

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение высшего
образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет «Вычислительная техника» (наименование)	Кафедра «Системы автоматизированного проектирования» (наименование)
Направление подготовки	09.03.01 «Информатика и вычислительная техника» (код и наименование)
Профиль	«Системы автоматизированного проектирования» (наименование)

БАКАЛАВРСКАЯ РАБОТА
на тему

Разработка группового поведения противников в играх
жанра RPG

Студент

 01.06.2017
(подпись, дата)

Сурков М.О.
(фамилия, инициалы)

Руководитель


(подпись, дата)

Эми В.В.
(фамилия, инициалы)

Нормоконтролёр

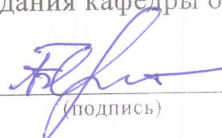
зав. каф. САГР ПГУ
(должность, место работы)

 5.06.14
(подпись, дата)

Бождай А.С.
(фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____ № _____)

Заведующий кафедрой


(подпись)

Бершаров Т.И.
(фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____ № _____)

Секретарь ГЭК

(подпись)

(фамилия, инициалы)

Пенза 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ	6
1.1. Описание предметной области	6
1.2. Описание аналогов	6
1.2.1. Diablo 3.....	7
1.2.2. S.T.A.L.K.E.R. Чистое небо.....	7
1.2.3. Warcraft 3.....	8
1.3. Сравнение аналогов.....	9
1.4. Выбор игрового движка	9
1.4.1. Unity	10
1.4.2. Unreal Engine.....	10
1.4.3. CryEngine	13
1.5. Выбор языка программирования	14
1.6. Обзор среды разработки.....	15
2 ПРОЕКТИРОВАНИЕ АЛГОРИТМОВ ГРУППОВОГО ПОВЕДЕНИЯ	17
2.1. Структура проекта.....	17
2.2. Проектирование алгоритмов поведения	18
2.3. Описание алгоритмов поведения	24
3 РАЗРАБОТКА АЛГОРИТМОВ ГРУППОВОГО ПОВЕДЕНИЯ.....	27
3.1. Описание основных структур	27
3.2. Описание основных классов	29
3.3. Контрольный пример	40
ЗАКЛЮЧЕНИЕ	45
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ	46
ПРИЛОЖЕНИЕ	48
ПРИЛОЖЕНИЕ А. ЛИСТИНГ	49

ВВЕДЕНИЕ

Компьютерные технологии развиваются всё быстрее и быстрее, параллельно внедряясь во все сферы жизни человека. Одной из них является досуг и развлечение. На данном этапе развития человечества, всё более популярным становится такой вид развлечения как компьютерные игры.

Компьютерные игры являются одной из актуальных проблем современной науки, так как представляют собой уникальный продукт развития техники и современной личности. Игры помогают смоделировать различные жизненные ситуации, проблемы и выдают некоторые возможные пути их решения.

Основной проблемой игровой индустрии на данный момент являются требования потребителей к нововведениям. Важной задачей в разработке компьютерных игр является разработка алгоритмов поведения противников, с которыми сталкивается игрок во время прохождения.

На сегодняшний день в играх жанра RPG мало времени уделяется разработке алгоритмов, при которых персонажи объединены в группы и могут взаимодействовать с миром и принимать решения как единое целое.

Целью выпускной квалификационной работы является разработка алгоритмов группового поведения противников в играх жанра RPG.

В связи с поставленной целью необходимо решить ряд задач:

- Исследовать предметную область и существующие аналоги программного продукта;
- Выявить их достоинства и недостатки;
- Составить алгоритмы группового поведения с учетом достоинств и недостатков существующих аналогов;
- Реализовать составленные алгоритмы;
- Обеспечить визуальное представление.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Описание предметной области

Выпускная квалификационная работа направлена на создание алгоритмов группового поведения противников в жанре RPG (англ. Role-playing game).

Компьютерной игрой называется компьютерная программа, служащая для организации игрового процесса, связи с партнёрами по игре, или сама выступающая в качестве партнёра [1].

RPG, или ролевая игра – жанр компьютерных игр, основанный на элементах игрового процесса традиционных настольных ролевых игр [2]. В ролевой игре игрок управляет одним или несколькими персонажами, каждый из которых описан набором численных характеристик, списком способностей и умений; примерами таких характеристик могут быть здоровье, показатели силы, ловкости, интеллекта, уровень развития того или иного навыка и т.п. В ходе игры они могут меняться. Одним из характерных элементов игрового процесса является повышение возможностей персонажей за счёт улучшения их параметров и изучения новых способностей.

1.2. Описание аналогов

При рассмотрении поведения противников в разных играх можно заметить, что алгоритмы поведения не закреплены за определенным жанром, поэтому для сравнения выбирались игры разных жанров, которые, тем не менее, имеют элементы RPG. Основными критериями для сравнения будут наличие группового поведения и сложность поведения в целом.

В качестве аналогов для сравнения были выбраны такие игры, как «Diablo 3», «S.T.A.L.K.E.R. Чистое небо» и «Warcraft 3».

1.2.1. Diablo 3.

Игра жанра Action RPG, разработанная студией Blizzard Entertainment и выпущенная в 2012 году [3]. В игре присутствуют элементы группового поведения. Противники объединяются в группы, при этом в группе может присутствовать персонаж, способный лечить и воскрешать союзников. Если в группе осталось мало членов, то они стараются убежать от игрока. В целом поведение достаточно примитивное, и враги нападают на врага сразу, как только он появляется в зоне их видимости. На рисунке 1 представлен бой между игроком и группой противников.



