

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

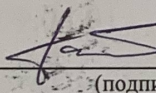
Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

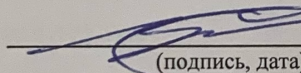
**ИГРОВОЕ ПРИЛОЖЕНИЕ С ИСПОЛЬЗОВАНИЕМ
ИСКУССТВЕННОГО ИНТЕЛЛЕКТА**

Студент

 30.05.24
(подпись, дата)

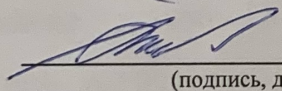
Гаевский Илья Романович
(ФИО полностью)

Руководитель

 30.05.24
(подпись, дата)

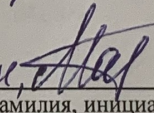
Зинкин С.А.
(фамилия, инициалы)

Нормоконтролёр

 31.05.2024
(подпись, дата)

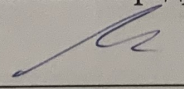
Бычков А.С.
(фамилия, инициалы)

Рецензент

зав. каф. Программирования ПензГУ, к.т.н.  Мартыненко А.И.
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 27.05.24 № 11)

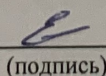
Заведующий кафедрой


(подпись)

М.А. Митрохин
(фамилия, инициалы)

Работа защищена с отметкой отлично (протокол заседания ГЭК
от 14.06.2024 № 18)

Секретарь ГЭК


(подпись)

Е.И. Гурин
(фамилия, инициалы)

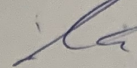
Пенза 2024

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет
вычислительной техники

Кафедра
«Вычислительная техника»

«Утверждаю»
Заведующий кафедрой



М. А. Митрохин
«13» сентября 2022г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

1. Студент Гаевский И.Р. гр. 22ВВИм1 факультета вычислительной техники, направления 09.04.01.

2. Тема работы «Игровое приложение с использованием искусственного интеллекта».

Тема утверждена приказом ПГУ № 765/о от "13" сентября 2022 г.

3. Руководитель магистерской диссертации к.т.н., доцент кафедры ВТ ПГУ Зинкин С.А.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

Назначение разработки: игровое приложение с использованием искусственного интеллекта для создания конкуренции игроку во время игры.

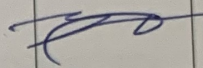
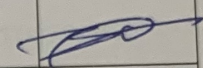
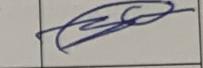
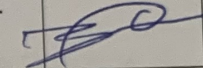
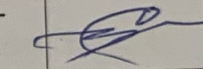
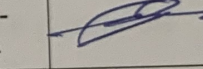
Разработать игровое приложение, включающее в себя искусственный интеллект. В качестве сферы применения использовать игровое приложение в жанре «Стратегия реального времени».

5. Перечень подлежащих разработке вопросов:

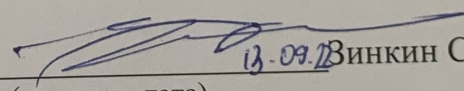
1) Анализ предметной области использования искусственного интеллекта в игровых приложениях,

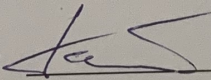
- 2) Проектирование структуры разрабатываемого приложения,
- 3) Разработка всех компонентов игрового приложения,
- 4) Обучение модели поиска для интеграции в игровое приложение,
- 5) Отладка и тестирование готового приложения,
- 6) Оформление выпускной работы.

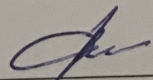
6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководи теля
1.	Анализ предметной области пользования искусственного интеллекта игровых приложениях	10%	13.09.2022- 13.11.2022	
2.	Проектирование структуры разрабатываемого приложения	20%	14.11.2022- 14.02.2022	
3.	Разработка всех компонентов игрового приложения	40%	15.02.2023- 15.10.2023	
4.	Обучение модели поиска для интеграции в игровое приложение	15%	16.10.2023- 05.01.2024	
5.	Отладка и тестирование готового приложения	5%	06.01.2024- 30.02.2024	
6.	Оформление выпускной работы	10%	31.02.2024- 31.05.2024	

Дата выдачи задания "13" сентября 2022 г.

Руководитель магистерской диссертации  13.09.22 Винкин С.А.
(подпись, дата)

Задание к исполнению принял студент  13.09.22 Гаевский И.Р.
(подпись, дата)

Работу к защите допустить
Декан факультета  11.06.2024 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

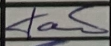
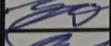
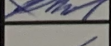
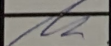
Пояснительная записка содержит 164 листа, 19 рисунков, 1 таблицу, 12 источников, 7 приложений

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, ИГРОВОЙ ИСКУССТВЕННЫЙ
ИНТЕЛЛЕКТ, МАШИННОЕ ОБУЧЕНИЕ, ОБУЧЕНИЕ С
ПОДКРЕПЛЕНИЕМ, ИИ, AI, ML, ARTIFICIAL INTELLIGENCE, UNITY

Цель работы разработать игровое приложение в жанре «Стратегия в реальном времени», включающее модули искусственного интеллекта для планирования действий в реальном времени, передвижения неигровых персонажей и поиска объектов.

Объектом разработки является игровое приложение с использованием искусственного интеллекта. Игровое приложение в жанре «Стратегия в реальном времени» содержит искусственный интеллект, который составляет конкуренцию для игрока.

В результате проделанной работы разработано игровое приложение в жанре «Стратегия в реальном времени», включающее в себя модуль искусственного интеллекта планирующий задачи для выполнения в реальном времени в зависимости от окружающего мира, модуль для поиска путей для агентов и модель для поиска объектов на основе алгоритмов глубокого обучения с подкреплением.

					ПГУ 1.09.04.01.02.001 ПЗ		
Изм.	Лист	№докум.	Подпись	Дата			
Разраб.		Гаевский И.Р.		31.05.24	Игровое приложение с использованием искусственного интеллекта Пояснительная записка	Лит.	Лист
Провер.		Зинкин С.А.		31.05.24			2
Н.Контр		Бычков А.С.		31.05.24		ФВТ,каф.ВТ гр.22ВВИм1	
Утверд.		Митрохин М.А.		31.05.24			
						Листов	164

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	7
ВВЕДЕНИЕ	8
1 Актуальность	10
2 Обзор существующих решений	12
2.1 Extreme AI	12
3 Выбор инструментов разработки	15
4 Архитектура приложения	19
4.1 Игровое приложение	19
4.2 Принятие решений	20
4.3 Поиск пути	28
4.4 Модель поиска объектов	34
4.4.1 Proximal Policy Optimization (PPO)	34
4.4.2 Soft Actor-Critic (SAC)	37
5 Реализация игровой логики	39
5.1 Настройка сцены и игрового мира	39
5.2 Менеджер местности	39
5.3 Игровой менеджер	42
5.4 Создание и настройка юнитов	43
5.5 Создание игровых ресурсов	44
5.6 Настройка атаки	44
6 Реализация модуля планирования действий	48
6.1 Класс Action	49
6.2 Планировщик	50
6.3 Класс агента	54
7 Обучение модели поиска ресурсов	60

7.1 Настройка гиперпараметров.....	61
7.2 Структура агента.....	63
7.3 Реализация агента.....	64
7.4 Обучение с помощью PPO.....	69
7.5 Обучение с SAC.....	70
7.6 Сравнение результатов.....	70
8 Реализация алгоритма AStar.....	75
8.1 Модификаторы.....	78
8.1.1 Smooth модификатор	78
8.1.2 Модификатор Funnel.....	78
8.1.3 Модификатор Raycast.....	85
8.2 Генерация графиков	90
8.2.1 Recast график.....	90
8.2.2 График Navmesh.....	96
8.2.3 Точечный график	98
8.3 Генератор графов	99
ЗАКЛЮЧЕНИЕ	104
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ.....	105
ПРИЛОЖЕНИЕ А Листинг программы	106
ПРИЛОЖЕНИЕ А.1 Файл «Planner.cs».....	107
ПРИЛОЖЕНИЕ А.2 Файл «AgentPlanner.cs».....	115
ПРИЛОЖЕНИЕ А.3 Файл «AIBase.cs».....	120
ПРИЛОЖЕНИЕ А.4 Файл «AstarData.cs».....	131
ПРИЛОЖЕНИЕ А.5 Файл «Path.cs».....	142
ПРИЛОЖЕНИЕ А.6 Файл «PathUtilities.cs».....	153
ПРИЛОЖЕНИЕ Б Презентация.....	160

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей работе применяют следующие сокращения и обозначения

ИИ – искусственный интеллект

ПК – персональный компьютер

AI (Artificial intelligence) – Искусственный интеллект

GOAP (Goal-Oriented Action Planning) – Целенаправленное планирование действий

PPO – Proximal Policy Optimization

SAC – Soft Actor-Critic

ML (Machine learning) – Машинное обучение

FSM (finite-state machine) – Конечный автомат состояний задающий поведение для персонажа в зависимости от произошедших событий

ВВЕДЕНИЕ

История создание игровых приложений с использованием искусственного интеллекта началась еще в 1949 году, когда криптограф Клод Шеннон размышлял над игрой в шахматы для одного игрока на компьютере, а Кристофер Стрэчи написал программу, которая играла в шашки. Артуром Самуэлем в середине 50-х и начале 60-х годов на основе программы Клода Шеннона для шахмат разработал симлюлятор шашек, который в конечном счёте достиг достаточного мастерства, чтобы бросить вызов чемпиону мира. Работа над компьютерными шашками и шахматами достигла кульминации в 1997 году, когда компьютер DeepBlue выиграл матч по шахматам у чемпиона мира Гарри Каспарова с третьего раза. С тех пор программное обеспечение игрового интеллекта, которое управляло ракеткой Pong или призраком Pac-Man, выросло до алгоритмов построения вселенной космической элиты.

В современном мире индустрия видеоигр развивается очень быстро. Некоторые игры могут приносить больше денег, чем фильмы и другие виды электронных развлечений. Вместе со спросом на игры растет и планка качества, ведь современным играм недостаточно иметь красивую картинку и простой сюжет, чтобы понравиться игроку. Современных игроков уже сложно удивить, поэтому разработчики игр делают упор на захватывающий и правдоподобный игровой процесс. Игровой ИИ играет в этом важную роль.

Искусственный интеллект в играх служит различным целям, таким как моделирование правдоподобное поведение NPC, привнесение веселья в игру, создание некоторых трудностей для игрока, случайная генерация уровней и многое другое. Искусственный интеллект не должен побеждать игрока, а скорее делать игровой процесс приятным и забавным, из-за этого многие традиционные методы проектирования искусственного интеллекта могут оказаться бесполезными в контексте видеоигр, поскольку они ориентированы на результат, такой как победа, а не на процесс.

Искусственный интеллект в видеоиграх не обязательно должен быть умным и легко переигрывать игроков, он должен казаться умным и

приносить удовольствие от игры. Разработчики видеоигр перепробовали множество приемов и алгоритмов создания искусственного интеллекта, но до сих пор нет ответа на вопрос, какие приемы наиболее подходят для разных жанров и возможно ли создать универсальную технику.

1 Актуальность

В последние два десятилетия на рынке появляется всё больше бесплатных игровых движков, даже те, что были ранее доступны только ограниченному кругу лиц выходят на массовый рынок. В следствии такой тенденции у разработчиков растёт интерес к сфере создания игр.

Разработку игрового приложения можно условно поделить на несколько частей: сюжет, базовые механики, искусственный интеллект для неигровых персонажей, дизайн, мультиплеер, если он предусмотрен, UI для игры. В больших компаниях за каждую часть отвечает своя группа людей, но у индивидуального разработчика в большинстве случаев нет возможности нанять команду для своего проекта, поэтому ему приходится решать все поставленные задачи самостоятельно. С такой проблемой столкнулись многие люди и вслед за этим на рынке стали появляться плагины, расширения, для игровых движков. Они позволяют упростить и ускорить процесс разработки конкретной задачи. Например, существуют готовые решения для игровой логики игр в жанре шутер или решения для создания ИИ в играх. Когда разработчик получает такие расширения для своего проекта, ему остаётся их настроить под себя и переходить для решения следующих задач.

Одной из самых долгих частей в разработке игрового приложения является создания искусственного интеллекта для неигровых персонажей. Он позволяет добавить динамичности, разнообразности и уникальности в игровой процесс. Некоторые игровые движки предоставляют базовые возможности ИИ, но как правило их мало для создания даже удовлетворительного ИИ. Зачастую их используют как основу для дальнейшего расширения возможностей своими силами.

В рамках диссертации будет разработано готовое решения для индивидуальных разработчиков позволяющее в кратчайшие сроки создавать ИИ для неигровых персонажей любого плана в своём игровом приложении и настраивать его под свои задачи и нужды. Решение будет включать в себя

несколько линий поведений для персонажей под управлением искусственного поведения, которые будут определяться в зависимости от взаимодействий его с игроками. Игроки смогут общаться с ИИ посредством текста и пытаться выстраивать с ним отношения

2 Обзор существующих решений

2.1 Extreme AI

Расширение ExtremeAI предоставляет готовое модульное решение искусственного интеллекта для игровых приложения разрабатываемых при помощи игрового движка Unity. Модульность позволяет разработчикам использовать только необходимые части актива без страха потери производительности своей игры. ExtremeAI реализован на основе FSM с возможностью «обучения» некоторых отдельных модулей. Искусственный интеллект на основе FSM, FiniteStateMachine или конечного автомата состояний, представляет множество заданных заранее правил, по которым ИИ определяет основу поведения в конкретный момент времени. Ограничение такого подхода является, что одновременно может быть активным только одно состояние. На рисунке 1 представлен простой вариант FSM в виде диаграммы.

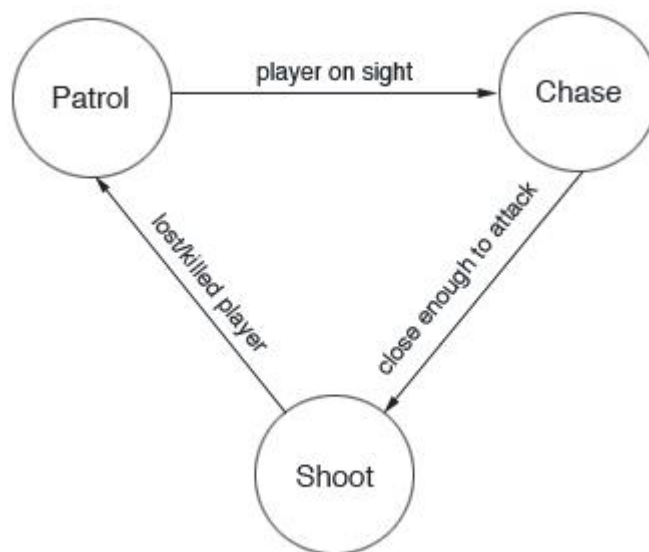


Рисунок 1 – Пример конечного автомата состояний

Зачатую конечный автомат используется только для основных действий, например, патрулирование, атака, преследование, диалог. Для действий, которые могут выполняться параллельно этим задачам создаются правила поведения на основе триггеров. Например, в ExtremeAI входит детектор звука, который позволяет агенту, это объект под управление

искусственного интеллекта, возможность слышать невидимые цели. Когда цели игроков попадают в радиус слышимости ИИ, их движение и скорость отслеживаются для имитации уровня шума. Если игрок движется достаточно быстро, это увеличит шанс, что ИИ обнаружит его. Так же есть модуль урона в зависимости от местоположения, который позволяет определять в какую часть модели нанесён урон и в зависимости от этого строить реакцию и высчитывать количество получаемого урона на это.

ExtremeAI предоставляет модуль ведения «осознанного» боя. Неигровые персонажи понимают кого нужно атаковать первым, в зависимости от получаемого урона, прошлого опыта столкновения или на основе ближайшего атакующего, последнего атакующего, случайного атакующего. Для выбора следующего действия в бою используется распознавание закономерностей прошлых ходов или случайные догадки.

Для отслеживания решений, принятых ранее, и оценка их успешности создаются таблицы взаимодействий. Система ИИ регистрирует выбор, сделанный игроком в прошлом. Все принятые в прошлом решения оцениваются по мере успешности, например, в качестве неё можно использовать полученное или утраченное преимущество, потерянное здоровье или нанесенный урон. Так же собираются дополнительные сведения о ситуации, чтобы образовать контекст для решений, например относительный уровень здоровья, прежние действия и положение на уровне.

Можно оценивать историю для определения успешности прежних действий и принятия решения о том, нужно ли изменять тактику. До создания списка прежних действий объект может использовать стандартную тактику или действовать произвольно. Эту систему можно увязать с системами на основе правил и с различными состояниями.

В игре история прошлых боев поможет выбрать наилучшую тактику для использования против команды игрока, например ИИ может играть от обороны, выбрать наступательную тактику, атаковать всеми силами невзирая на потери или же избрать сбалансированный подход. Такое решение поможет

ИИ лучше приспособиться к естественному стилю игрока, изучая его действия [1].

Агенты под управлением ExtremeAI обладают полноценной системой обзора для обнаружения противников и ориентации в окружающем мире. У каждого объекта есть свое поле зрения и углы обзора. Агент не может видеть сквозь и препятствия и может потерять противника или напарника, если тот вышел за пределы его поля видимости. Такой функционал позволяет создавать более реалистичных противников в игровом приложении. Для перемещения по карте ExtremeAI использует систему обзора и встроенную систему навигации по карте от Unity с названием NavMesh. При помощи NavMesh автоматически создаются собственные навигационные сетки и путевые точки. Постобработка пути с использованием рейкастинга, сглаживания и алгоритма воронки. Графики пути можно сохранять в файлы, что позволяет строить более быстрые и правильные маршруты при повторном перемещении на этой местности.

Главным недостатком ExtremeAI является оптимизация его для разных игровых платформ. Если актив использовать для разработки игровых приложения для смартфонов, то модуль умного управления и модуль взаимоотношений не могут быть использованы. Это связано с малой производительностью данных устройств. При активации всех модулей на компьютере финальная частота кадров в секунду падает до 20-25, что является ужасным, при том, что для комфортной игры пользователю нужно минимум 60 кадров.

3 Выбор инструментов разработки

Игровой движок предоставляет интегрированную среду разработки или редактор, который позволяет разработчикам проектировать игровые уровни, создавать и импортировать ресурсы, писать код или сценарии, а также тестировать и отлаживать игру. Они предлагают множество функций, которые упрощают процесс разработки и позволяют разработчикам сосредоточиться на создании игрового контента и механики, а не создавать все элементы с нуля. Каждый игровой движок имеет свои сильные стороны, особенности и модели лицензирования, и разработчики выбирают тот движок, который лучше всего соответствует потребностям их проекта и уникальным настройкам разработки.

UNITY на данный момент является самым популярным игровым движком среди независимых разработчиков и небольших студий. Предоставляет удобный интерфейс и визуальный редактор, которые оптимизируют процесс разработки игр и позволяют разработчикам быстро создавать прототипы и реализовывать идеи.

UNITY поддерживает широкий спектр платформ ПК, Mac, консоли, iOS и Android, веб-браузеры, дополненную и виртуальную реальность. UNITY позволяет расширять свою функциональность при помощи пользовательских плагинов и расширений.

При всех многочисленных преимуществах UNITY у него также есть недостатки, которые следует учитывать. Главной проблемой из них - это проблемы оптимизации больших проектов. Разработчикам может потребоваться вручную оптимизировать использование ресурсов, внедрить эффективные методы кодирования и использовать инструменты профилирования UNITY для выявления и устранения узких мест в производительности.

Редактор UNITY может испытывать проблемы с производительностью, особенно при работе с большими проектами или сложными сценами. Это может привести к замедлению работы,

запаздыванию и снижению производительности в процессе разработки. Хотя UNITY имеет эти ограничения, многие из них могут быть смягчены с помощью тщательного планирования, методов оптимизации и использования обширных ресурсов, доступных в сообществе UNITY. Кроме того, активное развитие UNITY и стремление к совершенствованию означают, что многие из этих недостатков постоянно устраняются в новых обновлениях и версиях движка.

Главным конкурентом UNITY является игровой движок UnrealEngine, который перекрывает его недостатки, но обретает свои. UnrealEngine славится своими исключительными графическими возможностями и мастерством рендеринга. Он обеспечивает высококачественную физическую систему рендеринга, которая создает реалистичное освещение, тени, отражения и эффекты постобработки, поддерживает динамическое освещение, глобальное освещение в режиме реального времени и предварительно рассчитанные сценарии освещения. Система визуальных сценариев UnrealEngine, называемая Blueprint, позволяет разработчикам создавать логику игрового процесса, поведение искусственного интеллекта и сложные взаимодействия без написания кода. UnrealEngine включает в себя мощное физическое моделирование и системы частиц. Он предлагает надежные взаимодействия, основанные на физике, физику тряпичной куклы и разрушаемое окружение. UnrealEngine предоставляет ряд инструментов, помогающих разработчикам оптимизировать свои игры для повышения производительности. Это включает в себя встроенные инструменты профилирования, анализаторы производительности и рекомендации по оптимизации. Разработчики могут выявлять и устранять узкие места в производительности, повышать частоту кадров и оптимизировать использование ресурсов для обеспечения плавного и оптимизированного игрового процесса [2].

Недостатком является расширенные функции и графические возможности UnrealEngine, которые реализуются за счет увеличения

сложности и накладных расходов на производительность. Система рендеринга движка и визуальные эффекты могут быть требовательны к системным ресурсам, требуя тщательной оптимизации и управления для достижения оптимальной производительности, особенно на слабом оборудовании или мобильных платформах.

Несмотря на эти недостатки, UnrealEngine остаётся мощным и широко используемым игровым движком для создания больших проектов или проектов с реалистичной графикой.

Таблица 1 - Достоинства и недостатки игровых движков

Наименование	Достоинства	Недостатки
UNITY	Простота использования и Скорость разработки	Проблемы оптимизации производительности
	Кроссплатформенная разработка	Графика
	Расширяемость с помощью плагинов	Проблемы редактора
Unreal Engine	Графика	Долгое и сложное обучения
	Системы физики и частиц	Большие размеры файлов готовых файлов
	Кроссплатформенная разработка	Проблемы с мобильной платформой
	Инструменты оптимизации и повышения производительности	Требования к оборудованию разработчика

В качестве игрового движка для выполнения работы был выбран UNITY и язык C#. C# — это язык программирования, разработанный и

запущенный Microsoft в 2001 году. С# — строго типизированный объектно-ориентированный язык программирования, который предоставляет современным разработчикам гибкость и лёгкость для создания программного обеспечения. Так же он является кроссплатформенным языком программирования. Приложения, написанные на С#, используют среду выполнения .NET, библиотеки классов. В настоящее время .NET представляет собой платформу и среду программирования для кроссплатформенной разработки. Благодаря платформе .NET можно создавать приложения, которые могут быть запущены на платформах Windows, Linux и Mac [3]. Во время разработки приложений в UNITY используется среда выполнения Mono. Она представляет собой реализацию .NETFramework, основанную на стандартах ECMA для С# и CommonLanguageRuntime. Стоит упомянуть ещё стандартный класс UNITY - MonoBehaviour, который является базовым классом для всех скриптов. При помощи этого класса программист может получать доступ к игровым объектам сцены, например, к объекту персонажа, чтобы заставить его двигаться. Если создать класс без наследования от MonoBehaviour, тогда он будет являться обычным классом .NET и его нельзя будет использовать с UNITY редактором напрямую.

4 Архитектура приложения

4.1 Игровое приложение

Стратегия в реальном времени - жанр компьютерных игр, в которой игрок, управляя группой сущностей и противостоит противнику. Главное свойство стратегий - управление игровым процессом происходит путем отдачи приказов, которые юниты самостоятельно выполняют.

Целью игрового процесса RTS является уничтожение противника, но в рамках игрового процесса могут быть и другие цели, например, захват ключевой точки или оборона от наступающих сил противника, но все они связаны с уничтожением врага.

Идеальный геймплей зависит от эффективности действий игрока, от его навыка стратегического и тактического планирования, а не от скорости реакции. Для комфортной игры нужно соблюсти некоторые требования к игровым механикам.

Все игровое поле должно иметь одинаковую ценность для игрока, то есть результат боевых действий не изменится, если перенести их в другую точку карты. Не должно быть такого, что в одних местах обороняться проще, чем в других, что меняет баланс сил, например, мосты, холмы. Коэффициент обороны достаточно велик, чтобы был смысл оборонять позиции, но недостаточен, чтобы цепляться за каждый клочок земли.

Цена потерь должна быть такой, чтобы потеря одного юнита была досадным событием, но не ставил крест на дальнейшей игре. Соответственно, должна быть возможность пополнения армии во время игровой сессии. Жёстко заданный стартовый набор юнитов приведёт к тому, что игроки будут стремиться любой ценой избегать риска.

Правила игры должны быть таковы, чтобы игрок, имеющий равные силы с врагом, не мог победить, просто атаковав все толпой. Любые успешные действия должны требовать осознанности и понимания игровых механик. Сложные и неочевидные игровые механики, обход с тыла,

окружение, должны иметь существенно большую эффективность, чем прямые действия.

Для развития базы и создания своей армии необходимы ресурсы. Системы ресурсов одна из основ стратегий в реальном времени, так как на её основе игрок может анализировать свою готовность делать следующие шаги. Учитывая важность системы, она должна приятной и интересной частью игры. Обычно борьбе с противником или созданию инфраструктуры уделяется очень много внимания, чего нельзя сказать о добыче ресурсов.

4.2 Принятие решений

Существует несколько способов реализации способности ИИ принимать решения. Самый простой метод - это конечные автоматы (FSM). FSM моделируют поведение искусственного интеллекта как набор состояний и переходов между ними. Каждое состояние представляет собой определённое поведение, и переходы происходят на основе предопределённых условий. Этот подход полезен для представления персонажей искусственного интеллекта с различным поведением и реакциями.

Вторым методом принятия решений являются - деревья поведения (BehaviorTree). Деревья поведения обеспечивают иерархическую структуру для определения процесса принятия решений искусственным интеллектом. Они состоят из узлов, которые представляют действия, условия или составные части. Дерево просматривается от корня до листьев, направляя процесс принятия решений искусственным интеллектом.

Третий способ принятий решений ИИ - это системы, основанные на полезности. Такие системы присваивают значения различным действиям в зависимости от их желательности для достижения конкретных целей. Затем искусственный интеллект выбирает действие с наибольшей полезностью. Такой подход позволяет гибко принимать решения, основанные на меняющихся обстоятельствах. Все действия заранее предопределены и

хранятся в базе данных, во время игры их значение полезности для достижения конкретной цели меняется.

Четвёртый способ – HFSM. В традиционном автомате с конечными состояниями (FSM) поведение системы определяется набором состояний и переходов между этими состояниями, основанных на определенных условиях или событиях. Однако, когда имеешь дело со сложными системами, FSM может стать трудным для управления и понимания из-за огромного количества состояний и переходов. HFSM решают эту проблему путем введения иерархической структуры. В HFSM состояния могут быть организованы в древовидную иерархию, где состояния более высокого уровня инкапсулируют и управляют поведением состояний более низкого уровня. Такая иерархическая организация обеспечивает более управляемое и модульное проектирование, а также лучшее представление сложных систем [3].

Во всех этих подходах процесс принятия решений является детерминированным, что означает, что действия модуля искусственного интеллекта основаны на наборе предопределённых правил и стратегий, а не на алгоритмах машинного обучения, которые корректируются на основе данных. Это может быть преимуществом для игр в жанре «Стратегия в реальном времени», где на исход сражения может повлиять широкий спектр факторов, и трудно предсказать, какие действия приведут к победе. Используя детерминированный подход к принятию решений, модуль искусственного интеллекта может обеспечить игрокам последовательный и сложный опыт, не полагаясь на алгоритмы машинного обучения, которые не всегда могут давать желаемые результаты. Так же машинное обучение не используется из-за ограниченной производительности систем, на которых должно запускать приложение. Третье причиной малой популярности машинного обучения - это сложность обучения ИИ во время разработки и тестирования. Когда игра только разрабатывается модель невозможно

обучить правилам игры за отсутствием на этом этапе примеров игры для обучения.

Еще один алгоритм планирования, который разбивает проблему на ряд более мелких подзадач – это STRIPS. Затем он использует алгоритм эвристического поиска для поиска решения каждой подзадачи. Окончательное решение затем объединяется из решений подзадач. STRIPS использовался для решения множества проблем в области искусственного интеллекта, включая проблемы планирования, проблемы распределения ресурсов и проблемы поиска пути. Это способ представления действий и целей как набора предварительных условий и последствий.

Одним из примеров применения STRIPS являются автоматизированные системы планирования. Эти системы используют STRIPS для создания планов достижения целей. Например, систему планирования можно использовать для создания плана сборки автомобиля. Система начинала с цели сборки автомобиля, а затем использовала STRIPS для создания плана достижения этой цели. В плане будут указаны этапы, необходимые для сборки автомобиля, такие как установка двигателя в автомобиль и прикрепление колес.

STRIPS имеет некоторые ограничения. Во-первых, он может иметь дело только с ограниченным числом действий и целей. Во-вторых, он может найти только те планы, которые гарантированно будут работать. В-третьих, он не может найти планы, которые могут сработать, но могут потерпеть неудачу. Также поиск плана может занять много времени, а найденные планы часто бывают длинными и сложными.

GOAP - метод является конкретной реализацией алгоритма STRIPS. обычно используемый в игровом ИИ для создания агентов, которые автономно определяют свои действия и достигают поставленных целей в игровой среде. GOAP можно использовать для создания сложного и адаптивного поведения неигровых персонажей в игре.

Основная идея GOAP - разложить сложную задачу на серию мелких и простых действий, которые может выполнить агент. Эти действия организуют план или последовательность действий, которые приведут агента к желаемой цели. GOAP начинает с того, что агент оценивает свое текущее состояние и желаемое целевое состояние. Затем агент просматривает доступные действия, чтобы найти серию действий, которые преобразуют текущее состояние в желаемое целевое состояние. Каждое действие связано с набором предварительных условий, которые должны быть выполнены для выполнения действия. Например, действие по получению ключа может иметь предварительное условие, что ключ находится в той же комнате, что и агент.

После создания плана агент выполняет первое действие в плане. По мере того, как агент завершает каждое действие, он переоценивает свое состояние и оставшиеся действия в плане, чтобы убедиться, что он все еще находится на пути к своей цели. Если агент сталкивается с препятствием или изменением игровой среды, он может динамически корректировать свой план, чтобы найти новую последовательность действий, которая все равно приведет к желаемой цели. GOAP полезен в играх со сложной средой и множеством целей, поскольку позволяет неигровым персонажам адаптироваться к меняющимся ситуациям и принимать решения на основе их текущего состояния и желаемого результата. Его также можно использовать для создания NPC с разными личностями или моделями поведения, регулируя вес различных действий или целей в их процессе принятия решений. В целом, GOAP — это мощный метод создания интеллектуальных и адаптивных агентов в играх, которые могут выполнять сложные задачи и достигать конкретных целей в игровом мире.

