создание и изменение карты игры.

Для решения этой задачи в классе реализованы функции, соответствующие описанному в 2.3.2 алгоритму генерации, такие как *ReshapeAllDarkness*, *FixRoom*, *ExpandInDepth*, *ExpandRoom*, *SpawnFillInDepth*, *SpawnRoom*, *FillRoom* и другие. Множество функций существуют и на более низком уровне, например, для спауна конкретного объекта (*SpawnFlashlight*, *SpawnBasicWall* и др.), для проверки пересечения комнат (*MapSpaceIsFree*) и прочее.

В соответствии с описанным ранее паттерном Update Method в классе на каждом кадре проверяется расположение персонажа, после чего специальными методами определяется его комната и в случае несовпадении новой комнаты с предыдущей (что проверяется), вызывается *OnRoomEnter*, где чаще всего и происходит генерация карты.

```
// Called every frame
void AMainGameMode::Tick(const float deltaTime)
{
    Super::Tick(deltaTime);

    // Updates PlayerRoom, calls OnEnterRoom
    GetCharacterRoom();
```

Рисунок 48. Основа генерации карты

Из других важных функций можно выделить *CanSee*, определяющую видимость одной точки из другой, *GetLightingAmout*, возвращающую уровень освещенности, *WorldToGrid* и *GridToWorld*, переводящие одну систему координат в другую и функции вроде *PlaceObject*, *PoolObject*, позволяющие размещать объекты на карте и удалять их с нее.

Класс *AMainGameMode* является наследником *AGameModeBase* – класса, определяющего базовые характеристики и компоненты игры, такие как стандартного контроллера и управляемого персонажа, поэтому помимо свойств связанных с картой, ее генерацией и с пулами (о которых рассказано в 3.3.1.1.) среди параметров *AMainGameMode* можно выделить указатели на контроллеры *AMainPlayerController* и *ADarknessController*, а также на пользовательский интерфейс *UGameHUD*, о чем речь пойдет далее.

```
    // Pointers to existing controllers and HUD
    UPROPERTY()
    class ADarknessController* DarknessController;
    UPROPERTY()
    class AMainPlayerController* MainPlayerController;
public:
    UPROPERTY()
    class UGameHUD* GameHUD;
```

Рисунок 49. Важные указатели в AMainGameMode

Но прежде кратко упомянем еще одного наследника *AGameModeBase* – **AMenuGameMode**. Этот класс не обладает какими-то особыми свойствами и всего лишь занимает место *AMainGameMode* на карте с главным меню, где весь функционал основного режима игры не нужен. На деле в коде этого класса на данный момент даже ничего не реализовано, однако, от него унаследован *AMenuGameModeBP* – класс-blueprint, о которых будет сказано в 3.3.2. Далее переходим к контроллерам и контролируемым.

Первая пара – *AMainPlayerController* и *AMainCharacter*. Начнем со второго.

AMainCharacter наследует от *ACharacter*. Оба класса представляют персонажа на поле, имеющего некоторые стандартные характеристики, такие как физическую капсулу, возможность движения и прочее. Кроме того в *AMainCharacter* встроена камера, с которой игрок наблюдает сцену. Еще персонаж обладает инвентарем – массивом *IPickupable* объектов – и снаряженным *IEquipable* предметом. Отдельно реализована проверка наличия у персонажа ключа-карты определенного цвета, чтобы другие классы легко имели доступ к этой информации.

В этом классе определены функции, вызываемые в дальнейшем контроллером, нацеленные на совершение действий персонажем (рис.50). Также в классе проверяются пересечения с активируемыми объектами на карте (двери, подбираемые предметы) и есть метод для получения ближайшего самого приоритетного из них, где приоритет определен одновременно расстоянием до персонажа и близостью к центру. Активируемый объект подсвечивается отсюда.

```
// The main character of the game
UCLASS(Blueprintable)
class DARKLAB_API AMainCharacter : public ACharacter
{
    GENERATED_BODY()

public:
    // Movement
    void Move(FVector direction, const float value);
    void MoveUp(const float value);
    void MoveRight(const float value);
    void Look(const FVector direction);

    // Other controls
    void UseEquiped();
    void Activate();
    void Equip1();
    void Equip2();
```

Рисунок 50. Некоторые функции AMainCharacter

AMainPlayerController – это производный класс *APlayerController*. Если его родитель отвечает за управление *ACharacter*, то *AMainPlayerController* управляет *AMainCharacter*. Главная задача контроллера – отлавливание и обработка пользовательского ввода и дальнейшее применение его, например, вызовом показанных выше функций *AMainCharacter*. Кроме того, контроллер следит за состоянием персонажа, проверяет, когда случается поражение в игре. Кроме персонажа контроллер также управляет пользовательским интерфейсом (классы *UGameHUD* и *UMenuHUD)*, причем, как в основном режиме игры, так и в главном меню (где работает *AMenuGameMode*). Более того, именно отсюда эти интерфейсы создаются на старте игры.

```
InputComponent->BindAxis("MoveUp", this, &AMainPlayerController::MoveUp);
InputComponent->BindAxis("MoveRight", this, &AMainPlayerController::MoveRight);

InputComponent->BindAxis("LookUp");
InputComponent->BindAxis("LookRight");

InputComponent->BindAction("UseEquiped", IE_Pressed, this, &AMainPlayerController::UseEquiped);
InputComponent->BindAction("Activate", IE_Pressed, this, &AMainPlayerController::Activate);
InputComponent->BindAction("Equip1", IE_Pressed, this, &AMainPlayerController::Equip1);
InputComponent->BindAction("Equip2", IE_Pressed, this, &AMainPlayerController::Equip2);
InputComponent->BindAction("Equip3", IE_Pressed, this, &AMainPlayerController::Equip3);

InputComponent->BindAction("Menu", IE_Pressed, this, &AMainPlayerController::ShowHideMenu);
InputComponent->BindAction("Help", IE_Pressed, this, &AMainPlayerController::ShowHideHelp);

InputComponent->BindAction("DebugReset", IE_Pressed, Cast<AMainGameMode>(GetWorld()-
  >GetAuthGameMode()), &AMainGameMode::ResetMap);
InputComponent->BindAction("DebugShow", IE_Pressed, Cast<AMainGameMode>(GetWorld()-
  >GetAuthGameMode()), &AMainGameMode::ShowHideDebug);
```

Рисунок 51. Подключение контроллера к определенным ранее командам и осям пользовательского ввода (как в пункте 3.2.1.)

Следующие из второй группы классов – пара *ADarkness* и *ADarknessController*.

ADarkness является наследником *APawn* – класса, представляющего что угодно управляемое (*ACharacter* также является *APawn*). Этот класс отвечает за «тьму» и ее прямые действия, физическое воплощение, но не за поведение.

Предоставляются различные функции, которые вызываются из контроллера «тьмы», такие как движение, телепортация, страх света и другие. Здесь же проводятся проверки освещения с использованием функций *AMainGameMode*.

```
// The darkness that hunts the player
UCLASS(Blueprintable)
class DARKLAB_API ADarkness : public APawn
{
    GENERATED_BODY()

public:
    // Movement
    void Move(const FVector direction);
    void MoveToLocation(FVector location);
    void MoveToActor(AActor* actor);
    void Stop();
    void TeleportToLocation(FVector locaton);
    // When light is too strong goes backwards and returns true
    bool RetreatFromLight();
    // Tracks something
    void Tracking();
    // Goes away from last brightest light
    void IntoDarkness();
```

Рисунок 52. Некоторые функции ADarkness

Кроме того, обработка настижения «тьмой» персонажа происходит именно в этои классе и в случае подобного события персонаж убивается, а контроллер «тьмы» получает оповещение.

Во время охоты «тьма» может преследовать одну цель и для этого ей испольщется **ETrackingEnum** с возможными значениями None, Actor и Location, соответственно описывающими тип цели, за которой ведется охота.

