

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное  
учреждение высшего образования  
«Иркутский государственный университет»  
(ФГБОУ ВО «ИГУ»)  
Институт математики и информационных технологий  
Кафедра алгебраических и информационных систем

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**  
**по направлению «09.03.03 Прикладная информатика»**  
**профиль подготовки «Проектирование и разработка**  
**информационных систем»**

**РАЗРАБОТКА КОМПЬЮТЕРНОЙ ИГРЫ**  
**С СИСТЕМОЙ ДИНАМИЧЕСКОЙ ГЕНЕРАЦИИ**  
**ИГРОВЫХ ЛОКАЦИЙ И АДАПТИВНЫХ БОЕВЫХ СЦЕНАРИЕВ**

Студент 4 курса очного отделения  
Группа 02461-ДБ  
Ковалев Егор Юрьевич

Руководитель:  
к.ф.-м.н., доцент  
\_\_\_\_\_ Кириченко К.Д.

Допущен к защите  
Зав. кафедрой АиИС, д.ф.-м.н., доцент  
\_\_\_\_\_ Пантелейев В.И.

Иркутск 2025

# СОДЕРЖАНИЕ

|  |           |
|--|-----------|
| <b>ВВЕДЕНИЕ</b>  | <b>4</b>  |
| <b>Глава 1. Основы разработки игры</b>                               | <b>7</b>  |
| 1.1 Процедурная генерация локаций . . . . .                          | 8         |
| 1.2 Тактические сражения на основе подкрепляющего обучения . . . . . | 8         |
| 1.3 Технологии разработки . . . . .                                  | 9         |
| <b>Глава 2. Описание используемых алгоритмов</b>                     | <b>11</b> |
| 2.1 Алгоритм Fast Poisson Disk Sampling . . . . .                    | 11        |
| 2.2 Алгоритм триангуляции Делоне . . . . .                           | 14        |
| 2.3 Алгоритм Краскала . . . . .                                      | 16        |
| 2.4 Спектральная кластеризация . . . . .                             | 18        |
| 2.5 Генетический алгоритм . . . . .                                  | 20        |
| 2.6 Поиск по дереву Монте-Карло . . . . .                            | 24        |
| <b>Глава 3. Описание реализации проекта</b>                          | <b>29</b> |
| 3.1 Процедурная генерация игровой локации . . . . .                  | 29        |
| 3.1.1 Распределение точек интереса . . . . .                         | 29        |
| 3.2 Построение топологии игровой локации . . . . .                   | 33        |
| 3.3 Размещение квестовых триггеров . . . . .                         | 40        |
| 3.4 Взаимодействие с базой сценариев . . . . .                       | 50        |
| 3.5 Адаптивные боевые сценарии . . . . .                             | 54        |
| 3.5.1 Формализация правил сражений . . . . .                         | 54        |
| 3.6 Реализация системы сражений в Unreal Engine . . . . .            | 59        |
| 3.6.1 Система взаимодействия классов . . . . .                       | 62        |
| 3.6.2 Привязка к формализации правил сражений . . . . .              | 63        |

|  |           |
|--|-----------|
| 3.7 Описание архитектуры искусственного интеллекта для тактической пошаговой системы боевых сценариев . . . . .        | 64        |
| <b>Выводы и перспективы развития</b>   | <b>74</b> |
| <b>ЗАКЛЮЧЕНИЕ</b>  | <b>76</b> |
| <b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>  | <b>78</b> |
| <b>ПРИЛОЖЕНИЕ 1. Реализация структуры узла в алгоритме Fast Poisson Disk Sampling распределения точек на плоскости</b> | <b>81</b> |
| <b>ПРИЛОЖЕНИЕ 2. Реализация структуры треугольника и алгоритма Боуера-Ватрсона построения триангуляции Делоне</b>      | <b>86</b> |
| <b>ПРИЛОЖЕНИЕ 3. Объявление структур генетического алгоритма распределения сценарных триггеров по графу</b>            | <b>92</b> |
| <b>ПРИЛОЖЕНИЕ 4. Объявление и реализация основного метода алгоритма поиска по дереву Монте-Карло</b>                   | <b>94</b> |
| <b>ПРИЛОЖЕНИЕ 5. Класс, описывающий боевую фигуру (юнита) на поле сражения</b>   | <b>97</b> |

# ВВЕДЕНИЕ

В современном мире компьютерные игры занимают значительное место в индустрии развлечений, становясь не только источником досуга, но и средством обучения, развития когнитивных навыков и социального взаимодействия. С развитием технологий и ростом ожиданий пользователей игровая индустрия сталкивается с необходимостью создания более сложных, разнообразных и персонализированных игровых миров. Традиционные методы разработки, основанные на ручном проектировании уровней и сценариев, становятся все менее эффективными в условиях увеличения масштабов проектов и стремления к высокой реиграбельности\*. В данном контексте, использование алгоритмов процедурной генерации контента и методов машинного обучения представляет собой перспективное направление, способное радикально изменить подход к созданию игрового опыта.

**Основной целью** данной работы является разработка компьютерной игры в жанре стратегии, с использованием технологий процедурной генерации игрового пространства и адаптивных боевых сценариев, реализованных с использованием подкрепляющего обучения. Проект направлен на создание динамической игровой среды, в которой локации и топология прохождения формируются автоматически, а искусственный интеллект противников адаптируется к действиям игрока, обеспечивая уникальный и интеллектуально стимулирующий опыт. Игра представляет собой стратегический проект, в котором игрок принимает тактические решения, сталкиваясь с запутанным окружением и противником, чье поведение эволюционирует в зависимости от действий игрока. Для достижения поставленной цели были сформулированы следующие задачи:

1. разработать систему процедурной генерации игровых локаций;
2. создать алгоритм построения графа игрового пространства;
3. реализовать алгоритм размещения контента на графике игрового простран-

ства;

4. разработать систему обучения с подкреплением для адаптивного ИИ противников;
5. объединить реализованные части проекта в движке Unreal Engine для полноценного формирования проекта;
6. провести тестирование и анализ на эффективность.

**Актуальность данной** работы определяется несколькими ключевыми факторами. Во-первых, в условиях стремительного роста игровой индустрии разработчики сталкиваются с необходимостью создания контента, который одновременно обладает высокой реиграбельностью<sup>1</sup> и минимальными затратами на производство. Процедурная генерация позволяет автоматизировать процесс создания игровых миров, освобождая ресурсы для работы над другими аспектами игры, такими как механики или повествование. Во-вторых, современные игроки ожидают более глубокого взаимодействия с виртуальной средой, включая интеллектуально сложных противников, способных адаптироваться к их стилю игры. Использование машинного обучения, в частности подкрепляющего обучения, открывает новые горизонты для создания таких ИИ-агентов, что делает игровой процесс более увлекательным и непредсказуемым.

**Новизна** работы состоит в интеграции процедурной генерации и методов машинного обучения для создания целостной игровой системы в жанре стратегии. В отличие от традиционных подходов, где процедурная генерация часто ограничивается созданием статичных уровней, в данном проекте она используется для динамического формирования графа прохождения, на котором размещаются квесты и сражения. Это позволяет достичь высокого уровня вариативности игрового мира, сохраняя при этом его логическую целостность.

---

<sup>1</sup>Реиграбельность (англ. *replayability*) — это характеристика видеоигры, отражающая степень ее привлекательности для повторного прохождения пользователем после завершения основного игрового процесса.

Еще одним аспектом является применение подкрепляющего обучения для реализации адаптивных боевых сценариев в стратегической игре. ИИ-противник, обучающийся на действиях игрока, представляет собой шаг вперед по сравнению с традиционными скриптовыми системами, обеспечивая более естественное и сложное поведение.

**Практическая значимость** заключается в том, что сочетание процедурной генерации и адаптивного ИИ в рамках одной игры представляет собой относительно малоизученную область, что подчеркивает актуальность исследования с научной точки зрения. Разработка подобных систем может найти применение не только в игровой индустрии, но и в смежных областях.

# Глава 1. Основы разработки игры

Проект ориентирован на жанр стратегических игр с пошаговым геймплеем<sup>2</sup>. В таких проектах пользователи поочередно совершают ходы, управляя юнитами<sup>3</sup> на игровом поле, что требует глубокого стратегического мышления и планирования. В рамках данного проекта, одним из элементов геймплея являются боевые сценарии, где игрок управляет стилизованными фигурками-животными с уникальными характеристиками, сражаясь против ИИ-противника. Данный жанр обладает следующими ключевыми особенностями.

- **Пошаговость**, где каждый ход тщательно обдумывается, что создает напряженную тактическую динамику.
- **Управление ресурсами**, когда игрок должен эффективно использовать юниты и их способности.
- **Тактические сражения** представляют собой позиционирование и выбор действий, которые определяют исход сражений.

Тем не менее, данные элементы, являясь основой жанра и предоставляя пользователю возможности для стратегического маневра, зачастую ограничены установленными правилами и созданными разработчиками условиями для игры. Для того, чтобы повысить вариативность и непредсказуемость игрового процесса, было принято решение о внедрении алгоритмов, формирующих практически неповторимые игровые сценарии, где процедурно генерируемые карты и адаптивный ИИ обеспечивают уникальный опыт в каждой сессии.

---

<sup>2</sup>Геймплей (англ. *gameplay*) — это совокупность игровых механик, правил и взаимодействий, определяющих процесс участия игрока в игре и формирующих его игровой опыт.

<sup>3</sup>Юнит (англ. *unit*) — это базовая управляемая или автономная сущность на игровом поле.

## 1.1 Процедурная генерация локаций

Procedural content generation is the algorithmic creation of game content with limited or indirect user input[3].<sup>4</sup>

Процедурная генерация контента — это метод создания данных или объектов в видеоиграх с использованием алгоритмов вместо ручного проектирования. Этот подход позволяет автоматически генерировать разнообразный и уникальный контент. Основная цель PCG — повысить реиграбельность игр и снизить затраты на их разработку, предоставляя игрокам новые впечатления при каждом прохождении. К основным преимуществам данного инструмента разработки следует отнести уникальность каждой игровой сессии, экономию времени и ресурсов разработки, а также возможность создания обширных игровых миров.

Для реализации процедурной генерации в данном проекте, были выбраны алгоритмы, позволяющие создать структурированные и сбалансированные игровые карты, где точки интереса<sup>5</sup> связаны в единый граф.

## 1.2 Тактические сражения на основе подкрепляющего обучения

Для полноценной реализации сценарных и разработанных механик игрового процесса требуется внедрение специализированного механизма организации сражений. По мере продвижения пользователя в ходе игры и увеличения числа проведенных боевых столкновений, специальный алгоритм анализирует тактические приемы игрока и впоследствии применяет усвоенные стратегии против него самого, делая каждое сражение уникальным и требующим от игро-

---

<sup>4</sup>Процедурная генерация — это алгоритмическое создание игрового контента с ограниченным или косвенным участием пользователя. (*Перевод автора*)

<sup>5</sup>Точка интереса (англ. *point of interest, POI*) — это выделенный элемент игрового пространства, представляющий собой значимое место или событие, с которым игрок может взаимодействовать.

ка постоянного совершенствования тактик. Такой подход стимулирует пользователя к поиску новых методов и тактических решений, тем самым способствуя развитию его навыков в процессе прохождения. Для создания адаптивного противника на основе искусственного интеллекта в рамках проекта применяется методика обучения с подкреплением (Reinforcement Learning, RL), позволяющая системе анализировать действия игрока и динамически корректировать собственные стратегии в реальном времени[12]. Основные компоненты RL:

- агент — ИИ-противник, принимающий решения;
- окружение — игровое поле и правила;
- состояние — текущая конфигурация игры;
- действия — возможные ходы ИИ;
- награда — оценка успеха действий (например, победа или захват юнита);

В качестве алгоритма выбран Proximal Policy Optimization (PPO), который обеспечивает стабильное и эффективное обучение. Такой подход позволяет ИИ становиться сложным и непредсказуемым соперником, реагирующим на тактику игрока.

### 1.3 Технологии разработки

#### Игровой движок Unreal Engine

Разработка видеоигры представляет собой многоэтапный и комплексный процесс, предполагающий интеграцию широкого спектра ресурсов. Центральным программным обеспечением, обеспечивающим объединение и координацию всех компонентов, являются игровые движки — специализированные программные платформы, предназначенные для создания интерактивных виртуальных сред. В рамках реализации данного проекта в качестве основного технологического решения выбран игровой движок Unreal Engine[19] версии 5.4.4. Ос-

новными факторами, определившими данный выбор, являются высокая производительность, широкие возможности масштабируемой разработки, а также гибкость архитектуры движка, позволяющая эффективно реализовывать как визуальные компоненты, так и алгоритмы любой сложности.

### **Язык программирования: C++**

В качестве основного языка программной реализации выбран C++<sup>20</sup>, что обусловлено его высокой производительностью, эффективным управлением системными ресурсами и широкими возможностями для разработки программного обеспечения повышенной сложности[2]. Язык предоставляет низкоуровневый контроль над памятью и вычислительными процессами, что особенно важно при создании ресурсоемких приложений, таких как видеоигры. Дополнительным фактором является тесная интеграция C++ с архитектурой игрового движка Unreal Engine, для которого данный язык является основным средством разработки пользовательской логики.

### **Дополнительные инструменты и библиотеки.**

1. **Eigen/Dense** это C++ библиотека для линейной алгебры, используемая в спектральной кластеризации[16].
2. **nlohmann/json** Заголовочный модуль для C++, позволяющий сериализовать и парсить JSON, без внешних зависимостей.[18];
3. **Visual Studio 2022** является средой разработки и отладки программного кода[20].
4. **Unreal Engine Editor** встроенный в движок инструмент для настройки игровых сцен и тестирования[7].

## Глава 2. Описание используемых алгоритмов

### 2.1 Алгоритм Fast Poisson Disk Sampling

При генерации игровых карт часто возникает задача равномерного и при этом случайного размещения точек  $p$  на двумерной плоскости таким образом, чтобы было возможным избежать их избыточной кластеризации. Формулировка задачи сводится к следующему: для заданного прямоугольного пространства с размерами  $width \times height$  требуется случайно разместить набор точек таким образом, чтобы расстояние между любыми двумя точками не было меньше заранее заданного радиуса  $r$ , то есть для любых точек  $p_i = (x_i, y_i)$  и  $p_j = (x_j, y_j)$  должно выполняться неравенство

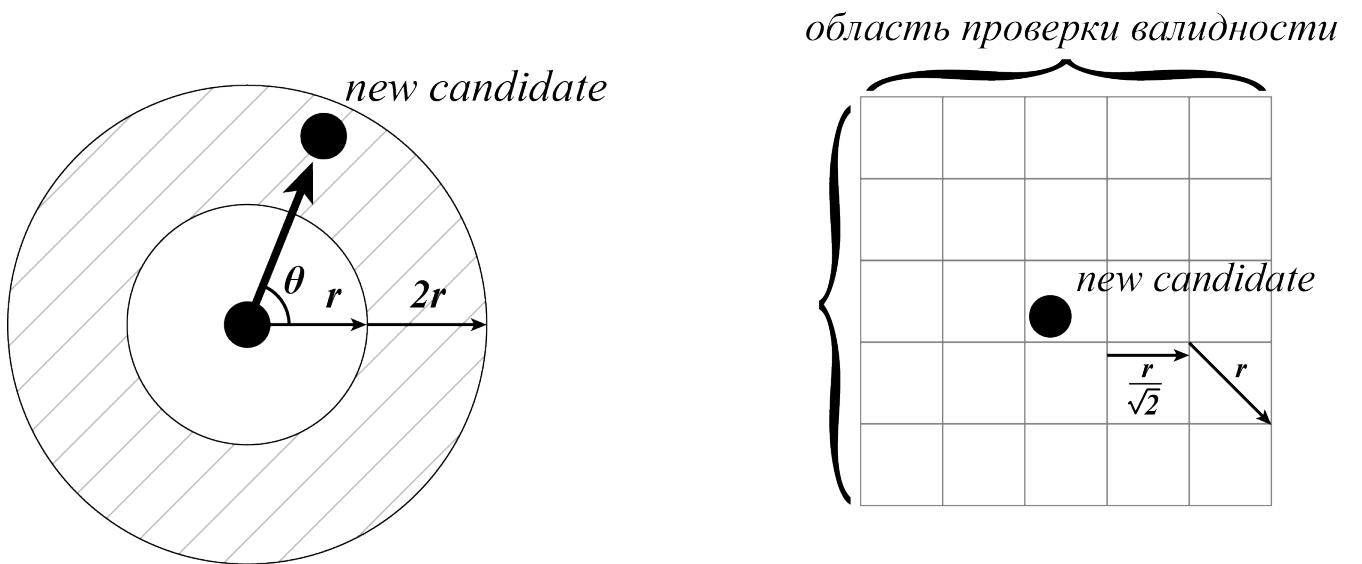
$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq r. \quad (1)$$

Для решения поставленной задачи в проекте используется эвристический алгоритм Fast Poisson Disk Sampling<sup>6</sup>. Алгоритм[17] не является строго детерминированным, но на практике обеспечивает расположение точек при вычислительной сложности  $O(n)$ .

Ключевая структура данных алгоритма — это двумерная сетка, в которую каждая принятая точка заносится по индексам, вычисленным на основе ее координат и размера квадратной ячейки со стороной  $\frac{r}{\sqrt{2}}$ . Такая структура позволяет эффективно выполнять проверку на минимальное расстояние до уже существующих точек путем локального поиска в соседних ячейках на расстоянии  $2 \times r$ , вместо сравнения со всеми имеющимися точками. Диагональ ячейки равна  $r$ , что гарантирует установку не более одной точки на ячейку. Дополнительно вводится параметр  $k$ , который определяет количество попыток генерации допустимой новой точки вокруг каждой активной точки прежде, чем она будет удалена из активного списка.

---

<sup>6</sup>Быстрый алгоритм размещения точек по методу дисковой выборки



(a) Определение области генерации координат новых точек

(b) Область проверки точки на валидность расстояния до других точек

Рисунок 1. Определение области проверки точки на валидность расстояния до других точек

Для хранения точек, вокруг которых допустимо размещение соседей, создается активный список. Этот список изменяется в зависимости от количества точек, имеющих вокруг себя свободную область для размещения новых. Здесь используется функция `erase()` для удаления элементов, которая линейна по текущему размеру активного списка. В результате общая временная скорость алгоритма составит  $O(n^2)$ , из-за количества точек на размещение и функции на их удаление. Для ускорения работы алгоритма до  $O(\log n)$  в дальнейшем планируется реализация структуры дерева по неявному ключу, которое позволит снизить временные затраты на удаление точек из активного списка. Результирующие точки хранятся в отдельном массиве, который изменяется динамически при добавлении каждой новой точки.

Алгоритм начинается с генерации первой точки со случайными координатами  $x$  и  $y$  внутри границ установленного прямоугольника, которая добавляется как в итоговый набор точек, так и в активный список. Далее процесс повторя-

ется в цикле: пока активный список не опустеет, из него случайным образом выбирается одна активная точка, вокруг которой выполняются до  $k$  попыток генерации новой точки на случайном расстоянии от  $r$  до  $2r$  и под случайнм углом, как показано на рис. 1а. Для  $p$  генерируется до  $k$  точек в кольце  $[r, 2r]$ :

$$x_{\text{new}} = p.x + \text{radius} \cdot \cos(\theta); \quad (2)$$

$$y_{\text{new}} = p.y + \text{radius} \cdot \sin(\theta), \quad (3)$$

где  $\theta \sim U[0, 2\pi]$ ,  $\text{radius} \sim U[r, 2r]$ .

Далее точка проходит проверку на расположение внутри границ прямоугольника:  $x \in [0, \text{width}]$ ,  $y \in [0, \text{height}]$ , а также удовлетворение условию проверки расстояния до соседних точек  $< r$  см. рис. 1б. Если точка удовлетворяет этим условиям, она добавляется в результирующий набор и активный список. Если после  $k$  попыток подходящей точки найдено не было, то исходная активная точка удаляется из активного списка и добавляется в результирующий.

В рамках реализации данного алгоритма в проекте использовались следующие параметры: радиус  $r$  регулирующий плотность размещения точек, и константа  $k$ , которая была установлена значением 30, что является эмпирически подобранный величиной.

На заключительном этапе генерации, если итоговое количество точек оказалось существенно ниже требуемого для равномерного покрытия карты, алгоритм продолжает заполнение свободных зон пространства.

Таким образом, Fast Poisson Disk Sampling обеспечивает генерацию равномерно распределенных точек интереса, которые далее используются в проекте как опорные элементы для построения маршрутов, размещения квестов и задания топологии игрового мира.

## 2.2 Алгоритм триангуляции Делоне

Задача триангуляции часто возникает при обработке двумерных точек для построения сетей, разбиений и графических структур. Требуется разделить заданное множество точек на треугольники таким образом, чтобы выполнялось условие триангуляции Делоне: окружность, описанная вокруг любого треугольника, не должна содержать другие точки из исходного множества[1]. Для треугольника с вершинами  $A(x_1, y_1)$ ,  $B(x_2, y_2)$ ,  $C(x_3, y_3)$  и точки  $P(x_p, y_p)$  проверка на расположение точки внутри описанной окружности этого треугольника выполняется через определитель матрицы

$$\begin{vmatrix} x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \\ x_p & y_p & x_p^2 + y_p^2 & 1 \end{vmatrix}$$

Если определитель положителен,  $P$  лежит внутри описанной окружности.