При наличии одной цели, лучше использовать конечный автомат или дерево поведения, потому что цель GOAP заключается в том, что ИИ имеет несколько целей, и он спланирует, как достичь её, основываясь на доступных действиях, поэтому, это станет хорошей отправной точкой как только появляются: действия, эффекты, условия и настроены затраты. Действия

имеют условия воздействия и базовые затраты, GOAP будет учитывать их при построении плана, поэтому действие с более высокой базовой стоимостью будет выбираться реже, когда доступны варианты с более низкой стоимостью. Определяются цели, которые ИИ будет достигать, а не состояния, которые он будет принимать. Определяются все действия, которые может выполнять конкретный агент, а затем система области действия использует действия. Помимо стоимости выполнения определенного действия, влияние, которое это действие окажет на состояние мира, и определяет, какой план действий следует предпринять для достижения определенной цели со всеми платформами ИИ, у нас также есть датчики, которые информируют ИИ о своих знаниях о состоянии мира или память, чтобы он мог принимать решения на основе того, используется ли конечный автомат или дерево поведения, он может принимать решения о том, к какому поведению перейти или к какому состоянию перейти в систему GOAP, будет использовать эти данные для определения того, какие действия для выполнения формы

GOAP содержит несколько основных определений:

- условие - флаг-признак чего-то, может быть либо в состоянии “Да” или “Нет”,
- действие - абстрактный элемент-узел плана действий. GOAP ничего не знает о том, что происходит в «Действии» и требует явно переключать «Условия», на которые влияет это «Действие»,
- требования - набор «Условий», которые должны выполняться для того, чтобы «Действие» стало возможным к использованию,
- планировщик - получает список действий, стартовое состояние «Условий», цель, состоящую из таких же «Условий». На выход подает список действий в нужной последовательности для достижения указанной цели. Если цель недостижима - информирует об этом.

Преимущества GOAP в том, что возможно добиться столько действий, сколько хотим, и пусть ИИ создаст план или последовательность действий,

которые нужно выполнить для достижения этой цели, например, цель добыть еду, есть действия: собирать ягоды, охотиться, перемещать оружие - и бездействие - это действия, которые доступны ИИ, и его цель - получить как можно больше еды наиболее эффективным способом, поэтому сначала нужно назначить некоторые предварительные условия или требования для этих действий, например, сбор ягод может быть выполнен до тех пор, пока объект ягодного куста находится в пределах досягаемости, но действие «Охота» требует, чтобы ИИ имел оружие и находился в радиусе действия животного, поэтому предварительным условием для сбора ягод является нахождение в пределах досягаемости, а охота находится в пределах досягаемости и наличие оружия. Оба действия приводят к еде, будь то ягоды или мясо, но один требует дополнительных шагов, чтобы сбалансировать, скажем, охота дает вам больше еды, поэтому вместо того, чтобы рассматривать это как преимущество, например, охота дает сотню еды, а ягоды дают семьдесят, это нужно рассматривать это как затраты, поэтому сбор ягод немного дороже, чем охота, поэтому стоимость сбора ягод равна десяти, охота равна шести, а добыча оружия равна единице, поэтому, когда создается этот график, нужно найти кратчайший путь между конечными узлами. В этом случае выбор падает в пользу действия «Охота» даже с дополнительным предварительным условием, потому что стоимость ниже.

Основная цель ИИ управления ресурсами — обеспечить оптимальное использование доступных ресурсов для достижения стратегических преимуществ. Это включает в себя несколько ключевых задач:

- сбор ресурсов: автоматизация процесса эффективного выявления и сбора ресурсов с учетом близости, количества и рисков, связанных с расположением ресурсов,
- распределение ресурсов: принятие решения о том, как лучше всего распределить ресурсы между различными конкурирующими потребностями, такими как производство юнитов, оборона, исследования и расширение,

- балансировка экономики: балансировка внутриигровой экономики путем управления темпами притока и потребления ресурсов, предотвращения истощения ресурсов или неэффективного накопления запасов.

Разработка ИИ для управления ресурсами предполагает создание алгоритмов и процессов принятия решений, которые могут динамически адаптироваться к состоянию игры. Вот более детальный взгляд на стратегию реализации:

- система приоритетов позволяет ИИ динамически корректировать свой фокус в зависимости от текущей ситуации. Например, если ИИ подвергается атаке, приоритет может сместиться в сторону производства и защиты юнитов. В мирное время акцент может сместиться в сторону расширения и модернизации технологий,

- ИИ должен постоянно анализировать эффективность своих стратегий сбора и распределения ресурсов. Это включает в себя мониторинг темпов производства ресурсов, экономической эффективности юнитов и технологий и соответствующую корректировку стратегий.

Данная система представляет собой механизм, предназначенный для динамического формирования последовательности операций с целью достижения заданного результата в соответствии с текущими условиями среды. Например, агент оснащён целью "Нанести урон цели" и имеет в распоряжении набор операций, включающий "Стрелять", "Искать укрытие", "Подойти к цели", "Запросить помощь". Планировщик анализирует мгновенное состояние игровой ситуации и определяет оптимальную последовательность действий. В случае наличия поблизости дополнительного агента, система может сформировать следующую цепочку действий: "Запросить помощь" -> "Подойти к цели" -> "Стрелять".

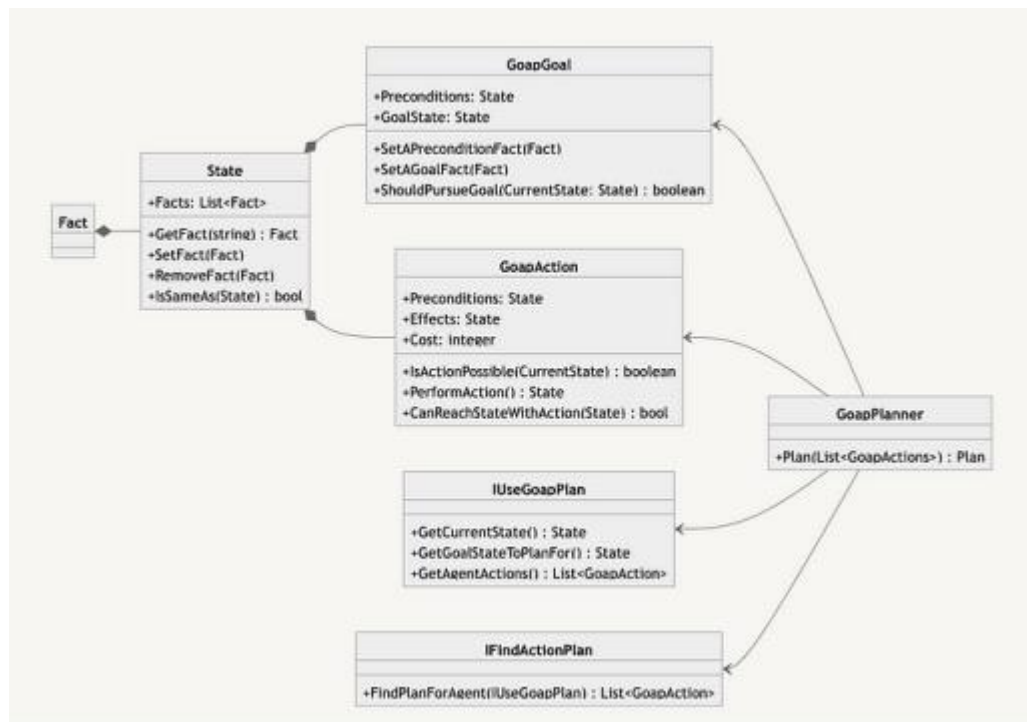


Рисунок 2 – UML диаграмма

Факт - это элементарная правда о мире, воспринимаемая агентом. Они хранятся в виде простого логического значения.

Состояние - это совокупность множества фактов, которые вместе описывают весь мир в том виде, в каком он воспринимается агентом. Для простоты в этой реализации использовался словарь фактов, где key- это название факта, а value - логическое значение факта.

Цель - это желаемый эффект или состояние, которого агент хочет достичь. Поскольку у агента может быть несколько целей, нам нужен способ расставить приоритеты и выбрать 1 цель из многих. По этой причине состояние предварительного условия и приоритет будут, как часть цели. Цель задается в инспекторе класса Agent. Можно выбрать, будет ли цель достигнута один раз или нет. Может быть несколько целей, и все они могут быть достигнуты. Планировщик выберет то, которое имеет более высокий приоритет, и это определит, какой набор действий будет использовать агент. Каждое действие направлено на достижение цели. Не все действия непосредственно приведут к цели. Предварительное состояние - это состояние, которому должно соответствовать текущее состояние для выполнения действия. Каждое совершенное действие может изменить

состояние агента, но действие будет выполнено только в том случае, если оно включено в плановую таблицу выполнения. Еще до того, как действие будет добавлено в таблицу выполнения, планировщик отфильтрует действия, которые не входят в план или не прошли проверку на возможность использования.

4.3 Поиск пути

Искусственный интеллект также применяется для передвижения неигровых персонажей по карте. При реализации алгоритмов поиска необходимо представить карту в виде графа или сетки, где узлы представляют местоположения на карте, а ребра представляют возможные соединения между узлами. Стоимость в каждом крае зависит от расстояние или сложность местности. Алгоритм оперирует этим графиком или сеткой, чтобы найти оптимальный путь от начальной до конечной точки, принимая во внимание затраты и оставшееся расстояние. Важным нюансом при поиске пути на графике, путь должен содержать в себе минимальный вес ребер, именно такой путь будет являться наиболее оптимальным. Присутствует множество разных алгоритмов поиска путей по графам: алгоритм Дейкстры, поиск в ширину, алгоритм A-star, иерархический поиск путей.

A-star, как правило, более эффективен, чем поиск по ширине и алгоритм Дейкстры, особенно на больших графиках или картах. Потому что A-star использует эвристическую функцию для определения приоритетов перспективных путей, что позволяет ему исследовать меньшее количество узлов и быстрее достигать цели. Алгоритм A-star является может быть применен к различным типам карт на основе сетки и может обрабатывать сетки с одинаковыми затратами или графики с различными весами ребер.

Эффективность алгоритма A-start в значительной степени зависит от выбора эвристической функции. Эвристическая функция – это функция, используемая в алгоритмах искусственного интеллекта для оценки и выбора наилучшего решения. Она основывается на правилах, которые позволяют предположить, какое действие или решение будет наилучшим в данной

ситуации. Распространенные функции включают манхэттенское расстояние и евклидово расстояние. Эти эвристические методы дают приемлемые оценки, поскольку они никогда не завышают истинную стоимость.

Реализация алгоритма A-start подразумевает создание графического или сеточного представления игровой карты, определение стоимости и эвристических функций, а также поддержание открытого набора и необходимых структур данных для эффективного поиска и построения траектории.

В процессе работы алгоритма для вершин рассчитывается функция

$$f(v)=g(v)+h(v), \quad (1)$$

где:

- g - стоимость перехода из начальной ячейки в текущую. Это сумма всех ячеек, которые были посещены с момента выхода из первой ячейки,
- h - предполагаемая стоимость,
- f - длина пути до цели, которая складывается из пройденного расстояния $g(v)$ и оставшегося расстояния $h(v)$.

Две распространённые эвристические функции, используемые в A-start, - это евклидово расстояние и манхэттенское расстояние.

Евклидово расстояние вычисляет расстояние по прямой между двумя точками в двумерном пространстве. В контексте A-start он измеряет прямое расстояние между текущим узлом и целевым узлом, предполагая, что возможно движение по прямой линии. Он вычисляется с использованием формулы

$$d = \text{sqrt}((x_2 - x_1)^2 + (y_2 - y_1)^2), \quad (2)$$

где $(x1, y1)$ и $(x2, y2)$ - координаты текущего узла и целевого узла соответственно.

Евклидово расстояние является хорошей эвристикой, когда разрешено диагональное перемещение. Эвристика манхэттенского расстояния вычисляет расстояние между двумя точками путем измерения суммы абсолютных разностей их координат. В контексте A-start он вычисляет минимальное количество горизонтальных и вертикальных перемещений, необходимых для достижения целевого узла из текущего узла. Он вычисляется по формуле

$$d = |x2 - x1| + |y2 - y1|, \quad (3)$$

где $(x1, y1)$ и $(x2, y2)$ - координаты текущего узла и целевого узла соответственно.

Манхэттенское расстояние особенно полезно, когда перемещение по диагонали запрещено. Эти эвристические функции обеспечивают допустимые оценки расстояния до целевого узла, что означает, что они никогда не завышают фактическую стоимость. A-start использует эти эвристические значения в сочетании с ориентировочными расстояниями от начального узла, чтобы направлять поиск по наиболее перспективным путям.

Алгоритм A-start имеет в основе несколько шагов:

- инициализировать открытый набор и закрытый набор. Открытый набор содержит узлы, которые были обнаружены, но еще не исследованы, а закрытый набор содержит узлы, которые были исследованы;
- назначить предварительное значение расстояния каждому узлу. Начальному узлу присваивается расстояние, равное 0, а всем остальным узлам присваивается расстояние, равное бесконечности;
- вычислить эвристическое значение для каждого узла. Эвристическая функция оценивает расстояние от узла до целевого узла. Это

помогает направлять поиск к цели и является важной частью алгоритма A-start;

- добавить начальный узел в открытый набор;
- повторять следующие шаги до тех пор, пока открытый набор не опустеет:

- 1) выбрать узел с наименьшим предварительным расстоянием и эвристическим значением из открытого набора. Этот узел становится текущим узлом;

- 2) удалить текущий узел из открытого набора и добавьте его в закрытый набор;

- 3) проверить, является ли текущий узел целевым узлом. Если это так, то алгоритм нашел оптимальный путь;

- 4) для каждого соседа текущего узла вычислите предварительное расстояние от начального узла до соседа через текущий узел. Обновить предварительное расстояние соседа, если оно меньше ранее присвоенного значения;

- 5) если сосед не входит в закрытый набор, добавить его в открытый набор. Если оно уже есть в открытом наборе, сравнить недавно рассчитанное предварительное расстояние с текущим предварительным расстоянием и при необходимости обновить его;

- 6) Если открытый набор пуст и целевой узел не был достигнут, путь от начального к целевому узлу отсутствует, и алгоритм завершается или, если целевой узел достигнут, строится путь, следуя цепочке узлов с наименьшими ориентировочными расстояниями от начального узла до конечного узла [4].

На основе AStart алгоритма можно реализовать 4 вида графов:

- 1) grid,
- 2) recast,
- 3) navmesh,
- 4) point.

Grid график является наиболее простым графиком, он генерирует узлы в виде сетки. Отлично подходит для большинства сцен и особенно хорош, когда нужно обновить график во время выполнения.

Плюсы:

- сеточные графики хорошо работают со штрафами и тегами,
- обновления графиков происходят быстро,
- сканирование графика выполняется сравнительно быстро.

Минусы:

- использование памяти выше, чем при преобразованном графике,
- поиск пути на больших расстояниях может быть намного медленнее, чем на измененном графике,
- не масштабируется до очень больших миров.

Recast график создает треугольную навигационную сетку вместо сетки. Генератор Recast графиков вокселизует мир, а затем построит из него навигационную сетку, которую можно использовать аналогично NavmeshGraph. Фактически, система Unity также основана на recast.

При создании Recast графика происходит вокселизация мира. Можно рассматривать это как построение аппроксимации мира из множества блоков.

Плюсы:

- может представлять мелкие детали и большие области одновременно,
- поиск пути происходит быстро из-за малого количества узлов,
- поддерживает большие миры,
- быстрое обновление с помощью вырезания навигационной сетки (ограничивается в основном вырезанием отверстий в навигационной сетке).
- достаточно быстрые обновления графика, которые пересчитывают все измененные фрагменты графика,
- сравнительно низкое использование памяти.

Минусы:

- сканирование может быть медленным, особенно если вам нужна очень высокая детализация или очень большие миры,

- обновление выполняется медленнее, чем в виде сетки, но вырезание навигационной сетки выполняется быстрее, если можно ее использовать,

- большие волнистые пустые области, такие как холм без каких-либо деревьев или препятствий, могут затруднять правильное представление измененного графика. Однако включение разбиения на листы может помочь разделить большие узлы.

График navmesh представляет данные поиска пути в виде треугольной сетки вместо квадратов (график сетки) или точек (график точек).

Это идеально подходит для плавного и быстрого поиска пути, когда график не требует значительных изменений во время выполнения. Часто это быстрее, чем сеточный график, поскольку он обычно содержит меньше узлов и, следовательно, требует меньше поиска. Пути, возвращаемые из него, можно использовать напрямую, но настоятельно рекомендуется использовать модификатор воронки.

Плюсы:

- может представлять мелкие детали и большие области одновременно,

- поиск пути выполняется быстро из-за малого количества узлов,

- поддерживает большие миры,

- довольно быстрые обновления с использованием вырезания навигационной сетки (ограничивается в основном вырезанием отверстий в навигационной сетке),

- сравнительно низкое использование памяти,

- быстрое сканирование.

Минусы:

- для создания навигационной сетки требуется много ручной работы,

- графики Navmesh плохо представляют теги и штрафы.

PointGraph - самый простой из всех типов графиков, но допускает множество настроек. Он состоит из набора размещенных пользователем точек, которые связаны друг с другом.

Плюсы:

- полностью контролируемое расположение узлов,
- быстрое сканирование для простых случаев, но может стать очень медленным, если не быть осторожны с настройками.

Минусы:

- требуется много ручной работы для размещения всех узлов,
- трудно получить пути хорошего качества без очень большого количества узлов,
- поиск пути часто происходит медленно из-за большого количества узлов, которое требуется,
- медленное обновление графика.

4.4 Модель поиска объектов

4.4.1 Proximal Policy Optimization (PPO)

Алгоритм PPO был представлен командой OpenAI в 2017 году и быстро стал одним из самых популярных методов RL, превзойдя метод обучения Deep-Q. Он включает в себя сбор небольшой порции опыта взаимодействия с окружающей средой и использование этой порции для обновления политики принятия решений. После обновления политики с помощью этого пакета опыт отбрасывается и собирается новый пакет с новой обновленной политикой. Именно по этой причине это подход «обучения в рамках политики», при котором собранные образцы опыта полезны только для однократного обновления текущей политики.

Ключевой вклад PPO заключается в обеспечении того, чтобы новое обновление политики не слишком сильно меняло ее по сравнению с предыдущей политикой. Это приводит к меньшей дисперсии в обучении за счет некоторой предвзятости, но обеспечивает более плавное обучение, а

также гарантирует, что агент не пойдет по необратимому пути совершения бессмысленных действий.

Одна из проблем, от которых страдает обучение с подкреплением, заключается в том, что генерируемые обучающие данные сами по себе зависят от текущей политики, поскольку наши агенты генерируют свои собственные обучающие данные, взаимодействуя со средой, а не полагаясь на статический набор данных, как в случае контролируемого выполнения. Это означает, что распределение данных наблюдений и вознаграждений постоянно меняется по мере обучения агента, что является основной причиной нестабильности всего процесса обучения. Помимо этой проблемы с различным распределением обучающих данных, обучение с подкреплением также страдает от очень высокой чувствительности к настройке и инициализации гиперпараметров.

Основная цель PPO - найти баланс между простотой реализации, эффективностью выборки и простотой настройки. Первое, что нужно осознать в отношении PPO, это то, что называется методом политического градиента. В отличие от популярных ключевых подходов к обучению, которые позволяют учиться на основе хранящихся в автономном режиме данных, PPO учится онлайн. А это значит, что он не использует буфер воспроизведения для хранения прошлого опыта, а вместо этого учится непосредственно на том, с чем его агент сталкивается в окружающей среде. После того, как пакет опыта был использован для обновления градиента, этот опыт отбрасывается и политика продолжается. Таким образом, методы градиента политики обычно менее эффективны на выборке, чем методы Q-обучения, поскольку они используют собранный опыт только один раз для выполнения обновления. В целом методы оптимизации политики обычно начинаются с описания законов градиента политики следующим образом:

$$LPG(\theta) = Et [\log \pi \theta(a_t | s_t) A_t], \quad (3)$$

где:

- E_t - это ожидание,
- $a_t|s_t$ - это действия по состоянию,
- A_t - это оценка функции преимущества.

Первый термин q -theta - это политика, нейронная сеть, которая принимает наблюдаемые состояния из окружающей среды в качестве входных данных и предлагает действия в качестве выходных. Вторым членом - это функция преимущества, которая в основном пытается оценить относительную ценность выбранного действия в текущем состоянии. Чтобы вычислить преимущество, нужны две вещи:

Сумма вознаграждений со скидкой (или Возврат) — это взвешенная сумма всех вознаграждений, полученных агентом за каждый временной шаг в текущем эпизоде.

- коэффициент дисконтирования (гамма) $\gamma \in [0,1]$ – это текущая стоимость будущего вознаграждения,
- значение получения вознаграждения R после $k+1$ временных шагов равно $\gamma^k R$,
- это ценит немедленное вознаграждение выше отсроченного вознаграждения.

Если γ близкое к 0 приводит к «близорукой» оценке и в другом случае, если γ близкое к 1 приводит к «дальновидной» оценке.

Коэффициент дисконтирования γ , который обычно находится в диапазоне 0,9–0,99, учитывает тот факт, что агент больше заботится о вознаграждении, которое будет получено очень быстро, а не о том вознаграждении, которое он получит через сто временных шагов.

Базовая оценка или функция стоимости пытается дать оценку дисконтированной суммы вознаграждений с этого момента. По сути, это предположение о том, каким будет окончательный результат в этом эпизоде, начиная с текущего состояния. Во время обучения эта нейронная сеть, представляющая функцию ценности, будет часто обновляться с

использованием опыта, который агенты собирают в окружающей среде. По сути, это задача контролируемого обучения: принимаются состояния в качестве входных данных, и нейронная сеть пытается предсказать, какой будет сумма вознаграждений со скидкой, начиная с этого состояния. Поскольку эта оценка значения является результатом работы нейронной сети, она будет зашумленной.

Таким образом, если вычесть базовую оценку, то, что произойдет, из дисконтированных вознаграждений, то, что произошло, получается то, что называется оценкой преимущества, которая показывает, насколько лучше были действия, предпринятые агентом на основе в ожидании того, что обычно происходит в том состоянии, в котором обычно находится агент.

Если A_t значение, из $LPG(\theta)$, положительное, то градиент будет положительным, и это увеличит вероятность повторного выбора в будущем, когда повторится это же состояние и если значение отрицательное, то это уменьшит вероятность таких действий.

Если продолжить использовать градиентный спуск для одной партии собранных данных, то произойдет следующее: обновятся параметры в сети настолько, что они выйдут за пределы диапазона, в котором были собраны эти данные, что, например, функция преимущества, которая, является зашумленной оценкой реального преимущества, будет совершенно неправильным. Таким образом, в некотором смысле нарушается политика, если продолжить использовать градиентный спуск для одной партии собранного опыта. Чтобы решить эту проблему, одним из успешных подходов является обеспечение того, чтобы при обновлении политики никогда не отходить слишком далеко от старой политики.

4.4.2 Soft Actor-Critic (SAC)

Второй алгоритм, который используется в реализации, — это алгоритм SAC, также предоставленный командой OpenAI. SoftActorCritic (SAC) — это алгоритм, оптимизирующий стохастический внеполитической политики, образуя мост между стохастической оптимизацией политики и подходами

глубокого детерминированного политического градиента (в стиле DDPG). SAC также является методом критики актеров, аналогичным DDPG и TD3. Что касается названия, «Soft» означает регуляризованную энтропию, которая играет важную роль в преобразованиях алгоритма внутри нейронной сети. Хотя алгоритм считается неполитическим алгоритмом, на самом деле он может использовать буфер воспроизведения, что является важным критерием его структуры. Этот буфер воспроизведения создает сходство, о котором упоминается ранее, с TD3 и DDPG.

Добавляя энтропийную регуляризацию, алгоритм способствует исследованию. Таким образом, это позволяет прийти к лучшему оптимуму. В противном случае вопрос соблюдения локального минимума будет иметь решающее значение для нашей подготовки.

Что касается характеристики стохастического актера, стоит отметить, что существуют реализации, которые вместо этого используют детерминированный актер. В этом случае SAC становится точно таким же, как TD3, поскольку при этом удаляется член энтропийной регуляризации, и поэтому решения являются строгими.

На рисунке 3 можно выделить различия и взаимосвязи между упомянутыми алгоритмами с точки зрения их структурных характеристик.



Рисунок 3 - График категоризации алгоритма обучения RL

- подходы TRPO и PPO: стохастическая политика, соблюдение политики, низкая эффективность выборки, стабильность,
- подход DDPG и TD3: детерминированные политики, буфер воспроизведения, лучшая эффективность выборки, нестабильность,
- подход SAC: стохастическая политика, буфер воспроизведения, энтропийная регуляризация, стабильность и эффективность выборки.

5 Реализация игровой логики

5.1 Настройка сцены и игрового мира

В создании сложных и многоуровневых игровых проектов, где каждый элемент играет ключевую роль в обеспечении непрерывности и динамики разработке, первостепенной задачей становится настройка и грамотная организация иерархии проекта в редакторе.

В центре сцены находится компонент GameManager, играющий роль основного элемента проекта. Этот компонент запускается первым при инициализации сцены и берет на себя ответственность за активацию и настройку всех прочих компонентов игры. GameManager объединяет в себе дочерние объекты, каждый из которых содержит определенные компоненты управления, специализированные на различных аспектах игрового процесса. Эти компоненты управления организованы в группы по принципу функциональности.

Также важную роль играет объект Terrain, представляющий собой базовую местность, видимую в режиме просмотра сцены. Он служит фоном для действия и задает первоначальную атмосферу игрового мира.

Объект Resources выполняет функцию контейнера для предварительно размещенных ресурсов на карте, то есть тех элементов, которые не генерируются динамически в процессе игры. Эти ресурсы могут включать разнообразные объекты, начиная от элементов окружения и заканчивая предметами, с которыми взаимодействует игрок.

Наконец, Factions описывает игровые сущности, принадлежащие различным фракциям, с которыми игрок встречается на старте игры. Эти объекты не создаются во время игры, но служат стартовой точкой для развития игровых событий и конфликтов между различными группами внутри игрового мира.

5.2 Менеджер местности

Скрипт TerrainManager играет ключевую роль в управлении и манипулировании ландшафтом игровой карты, обеспечивая разработчикам

мощный инструмент для настройки и адаптации игрового пространства под специфические потребности игры. Этот компонент предоставляет возможности для детальной настройки различных типов областей рельефа, включая горы, равнины, водные пространства и другие уникальные особенности местности, которые могут оказывать влияние на игровой процесс и стратегии игроков.

С помощью скрипта TerrainManager, возможно интегрировать физические свойства рельефа, влияющие на передвижение и взаимодействие объектов в игре, например, скорость перемещения по разнородным типам местности или доступность определенных участков карты для игрока и NPC.

TerrainManager также включает в себя инструменты для оптимизации производительности, такие как динамическая подгрузка и выгрузка участков местности в зависимости от положения игрока, что существенно снижает нагрузку на системные ресурсы и обеспечивает плавное и качественное отображение игрового мира даже на ограниченном оборудовании. Скрипт TerrainManager - важный компонент управления, отвечающий за обработку типов областей рельефа на карте и в настройке предусмотрено несколько ключевых полей, каждое из которых выполняет определенную функцию в процессе создания и управления местностью игрового поля.

Поле «HeightCaching», играет важную роль в оптимизации производительности, так как предусматривает предварительное кэширование высот объектов игрового поля до начала игры. Это необходимо, поскольку многие компоненты во время выполнения требуют доступа к данным о высоте, а использование кэширования исключает необходимость повторного просчета высот с помощью лучей для каждого такого запроса. При выборе объекта с компонентом TerrainManager на сцене отображается гизмо, указывающее область, которая будет кэшироваться. Важно, чтобы в процесс кэширования была включена вся карта. В разделе настроек кэширования высоты представлены поля, такие как "LowerLeftCorner", обозначающее левый нижний угол карты в качестве начальной точки для кэширования, и

"RightLeftCorner", указывающее на правый верхний угол карты. "HeightCacheDistance" задает расстояние перемещения для кэширования высоты с нижнего левого угла в направлении верхнего правого угла карты. Кроме того, "HeightCacheSampleOffset" добавляет смещение к позиции луча, используемого для определения и кэширования высоты местности, улучшая тем самым точность и эффективность процесса кэширования.

В процессе настройки местности важным элементом является конфигурация поля «BaseTerrain», которое задает основу для всей сцены карты. Это основание представляет собой коробчатый коллайдер высотой 1,0 у.е., охватывающий всю карту и даже ее пределы, создающийся автоматически при запуске сцены. Коллайдер базового рельефа служит множеству целей, включая взаимодействие с камерой через щелчки и кэширование моделей объектов. Ключевым аспектом является, что любой луч, выпущенный из основной камеры в любом допустимом положении, должен сталкиваться с этим коллайдером, гарантируя тем самым, что камера всегда будет иметь опору в пространстве сцены.

Для детализации базового рельефа предусмотрено несколько полей настройки. Поле «Базовый слой» определяет предварительно установленный слой для создания коллайдера, который используется не только для кэширования и взаимодействия с камерой, но и для функционирования миникарты, по умолчанию назначенный как «BaseTerrain». Центр базовой местности, указываемый в поле «Центр базовой местности», должен располагаться ниже остальной местности, обеспечивая тем самым единую основу для всей карты. Размер базового рельефа, задаваемый в поле «Размер базового рельефа», должен превышать общую площадь карты, чтобы обеспечить постоянное взаимодействие лучей камеры с коллайдером из любой точки и при любом положении камеры.

После настройки базового рельефа следующим шагом является конфигурация компонента «GridSearchHandler» в менеджере местности. Этот компонент критически важен для оптимизации процесса поиска объектов,

таких как юниты, здания и ресурсы, на карте. Он работает путем разделения карты на сетку ячеек, что значительно уменьшает вычислительные затраты для операций поиска, поскольку поиск теперь не требует обработки всей местности за один раз, а ограничивается только релевантными ячейками. Это повышает эффективность и скорость поиска, делая процесс более управляемым и менее затратным с точки зрения производительности.

5.3 Игровой менеджер

Основной компонент в сцене игровой карты – это игровой менеджер. При запуске сцены это первый компонент, который инициализируется и отвечает за инициализацию остальных компонентов сцены. Все компоненты менеджера должны быть прикреплены к дочерним объектам объекта, к которому прикреплён этот компонент. Кроме того, он обрабатывает определение слотов фракций, которые могут быть заполнены фракциями, контролируемыми игроками или неигровыми. Менеджер содержит в себе много настраиваемых свойств для настройки игрового процесса, но самое интересно это поле «Условие поражения». По умолчанию, определяющее, как выиграть/проиграть эту карту. Условия поражения включают:

- устранение главного - все основные сущности фракции из слота фракции выбывают, слот фракции помечается как побежденный.
- устранить всех - все объекты фракции в слоте фракции устранены, слот фракции помечается как побежденный.