Рисунок 1 – Игровой процесс Diablo 3

1.2.2. S.T.A.L.K.E.R. Чистое небо.

Игра жанра FPS (англ. First Person Shooter – Шутер от первого лица) с элементами RPG, разработанная студией GSC Game World и выпущенная в 2008 году [4]. В этой игре противников можно разделить на 2 типа: люди и монстры (в т. ч. и человекоподобные). Сильные монстры ходят, в основном, по одиночке, слабые же (например, собаки или кабаны) стараются собраться в стаю (то есть в группу). В целом, монстры всегда агрессивны по отношению к игроку. На рисунке 2 представлена атака группы монстров на

игрока. Люди почти всегда объединены в группы, при этом каждая группа принадлежит определенной фракции. Отношение группы к игроку полностью определяется отношениями игрока с фракцией в целом.



Рисунок 2 – Игровой процесс S.T.A.L.K.E.R. Чистое небо

1.2.3. Warcraft 3.

Игра жанра RTS (англ. Real-Time Strategy – Стратегия в реальном времени) с элементами RPG, разработанная студией Blizzard Entertainment и выпущенная в 2003 году [5]. Подход к организации поведения в этой игре немного отличается, с учетом основного жанра игры. Здесь присутствует система, координирующая поведение всех групп и отдельных противников. Это является как преимуществом, так и недостатком игры. С одной стороны создается иллюзия, что всех противников контролирует один компьютерный игрок, и это добавляет сложности в их поведение. С другой же стороны каждая группа не может принимать решения самостоятельно и всегда знает о состоянии любой другой группы. На рисунке 3 представлен игровой процесс Warcraft 3.



Рисунок 3 – Игровой процесс Warcraft 3

1.3. Сравнение аналогов

Так как большинство разработчиков не выкладывают исходный код своих игр, это затрудняет их сравнение. Все выводы сделаны на основе визуального анализа игр, рассмотренных выше.

Проведя сравнение аналогов можно выделить несколько недостатков, которые будут устранены в разрабатываемой игре:

- Простота поведения – в большинстве случаев враги сразу нападают на игрока, не оценивая свои риски и возможности
- Контроль группы извне – группы контролирует единая система, которая не дает самой группе решать, что ей делать, и позволяет получить информацию, которую группа не должна знать.

1.4. Выбор игрового движка

При создании любого проекта в первую очередь необходимо определиться с рабочим инструментом. Для игр такой инструмент называется игровым движком. Он или разрабатывается самостоятельно, или

лицензируется. Так как написание игрового движка достаточно сложный и трудоемкий процесс, для разрабатываемой игры было решено выбрать готовый инструментарий.

Далее приведен список игровых движков, среди которых велся выбор.

1.4.1. Unity

Unity — это инструмент для разработки двух и трёхмерных приложений и игр, работающий под операционными системами Windows, Linux и OS X. Созданные с помощью Unity приложения работают под операционными системами Windows, OS X, Windows Phone, Android, Apple iOS, Linux, а также на игровых приставках Wii, PlayStation 3, PlayStation 4, Xbox 360, Xbox One. Есть возможность создавать приложения для запуска в браузерах с помощью специального подключаемого модуля Unity (Unity Web Player), а также с помощью реализации технологии WebGL.

Приложения, созданные с помощью Unity, поддерживают DirectX и OpenGL. Активно движок используется как крупными разработчиками, так и разработчиками Indie-игр в силу наличия бесплатной версии, удобного интерфейса и простоты работы с движком.

Редактор Unity имеет простой Drag&Drop интерфейс, который легко настраивать. Он состоит из различных окон, благодаря чему можно производить отладку игры прямо в редакторе. Движок поддерживает два сценарных языка: C#, JavaScript (модификация). Расчёты физики производит физический движок PhysX от NVIDIA [6].

На рисунке 4 представлен интерфейс Unity.

1.4.2. Unreal Engine

Unreal Engine — игровой движок, разрабатываемый и поддерживаемый компанией Epic Games.

Первая игра, созданная на этом движке появилась в 1998 году. С тех пор различные версии движка были использованы в более чем сотне игр и других проектов.