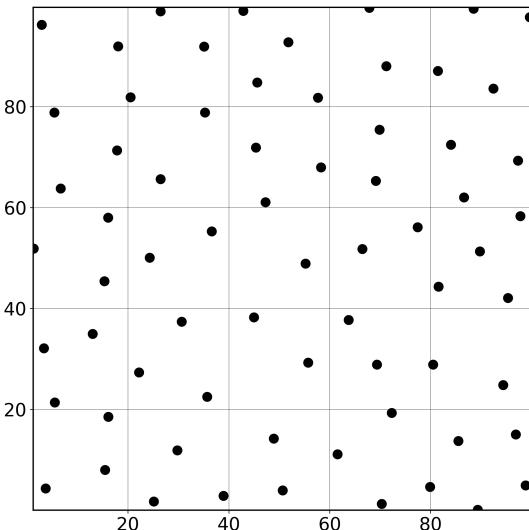
Для решения данной задачи используется инкрементальный алгоритм Боуера-Ватсона[15]. На практике он обеспечивает корректное построение триангуляционной сетки при средней вычислительной сложности  $O(n \log n)$ , где  $n$  — количество входных точек.

Основными структурами данных в реализации алгоритма являются треугольники, которые представляют собой структуры из трех точек и служат элементарными ячейками триангуляции. Ребра представляют из себя пары точек, используемые для построения границ треугольников и поиска "границы полигона" при вставке новой точки. А структуры `set` и `map` применяются для поиска уникальных ребер и фильтрации дубликатов при пересчете триангуляции.

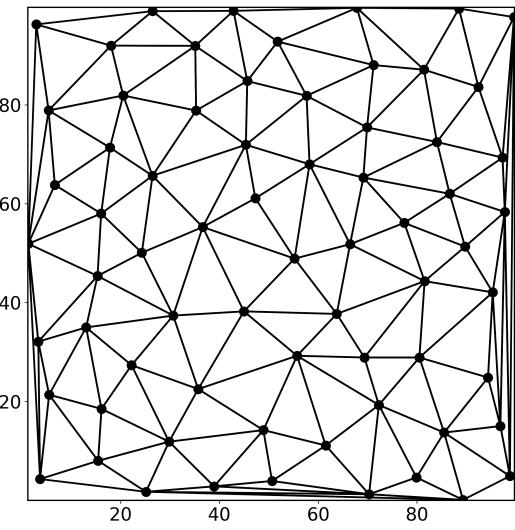
Алгоритм начинается с построения так называемого «супертреугольника», который охватывает все входные точки. Этот треугольник добавляется в начальный список треугольников. Далее процесс триангуляции выполняется

поэтапно для каждой новой точки: на каждом шаге алгоритм ищет все треугольники, окружности которых содержат текущую вставляемую точку. Такие треугольники помечаются как некорректные («плохие» треугольники). На следующем этапе из ребер этих плохих треугольников строится граница полигона. Для этого выбираются ребра, которые не повторяются более одного раза — это и есть ребра на внешнем контуре. Плохие треугольники удаляются из текущего списка. На месте удаленных треугольников создаются новые треугольники, соединяющие каждое ребро полигона с вставляемой точкой. После завершения цикла удаляются все треугольники, которые содержат вершины супертреугольника, так как они служили лишь для инициализации процесса.

Ключевая проверка корректности триангуляции основана на вычислении детерминанта для определения положения точки относительно описанной окружности треугольника. Для этого используется геометрическая функция, которая определяет, лежит ли точка внутри окружности заданного треугольника.



(a) Распределение узлов методом Fast Poisson Disk Sampling



(b) Построение триангуляции Делоне алгоритмом Боуера-Ватсона

С точки зрения вычислительной сложности, алгоритм имеет среднюю сложность  $O(n \log n)$  за счет эффективной локальной перестройки триангуляции при добавлении новых точек. Однако, из-за использования структуры удаления при условии нахождения плохого треугольника:

```
triangles.erase(std::remove_if(badTriangles))
```

сложность алгоритма составит  $O(T \cdot B)$ , где  $T$  — это количество всех треугольников, а  $B$  — число «плохих треугольников».

## 2.3 Алгоритм Краскала

При работе с взвешенными графами часто возникает задача нахождения минимального остовного дерева (МОД), которое соединяет все вершины графа с минимально возможным суммарным весом ребер. Формально задача формулируется следующим образом: для заданного связного неориентированного графа  $G = (V, E)$ , где  $V$  — множество вершин, а  $E$  — множество ребер с весами  $w : E \rightarrow \mathbb{R}^+$ , требуется найти подмножество ребер  $T \subseteq E$ , образующее дерево, для которого выполняется:

$$\sum_{e \in T} w(e) \rightarrow \min, \quad (4)$$

при условии, что  $T$  соединяет все вершины  $V$ .

Для решения этой задачи применяется алгоритм Краскала[8], основанный на жадной стратегии. Алгоритм не требует строгой детерминированности при равных весах ребер, но гарантирует построение МОД за вычислительную сложность  $O(|E| \log |E|)$ , что обусловлено этапом сортировки ребер.

Ключевой структурой данных в алгоритме является система непересекающихся множеств (DSU — Disjoint Set Union), которая обеспечивает эффективную проверку и объединение компонент связности. Каждая вершина графа изначально принадлежит собственному множеству. Структура DSU поддерживает две операции:  $find(u)$  — поиск представителя множества, содержащего

вершину  $u$  и  $\text{union}(u, v)$  — объединение множеств, содержащих вершины  $u$  и  $v$ .

Оптимизации сжатия пути и объединения по рангу позволяют снизить сложность операций до практически постоянного времени  $O(\alpha(|V|))$ , где  $\alpha$  — обратная функция Аккермана. Функция Аккермана  $A(m, n)$ , лежащая в основе этой оценки, определяется рекурсивно

$$A(m, n) = \begin{cases} n + 1, & m = 0, \\ A(m - 1, 1), & m > 0, n = 0, \\ A(m - 1, A(m, n - 1)), & m > 0, n > 0. \end{cases}$$

Она растет катастрофически быстро даже для небольших  $m$  и  $n$ . Например,  $A(4, 2)$  настолько велико, что не имеет физического смысла в реальных вычислениях.

Обратная функция Аккермана  $\alpha(n)$  — это минимальное  $m$ , при котором  $A(m, m) \geq n$ . Ее рост, напротив, крайне медленный: для  $n \leq 2^{65536}$  значение  $\alpha(n) \leq 4$ . Это объясняет, почему в DSU амортизированная сложность операций `find` и `union` считается практически константной.

Таким образом, функция Аккермана служит теоретической основой для анализа сложности DSU. Ее обратная версия отражает, насколько эффективно оптимизации компенсируют рекурсивную природу исходной функции, делая операции почти мгновенными даже для графов с миллионами вершин.

Алгоритм начинается с сортировки всех ребер графа по возрастанию веса. Далее последовательно рассматриваются ребра в порядке увеличения их веса. Для каждого ребра  $e = (u, v)$  выполняется проверка условия

$$\text{find}(u) \neq \text{find}(v) \tag{5}$$

Если представители множеств вершин  $u$  и  $v$  различны, ребро  $e$  добавляется в остовное дерево, а множества вершин объединяются операцией  $\text{union}(u, v)$ . В

противном случае ребро игнорируется, так как его включение привело бы к образованию цикла. Процесс продолжается до тех пор, пока не будут обработаны все ребра или пока размер оставшегося дерева не достигнет  $|V| - 1$ .

Для обеспечения корректности алгоритма критически важна эффективная реализация DSU. В проекте используется двукратное сжатие пути при операции `find` и объединение по рангу при `union`, что гарантирует амортизированную сложность  $O(1)$  на операцию. Результирующие ребра хранятся в динамическом массиве, который пополняется при каждом успешном добавлении ребра в дерево.

## 2.4 Спектральная кластеризация

При обработке графов часто возникает необходимость группировки данных, его вершин, на логически обоснованные участки. То есть необходимо выполнить задачу разделения вершин графа на группы (кластеры), так чтобы вершины внутри одного кластера были сильно связаны между собой, а между кластерами связи были слабыми. Для реализации данного рода задач применяется алгоритм спектрального разбиения графа.

Спектральная кластеризация — это метод разделения данных на группы, основанный на анализе собственных значений и векторов матриц, связанных с графиком. Основная идея заключается в использовании линейно-алгебраических свойств графа для выявления его «естественных» кластеров[11; 14]. Для этого рассматривается неориентированный взвешенный граф  $G = (V, E)$ , где  $V$  — множество вершин, а  $E$  — ребра с весами  $w_{ij}$ , отражающими степень сходства между вершинами  $i$  и  $j$ . Ключевыми матрицами в этом методе являются матрица смежности  $A$

$$A_{ij} = \begin{cases} w_{ij}, & \text{если } (i, j) \in E, \\ 0, & \text{иначе} \end{cases}$$

которая кодирует связи между вершинами, матрица степеней  $D$ , диагональная матрица, где  $D_{ii} = \sum_j A_{ij}$ , суммирующая веса ребер, исходящих из каждой вершины, а также матрица Лапласа  $L = D - A$ , а ее нормированная версия  $L_{\text{sym}} = D^{-1/2} L D^{-1/2}$  обеспечивает устойчивость к различиям в плотности кластеров.

Алгоритм реализации начинается с построения матрицы сходства. Для каждой пары вершин  $(i, j)$  вычисляется мера близости через гауссово ядро  $w_{ij} = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right)$ , где  $\sigma$  — параметр масштаба. Реализуется через двойной цикл, заполняющий матрицу  $S$  размера  $N \times N$ , что соответствует полному графу сходства. На основе матрицы  $S$  строится диагональная матрица степеней  $D$ , после чего вычисляется нормированный Лапласиан  $L_{\text{sym}} = I - D^{-1/2} S D^{-1/2}$ . В реализации используется библиотека Eigen для работы с матричными операциями, включая обращение диагональных матриц. С помощью метода `Eigen::SelfAdjointEigenSolver` находятся собственные векторы матрицы  $L_{\text{sym}}$ . Первые  $k$  векторов (исключая первый, соответствующий  $\lambda = 0$ ) формируют матрицу  $U$ , строки которой нормируются для подготовки к кластеризации. Нормированные строки матрицы  $U$  обрабатываются алгоритмом k-средних. Инициализация центроидов<sup>7</sup> выполняется случайным выбором  $k$  строк из  $U$ , после чего итеративно обновляются метки кластеров и центроиды. Для ускорения расчетов расстояний применяются векторные операции Eigen.

Собственные значения и векторы матрицы Лапласа играют центральную роль. Все ее собственные значения неотрицательны ( $\lambda_i \geq 0$ ), причем наименьшее значение  $\lambda_1 = 0$  соответствует константному вектору  $\mathbf{v}_1 = [1, 1, \dots, 1]^T$ . Количество нулевых собственных значений равно числу связных компонент графа. Если граф содержит  $k$  «естественных» кластеров, то первые  $k$  собственных векторов, соответствующих наименьшим ненулевым  $\lambda_i$ , кодируют информацию о разделении данных. Например, для графа с двумя слабо связанными кластера-

---

<sup>7</sup> Центроид — это усредненная характеристика всех точек, принадлежащих одному кластеру.

ми второе наименьшее собственное значение  $\lambda_2$  будет близко к нулю, а соответствующий вектор  $\mathbf{v}_2$  резко изменится между кластерами, что визуализируется на рисунке.

Теоретической основой метода является теорема Донат-Хоффмана, утверждающая, что собственные векторы матрицы Лапласа сохраняют постоянство внутри связных компонент. Это свойство позволяет алгоритму устойчиво работать даже при наличии шума. Однако, реализация имеет вычислительные ограничения: построение матрицы сходства требует  $O(N^2)$  операций, а собственное разложение —  $O(N^3)$ , что делает метод неприменимым для очень больших графов. Для оптимизации в коде используются методы библиотеки Eigen, но в данном проекте не планируется такое количество данных, время обработки которых будет превышать 30 секунд.

## 2.5 Генетический алгоритм

В задачах оптимизации, особенно тех, что характеризуются высокой раз мерностью пространства поиска, наличием множества локальных экстремумов или сложностью аналитического описания целевой функции, традиционные методы, такие как градиентный спуск или жадные алгоритмы, становятся неэффективными. Генетические алгоритмы, вдохновленные механизмами биологической эволюции и естественного отбора, представляют собой мощный метаэвристический<sup>8</sup> подход к решению подобных задач[6]. Они находят применение в самых разных областях, включая инженерию, машинное обучение, планирование и процедурную генерацию контента, благодаря способности эффективно исследовать сложные пространства решений и находить близкие к оптимальным результаты без необходимости исчерпывающего анализа всех возможных

<sup>8</sup>Метаэвристический — это обобщенный алгоритмический подход, использующий абстрактные принципы для эффективного поиска близких к оптимальным решений в сложных задачах оптимизации без строгой зависимости от их специфики.

вариантов.

Формализация задачи для генетического алгоритма заключается в поиске решения, которое максимизирует или минимизирует заданную целевую функцию  $f(\mathbf{x})$ , где  $\mathbf{x}$  — элемент пространства поиска  $X$ . Пространство  $X$  может быть дискретным, непрерывным или комбинированным, а целевая функция часто недифференцируема или содержит множество ограничений. Решение кодируется в виде особи (хромосомы), которая представляет собой структурированное описание потенциального кандидата. Целевая функция, также называемая функцией приспособленности, оценивает качество каждой особи, присваивая ей числовое значение, которое отражает ее соответствие заданным критериям. Задача состоит в том, чтобы итеративно эволюционировать популяцию особей, находя ту, которая обеспечивает экстремальное значение  $f(\mathbf{x})$ , при этом удовлетворяя поставленным требованиям.

Применение генетического алгоритма основано на итеративном процессе, моделирующем эволюцию популяции. Каждая особь в популяции представляет потенциальное решение, а ее приспособленность определяет вероятность ее выживания и размножения. Алгоритм использует операторы, имитирующие биологические процессы: отбор, скрещивание и мутацию. Отбор выделяет наиболее приспособленные особи, которые с большей вероятностью передают свои характеристики потомкам. Скрещивание комбинирует части хромосом двух родительских особей, создавая новые решения, которые наследуют черты предков. Мутация вносит случайные изменения в хромосомы, способствуя разнообразию популяции и предотвращая застревание в локальных экстремумах. Этот процесс повторяется на протяжении нескольких поколений, пока не будет достигнуто удовлетворительное решение или не выполнится критерий остановки.

Алгоритм начинается с инициализации популяции, состоящей из случайно сгенерированных особей. Размер популяции, обычно варьирующийся от десятков до сотен особей, выбирается с учетом компромисса между вычислительны-

ми затратами и разнообразием решений. Каждая особь оценивается по функции приспособленности, которая зависит от специфики задачи. На этапе отбора применяются различные стратегии, такие как турнирный отбор, при котором из небольшого случайного подмножества особей выбирается лучшая, или пропорциональный отбор, где вероятность выбора пропорциональна приспособленности. Скрещивание реализуется через операторы, такие как одноточечное или многоточечное скрещивание, при которых части хромосом обмениваются между родителями. Мутация выполняется с низкой вероятностью, изменяя случайные элементы хромосомы. Новый поколение формируется из отобранных особей, их потомков и, при необходимости, небольшого числа элитных особей, сохраняемых без изменений для предотвращения потери лучших решений. Процесс продолжается до достижения критерия остановки, после чего выбирается особь с наивысшей приспособленностью.

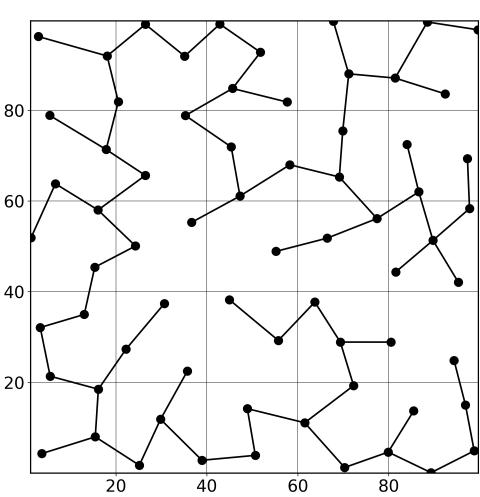
Анализ вычислительной сложности генетического алгоритма зависит от размеров популяции  $P$ , числа поколений  $G$ , сложности вычисления функции приспособленности и операций над хромосомами. Инициализация популяции обычно имеет сложность  $O(P \cdot l)$ , где  $l$  — длина хромосомы, определяемая структурой кодирования решения. Вычисление приспособленности для всей популяции требует  $O(P \cdot C_f)$ , где  $C_f$  — сложность функции приспособленности, которая может варьироваться от  $O(1)$  для простых аналитических функций до  $O(n^k)$  для задач, связанных с обработкой графов или симуляциями. Операции отбора, скрещивания и мутации для одной особи обычно имеют сложность  $O(l)$ , что для всей популяции составляет  $O(P \cdot l)$ . Таким образом, сложность одного поколения равна  $O(P \cdot (C_f + l))$ , а для  $G$  поколений —  $O(G \cdot P \cdot (C_f + l))$ . В задачах с большими популяциями или сложными функциями приспособленности доминирующим фактором становится  $C_f$ , что требует оптимизации, например, кэширования результатов или параллельного вычисления. Для задач с умеренным размером популяции и числом поколений алгоритм демонстрирует

приемлемую производительность.

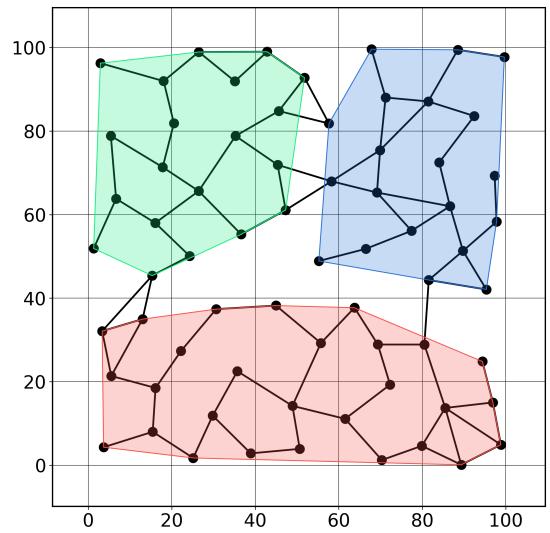
Используемые структуры данных зависят от специфики задачи, но в общем случае включают представления хромосом и механизмы управления популяцией. Хромосомы обычно кодируются в виде массивов, строк, бинарных последовательностей или более сложных структур, таких как деревья или графы, в зависимости от природы решения. Например, в задачах оптимизации параметров хромосома может быть вектором вещественных чисел, реализованным как массив `double`, а в задачах планирования — списком действий, представленным в виде связного списка или дерева. Популяция хранится как массив или список хромосом, обеспечивающий быстрый доступ для вычисления приспособленности и операций отбора. Для оценки приспособленности могут использоваться вспомогательные структуры, такие как хэш-таблицы для кэширования результатов или матрицы для хранения промежуточных вычислений, особенно в задачах, связанных с графиками. Генерация случайных чисел, необходимая для инициализации, скрещивания и мутации, осуществляется с использованием высококачественных генераторов псевдослучайных чисел, таких как Мерсенн-Твистер, обеспечивающих равномерное распределение и воспроизводимость. В задачах с ограничениями часто применяются множества или словари для проверки допустимости решений, что ускоряет обработку сложных условий.

Генетические алгоритмы обладают универсальностью и гибкостью, позволяя адаптировать их к широкому спектру задач путем настройки кодирования хромосом, функции приспособленности и эволюционных операторов. Их способность находить близкие к оптимальным решения в условиях неопределенности и нелинейности делает их ценным инструментом в современных вычислительных приложениях, от оптимизации инженерных конструкций до генерации игрового контента. В то же время, успех алгоритма зависит от правильного выбора параметров и структуры данных, а также от баланса между исследованием пространства поиска и эксплуатацией найденных решений, что требует тщательной

настройки и, в некоторых случаях, интеграции с другими методами оптимизации.



(а) Построение минимального остовного дерева алгоритмом Краскала



(б) Распределение класторов гарафа алгоритмом спектральной кластеризации

Рисунок 3. Отображение результатов работы алгоритмов при помощи инструментов языка программирования Python

## 2.6 Поиск по дереву Монте-Карло

В задачах с неполной информацией оптимальное планирование является вычислительно трудной задачей из-за «проклятия размерности»<sup>9</sup> пространства состояний и наблюдений. Стандартные алгоритмы машинного обучения, где агенту нужно принимать решения в условиях неполной или неточной информации о состоянии среды, становятся неэффективными. Именно из-за этого часто применяются методы снижения размерности перед анализом высокоразмерных данных. Однако, в условиях, когда критически важно сохранить и учесть все

<sup>9</sup>Проклятие размерности (англ. curse of dimensionality) — термин, описывающий ряд проблем, которые возникают при работе с данными или алгоритмами в пространствах высокой размерности, с большим числом переменных или признаков.

переменные и признаки среды, используется алгоритм поиска по дереву Монте-Карло[5; 13]. Он помогает смягчить проклятие размерности за счет того, что не пытается явно перечислять и оценивать все возможные состояния огромного пространства, а вместо этого использует выборочную симуляцию и строит дерево поиска только там, где это наиболее важно для принятия решения.

Алгоритмы типа Monte Carlo Tree Search (MCTS) строят дерево поиска на основе статистических симуляций и могут со временем сходиться к оптимальному решению. Такое построение позволяет гибко планировать, не требуя полного перебора всех ветвей, но гарантирует только асимптотическую оптимальность без конечновременных гарантий качества.