Каждая сцена карты должна иметь список слотов фракций, определенных в компоненте GameManager, который устанавливает максимальное количество фракций, разрешенных для игры в этой сцене карты, и устанавливает данные и конфигурации по умолчанию для каждого слота фракций, которые принимаются во внимание только при непосредственном запуске сцены карты и воспроизведении ее.

Слоты фракций включают информацию об игровом автомате, которая определяет права доступа к нему в игре. Например, поле «Роль»

присваивается «Хосту», что означает, что игрок, использующий этот слот, является хостом игры и имеет полные административные права.

Поле «Тип» позволяет вам определить конкретный тип для фракции, занимающей слот, который позволит другим компонентам вести себя по-разному в зависимости от этого типа. Например, может быть здание, которое нужно разделить между разными типами фракций, но одно и то же здание позволяет создавать разные юниты в зависимости от типа фракции. Это можно сделать, указав, какие задачи по созданию юнитов будут видны в одном и том же здании в зависимости от типа фракции, и всякий раз, когда фракция размещает это здание, она инициализирует свои задачи по созданию юнитов, запрашивая тип фракции, частью которой она является, и включает только задачи по созданию юнитов, связанные с этим типом. Также возможно оставить это поле неназначенным, что означает, что слот фракции не имеет какого-либо определенного типа.

5.4 Создание и настройка юнитов

Компонент «Unit» является основным компонентом объектов определяющих игровых персонажей на карте, он определяет его тип и общие конфигурации. Этот компонент должен быть прикреплён к родительскому объекту каждого персонажа.

Компонент «UnitHealth» является вторым обязательным компонентом игровых персонажей, который должен быть присоединён к ним. Он управляет основными характеристиками, обновлением здоровья юнита, обработкой его уничтожения, а также позволяет визуально обновлять объекты юнита, чтобы показать игроку состояние его здоровья.

Компонент «UnitMovement» является основным компонентом, управляющим перемещением персонажей. Он содержит в себе поле «Приоритет», которое определяет порядок, в котором компоненты будут рассматриваться для задания целевого значения. Компонент с наименьшим значением приоритета будет рассмотрен первым, и если он не позволяет достичь цели, установленной игроком, то компонент со следующим более

высоким значением приоритета будет рассматриваться следующим, пока один из компонентов не примет цель как действительную. В нашем случае в настоящее время у нас есть только компонент

5.5 Создание игровых ресурсов

Ввод ресурсов в игру начинается с разработки и определения типов ресурсов, которые будут играть роль в игровом процессе. Каждый ресурс представлен объектом, обладающим уникальными характеристиками и поддерживающим нужные ему игровые механики. Каждый объект включает в себя атрибут, устанавливающий ограничение по объему. Некоторые ресурсы могут иметь ограниченный объем, что позволяет контролировать их доступность и количество. В случае с ресурсом, как древесина, ограничения на объем не накладывается, оставляя возможность его неограниченного использования и хранения. Следующим атрибутом ресурса является начальное количество. Это значение определяет, сколько единиц ресурса изначально доступно каждой фракции или игроку. Следующие компоненты “Resource” и “ResourceHealth”. Они задают основные параметры и настройки типа ресурса, а также его состояния и возможные интеракции. Необходимо ассоциировать эти компоненты с моделью объекта ресурса, обеспечивая идентичность настройки полям компонентов “Unit” и “UnitHealth”.

5.6 Настройка атаки

Чтобы юниты могли атаковать и сражаться друг с другом, им необходимо добавить и настроить скрипт "UnitAttack". Этот скрипт является ключевым элементом игровой механики, позволяя юнитам осуществлять атаки на противников, используя заданные параметры для определения силы удара, дальности атаки и скорости нанесения урона. В процессе настройки данного скрипта разработчикам предстоит решить ряд задач, связанных с балансировкой и адаптацией поведения юнитов под конкретные условия игры.

Первым шагом в настройке скрипта "UnitAttack" является определение типа атаки для каждого юнита. Это может быть ближний бой, дальний бой

или специальные атаки, такие как магия или поддержка. Каждый тип атаки требует настройки соответствующих параметров, радиус действия, время перезарядки и мощность удара.

Затем следует система наведения и выбора цели, позволяющая юнитам определять наиболее подходящего противника для атаки в текущих обстоятельствах. Эта система может включать в себя алгоритмы приоритизации целей на основе факторов, включая расстояние до цели, уровень угрозы и текущее состояние здоровья юнита. Интеграция системы обнаружения позволяет юнитам реагировать на изменения в окружающей среде, адаптируя своё поведение для максимальной эффективности в бою.

Кроме того, для обеспечения сбалансированного геймплея необходимо настраивать параметры урона и защиты юнитов, учитывая возможности их улучшения в ходе игры. Система прогрессии позволяет юнитам повышать свои боевые характеристики, приобретая новые умения и улучшая существующие, что добавляет глубины стратегическому планированию.

Настройки компонентов атаки в игровой системе позволяют контролировать, как и когда юниты могут атаковать врагов. В дальнейшем будут описаны основные из настроек, переменных и методы.

UnitAttack класс, который обрабатывает поведение атаки для юнитов в игре. Вот разбивка основных функций и возможностей. Класс содержит различные атрибуты, такие как сам отряд, настройки формирования атаки, опция движения при атаке, расстояние следования, настройки атаки-движения и т. д. Методы инициализации и прекращения поведения атаки, при которых обработчики событий для начала и остановки движения регистрируются и не регистрируются. Логика для остановки поведения атаки, включая обновление целей вращения и потенциальное возобновление атаки, если она была активна. Логика определения того, когда остановиться или продолжить атаку на цель, с учетом таких факторов, как движение, положение цели, препятствия и дальность атаки. Методы расчета и обновления позиции атаки, гарантирующие, что она находится в пределах

достижимости и не заблокирована препятствиями. Функции для установки и обновления цели с учетом таких факторов, как достоверность позиции и необходимость движения подразделения для атаки.

`OnAttackInit` метод вызывается при инициализации атакующего поведения юнита. Он настраивает необходимые компоненты и обработчики событий для поведения атаки. `OnAttackDisabled` вызывается, когда поведение атаки отключено. Он отменяет регистрацию обработчиков событий и выполняет всю необходимую очистку. `OnAttackStop` вызывается, когда атака прекращается. Он обрабатывает остановку движения юнита и потенциально возобновляет атаку-движение, если оно было активно.

`HandleMovementStart` вызывается, когда начинается движение юнита. Он проверяет, является ли движение частью команды атаки-движения, и обрабатывает его соответствующим образом. `HandleMovementStop` вызывается, когда движение юнита прекращается. Он проверяет, было ли движение частью команды атаки-движения, и при необходимости отключает атаку-движение.

`MustStopProgress` определяет, следует ли останавливать атаку в зависимости от различных условий, таких как движение цели, препятствия и дальность атаки. `CanEnableProgress` проверяет, можно ли включить прогресс атаки на основе таких условий, как способность юнита двигаться и находится ли цель в пределах досягаемости. `OnEnterTargetRange` вызывается, когда юнит входит в зону атаки своей цели. Он записывает начальную дистанцию поражения между устройством и целью. `OnInProgressEnabled` метод, вызываемый при включении прогресса атаки. Он выполняет все необходимые действия при запуске атаки. `OnComplete` вызывается, когда атака завершена. Он выполняет любые заключительные действия, необходимые после завершения атаки.

Настройка «Формирование» движения позволяет определить, как будут генерироваться позиции атаки нескольких юнитов одного типа, когда их попросят вступить в бой с одним и тем же целевым объектом. Существует

два predetermined типа построения: `circular_movement_formation_type` и `row_movement_formation_type`. В первом случае юниты будут образовывать круг в месте назначения цели, в то время как во втором будут генерироваться направления траекторий атаки в виде ряда.

6 Реализация модуля планирования действий

Общая схема работы система планировщика представлена на рисунке 4. Основные классы для работы планировщика будут описаны ниже в подпунктах.

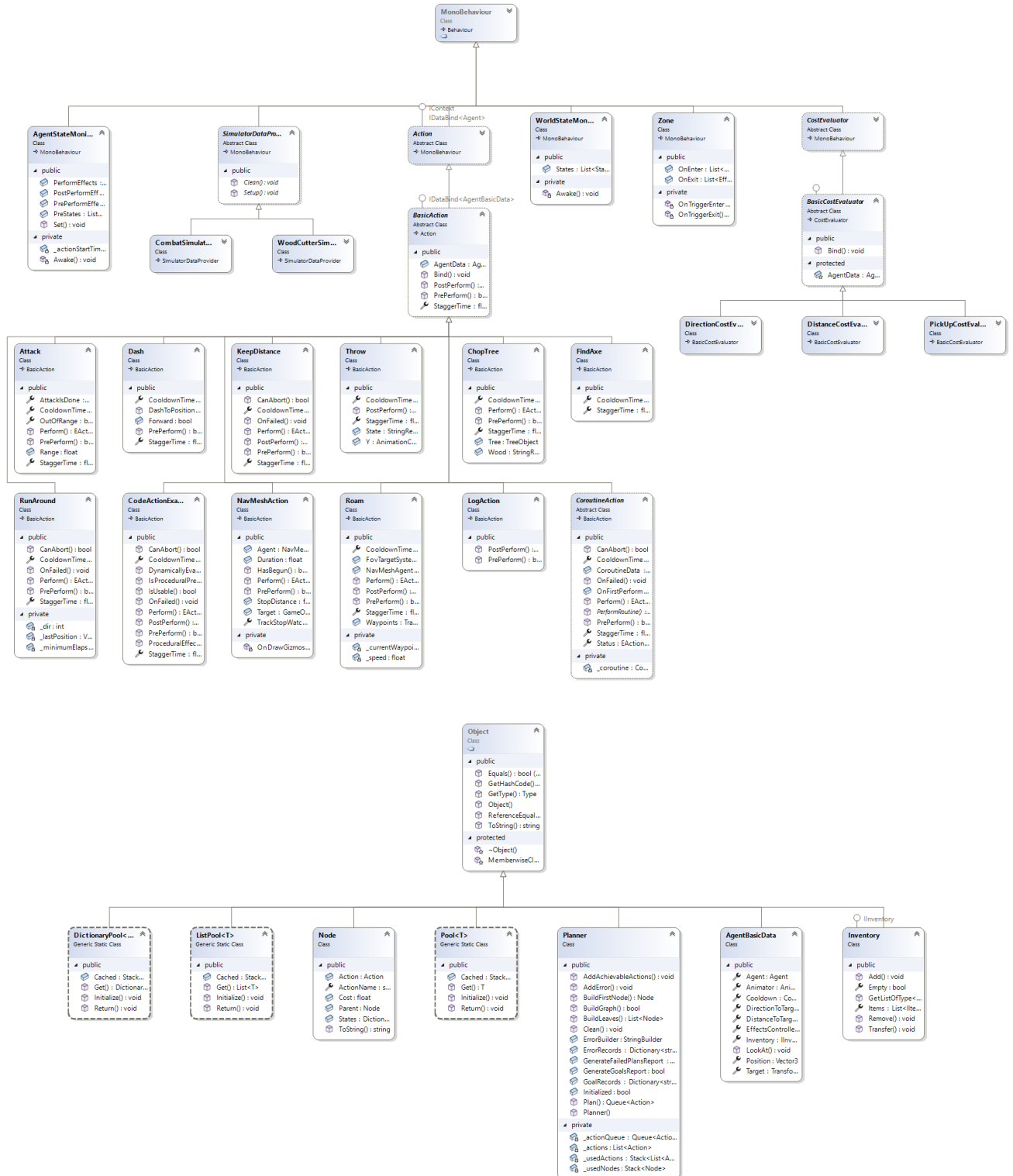


Рисунок 4 - UML диаграмма классов планировщика

Класс `MonoBehaviour` - это базовый класс, производный для многих скриптов Unity. `MonoBehaviour` предлагает функции жизненного цикла, которые упрощают разработку с Unity. Этот класс всегда существует как компонент игрового объекта и может быть создан с помощью `GameObject.AddComponent`.

6.1 Класс Action

Для изменения текущего состояния агента или мира, должны происходить игровые действия, описывающие изменения и последствия. Действие будет выполнено только в том случае, если оно включено в плановую таблицу выполнения. Важно знать, что еще до того, как действие будет рассмотрено для последовательности планирования, планировщик отфильтрует действия, которые не имеют `AlwaysIncludeInPlan` или не прошли проверку `Action.IsUsable`. Все классы, реализующие свою логику действия, должны наследоваться от класса «Action» и переопределять внутри себя пять основных методов:

1) `Action.DynamicallyEvaluateCost` - метод вызывается один раз для каждого действия, выполняемого агентом. Как правило, гораздо дешевле выполнять какие-либо динамические расчеты затрат здесь, чем позже,

2) `Action.IsPreconditionsMet` - предварительные условия здесь, то, что добавляется в компонент действия в инспекторе. Если действие не соответствует какому-либо предварительному условию, оно не будет включено в план,

3) `Action.IsProceduralPreconditionMet` - метод, который нужно переопределить, чтобы добавить предварительное условие с помощью кода, основанного на текущем состоянии планирования. Ничто из добавленного вами не повлияет на состояние мира, это просто для планировщика,

4) `Action.DynamicallyEvaluateCost` - в отличие от первого вызова `DynamicallyEvaluateCost`, этот вызов должен содержать в себе текущее состояние планировщика, то есть возможно рассчитать будущие затраты.

Например, текущее состояние говорит, что окажетесь в состоянии А после выполнения действия X,

5) `Action.ShouldAbort` - метод чтобы действие можно было прервать. Это означает, что, если в середине пути к дереву в вас стреляют, нужно иметь возможность выйти из этой задачи и снова спланировать действия.

Все действия, добавленные в таблицу, выполняются по одному циклу, представляющему повторяющийся вызов трех методов:

1) `Action.PrePerform` - метод вызывается перед выполнением действия, это идеальное место для любой настройки и повторной инициализации,

2) `Action.Perform` - основной метод действия, именно в нем нужно реализовать свою игровую логику, после завершения действия вызовет `PostPerform`. Метод продолжает выполняться до тех пор пока не вернется `True` или `False`. Возврат `True` происходит в случае, если действие завершилось самостоятельно, `False` возвращается, если происходит ошибка или вызывается метод `Action.ShouldAbort`,

3) `Action.PostPerform` - вызывается только после возврата `Action.Success`.

6.2 Планировщик

Класс `Planner` основной класс ИИ, который реализует базовый планировщик, использующий подход поиска в глубину для поиска последовательности действий, которые приводят к заданному целевому состоянию. Внутри себя он реализует различные оптимизации, такие как объединение памяти в пул, для эффективного управления использованием памяти.

Основным методом класса является метод `Plan`. Он принимает различные параметры, такие как агент, цель, состояния мира, действия и состояния агента. Затем он строит листья дерева планирования и выбирает оптимальный путь для достижения цели. Следующим методом по важности является `BuildLeaves`. Он строит листья дерева планирования на основе

заданной цели, состояний мира, действий и состояний агентов. Он использует рекурсивный поиск в глубину для изучения возможных последовательностей действий. Метод начинается с добавления достижимых действий с помощью `AddAchievableActions` метода, который заполняет `actions` список на основе предоставленного файла `actions`. Затем создается начальный узел `firstNode`, представляющий начальное состояние процесса планирования. Этот узел включает в себя как мировые состояния, так и состояния агентов, предоставленные в качестве параметров. Метод вызывает `BuildGraph` метод для рекурсивного построения графика планирования, начиная с `firstNode`. Он передает начальный узел, список для сбора конечных узлов, список действий и состояние цели. Результат этого исследования сохраняется в `leaves` списке. Затем метод проверяет, были ли найдены какие-либо листовые узлы во время построения. Если был найден хотя бы один листовой узел, он устанавливает флаг `success`, указывающий, что планировщик смог найти хотя бы один потенциальный план для достижения цели. В конце концов метод возвращает список конечных узлов, которые представляют потенциальные планы, позволяющие достичь цели.

```
public List<Node>BuildLeaves(string goal, Dictionary<string,
float>worldStates, List<Action> actions, Dictionary<string,
float>agentStates, out bool success, bool simulate = false)
{
    AddAchievableActions(actions, simulate);
    var firstNode = BuildFirstNode(worldStates, agentStates);
    _usedNodes.Push(firstNode);
    var leaves = ListPool<Node>.Get();
    BuildGraph(firstNode, leaves, _actions, goal);
    success = leaves.Count > 0;
    Profiler.EndSample();
    return leaves;
}
```

Еще один метод `BuildGraph`, рекурсивно строящий график планирования, расширяя узлы с применимыми действиями до тех пор, пока

цель не будет достигнута или больше действий нельзя будет применить. Метод перебирает каждое действие в usableActions списке, чтобы рассмотреть потенциальный вклад каждого действия в достижение цели. Для каждого действия проверяется, выполняются ли предварительные условия действия в текущем состоянии. Если предварительные условия действия не выполняются, оно пропускает действие, в другом случае предварительные условия действия выполнены, метод создает новый узел, представляющий применение этого действия из текущего состояния. Затем он рекурсивно вызывает BuildGraph, передавая новый узел в качестве родительского, эффективно исследуя граф, чтобы рассмотреть последствия применения этого действия. Если применение действия приводит к достижению цели, т. е. состояние goal присутствует в результирующем состоянии после применения действия, текущий узел добавляется в leaves список, указывая потенциальный план, который достигает цели.

```
public bool BuildGraph(Node parent, List<Node> leaves,
List<Action> usableActions, string goal)
{
    var foundPath = false;
    foreach (var usableAction in usableActions)
    {
        var usable = usableAction.IsPreconditionsMet(parent.States);
        if (!usable)
        {
            if (GenerateFailedPlansReport)
                AddError(parent, usableAction, parent.States);
            continue;
        }
        if (!usableAction.IsProceduralPreconditionMet(parent.States))
            continue;
        var currentState = DictionaryPool<string, float>.Get();
        var node = Pool<Node>.Get();
        _usedNodes.Push(node);
        node.States = currentState;
```

```

node.Action = usableAction;
node.ActionName = usableAction.Name;
node.Parent = parent;
node.Cost = parent.Cost + usableAction.Cost;
node.Cost +=
usableAction.DynamicallyEvaluateCost(parent.States);
foreach (var parentStateinparent.States)
    {
        if(!currentState.ContAInsKey(parentState.Key))
        currentState.Add(parentState.Key, parentState.Value);
    }
foreach (var effect inusableAction.AfterEffects)
    {
        if(!currentState.ContAInsKey(effect.Key))
        currentState.Add(effect.Key, effect.Value);
    }
usableAction.ProceduralEffect(currentState);
if (currentState.ContAInsKey(goal))
    {
        leaves.Add(node);
        foundPath = true;
    }
else
    {
        varsubsetActions = ListPool<Action>.Get();
        _usedActions.Push(subsetActions);
        foreach (var action inusableActions)
        subsetActions.Add(action);
        subsetActions.Remove(usableAction);
        foundPath = BuildGraph(node, leaves, subsetActions, goal);
    }
}

returnfoundPath;
}
}

```

Метод `AddAchievableActions` добавляет в планировщик достижимые действия. Он динамически оценивает удобство использования и стоимость каждого действия.

Для реализации дерева реализовано несколько служебных классов для объединения памяти: `Node`, `Pool`, `ListPool` и `DictionaryPool`. Эти служебные классы используются для эффективного управления и повторного использования экземпляров универсальных типов, сокращая накладные расходы на выделение памяти.

Класс `Node` представляет узел в графе планирования. Содержит такую информацию, как текущее состояние, связанное действие, родительский узел, стоимость и имя действия.

Класс `Pool` ответственен за объединение в пул экземпляров указанного типа. Предоставляет методы для инициализации пула определенным количеством экземпляров и получения или возврата экземпляров из/в пул. Гарантирует повторное использование экземпляров, когда это возможно, уменьшая необходимость частого выделения и освобождения памяти. Класс `ListPool` аналогичен `Pool`, но специализируется на объединении в пул экземпляров `List`. Класс `DictionaryPool` аналогичен классу `ListPool`, но специализируется на объединении экземпляров `Dictionary`.

Эти классы объединения предназначены для использования в сценариях, где существует необходимость частого создания и уничтожения объектов универсальных типов, например, в классе планировщика, где узлы, списки и словари неоднократно создаются и удаляются во время планирования. Использование пула снижает затраты на выделение памяти и сборку мусора, что приводит к повышению производительности и уменьшению фрагментации памяти.

6.3 Класс агента

Этот скрипт определяет `Agent` класс, который представляет сущность в игре, которая может достигать целей, планируя и выполняя действия. Он

предоставляет методы для планирования, выполнения действий, обработки результатов действий и динамического управления целями.

Список Goals охватывает все возможные цели, которые агент может ставить перед собой. Каждая цель определяется конкретными условиями или задачами, которые агент стремится выполнить. Цели могут варьироваться от простых до сложных и отражают конечные задачи, к которым стремится агент.

Все состояния агента, в которых он может находиться содержатся в списке States. Состояния включают информацию о местоположении агента, его здоровье, ресурсы, которыми он располагает, и любые другие параметры, отражающие его текущее положение и условия. Эти состояния играют ключевую роль в принятии решений о действиях предпринятых в будущем. Все действия, которые агент планирует выполнить, содержатся в очереди StoredActionQueue. Она действует как буфер о запланированных задачах, позволяя агенту организовывать свои действия по мере их выполнения.

Цель, к которой агент стремиться сейчас, хранится в переменной CurrentGoal. Текущая цель является основой для планирования и выполнения действий, и она может изменяться в зависимости от обстоятельств или после достижения предыдущей цели.

Действие CurrentAction, которое агент выполняет в данный момент. Текущее действие выбирается из очереди действий на основе текущей стратегии и целей агента. Это ключевой момент в процессе выполнения плана, где агент взаимодействует с окружающей средой или системой.

Список Actions всех действий, доступных агенту для выполнения. Эти действия могут включать перемещение, сбор ресурсов, взаимодействие с объектами или другими агентами и так далее. Выбор конкретного действия зависит от текущей стратегии и направлен на достижение текущей цели,

ActionQueue очередь содержит действия, которые агент планирует выполнить для достижения своих целей. Она обновляется по мере того, как агент переходит от одного действия к другому, и предоставляет

структурированный план действий, которые необходимо выполнить для достижения конечных результатов.

Метод Run является точкой входа для запуска цикла планирования и выполнения агентом. Он вызывается для активации процесса, который включает в себя поиск плана для достижения поставленной цели, последующее выполнение найденных действий, и переоценку состояния цели. Этот процесс продолжается до тех пор, пока не будут выполнены все задачи или пока не будет принято решение об их прекращении.

```
Public virtual void Run()
{
    if (ExecutePlan())
        return;
    FindPlan();
    EvaluationGoal();
    PrePerformPlanAction();
}
```

Задача метода FindPlan заключается в поиске оптимального набора действий, который приведет агента к достижению его текущей цели. Метод анализирует возможные пути и выбирает наиболее эффективный на основе заданных критериев и доступных данных.

```
Public virtual void FindPlan()
{
    if (_replan || ActionQueue == null)
    {
        foreach (var goal in _orderedGoals)
        {
            Profiler.BeginSample("Agent.Plan");
            ActionQueue = Planner.Plan(this, goal.Key,
World.Instance.StateMap, Actions, States.GetStates(), false);
            _planStartTime = Time.time;
            Profiler.EndSample();
            if (ActionQueue == null)
                continue;
        }
    }
}
```

```

CurrentGoal = goal;
    _replan = false;
StoredActionQueue.Clear();
foreach (var action in ActionQueue)
StoredActionQueue.Enqueue(action);
break;
    }
}
}

```

Метод `ExecutePlan` отвечает за выполнение текущего действия из очереди планируемых. Этот метод взаимодействует с средой или системой для выполнения конкретного задания и переходит к следующему действию в очереди после его завершения.

```

Public virtual bool ExecutePlan()
{
    if (CurrentAction != null && CurrentAction.Running)
    {
        if (ShouldAbort())
            return true;
        PerformPlanAction();
        Return true;
    }
    Return false;
}

```

Метод `PrePerformPlanAction` выполняет подготовительные операции перед непосредственным выполнением следующего действия. Это может включать в себя настройку параметров, проверку условий для выполнения действия и другие подготовительные действия, необходимые для успешного выполнения.

```

Public virtual void PrePerformPlanAction()
{
    Profiler.BeginSample("Agent.PrePerform");
    if (ActionQueue != null && ActionQueue.Count > 0)
    {

```

```

CurrentAction = ActionQueue.Dequeue();
if (CurrentAction.PrePerform())
{
    if (CurrentAction.TrackStopWatch)
        CurrentAction.Stopwatch.Restart();
    else
        CurrentAction.Stopwatch.Reset();
    CurrentAction.Running = true;
    CurrentAction.OnPrePerform?.Invoke();
}
else
    AbortPlan();
}
Profiler.EndSample();
}

```

Метод `PerformPlanAction` непосредственное выполнение действия и обработка его результата. После выполнения каждого действия этот метод анализирует результаты и определяет, как они влияют на общий прогресс достижения цели.

```

Public virtual void PerformPlanAction()
{
    Profiler.BeginSample("Agent.Run");
    Var actionStatus = CurrentAction.Perform();
    CurrentAction.OnPerform?.Invoke();
    CheckActionFailed(actionStatus);
    CheckActionSuccess(actionStatus);
    Profiler.EndSample();
    Profiler.BeginSample("Agent.CanAbortPlan");
    if
(PlannerSettings.CanAbortPlans&&CurrentAction.CanAbort()
&&Time.time - _planStartTime>PlannerSettings.PlanRate)
        AbortPlan();
    Profiler.EndSample();
}

```

Метод `EvaluationGoal` проверяет достигнута ли текущая цель. Если цель не достигнута или обстоятельства изменились, метод может инициировать процесс перепланирования, вызывая необходимость в поиске нового плана действий.

```
Public virtual void EvaluationGoal()  
{  
    if (ActionQueue != null&&ActionQueue.Count == 0)  
    {  
        if (CurrentGoal.Once)  
        {  
            Goals.Remove(CurrentGoal);  
            _orderedGoals.Remove(CurrentGoal);  
        }  
        _replan = true;  
    }  
}
```

Метод `ForceReplan` используется для принудительного перепланирования, в случаях, когда изменение внешних условий или новая информация требует пересмотра текущего плана.

Метод `InitializeActions` обновляет список действий, которые агент может выполнить. Это может включать добавление новых действий, удаление устаревших или изменение параметров существующих действий в соответствии с новой информацией или стратегией.

Метод `UpdateGoalOrderCache` переоценивает и обновляет порядок целей в соответствии с их текущими приоритетами. Это обеспечивает, что агент будет стремиться достичь наиболее важных целей в первую очередь, опираясь на изменяющуюся ситуацию и доступную информацию.

7 Обучение модели поиска ресурсов

В качестве библиотеки для обучения NPC будет использоваться библиотека ML-Agents от Unity - это проект с открытым исходным кодом, который позволяет играм и симуляциям служить средой для обучения интеллектуальных агентов. Предоставляется реализация на основе PyTorch для обучения агентов с использованием обучения с подкреплением, имитационного обучения, нейроэволюции или любых других методов.

С помощью ML-Agents можно обучать поведение неигровых персонажей, называемых Агентами, используя различные методы. Основная идея довольно проста. Нужно определять три объекта в каждый момент игры, называемые окружением:

- наблюдения - то, что агент воспринимает об окружающей среде. Наблюдения могут быть числовыми и / или визуальными. Числовые наблюдения измеряют атрибуты окружающей среды с точки зрения агента. В нашем случае это будут видимые ему атрибуты поля боя. Для большинства интересных окружений агенту потребуется несколько непрерывных числовых наблюдений. Визуальные наблюдения, с другой стороны, представляют собой изображения, генерируемые камерами, прикрепленными к агенту, и отражают то, что агент видит в данный момент времени. Состояние окружающей среды представляет информацию обо всей сцене, содержащей всех игровых персонажей. Однако наблюдение агентов содержит только информацию, о которой известно агенту, и обычно является подмножеством состояния среды. Например, наблюдение агента не может включать информацию о скрывающемся враге, о котором он не знает,

- действия - какие действия может предпринять агент. Подобно наблюдениям, действия могут быть непрерывными или дискретными в зависимости от сложности среды и агента. В текщемслуча, если среда представляет собой простой мир, где имеет значение только их местоположение, достаточно дискретного действия,

– сигналы вознаграждения - скалярное значение, указывающее, насколько хорошо работает агент. Сигнал о вознаграждении не обязательно подавать в каждый момент, а только тогда, когда агент выполняет хорошее или плохое действие. Сигналом вознаграждения является то, как цели задачи доводятся до агента, поэтому их необходимо настроить таким образом, чтобы максимизация вознаграждения порождала желаемое оптимальное поведение [10].

7.1 Настройка гиперпараметров

Ниже приводится список из 8 гиперпараметров и краткое описание того, что они используют

- алгоритм обучения:
 - 1) train_type - тип используемого тренера: PPO или SAC;
- Конфигурации трейнеров, общие для всех трейнеров:
 - 1) max_steps - общее количество шагов (т. е. собранных наблюдений и предпринятых действий), которые необходимо предпринять в окружающей среде;
 - 2) time_horizon - сколько шагов опыта нужно собрать для каждого агента, прежде чем добавлять его в буфер опыта;
 - 3) summary_freq - количество опыта, которое необходимо собрать перед созданием и отображением статистики обучения;
 - 4) keep_checkpoints - максимальное количество сохраняемых контрольных точек модели;
 - 5) checkpoint_interval - количество опыта, полученного тренером между каждой контрольной точкой;
 - 6) threaded -логическое значение устанавливает разрешить ли средам изменяться при обновлении модели;
- общие гиперпараметры:
 - 1) batch_size - количество опытов в каждой итерации градиентного спуска;

- 2) `buffer_size` - количество данных, которые необходимо собрать перед обновлением модели политики;
- 3) `learning_rate` - начальная скорость обучения для градиентного спуска;
- 4) `learning_rate_schedule` - определяет, как скорость обучения меняется с течением времени;
 - PPO-специфические гиперпараметры:
 - 1) `beta` - Сила энтропийной регуляризации, которая делает политику «более случайной»;
 - 2) `lambda` - параметр регуляризации (лямбда), используемый при расчете обобщенной оценки преимущества (GAE);
 - 3) `num_epoch` - количество проходов через буфер опыта при выполнении оптимизации градиентного спуска;
 - SAC-специфичные гиперпараметры:
 - 1) `buffer_init_steps` - количество событий, которые нужно собрать в буфер перед обновлением модели политики;
 - 2) `init_entcoef` - сколько агенту следует изучить в начале обучения;
 - 3) `save_replay_buffer` - следует ли сохранять и загружать буфер воспроизведения опыта, а также модель при выходе и повторном запуске обучения;
 - 4) `tau` - насколько активно обновлять целевую сеть, используемую для начальной оценки значения в SAC;
 - 5) `steps_per_update` - Среднее соотношение шагов (действий), предпринятых агентом, к обновлениям политики агента;
 - 6) `reward_signal_steps_per_update` - количество шагов на мини-пакет, выбранное и используемое для обновления сигналов вознаграждения;
 - network settings:
 - 1) `normalize` - применяется ли нормализация к входным данным векторного наблюдения;

2) `hidden_units` - количество блоков в скрытых слоях нейронной сети;

3) `num_layers` - количество скрытых слоев в нейронной сети;

4) `vis_encode_type` - тип кодировщика для кодирования визуальных наблюдений;

– `reward_signals`:

1) `gamma` - коэффициент дисконтирования для будущих вознаграждений, поступающих из окружающей среды;

2) `power` - коэффициент, на который можно умножить вознаграждение, получаемое от окружающей среды;

– `self_play`:

1) `save_steps` - частота сохранения снимков;

2) `team_change` - частота смены обучающейся команды;

3) `wap_steps` - частота изменения фиксированной политики;

4) `window` - количество снимков, которые хранятся в соответствии с фиксированной политикой;

5) `play_against_latest_model_ratio` - вероятность выбора последней политики или случайной политики между остальными в стеке;

7.2 Структура агента

Агенты в этой конкретной сцене имеют следующие 3 характеристики:

– наблюдения. На каждом этапе наблюдения агента, его знания о состоянии окружающей среды:

1) положение в трехмерном пространстве, прибавляет 3 наблюдения;

2) вращение только по оси Y, прибавляет 1 наблюдение;

3) положение ближайшей цели, прибавляет 3 наблюдения;

4) датчики лучевого восприятия, прибавляют по 10 наблюдений;

– действия. Для агентов доступны следующие действия:

1) движение вперед и назад, дискретный тип;

2) вращение оси Y, дискретный тип;

- 3) прыжок, дискретный тип;
 - награда. Награды распределяются в следующем порядке:
 - 1) в случае падения агента с платформы награда будет установлена на -1.
 - 2) на каждом этапе будет добавлено вознаграждение (положительное), равное нормализации, разделенной на distanceFromClosestTarget, где нормализация равна 0,001.
 - 3) В случае достижения агентом цели его награда будет установлена на уровне +1 и -1 для противника, который технически проиграл.