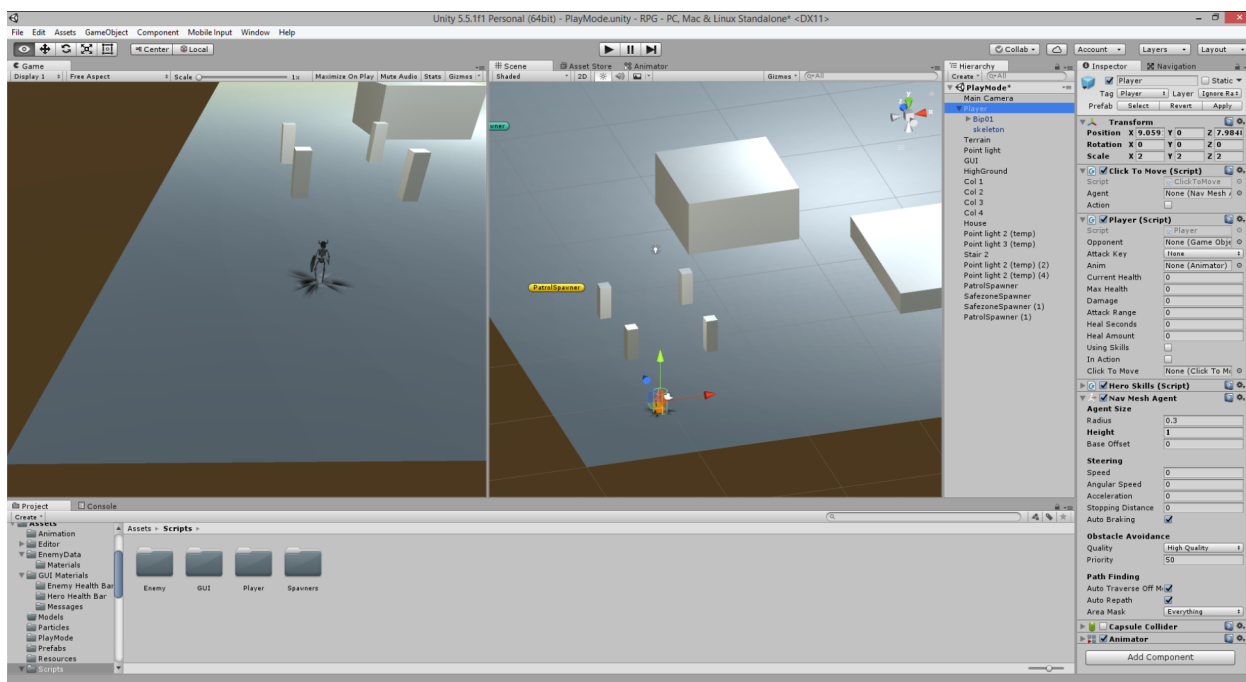


Рисунок 4 – Интерфейс Unity

Движок написан на языке C++ и позволяет создавать игры для большинства операционных систем и платформ: Microsoft Windows, Linux, MacOS и MacOSX; консолей Xbox, Xbox 360, XboxOne, PlayStation 2, PlayStation3, PlayStation 4, PSVita, Wii и др., а также на различных портативных устройствах, например, устройствах Apple, управляемых iOS и прочих [7].

Для игры по сети поддерживаются технологии Windows Live, Xbox Live и прочие, включая до 64 игроков (клиентов) одновременно.

Все элементы игрового движка представлены в виде объектов, имеющих набор характеристик, и класса, который определяет доступные характеристики. В свою очередь, любой класс является «дочерним» классом object.

Метод управления описан специальным объектом, такой объект называется контроллером. Контроллер описывает лишь общее поведение пешки во время игрового процесса, а такие параметры как «здоровье» (количество повреждений, после которых пешка перестаёт функционировать)

или, например, расстояние, на котором пешка обращает внимание на звуки, задаются для каждого объекта отдельно.

Мир, уровень — объект, характеризующий общие свойства «пространства», например, силу тяжести и туман, в котором располагаются все актёры. Также может содержать в себе параметры игрового процесса, как, например, игровой режим, для которого предназначен уровень.

Для работы с простыми и, как правило, неподвижными элементами игрового пространства (например, стены) используется двоичное разбиение пространства — всё пространство делится на «заполненное» и «пустое». В «пустой» части пространства располагаются все объекты, а также только в ней может находиться «точка наблюдения» при отрисовке сцены. Возможность полного или частичного помещения объектов в «заполненную» часть пространства не исключается, однако может привести к неправильной обработке таких объектов (например, расчёт физического взаимодействия) или неправильной отрисовки в случае помещения туда «точки наблюдения» (например, эффект «зала зеркал»). Все пешки, попадающие в «заполненную» часть пространства, сразу «погибают».

На рисунке 5 представлен интерфейс Unreal Engine.

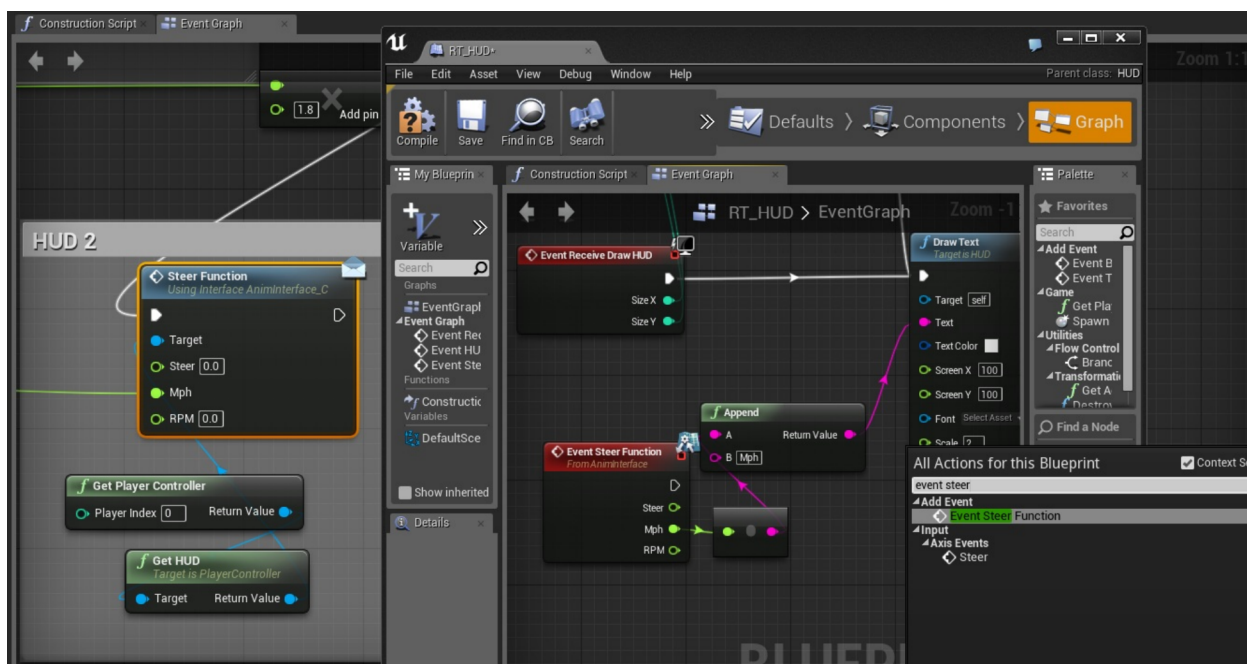


Рисунок 5 – Интерфейс Unreal Engine

1.4.3. CryEngine

CryEngine 3 — игровой движок, разработанный немецкой компанией с ограниченной ответственностью. CryEngine изначально является кроссплатформенным языком — он ориентирован на персональные компьютеры и игровые консоли Microsoft Xbox 360 и Sony PlayStation 3, а также на их последующие версии.