Будем считать среду поиска по дереву Монте-Карло с частичной наблюдаемостью, формально задаваемой кортежем  $(S, A, T, R, \Omega, O, \gamma)$ , где  $S$  – конечное множество состояний,  $A$  – действий,  $T(s, a, s') = P(s' | s, a)$  – вероятности перехода между состояниями,  $R(s, a)$  – функция немедленного вознаграждения,  $\Omega$  – множество наблюдений,  $O(o | s', a)$  – вероятности получения наблюдения  $o$  после перехода в состояние  $s'$  при действии  $a$ , а  $\gamma \in [0, 1]$  – дисконт-фактор<sup>10</sup>. Агент не видит истинное состояние  $s$ , вместо этого он получает наблюдения  $o \in \Omega$  и поддерживает состояние убеждения  $b$ , т.е. вероятностное распределение по возможным состояниям. Модель неполной информации требует, чтобы политика  $\pi$  отображала текущее распределение  $b$  в действие, а не текущее скрытое состояние. Цель агента – максимизировать математическое ожидание суммы дисконтированных вознаграждений  $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$ , где  $r_t = R(s_t, a_t)$ . При этом задание включает ограничение неполной информации, то есть агент не знает  $s_t$  точно и может лишь обновлять  $b$  на основе наблюдений.

В POMDP состояние убеждения  $b$  является достаточной статистикой ис-

---

<sup>10</sup>Дисконт-фактор (или дисконтирование наград) – это число  $\gamma$  из интервала от 0 до 1, которое используется для того, чтобы уменьшать "ценность" наград, полученных в будущем, по сравнению с наградами, которые получены сейчас.

тории. Формально  $b_t(s)$  – вероятность того, что мир в состоянии  $s$  в момент  $t$ . При выборе действия  $a_t$  и наблюдении  $o_{t+1}$  новое состояние убеждения  $b_{t+1}$  вычисляется по правилу Байеса:

$$b_{t+1}(s') = \eta O(o_{t+1} | s', a_t) \sum_{s \in S} T(s, a_t, s') b_t(s),$$

где  $\eta$  – нормировочная константа. Функция вознаграждения в терминах убеждений определяется как  $R(b, a) = \sum_{s \in S} b(s) R(s, a)$ . Целевая функция – ожидаемый дисконтированный доход при стратегии  $\pi$  – записывается аналогично MDP, но с учетом переходов по убедительным состояниям и наблюдениям. При этом пространство состояний убеждений бесконечномерно, и точное решение (обозначаемое  $\pi^*$ ) находится через решение «belief-MDP», что обычно вычислительно невыполнимо. Методы типа POMCTS обходятся без полного хранения belief-MDP и используют статистические методы («сэмплирование»). Например, алгоритм POMCP аппроксимирует текущее  $b$  набором частиц и использует генератор модели (симулятор), выбирая в каждой симуляции случайное состояние  $s \sim b$  для развертывания дерева поиска. Таким образом, математическое ядро POMCTS – это комбинаторика Байесова обновления  $b$  и критериев выбора действия (UCB), при этом теория гарантирует сходимость к оптимуму при бесконечном числе симуляций, но с практическими ограничениями по времени и ресурсам.

Алгоритм POMCTS строит поиск в пространстве историй (последовательности действий и наблюдений), поэтому дерево поиска содержит узлы наблюдений и узлы действий в чередующемся порядке. Основные этапы одной итерации поиска можно сформулировать так:

1. Выборка (Selection). Начиная с корневого узла (текущего убеждения  $b_t$ ), рекурсивно выбираем действие по правилу UCB1: для каждого возмож-

ногого действия  $a$  вычисляем оценку

$$Q^+(h, a) = Q(h, a) + c \sqrt{\frac{\ln N(h)}{N(h, a)}},$$

где  $Q(h, a)$  – среднее вознаграждение,  $N(\cdot)$  – счетчики посещений, а  $c$  – константа баланса «исследование/эксплуатация». При этом в начале симуляции истинное состояние  $s_0$  берется случайно из текущего убеждения  $b_t$ . Действие с наибольшей  $Q^+$  выбирается, и симуляция переходит в узел следующего наблюдения.

2. Расширение (Expansion). Если при выборе действий мы попадаем в лист дерева (новая история), то создается новый узел. После выполнения действия  $a$  получено наблюдение  $o$ ; состояние убеждения обновляется по формуле Байеса, и создается новый узел наблюдения для этой пары  $(a, o)$ . Таким образом, на каждом шаге после выбора действия дерево роста расширяется узлом наблюдения, соответствующим реальному наблюдению.
3. Симуляция (Rollout). Из нового узла (или из существующего листа) запускается рандомизированная «прокрутка» (rollout): выбираются последующие действия по простому эвристическому правилу (например, случайно). Симуляция продолжается до заранее заданной глубины или до терминального условия, накапливая получаемое вознаграждение. Это соответствует оценке последствий действий без дальнейшего построения дерева.
4. Обратное распространение (Backpropagation). Полученное кумулятивное вознаграждение из симуляции «распространяется» обратно вверх по дереву: для каждого посещенного узла (истории  $h$  и действия  $a$ ) увеличиваются счетчики  $N(h)$ ,  $N(h, a)$  и обновляется среднее  $Q(h, a)$  (обычно, как среднее по всем симуляциям через этот узел). Это позволяет постепенно уточнять оценки ценности узлов.

После множества итераций выбора–расширения–симуляции–обратного распро-

странения алгоритм выбирает в корне действие с наибольшим оценочным  $Q$ . Затем, когда агент выполняет это реальное действие и получает реальное наблюдение, соответствующая ветвь дерева остается, а все остальные ветви удаляются – т.е. дерево «урезается» до узла, соответствующего новой истории (предыдущее  $h$  дополняется этим  $a, o$ ). Далее поиск продолжается из обновленного узла по той же схеме.

Метод POMCTS обладает рядом достоинств и недостатков. К преимуществам относятся масштабируемость и гибкость, то есть он не требует явного полного описания модели и может работать с генератором состояний, адаптируясь к большим деревьям принятия решений. При достаточном количестве симуляций алгоритм сходится к оптимальному решению. Однако, на практике вычислительные ресурсы ограничены. Основная трудность – это быстрое разрастание дерева из-за большого числа возможных действий и наблюдений («проклятие размерности» и «проклятие истории»). В результате требуется очень много симуляций, чтобы получить надежные оценки  $Q$ . К тому же аппроксимация состояния убеждения частицами может приводить к ошибкам представления, требуя большого их числа. Алгоритм не гарантирует оптимальность за конечное время, а балансировка константы  $c$  критична для качества планирования.

Возможные оптимизации включают переиспользование и обобщение состояний убеждения, параллельные симуляции, эвристики для приближенных оценок узлов и прогрессивное расширение при большом пространстве действий/наблюдений. С точки зрения математики, алгоритм POMCTS опирается на классические результаты UCT: при условии, что каждое действие достаточно часто исследуется, оценки  $Q(h, a)$  сходятся к истинным ценностям действий. На практике POMCTS хорошо подходит для онлайн-планирования в сложных POMDP, но требует тщательной настройки и ресурсных затрат.

# Глава 3. Описание реализации проекта

## 3.1 Процедурная генерация игровой локации

### 3.1.1 Распределение точек интереса

Выбранный жанр проекта (пошаговая стратегия) подразумевает создание сбалансированного игрового пространства, в котором игрок имеет возможность управлять перемещением персонажа, обеспечивая продвижение по сценарию игры. В данном проекте игровое пространство представляет собой топологию из связанных между собой ключевых узлов. Каждый узел является вершиной графа, которая представляет определенный функционал в зависимости от своего назначения.

Построение локации начинается с распределения узлов по заданному прямоугольному пространству определенных заранее размеров. На данном этапе генерации размер узла не имеет значения, а лишь его координаты в двумерном пространстве. Для реализации алгоритма Fast Poisson Disk Sampling, обеспечивающего равномерное и случайное распределение узлов с минимальным расстоянием между ними, разработана структура данных, представляющая узел, и набор функций для управления процессом генерации.

Структура узла реализована следующим образом:

Листинг 1. Структура узла

```
struct Knot {  
    float x, y;  
    Knot() : x(-1.0f), y(-1.0f) {}  
    Knot(float _x, float _y) : x(_x), y(_y) {}  
    bool operator==(const Knot& another) const;  
    bool operator<(const Knot& another) const;  
};
```

Данная структура содержит координаты узла в двумерном пространстве

и предоставляет конструкторы для инициализации как пустого узла с координатами по умолчанию, так и узла с заданными координатами. Операторы сравнения обеспечивают возможность сортировки и проверки равенства узлов, что используется для их хранения и обработке.

Для реализации алгоритма Fast Poisson Disk Sampling разработана функция `poissonDiskSampling`, которая принимает параметры ширины и высоты прямоугольного пространства, минимальный радиус между узлами и количество попыток генерации новых узлов. Алгоритм опирается на двумерную сетку для эффективной проверки расстояний между узлами и активный список для управления процессом генерации.

Основной процесс начинается с создания двумерной сетки, размер ячеек которой определяется как  $\frac{r}{\sqrt{2}}$ , где  $r$  — заданный минимальный радиус. Это обеспечивает, что диагональ ячейки равна  $r$ , гарантируя не более одного узла на ячейку. Первая точка генерируется со случайными координатами внутри заданного пространства и добавляется в сетку, результирующий массив узлов и активный список.

### Листинг 2. Инициализация сетки и первой точки

```
float cellsize = static_cast<float>(radius / std::sqrt(N));
cellsize = std::max(cellsize, 1.0f);
int ncells_width = static_cast<int>(std::ceil(width / cellsize)) + 1;
int ncells_height = static_cast<int>(std::ceil(height / cellsize)) + 1;
std::vector<std::vector<Knot>> grid(ncells_width,
                                         std::vector<Knot>(ncells_height));
Knot k0(dist_x(gen), dist_y(gen));
insertKnot(grid, k0, cellsize);
knots.push_back(k0);
active.push_back(k0);
```

Функция `insertKnot` вычисляет индексы ячейки на основе координат узла и размера ячейки, добавляя узел в соответствующую позицию сетки. Это

позволяет быстро находить соседние узлы при проверке минимального расстояния.

Далее алгоритм итерируется, пока активный список не опустеет. На каждой итерации случайным образом выбирается активный узел, вокруг которого предпринимаются до  $k = 30$  попыток генерации новой точки. Новая точка создается в кольце радиусом от  $r$  до  $2r$  под случайным углом, что соответствует формулам:

$$x_{\text{new}} = p.x + \text{radius} \cdot \cos(\theta), \quad y_{\text{new}} = p.y + \text{radius} \cdot \sin(\theta),$$

где  $\theta$  — случайный угол, а  $\text{radius}$  — случайное расстояние в диапазоне  $[r, 2r]$ . Код генерации новой точки представлен следующим образом:

### Листинг 3. Генерация новой точки

```
std::uniform_real_distribution<float> angle_dist(0.0f, 2.0f *
    static_cast<float>(M_PI));
float theta = angle_dist(gen);
std::uniform_real_distribution<float> radius_dist(radius, 2.0f * radius);
float random_radius = radius_dist(gen);
Knot knew(
    active_knot.x + random_radius * std::cos(theta),
    active_knot.y + random_radius * std::sin(theta)
);
```

Каждая сгенерированная точка проходит проверку на валидность с помощью функции `isKnotValid`, которая подтверждает, что точка находится внутри границ пространства и не нарушает условие минимального расстояния  $r$  до соседних узлов. Проверка осуществляется путем анализа соседних ячеек в радиусе  $2r$ , что оптимизирует процесс, избегая сравнения со всеми узлами.

### Листинг 4. Проверка валидности точки

```

bool isKnotValid(const std::vector<std::vector<Knot>>& grid, int g_width, int
g_height,
    const Knot& knot, float radius, float width, float height, float cellsize) {
    if (knot.x < 0 || knot.x >= width || knot.y < 0 || knot.y >= height) {
        return false;
    }
    int xindex = static_cast<int>(std::floor(knot.x / cellsize));
    int yindex = static_cast<int>(std::floor(knot.y / cellsize));
    int i0 = std::max(xindex - 2, 0);
    int i1 = std::min(xindex + 2, g_width - 1);
    int j0 = std::max(yindex - 2, 0);
    int j1 = std::min(yindex + 2, g_height - 1);
    for (int i = i0; i <= i1; ++i) {
        for (int j = j0; j <= j1; ++j) {
            const Knot& other = grid[i][j];
            if (other.x != -1 && distanceBetweenPoints(other.x, other.y, knot.x,
knot.y) < radius) {
                return false;
            }
        }
    }
    return true;
}

```

Если точка валидна, она добавляется в сетку, результирующий массив и активный список. В противном случае, после  $k$  неудачных попыток активный узел удаляется из списка. Удаление реализовано с использованием функции `erase`, что в текущей реализации приводит к временной сложности  $O(n^2)$  из-за линейного времени удаления. Планируется оптимизация с использованием структуры дерева для достижения сложности  $O(\log n)$ .

#### Листинг 5. Удаление активного узла

```
if (!found) { active.erase(active.begin() + random_knot_index); }
```

Полученные узлы используются в проекте как опорные точки для постро-

ения маршрутов, размещения квестов и задания топологии игрового мира. Алгоритм обеспечивает равномерное распределение узлов, что способствует созданию сбалансированного игрового пространства, где игрок может свободно перемещаться между ключевыми точками, избегая избыточной кластеризации объектов.

## 3.2 Построение топологии игровой локации

После генерации равномерно распределенных узлов с использованием алгоритма Fast Poisson Disk Sampling следующим этапом в создании игровой локации является формирование связной топологии, которая определяет маршруты перемещения игрока между ключевыми узлами. Этот процесс включает построение триангуляции Делоне, создание минимального остовного дерева с использованием алгоритма Краскала и добавление случайных ребер для повышения вариативности путей. Целью данного этапа является создание сбалансированного и интерактивного игрового пространства, обеспечивающего как структурную целостность, так и разнообразие маршрутов.

Для начального связывания узлов применяется триангуляция Делоне, которая обеспечивает создание набора треугольников, максимизирующих минимальный угол каждого треугольника, что предотвращает появление слишком узких или вытянутых фигур. Это свойство делает триангуляцию Делоне подходящей для построения базовой структуры связей между узлами, так как она минимизирует вероятность создания неестественных или неудобных для игрока путей. В проекте используется алгоритм Бойера-Уотсона, который итеративно добавляет узлы к триангуляции, начиная с супертреугольника, охватывающего все узлы.

Процесс начинается с определения границ множества узлов и создания супертреугольника, размеры которого достаточно велики, чтобы охватить все

узлы. Координаты вершин супертреугольника вычисляются на основе минимальных и максимальных координат узлов, увеличенных на коэффициент, пропорциональный максимальной разнице координат по осям. Это гарантирует, что все узлы находятся внутри начального треугольника.

#### Листинг 6. Создание супертреугольника

```
float minX = std::numeric_limits<float>::max();
float minY = std::numeric_limits<float>::max();
float maxX = std::numeric_limits<float>::lowest();
float maxY = std::numeric_limits<float>::lowest();
for (const auto& knot : knots) {
    if (knot.x < minX) minX = knot.x;
    if (knot.y < minY) minY = knot.y;
    if (knot.x > maxX) maxX = knot.x;
    if (knot.y > maxY) maxY = knot.y;
}
float dx = maxX - minX;
float dy = maxY - minY;
float deltaMax = std::max(dx, dy);
float midX = (minX + maxX) / 2.0f;
float midY = (minY + maxY) / 2.0f;
Triangle superTriangle = {
    {midX - 20.0f * deltaMax, midY - 10.0f * deltaMax},
    {midX, midY + 20.0f * deltaMax},
    {midX + 20.0f * deltaMax, midY - 10.0f * deltaMax}
};
triangles.push_back(superTriangle);
```

Каждый узел добавляется к триангуляции путем определения треугольников, в описанную окружность которых он попадает («плохие» треугольники). Эти треугольники удаляются, а их ребра, не разделяемые другими «плохими» треугольниками, формируют полигон, который затем используется для создания новых треугольников с добавляемым узлом. Проверка попадания узла в

описанную окружность треугольника осуществляется через вычисление центра и радиуса окружности, что требует расчета определителя для определения параметров окружности.

#### Листинг 7. Проверка узла в описанной окружности

```
bool isKnotInCircumcircle(const Triangle& triangle, const Knot& knot) {
    Knot a = triangle.knot_1;
    Knot b = triangle.knot_2;
    Knot c = triangle.knot_3;

    float d = (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y));
    if (d == 0) return false;

    float ux = ((a.x * a.x + a.y * a.y) * (b.y - c.y) +
                ((b.x * b.x + b.y * b.y) * (c.y - a.y)) +
                ((c.x * c.x + c.y * c.y) * (a.y - b.y)));
    float uy = ((a.x * a.x + a.y * a.y) * (c.x - b.x) +
                ((b.x * b.x + b.y * b.y) * (a.x - c.x)) +
                ((c.x * c.x + c.y * c.y) * (b.x - a.x)));
    Knot center(ux / (2 * d), uy / (2 * d));
    float radius_sq = (center.x - a.x) * (center.x - a.x) +
                      (center.y - a.y) * (center.y - a.y);
    float dist_sq = (knot.x - center.x) * (knot.x - center.x) +
                    (knot.y - center.y) * (knot.y - center.y);
    return dist_sq <= radius_sq;
}
```

После обработки всех узлов треугольники, содержащие вершины супертреугольника, удаляются, оставляя только те, которые формируют триангуляцию Делоне. Этот процесс имеет вычислительную сложность  $O(n^2)$  в худшем случае из-за необходимости проверки каждого треугольника для каждого узла, хотя на практике средняя сложность ближе к  $O(n \log n)$  для случайных наборов точек. Полученные треугольники преобразуются в список уникальных ребер, которые служат основой для следующего этапа — построения минимального

остовного дерева.

### Листинг 8. Удаление супертреугольника

```
triangles.erase(std::remove_if(triangles.begin(), triangles.end(),
    [&superTriangle](const Triangle& t) {
        return t.knot_1 == superTriangle.knot_1 || t.knot_1 ==
superTriangle.knot_2 || t.knot_1 == superTriangle.knot_3 ||
        t.knot_2 == superTriangle.knot_1 || t.knot_2 == superTriangle.knot_2
|| t.knot_2 == superTriangle.knot_3 ||
        t.knot_3 == superTriangle.knot_1 || t.knot_3 == superTriangle.knot_2
|| t.knot_3 == superTriangle.knot_3;
}), triangles.end());
```

На основе ребер, полученных из триангуляции, строится минимальное остовное дерево с использованием алгоритма Краскала. Этот алгоритм обеспечивает создание связного графа с минимальной суммарной длиной ребер, что как нельзя хорошо подходит для построения базовых маршрутов локации, гарантируя связность всех узлов. Алгоритм Краскала включает сортировку ребер по весу, использование структуры данных «Система непересекающихся множеств» (Disjoint Set Union, DSU) для отслеживания связности и выбор ребер, не создающих циклы.

Сначала ребра триангуляции преобразуются в ребра с весами, где вес равен евклидову расстоянию между узлами. Затем ребра сортируются по возрастанию веса, что позволяет алгоритму Краскала начинать с самых коротких ребер, минимизируя общую длину остова.

### Листинг 9. Подготовка и сортировка ребер

```
for (const auto& edge : edges) {
    float dx = edge.second.x - edge.first.x;
    float dy = edge.second.y - edge.first.y;
    float weight = std::sqrt(dx * dx + dy * dy);
    weightedEdges.emplace_back(edge.first, edge.second, weight);
```

```

}

std::sort(weightedEdges.begin(), weightedEdges.end(),
[](const EdgeWithWeight& a, const EdgeWithWeight& b) {
    return a.weight < b.weight;
});

```

Структура DSU используется для эффективного объединения узлов в компоненты связности. Операции `makeSet`, `find` и `unite` реализованы с оптимизациями сжатия путей и объединения по рангу, что обеспечивает амортизированную сложность операций, близкую к  $O(1)$ .

Листинг 10. Объединение узлов в DSU

```

Knot DisjointSetUnion::find(const Knot& k) {
    if (parent[k] == k) return k;
    return parent[k] = find(parent[k]);
}

void DisjointSetUnion::unite(const Knot& a, const Knot& b) {
    Knot rootA = find(a);
    Knot rootB = find(b);
    if (rootA == rootB) return;
    if (rank[rootA] < rank[rootB]) {
        parent[rootA] = rootB;
    }
    else {
        parent[rootB] = rootA;
        if (rank[rootA] == rank[rootB]) rank[rootA]++;
    }
}

```

Алгоритм Краскала итерируется по отсортированным ребрам, добавляя те, которые соединяют различные компоненты связности, в результирующий остов. Сложность этой операции составляет  $O(E \log E)$ , где  $E$  — количество ребер, из-за сортировки, а операции DSU добавляют незначительные затраты

благодаря оптимизациям.

### Листинг 11. Построение MST

```
std::vector<EdgeWithWeight> mst;
for (const auto& edge : weightedEdges) {
    if (dsu.find(edge.start) != dsu.find(edge.end)) {
        mst.push_back(edge);
        dsu.unite(edge.start, edge.end);
    }
}
```

Для повышения вариативности маршрутов к полученному MST добавляются случайные ребра из исходного набора ребер триангуляции. Вероятность добавления ребра определяется параметром `probability`, а длина ребра ограничивается 120% от средней длины всех ребер, чтобы избежать добавления слишком длинных путей, которые могут нарушить баланс локации. Этот процесс использует хэш-множество для исключения уже существующих ребер MST и генератор случайных чисел для выбора новых ребер.