Базовым классом **ADarknessController** – контроллера «тьмы» – является *AAIController*. Это стандартный класс для реализации управляющего модуля, контролирующего *APawn* автоматически. В данном случае подконтрольным является *ADarkness*. В контроллере реализована логика поведения «тьмы», описанная в 2.4. Для состояний тьмы используется **EDarkStateEnum**. Основная обработка логики в классе происходит на каждом кадре в методе Tick.

```
switch (State)
{
case EDarkStateEnum::VE_Passive:
    // If its been some time, start the hunt
    if (SinceLastStateChange >= CurrentMaxTimePassive)
        StartHunting();
    // TODO else?
    break;
case EDarkStateEnum::VE_Hunting:
    // If hunting for some time, start retreating
    if (!bIsPersistent &&
        SinceLastStateChange >= CurrentMaxTimeHunting)
        StartRetreating();
    // Otherwise keep hunting
    else
    {
        UE_LOG(LogTemp, Warning, TEXT("Hunting"));
        // Tries to teleport if possible
        TeleportToCharacter();
        UE_LOG(LogTemp, Warning, TEXT("> After teleport"));
        // Then just moves
        Darkness->Tracking();
        UE_LOG(LogTemp, Warning, TEXT("> After tracking"));
    }
    break;
case EDarkStateEnum::VE_Retreating:
    // If retreating for too long or
    // if already escaped into darkness, become passive
    if (SinceLastStateChange >= MaxTimeRetreating ||
        Darkness->TimeInDark >= MinTimeInDark)
        BecomePassive();
    // Otherwise keep retreating
    else
        Darkness->IntoDarkness();
    break;
}
```

Рисунок 53. Базовое поведение "тьмы" (когда нет испуга от света)

Наконец, классы четвертой группы: **UMenuHUD** и **UGameHUD**. Оба наследуются от *UUserWidget* – класса, определяющего пользовательские элементы интерфейса. Оба, соответственно, представляют собой этот интерфейс и его логику. К обоим обращается *AMainPlayerController* для управления ими. Главная их разница в том, что первый отвечает за интерфейс в главном меню игры, а второй за интерфейс основного режима (об обоих подробнее в 3.4.2.). Помимо этого важно отметить, что многие классы обращаются к методу *ShowHideWarning* класса *UGameHUD* для вывода на экран важного для игрока сообщения. К примеру, о том что дверь не может быть открыта.

```
// Call to show game end messages
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void ShowVictoryMessage();
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void ShowLossMessage();

// Shows or hides a warning
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void ShowHideWarning(bool show, const FText& warning);

// Shows or hides menu
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void ShowHideMenu(bool show);
// Shows or hides help
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void ShowHideHelp(bool show);

// Shows overlay during transition
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void OnRestart();
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void OnChangeMap();
UFUNCTION(BlueprintImplementableEvent, Category = "Game HUD")
void OnExit();
```

Рисунок 54. Пример функций в UGameHUD. Эти функции объявлены, но не реализованы в коде, вместо этого их реализация представлена в UGameHUDBP – blueprint-классе-наследнике (подробнее в 3.3.2)

Далее отдельно рассмотрены классы и интерфейсы объектов лаборатории.

3.2.3. Объекты на уровне

Бо**л**ьшая часть классов, участвующих в реализации программы, используются для создания объектов лаборатории, таких как стены, двери, ключи-карты и других. Реализована разветвленная система интерфейсов и базовых классов, что позволяет легко расширять проект в будущем, добавляя классы с необходимыми свойствами.

На рис.55 представлена часть общей диаграммы классов (рис.45), включающая только пятую группу классов (группы определены в 3.2.1) и связанные с ними классы. Начнем их разбор двигаясь сверху-вниз.

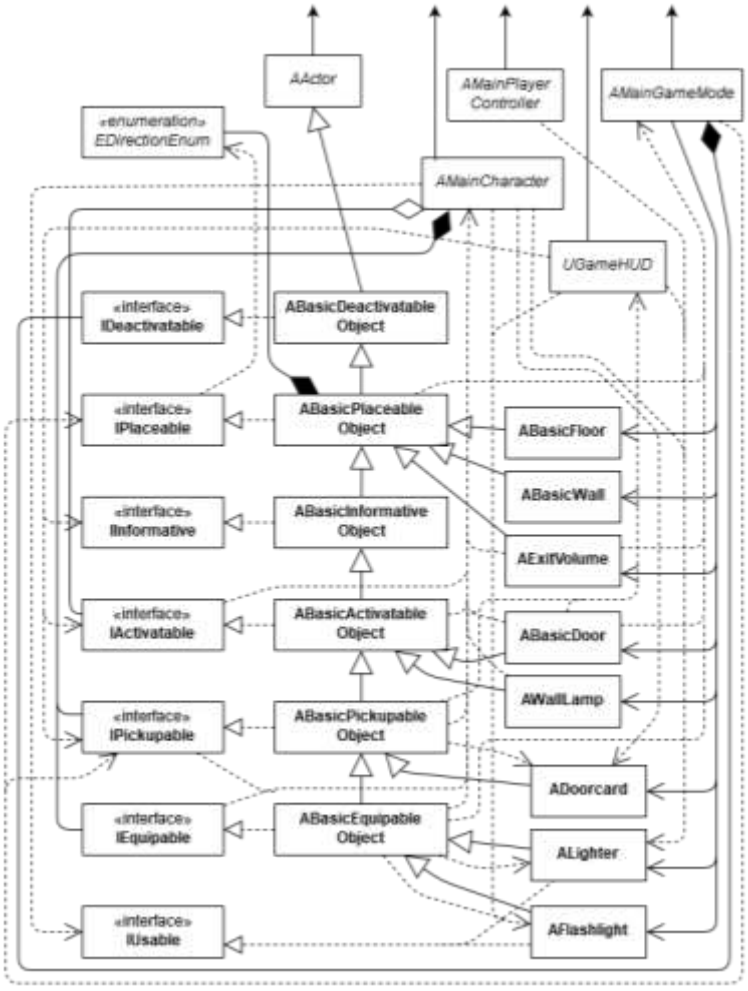

Рисунок 55. Диаграмма классов для классов и интерфейсов, связанных с объектами лаборатории в Dark Lab

IDeactivatable – интерфейс, определяющий такие функции как *SetActive* и *IsActive*. Он служит для обозначения объектов которые можно выключать, не удаляя, например, для pooling (3.3.1.1.). Реализует этот интерфейс **ABasicDeactivatableObject**. Этот класс наследуется напрямую от *AActor* и реализует базовый функционал активации/деактивации объекта. При деактивации объект прежде всего скрывается от игрока (*SetActorHiddenInGame*), далее у него отключается коллизия (collision, область в пространстве, отвечающая за столкновения с другими подобными областями для физических и других проверок) и, наконец, у объекта отключается вызов метода Tick на каждом кадре (*SetActorTickEnabled*). Соответственно, все это включается снова при активации объекта.

IPlaceable – интерфейс для объектов лаборатории, которые можно поместить на клетчатую сетку, задав определенные положение, размер и направление (*EDirectionEnum*). Интерфейс определяет функции *SetSize*, *SetSizeXY*, *GetSize* и *Place*. Базовая реализация интерфейса – класс **ABasicPlaceableObject**, наследующий от *ABasicDeactivatableObject*. В нем объявлены параметры BaseSize (базовый размер объекта, на основе которого считается реальный размер при изменении размера и помещении на карту), GridDirection (направление, в которое

«смотрит» объект) и ZOffset (константа, используемая при помещении объекта на плоскость, смещающего его на определенную величину по оси Z). При помещении объекта на карту используется функция *GridToWorld* класса *AMainGameMode* для определенного точного положения в мировых координатах.

```
// Places the object on the map, using bottom left corner
void ABasicPlaceableObject::Place_Implementation(const
 FIntVector botleftloc, const EDirectionEnum direction)
{
    GridDirection = direction;
    FIntVector size = Execute_GetSize(this);

    FVector location;
    AMainGameMode::GridToWorld(botleftloc.X, botleftloc.Y,
       size.X, size.Y, location.X, location.Y);
    location.Z = ZOffset;
    SetActorLocation(location);

    float rotation;
    switch (GridDirection)
    {
    case EDirectionEnum::VE_Up:
        rotation = 0;
        break;
    case EDirectionEnum::VE_Right:
        rotation = 90;
        break;
    case EDirectionEnum::VE_Down:
        rotation = 180;
        break;
    case EDirectionEnum::VE_Left:
        rotation = -90;
        break;
    }

    SetActorRotation(FQuat(FVector(0,0,1),
      FMath::DegreesToRadians(rotation)));
}
```

Рисунок 56. Реализация Place

Три объекта лаборатории наследуются напрямую от *ABasicPlaceableObject*: **ABasicFloor**, представляющий полы на карте, **ABasicWall**, представляющий обычные стены и **AExitVolume** – выход из лаборатории. Первые два не таят в себе никакой логики и лишь физически представляют объекты на карте, а потому не требуют дальнейшего рассмотрения. *AExitVolume* же имеет несколько важных элементов. Во-первых, свет. Этот объект обычно помещается за выходную дверь и, когда та открывается, внутри включается свет, разливаясь по комнате через дверной проем. Для включения света есть метод *ActivateLight*. Во-вторых, коллизия. Отлавливается событие ее пересечения с персонажем и если не возникло ошибок, объявляется победа игрока.

Следующий интерфейс – **IInformative**. Он используется на объектах, которые могут сообщить базовую информацию о себе через методы *GetName* и *GetBasicInfo*. Реализуется этот интерфейс классом **ABasicInformativeObject**, наследующим *ABasicPlaceableObject*. Базовая информация и имя объекта хранятся в полях класса и доступны через реализации методов интерфейса. Наследники этого класса должны изменять поля, но не трогать методы.