CryEngine является достаточно популярным игровым движком, о чем свидетельствует количество «больших игр» созданных на нем: Giant, Sniper II: Ghost Warrior, Cabal II, Far Cry, Ryse: Son of Rome. Он привлек к себе внимание разработчиков после выхода Crysis. Эта игра продемонстрировала новый уровень графики благодаря CryEngine. Позже, были выпущены несколько версий CryEngine с поддержкой новых функциональных возможностей [8].

На рисунке 6 представлен интерфейс CryEngine.

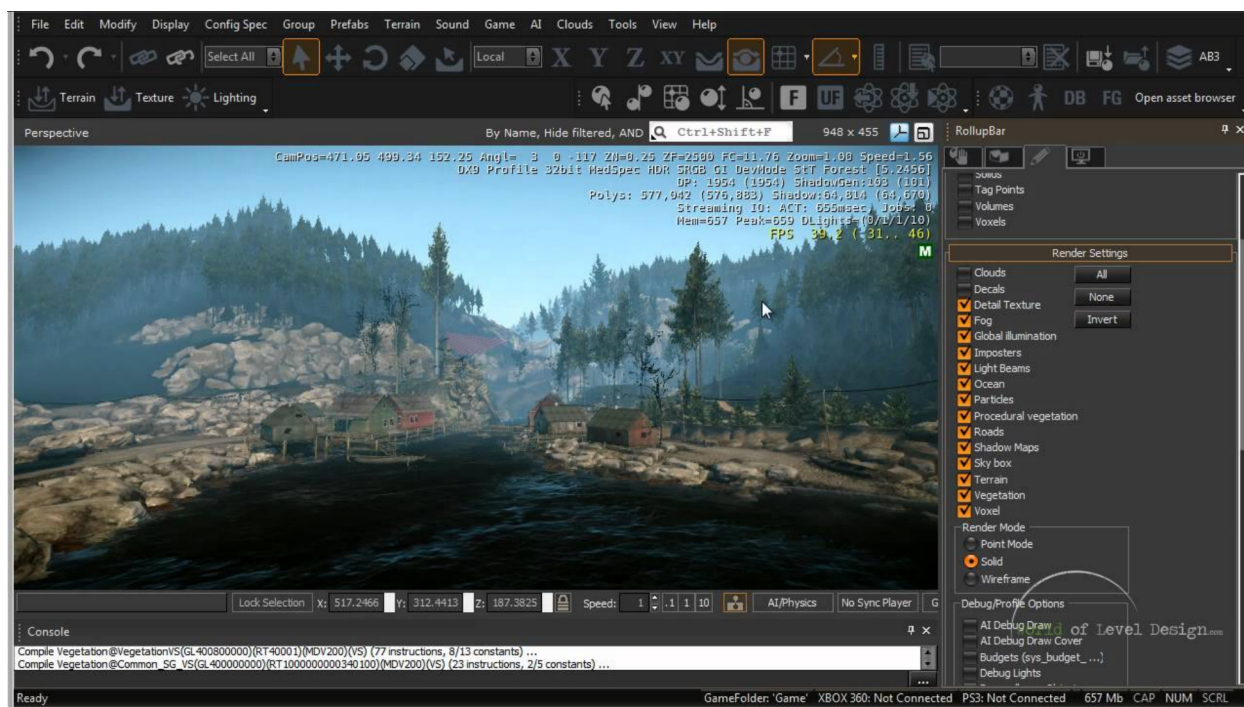


Рисунок 6 – Интерфейс CryEngine

В итоге выбор был сделан в пользу Unity. Он хорошо подходит для реализации небольших проектов, а также имеет простой и удобный интерфейс.

1.5. Выбор языка программирования

После выбора игрового движка необходимо выбрать язык программирования, на котором будет вестись разработка.

Так как Unity предоставляет возможность программирования только на языках C# и JavaScript, среди них и велся выбор.

Язык программирования C# – это высокоуровневый язык программирования, разработанный и до сих пор активно поддерживаемый компанией Microsoft [9]. C# на сегодняшний день является одним из самых популярных языков для написания программ.

В основе написания программ на этом языке лежит объектно-ориентированный подход к программированию. Его суть заключается в том, что программа представляется в виде совокупности объектов, для описания которых используются классы – структуры, имеющие высокий уровень абстракции [10].

Программа, написанная на объектно-ориентированном языке C#, является описанием набора классов исследуемой предметной области.

JavaScript — высокоуровневый язык программирования. Обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

На JavaScript оказали влияние многие языки, при разработке была цель сделать язык похожим на Java, но при этом лёгким для использования непрограммистами. Языком JavaScript не владеет какая-либо компания или организация, что отличает его от ряда языков программирования, используемых в веб-разработке [11].

При выборе языка программирования основным критерием стало наличие опыта разработки на этом языке. В результате был выбран язык C#.