### Листинг 12. Добавление случайных ребер

```
float total_length = 0.0f;
for (const auto& edge : all_edges) {
    float dx = edge.second.x - edge.first.x;
    float dy = edge.second.y - edge.first.y;
    total_length += std::sqrt(dx * dx + dy * dy);
}
float average_length = total_length / all_edges.size();
float max_allowed_length = average_length * 1.20f;
std::unordered_set<Edge, EdgeHash> mst_set(mst_edges.begin(), mst_edges.end());
static std::mt19937 gen(std::random_device{}());
std::uniform_real_distribution<float> dist(0.0f, 1.0f);
for (const auto& edge : all_edges) {
    if (mst_set.find(edge) != mst_set.end()) {
```

```

        continue;
    }

    if (dist(gen) < probability) {
        float dx = edge.second.x - edge.first.x;
        float dy = edge.second.y - edge.first.y;
        float length = std::sqrt(dx * dx + dy * dy);
        if (length <= max_allowed_length) {
            result.push_back(edge);
        }
    }
}

```

Общая вычислительная сложность процесса формирования топологии складывается из нескольких этапов. Триангуляция Делоне имеет сложность  $O(n^2)$  в худшем случае, хотя средняя сложность ближе к  $O(n \log n)$ . Построение MST с использованием алгоритма Краскала требует  $O(E \log E)$  для сортировки ребер и почти линейного времени для операций DSU. Добавление случайных ребер занимает  $O(E)$  из-за перебора всех ребер и константного времени проверки хэш-множества. Таким образом, доминирующим фактором является триангуляция, что делает общую сложность  $O(n^2)$  в худшем случае. Оптимизация триангуляции, например, с использованием инкрементальных алгоритмов или структур данных, таких как квадродеревья, может снизить эту сложность в будущих итерациях проекта.

Полученная топология, состоящая из ребер MST и дополнительных случайных ребер, формирует сеть маршрутов, по которым игрок может перемещаться между ключевыми узлами. Минимальное оствовное дерево гарантирует связность всех узлов с минимальной общей длиной путей, что способствует созданию логичной и эффективной структуры локации. Случайные ребра добавляют альтернативные пути, повышая реиграбельность и разнообразие игрового опыта. Эта структура используется для размещения игровых объектов, квестов

и событий, обеспечивая сбалансированное и увлекательное игровое пространство.

### 3.3 Размещение квестовых триггеров

После формирования топологии игровой локации, включающей узлы, связанные ребрами минимального оствового дерева и дополнительными случайными путями, следующим этапом является размещение сценарных (квестовых) триггеров<sup>11</sup>. Эти триггеры представляют собой ключевые события или объекты, такие как точки появления персонажа, встречи с квестодателями (*quest\_giver*) или сражения, которые определяют повествовательную структуру и игровой процесс. Для обеспечения сбалансированного и увлекательного опыта требуется оптимальное распределение триггеров по узлам с учетом желаемых расстояний от начальной точки, минимальных расстояний между определенными триггерами и топологических ограничений графа. В проекте для решения этой задачи применяется генетический алгоритм, который позволяет эффективно оптимизировать размещение триггеров в условиях сложного пространства поиска с множеством ограничений[10].

Генетический алгоритм моделирует процесс естественного отбора для поиска оптимального решения. В контексте размещения квестовых триггеров каждая особь (хромосома) представляет собой возможное распределение тегов триггеров по узлам графа. Целевая функция приспособленности оценивает качество такого распределения, учитывая соответствие фактических расстояний между узлами желаемым значениям, соблюдение минимальных расстояний между определенными тегами и уникальность назначений. Алгоритм итеративно эволюционирует популяцию хромосом через процессы отбора, скрещивания и му-

---

<sup>11</sup>Триггер — это механизм, активируемый действиями игрока или выполнением условий, который запускает предопределенное событие.

тации, стремясь максимизировать приспособленность и найти оптимальное или близкое к оптимальному решение[9].

Особь (или хромосома) реализована как структура `Chromosome`, содержащая вектор идентификаторов узлов, где каждый индекс соответствует определенному тегу из списка `tag_list`. Например, если список тегов включает «quest\_giver\_heron» и «wolf\_attack\_1», то хромосома определяет, какой узел назначен каждому из этих тегов. Особые теги `spawn`, `asylum` и `exit` фиксированы на узлах с идентификаторами `spawn_node_id`, `asylum_node_id` и `exit_node_id`, что отражает начальную точку игрока в локации, точку места убежища и точку перехода на другую локацию. Данные узлы определяются до запуска генетического алгоритма. Узлы `spawn` и `exit` определены как вершины графа, находящиеся на минимально-максимальном расстоянии (по количеству вершин) друг от друга, а узел `asylum` определяется как вершина графа с наименьшим эксцентриситетом.

#### Листинг 13. Структура хромосомы

```
struct Chromosome {  
    std::vector<int> node_ids; // node_ids[i] - ID узла для тега i  
};
```

Процесс начинается с инициализации популяции, состоящей из хромосом `population_size`, каждая из которых представляет случайное распределение тегов по узлам. Для обеспечения уникальности назначений, за исключением узла «`spawn`», используется случайная перестановка доступных идентификаторов узлов. Узлы `spawn`, `asylum` и `exit` фиксируются и задаются в соответствии с `spawn_node_id`, `asylum_node_id` и `exit_node_id`, что гарантирует соблюдение начальных условий игрового сценария.

#### Листинг 14. Инициализация популяции

```
std::vector<Chromosome> QuestPlacement::initialize_population(int num_nodes, int
```

```

num_tags, std::mt19937& rng) {
    std::vector<Chromosome> population;
    int spawn_tag_idx = SPECIALTAG; // SPECIALTAG = -1 (нemагическое число)
    for (size_t i = 0; i < tag_list.size(); ++i) {
        if (tag_list[i] == <<spawn>>) {
            spawn_tag_idx = i;
            break;
        }
    }
    for (int i = 0; i < population_size; ++i) {
        Chromosome chrom;
        chrom.node_ids.resize(num_tags);
        std::vector<int> available_nodes;
        for (int j = 0; j < num_nodes; ++j) if (j != spawn_node_id)
            available_nodes.push_back(j);
        std::shuffle(available_nodes.begin(), available_nodes.end(), rng);
        if (spawn_tag_idx != SPECIALTAG) {
            chrom.node_ids[spawn_tag_idx] = spawn_node_id;
        }
        int avail_idx = 0;
        for (int j = 0; j < num_tags; ++j) {
            if (j != spawn_tag_idx) {
                chrom.node_ids[j] = available_nodes[avail_idx++];
            }
        }
        population.push_back(chrom);
    }
    return population;
}

```

Для оценки приспособленности хромосомы необходимо вычислить расстояния между узлами в терминах количества ребер в графе. Расстояния от узла «spawn» до всех остальных узлов определяются с помощью поиска в ширину

(BFS), который эффективно обходит граф, используя список смежности. Кроме того, для проверки минимальных расстояний между тегами применяется алгоритм Флойда-Уоршалла, который строит матрицу кратчайших путей между всеми парами узлов. Эти предварительные вычисления обеспечивают быстрый доступ к топологической информации во время оценки приспособленности.

Листинг 15. Вычисление расстояний от узла spawn

```
std::vector<int> compute_distances(const std::unordered_map<Knot,
    std::vector<Knot>, std::hash<Knot>>& adj_list,
    const std::vector<Knot>& knots, int start_id) {
    std::vector<int> distances(knots.size(), -1);
    std::queue<int> q;
    distances[start_id] = 0;
    q.push(start_id);
    while (!q.empty()) {
        int curr = q.front();
        q.pop();
        const Knot& current_knot = knots[curr];
        auto it = adj_list.find(current_knot);
        if (it == adj_list.end()) continue;
        for (const auto& nb : it->second) {
            int nb_id = -1;
            for (size_t i = 0; i < knots.size(); ++i) {
                if (nb.x == knots[i].x && nb.y == knots[i].y) {
                    nb_id = i;
                    break;
                }
            }
            if (nb_id != -1 && distances[nb_id] == -1) {
                distances[nb_id] = distances[curr] + 1;
                q.push(nb_id);
            }
        }
    }
}
```

```
    }

    return distances;
}
```

Функция приспособленности `compute_fitness` является ключевым компонентом генетического алгоритма, оценивая качество хромосомы по нескольким критериям, чтобы обеспечить соответствие повествовательным и геймплейным требованиям. Во-первых, проверяется уникальность назначений узлов: каждый тег должен соответствовать уникальному узлу, иначе хромосома получает штраф  $-1e9$ , исключающий ее из дальнейшего рассмотрения. Во-вторых, подтверждается корректность фиксации тегов `spawn`, `asylum` и `exit` на заданных узлах (`spawn_node_id`, `asylum_node_id`, `exit_node_id`); несоответствие также приводит к штрафу  $-1e9$ . Основной вклад в приспособленность вносится минимизацией абсолютной разницы между фактическими расстояниями от узла `spawn` до узлов, назначенных тегам, и желаемыми расстояниями, заданными в `desired_distances`. Для каждого тега  $t_i$  штраф вычисляется как  $-|actual\_distance - desired\_distance|$ , где `actual_distance` берется из вектора `distances`. Дополнительно функция проверяет минимальные расстояния между парами тегов, указанные в `min_distance_constraints`. Если расстояние между узлами, назначенными тегам `tag1` и `tag2`, меньше `min_dist`, добавляется штраф  $-(min\_dist - dist) \times 10$ , где расстояние берется из матрицы `dist_matrix`. Коэффициент 10 усиливает значимость этих ограничений, обеспечивая их приоритетное соблюдение. Функция поддерживает множественные экземпляры тегов, проверяя все пары таких узлов на соответствие минимальному расстоянию. Эта гибкость позволяет моделировать сложные повествовательные структуры, где одни и те же события могут происходить в нескольких местах, сохраняя топологическую согласованность.

Листинг 16. Вычисление приспособленности

```

float QuestPlacement::compute_fitness(const Chromosome& chrom, const
    std::vector<int>& distances,
    const std::vector<std::vector<int>>& dist_matrix) {
    if (distances.size() != knots.size() || dist_matrix.size() != knots.size()) {
        throw std::runtime_error("Invalid distances or dist_matrix size");
    }
    std::set<int> assigned_nodes;
    for (int node_id : chrom.node_ids) {
        if (node_id < 0 || node_id >= static_cast<int>(knots.size())) {
            return -1e9;
        }
        if (assigned_nodes.count(node_id)) return -1e9;
        assigned_nodes.insert(node_id);
    }
    int spawn_tag_idx = -1, asylum_tag_idx = -1, exit_tag_idx = -1;
    for (size_t i = 0; i < tag_list.size(); ++i) {
        if (tag_list[i] == "spawn") spawn_tag_idx = i;
        else if (tag_list[i] == "asylum") asylum_tag_idx = i;
        else if (tag_list[i] == "exit") exit_tag_idx = i;
    }
    if (spawn_tag_idx != -1 && chrom.node_ids[spawn_tag_idx] != spawn_node_id)
        return -1e9;
    if (asylum_tag_idx != -1 && chrom.node_ids[asylum_tag_idx] !=
        asylum_node_id) return -1e9;
    if (exit_tag_idx != -1 && chrom.node_ids[exit_tag_idx] != exit_node_id)
        return -1e9;
    float fitness = 0.0f;
    for (size_t i = 0; i < tag_list.size(); ++i) {
        std::string tag = tag_list[i];
        int node_id = chrom.node_ids[i];
        if (desired_distances.find(tag) == desired_distances.end()) { return
-1e9; }
        int actual_distance = distances[node_id];

```

```

        int desired = desired_distances.at(tag);
        fitness -= std::abs(actual_distance - desired);
    }

    for (const auto& [tag1, tag2, min_dist] : min_distance_constraints) {
        auto it1 = std::find(tag_list.begin(), tag_list.end(), tag1);
        auto it2 = std::find(tag_list.begin(), tag_list.end(), tag2);
        if (it1 == tag_list.end() || it2 == tag_list.end()) continue;
        int idx1 = it1 - tag_list.begin();
        int idx2 = it2 - tag_list.begin();
        int node1 = chrom.node_ids[idx1];
        int node2 = chrom.node_ids[idx2];
        int dist = dist_matrix[node1][node2];
        if (dist < min_dist) fitness -= (min_dist - dist) * 10.0f;
    }

    return fitness;
}

```

На каждой итерации генетического алгоритма формируется новое поколение через отбор, скрещивание и мутацию. Отбор реализован с использованием модифицированного турнирного метода. Все хромосомы популяции ранжируются по убыванию приспособленности, от лучшей к худшой. Затем случайным образом выбирается `tournament_size` хромосом, которые также сортируются по приспособленности относительно друг друга. Каждой хромосоме в турнире присваивается вероятность выбора, пропорциональная квадрату ее ранга  $rank^2$ , где ранг равен `tournament_size` для лучшей хромосомы и 1 для худшей. Например, при `tournament_size = 3` вероятности выбора лучшей, средней и худшей хромосом составляют примерно 64%, 29% и 7% соответственно. Это обеспечивает предпочтение более приспособленным особям, сохраняя шанс выбора менее приспособленных для поддержания разнообразия. Скрещивание комбинирует двух родительских хромосом, случайным образом выбирая узлы из одного из родителей с учетом уникальности назначений и фиксации тегов `spawn`, `asylum`, `exit`.

Мутация с вероятностью `mutation_rate * (num_tags - 1)` выбирает случайный изменяемый тег (исключая `spawn`, `asylum`, `exit`) и переносит его на случайный соседний узел текущего узла в графе, определенный списком смежности. Если соседний узел свободен, тег перемещается туда, освобождая предыдущий узел, а если узел занят другим изменяемым тегом, выполняется их обмен. Фиксированные узлы остаются нетронутыми, а уникальность назначений сохраняется.

### Листинг 17. Скрещивание хромосом

```
Chromosome QuestPlacement::crossover(const Chromosome& parent1, const
    Chromosome& parent2,
    int num_nodes, std::mt19937& rng) {
    Chromosome child;
    child.node_ids.resize(parent1.node_ids.size());
    std::set<int> assigned;
    // Назначение фиксированных тегов (spawn, asylum, exit)
    for (size_t i = 0; i < parent1.node_ids.size(); ++i) {
        if (fixed_tags.count(tag_list[i])) continue;
        int node = (dist(rng) == 0) ? parent1.node_ids[i] : parent2.node_ids[i];
        if (assigned.count(node)) {
            std::vector<int> available;
            for (int j = 0; j < num_nodes; ++j) if (!assigned.count(j))
                available.push_back(j);
            if (!available.empty()) node = available[rng() % available.size()];
            else node = parent1.node_ids[i];
        }
        child.node_ids[i] = node;
        assigned.insert(node);
    }
    return child;
}
```

Основной цикл генетического алгоритма выполняется в течение определенного количества итераций `generations`. На каждой итерации вычисляется

приспособленность текущей популяции, создается новое поколение через отбор, скрещивание и мутацию, после чего популяция обновляется. По завершении выбирается хромосома с наивысшей приспособленностью, и ее назначения преобразуются в словарь, сопоставляющий теги узлам.

Листинг 18. Основной цикл генетического алгоритма

```
std::map<std::string, int> QuestPlacement::run_genetic_algorithm(std::mt19937&
rng) {
    int num_nodes = knots.size();
    int num_tags = tag_list.size();
    std::vector<int> distances = compute_distances(adjacency_list, knots,
spawn_node_id);
    std::vector<std::vector<int>> dist_matrix = floyd_marshall(adjacency_list,
knots);
    auto population = initialize_population(num_nodes, num_tags, rng);
    std::vector<float> fitnesses(population_size);
    for (int gen = 0; gen < generations; ++gen) {
        for (int i = 0; i < population_size; ++i) {
            fitnesses[i] = compute_fitness(population[i], distances,
dist_matrix);
        }
        std::vector<Chromosome> new_population;
        while (new_population.size() < population_size) {
            auto parent1 = tournament_selection(population, fitnesses, 3, rng);
            auto parent2 = tournament_selection(population, fitnesses, 3, rng);
            auto child = crossover(parent1, parent2, num_nodes, rng);
            mutate(child, num_nodes, rng);
            new_population.push_back(child);
        }
        population = new_population;
    }
    float best_fitness = -std::numeric_limits<float>::max();
    Chromosome best_chrom;
```

```

    for (const auto& chrom : population) {
        float fit = compute_fitness(chrom, distances, dist_matrix);
        if (fit > best_fitness) {
            best_fitness = fit;
            best_chrom = chrom;
        }
    }

    std::map<std::string, int> tag_assignments;
    for (size_t i = 0; i < tag_list.size(); ++i) {
        tag_assignments[tag_list[i]] = best_chrom.node_ids[i];
    }

    return tag_assignments;
}

```

Вычислительная сложность генетического алгоритма определяется несколькими факторами. Инициализация популяции занимает  $O(P \cdot T)$ , где  $P$  — размер популяции (`population_size`), а  $T$  — количество тегов (`num_tags`), из-за необходимости генерации случайных перестановок. Вычисление расстояний с помощью BFS имеет сложность  $O(V + E)$ , где  $V$  — количество узлов, а  $E$  — количество ребер в графе. Алгоритм Флойда-Уоршалла требует  $O(V^3)$  для построения матрицы кратчайших путей. Функция приспособленности для одной хромосомы выполняется за  $O(T + C)$ , где  $C$  — количество ограничений минимальных расстояний (`min_distance_constraints`), а для всей популяции — за  $O(P \cdot (T + C))$ . Турнирный отбор, скрещивание и мутация для одной хромосомы имеют сложность  $O(T)$ , что для нового поколения составляет  $O(P \cdot T)$ . Таким образом, сложность одной итерации равна  $O(P \cdot (T + C))$ , а для  $G$  поколений —  $O(G \cdot P \cdot (T + C))$ . Предварительные вычисления расстояний добавляют  $O(V^3)$ , что может быть доминирующим фактором для больших графов. В текущей реализации параметры выбраны умеренными ( $P = 100$ ,  $G = 50$ ,  $T$  и  $C$  порядка десятков), что обеспечивает приемлемую производительность, хотя

оптимизация, такая как кэширование матрицы расстояний или использование инкрементальных алгоритмов для графов, может улучшить масштабируемость.

Полученное распределение тегов определяет, где в игровой локации будут размещены ключевые элементы сценария, такие как квестодатели (например, "quest\_giver\_heron" "quest\_giver\_cobra"), зоны сражений (например, "heron\_battle" "wolf\_attack\_1") или точки получения артефактов (например, "cobra\_hunters" для получения "TravelerBoots"). Генетический алгоритм гарантирует, что эти элементы расположены в соответствии с повествовательными и геймплейными требованиями, определенными в файле `DomainDatabase.json`. Например, тег "wolf\_attack\_4" (сражение с альфа-волком) размещается на желаемом расстоянии 20 узлов от точки появления, а минимальное расстояние между "quest\_giver\_heron" и "quest\_giver\_cobra" составляет 5 узлов, что предотвращает их кластеризацию. Эта структура поддерживает динамическое развитие сюжета, позволяя игроку взаимодействовать с квестами в логичной и увлекательной последовательности, с учетом топологии локации и повествовательных ограничений.

### 3.4 Взаимодействие с базой сценариев

После размещения квестовых тегов на узлах игровой локации с использованием генетического алгоритма следующим этапом является интеграция этих тегов с повествовательной структурой игры, определенной в файле `DomainDatabase.json`. Этот файл содержит описание игровых существ, начальных состояний, действий, квестов, правил активации и ограничений на размещение, обеспечивая связь между топологией локации и сценарием игры. Формирование базы сценариев и ее взаимодействие с алгоритмом размещения квестовых тегов позволяют создавать динамичные, логически согласованные сюжетные события, поддерживая увлекательный игровой процесс.

Файл `DomainDatabase.json` — это структурированный JSON-документ, содержащий разделы, описывающие аспекты игрового мира. Раздел `objects` определяет сущности: персонажей (например, "Hero", "Heron", "AlphaWolf") с атрибутами, такими как здоровье или роль в квестах; предметы (например, "TravelerBoots"); и узлы ("knots") с тегами, такими как "spawn" или "quest\_giver\_heron". Раздел `facts` задает начальное состояние мира при помощи заранее определенных предикатов, включая местоположение персонажей (например, "at(Hero, spawn)") и их статус (например, "alive(Hero)"). Раздел `actions` описывает действия игрока, такие как "accept\_heron\_quest" или "fight\_one\_wolf", с предусловиями, эффектами, затратами времени и вкладом в напряженность. Раздел `quests` структурирует квесты ("Heron Quest", "Cobra Quest", "Wolf Quest") через этапы, условия активации и выборы игрока. Раздел `rules` содержит правила активации этапов квестов, связывающие условия (например, "at(Hero, wolf\_attack\_1)") с эффектами ("trigger\_stage(wolf\_quest, 0)"). Раздел `constraints` задает ограничения на размещение тегов, включая желаемые расстояния от "spawn" ("desired") и минимальные расстояния между тегами ("min"). Раздел `tension_profile` определяет изменение напряженности для каждого действия, регулируя эмоциональный ритм игры.