IActivatable – один из важнейших интерфейсов, описывающий объекты, обладающие определенным собственным встроенным функционалом, который можно активировать. Эта «активация» не связана с «активацией/деактивацией» интерфейса *IDeactivatable*. Активируемые объекты могут быть активированы двумя способами: напрямую (персонажем) или косвенно (иначе). Считается, что результат активации может зависеть от способа, а так же необязательно, что оба способа доступны. Поэтому интерфейс определяет четыре

```
// Called when the object is activated by character
void ABasicActivatableObject::Activate_Implementation(AMainCharacter * character)
{
    if (!bActivatableDirectly)
        return;

    ActivateObject(character);
}
// Called when the object is activated indirectly
void ABasicActivatableObject::ActivateIndirectly_Implementation()
{
    if (!bActivatableIndirectly)
        return;

    ActivateObject(nullptr);
}
// Returns true if activatable directly
bool ABasicActivatableObject::IsActivatableDirectly_Implementation()
{
    return bActivatableDirectly;
}
// Returns true if activatable indirectly
bool ABasicActivatableObject::IsActivatableIndirectly_Implementation()
{
    return bActivatableIndirectly;
}

// Called when the object is activated (should always be overridden)
void ABasicActivatableObject::ActivateObject(AMainCharacter * character)
{
    UE_LOG(LogTemp, Warning, TEXT("Activated"));
}
```

Рисунок 57. Реализация функций IActivatable в ABasicActivatableObject

функции: *Activate (Directly), ActivateIndirectly, IsActivatableDirectly, IsActivatableIndirectly*. **ABasicActivatableObject**, наследующийся от *ABasicInformativeObject*, реализует *IActivatable* с упрощением функционала. Класс вводит параметры bActivatableDirectly и bActivatableIndirectly для определения, как объект можно активировать (и их поменяют, если понадобится, наследники), но для обоих вариантов активации вызывает одну функцию *ActivateObject*, которая должна перегружаться производными классами (рис.57).

```
// Called when the object is activated
void AWallLamp::ActivateObject(AMainCharacter * character)
{
    // UE_LOG(LogTemp, Warning, TEXT("Toggled lamp"));

    Light->ToggleVisibility();
    UpdateMeshColor(Light->IsVisible() ? Color : FLinearColor::Black);
}
```

Рисунок 58. Простой пример активации объекта

В итоге *ABasicActivatableObject* представляет объекты, которые можно выключать, можно помещать на карту и можно активировать, используя их встроенный функционал. Они также способны предоставлять информацию о себе. Такими объектами являются предметы, которые можно поднять (о них позже), а также двери и лампы в лаборатории. Поэтому **ABasicDoor** и **AWallLamp** являются производными классами *ABasicActivatableObject*. Первый представляет собой открываемую дверь, имеющую определенный цвет и знающую, выход ли она. При ее активации прежде всего проверяется, может ли персонаж открыть эту дверь через *HasDoorcardOfColor* класса *AMainCharacter*. Если может, дверь открывается (или закрывается). Второй класс – *AWallLamp* – представляет настенные лампы в лаборатории. Они, как и дверь, обладают цветом, а также имеют проверку включенности (*IsOn*). Объекты этого класса не могут быть активированы напрямую, только косвенно. При активации свет включается или выключается, кроме света так же обновляется цвет модели лампы.

Следующий интерфейс – **IPickupable**. Его реализуют объекты, которые персонаж может поднять и поместить в инвентарь (поэтому инвентарь в AMainCharacter – это массив *IPickupable*). Интерфейс определяет всего одну функцию – *PickUp*. Реализует *IPickupable* и наследуется от *ABasicActivatableObject* класс **ABasicPickupableObject**. Его объекты – это активируемые предметы лаборатории, которые персонаж может поднять (поэтому активируемы они только напрямую). Соответственно в классе реализованы *PickUp* и *ActivateObject* и первый вызывается из второго (рис.59).

Рисунок 59. Реализация ABasicPickupableObject

Напрямую от *ABasicPickupableObject* наследуется **ADoorcard**. Это то же самое, что и keycard – ключ-карта для открытия дверей в лаборатории. Главный ее параметр (помимо всех унаследованных) это цвет, который влияет как на то, как выглядит карта, так и на то, какие двери ей можно открыть (того же цвета).

IEquipable – интерфейс для объектов, которые персонаж может «надеть», то есть экипировать или взять в руки, *Equip*. Можно этот объект и снять, убрать в карман, *Unequip*. **ABasicEquipableObject** реализует этот интерфейс. Он также наследует *ABasicPickupableObject,* ведь обычно надеваемые объекты должны быть сначала подняты. Этот класс не добавляет параметры, а лишь реализует *Equip, Unequip* и перегружает *ActivateObject*. Первая функция добавляет объект в EquipedObject у персонажа и прикрепляет его к нему, вторая наоборот очищает EquipedObject и выключает объект (используя функцию из *IDeactivatable*). Третья же первым делом поднимает объект как в *ABasicPickupableObject,* а затем, если персонаж не экипирован, экипирует предмет. Также для фонаря и для зажигался при поднятии вызывается *ResetPowerLevel,* восстанавливающая энергию или газ. Но прежде чем говорить про эти два предмета, разберем последний интерфейс – *IUsable*.

```
// Called when the object is to be unequiped
void ABasicEquipableObject::Unequip_Implementation(AMainCharacter* character)
{
    // We disable the object
    Execute_SetActive(this, false);

    // No need to destroy it
    character->EquipedObject = nullptr;

    UE_LOG(LogTemp, Warning, TEXT("Unequiped %s"), *(Name.ToString()));
}
```

Рисунок 60. Реализация Unequip в ABasicEquipableObject

IUsable объекты могут быть каким-то образом использованы. Идеологически во многом он идентичен с *IActivatable,* но считается что *IUsable* применим для предметов, которыми уже обладает персонаж, а не для объектов в лаборатории, а *IActivatable* наоборот. Интерфейс определяет единственную функцию – *Use*. В отличие от других интерфейсов не существует базового объекта, определяющего стандартный функционал для *Use,* поэтому этот интерфейс отдельно реализуют разные объекты.

AFlashlight и **ALighter** – фонарик и зажигалка – оба являются производными классами от *ABasicEquipableObject,* а потому они оба могут предоставлять о себе информацию и их можно деактивировать, помещать на поле, поднимать и надевать. Так же они реализуют интерфейс *IUsable* каждый своим (схожим) образом. Так как эти предметы в игре имеют схожую функцию (освещают пространство впереди/вокруг игрока), многие их функции и параметры совпадают. Так, использование обоих приводит к включению/выключению света, а для того чтобы понять, включен ли предмет, у обоих есть функция *IsOn* как у лампы. Кроме того, оба со временем теряют

PowerLevel (заряд или газ) и их свечение в связи с этим становится слабее. У обоих есть методы *ResetPowerLevel*. В основном разница между объектами визуальная. Используются разные объекты, разный цвет света, разная его форма и т.п. Кроме того, в зажигалке реализовано случайное изменение яркости для того, чтобы создать иллюзию огня (рис.61).

```
// Called every frame
void ALighter::Tick(const float deltaTime)
{
    Super::Tick(deltaTime);

    if (Light->IsVisible())
    {
        PowerLevel -= PowerLossPerSecond * deltaTime;
        PowerLevel = PowerLevel > 0.f ? PowerLevel : 0.f;
        if (PowerLevel != 0.f)
        {
            float bottomLine = PowerLevel / 2.f;
            float topLine = PowerLevel;

            float min = (bottomLine + CurrentlySeenLevel) / 2.f;
            float max = (topLine + CurrentlySeenLevel) / 2.f;

            CurrentlySeenLevel = FMath::FRandRange(min, max);
        }
        else
            CurrentlySeenLevel = 0.f;
        Light->SetLightColor(NormalColor * CurrentlySeenLevel);

        if (PowerLevel == 0.f)
        {
            UE_LOG(LogTemp, Warning, TEXT(" %s lost all gas"), *(Name.ToString()));
            Light->ToggleVisibility();
        }
    }
}
```

Рисунок 61. Случайное изменение яркости зажигалки

Это были последние классы из пятой группа и на этом завершен разбор архитектуры в целом. Как уже было сказано, такая система классов и интерфейсов в первую очередь необходима для гибкого расширения проекта в будущем. Далее рассказано про некоторые особенности реализации *Dark Lab*.

3.3. Особенности реализации

В данном разделе представлены важные особенности разработанной программы. Первым делом описаны некоторые детали реализации алгоритмов, повлиявшие положительно на производительность продукта. Затем рассказано про используемую в *Unreal Engine* технологию Blueprints и про ее применение в *Dark Lab*.