7.3 Реализация агента

Для обучения агента следует реализовать свой класс Агента, который наследуется от класса Agent предоставленным библиотекой UnityML.

Этот метод отвечает за настройку датчиков, собирающих данные из окружающей среды. InitializeSensors метод инициализирует датчики для агента. Сначала он проверяет фабрику политик, чтобы определить, как обрабатывать наблюдаемые атрибуты. Если указана обработка наблюдаемых атрибутов, на основе этих атрибутов создаются датчики. Затем он собирает все SensorComponent экземпляры, прикрепленные к агенту или его дочерним объектам, если это настроено для этого. Затем он создает поле, VectorSensor если параметры мозга агента указывают векторное пространство наблюдения. Если требуется суммирование наблюдений, оно оборачивается VectorSensor расширением StackingSensor. Для обеспечения единообразия датчики отсортированы в алфавитном порядке по названиям. Это гарантирует, что имя каждого датчика уникально и не конфликтует с именем другого датчика.

Датчики собирают информацию об окружающей среде, которая важна для процесса принятия решений агентом. Эта информация может включать в себя положение, скорость, расстояние до объектов или любые другие данные, необходимые агенту для понимания своего окружения.

```
internal void InitializeSensors ()
```

```

        {
            if (m_PolicyFactory.ObservableAttributeHandling != 0)
            {
                boolexcludeInherited =
m_PolicyFactory.ObservableAttributeHandling ==
ObservableAttributeOptions.ExcludeInherited;
                using (TimerStack.Instance.Scoped("CreateSensors"))
                {
                    List<ISensor> collection =
ObservableAttribute.CreateObservableSensors(this,
excludeInherited);
                    sensors.AddRange(collection);
                }
            }

            SensorComponent[] array =
(!m_PolicyFactory.UseChildSensors) ?
GetComponent<SensorComponent>() :
GetComponentInChildren<SensorComponent>());
            sensors.Capacity += array.Length;
            SensorComponent[] array2 = array;
            foreach (SensorComponent sensorComponent in array2)
            {
                sensors.Add(sensorComponent.CreateSensor());
            }

            BrAInParameters brAInParameters =
m_PolicyFactory.BrAInParameters;
            if (brAInParameters.VectorObservationSize > 0)
            {
                collectObservationsSensor =
newVectorSensor(brAInParameters.VectorObservationSize);
                if (brAInParameters.NumStackedVectorObservations > 1)
                {
                    StackingSensor item =
newStackingSensor(collectObservationsSensor,
brAInParameters.NumStackedVectorObservations);
                    sensors.Add(item);
                }
            }
        }
    }
}

```

```

        }
    else
    {
        sensors.Add(collectObservationsSensor);
    }
}

sensors.Sort((ISensor x, ISensor y)
=>x.GetName().CompareTo(y.GetName()));
for (int j = 0; j <sensors.Count - 1; j++)
{
    Debug.Assert(!sensors[j].GetName().Equals(sensors[j +
1].GetName()), "Sensor names must be unique.");
}
}

```

Главной целью `InitializeActuators` метода инициализировать исполнительных механизмов для агента, выполняющих действия на основе решений агента. Собирает все `ActuatorComponent` экземпляры, прикрепленные к агенту или его дочерним объектам. Далее создает `VectorActuator` агента, который отвечает за выполнение непрерывных действий. Создает систему `ActuatorManager` для управления всеми исполнительными механизмами. Инициализирует массивы для кэширования действий и эвристики для агента.

Исполнители отвечают за выполнение действий, выбранных политикой агента. Действия могут включать в себя такие вещи, как движение, прыжки или любое другое поведение, необходимое агенту для взаимодействия с окружающей средой.

```

private void InitializeActuators()
{
    ActuatorComponent[] array =
        ((!m_PolicyFactory.UseChildActuators) ?
        GetComponents<ActuatorComponent>() :
        GetComponentsInChildren<ActuatorComponent>());
}

```

```

        BrAInParameters brAInParameters =
m_PolicyFactory.BrAInParameters;
        m_VectorActuator = newAgentVectorActuator(this, this,
brAInParameters.ActionSpec);
        m_ActuatorManager = newActuatorManager(array.Length + 1);
        m_LegacyActionCache =
newfloat[m_VectorActuator.TotalNumberOfActions()];
        m_LegacyHeuristicCache =
newfloat[m_VectorActuator.TotalNumberOfActions()];
        m_ActuatorManager.Add(m_VectorActuator);
        ActuatorComponent[] array2 = array;
        foreach (ActuatorComponent actuatorComponent in array2)
        {

m_ActuatorManager.AddActuators(actuatorComponent.CreateActuators
());
        }
    }
}

```

Метод `CollectObservations` вызывается каждый шаг, на котором Агент запрашивает решение. Это один из возможных способов сбора данных наблюдений Агента за окружающей средой. Поскольку метод `CollectObservations` вызывает `VectorSensor.AddObservation` таким образом, чтобы размер вектора составлял 8, параметры поведения Агента задаются с помощью векторного пространства наблюдения с размером состояния.

```

public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(transform.localRotation.y);
    sensor.AddObservation(targetTransform.localPosition);
    sensor.AddObservation(enemyTransform.localPosition);
}

```

Вызывается каждый раз, когда Агент получает задание на выполнение действия. Получает действие, выбранное Агентом. В этом методе также часто назначается вознаграждение.

```
public override void OnActionReceived(ActionBuffers actions)
{
    playerIsGrounded = characterController.isGrounded;
    Vector3 inputMovement = transform.forward *
movementSpeed * actions.ContinuousSActions[0];
    characterController.Move(inputMovement * Time.deltaTime);
    transform.Rotate(Vector3.up * actions.ContinuousSActions[1] *
rotationSpeed);

    if (Mathf.RoundToInt(actions.DiscreteActions[0]) == 1
&&playerIsGrounded)
    { movementDirection.y = jumpSpeed;    }
    movementDirection.y -= gravity * Time.deltaTime;
    characterController.Move(movementDirection *
Time.deltaTime);
    rewardAgent();
    if (hasAnimation)
    {changeAnimationState(actions);    }
}
```

Если в параметрах поведения Агента для типа поведения установлено значение "Только эвристический", Агент будет использовать метод `Heuristic()` для генерации действий Агента. Таким образом, метод `Heuristic()` выполняет запись в массив значений с плавающей точкой, предоставленный эвристическому методу в качестве аргумента. Примечание: Не создавайте новый массив действий с плавающей точкой в методе `Heuristic()`, так как это предотвратит запись значений с плавающей точкой в исходный массив действий.

```
public override void Heuristic(in ActionBuffers actionsOut)
{
    ActionSegment<float> continuousSActions =
actionsOut.ContinuousSActions;
```

```

    ActionSegment<int>discreateActions =
actionsOut.DiscreteActions;

    continuouSACTIONS[0] = Input.GetAxisRaw("Vertical");
    continuouSACTIONS[1] = Input.GetAxisRaw("Horizontal");
    discreateActions[0] = Input.GetButton("Jump") ? 1 : 0;
}

```

7.4 Обучение с помощью PPO

Первое обучение будет производится при помощи алгоритма PPO. Общее количество шагов обучения всего на 2.800.000 шагов. Первые 400.000 шагов агенты прыгают по кругам небольшого радиуса. Такое поведение связано со спамом всех возможных действий, постоянно вращающимся бегом и прыжками. Переходя к тренировочному процессу, на 800.000 шагов радиус кругов больше, но все равно смотреть здесь не на что. Около 1.200.000 до 1.600.000, вот где их любопытство начинает проявляться и можно наблюдать больше движений вперед. В процессе и на протяжении 2.000.000 шагов агенты понимают, что им нужно приближаться к целям, но им все еще не хватает знаний, чтобы правильно использовать лучевые приведения, поэтому они продолжают застревать на краях внутреннего пространства. стены. Огромное улучшение можно увидеть около 2.400.000. шаги, на которых агенты начали патрулировать территорию, бегая кругами большого радиуса. Всякий раз, когда они идентифицируют метку цели с помощью лучей, они перемещаются прямо к ней. Конечно, результаты не оптимальны, и у нас все еще есть некоторые промахи.

Наконец, на 2.800.000 шагах поведение оптимизируется. Агентам удается захватить большой процент целей, которые они видят. Стоит отметить, что агенты сначала больше полагаются на расстояние от ближайшей целевой цели, но в процессе захватывают цели, которые видят лучами. Вот почему они продолжают бегать кругами как можно быстрее, чтобы достичь цели скорее благодаря удаче, чем руководствуясь своим положением.

7.5 Обучение с SAC

Вторая тренировка производилась с использованием алгоритма SAC на 2.800.000 шагов. Сначала наблюдается то же поведение, что и при обучении PPO. Агенты просто прыгают небольшими кругами, чтобы получить знания о том, как передвигаться по платформе. По мере процесса обучения круги становятся все больше и больше в радиусе. Подобное поведение схоже с обучением по PPO. Ожидалось, что результаты состоят в том, что за меньшее время, чем PPO, агент сможет структурировать более достойное поведение с точки зрения точности, но агенты не могут этого сделать. Единственное изменение в их движении от начала до конца — это изменение размера радиуса. Фактически, единственная награда, которую обычно получают агенты, — это удача, а не умение. Перемещаясь по платформе, им удается получить положительные награды за достижение цели.

Поскольку используются дискретные действия, которые возвращают непрерывное состояние среды, новые позиции, повороты и т. д., сложность значительно возрастает. Таким образом, SAC нуждается в более интенсивной подготовке, следует, что PPO более удобен для пользователя, когда речь идет об обучении модели меньшего масштаба, которая не требует огромной вычислительной мощности.

7.6 Сравнение результатов

Контраст между двумя результатами огромен. С одной стороны, есть PPO с большим успехом по всем направлениям, с другой, есть SAC без точки оптимизации. В реализации дискретные действия возвращают непрерывные результаты. Когда поведение решает, что агент должен двигаться, например, вперед, в коде применяется цифра «1». Результат этого дискретного действия не заставляет агента продвинуться хотя бы на шаг вперед. Вместо этого он применяет силу к твердому телу агента, новое непрерывное положение применяется к новому состоянию окружающей среды. Большое количество возможных дискретных действий по значимости для настройки равно меньшему числу непрерывных. Рисунки с 5 по 10 представляют собой

графики для эксперимента, на них красная линия отображает процесс обучения при помощи алгоритма PPO, а синяя с помощью SAC. Совокупное вознаграждение за SAC было нестабильным на протяжении всего процесса обучения. PPO, с другой стороны, значительно увеличился до 2.000.000 шагов. Последующий спуск легко объяснить. Как описано в разделе «self_play», обучение агента сравнивается с устаревшей версией самого себя. В какой-то момент агенту удастся сформировать поведение, полностью уничтожающее противника. Но когда и фиксированная политика, и политика обучения находятся на высоком уровне, возможностей для улучшения мало. Таким образом, это индикатор оптимизации обучения самостоятельной игре. Другим индикатором является график «EpisodLength» на рисунке 5. Отсутствие каких-либо улучшений для SAC очевидно.

Что касается PPO, всего за 2.500.000 шагов получается оптимальное поведение. Разделы о потерях и политике для SAC просто доказывают, что алгоритм не смог должным образом обучиться. Все показатели, которые должны были увеличиваться, вместо этого уменьшались, и наоборот. Это увеличение доказывает, что алгоритм смог предсказать размер вознаграждения, которое он получит, еще до того, как он его получит. В разделе политики PPO есть бета, энтропия, эпсилон и скорость обучения. Эти значения плавно уменьшались, давая надежный результат для предположений. Логическое значение «IsTraining» сообщает нам, что оба обучения не были прерваны какой-либо ошибкой или ошибкой среды. В целом показатель «ELO» для самостоятельной игры является конечным результатом конкурентного моделирования, на рисунке 10 можно явно наблюдать как изменялось значение во время обучения. SAC не удалось набрать очки ELO, поскольку он никогда не тренировался должным образом. PPO, с другой стороны, продемонстрировал рост всего за один финальный спуск только потому, что тренировка для этих последних шагов была остановлена, а затем возобновлена. Поскольку ELO инициализируется значением 1200, этот спад не важен. На этом этапе нужно передать его PPO,

поскольку он обучил сложную задачу за небольшое количество шагов. Таким образом, настройка гиперпараметра представляется обязательным процессом перед началом любого обучения.

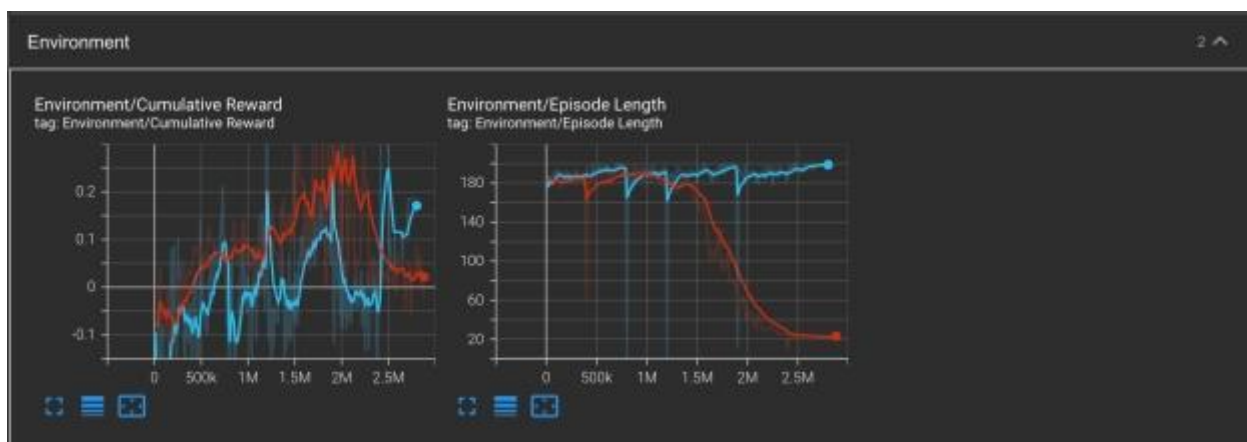


Рисунок 5 – Графики «Cumulative Revard» «Episode Length»

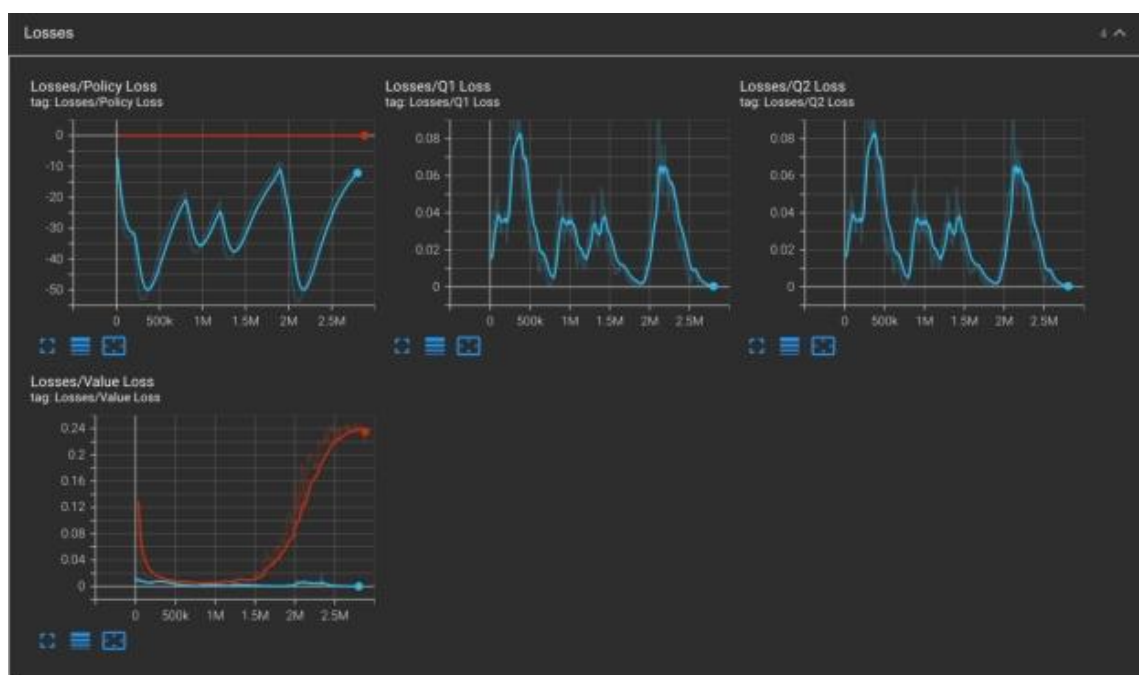


Рисунок 6 – Графики «Policy Loss», «Value Loss»

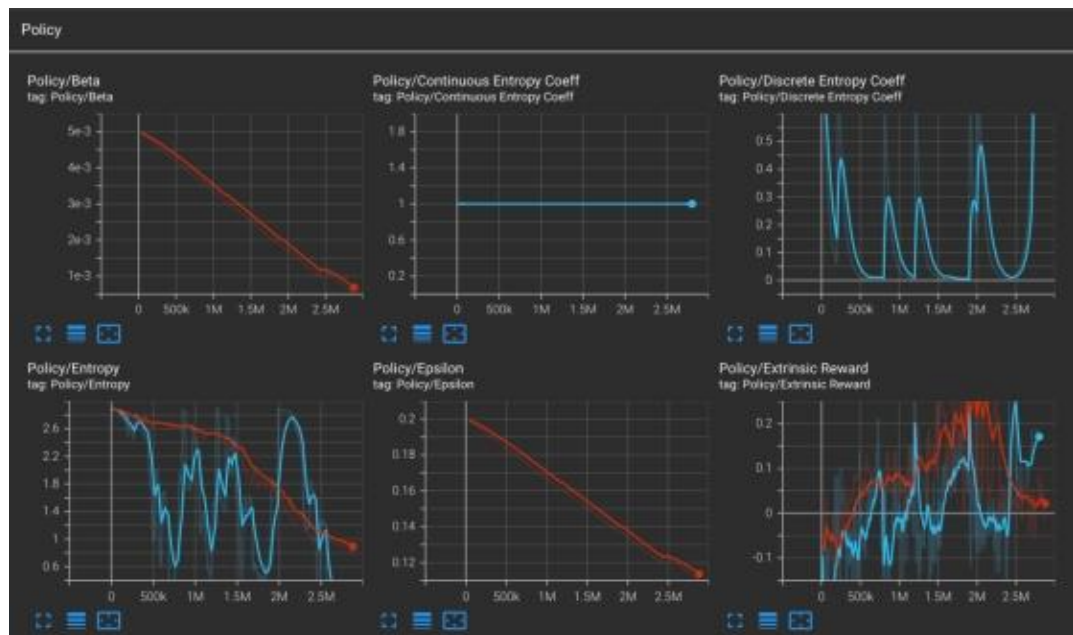


Рисунок 7 – Графики «Beta», «Entropy», «Epsilon», «Extrinsic Reward», «Contnius Entropy Coeff», «Discrete Entropy Coeff»

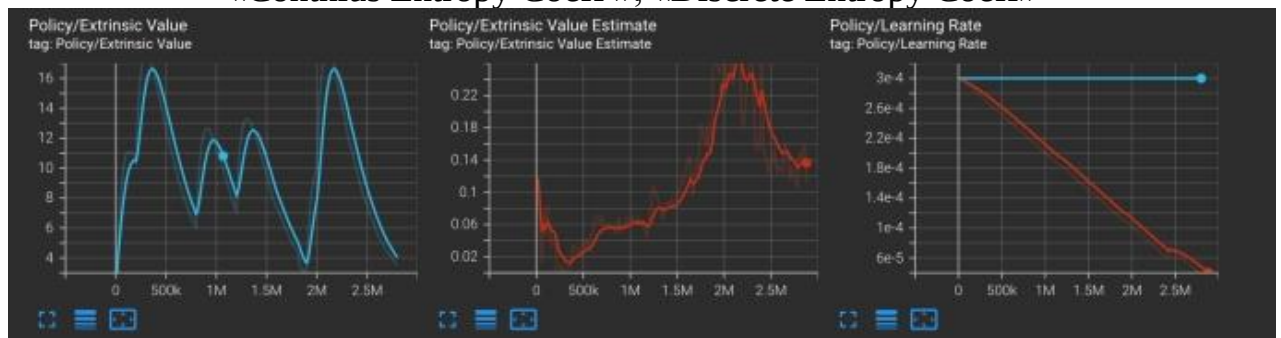


Рисунок 8 – Графики «Extrimic Value», «Extrimic Value Estimate», «Learning Rate»



Рисунок 9 – График «isTraining»

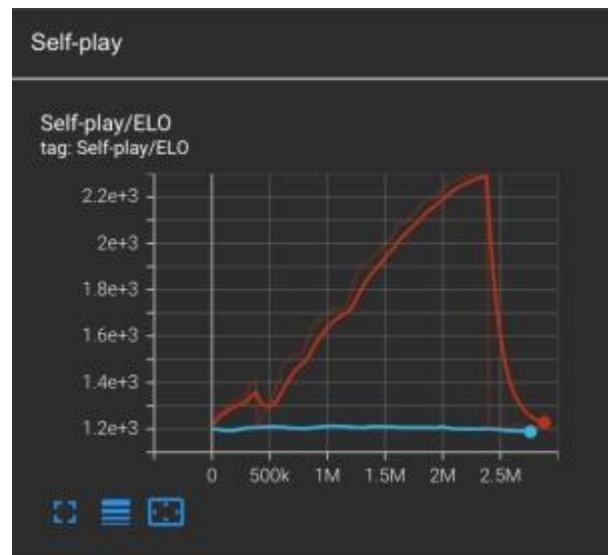


Рисунок 10 – График «ELO»

8 Реализация алгоритма AStar

На рисунках 11 и 12 представлены UML диаграммы классов для модуля поиска путей при помощи алгоритмы AStar и класс модификаторы для улучшения поиска алгоритма.

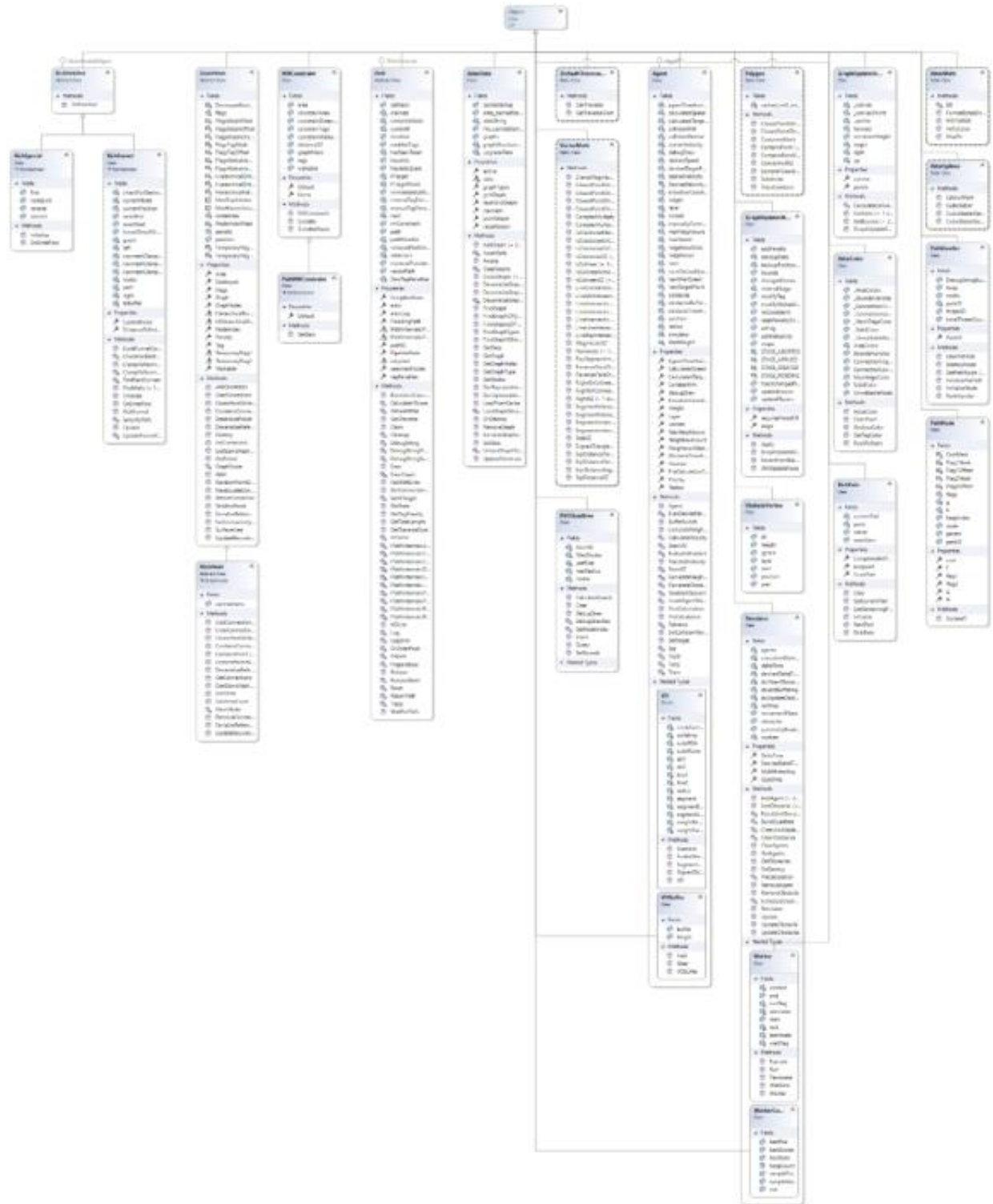


Рисунок 11 - UML диаграмма классов наследуемых от общего класса object модуля поиска пути



Рисунок 12 - UML диаграмма классов наследуемых от общего интерфейса
VersionedMonoBehaviour модуля поиска пути

Реализовано несколько видов сценариев движения. Это сценарии, которые фактически перемещают какой-либо объект на сцене по траектории, обычно это какой-либо персонаж. Основная роль сценариев движения заключается в поиске путей и следовании по ним.

Существует три ключевых сценария перемещения, которые могут использоваться для создания реалистичного поведения персонажей: AIPath, RichAI, и AILerp. Каждый из этих сценариев обладает уникальными характеристиками и предназначен для выполнения определенных задач, в зависимости от требований к игре и используемой среде.

Сценарий AIPath представляет собой мощный и универсальный инструмент для создания движения персонажей, который эффективно работает с различными типами графиков. Этот сценарий идеально подходит для создания плавного следования по заданным траекториям и эффективного реагирования на физические взаимодействия в игровой среде. Благодаря способности корректно работать с локальным избеганием препятствий, AIPath позволяет персонажам эффективно направлять в сложных условиях, обеспечивая при этом высокую степень реалистичности и естественности движений.

RichAI является специализированным сценарием, разработанным исключительно для использования с графиками navmesh и recast. Он не совместим с другими типами графиков, но предлагает значительные преимущества при работе с navmesh, включая улучшенное управление траекториями и более точное следование заданному пути. RichAI обладает продвинутыми возможностями по обработке ссылок вне сетки, что делает его идеальным выбором для проектов, требующих высокой степени точности и плавности движения в трёхмерных игровых мирах.

Сценарий AILerp использует метод линейной интерполяции для перемещения персонажей вдоль заданного пути. Преимуществом AILerp является его способности точно следовать за заданным маршрутом без каких-либо отклонений, что делает его идеальным вариантом в ситуации при необходимости максимальной точности движения. Так как AILerp сценарий игнорирует физику и не поддерживает локальное избегание препятствий, его использование ограничено сценариями, где эти аспекты не играют роли. Этот сценарий является самым быстрым среди всех трех, благодаря своей простоте, но следует тщательно рассмотреть его применимость, если в игровом приложении требуется имитация физических взаимодействий.

Если используется график на основе navmesh, тогда нужно использовать скрипт RichAI, в противном случае либо AIPath, либо AILerp, в зависимости от того, какой стиль движения нужен игре. Все сценарии

движения реализуют интерфейс поиска Pathfinding.IAstarAI. Выбор между сценариями AIPath, RichAI, и AILerp зависит от конкретных требований проекта и желаемого уровня реализма и точности движения [11].

8.1 Модификаторы

Модификаторы - небольшие скрипты, которые могут подключаться к искателю и предварительно постобрабатывать путь, прежде чем он будет возвращен вызывающей стороне.

Модификаторы используются путем простого прикрепления скриптов к тому же игровому объекту, к которому прикреплен компонент Seeker. Они будут работать автоматически без каких-либо изменений кода.

Это может быть, например, модификатор Raycast, который удаляет ненужные узлы в пути, чтобы сделать его короче, но все еще действительным. Или модификатор Smooth, который разделяет путь и сглаживает его, чтобы сделать его более эстетически привлекательным.