1.6. Обзор среды разработки

В качестве среды программирования была выбрана Visual Studio 2015, созданная компанией Microsoft.

Среда включает интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Она позволяет разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, а также веб-сайты, веб-приложения, веб-службы для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight [12].

Среда содержит обширный инструментарий для работы с множеством языков программирования, включая C#, JavaScript, C++ и др. Имеет возможность подключать сторонние библиотеки и работать с ними. Обладает возможностью интеграции с Unity.

Интерфейс Visual Studio 2015 Community представлен на рисунке 7.

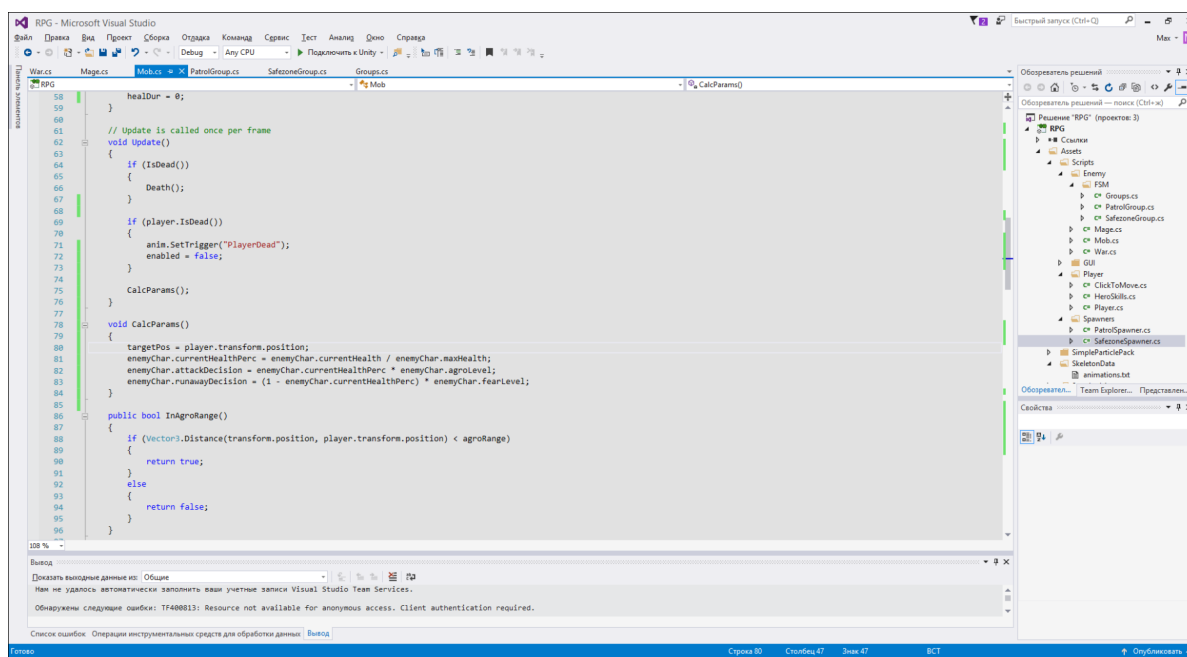


Рисунок 7 – Интерфейс Visual Studio 2015

В результате анализа предметной области было произведено сравнение аналогов игр. Были выделены недостатки в алгоритмах поведения

противников, которые будут устранены в этой работе. Разработка будет вестись на игровом движке Unity с помощью языка C# в интеграционной среде разработки Visual Studio 2015.

2 ПРОЕКТИРОВАНИЕ АЛГОРИТМОВ ГРУППОВОГО ПОВЕДЕНИЯ

2.1. Структура проекта

Перед тем, как реализовать алгоритмы необходимо разобраться с общей структурой приложения.

В основе всего в Unity лежит Игровой объект (Game Object). Их можно представить, как физические объекты из реального мира. Объекты находятся на сцене. Сцена, по сути, является игровым миром, заполненным объектами. На сцену могут попасть два идентичных объекта, которые при этом будут отличаться поведением или событиями, происходящими с ними. Впрочем, сцена не является единственной на всю игру. В любой момент мы можем выгрузить из памяти текущую сцену и загрузить новую. Однако объекты по большому счету ничего из себя не представляют. Как и сцена, они являются лишь контейнерами для компонентов.

Компонентов существует много. Одни компоненты отвечают за отображение анимаций объекта, другие могут воспроизводить звук. Также к компонентам относятся скрипты. Скрипты позволяют контролировать поведение объекта, активировать игровые события, изменять параметры компонентов, и отвечать на ввод пользователя указанным способом [13].

Скрипт взаимодействует с внутренними механизмами Unity за счет создания класса, наследованного от встроенного класса, называемого MonoBehaviour. Каждый раз, когда скриптовый компонент присоединяется к игровому объекту, создается новый экземпляр объекта. Имя класса берется из имени, которое указано при создании скрипта. Имя класса и имя файла должны быть одинаковыми, для того, чтобы скриптовый компонент мог быть присоединен к игровому объекту.

В каждом скрипте по умолчанию определены 2 основные функции. Функция Start() будет вызвана Unity до начала игрового процесса. В ней происходит инициализация необходимых переменных. Функция Update() – это место для размещения кода, который будет обрабатывать обновление кадра для игрового объекта. Это может быть движение, срабатывание действий и ответная реакция на ввод пользователя, то есть почти всё, что должно быть обработано с течением времени в игровом процессе.