3.3.1. Оптимизации спауна

Последняя фаза прохода цикла генерации карты – spawn – ощутимо сказывается на производительности всего алгоритма. Важно отметить, что под **производительностью** тут имеется в виду относительное время работы и, соответственно, под **оптимизацией** – уменьшение относительного времени работы, то есть увеличение производительности. Так, если алгоритм А работает в среднем быстрее чем алгоритм Б при достижении схожих с точки зрения игрока результатов, то первый алгоритм будем называть более оптимизированным.

Учитывая влияние фазы spawn было принято решение оптимизировать ее, достичь лучших результатов по времени. Прежде всего цель была избежать задержек генерации, чтобы игрок не ощущал «лагов» в процессе игры. Для этого были применены две различные техники: pooling и «умный» выбор комнат для спауна.

3.3.1.1. Pooling

Object Pool или **Pooling** – это оптимизационный паттерн проектирования, нацеленный на увеличение производительности и уменьшение использования памяти за счет переиспользования уже созданных, но уже ненужных старых объектов вместо создания новых объектов каждый раз, когда они оказываются нужны [26]. Иногда этот паттерн применяют не для повышения производительности, а для избегания фрагментации памяти. В нашем же проекте важнее эффект применения пулинга на время создания и удаления объектов.

В *Dark Lab* очень часто происходит пересоздание карты. Каждую новую комнату необходимо заспаунить. Это включает все ее стены, двери, основной пол, пол под проходами. Затем нужно заполнить комнату. Опять же, спаун объектов: лампы, фонарики, ключи-карты. В итоге даже при спауне одной комнаты создается множество объектов. Именно поэтому pooling применяется в *Dark Lab* для **оптимизации спауна на уровне объектов**.

Именно для этой цели все объекты карты реализуют интерфейс *IDeactivatable*, описанный в 3.2.3. Благодаря ему эти объекты можно не удалять (*Destroy*), а выключать, чтобы игрок просто не мог с ними взаимодействовать или видеть их. *AMainGameMode* следит за отключенными объектами лаборатории, сохраняя их в пулы (наборы объектов для пулинга, рис.62). Он же предоставляет функции для помещения объектов в эти пулы и их деактивации, в том числе и для целых частей лаборатории, вроде комнат (рис.63).

```
// Pools
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> DefaultPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> BasicFloorPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> BasicWallPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> BasicDoorPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> WallLampPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> FlashlightPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> DoorcardPool;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Pools")
TArray<TScriptInterface<IDeactivatable>> ExitVolumePool;
```

Рисунок 62. Пулы в AMainGameMode

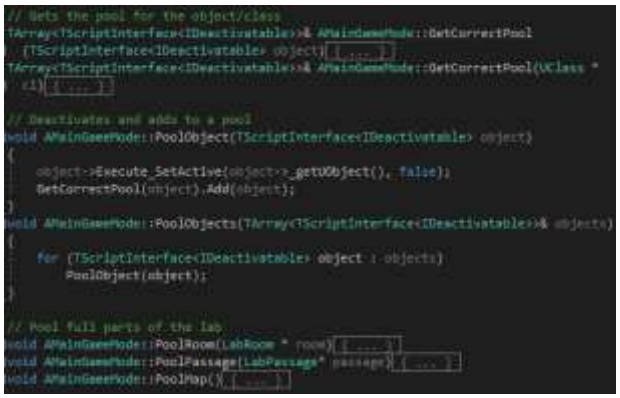

Рисунок 63. Pooling объектов и частей лаборатории

Рисунок 64. Поиск объекта в пуле перед спауном

Естественно, система не только пулит объекты, но и пользуется объектами из пула при создании новых. Когда необходимо заспаунить какой-то объект, сначала проверяется, нет ли в пуле для объектов этого класса деактивированного объекта (рис.64). Если есть, его параметры сбрасываются, затем вносятся новые и после этого его активируют. Если нет, спаунится новый объект (который позже может сам попасть в пул).

Существует также дефолтный пул для объектов любых классов, но на данный момент он не используется, потому как для всех классов, которые могут быть деактивированы, уже есть свои собственные пулы.

По итогам, система удачно справляется с задачей, ускоряя спаун отдельных объектов. Но иногда вместо того чтобы спаунить новый объект или даже доставать такой из пула, можно не создавать ничего с одинаковым результатом с точки зрения игрока. Об этом далее.

3.3.1.2. Умный выбор спауна

Изначально алгоритм спауна, описанный в пункте 2.3.2.3. работает следуя очень простому и прямолинейному принципу: мы берем комнату игрока и рекурсивно спауним все соседние с ней комнаты на заданное расстояние *n*. Соседними здесь называют любые две комнаты соединенные проходом. В итоге получается, что после фазы spawn любая комната на расстоянии *n-1* до игрока заспаунена, где расстоянием называется количество проходов, через которые нужно пройти, чтобы попасть из одной комнаты в другую. Такой результат фазы необходим, чтобы если вдруг персонаж сможет увидеть комнату на расстоянии *n-1*, он видел именно комнату, а не пустоту. Именно с расчетом на это подобрана константа *n* с учетом минимального размера любой комнаты и максимального расстояния освещения фонаря и зажигалки.

Ключевая фраза в предыдущем абзаце, на основе которой алгоритм может быть улучшен это «сможет увидеть». Да, уменьшать константу *n* нельзя дальше теоретического минимума, чтобы не показывать игроку незаспаунненное пространство. Однако, что можно сделать, так это никогда не спаунить такие комнаты, которые персонаж не увидит, как бы ни пытался и каким бы мощным фонарем ни посветил. Это **оптимизация спауна на уровне комнат**.

Рисунок 65. "Умный" спаун комнат. Глубина 4

Пример результата работы такого подхода показан на рис.65.

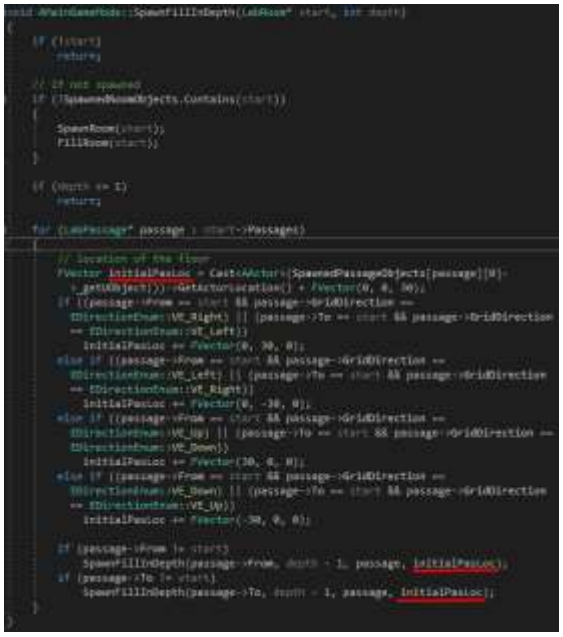

Рисунок 66. Функция рекурсивного спауна для стартовой комнаты

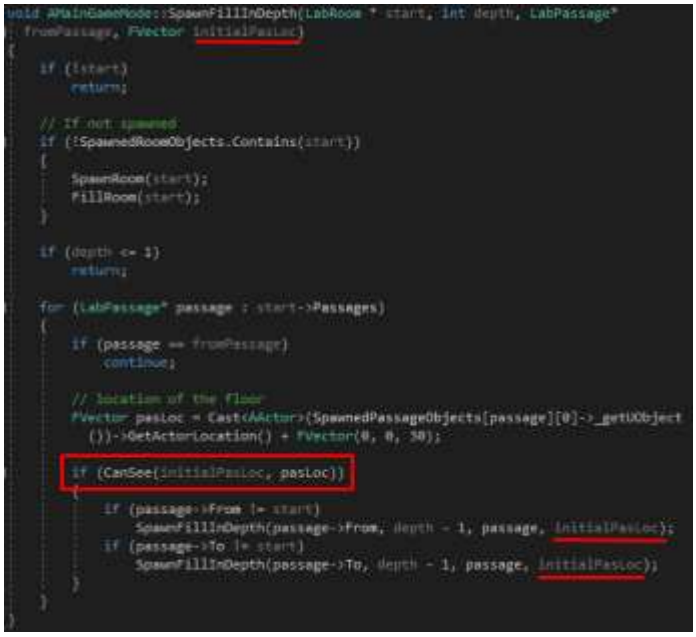

Рисунок 67. Функция рекурсивного спауна для всех последующих комнат

Идея заключается в следующем:

1) Для первой комнаты (комнаты игрока) алгоритм работает так же как обычно и распространяется на все соседние проходы, независимо от того, что и откуда видно. Но следующим комнатам при этом передается не только глубина (на 1 меньше) и предыдущий проход (для того чтобы туда не возвращаться), но также **положение стартового прохода** – координаты того прохода в стартовой комнате (игрока), из которого алгоритм распространился до этой новой комнаты (рис.66). На рис.68 стартовые – это зеленые проходы. В ходе рекурсии этот параметр остается неизменным.