Листинг 19. Пример структуры `DomainDatabase.json`

```
"objects": [
    {"type": "character", "id": "Hero", "properties": {...}},
    {"type": "knot", "id": "spawn", "properties": {"tags": ["spawn"]}},
    {"type": "knot", "id": "quest_giver_heron", "properties": {"tags": [
        "quest_giver_heron"]}}],
"facts": ["at(Hero, spawn)", "alive(Hero)", "at(Heron, quest_giver_heron)"],
"actions": [{"name": "accept_heron_quest",
```

```

"preconditions": ["at(Hero, quest_giver_heron)", "!ally(Hero, Heron)"] ,
"effects": ["ally(Hero, Heron)", "start_heron_quest"]...} ,
"quests": [{

  "name": "Heron Quest",

  "stages": [{

    "id": 0,
    "condition": "at(Hero, quest_giver_heron)",
    "description": "Meeting the heron",
    "choices": [
      {"text": "Accept heron quest", "action": "accept_heron_quest",
       "next_stage": 1}]]}]] ,
"rules": [ {

  "name": "heron_quest_trigger",
  "condition": "at(Hero, quest_giver_heron)",
  "effect": "trigger_stage(heron_quest, 0)"}] ,
"constraints": [
{"type": "distance", "tag1": "spawn", "tag2": "quest_giver_heron", "desired": 5},
 {"type": "distance", "tag1": "quest_giver_heron", "tag2": "quest_giver_cobra",
  "min": 5}], "tension_profile": [
 {"action": "accept_heron_quest", "tension": "+2"}]

```

Формирование базы сценариев начинается с проектирования повествовательной структуры, которая поддерживает игровые цели и разнообразие взаимодействий. Этот процесс включает определение персонажей, предметов, узлов и событий, составляющих основу сюжета. Квесты состоят из этапов, каждый из которых активируется при достижении узла с определенным тегом и предлагает игроку выборы, влияющие на состояние мира. Проектирование требует учета зависимостей между `objects`, `facts` и `actions`, чтобы обеспечить логическую последовательность событий. Например, действие `"accept_heron_quest"` требует факта `"at(Hero, quest_giver_heron)"` и отсутствия `"ally(Hero, Heron)"`, что проверяется в `preconditions`.

Генетический алгоритм, реализованный в классе `QuestPlacement`, взаимодействует с `DomainDatabase.json` через следующие компоненты:

- `tag_list` — список тегов, извлеченный из `objects` для узлов типа `knot`.
- `desired_distances` — отображение тегов на желаемые расстояния от узла с тегом `spawn`, полученные из `constraints` с полем `desired`.
- `min_distance_constraints` — набор ограничений на минимальные расстояния между тегами, извлеченный из `constraints` с полем `min`.
- `spawn_node_id`, `asylum_node_id`, `exit_node_id` — фиксированные идентификаторы узлов, соответствующие `at(Hero, spawn)`, `asylum` и `exit`.

Эти данные передаются в конструктор `QuestPlacement`, обеспечивая размещение тегов в соответствии с повествовательными требованиями. Функция `compute_fitness` интегрирует ограничения из `constraints`, минимизируя отклонения от желаемых расстояний и штрафуя нарушения минимальных расстояний. Парсинг `DomainDatabase.json` имеет сложность  $O(O+F+A+Q+R+C)$ , где  $O, F, A, Q, R, C$  — число объектов, фактов, действий, квестов, правил и ограничений, что незначительно из-за их малого размера. Извлечение `tag_list` и `constraints` занимает  $O(T+C)$ . Основная нагрузка связана с BFS ( $O(V+E)$ ), Флойдом-Уоршаллом ( $O(V^3)$ ) и итерациями алгоритма ( $O(G \cdot P \cdot (T + C))$ ).

Интеграция с `DomainDatabase.json` обеспечивает согласованность сюжета. Например, `Heron Quest` активируется правилом `heron_quest_trigger` при достижении `quest_giver_heron`, запуская `accept_heron_quest`, добавляющее `ally(Hero, Heron)`. Алгоритм размещает `quest_giver_heron` примерно в 5 ребрах от `spawn` и не менее чем в 5 ребрах от `quest_giver_cobra`. `Wolf Quest` использует теги `wolf_attack_1-4` с расстояниями 2, 7, 15, 20 ребер, создавая нарастающую сложность, отраженную в `tension_profile` (+3 до +6). Множественные `cobra_hunters` поддерживают вариативность событий, сохраняя минимальный интервал 8 ребер. Эта структура обеспечивает гармоничное сочетание.

тание топологии и сценария, создавая динамичный и увлекательный игровой процесс.

## 3.5 Адаптивные боевые сценарии

### 3.5.1 Формализация правил сражений

Система сражений, реализованная в данной игре, представляет собой детерминированную пошаговую игру с несовершенной информацией<sup>12</sup>, участниками которой являются игрок и адаптивный ИИ-противник. Сражения происходят на игровом поле переменной формы и размеров, определяемом в начале каждой боевой сессии.

#### Караван и его ограничения

До начала сражения игрок формирует свой «караван» из фигур, выбирая ограниченное их число, каждая из которых стилизована под определенное животное. Устанавливаются следующие правила:

- Игрок обязан выбрать хотя бы одну фигуру.
- В караване игрока не может быть более одной фигуры каждого типа одновременно.
- В отличие от игрока, ИИ может использовать произвольное количество фигур одного типа, что создает асимметрию условий боя.

#### Ранговая система и типы фигур

Каждая фигура имеет закрепленный за ней ранг  $X \in \mathbb{N}$ , определяющий ее позицию в боевой иерархии. Правила атаки формализуются следующим об-

<sup>12</sup>Игра с несовершенной информацией — это тип игровой модели, в которой участники не обладают полной осведомленностью о текущем состоянии всех элементов системы и стратегических намерениях оппонента. В контексте представленной боевой системы это выражается в недоступности игроку полной информации о составе армии ИИ-противника, расположении его фигур и их скрытых способностях до момента их активации.

разом:

Фигура с рангом  $X$  может атаковать фигуры с рангами  $R$ , где  $R \in [1, X + 1]$ .

Это отражает естественные поведенческие паттерны: например, животное среднего ранга избегает схваток с более сильными противниками, но может атаковать равных и слабейших.

*Исключение — состояние окружения.* Если фигура оказывается в позиции, где нет ни одной безопасной клетки для отступления, она переходит в режим **яростной защиты**, в котором ей разрешено атаковать любого противника из числа окружающих, независимо от иерархических ограничений:

$$\text{Если } \psi_g \in \text{окружении}, \text{ то } R \in [1, R_{\max}], \quad (6)$$

где  $R_{\max}$  — максимальный ранг среди фигур противника.

### Правила передвижения

Фигуры могут перемещаться согласно следующим правилам. Фигура с радиусом шага  $r \in \mathbb{N}$  может совершать ходы по квадратной доске. Множество допустимых ходов  $S(r)$  для фигуры с радиусом  $r \in \mathbb{N}$  задается формулой:

$$S(r) = \left\{ (\Delta x, \Delta y) \in \mathbb{Z}^2 \setminus \{(0, 0)\} \mid \begin{array}{l} (1 \leq |\Delta x| \leq r \wedge \Delta y = 0) \\ \vee (1 \leq |\Delta y| \leq r \wedge \Delta x = 0) \\ \vee \left(1 \leq |\Delta x| = |\Delta y| \leq \left\lfloor \frac{r}{2} \right\rfloor\right) \end{array} \right\}$$

где  $\Delta x$  — смещение по горизонтали (+ вправо, - влево),  $\Delta y$  — смещение по вертикали (+ вверх, - вниз),  $r$  — радиус шага ( $r \geq 1$ ),  $\lfloor \cdot \rfloor$  — округление вниз до целого числа,  $\mathbb{Z}^2$  — множество всех целочисленных координат на плоскости

Базовое перемещение осуществляется на две клетки в любом из восьми направлений, при условии, что целевая клетка свободна и входит в игровую область. Некоторые фигуры обладают особыми способностями, позволяющими им

перемещаться на несколько клеток за один ход. Примеры расчетов допустимых полей для фигур с разным радиусами перемещения.

Для  $r = 2$

- Ортогональные ходы (8):

$$(\pm 1, 0), (\pm 2, 0), (0, \pm 1), (0, \pm 2)$$

- Диагональные ходы (4):

$$(\pm 1, \pm 1) \quad \left( \left\lfloor \frac{2}{2} \right\rfloor = 1 \right)$$

- Всего:  $8 + 4 = 12$  ходов

Для  $r = 3$

- Ортогональные ходы (12):

$$(\pm 1, 0), (\pm 2, 0), (\pm 3, 0), (0, \pm 1), (0, \pm 2), (0, \pm 3)$$

- Диагональные ходы (4):

$$(\pm 1, \pm 1) \quad \left( \left\lfloor \frac{3}{2} \right\rfloor = 1 \right)$$

- Всего:  $12 + 4 = 16$  ходов

Для  $r = 1$

- Ортогональные ходы (4):

$$(\pm 1, 0), (0, \pm 1)$$

- Диагональные ходы (0):

$$\emptyset \quad \left( \left\lfloor \frac{1}{2} \right\rfloor = 0 \right)$$

- Всего:  $4 + 0 = 4$  хода

## Графическая интерпретация

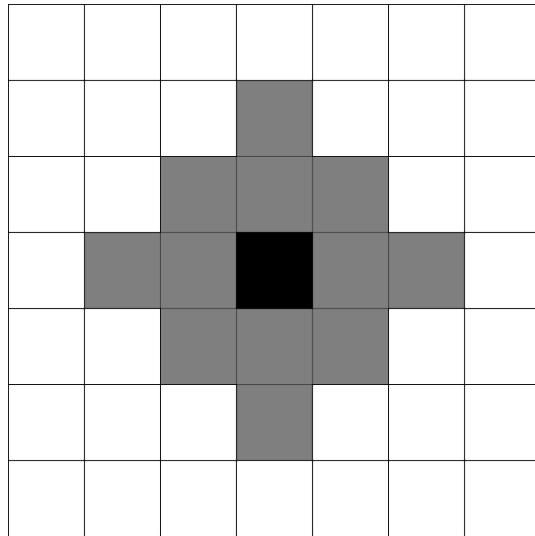


Рисунок 4. Поля для перемещения фигуры в пределах  $r = 2$  (серым)

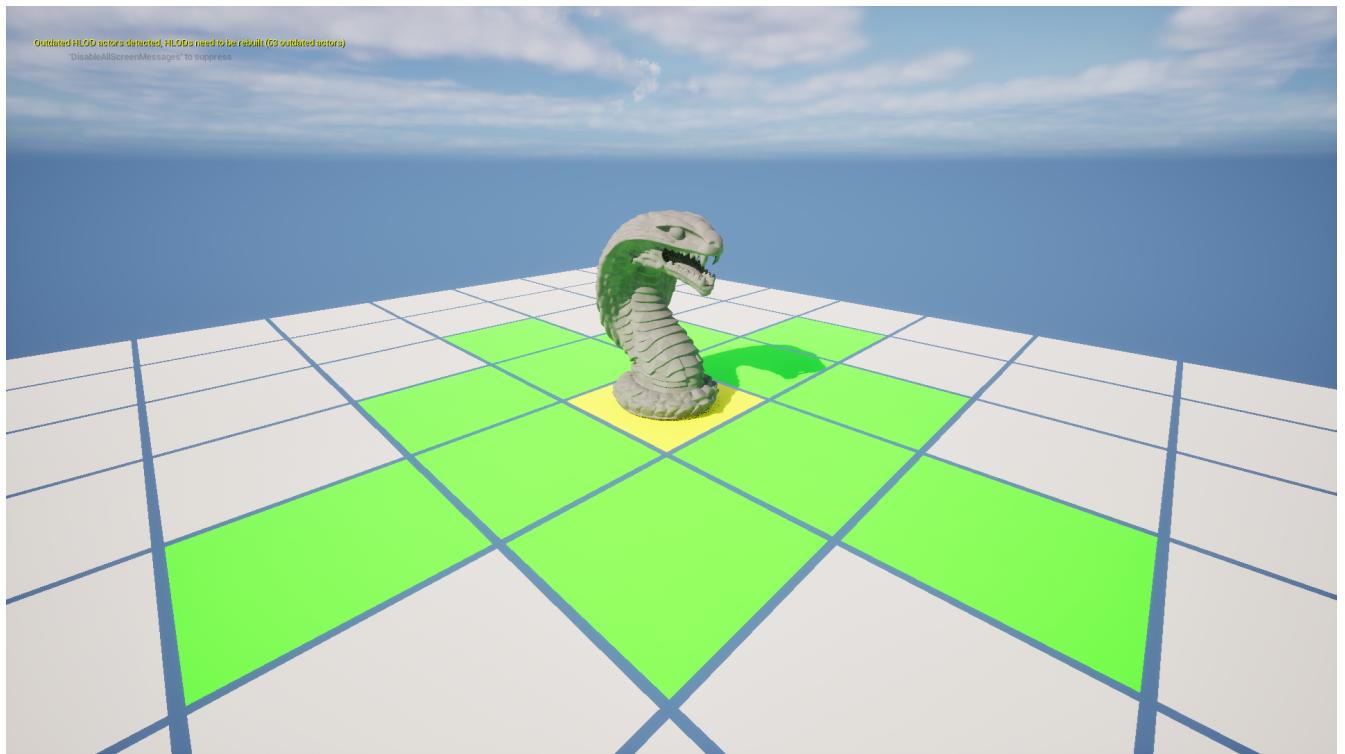


Рисунок 5. Демонстрация доступных полей для перемещений фигуры в UE-editor (желтый – исходная позиция, зеленый достижимые клетки, серый – недоступные клетки)

## Механика атакующих действий

Возможность атаки определяется следующими условиями:

- Стандартная атака возможна, если цель находится на соседней клетке;
- Некоторые фигуры имеют расширенный радиус атаки  $r > 1$ , позволяющий поражать цели на расстоянии.

### **Правило окружения**

Фигура считается окруженной, если все клетки, на которые она может переместиться, потенциально находятся под угрозой атаки со стороны противника. В этом случае активируется режим яростной защиты, как описано выше.

### **Правило стаи**

Если на поле боя присутствует несколько фигур одного типа, их атакующий потенциал возрастает линейно относительно их количества. Если базовый ранг фигуры равен  $X$ , и на ее стороне находится  $n$  союзных фигур того же типа, то разрешенный диапазон атак расширяется до

$$R \in [1, X + n]. \quad (7)$$

В таком случае, если новый ранговый диапазон позволяет юниту атаковать более сильного противника, то юниту необходимо выполнить  $X_{enemy} - X_{unit}$  атак по фигуре врага, чтобы она ушла с поля боя. Это правило стимулирует формирование «стайных» атакующих тактик и отражает эффект численного превосходства.

### **Структура поля боя**

Игровое поле содержит разнообразные элементы ландшафта:

- Препятствия, блокирующие перемещение и атаки;
- Узкие проходы, создающие стратегические точки контроля;

Конфигурация поля формируется алгоритмически в зависимости от положения сражения в графе сюжета, однако в сюжетно значимых битвах структура поля может быть задана вручную.

## **Условия победы**

Сражение считается завершенным при выполнении одного из следующих условий:

- Полное уничтожение армии противника или ее отступление;
- Выполнение заданной стратегической цели (например, захват точки, защита объекта и т.д.).

## **Адаптивная стратегия ИИ-противника**

ИИ-противник реализует эвристический подход к принятию решений, учивая:

- текущую расстановку фигур;
- соотношение рангов;
- наличие угроз окружения;

Кроме того, ИИ использует адаптивные алгоритмы обучения: анализирует стратегии, примененные игроком в предыдущих сражениях, и при возможности воспроизводит их для противодействия. Это обеспечивает нелинейную эволюцию стиля игры ИИ и усложняет предсказуемость его действий.

## **3.6 Реализация системы сражений в Unreal Engine**

Реализация системы сражений опирается на объектно-ориентированную архитектуру Unreal Engine, в которой взаимодействие классов[4] обеспечивает строгую формализацию игровых механик, включая перемещение, атаку и адаптивное поведение ИИ. Игровая логика реализована через пять основных классов: ABoard, ACell, AUnit, ATurnManager и ATacticalPlayerController. Каждый из них выполняет специализированные функции, взаимодействуя в рамках компонентной архитектуры Unreal Engine, которая позволяет эффективно управлять игровыми объектами и их поведением. Классы наследуются от

базовых классов Unreal Engine (**AActor** и **APlayerController**), что обеспечивает интеграцию с движком, включая управление сценой, рендеринг и обработку ввода.

Класс **ABoard** отвечает за создание и управление игровым полем. Он содержит сетку клеток, представленных экземплярами класса **ACell**, которые формируются в методе **GenerateField**. Каждая клетка инициализируется с координатами (**GridX**, **GridY**) и визуализируется через компонент **UStaticMeshComponent**, использующий плоский меш<sup>13</sup>. Класс **ABoard** также реализует паттерн Factory Method через метод **SpawnUnitAt**, который создает юниты (**AUnit**) различных типов (например, *Cobra*, *Elephant*) на основе шаблонов **TSubclassOf<AUt****i>**. Это позволяет динамически размещать юниты игрока и ИИ на противоположных сторонах поля, как указано в методах **SpawnPlayerUnits** и **SpawnEnemyUnits**. Метод **GetCellsInRange** обеспечивает пространственный запрос для определения клеток в радиусе действия, что используется для расчета перемещений и атак.

Класс **ACell** представляет отдельную клетку игрового поля. Он хранит информацию о координатах, занятости юнитом (**OccupyingUnit**) и визуальной подсветке через динамический материал (**DynMaterial**). Метод **SetHighlight** изменяет цвет клетки в зависимости от ее состояния (выбрана, достижима, занята врагом), обеспечивая визуальную обратную связь для игрока. Связь между **ACell** и **AUnit** реализована через двунаправленную ассоциацию: клетка может быть занята одним юнитом, а юнит всегда привязан к одной клетке (**CurrentCell**).

Класс **AUnit** моделирует игровые фигуры, каждая из которых характеризуется рангом (**Rank**), радиусом перемещения (**MovementRange**) и принадлежностью к стороне (**bIsEnemy**). Реализация методов **CanAttack** и **ReceiveHitFrom**

---

<sup>13</sup>Меш (mesh) — это цифровое представление трехмерного объекта, состоящее из вершин, ребер и граней, формирующих его геометрию и форму.

формализует правила атаки, где фигура с рангом  $X$  может атаковать противника с рангом  $R \in [1, X + 1]$ , а в состоянии яростной защиты (`bIsFrenzied`) — любого противника. Механика «стай» реализована в `CanAttack`, где количество союзных фигур того же типа увеличивает максимальный атакуемый ранг до  $X + n$ , где  $n$  — число союзников. Метод `UpdateFrenzyState` проверяет, окружена ли фигура, анализируя доступные клетки через `GetCellsInRange` и зоны угрозы противника. При получении удара (`ReceiveHitFrom`) юнит отслеживает количество атак от каждого типа противника в `IncomingHits` и уничтожается, если число ударов достигает порога, зависящего от разницы рангов.

Класс `ATurnManager` реализует паттерн State Machine, управляя чередованием ходов игрока и ИИ через перечисление `ETurnPhase`. Метод `EndTurn` переключает фазу хода, вызывая `HandleEnemyTurn` для ИИ, который выбирает случайного вражеского юнита, находит ближайшего игрока и решает атаковать или переместиться к нему. Логика ИИ в `HandleEnemyTurn` использует эвристический подход, учитывая расстояние до цели и возможность атаки, что соответствует адаптивной стратегии, описанной в формализации.

Класс `ATacticalPlayerController` обрабатывает ввод игрока, реализуя паттерны Observer и Strategy. Метод `SetupInputComponent` привязывает события клика мыши к `HandleLeftClick`, который интерпретирует действия игрока (выбор юнита, перемещение, атака) с использованием трассировки лучей (`GetHitResultUnderCursorByChannel`). Логика атаки и перемещения разделена: `TryMoveSelectedUnitToCell` перемещает юнит на свободную клетку, а `HandleLeftClick` обрабатывает атаку, проверяя условия через `CanAttack` и `ReceiveHitFrom`. Метод `ClearHighlights` сбрасывает визуальную подсветку, обеспечивая чистое состояние интерфейса после каждого действия. Паттерн Strategy проявляется в разделении логики атаки и перемещения, что позволяет легко модифицировать поведение.

### 3.6.1 Система взаимодействия классов

Взаимодействие классов в Unreal Engine организовано через компонентную архитектуру, где каждый класс наследуется от `AActor` или `APlayerController`, что позволяет использовать встроенные механизмы движка, такие как управление сценой, рендеринг и обработка событий.