2.2. Проектирование алгоритмов поведения

Каждый противник в игре будет появляться в определенной локации. Все противники, появившиеся в одной локации, будут записываться в одну группу. При этом будет создаваться пустой игровой объект. В нем будет храниться информация о созданной группе и алгоритм, контролирующий поведение группы. Информация о группе представлена в виде структуры данных, хранящей такие данные, как общая информация о каждом члене группы, уровень агрессии и уровень страха группы, суммарные максимальный и текущий уровни здоровья группы, координаты возрождения группы.

В разрабатываемой игре будет два основных типа групп. Первые будут патрулировать территорию, а вторые будут охранять безопасные зоны, в которые, при необходимости, будут отступать патрульные группы. Также будут отдельные противники-одиночки. Они будут заниматься разведкой.

В состав каждой группы могут входить 2 типа противников: воин («War») и маг («Mage»). Воин сражается в ближнем бою и имеет больше здоровья. Маг же является бойцом дальнего боя и занимается поддержкой группы. Поддержка заключается в отслеживании уровней здоровья всех членов группы и их лечении при необходимости.

Общий состав и список действий всех групп представлен на рисунке 8.

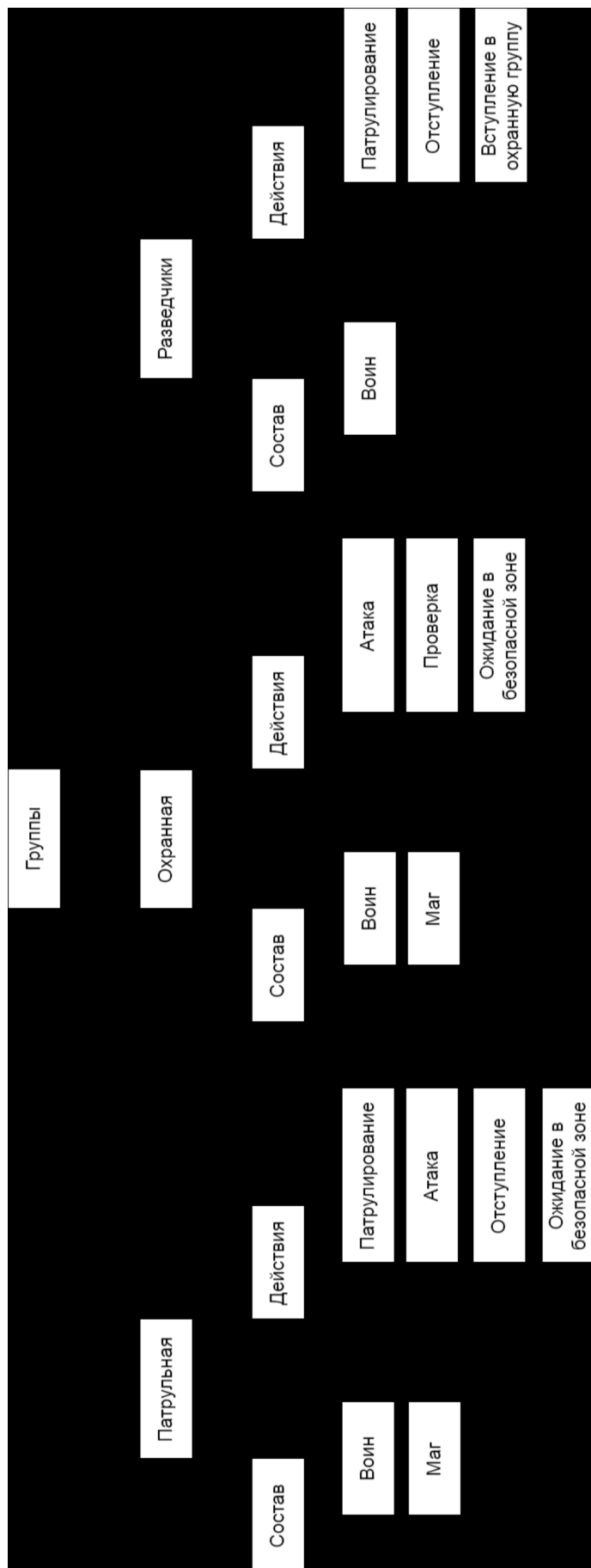


Рисунок 8 – Состав и список действий групп

Алгоритмы, контролирующие оба типа групп и отдельных противников, будут представлены в виде конечных автоматов. Под конечным автоматом понимается абстрактное устройство, имеющее конечное число состояний в любой данный момент времени. То есть любой объект, контролируемый конечным автоматом, может находиться только в одном состоянии и выполнять одни и те же действия, связанные с этим состоянием. Состояния соединены друг с другом с помощью переходов, каждый из которых имеет определенный набор условий, необходимых для перехода. И если все условия выполнены, то происходит переход в указанное состояние [14].

На рисунках 9 и 10 представлены диаграммы состояний патрульной и охранной группы. Диаграммы выполнены на языке графического описания UML.