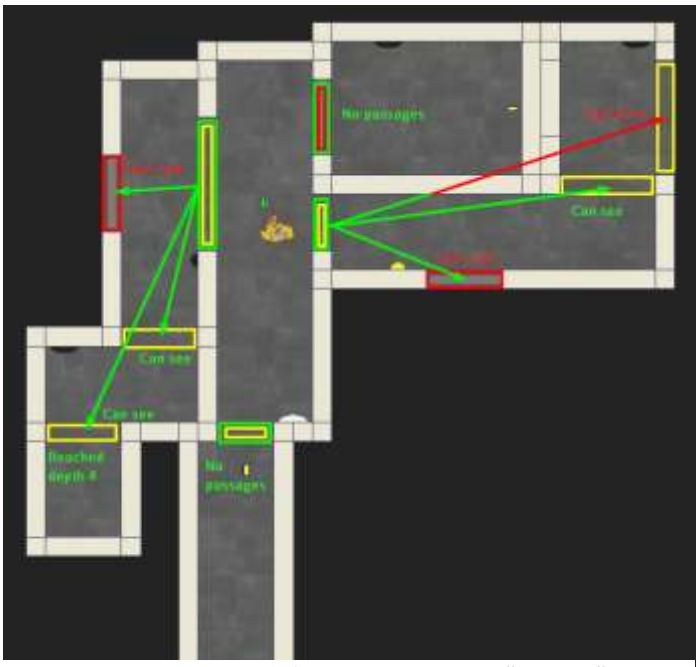

Рисунок 68. Проверка видимости во время "умного" спауна

2) Комнаты, до которых дошла рекурсия, используют обычный алгоритм, но когда выбирается по каким проходам продолжать spawn добавляется **проверка видимости** центральной точки этого выбираемого прохода из переданного положения стартового прохода (рис.67). На рис.68 стрелками показано, как осуществляется такая проверка видимости. Красными прямоугольниками выделены закрытые двери, через которые видеть невозможно. Желтыми – обычные проходы. То, что нельзя увидеть, не спаунят.

В итоге с такими ограничениями алгоритм отсекает часть комнат и работает эффективнее.

3.3.2. Связь кода и Blueprints

В ходе данной работы несколько раз упоминается такая система как **Blueprints**. В этом пункте описано, что это такое, как используется, и почему эта система является одним из сильнейших преимуществ *Unreal Engine* перед конкурентами.

Blueprints – это система визуального скриптинга, позволяющая задавать логику и настраивать элементы программы через удобный интуитивный интерфейс прямо в редакторе *Unreal Engine* [6]. Без кода это зачастую используют для прототипирования игр или для небольших проектов. Однако, что делает эту систему действительно неповторимой и мощной, так это возможность сочетать код на C++ и блупринты на разных уровнях абстракции. Программист задает б**о**льшую часть логики и архитектуры системы, после этого **наследует от своего класса блупринт** и дальше уже дизайнер или кто-то другой расширяет функционал и занимается визуальной составляющей через блупринт.

Такова идеология работы *Unreal Engine,* позволяющая совмещать сильные стороны разных членов команды разработчиков проекта.

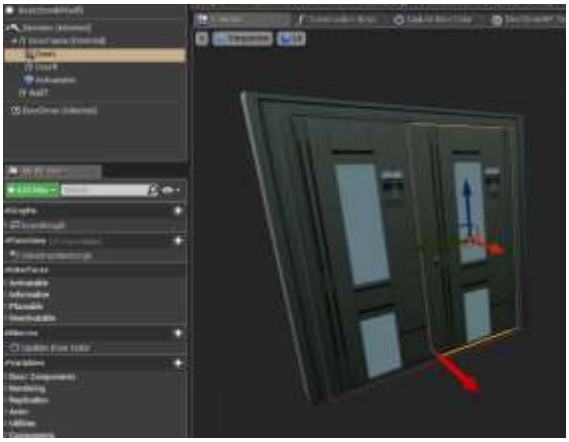

Рисунок 69. Пример редактирования внешнего вида объекта в системе Blueprint

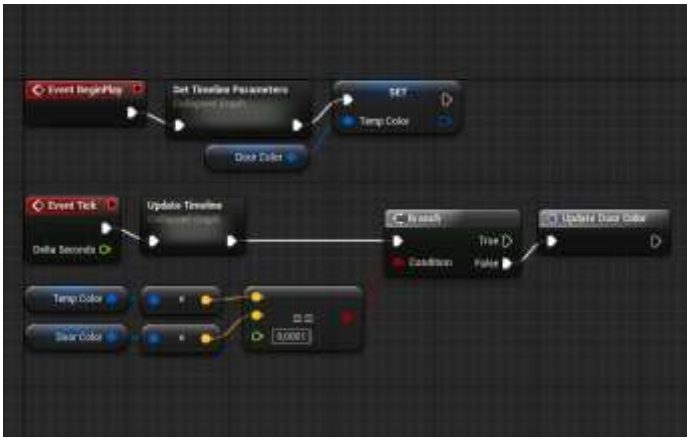

Рисунок 70. Пример редактирование функционала объекта в системе Blueprint

Несмотря на то что над *Dark Lab* работа мной велась в-одиночку, почти для всех важных классов на C++ были созданы наследники-блупринты и таким образом разделены обязанности разных модулей. Так, весь внешний вид как объектов лаборатории, так и пользовательского интерфейса задается через редакторы блупринтов (как дверь на рис.69), в то время как почти весь функционал программы, наоборот, реализован через код.

Рисунок 71. Blueprints в Dark Lab

Помимо блупринтов через редакторы движка были настроены работа анимаций, материалы некоторых объектов, система частиц «тьмы», постпроцессинг и другие детали.

3.4. Результаты работы

К этому моменту в работе уже освещены идея игры, используемые алгоритмы, особенности архитектуры и технологий, причины выбора тех или иных подходов и многое другое. Из написанного несложно понять основной функционал реализованного проекта. Поэтому в этом пункте функционал представлен другим способом, **глазами игрока**: описан через потенциальную игровую сессию с выделением важных составляющих, а не перечислением. Также в конце показан интерфейс *Dark Lab*.

3.4.1. Функционал прототипа глазами игрока

После старта игры персонаж появляется на **карте**. Система сообщает игроку уведомлением на экране (о чем подробнее в 3.4.2.), что он только проснулся, не помнит, где он и понимает лишь, что помещение похоже на лабораторию. На самом деле, на момент разработки прототипа помещение мало на что похоже, но это сюжетный, а не функциональный момент. Элементы карты, которые всегда сразу видит игрок, это **пол**, **стены** и **лампы** на стенах освещающие стартовую комнату. Временами лампы гаснут.

Рисунок 72. Игрок в стартовой комнате

Игрок сразу получает под контроль **персонажа**. При движении мыши меняется **направление взгляда**. При использовании WASD на клавиатуре персонаж начинает **двигаться**, причем, его движения анимированы. Возможно, что игрок использует контроллер для управления, вместо клавиатуры и мыши. Что игрок также замечает при старте так это наличие у его персонажа в руках **зажигалки** и белой **ключ-карты**. Он может сразу **использовать предмет** (зажигалку) в руках, включая и выключая его. Зажигалка испускает мигающий свет.

Какое-то время игрок ходит по лаборатории. Он замечает, что временами, когда он заходит в новую комнату, в ней включается освещение. Так же игрок обнаруживает **двери** в лаборатории. Когда персонаж подходит к двери, она подсвечивается по контуру. Некоторые

двери игроку удается **открыть**, другие остаются **заперты**, потому как у игрока нет нужного ключа-карты. Игрок догадывается до этого и решает, что ему нужно их искать.

Но в этот момент появляется уведомление о том, что что-то начало искать игрока. Проходит некоторое время и наконец перед игроком появляется **«тьма»**. Игрок, конечно, еще не знает, что это, но понимает, что с этим лучше не связываться. Но он также замечает, что **«тьма» замедлена на свету** и не так быстро его преследует. Все же он начинает убегать. Пока бежит, игрок натыкается на лежащий на земле продолговатый предмет. Это оказывается **фонарь**. Игрок **подбирает** его и благодаря обновившемуся интерфейсу осознает, что его можно **взять в руки**, что и делает. Игрок разворачивается к тьме с включившимся фонарем – она начинает **бояться и отходить** от игрока. Какое-то время это продолжается и игрок теснит ее. Однако со временем **фонарь теряет заряд** и его свет становится слабее. Игрок только сейчас осознает это, увидев индикатор на экране. Но оказывается слишком поздно. **«Тьма» настигает и убивает** персонажа.

Рисунок 73. Результат встречи игрока с тьмой

Игрок начинает игру заново.