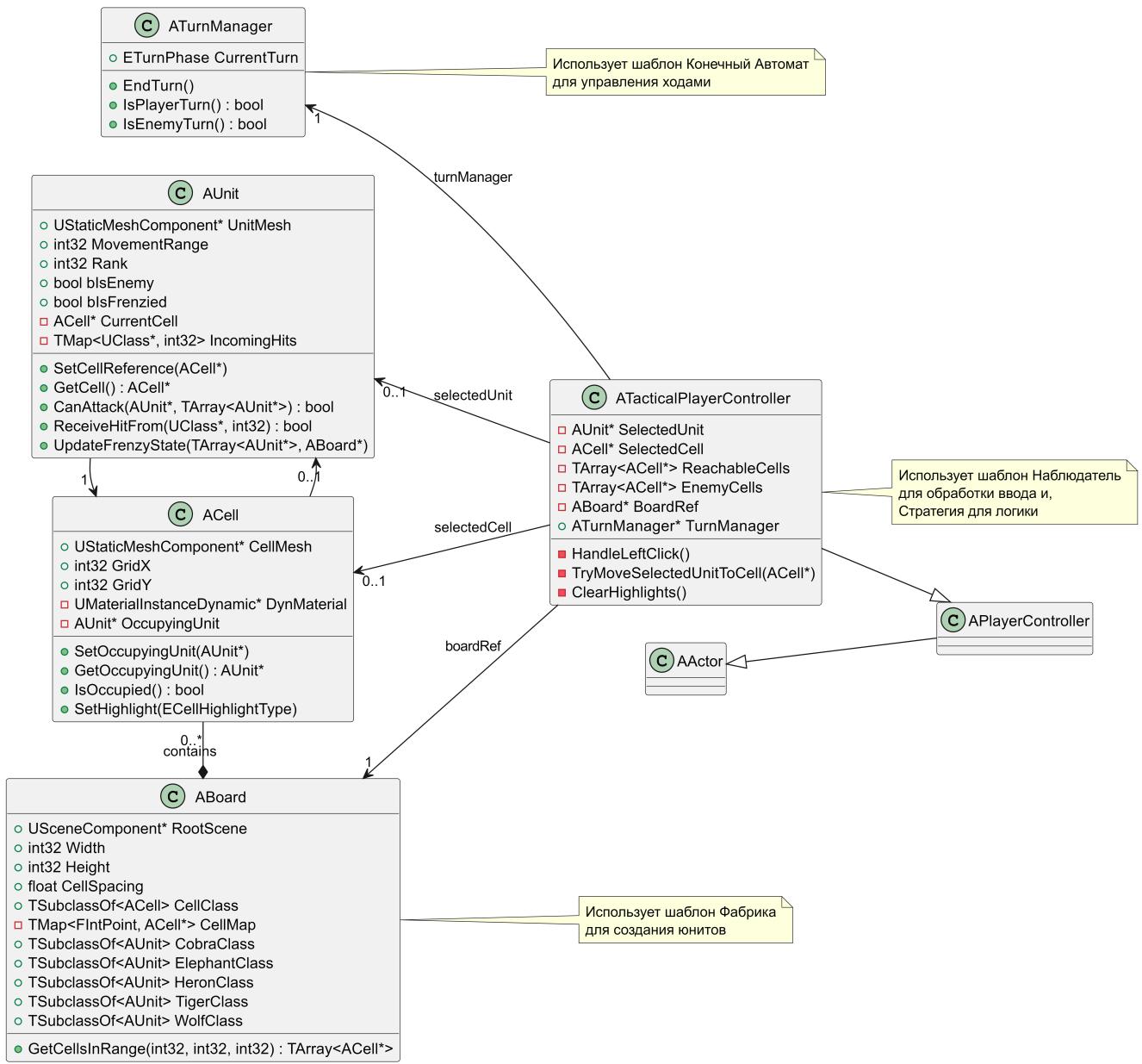


Рисунок 6. UML диаграмма классов ИИ противника

Класс `ABoard` выступает центральным компонентом, создавая и управляя сеткой клеток (`ACell`) и юнитами (`AUnit`). Композиция между `ABoard` и `ACell` ре-

ализована через `TMap<FIntPoint, ACell*>`, что обеспечивает быстрый доступ к клеткам по координатам. Ассоциация между `ACell` и `AUnit` позволяет клетке хранить ссылку на юнит, а юниту — на текущую клетку, обеспечивая двунаправленную связь для отслеживания позиций.

Класс `ATacticalPlayerController` взаимодействует с `ABoard` для получения клеток в радиусе (`GetCellsInRange`), с `AUnit` для управления выбранным юнитом (`SelectedUnit`) и с `ACell` для подсветки и перемещения (`SelectedCell`, `ReachableCells`, `EnemyCells`). Связь с `ATurnManager` позволяет синхронизировать действия игрока с фазами хода, блокируя ввод, если ход принадлежит ИИ. Метод `HandleEnemyTurn` в `ATurnManager` использует `ABoard` для пространственных запросов и `AUnit` для выполнения атак или перемещений, что обеспечивает согласованность игровой логики.

### 3.6.2 Привязка к формализации правил сражений

Формализация правил сражений, описанная ранее, напрямую реализована в коде. Правило ранговой системы воплощено в методе `CanAttack` класса `AUnit`, где проверяется условие  $R \in [1, X + 1]$ , а механика «стай» учитывает количество союзников ( $n$ ) для расширения диапазона до  $X + n$ . Состояние яростной защиты реализовано в `UpdateFrenzyState`, который анализирует доступные клетки и зоны угрозы противника, активируя `bIsFrenzied`, если отступление невозможно. Правила перемещения, заданные формулой  $S(r)$ , реализованы в `GetCellsInRange`, который вычисляет допустимые клетки на основе радиуса  $r$ , учитывая ортогональные и диагональные ходы.

Механика атакующих действий, требующая нахождения цели в радиусе атаки, поддерживается через `GetCellsInRange` и проверки в `HandleLeftClick` и `HandleEnemyTurn`. Правило окружения реализовано в `UpdateFrenzyState`, где проверяется отсутствие безопасных клеток. Условия победы, такие как уничтожение армии противника, частично реализованы через `ReceiveHitFrom`, кото-

рый уничтожает юнита при достижении порога ударов, хотя полная проверка победы требует дополнительных условий, не представленных в предоставленных файлах. Адаптивная стратегия ИИ в `HandleEnemyTurn` использует эвристику для выбора цели и действия, что соответствует описанному эвристическому подходу, хотя адаптивное обучение на основе действий игрока требует дополнительных реализаций.

### **3.7 Описание архитектуры искусственного интеллекта для тактической пошаговой системы боевых сценариев**

Разработка искусственного интеллекта (ИИ) для пошаговой тактической игры представляет собой задачу, требующую интеграцию алгоритмов принятия решений согласованных с игровыми механиками. В данном проекте ИИ реализован с использованием метода поиска по дереву Монте-Карло (Monte Carlo Tree Search, MCTS), адаптированного для управления поведением вражеских юнитов в условиях тактической игры на гексагональной сетке.

Архитектура ИИ состоит из нескольких ключевых классов, каждый из которых выполняет специализированные функции, обеспечивая модульность и гибкость системы. Основными классами являются `EnemyAIController`, `MCTS`, `MCTSNode`, `CombatState`, `TurnManager`, `Board`, `Unit` и `Cell`. Эти классы взаимодействуют для моделирования игрового мира, оценки возможных действий и принятия стратегических решений, обеспечивая динамичное и адаптивное поведение ИИ.

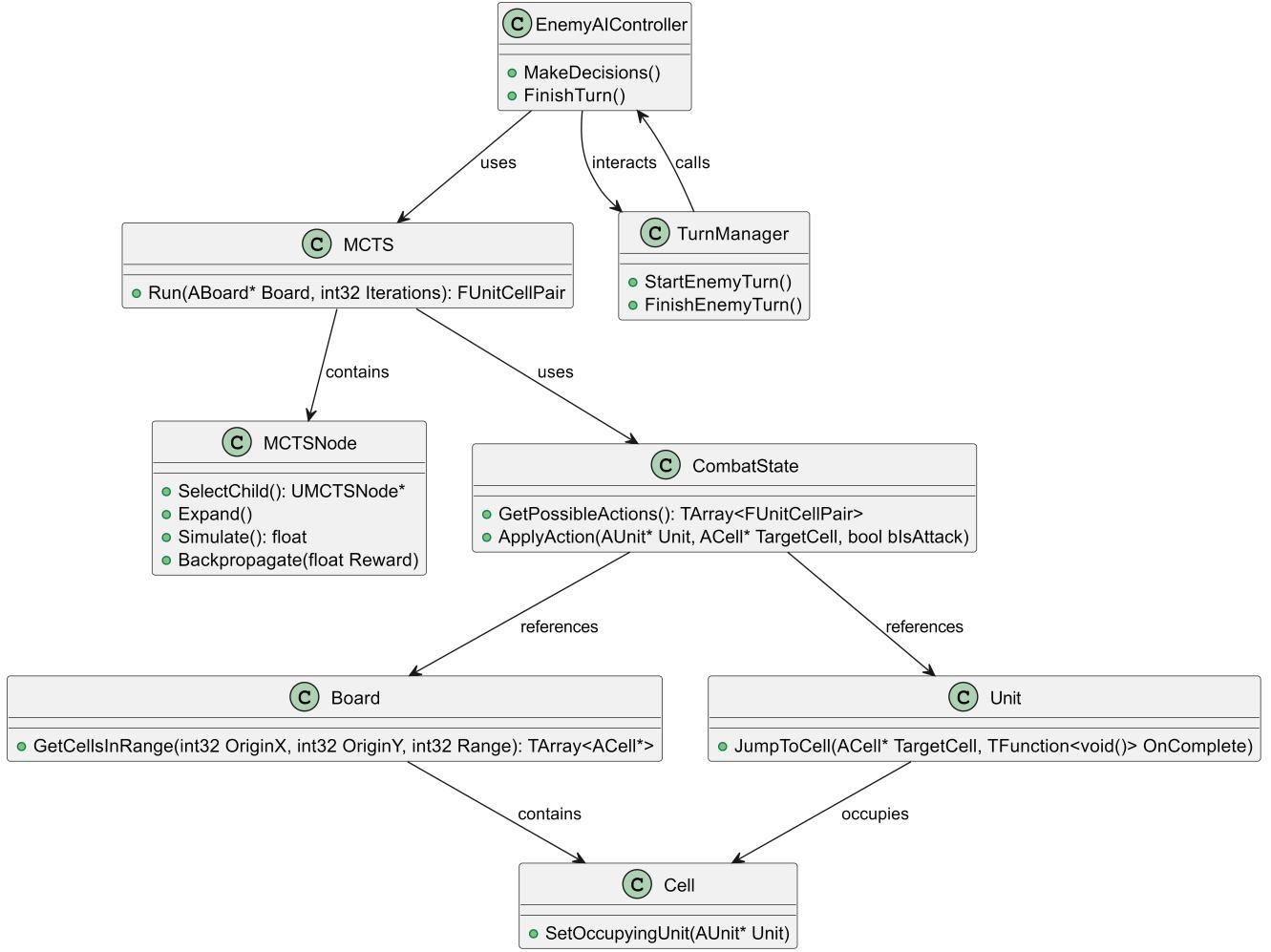


Рисунок 7. UML диаграмма классов ИИ противника

Класс `EnemyAIController` выполняет роль центрального управляющего компонента, координирующего поведение ИИ на высоком уровне. Этот класс отвечает за иницирование процесса принятия решений в начале каждого хода вражеских юнитов и применение выбранных действий к игровому миру. Он взаимодействует с другими игровыми компонентами для получения актуального состояния игры и передачи команд на выполнение действий, таких как перемещение или атака. Через метод `MakeDecisions()` класс инициирует алгоритм `MCTS`, получая оптимальное действие, которое затем применяется к игровому миру с учетом механик атаки и перемещения юнита. Интеграция результатов `MCTS` в контексте игровой логики и синхронизация с системой управления ходами

обеспечивают правильное чередование действий игрока и ИИ.

#### Листинг 20. Метод MakeDecisions

```
void AEnemyAIController::MakeDecisions()
{
    if (!BoardRef) {
        UE_LOG(LogTemp, Error, TEXT("BoardRef is null"));
        FinishTurn();
        return;
    }

    FUnitCellPair BestAction = MCTS->Run(BoardRef, 1000);
    if (!BestAction.Unit || !BestAction.Cell) {
        UE_LOG(LogTemp, Warning, TEXT("No valid action found"));
        FinishTurn();
        return;
    }

    BestAction.Unit->bHasActed = true;
    bool bIsAttack = BestAction.Cell->GetOccupyingUnit() != nullptr;
    if (bIsAttack) {
        AUnit* Target = BestAction.Cell->GetOccupyingUnit();
        bool bTargetKilled = BestAction.Unit->Attack(Target);
        if (bTargetKilled) {
            ACell* TargetCell = Target->GetCell();
            TargetCell->SetOccupyingUnit(nullptr);
            Target->Destroy();
            BestAction.Unit->JumpToCell(TargetCell, [this]() { FinishTurn(); });
        } else {
            FinishTurn();
        }
    } else {
        BestAction.Unit->JumpToCell(BestAction.Cell, [this]() { FinishTurn(); });
    }
}
```

`EnemyAIController` интегрирует результаты работы MCTS, интерпретируя их в контексте игровой механики, и обеспечивает синхронизацию с системой управления ходами, гарантируя правильное чередование действий игрока и ИИ.

Класс MCTS реализует основной алгоритм поиска по дереву Монте-Карло, который лежит в основе принятия решений ИИ. Этот алгоритм включает четыре этапа: выбор узла, расширение дерева, симуляцию и обратное распространение результатов. Принимая текущее состояние игры через метод `Run(ABoard* Board, int32 Iterations)`, MCTS выполняет итеративный поиск, оценивая возможные действия и выбирая наиболее перспективное на основе статистических данных, полученных в симуляциях. Для этого класс взаимодействует с `CombatState`, предоставляющим игровое состояние, и `MCTSNode`, формирующем структуру дерева поиска. Результат работы алгоритма передается в `EnemyAIController` для реализации в игровом мире, обеспечивая связь между вычислениями и игровой логикой.

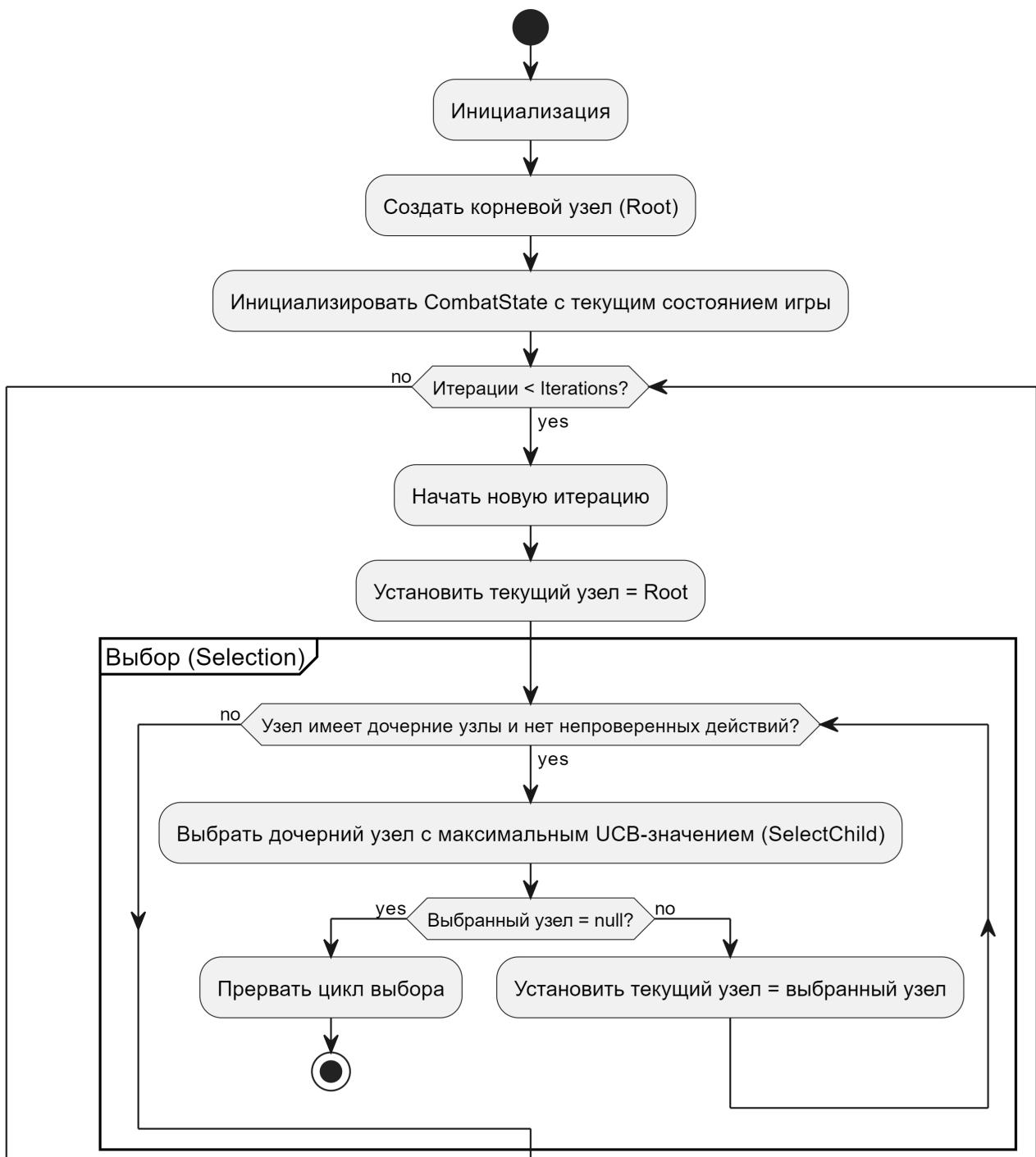


Рисунок 8. Алгоритм MCTS (часть 1)

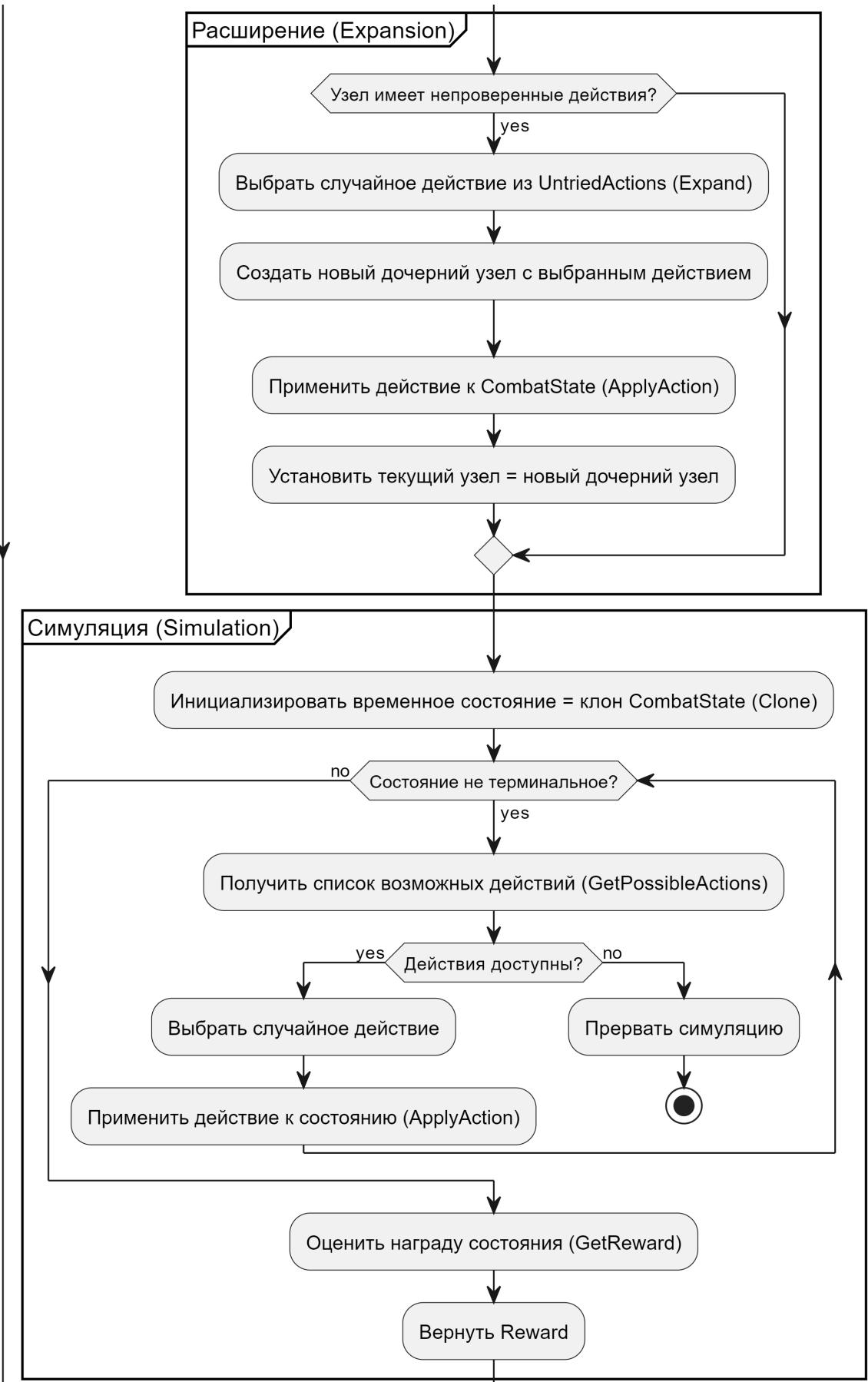


Рисунок 9. Алгоритм MCTS (часть 2)

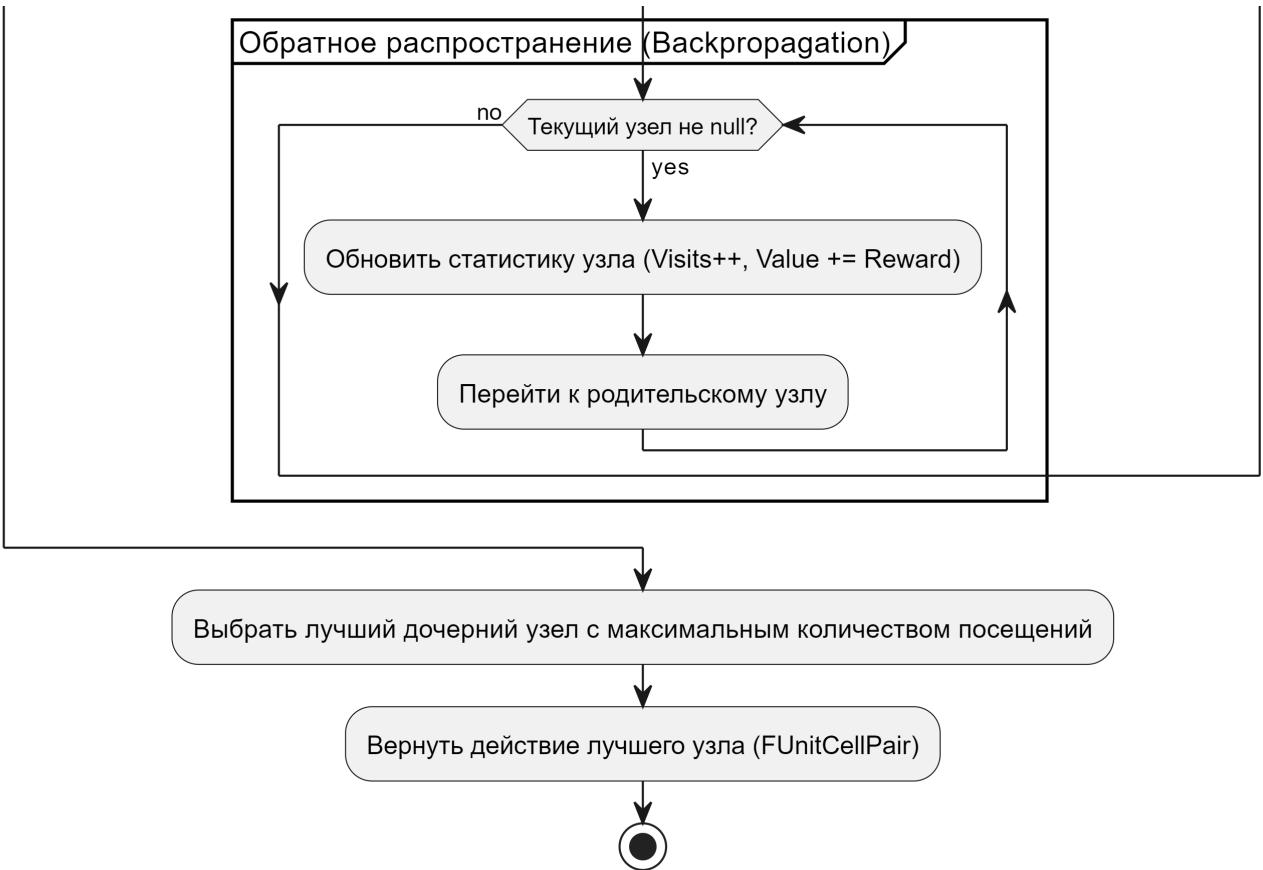


Рисунок 10. Алгоритм MCTS (часть 3)

Класс `MCTSNode` представляет отдельный узел в дереве поиска Монте-Карло. Каждый узел хранит информацию о состоянии игры, действии, приведшем к этому состоянию, статистике посещений и накопленной награде. Через методы `SelectChild()`, `Expand()`, `Simulate()` и `Backpropagate(float Reward)` он реализует этапы MCTS, поддерживая иерархическую организацию поиска через родительско-дочерние связи. Симуляции, выполняемые в `Simulate()`, оценивают ценность действий, взаимодействуя с `CombatState` для моделирования игровых сценариев. `MCTSNode` поддерживает структуру дерева, позволяя алгоритму эффективно исследовать пространство возможных ходов. Узлы взаимодействуют между собой через родительско-дочерние связи, обеспечивая иерархическую организацию поиска. Класс также отвечает за выполнение симуляций случайных игр из текущего состояния, что позволяет оценить потенциальную

ценность каждого действия.