8.1.1 Smooth модификатор

Модификатор smooth представляет собой инструмент, предназначенный для оптимизации и сглаживания траектории движения, делая её более естественной и плавной для восприятия. Достижение этого эффекта возможно двумя основными способами: первый заключается в ручной коррекции маршрута путём деления его на более мелкие сегменты и последующем перемещении ключевых точек ближе друг к другу, тем самым уменьшая резкость переходов между ними. Второй способ включает в себя применение математических кривых, таких как сплайны Безье, которые автоматически создают гладкий и непрерывный путь, идеально подходящий для анимаций или движения объектов в виртуальном пространстве.

8.1.2 Модификатор Funnel

Модификатор Funnel представляет собой высокоэффективный инструмент, разработанный специально для упрощения и оптимизации путей в навигационных системах, основанных на сеточных структурах или графиках. Этот модификатор служит быстрым и точным решением для

улучшения маршрутов движения, делая их более логичными и понятными для пользователя.

Основой работы модификатора Funnel является алгоритм Funnel, который способен анализировать заданный коридор движения, состоящий из узлов пути. Каждый из этих узлов характеризуется определённой площадью, часто имеющей форму, аналогичную треугольникам. Используя эту информацию, алгоритм Funnel способен вычислить наиболее эффективный и кратчайший маршрут внутри заданного коридора. Таким образом, внедрение данного модификатора в системы навигации позволяет значительно улучшить визуальное восприятие путей, делая их значительно более чистыми и плавными.

Эффективность модификатора Funnel особенно заметна в сложных и плотно заполненных навигационных сетках, где необходимость в точном и быстром определении оптимального пути крайне важна. Благодаря использованию алгоритма Funnel, возможно добиться высокой точности в расчёте маршрутов, минимизируя необходимость в ручной корректировке и значительно повышая общую эффективность системы навигации. Такие улучшения не только упрощают навигацию для конечного пользователя, но и могут существенно сократить время и ресурсы, необходимые для проектирования и поддержки навигационных систем, делая процесс более экономичным и доступным, на рисунке 13 приведен наглядный пример работы модификатора Funnel.

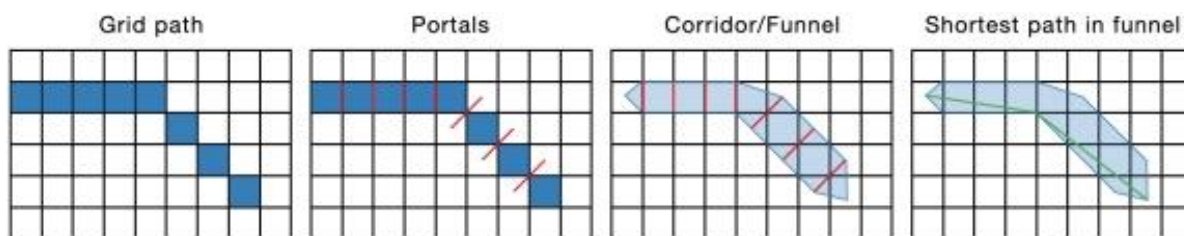


Рисунок 13 – Модификатор Funnel

Реализация алгоритма Funnel находится в одноименном классе. Он содержит в себе несколько ключевых компонентов и методов для реализации алгоритма Funnel, который имеет решающее значение для эффективного

поиска пути в игровых средах. Две структуры – это FunnelPortals, которая представляет левый и правый края пути воронки и структура PathPart, представляющая часть пути, последовательность соседних треугольников или ссылку.

Метод SplitIntoParts разбивает путь на части в зависимости от типа узлов (треугольники, узлы сетки или связи узлов).

```
Publicstatic List<PathPart>SplitIntoParts (Path path) {
    var nodes = path.path;

    var result = ListPool<PathPart>.ClAIM();

    if (nodes == null || nodes.Count == 0) {
        return result;
    }
    for (inti = 0; i<nodes.Count; i++) {
        if (nodes[i] isTriangleMeshNode || nodes[i]
isGridNodeBase) {
            var part = newPathPart();
            part.startIndex = i;
            uintcurrentGraphIndex = nodes[i].GraphIndex;
            for (; i<nodes.Count; i++) {
                if (nodes[i].GraphIndex !=
currentGraphIndex&& !(nodes[i] is NodeLink3Node)) {
                    break;
                }
            }
            i--;
            part.endIndex = i;
            if (part.startIndex == 0) {
                part.startPoint = path.vectorPath[0];
            } else {
                part.startPoint =
(Vector3)nodes[part.startIndex-1].position;
            }

            if (part.endIndex == nodes.Count-1) {
                part.endPoint =
path.vectorPath[path.vectorPath.Count-1];
            } else {
                part.endPoint =
(Vector3)nodes[part.endIndex+1].position;
            }
            result.Add(part);
        } elseif (NodeLink2.GetNodeLink(nodes[i]) != null) {
            var part = newPathPart();
            part.startIndex = i;
            varcurrentGraphIndex = nodes[i].GraphIndex;
```

```

        for (i++; i<nodes.Count; i++) {
            if (nodes[i].GraphIndex != currentGraphIndex)
            {
                break;
            }
        }
        i--;
        if (i - part.startIndex == 0) {
            continue;
        } elseif (i - part.startIndex != 1) {
            throw new System.Exception("NodeLink2 link
length greater than two (2) nodes. " + (i - part.startIndex +
1));
        }
        part.endIndex = i;
        part.isLink = true;
        part.startPoint =
(Vector3)nodes[part.startIndex].position;
        part.endPoint =
(Vector3)nodes[part.endIndex].position;
        result.Add(part);
    } else {
        throw new System.Exception("Unsupported node type or
null node");
    }
}

return result;
}

```

Метод Unwrap разворачивает порталы воронки из трехмерного пространства в двумерное, исправляя изгибы и загибы.

```

public static void Unwrap (FunnelPortals funnel, Vector2[] left,
Vector2[] right) {
    int startIndex = 1;
    var normal = Vector3.Cross(funnel.right[1] - funnel.left[0],
funnel.left[1] - funnel.left[0]);
    while (normal.sqrMagnitude <= 0.00000001f && startIndex + 1
< funnel.left.Count) {
        startIndex++;
        normal = Vector3.Cross(funnel.right[startIndex] -
funnel.left[0], funnel.left[startIndex] - funnel.left[0]);
    }

    left[0] = right[0] = Vector2.zero;

    var portalLeft = funnel.left[1];
    var portalRight = funnel.right[1];
    var prevPoint = funnel.left[0];
    Quaternion mRot = Quaternion.FromToRotation(normal,
Vector3.forward);
    Vector3 mOffset = mRot * (-funnel.right[0]);
}

```

```

        for (inti = 1; i<funnel.left.Count; i++) {
            if (UnwrapHelper(portalLeft, portalRight, prevPoint,
funnel.left[i], refmRot, refmOffset)) {
                prevPoint = portalLeft;
                portalLeft = funnel.left[i];
            }

            left[i] = mRot * funnel.left[i] + mOffset;

            if (UnwrapHelper(portalLeft, portalRight, prevPoint,
funnel.right[i], refmRot, refmOffset)) {
                prevPoint = portalRight;
                portalRight = funnel.right[i];
            }

            right[i] = mRot * funnel.right[i] + mOffset;
        }
    }
}

```

Метод `FixFunnel` пытается исправить устаревшие или недействительные воронки. Разворачивает порталы воронки из трехмерного пространства в двумерное. Результат сохраняется в массивах `left` и `right` структуры `FunnelPortals`, которые должны быть не меньше размера воронки. На выходе получится воронка в 2D-пространстве.

Метод `Calculate` - сердце скрипта реализует алгоритм воронки для поиска кратчайшего пути через порталы воронки. Рассчитывает кратчайший путь через воронку. Если опция развертывания включена, воронка разворачивается перед расчетом. Входными данными являются:

- развернутые порталы - кэш развернутых сегментов портала. Он может быть пустым, но будет заполнен развернутыми порталами, и в следующий раз, когда запустится алгоритм, он будет быстрее,
- левые порталы - Левая сторона воронки. Не должна содержать начальную точку,
- правые порталы - Правая сторона воронки. Не должна содержать конечную точку,
- начальная точка - начальная точка воронки. Отсюда агент переместится в лучшую точку между левыми порталами и правыми порталами,

- конечная точка - Конечная точка воронки,
- maxCorners - Будут рассчитаны первые N углов

оптимизированного пути.

Вычисление меньшего количества углов происходит быстрее,

– ось проекции - Если установлено значение, отличное от (0,0,0), то все порталы будут проецироваться на плоскость с этой нормалью,

– последний поворот – оповещает метод, если достигнут последний угол пути. Если значение true, то возвращаемое значение гарантированно будет не более maxCorners - 1 (если только maxCorners = 0).

```
Public static List<Vector3> Calculate (FunnelPortals funnel,
bool unwrap, boolsplitAtEveryPortal) {
    if (funnel.left.Count != funnel.right.Count)
        throw new System.ArgumentException("funnel.left.Count !=
funnel.right.Count");
    var leftArr = ArrayPool<Vector2>.ClAIM(funnel.left.Count);
    var rightArr = ArrayPool<Vector2>.ClAIM(funnel.left.Count);

    if (unwrap) {
        Unwrap(funnel, leftArr, rightArr);
    } else {
        for (int i = 0; i < funnel.left.Count; i++) {
            leftArr[i] = ToXZ(funnel.left[i]);
            rightArr[i] = ToXZ(funnel.right[i]);
        }
    }

    int startIndex = FixFunnel(leftArr, rightArr,
funnel.left.Count);
    var intermediateResult = ListPool<int>.ClAIM();
    if (startIndex == -1) {
        intermediateResult.Add(0);
        intermediateResult.Add(funnel.left.Count - 1);
    } else {
        bool lastCorner;
        Calculate(leftArr, rightArr, funnel.left.Count,
startIndex, intermediateResult, int.MaxValue, out lastCorner);
    }
    var result =
ListPool<Vector3>.ClAIM(intermediateResult.Count);

    Vector2 prev2D = leftArr[0];
    var prevIdx = 0;
    for (int i = 0; i < intermediateResult.Count; i++) {
        var idx = intermediateResult[i];

        if (splitAtEveryPortal) {
```

```

        var next2D = idx >= 0 ? leftArr[idx] : rightArr[-
idx];
        for (int j = prevIdx + 1; j < System.Math.Abs(idx);
j++) {

            if (!VectorMath.LineLineIntersectionFactor(leftArr[j],
rightArr[j] - leftArr[j], prev2D, next2D - prev2D, out float
factor)) {
                factor = 0.5f;
            }
            result.Add(Vector3.Lerp(funnel.left[j],
funnel.right[j], factor));
        }

        prevIdx = Mathf.Abs(idx);
        prev2D = next2D;
    }

    if (idx >= 0) {
        result.Add(funnel.left[idx]);
    } else {
        result.Add(funnel.right[-idx]);
    }
}
ListPool<int>.Release(ref intermediateResult);
ArrayPool<Vector2>.Release(ref leftArr);
ArrayPool<Vector2>.Release(ref rightArr);
return result;
}

```

Для применения модификатора слудует вызвать метод Apply, которая описана в интерфейсе MonoModifier.

```

Public override void Apply (Path p) {
    if (p.path == null || p.path.Count == 0 || p.vectorPath
== null || p.vectorPath.Count == 0) {
        return;
    }

    List<Vector3> funnelPath = ListPool<Vector3>.ClAIM();
    var parts = Funnel.SplitIntoParts(p);

    if (parts.Count == 0) {

        return;
    }

    for (inti = 0; i < parts.Count; i++) {
        var part = parts[i];
        if (!part.isLink) {
            var portals =
Funnel.ConstructFunnelPortals(p.path, part);

```

```

        var result = Funnel.Calculate(portals, unwrap,
splitAtEveryPortal);
        funnelPath.AddRange(result);
        ListPool<Vector3>.Release(ref portals.left);
        ListPool<Vector3>.Release(ref portals.right);
        ListPool<Vector3>.Release(ref result);
    } else {
        if (i == 0 || parts[i-1].isLink) {
            funnelPath.Add(part.startPoint);
        }
        if (i == parts.Count - 1 || parts[i+1].isLink) {
            funnelPath.Add(part.endPoint);
        }
    }
}
UnityEngine.Assertions.Assert.IsTrue(funnelPath.Count >= 1);
ListPool<Funnel.PathPart>.Release(ref parts);
ListPool<Vector3>.Release(ref p.vectorPath);
p.vectorPath = funnelPath;
}

```

8.1.3 Модификатор Raycast

Модификатор `raycast` упрощает пути, выполняя серию проверок по прямой линии. Этот модификатор в первую очередь полезен для грид-графиков, хотя иногда может быть полезен и при использовании точечного графика. Однако обратите внимание, что точечные графики не имеют встроенного `raycasting`, поэтому вам нужно использовать физическое `raycasting` для этого графика.

Для сеточных графиков рекомендуется рассмотреть возможность комбинированного использования модификатора `raycast` в сочетании с модификатором `Funnel`. Для этого достаточно прикрепить оба модификатора к объекту `GameObject` с помощью инструментов поиска путей. Такой подход может привести к значительному улучшению результатов, хотя и не гарантирует это во всех случаях. Как правило, при совместном применении этих модификаторов путь будет следовать границам графика более точно. Однако это может привести к пропуску некоторых возможностей для упрощения пути. В процессе совместного использования сначала активируется модификатор `Funnel`, который осуществляет первоначальное

упрощение пути, а затем модификатор raycast обрабатывает полученные данные, пытаясь добиться их дальнейшего упрощения.

RaycastModifier класс наследует MonoModifier и реализует функциональные возможности изменения путей. Apply метод вызывается для применения модификатора raycast к заданному пути. Сначала он проверяет, включено ли рейкастинг, и если да, то приступает к изменению пути. Он устанавливает фильтр на основе ограничений пути и перебирает точки пути, чтобы упростить их, используя либо жадный, либо динамический подход программирования, основанный на настройке качества.

```
public override void Apply (Path p) {
    if (!useRaycasting && !useGraphRaycasting) return;

    var points = p.vectorPath;
    cachedFilter.path = p;
    cachedNNConstrAInt.graphMask = p.nnConstrAInt.graphMask;

    if (ValidateLine(null, null, p.vectorPath[0],
        p.vectorPath[p.vectorPath.Count-1], cachedFilter.cachedDelegate,
        cachedNNConstrAInt)) {
        var s = p.vectorPath[0];
        var e = p.vectorPath[p.vectorPath.Count-1];
        points.ClearFast();
        points.Add(s);
        points.Add(e);
    } else {
        int iterations = iterationsByQuality[(int)quality];
        for (int it = 0; it < iterations; it++) {
            if (it != 0) {
                Polygon.Subdivide(points, buffer, 3);
                Memory.Swap(ref buffer, ref points);
                buffer.ClearFast();
                points.Reverse();
            }

            points = quality >= Quality.High ? ApplyDP(p,
                points, cachedFilter.cachedDelegate, cachedNNConstrAInt) :
                ApplyGreedy(p, points, cachedFilter.cachedDelegate,
                cachedNNConstrAInt);
        }
        if ((iterations % 2) == 0) points.Reverse();
    }

    p.vectorPath = points;
}
```

ApplyGreedy метод применяет упрощение пути с использованием жадного алгоритма. Он перебирает точки пути, начиная с самого начала. Для каждой точки он пытается найти самую дальнюю видимую точку с помощью двоичного поиска, тем самым уменьшая количество промежуточных точек.

```
List<Vector3>ApplyGreedy (Path p, List<Vector3> points,
System.Func<GraphNode, bool> filter, NNConstrAIntnnConstrAInt) {
    boolcanBeOriginalNodes = points.Count == p.path.Count;
    intstartIndex = 0;

    while (startIndex<points.Count) {
        Vector3 start = points[startIndex];
        varstartNode = canBeOriginalNodes&& points[startIndex]
== (Vector3)p.path[startIndex].position ? p.path[startIndex] :
null;
        buffer.Add(start);
        intmn = 1, mx = 2;
        while (true) {
            intendIndex = startIndex + mx;
            if (endIndex>= points.Count) {
                mx = points.Count - startIndex;
                break;
            }
            Vector3 end = points[endIndex];
            varendNode = canBeOriginalNodes&& end ==
(Vector3)p.path[endIndex].position ? p.path[endIndex] : null;
            if(!ValidateLine(startNode, endNode, start, end,
filter, nnConstrAInt)) break;
            mn = mx;
            mx *= 2;
        }

        while (mn + 1 < mx) {
            int mid = (mn + mx)/2;
            intendIndex = startIndex + mid;
            Vector3 end = points[endIndex];
            varendNode = canBeOriginalNodes&& end ==
(Vector3)p.path[endIndex].position ? p.path[endIndex] : null;

            if (ValidateLine(startNode, endNode, start, end,
filter, nnConstrAInt)) {
                mn = mid;
            } else {
                mx = mid;
            }
        }
        startIndex += mn;
    }

    Memory.Swap(ref buffer, ref points);
    buffer.ClearFast();
}
```

```

return points;
}

```

ApplyDP метод применяет упрощение пути с помощью динамического программирования. Рассчитывает оптимальный упрощенный путь, используя подход динамического программирования, учитывая стоимость каждого сегмента между точками. Он поддерживает массивы для хранения затрат и родительские индексы для восстановления упрощенного пути.

```

List<Vector3> ApplyDP (Path p, List<Vector3> points,
System.Func<GraphNode, bool> filter, NNConstrAInt nnConstrAInt) {
    if (DPCosts.Length < points.Count) {
        DPCosts = new float[points.Count];
        DPParents = new int[points.Count];
    }
    for (int i = 0; i < DPParents.Length; i++) DPCosts[i] =
DPParents[i] = -1;
    bool canBeOriginalNodes = points.Count == p.path.Count;

    for (int i = 0; i < points.Count; i++) {
        float d = DPCosts[i];
        Vector3 start = points[i];
        var startIsOriginalNode = canBeOriginalNodes && start ==
(Vector3)p.path[i].position;
        for (int j = i+1; j < points.Count; j++) {
            float d2 = d + (points[j] - start).magnitude +
0.0001f;
            if (DPParents[j] == -1 || d2 < DPCosts[j]) {
                var endIsOriginalNode = canBeOriginalNodes &&
points[j] == (Vector3)p.path[j].position;
                if (j == i+1 ||
ValidateLine(startIsOriginalNode ? p.path[i] : null,
endIsOriginalNode ? p.path[j] : null, start, points[j], filter,
nnConstrAInt)) {
                    DPCosts[j] = d2;
                    DPParents[j] = i;
                } else {
                    break;
                }
            }
        }
    }
}

```

ValidateLine метод проверяет, допустим ли прямой путь между двумя точками, учитывая препятствия и топологию графа. Он выполняет рейкастинг для обнаружения препятствий вдоль линии между точками. Кроме того, он использует рейкастинг графа для проверки наличия препятствий в графе навигации.

```

        protected bool ValidateLine (GraphNode n1, GraphNode n2,
        Vector3 v1, Vector3 v2, System.Func<GraphNode, bool> filter,
        NNConstrAInt nnConstrAInt) {
            if (useRaycasting) {
                if (use2DPhysics) {
                    if (thickRaycast && thickRaycastRadius > 0
&& Physics2D.CircleCast(v1 + raycastOffset, thickRaycastRadius,
v2 - v1, (v2 - v1).magnitude, mask)) {
                        return false;
                    }

                    if (Physics2D.Linecast(v1 + raycastOffset,
v2 + raycastOffset, mask)) {
                        return false;
                    }
                } else {
                    if (Physics.Linecast(v1 + raycastOffset,
v2 + raycastOffset, mask)) {
                        return false;
                    }
                    if (thickRaycast && thickRaycastRadius > 0)
{
                        if (Physics.CheckSphere(v1 +
raycastOffset + (v2 - v1).normalized * thickRaycastRadius,
thickRaycastRadius, mask)) {
                            return false;
                        }
                        if
(Physics.SphereCast(new Ray(v1 + raycastOffset, v2 - v1),
thickRaycastRadius, (v2 - v1).magnitude, mask)) {
                            return false;
                        }
                    }
                }
            }

            if (useGraphRaycasting) {
                bool betweenNodeCenters = n1 != null && n2 !=
null;

                if (n1 == null) n1 =
AstarPath.active.GetNearest(v1, nnConstrAInt).node;
                if (n2 == null) n2 =
AstarPath.active.GetNearest(v2, nnConstrAInt).node;

                if (n1 != null && n2 != null) {
                    NavGraph graph = n1.Graph;
                    NavGraph graph2 = n2.Graph;

                    if (graph != graph2) {
                        return false;
                    }
                }
            }
        }
    }
}

```

```

        varrayGraph = graph asIRaycastableGraph;
        GridGraph gg = graph asGridGraph;
        if (betweenNodeCenters&&gg != null) {
            return !gg.Linecast(n1
asGridNodeBase, n2 asGridNodeBase, filter);
        } else
        if (rayGraph != null) {
            return !rayGraph.Linecast(v1, v2,
outGraphHitInfo_, null, filter);
        }
    }
    }
    return true;
}

```

8.2 Генерация графиков

График представляет области мира, по которым можно ходить. Это набор узлов и связей между ними. Разные графики имеют очень разное представление о мире, от точек до плиток сетки и треугольной сетки. Это означает, что разные графики лучше подходят для разных типов игр. Все же существуют несколько графиков:

8.2.1 Recast график

Процесс построения графика можно описать следующим способом:

- воксельная форма создается из исходной треугольной сетки путем растеризации треугольников в многослойное поле высот. Затем к форме применяются несколько простых фильтров, чтобы исключить места, где персонаж не смог бы передвигаться.
- проходимые области, описываемые формой, разделены на простые наложенные друг на друга двумерные области. Результирующие области имеют только один непересекающийся контур, что значительно упрощает заключительный этап процесса.
- навигационные полигоны отделяются от областей путем сначала обводки границ, а затем их упрощения. Полученные многоугольники в конечном итоге преобразуются в треугольники, что делает их идеальными для поиска путей и пространственных рассуждений об уровне.

Процесс генерации пересчета обычно работает непосредственно с видимой геометрией в мире. Обычно это хорошо, потому что геометрия мира обычно более детализирована, чем у коллайдеров.

Однако можно указать, что вместо этого коллайдеры должны быть растеризованы. Если очень подробная геометрия мира, это может ускорить сканирование и обновление графика [12].

Функционал графа реализуется в классе `RecastGraph`. Одной из основных переменных в классе является `relevantGraphSurfaceMode`. Она указывает, что область навигационной сетки, внутри которой он находится, должна быть включена в навигационную сетку.

Если для этого параметра установлено значение `OnlyForCompletelyInsideTile`, область навигационной сетки включается в навигационную сетку, если внутри нее имеется `RelevantGraphSurface` или если она примыкает к границе плитки. Это может оставить некоторые небольшие области, которые не нужно включать, поскольку они примыкают к границам тайла, но это устраняет необходимость размещать компонент в каждом отдельном тайле, что может быть утомительно.

Если для этого параметра установлено значение `RequireForAll`, область навигационной сетки включается только в том случае, если внутри нее имеется `RelevantGraphSurface`. Обратите внимание: хотя навигационная сетка между плитками выглядит непрерывной, плитки вычисляются индивидуально, поэтому вам нужен компонент `RelevantGraphSurface` для каждого региона и для каждой плитки.

На рисунке 14 использовался режим `OnlyForCompletelyInsideTile`. Границы плитки выделены черным цветом. Нужно обратить внимание, поскольку все регионы примыкают к границе плитки, этот режим в данном случае ничего не удаляет и дает тот же результат, что и `DoNotRequire`.

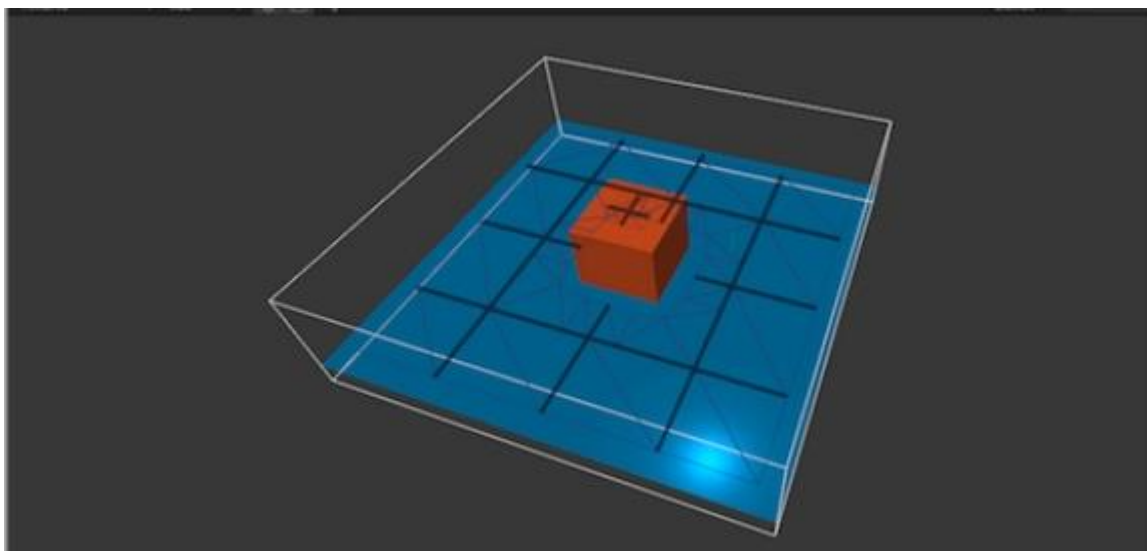


Рисунок 14 – Режим OnlyForCompletelyInsideTile

На рисунке 15 используется режим RequireForAll. Плитка не используется. Небольшая область в верхней части оранжевого куба теперь исчезла, поскольку она не находилась в той же области, что и соответствующий компонент поверхности графа. Результат был бы идентичен методу OnlyForCompletelyInsideTile, поскольку плиток нет.

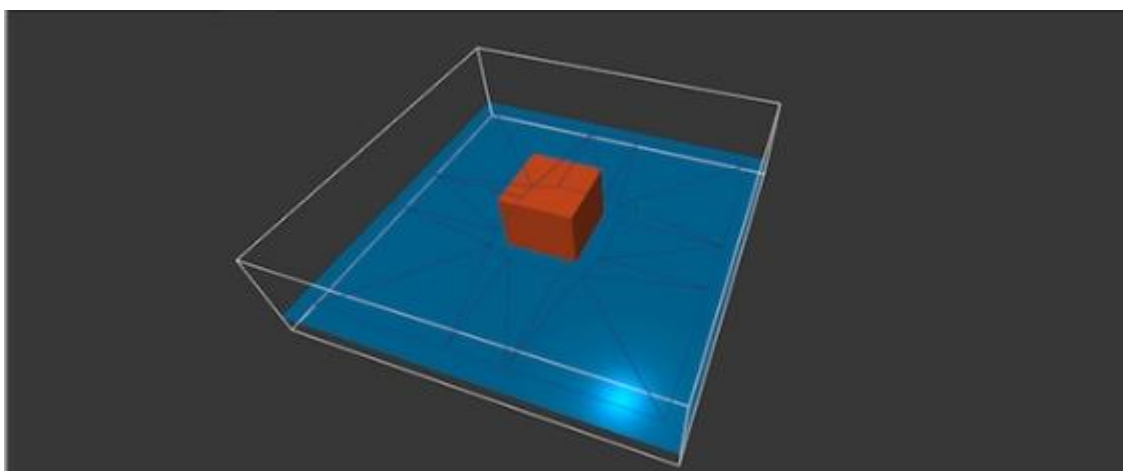


Рисунок 15 – Режим RequireForAll

На рисунке 16 используется режим RequireForAll. Поскольку существует только один компонент RelevantGraphSurface, будет включен только тот регион, в котором он находился, в плитке, в которой он расположен. Если бы в других плитках было несколько RelevantGraphSurface, эти регионы также можно было бы включить.

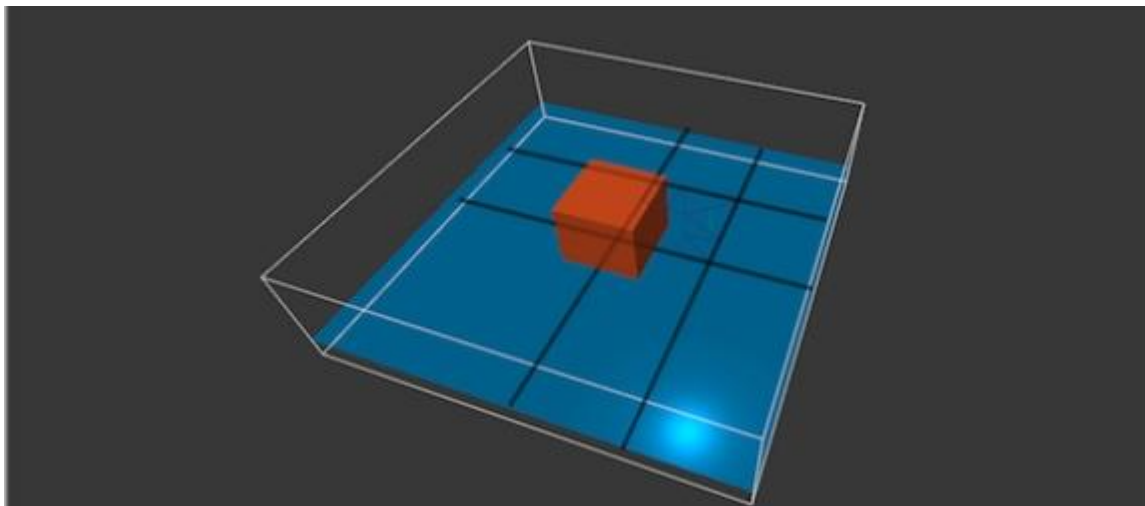


Рисунок 16 – Режим RequireForAll

На рисунке 17 вместе с OnlyForCompletelyInsideTile используется другой размер плитки. Область поверх оранжевого куба теперь исчезла, поскольку границы области не пересекают эту область и внутри нее нет компонента RelevantGraphSurface.

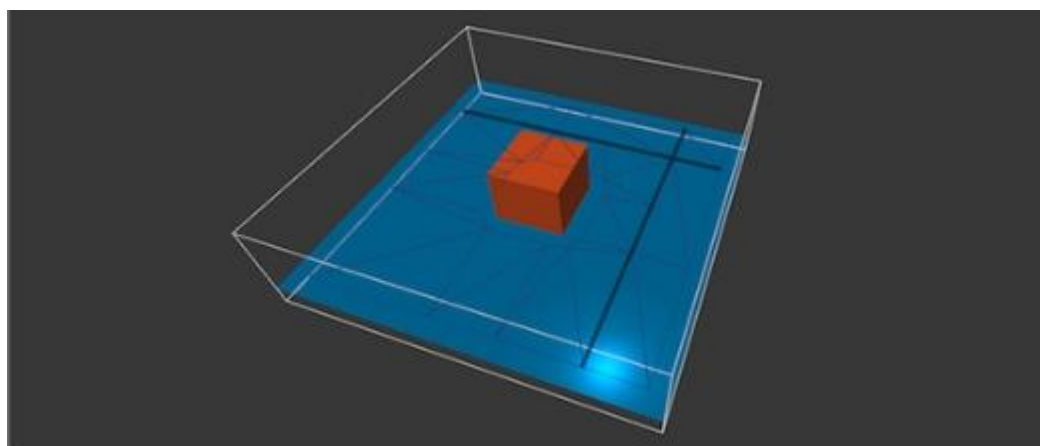


Рисунок 17 – Режим OnlyForCompletelyInsideTile

Ключевым методом класса RecastGraph - является ReplaceTiles, который принимает 2 параметра:

- 1) tileMeshes - Плитки для загрузки. Они будут загружены в график по координатам тайла TileMeshes.tileRect.
- 2) yOffset - Все вершины загруженных тайлов будут перемещены вверх (или вниз, если значение отрицательное) на эту величину

Загрузите плитки из объекта TileMeshes(представляет прямоугольную группу плиток графа) в этот граф. Это можно использовать для многих целей, например, для потоковой передачи мира или размещения больших

предварительно отсканированных префабов. Загруженные плитки должны иметь тот же размер мирового пространства, что и плитки этого графа. Размер мирового пространства для переработанного графа определяется значением `cellSize`, умноженным на `tileSizeX` или `tileSizeZ`.