Первое, что он осознает, это то что карта другая на этот раз. Он понимает, что она **случайно генерируется**. В этот раз игрок решает искать ключи-карты. Он идет куда-то, находит несколько ключей и осознает, что цветные **ключи-карты открывают двери своего цвета**.

Вдруг снова «тьма». Она выходит перед игроком. Помня, что тьма медленнее на свету, игрок начинает спиной отступать от нее, но оказывается, что движение спиной медленнее. Тогда игрок решает все-таки убежать, разворачивается и обнаруживает прямо за собой тупик там, где только что был обычный проход. Вспоминая подобные несостыковки, которые игрок невольно подмечал последние минуты, игрок понимает, что карта не просто случайно генерируется, **карта постоянно меняется в темноте**, пока игрок этого не видит. Оказавшись в тупике с «тьмой» и зная страх этой «тьмы» перед светом, игрок с включенной зажигалкой идет прямо на нее и

наблюдает, как та отлетает в стену. Стоит ему пройти мимо как тьма снова летит за ним. Однако проходит еще немного времени и **«тьма» прекращает преследовать игрока** и уходит в темноту.

Игрок продолжает искать ключ-карты и исследовать лабораторию. Он снова находит фонарик, но помня прошлый раз оставляет его на потом, а сам продолжает ходить зажигалкой. Однако игрок также замечает, что **у зажигалки заканчивается газ** и ее свет слабеет, но намного медленнее, чем у фонаря. Поэтому он начинает выключать зажигалку в освещенных комнатах для ее экономии. Другой такой он не находил.

Наконец, игрок находит черную ключ-карту в одной из комнат. Он уже видел черные двери, но карту – впервые. Он берет ее и тут игра уведомляет его, что «тьма» снова начала охоту, только в этот раз она **не остановится пока не убьет персонажа**. Осознавая, что это означает важность черной ключ-карты, игрок бежит в глубину лаборатории в поисках черных дверей. А тьма уже настигает его по пятам. Игрок пробует использовать фонарь против нее, но понимает, что у тьмы выросло **сопротивление свету** и теперь ее намного сложнее замедлить. Имея на руках почти все ключ-карты, игрок справляется с тем, чтобы не загнать себя в тупик и успешно убегает от «тьмы», но та сокращает дистанцию. В процессе игрок подбирает новый фонари и обнаруживает, что это перезаряжает старый. Но его света уже недостаточно против «тьмы». Однако игрок успевает найти большую дверь с черными полосами – **выход из лаборатории** – и успевает открыть ее. Из-за двери персонажа окутывает свет. Игрок заходит внутрь и покидает лабораторию. Он смог **победить**.

Рисунок 74. Заветный выход

На этом завершается потенциальная игровая сессия, если конечно игрок не захочет снова отыскать выход из лаборатории. Ознакомиться с полноценным списком функциональных требований при необходимости можно в Приложении А «Техническое задание».

3.4.2. Реализация пользовательского интерфейса

В *Dark Lab* почти вся информация доступна пользователю через саму трехмерную сцену. К примеру, не нужно знать значение заряда фонаря, чтобы понять, что он заканчивается, потому что видно, что свет стал слабее. Однако, хороший интерфейс упрощает жизнь игроку, потому его необходимо было реализовать.

Всего есть два пользовательских интерфейса. Один реализован в классе *UMenuHUD* и отражает интерфейс главного меню, а второй – в *UGameHUD* и является интерфейсом основного режима игры или же основным интерфейсом. Начнем с первого.

Рисунок 75. Dark Lab. Главное меню

В **главном меню** представлены всего четыре важных элемента: название игры, две управляющие кнопки, а так же информация о версии в правом нижнем углу. Для каждой из кнопок указана клавиша клавиатуры, вызывающая тот же эффект что и кнопка. Нижняя кнопка завершает работу программы, верхняя же переводит ее в основной режим. Но перед этим при нажатии на любую из них сначала появляется окно ожидания (переходное).

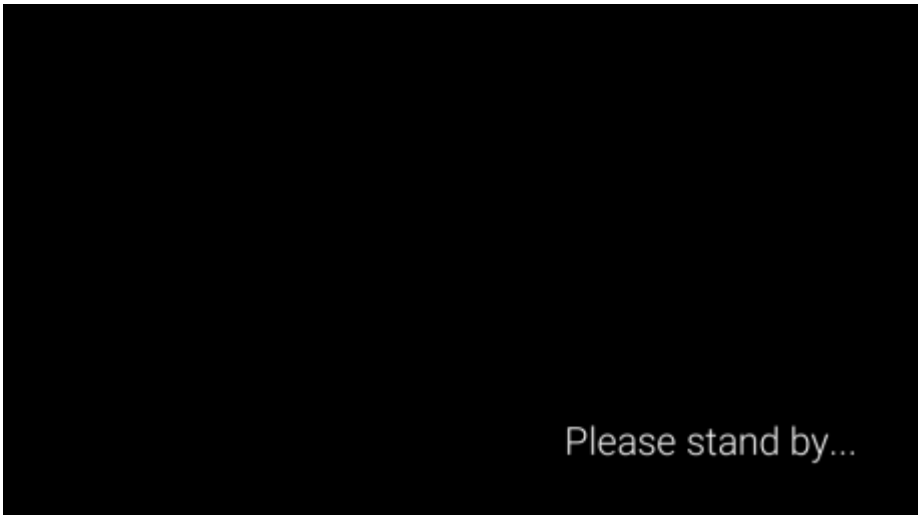

Рисунок 76. Переходное изображение

Рисунок 787. Начальное состояние основного пользовательского интерфейса

Сразу после запуска игры через “New Game” в главном меню или после рестарта игры пользовательский интерфейс входит в свое начальное состояние для **основного режима игры**. Это состояние изображено на рис.77.

Рисунок 778. Области основного пользовательского интерфейса

Больше деталей можно увидеть на рис.78, где представлено разделение интерфейса на области. Области 1, 4 и 6 являются интерактивными: при нажатии на определенные элементы в них происходят какие-то действия. Остальные же области лишь предоставляют ту или иную информацию. Сначала разберем их.

Область 2 (рис.79) предназначена для предупреждений или уведомлений. В ней игроку сообщают о начале игры, игрока предупреждают о начале охоты «тьмы» или о том, что та начала отступать. Здесь же появляется сообщение, когда игрок пытается открыть дверь, для которой у него нет ключ-карты. Есть еще несколько случаев. Сообщение исчезает через несколько секунд.

Рисунок 79. Стандартные уведомления в области 2

В **области 3** исключительно появляются сообщения о победе (“You escaped the laboratory”) или о поражении (“YOU DIED”).

Область 5 предназначена для информации об активируемых предметах в лаборатории. В ней записаны название и краткая сводка о предмете, который сейчас рядом с игроком (и подсвечен). Также в этой области указаны клавиши клавиатуры, которые необходимо нажать, чтобы использовать выделенный прибор. В области 5 ничего не отображается, если активируемых приборов в области активации у игрока нет.

Рисунок 80. После взятия зеленой ключ-карты

В **области 7** отображаются все собранные игроком ключи-карты, как видно из надписи над изображениями карточек. Изначально ключ-карта всего одна. Далее при взятии новой карты такого цвета, какого еще нет, ее изображение добавится слева от изображения последней взятой ключ-карты (рис.80).

Переходим к интерактивным областям. Начнем с четвертой.

Область 4 содержит несколько элементов. Во-первых это надпись, обозначающая, что область содержит используемые предметы. Далее прямо рядом с надписью подсказка клавиши или кнопки мыши, которыми можно использовать предметы. Под ними находятся поля для предметов. Изначально поле всего одно как на рис.77, однако, если подобрать фонарь, появится второе поле, как на рис.78. Для каждого предмета поле предмета включает:

1) Его кнопку-изображения, при нажатии на которую предмет берется в руки, а вокруг кнопки появляется белая обводка;
2) Название предмета;
3) Подсказку клавиши клавиатуры для взятия предмета
4) Постоянно обновляемое текущее состояние предмета (рис.81).

Когда новый предмет берется в руки с предыдущего обводка спадает.

Рисунок 81. Изменение состояния предмета со временем (уменьшение оставшегося в зажигалке газа)

Области 1 и **6** содержат элементы управления для вызова быстрого меню и помощи соответственно. Обе области имеют иконку-кнопку, название кнопки и подсказку для вызова.

Быстрое меню и меню помощи могут быть вызваны вместе или по-отдельности. Так или иначе, пока вызвано хотя бы одно из них, схема управления меняется (нельзя поворачивать взгляд, сильно ограничены другие действия), а также замедляется время и открываются соответствующие панели (рис.82).