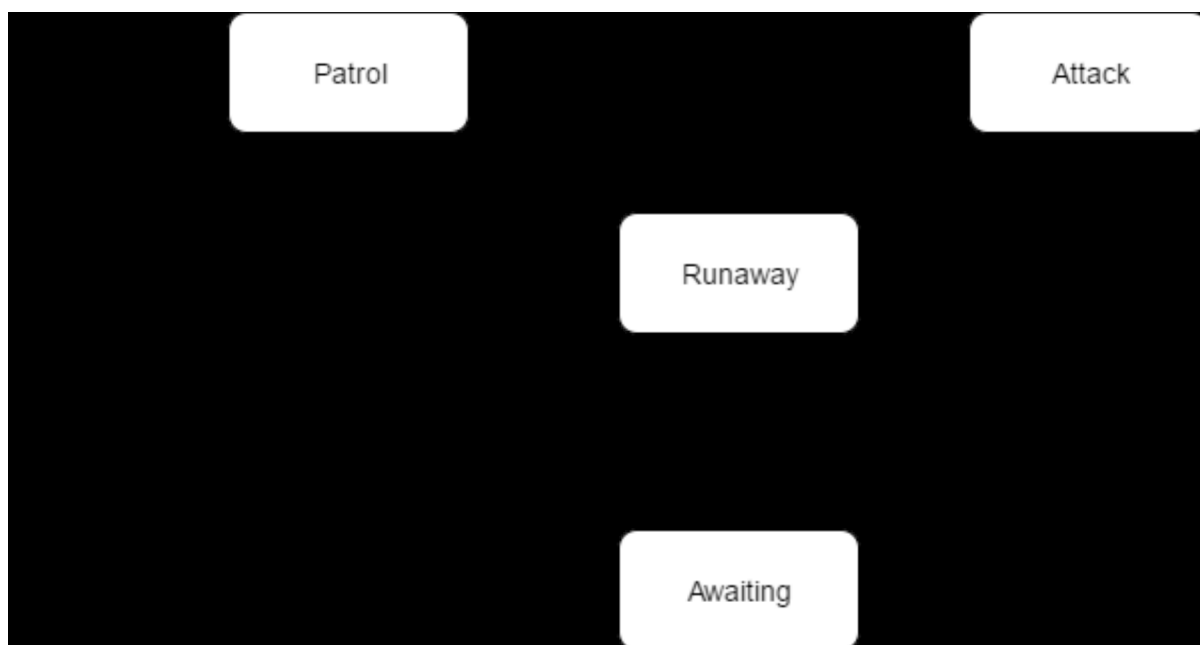


Рисунок 9 – Диаграмма состояний патрульной группы



Рисунок 10 – Диаграмма состояний охранной группы

Для каждого из переходов создана отдельная функция, проверяющая, выполняются ли условия перехода.

Патрульная группа может находиться в одном из четырех состояний:

- 1) Патрулирование («Patrol») – группа получает набор координат, между которыми она перемещается во время патрулирования. Координаты выбираются случайно. Также группа может остановиться в одной из точек на «отдых»;
- 2) Атака («Attack») – группа атакует игрока;
- 3) Отступление («Runaway») – группа отступает к ближайшей безопасной зоне (координаты всех таких зон известны);
- 4) Ожидание («Awaiting») – после отступления группа некоторое время будет находиться в безопасной зоне, не возвращаясь к патрулированию; если игрок решит напасть на безопасную зону, патрульная группа будет помогать охранной уничтожить игрока.

Рассмотрим подробнее условия перехода между состояниями в патрульной группе.

- 1) Переходы между состоянием патрулирования и атаки

Для перехода от патрулирования к атаке сначала проверяется, вошел ли игрок в зону видимости одного из членов группы, и если вошел, то производятся 2 проверки: хочет ли группа в целом нападать на игрока и хочет ли нападать отдельный член группы. Если группа в целом или более

50% членов группы хотят напасть на игрока, то происходит переход в состояние атаки. Также переход случится в любом случае, если группа будет атакована игроком.

Решение о нападении как для отдельного члена, так и для группы, вычисляется по следующей формуле:

$$D_A = H_{Cur} * L_A, \quad (1)$$

где D_A – решение о нападении,

H_{Cur} – текущее здоровье члена группы / группы в процентах,

L_A – уровень агрессии члена группы / группы,

Уровень агрессии для каждого члена группы задается случайно в момент его создания в диапазоне от 0.1 до 2.

Для группы высчитывается средний уровень агрессии всех членов группы и добавляется фиксированная величина в зависимости от текущего размера группы.

Если значение D_A выше 0.5, то член группы / группа хотят напасть.

Для перехода от атаки к патрулированию, необходимо, чтобы игрок вышел из зоны видимости всех членов группы.

2) Переходы к состоянию отступления

Оба перехода похожи в своей логике: проверяется хочет ли группа в целом отступить и хочет ли отступить отдельный член группы. Если группа в целом или более 60% членов группы хотят отступить, то происходит переход в состояние отступления.

Решение об отступлении как для отдельного члена, так и для группы, вычисляется по следующей формуле:

$$D_F = (1 - H_{Cur}) * L_F, \quad (2)$$

где D_F – решение об отступлении,

H_{Cur} – текущее здоровье члена группы / группы в процентах,

L_F – уровень страха члена группы / группы,

Уровень страха для каждого члена группы задается случайно в момент его создания в диапазоне от 0 до 2.

Для группы высчитывается средний уровень страха всех членов группы и отнимается фиксированная величина в зависимости от текущего размера группы.

Если значение D_F выше 0.5, то член группы / группа хотят отступить.

3) Переход из состояния побега в состояние ожидания

Переход происходит как только группа добирается до безопасной зоны. Во время этого перехода на некоторое время блокируется возможность вернуться к патрулированию, чтобы группа могла переждать опасность и восстановить силы.

4) Переходы из состояния ожидания

Переход в состояние атаки происходит, как только один из членов группы заметит игрока. Если возможность вернуться к патрулированию заблокирована, то группа вернется к состоянию ожидания, как только все члены группы потеряют игрока из зоны видимости.

Возвращение к патрулированию произойдет, когда закончится блокировка.