Если граф не сканируется при вызове этого метода, граф будет инициализирован и будет состоять только из плиток, загруженных этим вызовом. Код ниже приводит пример вызова этого метода

```
Vargraph=AstarPath.active.data.recastGraph;  
varbuildSettings=RecastBuilder.BuildTileMeshes(graph,newTileLayout(graph),newIntRect(0,0,5,5));  
vardisposeArena=newPathfinding.Jobs.DisposeArena();  
var promise =buildSettings.Schedule(disposeArena);  
AstarPath.active.AddWorkItem(()=>  
var result =promise.Complete();  
TileMeshestiles=result.tileMeshes.ToManaged();  
tiles.tileRect=tiles.tileRect.Offset(newInt2(2,0));  
tiles.Rotate(1);  
graph.ReplaceTiles(tiles);  
disposeArena.DisposeAll();  
result.Dispose();  
});
```

Метод `SnapForceBoundsToScene` изменяет границы графа, чтобы точно инкапсулировать все объекты сцены, которые могут быть включены в процесс сканирования в зависимости от настроек.

Какие объекты используются, зависит от настроек. Если объект повлиял бы на график с текущими настройками, если бы он находился внутри границ графа, он будет обнаружен, и границы будут расширены, чтобы содержать этот объект

Метод `TranslateInDirection` принимает два параметра – это количество плиток вдоль оси X графика, на которые необходимо перемещаться и количество плиток вдоль оси Z графика, на которые необходимо перемещаться.

Перемещает граф пересчета на количество плиток, отбрасывая старые плитки и сканируя новые. Поддерживается только перевод в одном направлении. $dx == 0$ или $dz == 0$ должно сохраняться. Если вам нужно переместить график по диагонали, то вы можете вызвать эту функцию дважды, по одному разу для каждой оси. Это используется компонентом ProceduralGraphMover для эффективного перемещения графика. Все плитки, которые могут оставаться в одном положении, останутся. Те, которые упали бы за край графика, будут отброшены, а новые плитки будут созданы и отсканированы на другой стороне графика.

Класс ProceduralGraphMover реализует перемещение сетки или изменение графика, чтобы он следовал за целью. Это полезно, когда очень большой или даже бесконечный мир, но поиск пути необходим только в небольшой области вокруг объекта, например, игрока. Этот компонент будет перемещать график так, чтобы его центр оставался ближе к целевому объекту. На рисунках 15 и 16 приведен пример того, как работает этот класс. Он перестраивает граф передвижения в зависимости от местоположения игрока, к которому он привязан.

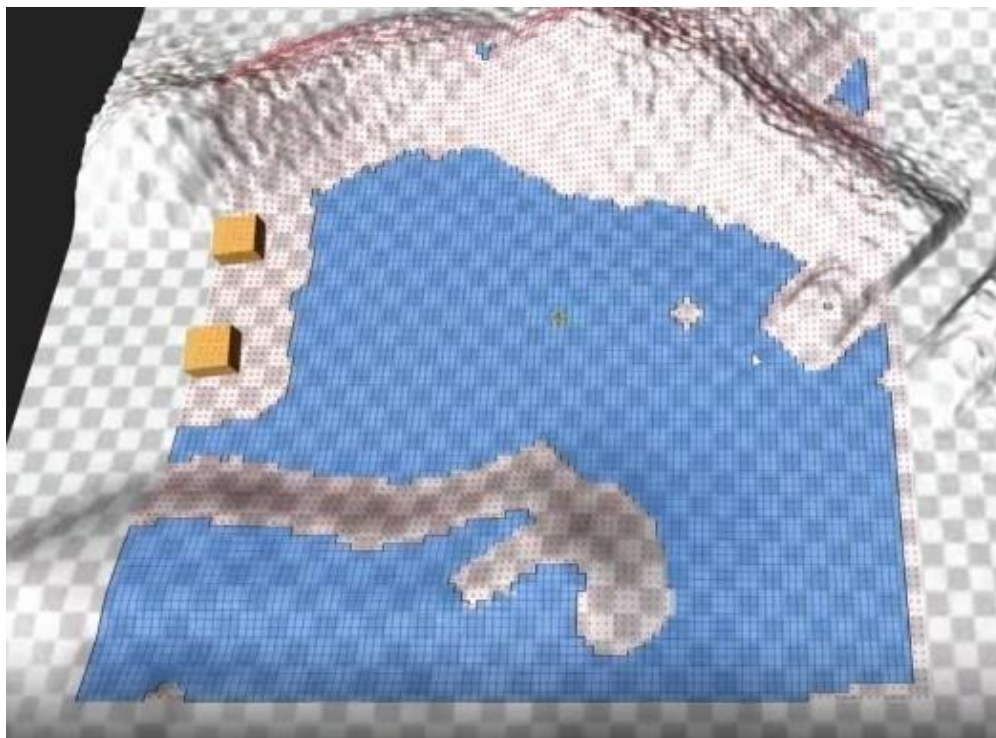


Рисунок 18 – Пример работы ProceduralGraphMover состояние 1

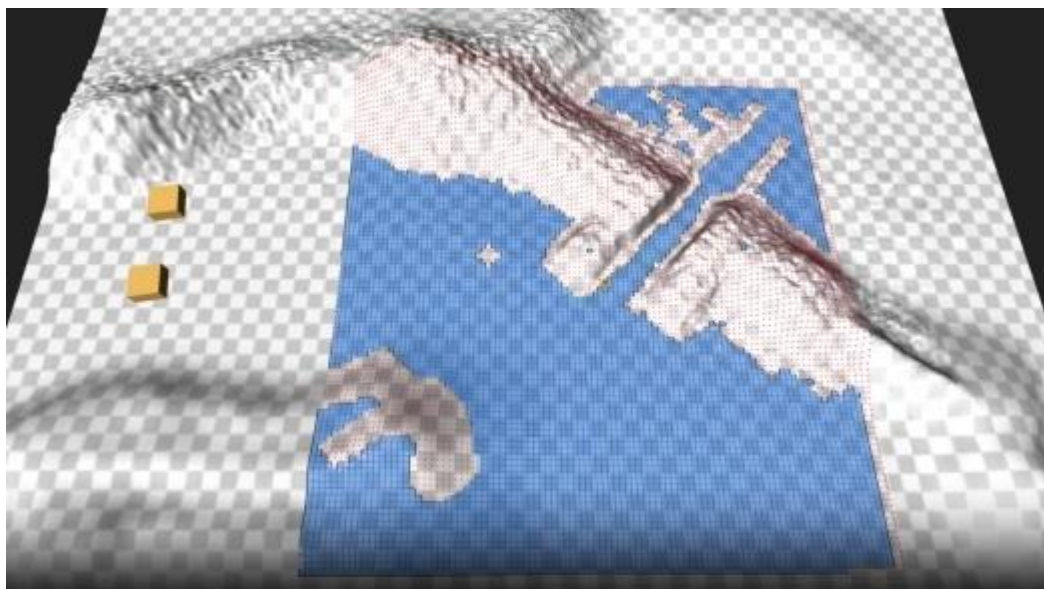


Рисунок 19 – Пример работы ProceduralGraphMover состояние 2

8.2.2 График Navmesh

Система может автоматически генерировать навигационные сетки, но с этим графиком вам придется создавать их самостоятельно в вашем любимом приложении для 3D-моделирования. Навигационная сетка - это сетка, в которой полигоны описывают область, по которой можно пройти. Вершины всегда, за исключением некоторых особых случаев, должны располагаться по краям сетки, а не в ее середине, т. е. вершина с окружающими ее полигонами. Также может быть полезно разделить очень длинные ребра, потому что полигоны одинакового размера дают лучшие пути, чем действительно большие и действительно маленькие полигоны рядом друг с другом.

Графики Navmesh иногда могут давать неоптимальные пути. Это связано с тем, что поиск путей выполняется в центрах узлов (треугольников). В основном это происходит, когда navmesh имеет очень большие и очень маленькие узлы (треугольники), расположенные близко друг к другу. В этом случае поиск пути обычно предпочитает путь с треугольниками меньшего размера (иногда с треугольниками большего размера, но не такой распространенный).

Реализация графика расположена в классе NavMeshGraph и наследуется от 2-х базовых классов NavmeshBase и IUpdatableGraph.

Важной логической переменной в этом классе является `recalculateNormals`, которая определяет способ вычисления нормалей. Отключен для сферических графиков или других сложных поверхностей, которые позволяют агентам, например, ходить по стенам или потолкам. По умолчанию нормали сетки будут перевернуты таким образом, чтобы они были максимально направлены вверх. Нормали важны при соединении смежных узлов. Два смежных узла будут соединены только в том случае, если они ориентированы одинаково. Это особенно важно, если у есть навигационная сетка на стенах или даже на потолке комнаты. Или если попытаться создать сферическую навигационную сетку. Если выполнитб одно из этих действий, следует отключить этот параметр и убедиться, что нормали в исходной сетке установлены правильно. Если установлено значение `true`, нормали сетки будут перевернуты, чтобы указывать максимально вверх, что полезно в большинстве случаев. Если установлено значение `false`, нормали не изменяются, что может быть полезно для сложных поверхностей, таких как стены или потолки.

В классе содержатся много методов, но осным является `ScanInternal` метод выполняет процесс внутреннего сканирования для создания графа навигационной сетки. Он дает `Progress` объекты, указывающие ход процесса сканирования. Он вычисляет преобразование, сжимает вершины, строит узлы и заменяет плитки по мере необходимости для создания навигационной сетки

Второй метод по важности для работы графа – это `UpdateArea`. Он обновляет определенную область графа навигационной сетки на основе предоставленного файла `GraphUpdateObject`. В качестве параметров он принимает объект обновления и граф, на чем основанна навигационная сетка. Вычисляет ограничивающий прямоугольник области обновления, проверяет, какие узлы пересекают эту область, и соответствующим образом обновляет их.

8.2.3 Точечный график

Точечный график сканируется путем выполнения корневого преобразования и обработки каждого его дочернего элемента как узла. Затем он проверяет возможные соединения между узлами с помощью `raycasts`, чтобы увидеть, должны ли они быть связаны друг с другом. Получение хороших и плавных путей из точечного графика может быть затруднено, поскольку они определяют только точку, по которой можно пройти, а не область, как в других типах графиков. Однако модификатор `raycast` выполняет достойную работу.

Распространенная проблема с точечными графами заключается в том, что получение "правильного" ближайшего узла может быть затруднено без добавления большого количества узлов. Например, если агент перемещается близко к стене, ближайший узел может находиться по другую сторону стены. Можно смягчить это, убедившись, что узлы не размещаются слишком редко. Использование этого типа графиков рекомендуется, если только другие типы графиков не подходят для вашей игры.

Основная логика данного графика описана в классе `PointGraph` содержащий в себе 3 основных переменных и 5 методов реализации.

1) переменная `PointNode`. Это внутренний класс, представляющий узел точечного графа. Содержит такие поля, как `walkable`, и `connections`.

2) `transform` - переменная представляет корневое преобразование графа. Это родительский элемент всех узлов графа.

3) логическое значение `optimizeForSparseGraph` указывающее, следует ли оптимизировать граф для разреженной структуры графа. Разреженные графы более эффективно используют память, но имеют более медленное время поиска узлов. Это может значительно сократить время вычислений как для сканирования, так и для обычных запросов путей. Это сокращает количество проверок между узлами, которые необходимо выполнить во время сканирования, а также может оптимизировать получение

ближайшего узла из графика, например, при запросе пути. Выигрыш от использования этой оптимизации увеличивается при использовании графиков большего размера, алгоритм сканирования по умолчанию используется методом перебора и требует $O(n^2)$ проверок, эта оптимизация вместе с графиком, подходящим для этого, требует только $O(n)$ проверок во время сканирования, при условии, что ограничения расстояния соединения являются разумными.

4) `addNode` метод добавляет узел на график в указанной позиции. Принимает на вход позицию, в которую будет добавлен узел и возвращает новый добавленный узел. Также он позволяет указать тип добавляемого узла. В таком случае узел, который нужно добавить, должен быть создан с помощью `newT(AstarPath.active)` прямо перед вызовом этого метода.

5) `addChildren` метод рекурсивно добавляет дочерние элементы преобразования в качестве узлов в граф.

6) метод `RebuildNodeLookup` перестраивает структуру поиска для узлов. Этот метод используется, когда `optimizeForSparseGraph` включен. Обеспечивает правильное поведение поиска пути после ручного манипулирования узлами или при добавлении большого количества узлов с помощью `AddNode`.

7) `connectNodes` метод пересчитывает соединения для всех узлов графа. Полезно после создания узлов вручную и необходимости соединить их так же, как обычно соединяет узлы точечный граф.

8) `updateArea` (явная реализация интерфейса) обновляет область точечного графика. Пересчитывает возможно затронутые соединения.

8.3 Генератор графов

Все графики в проекте написаны как дополнения к системе, это позволяет относительно легко добавлять свои собственные специализированные типы графиков. Самый простой из графиков выглядит следующим образом:

```

        [Pathfinding.Util.Preserve]
public class SimpleGraph:NavGraph{
public override bool isScanned=>true;
class SimpleGraph ScanPromise:IGraphUpdatePromise{
public SimpleGraph graph;
public IEnumerator<JobHandle>Prepare ()=>null;
public voidApply(IGraphUpdateContextctx) {
graph.DestroyAllNodes ();
}
}
protected override IGraphUpdatePromise
ScanInternal ()=>newSimpleGraphScanPromise{ graph =this};
public override void GetNodes(System.Action<GraphNode> action) {
}
}

```

Метод `ScanInternal` вызывается, когда график должен быть отсканирован. Метод сканирования должен создавать несколько узлов и устанавливать связи между ними. Создается полярный график, то есть что-то вроде сетки, но в виде круга с кольцами вместо строк. Будут использовать "круги" для обозначения количества концентрических колец на графике, а "шаги" - как количество узлов в каждом круге. Первым шагом является создание временного массива для хранения узлов и добавление некоторых переменных для настройки графика. Будет использоваться тип узла `PointNode`, который использует `PointGraph`.

```

        publicint circles =10;
publicint steps =20;
publicVector3 center =Vector3.zero;
publicfloat scale =2;
PointNode[] nodes;
GraphTransform transform;
PointNodeCreateNode(Vector3 position) {
var node =newPointNode(active);
node.position=(Int3)position;
return node;
}

```

```

}
public override IEnumerable<Progress> ScanInternal () {
    PointNode [][] circleNodes = new PointNode [circles] [];
    yield break;
}

```

Каждый фрагмент кода ниже этого пункта должен перейти в функцию сканирования. Будет использоваться матрица для размещения, чтобы включить вращение, позиционирование и так далее узлов.

Первым узлом в массиве должен быть центральный узел, который размещен в `Vector3.zero`. Позиции узлов хранятся как значения `Int3`. Они похожи на значения `Vector3`, но используют целочисленные координаты вместо координат с плавающей запятой.

```

graph.DestroyAllNodes ();
var circles = graph.circles;
var steps = graph.steps;
PointNode [][] circleNodes = new PointNode [circles] [];
graph.transform = new GraphTransform (Matrix4x4.TRS (graph.center, Quaternion.identity, Vector3.one * graph.scale));
circleNodes [0] = new PointNode [] {
    graph.CreateNode (CalculateNodePosition (0, 0, graph.transform))
};

```

Далее есть узел под заданным углом и в заданной окружности. Нужно вычислить положение для этого узла.

```

static Vector3 CalculateNodePosition (int circle, float angle, GraphTransform transform) {
    var pos = new Vector3 (Mathf.Sin (angle), 0, Mathf.Cos (angle));
    pos *= circle;
    pos = transform.Transform (pos);
    return pos;
}

```

```

float anglesPerStep = (2 * Mathf.PI) / steps;
for (int circle = 1; circle < circles; circle++) {
    circleNodes [circle] = new PointNode [steps];
    for (int step = 0; step < steps; step++) {

```

```
float angle = step *anglesPerStep;
```

```
Vector3 pos =CalculateNodePosition(circle,  
angle,graph.transform);  
circleNodes[circle][step]=graph.CreateNode(pos);  
}  
}
```

Теперь нужно установить все узлы в их правильные положения и добавить соединения между ними, так как сейчас у них нет их:

```
for(int circle =1; circle < circles; circle++){  
for(int step =0; step < steps; step++){  
PointNode node =circleNodes[circle][step];  
intnumConnections= circle < circles-1?4:3;  
var connections =newConnection[numConnections];  
connections[0].node =circleNodes[circle][(step+1)% steps];  
connections[1].node =circleNodes[circle][(step-1+steps)% steps];  
if(circle >1){  
connections[2].node =circleNodes[circle-1][step];  
}else{  
connections[2].node =circleNodes[circle-1][0];  
}  
if(numConnections==4){  
connections[3].node =circleNodes[circle+1][step];  
}  
for(int q =0; q <connections.Length; q++){  
connections[q].cost  
    (uint)(node.position-  
connections[q].node.position).costMagnitude;  
}  
node.connections= connections;  
}  
}
```

Если разместить все предыдущие фрагменты друг за другом в функции сканирования, как описано ранее, то получится работающая функция

сканирования. Наконец, чтобы иметь возможность сканировать график, необходимо реализовать метод `GetNodes`. Поскольку все узлы просто хранятся в массиве, это очень просто.

```
public override void GetNodes(System.Action<GraphNode>
action) {
    if(nodes ==null) return;
    for(int i=0;i<nodes.Length;i++){
        action(nodes[i]);
    }
}
```

Другие методы могут быть переопределены для настройки других функциональных возможностей. Особенно методы `GetNearest` и `GetNearestForce`. Которые управляют поиском ближайшего узла к точке. Существует реализация по умолчанию, но она будет выполнять поиск по всем узлам графика каждый раз, что может быть немного медленным, если ваш график содержит много узлов. Также возможно переопределить метод `OnDrawGizmos`, если нужно нарисовать график иначе, чем в реализации по умолчанию.

ЗАКЛЮЧЕНИЕ

В рамках выполнения работы было проведено исследование о использовании искусственного интеллекта в игровых приложениях, проанализированы существующие решения данной проблемы, выявлены их сильные и слабые стороны. Определена актуальность данной темы и поставлены цели выполнения для дальнейшего выполнения научно-исследовательской работы. Также были реализованы ключевые аспекты игровой логики и инфраструктуры, обеспечивающие создание динамичного и интерактивного игрового мира. Был разработан менеджер местности, обеспечивающий генерацию, загрузку и оптимизацию игровых ландшафтов. Раздел, посвященный модулю планирования действий, охватывал проектирование и разработку интеллектуальной системы, способной анализировать текущее состояние игры и принимать решения относительно действий персонажей. Спроектирован и разработан модуль передвижения игровых персонажей по карте с помощью алгоритма AStar. Разработаны и интегрированы разные версии алгоритма для лучшего результата в зависимости от ситуации. Изучены и добавлены методы оптимизации алгоритма.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

- 1) Habr. Создание искусственного интеллекта для игр [Электронный ресурс]. URL: <https://habr.com/ru/company/intel/blog/265679/>
- 2) Habr. Машинное обучение в Unity [Электронный ресурс]. <https://habr.com/ru/company/pixonix/blog/492548/>
- 3) Habr. Обучение с подкреплением [Электронный ресурс]. URL: <https://habr.com/ru/company/wunderfund/blog/670562/>
- 4) UNITY [Электронный ресурс]. URL: <https://forum.unity.com/>.
- 5) incredibuild.com [Электронный ресурс]. URL: <https://www.incredibuild.com/blog/unity-vs-unreal-what-kind-of-game-dev-are-you>
- 6) web.stanford.edu [Электронный ресурс]. URL: https://web.stanford.edu/class/cs123/lectures/CS123_lec08_HFSM_BT.pdf
- 7) Simplilearn [Электронный ресурс]. URL: <https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm>
- 8) Unity в действии, Джозеф Хокинг [Книга].
- 9) Creating an RTS Game in Unity, Бруно Чиканчи [Книг].
- 10) The Total Beginner's Guide to Game AI [Электронный ресурс]. URL: <https://www.gamedev.net/tutorials/programming/artificial-intelligence/the-total-beginners-guide-to-game-AI-r4942/>
- 11) A* search algorithm [Электронный ресурс]. URL: https://en.wikipedia.org/wiki/A*_search_algorithm
- 12) A*-Suchalgorithmus [Электронный ресурс]. URL: <https://www.geeksforgeeks.org/a-search-algorithm>

ПРИЛОЖЕНИЕ А

(Обязательное)

ИГРОВОЕ ПРИЛОЖЕНИЕ С ИСПОЛЬЗОВАНИЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Листинг программы

ПРИЛОЖЕНИЕ А.1

Файл «Planner.cs»

```
using System;
using System.Collections.Generic;
using System.Text;
using Newtonsoft.Json;
using UnityEngine.Profiling;
namespace SGoap
{
    public class Planner
    {
        private List<Action> actions_ = new List<Action>();
        private Stack<Node> usedNodes_ = new Stack<Node>(10000);
        private Stack<List<Action>> usedActions_ = new
Stack<List<Action>>(10000);
        private Queue<Action> actionQueue_ = new
Queue<Action>(20);
        public static bool Initialized;
        public Dictionary<string, string> GoalRecords = new
Dictionary<string, string>();
        public Dictionary<string, string> ErrorRecords = new
Dictionary<string, string>();
        public StringBuilder ErrorBuilder = new StringBuilder();
        public bool GenerateGoalsReport = false;
        public bool GenerateFailedPlansReport = false;
        public Planner()
        {
            if (Initialized)
                return;
            DictionaryPool<string, float>.Initialize(10000);
            Pool<Node>.Initialize(10000);
            ListPool<Action>.Initialize(10000);
            ListPool<Node>.Initialize(10000);
            Initialized = true;
        }
        public void AddAchievableActions(List<Action> actions,
bool simulate = false)
        {
            actions_.Clear();
            foreach (var action in actions)
            {
                if (simulate)
                {
                    actions_.Add(action);
                    continue;
                }
            }
        }
    }
}
```

```

        }
        if (action.IsUsable() ||
action.AlwaysIncludeInPlan)
        {
            action.DynamicallyEvaluateCost();
            actions_.Add(action);
        }
    }

    public Node BuildFirstNode(Dictionary<string, float>
worldStates, Dictionary<string, float> agentStates)
    {
        var firstNode = Pool<Node>.Get();
        firstNode.States = DictionaryPool<string,
float>.Get();
        foreach (var agentState in agentStates)
        {
            if
(!firstNode.States.ContainsKey(agentState.Key))
                firstNode.States.Add(agentState.Key,
agentState.Value);
        }
        foreach (var state in worldStates)
        {
            if (!firstNode.States.ContainsKey(state.Key))
                firstNode.States.Add(state.Key,
state.Value);
        }
        return firstNode;
    }

    public Queue<Action> Plan(Agent agent, string goal,
Dictionary<string, float> worldStates, List<Action> actions,
Dictionary<string, float> agentStates, bool simulate = false)
    {
        GenerateFailedPlansReport =
agent.PlannerSettings.GenerateFailedPlansReport;
        GenerateGoalsReport =
agent.PlannerSettings.GenerateGoalReport;
        Profiler.BeginSample("Plan Editor Reports");
        if (GenerateFailedPlansReport)
            ErrorBuilder.Clear();
        Profiler.EndSample();
        AddAchievableActions(actions, simulate);
        var firstNode = BuildFirstNode(worldStates,
agentStates);
        usedNodes_.Push(firstNode);
    }

```

```

        var leaves = BuildLeaves(goal, worldStates, actions,
agentStates, out var success, simulate);
        Profiler.BeginSample("Plan Editor Reports");
        if(GenerateGoalsReport)
            GoalRecords[goal] =
JsonConvert.SerializeObject(leaves);
        if (GenerateFailedPlansReport)
            ErrorRecords[goal] = ErrorBuilder.ToString();
        Profiler.EndSample();
        if (!success)
        {
            Clean(leaves);
            return null;
        }
        Node cheapest = null;
        foreach (var leaf in leaves)
        {
            if (cheapest == null)
                cheapest = leaf;
            else
            {
                if (leaf.Cost < cheapest.Cost)
                    cheapest = leaf;
            }
        }
        var result = ListPool<Action>.Get();
        usedActions_.Push(result);
        var n = cheapest;
        while (n != null)
        {
            if (n.Action != null)
                result.Insert(0, n.Action);
            n = n.Parent;
        }
        actionQueue_.Clear();
        foreach (var action in result)
            actionQueue_.Enqueue(action);
        Clean(leaves);
        return actionQueue_;
    }
    public void Clean(List<Node> leaves)
    {
        while (usedNodes_.Count > 0)
        {
            var node = usedNodes_.Pop();

```

```

        DictionaryPool<string,
float>.Return(node.States);
        node.Parent = null;
        node.States = null;
        node.Action = null;
        node.ActionName = string.Empty;
        node.Cost = 0;
        Pool<Node>.Return(node);
    }
    while (usedActions_.Count > 0)
        ListPool<Action>.Return(usedActions_.Pop());
    ListPool<Node>.Return(leaves);
}

public List<Node> BuildLeaves(string goal,
Dictionary<string, float> worldStates, List<Action> actions,
Dictionary<string, float> agentStates, out bool success, bool
simulate = false)
{
    Profiler.BeginSample("Build Leaves");
    AddAchievableActions(actions, simulate);
    var firstNode = BuildFirstNode(worldStates,
agentStates);
    usedNodes_.Push(firstNode);
    var leaves = ListPool<Node>.Get();
    //success = BuildGraph(firstNode, leaves, actions_,
goal); - This leads to a bug where success is false when there
is a leaf
    BuildGraph(firstNode, leaves, actions_, goal);
    success = leaves.Count > 0;
    Profiler.EndSample();
    return leaves;
}

public void AddError(Node node, Action currentAction,
Dictionary<string, float> currentStates)
{
    Profiler.BeginSample("Plan Editor Report");
    var n = node.Parent;
    ErrorBuilder.Append("<color=yellow>");
    while (n != null && n.Action != null)
    {
        ErrorBuilder.Append($" {n.Action.Name} -->");
        n = n.Parent;
    }
    ErrorBuilder.Append($" {currentAction.Name}");
    ErrorBuilder.Append("</color>");
}

```

```

        ErrorBuilder.Append("\n<color=red>      [Preconditions
not met]</color>");
        ErrorBuilder.Append("\n      [Preconditions]\n      ");
        foreach (var condition in
currentAction.PreConditions)
            ErrorBuilder.Append($"{condition.Operator}
{condition.Key}, ");
        ErrorBuilder.Append("\n      [Current States]\n
");
        foreach (var state in currentStates)
            ErrorBuilder.Append($"{state.Key}, ");
        ErrorBuilder.Append("\n\n");
        Profiler.EndSample();
    }
    public bool BuildGraph(Node parent, List<Node> leaves,
List<Action> usableActions, string goal)
    {
        var foundPath = false;
        foreach (var usableAction in usableActions)
        {
            var usable =
usableAction.IsPreconditionsMet(parent.States);
            if (!usable)
            {
                if (GenerateFailedPlansReport)
                    AddError(parent, usableAction,
parent.States);
                continue;
            }
            if
(!usableAction.IsProceduralPreconditionMet(parent.States))
                continue;
            var currentState = DictionaryPool<string,
float>.Get();
            var node = Pool<Node>.Get();
            usedNodes_.Push(node);
            node.States = currentState;
            node.Action = usableAction;
            node.ActionName = usableAction.Name;
            node.Parent = parent;
            node.Cost = parent.Cost + usableAction.Cost;
            node.Cost +=
usableAction.DynamicallyEvaluateCost(parent.States);
            foreach (var parentState in parent.States)
            {

```

```

if (!currentState.ContainsKey(parentState.Key))
    currentState.Add(parentState.Key,
parentState.Value);
    }
    foreach (var effect in
usableAction.AfterEffects)
    {
        if (!currentState.ContainsKey(effect.Key))
            currentState.Add(effect.Key,
effect.Value);
    }
    usableAction.ProceduralEffect(currentState);
    if (currentState.ContainsKey(goal))
    {
        leaves.Add(node);
        foundPath = true;
    }
    else
    {
        var subsetActions = ListPool<Action>.Get();
        usedActions_.Push(subsetActions);
        foreach (var action in usableActions)
            subsetActions.Add(action);
        subsetActions.Remove(usableAction);
        foundPath = BuildGraph(node, leaves,
subsetActions, goal);
    }
    }
    return foundPath;
}

}

public class Node
{
    public Dictionary<string, float> States;
    [JsonIgnore]
    public Action Action;
    public string ActionName { get; set; }
    public Node Parent;
    public float Cost;
    public override string ToString()
    {
        if (Action == null)
            return "Null";
        return Action.Name;
    }
}

```

```

    }
    public static class Pool<T> where T : new()
    {
        public static Stack<T> Cached;
        public static void Initialize(int count = 100)
        {
            Cached = new Stack<T>();
            for (int i = 0; i < count; i++)
                Cached.Push(new T());
        }
        public static T Get()
        {
            if (Cached.Count == 0)
                Initialize(100);
            return Cached.Pop();
        }
        public static void Return(T instance)
        {
            Cached.Push(instance);
        }
    }
    public static class ListPool<T>
    {
        public static Stack<List<T>> Cached = new
Stack<List<T>>();
        public static void Initialize(int count = 100)
        {
            Cached = new Stack<List<T>>();
            for (int i = 0; i < count; i++)
                Cached.Push(new List<T>(20));
        }
        public static List<T> Get()
        {
            return Cached.Pop();
        }
        public static void Return(List<T> instance)
        {
            instance.Clear();
            Cached.Push(instance);
        }
    }
    public static class DictionaryPool<T, W>
    {
        public static Stack<Dictionary<T, W>> Cached;
        public static void Initialize(int count = 100)
        {