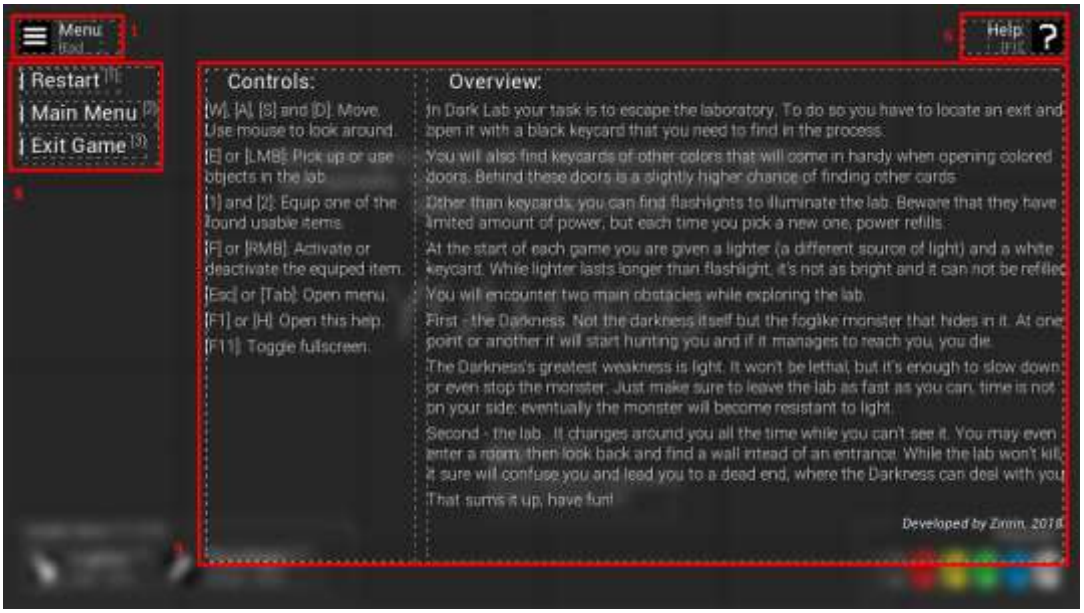

Рисунок 82. Области пользовательского интерфейса при вызове панелей быстрого меню и помощи

Область 9 – панель помощи – не является интерактивной. В ней представлена информация об игре и об управлении. Повторное нажатие на кнопку помощи (или соответствующий вызов с клавиатуры/геймпада) скрывают ее.

Область 8 – быстрое меню – интерактивно. В ней присутствуют три дополнительные кнопки и подсказки клавиш к ним. Первая перезапускает игру. Вторая возвращает в главное меню. Третья завершает работу приложения. При выборе любой из них прежде появится переходный экран (рис.76). Повторное нажатие на кнопку быстрого меню (или соответствующий вызов из контроллера) скрывает эту панель.

3.5. Выводы по третьей главе

Первым делом был обоснован выбор *Unreal Engine* в качестве основного инструмента разработки приложение. Вслед за этим отмечены некоторые особенности выбранного движка и описана архитектура всего разработанного приложения с разделением классов на группы и разбором каждого класса по-отдельности. Далее раскрыты особенности реализации, включающие два способа ускорения работы программы и использование blueprint-классов, наследующих от классов из кода. Наконец, показаны результаты работы на примере потенциальной игровой сессии и уделено внимание пользовательскому интерфейсу.

Эта глава логично завершает описание проекта *Dark Lab*. Далее к заключению.

Заключение

В рамках выполнения ВКР была разработана игра *Dark Lab* для ПК в жанре survival horror, в которой игроку предстоит отыскать выход из лаборатории скрываясь от монстра-«тьмы», в то время как помещение вокруг игрока постоянно меняется в темноте.

Были изучены существующие проекты в этом же жанре и на их основе составлен необходимый для реализации функционал, который позже был расширен до полноценных функциональных требований к программе.

Далее был составлен перечень требований, которым должен удовлетворять алгоритм генерации карты. Основные существующие алгоритмы генерации карт были проанализированы в сравнении с составленными требованиями и было принято решение базировать собственный алгоритм на идеях агентных алгоритмов.

Для основных элементов игры были разработаны структуры данных. В частности, было решено представлять карту через комнаты и проходы между ними, где комнаты соединяются проходами, а любой проход соединяет две комнаты.

После этого используя разработанную структуру, базируясь на составленных ранее требованиях к алгоритмам генерации и основываясь на агентных алгоритмах был разработан собственный алгоритм случайной динамической генерации карты, способный изменять лабораторию вокруг персонажа в процессе игры.

Также была определена базовая логика поведения монстра.

Приступая к разработке самой программы были проанализированы различные инструменты и выбор был остановлен на *Unreal Engine 4*. Технология была изучена, ее особенности, сильные и слабые стороны приняты к сведению.

Наконец, была разработана архитектура программы и сама программа, соответствующая поставленным функциональным требованиям и реализующая разработанный алгоритм. Производительность реализации алгоритма была повышена несколькими техниками.

Кроме того, для программы была написана документация.

В дальнейшем для развития *Dark Lab* существует несколько путей:

1) Расширение контента, ведь необходимо, чтобы игроку было чем заняться в игре и чтобы был стимул возвращаться играть снова. Из скорейших возможных дополнений: мебель в лаборатории; различные электрические системы, генераторы, рубильники, провода, которые необходимо подключать; записки; ловушки; собственная модель для персонажа; анимации использования предметов и прочее. Также возможно применение алгоритмов для генерирования тех или иных элементов геймплея во время игры.

2) Улучшение алгоритма генерации карты: новые возможные структуры карты; запоминание некоторых исчезающих комнат с их последующим появлением в других местах лаборатории; различные новые типы помещений и закрытые зоны; несколько уровней лаборатории и механика лифтов; оптимизация.

3) Продвинутые алгоритмы поведения «тьмы», новые возможности для нее: ориентирование на проходы; способность «разряжать» скопленный заряд электричества для отключения электроприборов рядом, способность разделяться на две части и др.

4) Новые противники и иные опасности.

Естественно, нет нужды следовать лишь одному из этих путей. Можно реализовывать элементы из разных, однако на данный момент фокус ожидается на первом и частично втором.

Также важнейшей задачей на будущее является выход на рынок.

Список использованных источников

1. 2015 is the year of horror games. Here's a peek [Электронный ресурс] / Hindustan Times. Режим доступа: www.hindustantimes.com/tech-reviews/2015-is-the-year-of-horror-games-here-s-a-peek/story-KaOQMmGwv6XTcEzGHZhYyK.html, свободный. (дата обращения: 28.05.18)
2. Alan Wake [Электронный ресурс] / Remedy Games. Режим доступа: www.alanwake.com/alan-wake.html, свободный. (дата обращения: 28.05.18)
3. Basic BSP Dungeon generation [Электронный ресурс] / RogueBasin. Режим доступа: roguebasin.roguelikedevelopment.org/index.php?title=Basic_BSP_Dungeon_generation, свободный. (дата обращения: 28.05.18)
4. Guide to Game Mechanics [Электронный ресурс] / GameDesigning. Режим доступа: www.gamedesigning.org/learn/basic-game-mechanics/, свободный. (дата обращения: 28.05.18)
5. Beneath Apple Manor [Электронный ресурс] / PCG Wiki. Режим доступа: pcg.wikidot.com/pcg-games:beneath-apple-manor, свободный. (дата обращения: 28.05.18)
6. Blueprints Visual Scripting [Электронный ресурс] / Unreal Engine. Режим доступа: docs.unrealengine.com/en-us/Engine/Blueprints, свободный. (дата обращения: 28.05.18)
7. Cooldowns [Электронный ресурс] / GiantBomb. Режим доступа: www.giantbomb.com/cooldowns/3015-2589/, свободный. (дата обращения: 28.05.18)
8. CryEngine [Электронный ресурс] / Crytek. Режим доступа: www.cryengine.com, свободный. (дата обращения: 28.05.18)
9. Darkwood [Электронный ресурс] / Acid Wizard Studio. Режим доступа: www.darkwoodgame.com, свободный. (дата обращения: 28.05.18)
10. Detention [Электронный ресурс] / Red Candle Games. Режим доступа: redcandlegames.com/detention, свободный. (дата обращения: 28.05.18)
11. Developing games [Электронный ресурс] / Microsoft. Режим доступа: docs.microsoft.com/en-us/previous-versions/windows/desktop/apps/hh452744(v=win.10), свободный. (дата обращения: 28.05.18)
12. Drunkard walk [Электронный ресурс] / PCG Wiki. Режим доступа: pcg.wikidot.com/pcg-algorithm:drunkard-walk, свободный. (дата обращения: 28.05.18)
13. Dynamic vs Static Procedural Generation [Электронный ресурс] / Medium. Режим доступа: medium.com/@eigenbom/dynamic-vs-static-procedural-generation-ed3e7a7a68a3, свободный. (дата обращения: 28.05.18)
14. Gamedev Glossary: Library VS Framework VS Engine [Электронный ресурс] / GameFromScratch. Режим доступа: www.gamefromscratch.com/post/2015/06/13/GameDev-Glossary-Library-Vs-Framework-Vs-Engine.aspx, свободный. (дата обращения: 28.05.18)