Класс `CombatState` служит для представления игрового состояния в контексте MCTS. Он инкапсулирует информацию о текущей конфигурации игровой доски, положении юнитов, очередности ходов и других параметрах, необходимых для моделирования игры. Класс предоставляет методы для генерации списка возможных действий, применения выбранных действий к состоянию и оценки итоговой награды, которая отражает успех или неудачу ИИ. Метод `GetPossibleActions()` генерирует список доступных действий на основе информации от `Board` и `Unit`, а `ApplyAction(AUnit* Unit, ACell* TargetCell, bool bIsAttack)` применяет действие, обновляя состояние и оценивая награду. Этот класс обеспечивает независимость симуляций от реального игрового мира, позволяя MCTS проводить вычисления без влияния на фактическое состояние игры.

Класс `TurnManager` управляет последовательностью ходов в игре, обеспечивая чередование действий игрока и ИИ. Он отслеживает текущую фазу хода и инициирует соответствующие действия, такие как запуск процесса принятия решений ИИ или предоставление контроля игроку. `TurnManager` взаимодействует с `EnemyAIController` для запуска хода ИИ и с `Board` для обновления игрового состояния. Этот класс играет ключевую роль в синхронизации всех компонентов системы, гарантируя корректное выполнение игровой логики.

Класс `Board` моделирует игровую доску, представляя ее как сетку клеток, каждая из которых может быть занята юнитом или быть свободной. `Board` предоставляет интерфейс для доступа к клеткам, расчета доступных перемещений и определения зон атаки. Он служит основой для пространственной логики игры, обеспечивая `Unit` и `CombatState` необходимыми данными о топологии игрового мира. `Board` также поддерживает визуальное представление игрового поля, взаимодействуя с `Cell` для отображения состояния клеток.

Класс `Unit` представляет игровые юниты, которые могут быть подкон-

трольны игроку или ИИ. Каждый юнит характеризуется такими атрибутами, как радиус перемещения, ранг и принадлежность к стороне (игрок или ИИ). **Unit** реализует логику перемещения и атаки, взаимодействуя с **Cell** для определения текущей позиции и с **Board** для расчета возможных действий. Этот класс является основным действующим лицом в игровой механике, предоставляя данные для **CombatState** и **EnemyAIController**.

Класс **Cell** моделирует отдельную клетку игровой доски, храня информацию о ее координатах, проходимости и текущем состоянии (например, занята ли она юнитом). **Cell** взаимодействует с **Unit** для управления занятостью и с **Board** для интеграции в общую структуру игрового поля. Этот класс также поддерживает визуальное представление клетки, что важно для отображения доступных действий игроку.

Взаимодействие между классами организовано таким образом, чтобы обеспечить модульность и масштабируемость системы. **EnemyAIController** выступает в роли координатора, запрашивая действия от **MCTS** и применяя их через **Unit** и **Cell**. **MCTS** использует **MCTSNode** и **CombatState** для построения и оценки дерева поиска, полагаясь на **Board** и **Unit** для получения данных о состоянии игры. **TurnManager** синхронизирует работу всех компонентов, обеспечивая правильное чередование ходов. Такая архитектура позволяет эффективно интегрировать алгоритм **MCTS** в игровую механику, обеспечивая адаптивное и стратегическое поведение ИИ.

В контексте данной тактической пошаговой игры алгоритм Monte Carlo Tree Search играет ключевую роль в системе искусственного интеллекта противника, обеспечивая принятие стратегических решений для управления поведением вражеских юнитов. Основанный на построении дерева поиска, где каждый узел представляет состояние игры, **MCTS** использует случайные симуляции для оценки ценности возможных действий. Это позволяет искусственному интеллекту моделировать различные сценарии и выбирать оптимальные ходы с

учетом текущей конфигурации игрового поля и действий игрока. Применение в данном проекте MCTS способствует созданию динамичного и адаптивного поведения юнитов противника, что повышает сложность и увлекательность игрового процесса. Таким образом, MCTS вносит существенный вклад в обогащение стратегического опыта игрока и повышение реиграбельности, что делает его неотъемлемой частью данной игровой системы.

## Выводы и перспективы развития

В данной выпускной квалификационной работе была поставлена задача разработать прототип компьютерной игры с процедурной генерацией игровых локаций и адаптивным поведением ИИ в боевых сценариях, интегрированного в движок Unreal Engine.

Реализация системы процедурной генерации локаций основана на алгоритме Fast Poisson Disk Sampling для равномерного разброса ключевых точек на игровом поле и последующем построении связного графа с помощью алгоритмов триангуляции и минимального оствового дерева. Для размещения контента на полученном графе был разработан генетический алгоритм, в котором популяция кандидатов и функции пригодности опираются на предвычисленные расстояния между узлами и спаун-точками, что позволяет автоматически находить сбалансированные варианты расположения игровых объектов. Выбранный подход обеспечивает достаточную вариативность при повторных запусках, сохраняя контролируемость ключевых показателей, таких как равномерность распределения и достижимость целей.

Система адаптивного ИИ для боевых сценариев реализована на основе Monte Carlo Tree Search, что позволяет противникам в ходе сессий принимать решения, прогнозируя последствия действий и оценивая стратегические варианты. Интеграция MCTS выполнена через соответствующие классы и структуры данных, задействованные в игровом цикле, такие как информация о состоянии поля и агентах используется для симуляций, после чего алгоритм выбирает оптимальное действие с учетом текущей ситуации.

Все модули объединены и интегрированы в Unreal Engine версии 5.4.4. Разработаны C++ классы для процедурной генерации локаций, размещения контента, навигации и AI-контроллеры, а также реализован интерфейс запуска процедур генерации до начала уровня, инструменты визуализации и отладки.

Проведенное тестирование и анализ производительности показывают, что время процедурной генерации укладывается в приемлемые рамки для целевых платформ, а ограничение итераций в MCTS позволяет сохранять отзывчивость ИИ без значительных задержек. При этом в работе обсуждены критерии масштабируемости и методы оптимизации для будущих версий проекта. Достигнутый результат демонстрирует работоспособность предложенных подходов и их пригодность для игрового контекста.

Разработанные методы могут служить основой для коммерческого проекта или дополнения к существующим игровым механикам. Благодаря использованию Unreal Engine и стандартных C++ решений обеспечена возможность быстрой интеграции и доработки в составе команды разработчиков.

Тем не менее выявлены ограничения, связанные с балансом случайности и управляемости процедурной генерации, необходимостью дополнительной ручной фильтрации «исключительных» вариантов, а также ростом времени генерации и сложности AI при значительном увеличении масштаба локаций или глубины стратегий. Важно отметить, что оценка выполнялась преимущественно в синтетических условиях без масштабного тестирования с реальными игроками, что требует организации фокус-групп и сбора телеметрии для уточнения восприятия разнообразия и сложности.

Перспективы дальнейшего развития включают исследование альтернативных алгоритмов оптимизации размещения контента (симулированный отжиг, эвристику или обучение с подкреплением), переход к гибридным MCTS + нейросетевым подходам к обучению с подкреплением для AI, многопоточную и инкрементальную генерацию для повышения производительности. Не менее важным является проведение пользовательского тестирования, сбор аналитики и автоматическая корректировка алгоритмов на основе реальных данных, а также подготовка маркетинговых материалов и интеграция с системами распространения для выпуска игры на рынок.

# ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы были достигнуты следующие результаты:

- Реализована система процедурной генерации игровых локаций с использованием алгоритма Poisson Disk Sampling и построением связного графа с применением Delaunay-триангуляции и построение минимального оствновного дерева. Обеспечена равномерная дискретизация игрового пространства с контролем плотности узлов и связности карты.
- Разработан алгоритм размещения контента на графе игровой локации, основанной на генетическом подходе. Сформированы функции пригодности и механизмы отбора, скрещивания и мутации, что позволило автоматически находить сбалансированные конфигурации расположения ключевых объектов.
- Создана система адаптивного поведения вражеских юнитов на базе метода Monte Carlo Tree Search. ИИ-противник способен обучаться и принимать стратегические решения, моделируя последствия своих действий и реагируя на изменение игрового окружения.
- Выполнена интеграция всех модулей в игровом движке Unreal Engine 5.4.4. Разработаны C++ классы, отвечающие за генерацию локаций, размещение сценарного контента, визуализацию узлов и поведение ИИ, а также реализован интерфейс управления процедурой генерации перед стартом игрового уровня.

Разработанная система будет использована как прототип для создания компьютерной игры с высокой степенью реиграбельности и адаптивности. Она позволяет автоматически формировать уникальные локации и сценарии без необходимости ручной проработки каждого уровня. Адаптивный искусствен-

ный интеллект повышает реализм сражений и делает игровой процесс более интерактивным для игрока.

Полученные результаты обладают практической значимостью для разработки игр, ориентированных на процедурную генерацию и интеллектуальное поведение противников. Проект обладает потенциалом дальнейшего развития: возможно расширение процедурной генерации семантическими признаками, внедрение нейросетевых методов размещения контента, доработка ИИ на основе обучения с подкреплением, а также адаптация механик под многопользовательскую игру. Также запланировано тестирование с целевой аудиторией и последующий выпуск проекта на коммерческий рынок.

Таким образом, выполненная выпускная квалификационная работа достигла поставленных целей и задач. Реализованы и интегрированы ключевые компоненты процедурной генерации, размещения контента и адаптивного ИИ в рамках Unreal Engine, что подтвердило их практическую применимость и потенциал для коммерциализации. Выявленные ограничения и предложенные рекомендации открывают широкие возможности для дальнейших исследований и доработок, позволяя проекту эволюционировать от прототипа к полноценному игровому продукту.

Работа была представлена на студенческой научной конференции и заняла второе место.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Boissonnat J.-D., Dyer R., Ghosh A. Delaunay Triangulation of Manifolds. Foundations of Computational Mathematics // HAL open science. — hal.science/ : HAL, 2017.
2. C++ International Standard : ISO/IEC JTC1 SC22 WG21 N4860. — 31.03.2020. — Introduced on March 31, 2020.
3. Shaker N., Togelius J., Nelson M. J. Procedural Content Generation in Games. — Aalborg University Copenhagen : Springer, 2016. — 247 с.
4. Батлер С. и Оливер Т. Паттерны разработки игр в Unreal Engine 5. — Бирмингем : Packt Publishing, 2024. — ISBN 9781803243252. — URL: [www.packtpub.com/en-fr/product/game-development-patterns-with-unreal-engine-5-9781803243252](http://www.packtpub.com/en-fr/product/game-development-patterns-with-unreal-engine-5-9781803243252) ; Game Development Patterns with Unreal Engine 5.
5. Браун К. и. д. Обзор методов Monte Carlo Tree Search // IEEE Transactions on Computational Intelligence and AI in Games. — 2012. — Т. 4, № 1. — С. 1—43. — DOI: 10.1109/TCIAIG.2012.2186810. — Основной survey, первые 5 лет MCTS.
6. Генетический алгоритм: обзор, реализации и применения / Т. Алам [и др.] // ArXiv. — 2020. — arXiv:2007.12673.
7. Доран Д. П. Разработка игр в Unreal Engine 5 с использованием сценариев на C++. — Бирмингем : Packt Publishing, 2022. — ISBN 9781801071873. — Оригинал: John P. Doran. Unreal Engine 5 Game Development with C++ Scripting.
8. Игнатьев А. В. Теория графов. Лабораторные работы: учебное пособие для вузов. — Волгоградский ГТУ : Лань, 2022. — 61 с.

9. Исмаилова Н., Садикова А. Использование генетических алгоритмов в дизайне игр // PAHTEI–Proceedings of Azerbaijan High Technical Educational Institutions. — 2023. — Т. 28, № 05. — С. 06–15. — DOI: 10 . 36962 / PAHTEI28052023-06.
10. Кук М., Колтон С. и Гоу Д. Процедурная генерация ветвящихся квестов для игр // Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015). — Pacific Grove, CA, USA : Society for the Advancement of the Science of Digital Games, 2015. — С. 1–8. — URL: [www.fdg2015.org/papers/fdg2015\\_paper\\_26.pdf](http://www.fdg2015.org/papers/fdg2015_paper_26.pdf).
11. Луо Б., Вилсон Р. С., Хэнкок Э. Р. Спектральная кластеризация графов // Graph Based Representations in Pattern Recognition (GbRPR 2003). Т. 2726 / под ред. Э. Р. Хэнкок, М. Венто. — Springer, Berlin, Heidelberg, 2003. — С. 190–201. — (Lecture Notes in Computer Science). — DOI: 10 . 1007/3-540-45028-9\_17.
12. Саттон Р., Барто Э. Обучение с подкреплением: Введение. — Кембридж : Издательство МИТ Пресс, 2018. — 2-е изд. Пер. с англ. возможно на русский (неофициальный), оригинал: Sutton R.S., Barto A.G. Reinforcement Learning: An Introduction.
13. Свеховский М. и. д. Monte Carlo Tree Search: Обзор недавних модификаций и применений // Artificial Intelligence Review. — 2022. — Обновление survey, MCTS + ML, AutoMCTS.
14. Фон Люксбург У. Обзор спектральной кластеризации // Statistics and Computing. — 2007. — Т. 17, № 4. — С. 395–416. — DOI: 10 . 1007/s11222-007-9033-z. — URL: [arxiv.org/abs/0711.0189](https://arxiv.org/abs/0711.0189).

15. Bowyer-Watson Algorithm for Delaunay Triangulation. — 2022. — URL: [www.gorillasun.de/blog/bowyer-watson-algorithm-for-delaunay-triangulation](http://www.gorillasun.de/blog/bowyer-watson-algorithm-for-delaunay-triangulation) (дата обр. 6.12.2022).
16. Eigen a C++ template library. — 2024. — URL: [eigen.tuxfamily.org/index.php?title>Main\\_Page](http://eigen.tuxfamily.org/index.php?title>Main_Page).
17. Fast Poisson Disk Sampling in Arbitrary Dimensions : University of British Columbia. — URL: [www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf](http://www.cs.ubc.ca/~rbridson/docs/bridson-siggraph07-poissondisk.pdf).
18. nlohmann/JSON. — 2025. — URL: [github.com/nlohmann/json](https://github.com/nlohmann/json).
19. Unreal Engine 5.4 Documentation. — 2024. — URL: [dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-4-documentation?application\\_version=5.4](https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-5-4-documentation?application_version=5.4).
20. Visual Studio documentation. — 2024. — URL: [learn.microsoft.com/en-us/visualstudio/ide/?view=vs-2022](https://learn.microsoft.com/en-us/visualstudio/ide/?view=vs-2022).

# ПРИЛОЖЕНИЕ 1.

## Реализация структуры узла в алгоритме Fast Poisson Disk Sampling распределения точек на плоскости

Листинг 21. Knot.h

```
#pragma once

#define _USE_MATH_DEFINES

#include <cmath>
#include <vector>
#include <random>
#include <chrono>
#include <algorithm>

struct Knot {
    float x, y;
    Knot() : x(-1.0f), y(-1.0f) {}
    Knot(float _x, float _y) : x(_x), y(_y) {}

    bool operator==(const Knot& another) const;
    bool operator<(const Knot& another) const;
};

using Edge = std::pair<Knot, Knot>;
bool operator<(const Edge& a, const Edge& b);
bool operator==(const Edge& a, const Edge& b);

namespace std {

    template<> struct hash<Knot> {
        size_t operator()(const Knot& k) const {
            return hash<float>()(k.x) ^ (hash<float>()(k.y) << 1);
        }
    }
}
```

```

};

void insertKnot(std::vector<std::vector<Knot>>& grid, const Knot& knot, float
cellsize);

bool isKnotValid(const std::vector<std::vector<Knot>>& grid, int g_width, int
g_height,
const Knot& knot, float radius, float width, float height, float cellsize);

float distanceBetweenPoints(float x1, float y1, float x2, float y2);

std::vector<Knot> poissonDiskSampling(float width, float height, double radius,
int limit);

```

### Листинг 22. Knot.cpp

```

#include "Knot.h"
#include <iostream>
#include <random>
#include <chrono>

bool Knot::operator==(const Knot& another) const {
    return x == another.x && y == another.y;
}

bool Knot::operator<(const Knot& another) const {
    if (x < another.x) return true;
    if (x > another.x) return false;
    return y < another.y;
}

void insertKnot(std::vector<std::vector<Knot>>& grid, const Knot& knot, float
cellsize) {

```

```

int xindex = static_cast<int>(std::floor(knot.x / cellsize));
int yindex = static_cast<int>(std::floor(knot.y / cellsize));
if (xindex >= 0 && xindex < grid.size() &&
    yindex >= 0 && yindex < grid[0].size()) {
    grid[xindex][yindex] = knot;
}

}

bool isKnotValid(const std::vector<std::vector<Knot>>& grid, int g_width, int
g_height,
const Knot& knot, float radius, float width, float height, float cellsize) {
if (knot.x < 0 || knot.x >= width || knot.y < 0 || knot.y >= height) {
return false; }

int xindex = static_cast<int>(std::floor(knot.x / cellsize));
int yindex = static_cast<int>(std::floor(knot.y / cellsize));
int i0 = std::max(xindex - 2, 0);
int i1 = std::min(xindex + 2, g_width - 1);
int j0 = std::max(yindex - 2, 0);
int j1 = std::min(yindex + 2, g_height - 1);

for (int i = i0; i <= i1; ++i) {
    for (int j = j0; j <= j1; ++j) {
        const Knot& other = grid[i][j];
        if (other.x != -1 && distanceBetweenPoints(other.x, other.y, knot.x,
knot.y) < radius) {
            return false;
        }
    }
}
return true;
}

```

```

float distanceBetweenPoints(float x1, float y1, float x2, float y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    return std::sqrt(dx * dx + dy * dy);
}

std::vector<Knot> poissonDiskSampling(float width, float height, double radius,
int limit) {
    const int N = 2;
    std::vector<Knot> knots;
    std::vector<Knot> active;

    float cellsize = static_cast<float>(radius / std::sqrt(N));
    cellsize = std::max(cellsize, 1.0f);

    int ncells_width = static_cast<int>(std::ceil(width / cellsize)) + 1;
    int ncells_height = static_cast<int>(std::ceil(height / cellsize)) + 1;

    std::vector<std::vector<Knot>> grid(ncells_width,
    std::vector<Knot>(ncells_height));

    static std::mt19937 gen(std::random_device{}());
    std::uniform_real_distribution<float> dist_x(0.0f, width);
    std::uniform_real_distribution<float> dist_y(0.0f, height);

    Knot k0(dist_x(gen), dist_y(gen));
    insertKnot(grid, k0, cellsize);
    knots.push_back(k0);
    active.push_back(k0);

    while (!active.empty()) {
        std::uniform_int_distribution<size_t> RKI_gen(0, active.size() - 1);
        size_t random_knot_index = RKI_gen(gen);