```

```

        Cached = new Stack<Dictionary<T, W>>();
        for (int i = 0; i < count; i++)
            Cached.Push(new Dictionary<T, W>(20));
    }
    public static Dictionary<T, W> Get()
    {
        return Cached.Pop();
    }
    public static void Return(Dictionary<T, W> instance)
    {
        instance.Clear();
        Cached.Push(instance);
    }
}

[Flags]
public enum ELogLevel
{
    Default = 1 << 0,
    DeepBreakdown = 1 << 1,
    Goals = 1 << 2
}

```

ПРИЛОЖЕНИЕ А.2

Файл «AgentPlanner.cs»

```
namespace SGoap
{
    public class Agent : MonoBehaviour
    {
        public List<Goal> Goals;
        public PlannerSettings PlannerSettings;
        public States States = new States();
        public Queue<Action> StoredActionQueue = new
Queue<Action>();
        private float _planStartTime;
        private bool _replan = true;
        private List<Goal> _orderedGoals;
        public Goal CurrentGoal { get; private set; }
        public Action CurrentAction { get; private set; }
        public List<Action> Actions { get; private set; }
        public Queue<Action> ActionQueue { get; private set; }
        public bool Initialized { get; private set; }
        public Planner Planner { get; } = new Planner();
        public virtual void Start()
        {
            var agentDependencies =
GetComponentInChildren<IDataBind<Agent>>(includeInactive:
true);
            foreach (var dependency in agentDependencies)
                dependency.Bind(this);
            var stateMonitor =
GetComponentInChildren<AgentStateMonitor>();
            if (stateMonitor != null)
            {
                foreach (var state in stateMonitor.PreStates)
                    States.AddState(state.Key, state.Value);
            }
            InitializeActions();
            _orderedGoals = Goals.OrderByDescending(entry =>
entry.Priority).ToList();
            if (!Application.isEditor)
            {
                PlannerSettings.GenerateFailedPlansReport =
false;
                PlannerSettings.GenerateGoalReport = false;
            }
            Initialized = true;
        }
    }
}
```

```

public bool ShouldAbort()
{
    if (CurrentAction.ShouldAbort())
    {
        AbortPlan();
        return true;
    }
    return false;
}
public virtual void LateUpdate()
{
    if (PlannerSettings.RunOnLateUpdate)
        Run();
}
public virtual void Run()
{
    if (ExecutePlan())
        return;
    FindPlan();
    PrePerformPlanAction();
    EvaluationGoal();
}
public virtual bool ExecutePlan()
{
    if (CurrentAction != null && CurrentAction.Running)
    {
        if (ShouldAbort())
            return true;
        PerformPlanAction();
        return true;
    }
    return false;
}
public virtual void PrePerformPlanAction()
{
    Profiler.BeginSample("Agent.PrePerform");
    if (ActionQueue != null && ActionQueue.Count > 0)
    {
        CurrentAction = ActionQueue.Dequeue();
        if (CurrentAction.PrePerform())
        {
            if (CurrentAction.TrackStopWatch)
                CurrentAction.Stopwatch.Restart();
            else
                CurrentAction.Stopwatch.Reset();
            CurrentAction.Running = true;
            CurrentAction.OnPrePerform?.Invoke();
        }
    }
}

```

```

        }
        else
            AbortPlan();
    }
    Profiler.EndSample();
}
public virtual void PerformPlanAction()
{
    Profiler.BeginSample("Agent.Run");
    var actionStatus = CurrentAction.Perform();
    CurrentAction.OnPerform?.Invoke();
    CheckActionFailed(actionStatus);
    CheckActionSuccess(actionStatus);
    Profiler.EndSample();
    Profiler.BeginSample("Agent.CanAbortPlan");
    if (PlannerSettings.CanAbortPlans &&
        CurrentAction.CanAbort() && Time.time - _planStartTime >
        PlannerSettings.PlanRate)
        AbortPlan();
    Profiler.EndSample();
}
public virtual void CheckActionFailed(EActionStatus
status)
{
    if (status == EActionStatus.Failed)
    {
        CurrentAction?.OnPerformFailed?.Invoke();
        CurrentAction.OnFailed();
        CurrentAction.Running = false;
    }
}
public virtual void CheckActionSuccess(EActionStatus
status)
{
    if (status == EActionStatus.Success)
    {
        CurrentAction.PostPerform();
        CurrentAction.OnPostPerform?.Invoke();
        CurrentAction.Running = false;
    }
}
public virtual void FindPlan()
{
    if (_replan || ActionQueue == null)
    {
        foreach (var goal in _orderedGoals)
        {

```

```

        Profiler.BeginSample("Agent.Plan");
        ActionQueue = Planner.Plan(this, goal.Key,
World.Instance.StateMap, Actions, States.GetStates(), false);
        _planStartTime = Time.time;
        Profiler.EndSample();
        if (ActionQueue == null)
            continue;
        CurrentGoal = goal;
_replan = false;
        StoredActionQueue.Clear();
        foreach (var action in ActionQueue)
            StoredActionQueue.Enqueue(action);
        break;
    }
}
}
public virtual void EvaluationGoal()
{
    if (ActionQueue != null && ActionQueue.Count == 0)
    {
        if (CurrentGoal.Once) {
            Goals.Remove(CurrentGoal);
_orderedGoals.Remove(CurrentGoal);
        }
        _replan = true;
    }
}
public void ForceReplan()
{
    _replan = true;
    AbortPlan();
}
public void AbortPlan()
{
    if (CurrentAction != null)
    {
        CurrentAction.OnFailed();
CurrentAction.Running = false;
    }
    CleanPlan();
}
public void CleanPlan()
{
    if (CurrentAction != null)
        CurrentAction.Running = false;
}

```

```

        ActionQueue = null;
StoredActionQueue.Clear();        CurrentAction = null;
    }
    public void InitializeActions(List<Action> actions =
null)
    {
        Actions = actions ??
GetComponentInChildren<Action>().ToList();
        AbortPlan();
    }
    public void UpdateGoalOrderCache()
    {
        _orderedGoals = Goals.OrderByDescending(entry =>
entry.Priority).ToList();
    }
}

```

ПРИЛОЖЕНИЕ А.3

Файл «AIBase.cs»

```
using UnityEngine;
using System.Collections;
using UnityEngine.Serialization;
namespace Pathfinding {
    using Pathfinding.RVO;
    using Pathfinding.Util;
    [RequireComponent(typeof(Seeker))]
    public abstract class AIBase : VersionedMonoBehaviour {
        public float radius = 0.5f;
        public float height = 2;
        public float repathRate {
            get {
                return this.autoRepath.period;
            }
            set {
                this.autoRepath.period = value;
            }
        }
        public bool canSearch {
            get {
                return this.autoRepath.mode !=
AutoRepathPolicy.Mode.Never;
            }
            set {
                if (value) {
                    if (this.autoRepath.mode ==
AutoRepathPolicy.Mode.Never) {
                        this.autoRepath.mode =
AutoRepathPolicy.Mode.EveryNSeconds;
                    }
                } else {
                    this.autoRepath.mode =
AutoRepathPolicy.Mode.Never;
                }
            }
        }
        public bool canMove = true;

        [UnityEngine.Serialization.FormerlySerializedAs("speed")]
        public float maxSpeed = 1;
        public Vector3 gravity = new Vector3(float.NaN,
float.NaN, float.NaN);
        public LayerMask groundMask = -1;
    }
}
```

```

[System.Obsolete("Use the height property instead (2x
this value)")]
    public float centerOffset {
        get { return height * 0.5f; } set { height = value
* 2; }
    }
    [SerializeField]
    [HideInInspector]
    [FormerlySerializedAs("centerOffset")]
    float centerOffsetCompatibility = float.NaN;
    [SerializeField]
    [HideInInspector]

    [UnityEngine.Serialization.FormerlySerializedAs("repathRate"
)]
        float repathRateCompatibility = float.NaN;
        [SerializeField]
        [HideInInspector]

    [UnityEngine.Serialization.FormerlySerializedAs("canSearch")
]

    [UnityEngine.Serialization.FormerlySerializedAs("repeatedlyS
earchPaths")]
        bool canSearchCompability = false;

    [UnityEngine.Serialization.FormerlySerializedAs("rotationIn2
D")]
        public OrientationMode orientation =
OrientationMode.ZAxisForward;
        [System.Obsolete("Use orientation instead")]
        public bool rotationIn2D {
            get { return orientation ==
OrientationMode.YAxisForward; }
            set { orientation = value ?
OrientationMode.YAxisForward : OrientationMode.ZAxisForward; }
        }
        public bool enableRotation = true;
        protected Vector3 simulatedPosition;
        protected Quaternion simulatedRotation;
        public Vector3 position { get { return updatePosition ?
tr.position : simulatedPosition; } }
        public Quaternion rotation {
            get { return updateRotation ? tr.rotation :
simulatedRotation; }
            set {

```

```

        if (updateRotation) {
            tr.rotation = value;
        } else {
            simulatedRotation = value;
        }
    }
}
Vector3 accumulatedMovementDelta = Vector3.zero;
protected Vector2 velocity2D;
protected float verticalVelocity;
protected Seeker seeker;
protected Transform tr;
protected Rigidbody rigid;
protected Rigidbody2D rigid2D;
protected CharacterController controller;
protected RVOController rvoController;
public IMovementPlane movementPlane =
GraphTransform.identityTransform;
[System.NonSerialized]
public bool updatePosition = true;
[System.NonSerialized]
public bool updateRotation = true;
public AutoRepathPolicy autoRepath = new
AutoRepathPolicy();
protected bool usingGravity { get; set; }
protected float lastDeltaTime;
protected int prevFrame;
protected Vector3 prevPosition1;
protected Vector3 prevPosition2;
protected Vector2 lastDeltaPosition;
protected bool waitingForPathCalculation = false;

[UnityEngine.Serialization.FormerlySerializedAs("target")] [S
erializeField] [HideInInspector]
    Transform targetCompatibility;
    bool startHasRun = false;
    [System.Obsolete("Use the destination property or the
AIDestinationSetter component instead")]
    public Transform target {
        get {
            var setter =
GetComponent<AIDestinationSetter>();
            return setter != null ? setter.target : null;
        }
        set {
            targetCompatibility = null;

```

```

        var setter =
GetComponent<AIDestinationSetter>();
        if (setter == null) setter =
gameObject.AddComponent<AIDestinationSetter>();
        setter.target = value;
        destination = value != null ? value.position
: new Vector3(float.PositiveInfinity, float.PositiveInfinity,
float.PositiveInfinity);
    }
}
public Vector3 destination { get; set; }
public Vector3 velocity {
    get {
        return lastDeltaTime > 0.000001f ?
(prevPosition1 - prevPosition2) / lastDeltaTime : Vector3.zero;
    }
}
public Vector3 desiredVelocity { get { return
lastDeltaTime > 0.00001f ?
movementPlane.ToWorld(lastDeltaPosition / lastDeltaTime,
verticalVelocity) : Vector3.zero; } }
public bool isStopped { get; set; }
public System.Action onSearchPath { get; set; }
protected virtual bool shouldRecalculatePath {
    get {
        return !waitingForPathCalculation &&
autoRepath.ShouldRecalculatePath(position, radius, destination);
    }
}
protected AIBase () {
    destination = new Vector3(float.PositiveInfinity,
float.PositiveInfinity, float.PositiveInfinity);
}
public virtual void FindComponents () {
    tr = transform;
    seeker = GetComponent<Seeker>();
    rvoController = GetComponent<RVOController>();
    controller = GetComponent<CharacterController>();
    rigid = GetComponent<Rigidbody>();
    rigid2D = GetComponent<Rigidbody2D>();
}
protected virtual void OnEnable () {
    FindComponents();
    seeker.pathCallback += OnPathComplete;
    Init();
}
}

```

```

protected virtual void Start () {
    startHasRun = true;
    Init();
}
void Init () {
    if (startHasRun) {
        if (canMove) Teleport(position, false);
        autoRepath.Reset();
        if (shouldRecalculatePath) SearchPath();
    }
}
public virtual void Teleport (Vector3 newPosition, bool
clearPath = true) {
    if (clearPath) ClearPath();
    prevPosition1 = prevPosition2 = simulatedPosition
= newPosition;
    if (updatePosition) tr.position = newPosition;
    if (rvoController != null)
rvoController.Move(Vector3.zero);
    if (clearPath) SearchPath();
}
protected void CancelCurrentPathRequest () {
    waitingForPathCalculation = false;
    if (seeker != null)
seeker.CancelCurrentPathRequest();
}
protected virtual void OnDisable () {
    ClearPath();
    seeker.pathCallback -= OnPathComplete;
    velocity2D = Vector3.zero;
    accumulatedMovementDelta = Vector3.zero;
    verticalVelocity = 0f;
    lastDeltaTime = 0;
}
protected virtual void Update () {
    if (shouldRecalculatePath) SearchPath();
    usingGravity = !(gravity == Vector3.zero) &&
(!updatePosition || ((rigid == null || rigid.isKinematic) &&
(rigid2D == null || rigid2D.isKinematic)));
    if (rigid == null && rigid2D == null && canMove) {
        Vector3 nextPosition;
        Quaternion nextRotation;
        MovementUpdate(Time.deltaTime, out
nextPosition, out nextRotation);
        FinalizeMovement(nextPosition, nextRotation);
    }
}

```

```

    }
    protected virtual void FixedUpdate () {
        if (!(rigid == null && rigid2D == null) &&
canMove) {
            Vector3 nextPosition;
            Quaternion nextRotation;
            MovementUpdate(Time.fixedDeltaTime, out
nextPosition, out nextRotation);
            FinalizeMovement(nextPosition, nextRotation);
        }
    }
    public void MovementUpdate (float deltaTime, out
Vector3 nextPosition, out Quaternion nextRotation) {
        lastDeltaTime = deltaTime;
        MovementUpdateInternal(deltaTime, out
nextPosition, out nextRotation);
    }
    protected abstract void MovementUpdateInternal(float
deltaTime, out Vector3 nextPosition, out Quaternion
nextRotation);
    protected virtual void CalculatePathRequestEndpoints
(out Vector3 start, out Vector3 end) {
        start = GetFeetPosition();
        end = destination;
    }
    public virtual void SearchPath () {
        if (float.IsPositiveInfinity(destination.x))
return;

        if (onSearchPath != null) onSearchPath();
        Vector3 start, end;
        CalculatePathRequestEndpoints(out start, out end);
        ABPath p = ABPath.Construct(start, end, null);
        SetPath(p, false);
    }
    public virtual Vector3 GetFeetPosition () {
        return position;
    }
    protected abstract void OnPathComplete(Path newPath);
    protected abstract void ClearPath();
    public void SetPath (Path path, bool
updateDestinationFromPath = true) {
        if (updateDestinationFromPath && path is ABPath
abPath && !(path is RandomPath)) {
            this.destination = abPath.originalEndPoint;
        }
        if (path == null) {

```

```

        CancelCurrentPathRequest();
        ClearPath();
    } else if (path.PipelineState ==
PathState.Created) {
        waitingForPathCalculation = true;
        seeker.CancelCurrentPathRequest();
        seeker.StartPath(path);
        autoRepath.DidRecalculatePath(destination);
    } else if (path.PipelineState ==
PathState.Returned) {
        if (seeker.GetCurrentPath() != path)
seeker.CancelCurrentPathRequest();
        else throw new System.ArgumentException("If
you calculate the path using seeker.StartPath then this script
will pick up the calculated path anyway as it listens for all
paths the Seeker finishes calculating. You should not call
SetPath in that case.");
        OnPathComplete(path);
    } else {
        throw new System.ArgumentException("You must
call the SetPath method with a path that either has been
completely calculated or one whose path calculation has not been
started at all. It looks like the path calculation for the path
you tried to use has been started, but is not yet finished.");
    }
}

protected void ApplyGravity (float deltaTime) {
    if (usingGravity) {
        float verticalGravity;
        velocity2D += movementPlane.ToPlane(deltaTime
* (float.IsNaN(gravity.x) ? Physics.gravity : gravity), out
verticalGravity);
        verticalVelocity += verticalGravity;
    } else {
        verticalVelocity = 0;
    }
}

protected Vector2 CalculateDeltaToMoveThisFrame
(Vector2 position, float distanceToEndOfPath, float deltaTime) {
    if (rvoController != null &&
rvoController.enabled) {
        return
movementPlane.ToPlane(rvoController.CalculateMovementDelta(movem
entPlane.ToWorld(position, 0), deltaTime));
    }
}

```

```

        return Vector2.ClampMagnitude(velocity2D *
deltaTime, distanceToEndOfPath);
    }
    public Quaternion SimulateRotationTowards (Vector3
direction, float maxDegrees) {
        return
SimulateRotationTowards(movementPlane.ToPlane(direction),
maxDegrees);
    }
    protected Quaternion SimulateRotationTowards (Vector2
direction, float maxDegrees) {
        if (direction != Vector2.zero) {
            Quaternion targetRotation =
Quaternion.LookRotation(movementPlane.ToWorld(direction, 0),
movementPlane.ToWorld(Vector2.zero, 1));
            if (orientation ==
OrientationMode.YAxisForward) targetRotation *=
Quaternion.Euler(90, 0, 0);
            return
Quaternion.RotateTowards(simulatedRotation, targetRotation,
maxDegrees);
        }
        return simulatedRotation;
    }
    public virtual void Move (Vector3 deltaPosition) {
        accumulatedMovementDelta += deltaPosition;
    }
    public virtual void FinalizeMovement (Vector3
nextPosition, Quaternion nextRotation) {
        if (enableRotation)
FinalizeRotation(nextRotation);
        FinalizePosition(nextPosition);
    }
    void FinalizeRotation (Quaternion nextRotation) {
        simulatedRotation = nextRotation;
        if (updateRotation) {
            if (rigid != null)
rigid.MoveRotation(nextRotation);
            else if (rigid2D != null)
rigid2D.MoveRotation(nextRotation.eulerAngles.z);
            else tr.rotation = nextRotation;
        }
    }
    void FinalizePosition (Vector3 nextPosition) {
        Vector3 currentPosition = simulatedPosition;
        bool positionDirty1 = false;

```

```

        if (controller != null && controller.enabled &&
updatePosition) {
            tr.position = currentPosition;
            controller.Move((nextPosition
currentPosition) + accumulatedMovementDelta);
            currentPosition = tr.position;
            if (controller.isGrounded) verticalVelocity =
0;
        } else {
            float lastElevation;
            movementPlane.ToPlane(currentPosition, out
lastElevation);
            currentPosition = nextPosition +
accumulatedMovementDelta;
            if (usingGravity) currentPosition =
RaycastPosition(currentPosition, lastElevation);
            positionDirty1 = true;
        }
        bool positionDirty2 = false;
        currentPosition = ClampToNavmesh(currentPosition,
out positionDirty2);
        if ((positionDirty1 || positionDirty2) &&
updatePosition) {
            if (rigid != null)
rigid.MovePosition(currentPosition);
            else if (rigid2D != null)
rigid2D.MovePosition(currentPosition);
            else tr.position = currentPosition;
        }
        accumulatedMovementDelta = Vector3.zero;
        simulatedPosition = currentPosition;
        UpdateVelocity();
    }
    protected void UpdateVelocity () {
        var currentFrame = Time.frameCount;
        if (currentFrame != prevFrame) prevPosition2 =
prevPosition1;
        prevPosition1 = position;
        prevFrame = currentFrame;
    }
    protected virtual Vector3 ClampToNavmesh (Vector3
position, out bool positionChanged) {
        positionChanged = false;
        return position;
    }
}

```

```

        protected Vector3 RaycastPosition (Vector3 position,
float lastElevation) {
            RaycastHit hit;
            float elevation;
            movementPlane.ToPlane(position, out elevation);
            float rayLength = tr.localScale.y * height * 0.5f
+ Mathf.Max(0, lastElevation-elevation);
            Vector3 rayOffset =
movementPlane.ToWorld(Vector2.zero, rayLength);
            if (Physics.Raycast(position + rayOffset, -
rayOffset, out hit, rayLength, groundMask,
QueryTriggerInteraction.Ignore)) {
                verticalVelocity *= System.Math.Max(0, 1 - 5
* lastDeltaTime);
                return hit.point;
            }
            return position;
        }
        protected virtual void OnDrawGizmosSelected () {
            if (Application.isPlaying) FindComponents();
        }
        public static readonly Color ShapeGizmoColor = new
Color(240/255f, 213/255f, 30/255f);
        protected virtual void OnDrawGizmos () {
            if (!Application.isPlaying || !enabled)
FindComponents();
            var color = ShapeGizmoColor;
            if (rvoController != null && rvoController.locked)
color *= 0.5f;
            if (orientation == OrientationMode.YAxisForward) {
                Draw.Gizmos.Cylinder(position,
Vector3.forward, 0, radius * tr.localScale.x, color);
            } else {
                Draw.Gizmos.Cylinder(position, rotation *
Vector3.up, tr.localScale.y * height, radius * tr.localScale.x,
color);
            }
            if (!float.IsPositiveInfinity(destination.x) &&
Application.isPlaying) Draw.Gizmos.CircleXZ(destination, 0.2f,
Color.blue);
            autoRepath.DrawGizmos(position, radius);
        }
        protected override void Reset () {
            ResetShape();
            base.Reset();
        }

```

```

void ResetShape () {
    var cc = GetComponent<CharacterController>();
    if (cc != null) {
        radius = cc.radius;
        height = Mathf.Max(radius*2, cc.height);
    }
}

protected override int OnUpgradeSerializedData (int
version, bool unityThread) {
    if (unityThread &&
!float.IsNaN(centerOffsetCompatibility)) {
        height = centerOffsetCompatibility*2;
        ResetShape();
        var rvo = GetComponent<RVOController>();
        if (rvo != null) radius =
rvo.radiusBackingField;
        centerOffsetCompatibility = float.NaN;
    }
    #pragma warning disable 618
    if (unityThread && targetCompatibility != null)
target = targetCompatibility;
    #pragma warning restore 618
    if (version <= 3) {
        repathRate = repathRateCompatibility;
        canSearch = canSearchCompability;
    }
    return 5;
}
}
}

```

ПРИЛОЖЕНИЕ А.4

Файл «AstarData.cs»

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using Pathfinding.WindowsStore;
using Pathfinding.Serialization;
#if UNITY_WINRT && !UNITY_EDITOR
#endif
namespace Pathfinding {
    [System.Serializable]
    public class AstarData {
        public static AstarPath active {
            get {
                return AstarPath.active;
            }
        }
        #region Fields
        public NavMeshGraph navmesh { get; private set; }
#if !ASTAR_NO_GRID_GRAPH
        public GridGraph gridGraph { get; private set; }
        public LayerGridGraph layerGridGraph { get; private
set; }
#endif
#if !ASTAR_NO_POINT_GRAPH
        public PointGraph pointGraph { get; private set; }
#endif
        public RecastGraph recastGraph { get; private set; }
        public System.Type[] graphTypes { get; private set; }
#if ASTAR_FAST_NO_EXCEPTIONS || UNITY_WINRT || UNITY_WEBGL
        public static readonly System.Type[] DefaultGraphTypes
= new System.Type[] {
#if !ASTAR_NO_GRID_GRAPH
            typeof(GridGraph),
            typeof(LayerGridGraph),
#endif
#if !ASTAR_NO_POINT_GRAPH
            typeof(PointGraph),
#endif
            typeof(NavMeshGraph),
            typeof(RecastGraph),
        };
#endif
        [System.NonSerialized]
        public NavGraph[] graphs = new NavGraph[0];
    }
}
```

```

[SerializeField]
string dataString;
[SerializeField]

[UnityEngine.Serialization.FormerlySerializedAs("data")]
private byte[] upgradeData;
private byte[] data {
    get {
        if (upgradeData != null && upgradeData.Length
> 0) {
            data = upgradeData;
            upgradeData = null;
        }
        return dataString != null?
System.Convert.FromBase64String(dataString) : null;
    }
    set {
        dataString = value != null?
System.Convert.ToBase64String(value) : null;
    }
}

public TextAsset file_cachedStartup;
public byte[] data_cachedStartup;
[SerializeField]
public bool cacheStartup;
List<bool> graphStructureLocked = new List<bool>();
#endregion
public byte[] GetData () {
    return data;
}

public void SetData (byte[] data) {
    this.data = data;
}

public void Awake () {
    graphs = new NavGraph[0];
    if (cacheStartup && file_cachedStartup != null) {
        LoadFromCache();
    } else {
        DeserializeGraphs();
    }
}

internal void LockGraphStructure (bool
allowAddingGraphs = false) {
    graphStructureLocked.Add(allowAddingGraphs);
}

internal void UnlockGraphStructure () {

```

```

        if (graphStructureLocked.Count == 0) throw new
System.InvalidOperationException();

        graphStructureLocked.RemoveAt(graphStructureLocked.Count -
1);
    }
    PathProcessor.GraphUpdateLock AssertSafe (bool
onlyAddingGraph = false) {
        if (graphStructureLocked.Count > 0) {
            bool allowAdding = true;
            for (int i = 0; i <
graphStructureLocked.Count; i++) allowAdding &=
graphStructureLocked[i];
            if (!(onlyAddingGraph && allowAdding)) throw
new System.InvalidOperationException("Graphs cannot be added,
removed or serialized while the graph structure is locked. This
is the case when a graph is currently being scanned and when
executing graph updates and work items.\nHowever as a special
case, graphs can be added inside work items.");
        }
        var graphLock = active.PausePathfinding();
        if (!active.IsInsideWorkItem) {
            active.FlushWorkItems();
            active.pathReturnQueue.ReturnPaths(false);
        }
        return graphLock;
    }
    public void GetNodes (System.Action<GraphNode>
callback) {
        for (int i = 0; i < graphs.Length; i++) {
            if (graphs[i] != null)
graphs[i].GetNodes(callback);
        }
    }
    public void UpdateShortcuts () {
        navmesh =
(NavMeshGraph) FindGraphOfType(typeof(NavMeshGraph));
#ifdef !ASTAR_NO_GRID_GRAPH
        gridGraph =
(GridGraph) FindGraphOfType(typeof(GridGraph));
        layerGridGraph =
(LayerGridGraph) FindGraphOfType(typeof(LayerGridGraph));
#endif
#ifdef !ASTAR_NO_POINT_GRAPH
        pointGraph =
(PointGraph) FindGraphOfType(typeof(PointGraph));

```

```

#endif

        recastGraph
(RecastGraph) FindGraphOfType (typeof (RecastGraph));
    }
    public void LoadFromCache () {
        var graphLock = AssertSafe();
        if (file_cachedStartup != null) {
            var bytes = file_cachedStartup.bytes;
            DeserializeGraphs (bytes);

            GraphModifier.TriggerEvent (GraphModifier.EventType.PostCache
Load);
        } else {
            Debug.LogError("Can't load from cache since
the cache is empty");
        }
        graphLock.Release();
    }
    #region Serialization
    public byte[] SerializeGraphs () {
        return
SerializeGraphs (SerializeSettings.Settings);
    }
    public byte[] SerializeGraphs (SerializeSettings
settings) {
        uint checksum;
        return SerializeGraphs (settings, out checksum);
    }
    public byte[] SerializeGraphs (SerializeSettings
settings, out uint checksum) {
        var graphLock = AssertSafe();
        var sr = new AstarSerializer (this, settings,
active.gameObject);
        sr.OpenSerialize();
        sr.SerializeGraphs (graphs);
        sr.SerializeExtraInfo();
        byte[] bytes = sr.CloseSerialize();
        checksum = sr.GetChecksum();
        #if ASTARDEBUG
            Debug.Log ("Got a whole bunch of data,
"+bytes.Length+" bytes");
        #endif
        graphLock.Release();
        return bytes;
    }
    public void DeserializeGraphs () {

```

```

        if (data != null) {
            DeserializeGraphs(data);
        }
    }

    void ClearGraphs () {
        if (graphs == null) return;
        for (int i = 0; i < graphs.Length; i++) {
            if (graphs[i] != null) {

                ((IGraphInternals)graphs[i]).OnDestroy();
                graphs[i].active = null;
            }
        }
        graphs = new NavGraph[0];
        UpdateShortcuts();
    }

    public void OnDestroy () {
        ClearGraphs();
    }

    public void DeserializeGraphs (byte[] bytes) {
        var graphLock = AssertSafe();
        ClearGraphs();
        DeserializeGraphsAdditive(bytes);
        graphLock.Release();
    }

    public void DeserializeGraphsAdditive (byte[] bytes) {
        var graphLock = AssertSafe();
        try {
            if (bytes != null) {
                var sr = new AstarSerializer(this,
active.gameObject);

                if (sr.OpenDeserialize(bytes)) {
                    DeserializeGraphsPartAdditive(sr);
                    sr.CloseDeserialize();
                } else {
                    Debug.Log("Invalid data file
(cannot read zip).\nThe data is either corrupt or it was saved
using a 3.0.x or earlier version of the system");
                }
            } else {
                throw new
System.ArgumentNullException("bytes");
            }
            active.VerifyIntegrity();
        } catch (System.Exception e) {

```

```

        Debug.LogError("Caught exception while
deserializing data.\n"+e);
        graphs = new NavGraph[0];
    }
    UpdateShortcuts();
    graphLock.Release();
}
void DeserializeGraphsPartAdditive (AstarSerializer sr)
{
    if (graphs == null) graphs = new NavGraph[0];
    var gr = new List<NavGraph>(graphs);
    sr.SetGraphIndexOffset(gr.Count);
    if (graphTypes == null) FindGraphTypes();
    gr.AddRange(sr.DeserializeGraphs(graphTypes));
    graphs = gr.ToArray();
    sr.DeserializeEditorSettingsCompatibility();
    sr.DeserializeExtraInfo();
    for (int i = 0; i < graphs.Length; i++) {
        if (graphs[i] == null) continue;
        graphs[i].GetNodes(node => node.GraphIndex =
(uint)i);
    }
    for (int i = 0; i < graphs.Length; i++) {
        for (int j = i+1; j < graphs.Length; j++) {
            if (graphs[i] != null && graphs[j] !=
null && graphs[i].guid == graphs[j].guid) {
                Debug.LogWarning("Guid Conflict
when importing graphs additively. Imported graph will get a new
Guid.\nThis message is (relatively) harmless.");
                graphs[i].guid =
Pathfinding.Util.Guid.NewGuid();
                break;
            }
        }
    }
    sr.PostDeserialization();
    active.hierarchicalGraph.RecalculateIfNecessary();
}
#endregion
public void FindGraphTypes () {
#if !ASTAR_FAST_NO_EXCEPTIONS && !UNITY_WINRT && !UNITY_WEBGL
    var graphList = new List<System.Type>();
    foreach (var assembly in
System.AppDomain.CurrentDomain.GetAssemblies()) {
        System.Type[] types = null;
        try {

```