15. GameMaker Studio 2 [Электронный ресурс] / Yoyo Games. Режим доступа: www.yoyogames.com/gamemaker, свободный. (дата обращения: 28.05.18)
16. Garbage Collection Overview [Электронный ресурс] / Unreal Engine. Режим доступа: wiki.unrealengine.com/Garbage_Collection_Overview, свободный. (дата обращения: 28.05.18)
17. Generate Random Cave Levels Using Cellular Automata [Электронный ресурс] / Tutsplus. Режим доступа: gamedevelopment.tutsplus.com/tutorials/generate-random-cave-levels-using-cellular-automata--gamedev-9664, свободный. (дата обращения: 28.05.18)
18. Genetic Algorithm [Электронный ресурс] / PCG Wiki. Режим доступа: pcg.wikidot.com/pcg-algorithm:genetic-algorithm, свободный. (дата обращения: 28.05.18)
19. Git [Электронный ресурс] / Software Freedom Conservancy. Режим доступа: git-scm.com, свободный. (дата обращения: 28.05.18)
20. Godot [Электронный ресурс] / J. Linietsky, A. Manzur. Режим доступа: godotengine.org, свободный. (дата обращения: 28.05.18)
21. Green, D. Procedural Content Generation for C++ Game Development / A. Neil, D. Indrajit, M. Priyanka, M. Vishal, and N. Vedangi. Birmingham, UK: Packt Publishing Ltd., 2016 – 279 c.
22. How would you describe difference between using vanilla C++ and the one in Unreal Engine 4? [Электронный ресурс] / Quora. Режим доступа: www.quora.com/How-would-you-describe-difference-between-using-vanilla-C++-and-the-one-in-Unreal-Engine-4, свободный. (дата обращения: 28.05.18)
23. Less is More: On the Need for a Return to Generic Horror [Электронный ресурс] / Heard Tell. Режим доступа: www.heardtell.com/movies-and-tv/return-to-generic-horror, свободный(дата обращения: 28.05.18)
24. Male and Female Gamers: How Their Similarities and Differences Shape the Games Market [Электронный ресурс] / Newzoo. Режим доступа: newzoo.com/insights/articles/male-and-female-gamers-how-their-similarities-and-differences-shape-the-games-market/, свободный. (дата обращения: 28.05.18)
25. Number of indie games released on Steam worldwide from 2015 to 2017 [Электронный ресурс] / Statista. Режим доступа: www.statista.com/statistics/809258/number-indie-games-steam/, свободный. (дата обращения: 28.05.18)
26. Object Pool [Электронный ресурс] / Game Programming Patterns. Режим доступа: gameprogrammingpatterns.com/object-pool.html, свободный. (дата обращения: 28.05.18)
27. Observer [Электронный ресурс] / Bloober Team. Режим доступа: www.observer-game.com, свободный. (дата обращения: 28.05.18)
28. OpenGL [Электронный ресурс] / Khronos Group. Режим доступа: www.opengl.org, свободный. (дата обращения: 28.05.18)

29. Outlast [Электронный ресурс] / Red Barrels. Режим доступа: redbarrelsgames.com/games/outlast/, свободный. (дата обращения: 28.05.18)
30. Procedural Content Generation In Games [Электронный ресурс] / N. Shaker, J. Togelius, M. J. Nelson. Режим доступа: pcgbook.com, свободный. (дата обращения: 28.05.18)
31. Releasing the Unity C# source code [Электронный ресурс] / Unity Blog. Режим доступа: blogs.unity3d.com/ru/2018/03/26/releasing-the-unity-c-source-code, свободный. (дата обращения: 28.05.18)
32. Rogue [Электронный ресурс] / PCG Wiki. Режим доступа: pcg.wikidot.com/pcg-games:rogue, свободный. (дата обращения: 28.05.18)
33. Rooms and Mazes: A Procedural Dungeon Generator [Электронный ресурс] / B. Nystrom. Режим доступа: journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/, свободный. (дата обращения: 28.05.18)
34. Scp Foundation [Электронный ресурс]. Режим доступа: www.scp-wiki.net/, свободный. (дата обращения: 28.05.18)
35. SCP: Containment Breach [Электронный ресурс] / Undertow Games. Режим доступа: www.scpcbgame.com, свободный. (дата обращения: 28.05.18)
36. Setting Up Visual Studio for UE4 [Электронный ресурс] / Unreal Engine. Режим доступа: docs.unrealengine.com/en-us/Programming/Development/VisualStudioSetup, свободный. (дата обращения: 28.05.18)
37. Silent Hill [Электронный ресурс] / Konami. Режим доступа: www.konami.jp/gs/game/vx131/, свободный. (дата обращения: 28.05.18)
38. Simple and Fast Multimedia Library [Электронный ресурс] / L. Gomila. Режим доступа: www.sfml-dev.org, свободный. (дата обращения: 28.05.18)
39. Stanley Parable [Электронный ресурс] / Galactic Cafe. Режим доступа: www.stanleyparable.com, свободный. (дата обращения: 28.05.18)
40. Steam [Электронный ресурс] / Valve. Режим доступа: store.steampowered.com/, свободный. (дата обращения: 28.05.18)
41. Stifled [Электронный ресурс] / Gattai Games. Режим доступа: stifledgame.com/, свободный. (дата обращения: 28.05.18)
42. Survival horror [Электронный ресурс] / Oxford Living Dictionaries. Режим доступа: en.oxforddictionaries.com/definition/survival_horror, свободный. (дата обращения: 28.05.18)
43. Survival Horror [Электронный ресурс] / Steamspy. Режим доступа: steamspy.com/tag/Survival+horror, свободный. (дата обращения: 28.05.18)

44. The State of Horror Video Games in 2015 [Электронный ресурс] / IGN. Режим доступа: www.ign.com/articles/2015/07/11/the-state-of-horror-video-games-in-2015, свободный. (дата обращения: 28.05.18)
45. The Year In Numbers 2017 [Электронный ресурс] / gameindustry.biz. Режим доступа: www.gamesindustry.biz/articles/2017-12-20-gamesindustry-biz-presents-the-year-in-numbers-2017, свободный. (дата обращения: 28.05.18)
46. Top Game Engines In 2018 [Электронный ресурс] / Instabud Blog. Режим доступа: blog.instabug.com/2017/12/game-engines, свободный. (дата обращения: 28.05.18)
47. Unity [Электронный ресурс] / Unity Technologies. Режим доступа: unity3d.com, свободный. (дата обращения: 28.05.18)
48. Unreal Engine 4 [Электронный ресурс] / Epic Games. Режим доступа: www.unrealengine.com/en-US/what-is-unreal-engine-4, свободный. (дата обращения: 28.05.18)
49. UnrealHeaderTool [Электронный ресурс] / Unreal Engine. Режим доступа: docs.unrealengine.com/en-us/Programming/UnrealBuildSystem#unrealheadertool, свободный. (дата обращения: 28.05.18)
50. Update Method [Электронный ресурс] / Game Programming Patterns. Режим доступа: gameprogrammingpatterns.com/update-method.html, свободный. (дата обращения: 28.05.18)
51. Using Raycasts (Tracing) [Электронный ресурс] / Unreal Engine. Режим доступа: docs.unrealengine.com/en-us/Gameplay/HowTo/UseRaycasts, свободный. (дата обращения: 28.05.18)
52. Visual Studio [Электронный ресурс] / Microsoft. Режим доступа: www.visualstudio.com, свободный. (дата обращения: 28.05.18)
53. Voronoi Diagram [Электронный ресурс] / PCG Wiki. Режим доступа: pcg.wikidot.com/pcg-algorithm:voronoi-diagram, свободный. (дата обращения: 28.05.18)
54. Why Sound Design In Horror Gaming Matters [Электронный ресурс] / CinemaBlend. Режим доступа: www.cinemablend.com/games/Why-Sound-Design-Horror-Gaming-Matters-129577.html, свободный. (дата обращения: 28.05.18)
55. W-what's that noise? Five horror games that use sound effectively [Электронный ресурс] / Destructoid. Режим доступа: www.destructoid.com/w-what-s-that-noise-five-horror-games-that-use-sound-effectively-469737.phtml, свободный. (дата обращения: 28.05.18)
56. Метро: Луч Надежды - [Электронный ресурс] / Бука. Режим доступа: www.metro2033.ru/game/, свободный. (дата обращения: 28.05.18)
57. Сравнение игровых движков. [Электронный ресурс] / Devgam. Режим доступа: devgam.com/sravnenie-igrovyx-dvizhkov-kakoj-vybrat, свободный. (дата обращения: 28.05.18)