```

```

Knot active_knot = active[random_knot_index];

bool found = false;

for (int tries = 0; tries < limit; ++tries) {
    std::uniform_real_distribution<float> angle_dist(0.0f, 2.0f *
static_cast<float>(M_PI));
    float theta = angle_dist(gen);

    std::uniform_real_distribution<float> radius_dist(radius, 2.0f *
radius);
    float random_radius = radius_dist(gen);

    Knot knew(
        active_knot.x + random_radius * std::cos(theta),
        active_knot.y + random_radius * std::sin(theta)
    );

    if (isKnotValid(grid, ncells_width, ncells_height, knew, radius,
width, height, cellsize)) {
        knots.push_back(knew);
        insertKnot(grid, knew, cellsize);
        active.push_back(knew);
        found = true;
        break;
    }
}

if (!found) { active.erase(active.begin() + random_knot_index); }

return knots;
}

```

## ПРИЛОЖЕНИЕ 2.

### Реализация структуры треугольника и алгоритма Боуера-Ватрсона построения триангуляции Делоне

Листинг 23. Triangulation.h

```
#pragma once

#include <vector>
#include <iostream>
#include <set>
#include "Knot.h"

struct Triangle {
    Knot knot_1, knot_2, knot_3;

    bool operator == (const Triangle& another) const;
};

struct EdgeHash {
    size_t operator()(const Edge& e) const {
        auto hash1 = std::hash<float>()(e.first.x) ^
        std::hash<float>()(e.first.y);
        auto hash2 = std::hash<float>()(e.second.x) ^
        std::hash<float>()(e.second.y);
        return hash1 ^ (hash2 << 1);
    }
};

double determinant(const Knot& a, const Knot& b, const Knot& c);
bool isKnotInCircumcircle(const Triangle& triangle, const Knot& knot);
std::vector<Edge> getEdgesFromTriangles(const std::vector<Triangle>& triangles);
std::vector<Triangle> delaunayTriangulation(const std::vector<Knot>& knots);
std::vector<Edge> getUniqueEdges(const std::vector<Triangle>& triangles);
```

## Листинг 24. Triangulation.cpp

```
#include "Triangulation.h"
#include <algorithm>
#include <map>

bool Triangle::operator == (const Triangle& another) const {
    std::set<Knot> this_knots = { knot_1, knot_2, knot_3 };
    std::set<Knot> another_knots = { another.knot_1, another.knot_2,
another.knot_3 };
    return this_knots == another_knots;
}

bool operator<(const Edge& a, const Edge& b) {
    if (a.first == b.first) {
        return a.second < b.second;
    }
    return a.first < b.first;
}

bool operator==(const Edge& a, const Edge& b) {
    return (a.first == b.first && a.second == b.second) ||
(a.first == b.second && a.second == b.first);
}

void printTriangle(Triangle& triangle) {
    std::cout << "\n(" << triangle.knot_1.x << ";" << triangle.knot_1.y << "
<<
    "(" << triangle.knot_2.x << ";" << triangle.knot_2.y << ") " <<
    "(" << triangle.knot_3.x << ";" << triangle.knot_3.y << ")";
}

double determinant(const Knot& a, const Knot& b, const Knot& c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

bool isKnotInCircumcircle(const Triangle& triangle, const Knot& knot) {
    Knot a = triangle.knot_1;
    Knot b = triangle.knot_2;
```

```

Knot c = triangle.knot_3;

float d = (a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y));
if (d == 0) return false;

float ux = ((a.x * a.x + a.y * a.y) * (b.y - c.y) +
            ((b.x * b.x + b.y * b.y) * (c.y - a.y)) +
            ((c.x * c.x + c.y * c.y) * (a.y - b.y)));

float uy = ((a.x * a.x + a.y * a.y) * (c.x - b.x) +
            ((b.x * b.x + b.y * b.y) * (a.x - c.x)) +
            ((c.x * c.x + c.y * c.y) * (b.x - a.x)));

Knot center(ux / (2 * d), uy / (2 * d));

float radius_sq = (center.x - a.x) * (center.x - a.x) +
                  (center.y - a.y) * (center.y - a.y);

float dist_sq = (knot.x - center.x) * (knot.x - center.x) +
                (knot.y - center.y) * (knot.y - center.y);

return dist_sq <= radius_sq;
}

std::vector<Edge> getEdgesFromTriangles(const std::vector<Triangle>& triangles) {
    std::map<std::pair<Knot, Knot>, bool> edge_map;
    for (const auto& triangle : triangles) {
        Edge edges[3] = {
            {std::min(triangle.knot_1, triangle.knot_2),
             std::max(triangle.knot_1, triangle.knot_2)},
            {std::min(triangle.knot_1, triangle.knot_3),
             std::max(triangle.knot_1, triangle.knot_3)},
            {std::min(triangle.knot_2, triangle.knot_3),
             std::max(triangle.knot_2, triangle.knot_3)}};

        for (const auto& edge : edges) {
            auto key = std::make_pair(edge.first, edge.second);
            edge_map[key] = !edge_map[key];
        }
    }
}

```

```

    std::vector<Edge> edges;
    for (const auto& pair : edge_map) {
        if (pair.second) {
            edges.emplace_back(pair.first.first, pair.first.second);
        }
    }
    return edges;
}

std::vector<Triangle> delaunayTriangulation(const std::vector<Knot>& knots) {
    std::vector<Triangle> triangles;
    float minX = std::numeric_limits<float>::max();
    float minY = std::numeric_limits<float>::max();
    float maxX = std::numeric_limits<float>::lowest();
    float maxY = std::numeric_limits<float>::lowest();
    for (const auto& knot : knots) {
        if (knot.x < minX) minX = knot.x;
        if (knot.y < minY) minY = knot.y;
        if (knot.x > maxX) maxX = knot.x;
        if (knot.y > maxY) maxY = knot.y;
    }
    float dx = maxX - minX;
    float dy = maxY - minY;
    float deltaMax = std::max(dx, dy);
    float midX = (minX + maxX) / 2.0f;
    float midY = (minY + maxY) / 2.0f;
    Triangle superTriangle = {
        {midX - 20.0f * deltaMax, midY - 10.0f * deltaMax},
        {midX, midY + 20.0f * deltaMax},
        {midX + 20.0f * deltaMax, midY - 10.0f * deltaMax}
    };
    triangles.push_back(superTriangle);
    for (const auto& knot : knots) {
        std::vector<Triangle> badTriangles;

```

```

    for (const auto& triangle : triangles) {
        if (isKnotInCircumcircle(triangle, knot)) {
            badTriangles.push_back(triangle);
        }
    }

    std::vector<Edge> polygon;
    for (const auto& triangle : badTriangles) {
        for (const auto& edge : getEdgesFromTriangles({ triangle })) {
            bool shared = false;
            for (const auto& otherTriangle : badTriangles) {
                if (triangle == otherTriangle) continue;
                for (const auto& otherEdge : getEdgesFromTriangles({
                    otherTriangle })) {
                    if (edge == otherEdge) {
                        shared = true;
                        break;
                    }
                }
                if (shared) break;
            }
            if (!shared) {
                polygon.push_back(edge);
            }
        }
    }
    triangles.erase(std::remove_if(triangles.begin(), triangles.end(),
        [&badTriangles](const Triangle& t) {
            return std::find(badTriangles.begin(), badTriangles.end(), t) !=
        badTriangles.end();
        }), triangles.end());
    for (const auto& edge : polygon) {
        triangles.push_back({ edge.first, edge.second, knot });
    }
}

```

```

    }

    triangles.erase(std::remove_if(triangles.begin(), triangles.end(),
        [&superTriangle](const Triangle& t) {
            return t.knot_1 == superTriangle.knot_1 || t.knot_1 ==
superTriangle.knot_2 || t.knot_1 == superTriangle.knot_3 ||
            t.knot_2 == superTriangle.knot_1 || t.knot_2 ==
superTriangle.knot_2 || t.knot_2 == superTriangle.knot_3 ||
            t.knot_3 == superTriangle.knot_1 || t.knot_3 ==
superTriangle.knot_2 || t.knot_3 == superTriangle.knot_3;
        }), triangles.end()));

    return triangles;
}

std::vector<Edge> getUniqueEdges(const std::vector<Triangle>& triangles) {
    std::set<Edge> unique_edges;
    for (const auto& triangle : triangles) {
        Edge edges[3] = {
            {std::min(triangle.knot_1, triangle.knot_2),
             std::max(triangle.knot_1, triangle.knot_2)},
            {std::min(triangle.knot_1, triangle.knot_3),
             std::max(triangle.knot_1, triangle.knot_3)},
            {std::min(triangle.knot_2, triangle.knot_3),
             std::max(triangle.knot_2, triangle.knot_3)}
        };

        for (const auto& edge : edges) {
            unique_edges.insert(edge);
        }
    }

    return std::vector<Edge>(unique_edges.begin(), unique_edges.end());
}

```

## ПРИЛОЖЕНИЕ 3.

### Объявление структур генетического алгоритма распределения сценарных триггеров по графу

Листинг 25. QuestPlacement.h

```
#pragma once

#include "Knot.h"
#include <vector>
#include <string>
#include <map>
#include <set>
#include <random>
#include <unordered_map>

struct Chromosome {

    std::vector<int> node_ids; // node_ids[i] - ID узла для тега i
};

class QuestPlacement {

private:

    const std::vector<Knot>& knots;

    const std::unordered_map<Knot, std::vector<Knot>, std::hash<Knot>>&
    adjacency_list;

    std::vector<std::string> tag_list; // Список тегов, таких как ["spawn",
    "asylum", "exit"]

    std::map<std::string, int> desired_distances; // Желаемые дистанции от
    spawn_knot

    std::vector<std::tuple<std::string, std::string, int>>
    min_distance_constraints; // Минимальные расстояния между тегами

    int spawn_node_id; // Фиксированный ID узла для тега spawn
    int asylum_node_id; // Фиксированный ID узла для тега asylum
    int exit_node_id; // Фиксированный ID узла для тега exit
```

```

int population_size;
int generations;
float mutation_rate;

std::vector<Chromosome> initialize_population(int num_nodes, int num_tags,
std::mt19937& rng);
float compute_fitness(const Chromosome& chrom, const std::vector<int>&
distances, const std::vector<std::vector<int>>& dist_matrix);
Chromosome crossover(const Chromosome& parent1, const Chromosome& parent2,
int num_nodes, std::mt19937& rng);
void mutate(Chromosome& chrom, int num_nodes, std::mt19937& rng);
Chromosome tournament_selection(const std::vector<Chromosome>& population,
const std::vector<float>& fitnesses, int tournament_size, std::mt19937& rng);

public:
QuestPlacement(
    const std::vector<Knot>& knots,
    const std::unordered_map<Knot, std::vector<Knot>, std::hash<Knot>>&
adj_list,
    const std::vector<std::string>& tags,
    const std::map<std::string, int>& desired_dists,
    const std::vector<std::tuple<std::string, std::string, int>>&
min_dist_constraints,
    int spawn_node_id,
    int asylum_node_id,
    int exit_node_id,
    int pop_size = 100,
    int gens = 50,
    float mut_rate = 0.01f
);
std::map<std::string, int> run_genetic_algorithm(std::mt19937& rng);
};

```

## ПРИЛОЖЕНИЕ 4.

### Объявление и реализация основного метода алгоритма поиска по дереву Монте-Карло

Листинг 26. MCTS.h

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "MCTSNode.h"
#include "MCTS.generated.h"

class AUnit;
class ACell;
class ABoard;

UCLASS()
class IVORY_MEMORYS_API UMCTS : public UObject
{
GENERATED_BODY()

public:
    UMCTS();
    FUnitCellPair Run(ABoard* Board, int32 Iterations); // Выполнение поиска
MCTS и возврат лучшего действия

private:
    UPROPERTY()
    UMCTSNode* Root;
};


```

Листинг 27. MCTS.cpp

```
#include "MCTS.h"
```

```

#include "MCTSNode.h"
#include "CombatState.h"
#include "Cell.h"

UMCTS::UMCTS()
{
    Root = nullptr;
}

FUnitCellPair UMCTS::Run(ABoard* Board, int32 Iterations) {
    if (!Board) {
        UE_LOG(LogTemp, Error, TEXT("MCTS::Run: Board is null"));
        return FUnitCellPair(nullptr, nullptr);
    }

    Root = NewObject<UMCTSNode>();
    UCombatState* InitialState = NewObject<UCombatState>();
    InitialState->Initialize(Board, true);
    Root->Initialize(InitialState, nullptr, FUnitCellPair(nullptr, nullptr));
    if (Root->UntriedActions.Num() == 0) {
        UE_LOG(LogTemp, Warning, TEXT("MCTS::Run: No actions available"));
        return FUnitCellPair(nullptr, nullptr);
    }

    for (int32 i = 0; i < Iterations; i++) {
        UMCTSNode* Node = Root;
        while (Node->UntriedActions.Num() == 0 && Node->Children.Num() > 0) {
            Node = Node->SelectChild();
            if (!Node) break;
        }

        if (Node && Node->UntriedActions.Num() > 0) {
            Node->Expand();
            Node = Node->Children.Last();
        }
        float Reward = Node->Simulate();
    }
}

```

```

    Node->Backpropagate(Reward);

}

UMCTSNode* BestChild = nullptr;
int32 MaxVisits = -1;
for (UMCTSNode* Child : Root->Children) {
    if (Child->Visits > MaxVisits) {

        MaxVisits = Child->Visits;
        BestChild = Child;
    }
}

if (!BestChild) {
    UE_LOG(LogTemp, Warning, TEXT("MCTS::Run: No best child found"));
    return FUnitCellPair(nullptr, nullptr);
}

return BestChild->Action;
}

```

## ПРИЛОЖЕНИЕ 5.

### Класс, описывающий боевую фигуру (юнита) на поле сражения

Листинг 28. Unit.h

```
#pragma once

#include "Board.h"
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Unit.generated.h"

class ACell;

UCLASS()
class IVORY_MEMORYS_API AUnit : public AActor
{
GENERATED_BODY()

public:
    AUnit();

    UPROPERTY(VisibleAnywhere)
    UStaticMeshComponent* UnitMesh;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Movement")
    int32 MovementRange = 1;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Combat")
    int32 Rank = 1;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Combat")
```

```

bool bIsEnemy = false;

UPROPERTY(BlueprintReadOnly)
bool bIsFrenzied = false;

UPROPERTY(BlueprintReadOnly, Category = "Turn")
bool bHasActed = false;

UFUNCTION(BlueprintCallable)
void ResetAction();

void JumpToCell(ACell* TargetCell, TFunction<void()> OnComplete = nullptr);

FVector JumpStart;
FVector JumpEnd;
float JumpElapsed = 0.f;
float JumpDuration = 0.3f;
bool bIsJumping = false;
TFunction<void()> OnJumpFinished;
virtual void Tick(float DeltaTime) override;
void SetMesh(UStaticMesh* Mesh);
void SetCellReference(ACell* Cell);
ACell* GetCell() const;
int32 GetIncomingHitsFrom(UClass* AttackerType) const;

bool CanAttack(const AUnit* Target, const TArray<AUnit*>& Allies) const;
bool ReceiveHitFrom(UClass* AttackerType, int32 RequiredHits);
void UpdateFrenzyState(const TArray<AUnit*>& EnemyUnits, const ABoard*
Board);
TArray<ACell*> GetMoveRange(const ABoard* Board) const;
TArray<ACell*> GetAttackRange(const ABoard* Board) const;
bool Attack(AUnit* Target);

```

```

private:
    ACell* CurrentCell = nullptr;
    TMap<UClass*, int32> IncomingHits;
};

```

Листинг 29. Unit.cpp

```

#include "Unit.h"
#include "Cell.h"
#include "Board.h"
#include "Components/StaticMeshComponent.h"

AUnit::AUnit()
{
    PrimaryActorTick.bCanEverTick = true;
    UnitMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("UnitMesh"));
    RootComponent = UnitMesh;
    bHasActed = false; // Инициализация

}

void AUnit::SetMesh(UStaticMesh* Mesh)
{
    if (Mesh)
    {
        UnitMesh->SetStaticMesh(Mesh);
    }
}

void AUnit::SetCellReference(ACell* Cell)
{
    if (CurrentCell) { CurrentCell->SetOccupyingUnit(nullptr); }
    CurrentCell = Cell;
    if (CurrentCell) { CurrentCell->SetOccupyingUnit(this); }
}

```

```

ACell* AUnit::GetCell() const { return CurrentCell; }

bool AUnit::CanAttack(const AUnit* Target, const TArray<AUnit*>& Allies) const
{
    if (!Target) return false;
    if (bIsFrenzied)
    {
        UE_LOG(LogTemp, Warning, TEXT("CanAttack: Frenzied, returning true"));
        return true;
    }
    int32 SameTypeCount = 0;
    for (const AUnit* Ally : Allies)
    {
        if (Ally && Ally->GetClass() == GetClass())
        {
            SameTypeCount++;
        }
    }
    int32 MaxRankToAttack = Rank + 1 + SameTypeCount;
    bool Result = Target->Rank >= 1 && Target->Rank <= MaxRankToAttack;
    UE_LOG(LogTemp, Warning, TEXT("CanAttack: Rank %d, Target Rank %d,
SameTypeCount %d, MaxRankToAttack %d, Result %s"),
        Rank, Target->Rank, SameTypeCount, MaxRankToAttack, Result ?
TEXT("true") : TEXT("false"));
    return Result;
}

bool AUnit::ReceiveHitFrom(UClass* AttackerType, int32 RequiredHits)
{
    if (!AttackerType || RequiredHits <= 0) return false;

    int32& HitCount = IncomingHits.FindOrAdd(AttackerType);

```

```

    HitCount++;

    if (HitCount >= RequiredHits)
    {
        IncomingHits.Remove(AttackerType);
        return true;
    }

    return false;
}

void AUnit::UpdateFrenzyState(const TArray<AUnit*>& EnemyUnits, const ABoard*
    Board)
{
    if (!Board) return;

    TArray<ACell*> MoveCells = GetMoveRange(Board);
    MoveCells.RemoveAll([](ACell* Cell) { return !Cell || Cell->IsOccupied(); });

    for (ACell* Cell : MoveCells)
    {
        for (AUnit* Enemy : EnemyUnits)
        {
            if (!Enemy || !Enemy->GetCell()) continue;

            TArray<ACell*> ThreatZone = Enemy->GetAttackRange(Board);

            if (!ThreatZone.Contains(Cell))
            {
                bIsFrenzied = false;
                return;
            }
        }
    }
}

```

```

    }

    bIsFrenzied = true;
}

int32 AUnit::GetIncomingHitsFrom(UClass* AttackerType) const
{
    if (!AttackerType) return 0;

    const int32* Hits = IncomingHits.Find(AttackerType);
    return Hits ? *Hits : 0;
}

void AUnit::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (bIsJumping)
    {
        JumpElapsed += DeltaTime;
        float Alpha = FMath::Clamp(JumpElapsed / JumpDuration, 0.f, 1.f);
        FVector NewLoc = FMath::Lerp(JumpStart, JumpEnd, Alpha);
        float ZOffset = 100.f * FMath::Sin(Alpha * PI);
        NewLoc.Z += ZOffset;
        SetActorLocation(NewLoc);

        if (Alpha >= 1.f)
        {
            bIsJumping = false;
            if (OnJumpFinished)
            {
                OnJumpFinished();
            }
        }
    }
}

```

```

        }

    }

}

void AUnit::JumpToCell(ACell* TargetCell, TFunction<void()> OnComplete)
{
    if (!TargetCell) return;

    JumpStart = GetActorLocation();

    FVector TargetLocation = TargetCell->GetActorLocation();
    if (UStaticMeshComponent* Mesh =
        FindComponentByClass<UStaticMeshComponent>())
    {
        const FBoxSphereBounds Bounds = Mesh->CalcBounds(FTransform::Identity);
        const float BottomZ = Bounds.Origin.Z - Bounds.BoxExtent.Z;
        TargetLocation -= FVector(0.f, 0.f, BottomZ);
    }

    JumpEnd = TargetLocation;
    JumpElapsed = 0.f;
    bIsJumping = true;
    OnJumpFinished = [this, TargetCell, OnComplete]()
    {
        SetActorLocation(JumpEnd);
        SetCellReference(TargetCell);
        if (OnComplete)
            OnComplete();
    };
}

TArray<ACell*> AUnit::GetMoveRange(const ABoard* Board) const
{

```

```

if (!Board || !CurrentCell) return TArray<ACell*>();

return Board->GetCellsInRange(CurrentCell->GridX, CurrentCell->GridY,
MovementRange);

}

TArray<ACell*> AUnit::GetAttackRange(const ABoard* Board) const
{
if (!Board || !CurrentCell) return TArray<ACell*>();

TArray<ACell*> Cells = Board->GetCellsInRange(CurrentCell->GridX,
CurrentCell->GridY, MovementRange);
if (bIsEnemy)
{
    UE_LOG(LogTemp, Warning, TEXT("Attack range for %s: %d cells"),
*GetName(), Cells.Num());
    for (ACell* Cell : Cells)
    {
        UE_LOG(LogTemp, Warning, TEXT("Cell at (%d, %d)"), Cell->GridX,
Cell->GridY);
    }
}
return Cells;
}

bool AUnit::Attack(AUnit* Target)
{
if (!Target) return false;

int32 RequiredHits = 1;
if (!bIsFrenzied && Rank < Target->Rank)
{
    RequiredHits = Target->Rank - Rank;
}

```

```
    bool bKilled = Target->ReceiveHitFrom(GetClass(), RequiredHits);
    return bKilled;
}

void AUnit::ResetAction()
{
    bHasActed = false;
}
```