```

        types = assembly.GetTypes();
    } catch {
        continue;
    }
    foreach (var type in types) {
#if NETFX_CORE && !UNITY_EDITOR
        System.Type baseType =
type.GetTypeInfo().BaseType;
#else
        var baseType = type.BaseType;
#endif

        while (baseType != null) {
            if (System.Type.Equals(baseType,
typeof(NavGraph))) {
                graphList.Add(type);
                break;
            }
#if NETFX_CORE && !UNITY_EDITOR
            baseType =
baseType.GetTypeInfo().BaseType;
#else
            baseType = baseType.BaseType;
#endif
        }
    }
    graphTypes = graphList.ToArray();
#if ASTARDEBUG
    Debug.Log("Found " + graphTypes.Length + " graph
types");
#endif
#else
    graphTypes = DefaultGraphTypes;
#endif
}
#region GraphCreation
[System.Obsolete("If really necessary. Use
System.Type.GetType instead.")]
public System.Type GetGraphType (string type) {
    for (int i = 0; i < graphTypes.Length; i++) {
        if (graphTypes[i].Name == type) {
            return graphTypes[i];
        }
    }
    return null;
}
}

```

```

[System.Obsolete("Use CreateGraph(System.Type)
instead")]
public NavGraph CreateGraph (string type) {
    Debug.Log("Creating Graph of type '"+type+"'");
    for (int i = 0; i < graphTypes.Length; i++) {
        if (graphTypes[i].Name == type) {
            return CreateGraph(graphTypes[i]);
        }
    }
    Debug.LogError("Graph type ('"+type+"') wasn't
found");
    return null;
}
internal NavGraph CreateGraph (System.Type type) {
    var graph = System.Activator.CreateInstance(type)
as NavGraph;
    graph.active = active;
    return graph;
}
[System.Obsolete("Use AddGraph(System.Type) instead")]
public NavGraph AddGraph (string type) {
    NavGraph graph = null;
    for (int i = 0; i < graphTypes.Length; i++) {
        if (graphTypes[i].Name == type) {
            graph = CreateGraph(graphTypes[i]);
        }
    }
    if (graph == null) {
        Debug.LogError("No NavGraph of type
 '"+type+"' could be found");
        return null;
    }
    AddGraph(graph);
    return graph;
}
public NavGraph AddGraph (System.Type type) {
    NavGraph graph = null;
    for (int i = 0; i < graphTypes.Length; i++) {
        if (System.Type.Equals(graphTypes[i], type))
{
            graph = CreateGraph(graphTypes[i]);
        }
    }
    if (graph == null) {

```

```

        Debug.LogError("No NavGraph of type
        '"+type+"' could be found, "+graphTypes.Length+" graph types are
        available");
        return null;
    }
    AddGraph(graph);
    return graph;
}
void AddGraph (NavGraph graph) {
    var graphLock = AssertSafe(true);
    bool foundEmpty = false;
    for (int i = 0; i < graphs.Length; i++) {
        if (graphs[i] == null) {
            graphs[i] = graph;
            graph.graphIndex = (uint)i;
            foundEmpty = true;
            break;
        }
    }
    if (!foundEmpty) {
        if (graphs != null && graphs.Length >=
        GraphNode.MaxGraphIndex) {
            throw new System.Exception("Graph Count
            Limit Reached. You cannot have more than " +
            GraphNode.MaxGraphIndex + " graphs.");
        }
        var graphList = new List<NavGraph>(graphs ??
        new NavGraph[0]);
        graphList.Add(graph);
        graphs = graphList.ToArray();
        graph.graphIndex = (uint)(graphs.Length-1);
    }
    UpdateShortcuts();
    graph.active =
    active;
    graphLock.Release();
}
public bool RemoveGraph (NavGraph graph) {
    var graphLock = AssertSafe();
    ((IGraphInternals)graph).OnDestroy();
    graph.active = null;
    int i = System.Array.IndexOf(graphs, graph);
    if (i != -1) graphs[i] = null;
    UpdateShortcuts();
    graphLock.Release();
    return i != -1;
}
}
#endif

```

```

#region GraphUtility
public static NavGraph GetGraph (GraphNode node) {
    if (node == null) return null;
    AstarPath script = AstarPath.active;
    if (script == null) return null;
    AstarData data = script.data;
    if (data == null || data.graphs == null) return
null;

    uint graphIndex = node.GraphIndex;
    if (graphIndex >= data.graphs.Length) {
        return null;
    }
    return data.graphs[(int)graphIndex];
}
public NavGraph FindGraph (System.Func<NavGraph, bool>
predicate) {
    if (graphs != null) {
        for (int i = 0; i < graphs.Length; i++) {
            if (graphs[i] != null &&
predicate(graphs[i])) {
                return graphs[i];
            }
        }
    }
    return null;
}
public NavGraph FindGraphOfType (System.Type type) {
    return FindGraph(graph =>
System.Type.Equals(graph.GetType(), type));
}
public NavGraph FindGraphWhichInheritsFrom (System.Type
type) {
    return FindGraph(graph =>
WindowsStoreCompatibility.GetTypeInfo(type).IsAssignableFrom(Win
dowsStoreCompatibility.GetTypeInfo(graph.GetType())));
}
public IEnumerable FindGraphsOfType (System.Type type)
{
    if (graphs == null) yield break;
    for (int i = 0; i < graphs.Length; i++) {
        if (graphs[i] != null &&
System.Type.Equals(graphs[i].GetType(), type)) {
            yield return graphs[i];
        }
    }
}
public IEnumerable GetUpdateableGraphs () {
    if (graphs == null) yield break;

```

```

        for (int i = 0; i < graphs.Length; i++) {
            if (graphs[i] is IUpdatableGraph) {
                yield return graphs[i];
            }
        }
    }

[System.Obsolete("Obsolete because it is not used by
the package internally and the use cases are few. Iterate
through the graphs array instead.")]
    public IEnumerable GetRaycastableGraphs () {
        if (graphs == null) yield break;
        for (int i = 0; i < graphs.Length; i++) {
            if (graphs[i] is IRaycastableGraph) {
                yield return graphs[i];
            }
        }
    }

    public int GetGraphIndex (NavGraph graph) {
        if (graph == null) throw new
System.ArgumentNullException("graph");
        var index = -1;
        if (graphs != null) {
            index = System.Array.IndexOf(graphs, graph);
            if (index == -1) Debug.LogError("Graph
doesn't exist");
        }
        return index;
    }
    #endregion
}

```

ПРИЛОЖЕНИЕ А.5

Файл «Path.cs»

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
namespace Pathfinding {
    public interface ITraversalProvider {
        bool CanTraverse(Path path, GraphNode node);
        uint GetTraversalCost(Path path, GraphNode node);
    }
    public static class DefaultITraversalProvider {
        public static bool CanTraverse (Path path, GraphNode
node) {
            return node.Walkable && (path.enabledTags >>
(int)node.Tag & 0x1) != 0;
        }
        public static uint GetTraversalCost (Path path,
GraphNode node) {
            return path.GetTagPenalty((int)node.Tag) +
node.Penalty;
        }
    }
    public abstract class Path : IPathInternals {
#if ASTAR_POOL_DEBUG
        private string pathTraceInfo = "";
        private List<string> claimInfo = new List<string>();
        ~Path() {
            Debug.Log("Destroying " + GetType().Name + "
instance");
            if (claimed.Count > 0) {
                Debug.LogWarning("Pool Is Leaking. See list
of claims:\n" +
                    "Each message below will list what
objects are currently claiming the path." +
                    " These objects have removed their
reference to the path object but has not called .Release on it
(which is bad).\n" + pathTraceInfo+"\n");
                for (int i = 0; i < claimed.Count; i++) {
                    Debug.LogWarning("- Claim "+ (i+1) + "
is by a " + claimed[i].GetType().Name + "\n"+claimInfo[i]);
                }
            } else {
                Debug.Log("Some scripts are not using
pooling.\n" + pathTraceInfo + "\n");
            }
        }
    }
}
```

```

    }
#endif

    protected PathHandler pathHandler;
    public OnPathDelegate callback;
    public OnPathDelegate immediateCallback;
    public PathState PipelineState { get; private set; }
    System.Object stateLock = new object();
    public ITraversalProvider traversalProvider;
    protected PathCompleteState completeState;
    public PathCompleteState CompleteState {
        get { return completeState; }
        protected set {
            lock (stateLock) {
                if (completeState !=
PathCompleteState.Error) completeState = value;
            }
        }
    }

    public bool error { get { return CompleteState ==
PathCompleteState.Error; } }
    public string errorLog { get; private set; }
    public List<GraphNode> path;
    public List<Vector3> vectorPath;
    protected PathNode currentR;
    public float duration;
    public int searchedNodes { get; protected set; }
    bool IPathInternals.Pooled { get; set; }
    [System.Obsolete("Has been renamed to 'Pooled' to use
more widely understood terminology", true)]
    internal bool recycled { get { return false; } }
    protected bool hasBeenReset;
    public NNConstraint nnConstraint =
PathNNConstraint.Default;
    internal Path next;
    public Heuristic heuristic;
    public float heuristicScale = 1F;
    public ushort pathID { get; private set; }
    protected GraphNode hTargetNode;
    protected Int3 hTarget;
    public int enabledTags = -1;
    static readonly int[] ZeroTagPenalties = new int[32];
    protected int[] internalTagPenalties;
    protected int[] manualTagPenalties;
    public int[] tagPenalties {
        get {
            return manualTagPenalties;

```

```

    }
    set {
        if (value == null || value.Length != 32) {
            manualTagPenalties = null;
            internalTagPenalties = ZeroTagPenalties;
        } else {
            manualTagPenalties = value;
            internalTagPenalties = value;
        }
    }
}

public virtual bool FloodingPath {
    get {
        return false;
    }
}

public float GetTotalLength () {
    if (vectorPath == null) return
float.PositiveInfinity;
    float tot = 0;
    for (int i = 0; i < vectorPath.Count-1; i++) tot
+= Vector3.Distance(vectorPath[i], vectorPath[i+1]);
    return tot;
}

public IEnumerator WaitForPath () {
    if (PipelineState == PathState.Created) throw new
System.InvalidOperationException("This path has not been started
yet");

    while (PipelineState != PathState.Returned) yield
return null;
}

public void BlockUntilCalculated () {
    AstarPath.BlockUntilCalculated(this);
}

internal uint CalculateHScore (GraphNode node) {
    uint h;
    switch (heuristic) {
        case Heuristic.Euclidean:
            h = (uint) (((GetHTarget() -
node.position).costMagnitude)*heuristicScale);
            if (hTargetNode != null) {
                h = System.Math.Max(h,
AstarPath.active.euclideanEmbedding.GetHeuristic(node.NodeIndex,
hTargetNode.NodeIndex));
            }
            return h;
    }
}

```

```

        case Heuristic.Manhattan:
            Int3 p2 = node.position;
            h = (uint)((System.Math.Abs(hTarget.x-p2.x) +
System.Math.Abs(hTarget.y-p2.y) + System.Math.Abs(hTarget.z-
p2.z))*heuristicScale);
            if (hTargetNode != null) {
                h = System.Math.Max(h,
AstarPath.active.euclideanEmbedding.GetHeuristic(node.NodeIndex,
hTargetNode.NodeIndex));
            }
            return h;
        case Heuristic.DiagonalManhattan:
            Int3 p = GetHTarget() - node.position;
            p.x = System.Math.Abs(p.x);
            p.y = System.Math.Abs(p.y);
            p.z = System.Math.Abs(p.z);
            int diag = System.Math.Min(p.x, p.z);
            int diag2 = System.Math.Max(p.x, p.z);
            h = (uint)((((14*diag)/10) + (diag2-dia) +
p.y) * heuristicScale);
            if (hTargetNode != null) {
                h = System.Math.Max(h,
AstarPath.active.euclideanEmbedding.GetHeuristic(node.NodeIndex,
hTargetNode.NodeIndex));
            }
            return h;
    }
    return 0U;
}
public uint GetTagPenalty (int tag) {
    return (uint)internalTagPenalties[tag];
}
protected Int3 GetHTarget () {
    return hTarget;
}
public bool CanTraverse (GraphNode node) {
    if (traversalProvider != null)
        return traversalProvider.CanTraverse(this,
node);
    unchecked { return node.Walkable && (enabledTags
>> (int)node.Tag & 0x1) != 0; }
}
public uint GetTraversalCost (GraphNode node) {
#if ASTAR_NO_TRAVERSAL_COST
    return 0;
#else

```

```

        if (traversalProvider != null)
            return
        traversalProvider.GetTraversalCost(this, node);
        unchecked { return GetTagPenalty((int)node.Tag) +
node.Penalty; }
    #endif
}
    public virtual uint GetConnectionSpecialCost (GraphNode
a, GraphNode b, uint currentCost) {
        return currentCost;
    }
    public bool IsDone () {
        return PipelineState > PathState.Processing;
    }
    void IPathInternals.AdvanceState (PathState s) {
        lock (stateLock) {
            PipelineState
(PathState)System.Math.Max((int)PipelineState, (int)s);
        }
    }
    [System.Obsolete("Use the 'PipelineState' property
instead")]
    public PathState GetState () {
        return PipelineState;
    }
    public void FailWithError (string msg) {
        Error();
        if (errorLog != "") errorLog += "\n" + msg;
        else errorLog = msg;
    }
    [System.Obsolete("Use FailWithError instead")]
    protected void LogError (string msg) {
        Log(msg);
    }
    [System.Obsolete("Use FailWithError instead")]
    protected void Log (string msg) {
        errorLog += msg;
    }
    public void Error () {
        CompleteState = PathCompleteState.Error;
    }
    private void ErrorCheck () {
        if (!hasBeenReset) FailWithError("Please use the
static Construct function for creating paths, do not use the
normal constructors.");
    }

```

```

        if (((IPathInternals)this).Pooled)
FailWithError("The path is currently in a path pool. Are you
sending the path for calculation twice?");
        if (pathHandler == null) FailWithError("Field
pathHandler is not set. Please report this bug.");
        if (PipelineState > PathState.Processing)
FailWithError("This path has already been processed. Do not
request a path with the same path object twice.");
    }
    protected virtual void OnEnterPool () {
        if (vectorPath != null)
Pathfinding.Util.ListPool<Vector3>.Release(ref vectorPath);
        if (path != null)
Pathfinding.Util.ListPool<GraphNode>.Release(ref path);
        callback = null;
        immediateCallback = null;
        traversalProvider = null;
        pathHandler = null;
    }
    protected virtual void Reset () {
#if ASTAR_POOL_DEBUG
        pathTraceInfo = "This path was got from the pool
or created from here (stacktrace):\n";
        pathTraceInfo += System.Environment.StackTrace;
#endif
        if
(System.Object.ReferenceEquals(AstarPath.active, null))
            throw new System.NullReferenceException("No
AstarPath object found in the scene. " +
                "Make sure there is one or do not create
paths in Awake");
        hasBeenReset = true;
        PipelineState = (int)PathState.Created;
        releasedNotSilent = false;
        pathHandler = null;
        callback = null;
        immediateCallback = null;
        errorLog = "";
        completeState = PathCompleteState.NotCalculated;
        path =
Pathfinding.Util.ListPool<GraphNode>.Claim();
        vectorPath =
Pathfinding.Util.ListPool<Vector3>.Claim();
        currentR = null;
        duration = 0;
        searchedNodes = 0;
    }

```

```

        nnConstraint = PathNNConstraint.Default;
        next = null;
        heuristic = AstarPath.active.heuristic;
        heuristicScale = AstarPath.active.heuristicScale;
        enabledTags = -1;
        tagPenalties = null;
        pathID = AstarPath.active.GetNextPathID();
        hTarget = Int3.zero;
        hTargetNode = null;
        traversalProvider = null;
    }
    private List<System.Object> claimed = new
List<System.Object>();
    private bool releasedNotSilent;
    public void Claim (System.Object o) {
        if (System.Object.ReferenceEquals(o, null)) throw
new System.ArgumentNullException("o");
        for (int i = 0; i < claimed.Count; i++) {
            if (System.Object.ReferenceEquals(claimed[i],
o))
                throw new System.ArgumentException("You
have already claimed the path with that object ("++"). Are you
claiming the path with the same object twice?");
        }
        claimed.Add(o);
#if ASTAR_POOL_DEBUG
        claimInfo.Add(o.ToString() + "\n\nClaimed from:\n"
+ System.Environment.StackTrace);
#endif
    }
    [System.Obsolete("Use Release(o, true) instead")]
    internal void ReleaseSilent (System.Object o) {
        Release(o, true);
    }
    public void Release (System.Object o, bool silent =
false) {
        if (o == null) throw new
System.ArgumentNullException("o");
        for (int i = 0; i < claimed.Count; i++) {
            if (System.Object.ReferenceEquals(claimed[i],
o)) {
                claimed.RemoveAt(i);
#if ASTAR_POOL_DEBUG
                claimInfo.RemoveAt(i);
#endif
                if (!silent) {

```

```

        releasedNotSilent = true;
    }
    if (claimed.Count == 0 &&
releasedNotSilent) {
        PathPool.Pool(this);
    }
    return;
}

}
if (claimed.Count == 0) {
    throw new System.ArgumentException("You are
releasing a path which is not claimed at all (most likely it has
been pooled already). " +
        "Are you releasing the path with the
same object (" + o + ") twice?" +
        "\nCheck out the documentation on path
pooling for help.");
}
throw new System.ArgumentException("You are
releasing a path which has not been claimed with this object
(" + o + "). " +
        "Are you releasing the path with the same
object twice?\n" +
        "Check out the documentation on path pooling
for help.");
}
protected virtual void Trace (PathNode from) {
    PathNode c = from;
    int count = 0;
    while (c != null) {
        c = c.parent;
        count++;
        if (count > 16384) {
            Debug.LogWarning("Infinite loop? >16384
node path. Remove this message if you really have that long
paths (Path.cs, Trace method)");
            break;
        }
    }
    AstarProfiler.StartProfile("Check List
Capacities");
    if (path.Capacity < count) path.Capacity = count;
    if (vectorPath.Capacity < count)
vectorPath.Capacity = count;
    AstarProfiler.EndProfile();
    c = from;

```

```

        for (int i = 0; i < count; i++) {
            path.Add(c.node);
            c = c.parent;
        }
        int half = count/2;
        for (int i = 0; i < half; i++) {
            var tmp = path[i];
            path[i] = path[count-i-1];
            path[count - i - 1] = tmp;
        }
        for (int i = 0; i < count; i++) {
            vectorPath.Add((Vector3)path[i].position);
        }
    }

    protected void DebugStringPrefix (PathLog logMode,
System.Text.StringBuilder text) {
        text.Append(error ? "Path Failed : " : "Path
Completed : ");
        text.Append("Computation Time ");
        text.Append(duration.ToString(logMode ==
PathLog.Heavy ? "0.000 ms " : "0.00 ms "));
        text.Append("Searched          Nodes
").Append(searchedNodes);
        if (!error) {
            text.Append(" Path Length ");
            text.Append(path == null ? "Null" :
path.Count.ToString());
        }
    }

    protected void DebugStringSuffix (PathLog logMode,
System.Text.StringBuilder text) {
        if (error) {
            text.Append("\nError: ").Append(errorLog);
        }
        if (logMode == PathLog.Heavy &&
!AstarPath.active.IsUsingMultithreading) {
            text.Append("\nCallback references ");
            if (callback != null)
text.Append(callback.Target.GetType().FullName).AppendLine();
            else text.AppendLine("NULL");
        }
        text.Append("\nPath          Number
").Append(pathID).Append(" (unique id)");
    }

    protected virtual string DebugString (PathLog logMode)
{

```

```

        if (logMode == PathLog.None || (!error && logMode
== PathLog.OnlyErrors)) {
            return "";
        }
        System.Text.StringBuilder text =
pathHandler.DebugStringBuilder;
        text.Length = 0;
        DebugStringPrefix(logMode, text);
        DebugStringSuffix(logMode, text);
        return text.ToString();
    }
    protected virtual void ReturnPath () {
        if (callback != null) {
            callback(this);
        }
    }
    protected void PrepareBase (PathHandler pathHandler) {
        if (pathHandler.PathID > pathID) {
            pathHandler.ClearPathIDs();
        }
        this.pathHandler = pathHandler;
        pathHandler.InitializeForPath(this);
        if (internalTagPenalties == null ||
internalTagPenalties.Length != 32)
            internalTagPenalties = ZeroTagPenalties;
        try {
            ErrorCheck();
        } catch (System.Exception e) {
            FailWithError(e.Message);
        }
    }
    protected abstract void Prepare();
    protected virtual void Cleanup () {}
    protected abstract void Initialize();
    protected abstract void CalculateStep(long targetTick);
    PathHandler IPathInternals.PathHandler { get { return
pathHandler; } }
    void IPathInternals.OnEnterPool () { OnEnterPool(); }
    void IPathInternals.Reset () { Reset(); }
    void IPathInternals.ReturnPath () { ReturnPath(); }
    void IPathInternals.PrepareBase (PathHandler handler) {
PrepareBase(handler); }
    void IPathInternals.Prepare () { Prepare(); }
    void IPathInternals.Cleanup () { Cleanup(); }
    void IPathInternals.Initialize () { Initialize(); }

```

```

        void IPathInternals.CalculateStep (long targetTick) {
CalculateStep(targetTick); }
        string IPathInternals.DebugString (PathLog logMode) {
return DebugString(logMode); }
    }
    internal interface IPathInternals {
        PathHandler PathHandler { get; }
        bool Pooled { get; set; }
        void AdvanceState(PathState s);
        void OnEnterPool();
        void Reset();
        void ReturnPath();
        void PrepareBase(PathHandler handler);
        void Prepare();
        void Initialize();
        void Cleanup();
        void CalculateStep(long targetTick);
        string DebugString(PathLog logMode);
    }
}

```

ПРИЛОЖЕНИЕ А.6

Файл «PathUtilities.cs»

```
using Pathfinding.Util;
using System.Collections.Generic;
using UnityEngine;
namespace Pathfinding {
    public static class PathUtilities {
        public static bool IsPathPossible (GraphNode node1,
        GraphNode node2) {
            return node1.Walkable && node2.Walkable &&
            node1.Area == node2.Area;
        }
        public static bool IsPathPossible (List<GraphNode>
        nodes) {
            if (nodes.Count == 0) return true;
            uint area = nodes[0].Area;
            for (int i = 0; i < nodes.Count; i++) if
            (!nodes[i].Walkable || nodes[i].Area != area) return false;
            return true;
        }
        public static bool IsPathPossible (List<GraphNode>
        nodes, int tagMask) {
            if (nodes.Count == 0) return true;
            if (((tagMask >> (int)nodes[0].Tag) & 1) == 0)
            return false;
            if (!IsPathPossible(nodes)) return false;
            var reachable = GetReachableNodes(nodes[0],
            tagMask);
            bool result = true;
            for (int i = 1; i < nodes.Count; i++) {
                if (!reachable.Contains(nodes[i])) {
                    result = false;
                    break;
                }
            }
            ListPool<GraphNode>.Release(ref reachable);
            return result;
        }
        public static List<GraphNode> GetReachableNodes
        (GraphNode seed, int tagMask = -1, System.Func<GraphNode, bool>
        filter = null) {
            Stack<GraphNode> dfsStack =
            StackPool<GraphNode>.Claim();
            List<GraphNode> reachable =
            ListPool<GraphNode>.Claim();
```

```

var map = new HashSet<GraphNode>();
System.Action<GraphNode> callback;
if (tagMask == -1 && filter == null) {
    callback = (GraphNode node) => {
        if (node.Walkable && map.Add(node)) {
            reachable.Add(node);
            dfsStack.Push(node);
        }
    };
} else {
    callback = (GraphNode node) => {
        if ((node.Walkable && ((tagMask >>
(int)node.Tag) & 0x1) != 0 && map.Add(node)) {
            if (filter != null &&
!filter(node)) return;
            reachable.Add(node);
            dfsStack.Push(node);
        }
    };
}
callback(seed);
while (dfsStack.Count > 0) {
    dfsStack.Pop().GetConnections(callback);
}
StackPool<GraphNode>.Release(dfsStack);
return reachable;
}

static Queue<GraphNode> BFSQueue;
static Dictionary<GraphNode, int> BFSMap;
public static List<GraphNode> BFS (GraphNode seed, int
depth, int tagMask = -1, System.Func<GraphNode, bool> filter =
null) {
#if ASTAR_PROFILE
    System.Diagnostics.Stopwatch watch = new
System.Diagnostics.Stopwatch();
    watch.Start();
#endif

    BFSQueue = BFSQueue ?? new Queue<GraphNode>();
    var que = BFSQueue;
    BFSMap = BFSMap ?? new Dictionary<GraphNode,
int>();

    var map = BFSMap;
    que.Clear();
    map.Clear();
    List<GraphNode> result =
ListPool<GraphNode>.Claim();

```

```

        int currentDist = -1;
        System.Action<GraphNode> callback;
        if (tagMask == -1) {
            callback = node => {
                if (node.Walkable && !map.ContainsKey(node)) {
                    if (filter != null && !filter(node)) return;
                    map.Add(node, currentDist+1);
                    result.Add(node);
                    que.Enqueue(node);
                }
            };
        } else {
            callback = node => {
                if ((node.Walkable && ((tagMask >>
(int)node.Tag) & 0x1) != 0 && !map.ContainsKey(node)) {
                    if (filter != null && !filter(node)) return;
                    map.Add(node, currentDist+1);
                    result.Add(node);
                    que.Enqueue(node);
                }
            };
        }
        callback(seed);
        while (que.Count > 0) {
            GraphNode n = que.Dequeue();
            currentDist = map[n];
            if (currentDist >= depth) break;
            n.GetConnections(callback);
        }
        que.Clear();
        map.Clear();
#ifdef ASTAR_PROFILE
        watch.Stop();

        Debug.Log((1000*watch.Elapsed.TotalSeconds).ToString("0.0
ms"));
#endif

        return result;
    }

    public static List<Vector3> GetSpiralPoints (int count,
float clearance) {
        List<Vector3> pts =
ListPool<Vector3>.Claim(count);

```

```

        float a = clearance/(2*Mathf.PI);
        float t = 0;
        pts.Add(InvoluteOfCircle(a, t));
        for (int i = 0; i < count; i++) {
            Vector3 prev = pts[pts.Count-1];
            float d = -t/2 + Mathf.Sqrt(t*t/4 +
2*clearance/a);

            float mn = t + d;
            float mx = t + 2*d;
            while (mx - mn > 0.01f) {
                float mid = (mn + mx)/2;
                Vector3 p = InvoluteOfCircle(a, mid);
                if ((p - prev).sqrMagnitude <
clearance*clearance) {
                    mn = mid;
                } else {
                    mx = mid;
                }
            }
            pts.Add(InvoluteOfCircle(a, mx));
            t = mx;
        }
        return pts;
    }

    private static Vector3 InvoluteOfCircle (float a, float
t) {
        return new Vector3(a*(Mathf.Cos(t) +
t*Mathf.Sin(t)), 0, a*(Mathf.Sin(t) - t*Mathf.Cos(t)));
    }

    public static void GetPointsAroundPointWorld (Vector3
p, IRaycastableGraph g, List<Vector3> previousPoints, float
radius, float clearanceRadius) {
        if (previousPoints.Count == 0) return;
        Vector3 avg = Vector3.zero;
        for (int i = 0; i < previousPoints.Count; i++) avg
+= previousPoints[i];
        avg /= previousPoints.Count;
        for (int i = 0; i < previousPoints.Count; i++)
previousPoints[i] -= avg;
        GetPointsAroundPoint(p, g, previousPoints, radius,
clearanceRadius);
    }

    public static void GetPointsAroundPoint (Vector3
center, IRaycastableGraph g, List<Vector3> previousPoints, float
radius, float clearanceRadius) {

```

```

        if (g == null) throw new
System.ArgumentNullException("g");
        var graph = g as NavGraph;
        if (graph == null) throw new
System.ArgumentException("g is not a NavGraph");
        NNInfoInternal nn = graph.GetNearestForce(center,
NNConstraint.Default);
        center = nn.clampedPosition;
        if (nn.node == null) {
            return;
        }
        clearanceRadius *= clearanceRadius;
        for (int i = 0; i < previousPoints.Count; i++) {
            Vector3 dir = previousPoints[i];
            float magn = dir.magnitude;
            if (magn > 0) dir /= magn;
            dir *= newMagn;
            GraphHitInfo hit;
            int tests = 0;
            while (true) {
                Vector3 pt = center + dir;
                if (g.Linecast(center, pt, out hit)) {
                    if (hit.point == Vector3.zero) {
                        tests++;
                        if (tests > 8) {
                            previousPoints[i] = pt;
                            break;
                        }
                    } else {
                        pt = hit.point;
                    }
                }
            }
            bool worked = false;
            for (float q = 0.1f; q <= 1.0f; q +=
0.05f) {
                Vector3 qt = Vector3.Lerp(center,
pt, q);

                worked = true;
                for (int j = 0; j < i; j++) {
                    if ((previousPoints[j] -
qt).sqrMagnitude < clearanceRadius) {
                        worked = false;
                        break;
                    }
                }
                if (worked || tests > 8) {

```

```

        worked = true;
        previousPoints[i] = qt;
        break;
    }
}
if (worked) {
    break;
}
clearanceRadius *= 0.9f;
dir = Random.onUnitSphere *
Mathf.Lerp(newMagn, radius, tests / 5);
dir.y = 0;
tests++;
}
}

    public static List<Vector3> GetPointsOnNodes
(List<GraphNode> nodes, int count, float clearanceRadius = 0) {
    if (nodes == null) throw new
System.ArgumentNullException("nodes");
    if (nodes.Count == 0) throw new
System.ArgumentException("no nodes passed");
    List<Vector3> pts =
ListPool<Vector3>.Claim(count);
    clearanceRadius *= clearanceRadius;
    if (clearanceRadius > 0 || nodes[0] is
TriangleMeshNode
#if !ASTAR_NO_GRID_GRAPH
    || nodes[0] is GridNode
#endif
    ) {
        List<float> accs =
ListPool<float>.Claim(nodes.Count);
        float tot = 0;
        for (int i = 0; i < nodes.Count; i++) {
            var surfaceArea =
nodes[i].SurfaceArea();
            surfaceArea += 0.001f;
            tot += surfaceArea;
            accs.Add(tot);
        }
        for (int i = 0; i < count; i++) {
            int testCount = 0;
            int testLimit = 10;
            bool worked = false;
            while (!worked) {

```

```

        worked = true;
        if (testCount >= testLimit) {
            clearanceRadius *= 0.9f*0.9f;
            testLimit += 10;
            if (testLimit > 100)
                worked = false;
        }
        float tg = Random.value*tot;
        int v = accs.BinarySearch(tg);
        if (v < 0) v = ~v;
        if (v >= nodes.Count) {
            worked = false;
            continue;
        }
        var node = nodes[v];
        var p = node.RandomPointOnSurface();
        if (clearanceRadius > 0) {
            for (int j = 0; j < pts.Count; j++) {
                if ((pts[j]-p).sqrMagnitude < clearanceRadius) {
                    worked = false;
                    break;
                }
            }
            if (worked) {
                pts.Add(p);
                break;
            }
            testCount++;
        }
    }
    ListPool<float>.Release(ref accs);
} else {
    for (int i = 0; i < count; i++) {
        pts.Add((Vector3)nodes[Random.Range(0, nodes.Count)].RandomPointOnSurface());
    }
}
return pts;
}
}
}

```

ПРИЛОЖЕНИЕ Б

Презентация