Охранная группа имеет три состояния:

- 1) Ожидание («Awaiting») – группа находится в безопасной зоне и ожидает появления нарушителей;
- 2) Атака («Attack») – группа атакует игрока.
- 3) Проверка («Check») – после вступления в группу разведчика, группа проверяет место, где разведчик последний раз видел игрока

Рассмотрим подробнее условия перехода между состояниями в патрульной группе.

1) Переходы между состояниями атаки и ожидания

Переход в состояние атаки происходит, как только один из членов группы заметит игрока. Обратный переход в состояние ожидания произойдет, когда все члены группы потеряют игрока из зоны видимости.

2) Переходы состояния проверки

Когда к группе присоединяется разведчик, он указывает группе место, где он нашел игрока, и группа отправляется проверить это место.

Если группа заметит игрока, то она нападет, если нет – то через некоторое время вернется к охраняемой зоне.

Противники-разведчики управляются так же, как и группы и имеют три состояния:

- 1) Патрулирование («Patrol») – разведчик имеет набор координат, по которым он случайным образом перемещается
- 2) Отступление («Runaway») – разведчик отступает к ближайшей безопасной зоне
- 3) Вступление в группу («Entering») – разведчик присоединяется к группе безопасной зоны.

Здесь всего два перехода. Переход от патрулирования к отступлению происходит, когда разведчик заметит игрока. Переход на вступление в группу происходит, когда разведчик добирается до охраняемой зоны.

2.3. Описание алгоритмов поведения

Общее поведение патрульной группы будет выглядеть следующим образом:

- Группа появляется и начинает патрулировать территорию
- Если кто-то из членов группы видит игрока, происходит проверка, хочет ли группа, на него нападать

- Если проверка дала положительный результат, то группа нападает игрока, в противном случае продолжает патрулировать территорию
- Если игрок сам атаковал группу, группа будет вынуждена атаковать игрока
- Во время всех вышеперечисленных действий группа, группа проверяет, нужно ли ей отступить, и если проверка дает положительный результат, то группа отступает к ближайшей безопасной зоне
- Находясь в безопасной зоне, группа будет ждать возможности вернуться к патрулированию
- Если группа, находясь в безопасной зоне, увидит игрока, она нападет на него.

Общее поведение охранной группы:

- Охранная группа появляется в безопасной зоне и ожидает появления игрока или разведчика
- Если появляется игрок, то группа нападает. Как только группа потеряет игрока из виду, она возвращается в безопасную зону
- Если появляется разведчик, группа добавляет его в свой состав и идет проверять место, где разведчик видел игрока последний раз
- Если группа встречает игрока, то нападает на него
- Если группа не встретила игрока, то через некоторое время она возвращается к безопасной зоне.

Общее поведение разведчика:

- Разведчик появляется и начинает патрулировать территорию
- Как только разведчик встречает игрока, он отступает к ближайшей безопасной зоне
- Когда разведчик добирается до безопасной зоны, он присоединяется к охранной группе и передает координаты места,

где он видел игрока последний раз. При вступлении в группу уничтожается игровой объект, контролировавший поведение разведчика до этого.

В результате проектирования были выделены группы, их состав и список действий. Для каждой группы были описаны алгоритмы их поведения. Алгоритмы представлены в виде конечных автоматов. Для каждого конечного автомата описаны состояния и условия перехода.

3 РАЗРАБОТКА АЛГОРИТМОВ ГРУППОВОГО ПОВЕДЕНИЯ

3.1. Описание основных структур

Для хранения необходимой информации о самих противниках и о группах были созданы 2 структуры данных, показанные на рисунках 11 и 12.

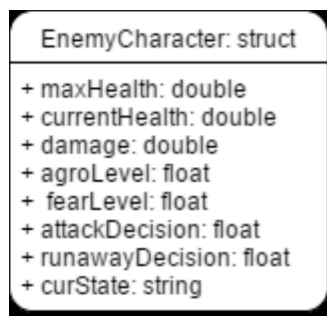


Рисунок 11 – Структура данных противника

Структура хранит данные об отдельном противнике.

Публичные атрибуты:

- double maxHealth – максимальное здоровье;
- double currentHealth – текущее здоровье;
- double damage – урон, наносимый противником;
- float agroLevel – уровень агрессии;
- float fearLevel – уровень страха;
- float attackDecision – решение об атаке, принимается на основе текущего уровня здоровья и уровня агрессии;
- float runawayDecision – решение об отступлении, принимается на основе текущего здоровья и уровня страха;
- string curState – текущее состояние.

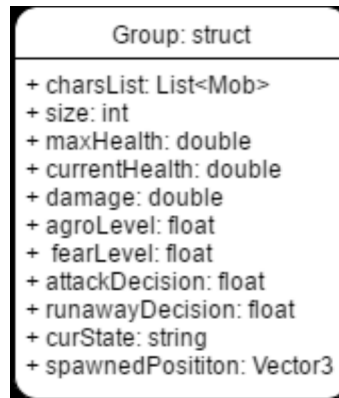


Рисунок 12 – Структура данных группы

Структура хранит данные об отдельной группе.

Публичные атрибуты:

- List<Mob> charsList – список, хранящий данные о всех членах группы
- int size – размер группы;
- double maxHealth – максимальное суммарное здоровье;
- double currentHealth – текущее суммарное здоровье;
- double damage – средний урон, наносимый группой;
- float agroLevel – уровень агрессии, вычисляется на основе агрессии каждого члена группы;
- float fearLevel – уровень страха, вычисляется на основе страха каждого члена группы;
- float attackDecision – решение об атаке, принимается на основе текущего уровня здоровья и уровня агрессии, также принимается в расчет решение об атаке каждого члена группы;
- float runawayDecision – решение об отступлении, принимается на основе текущего здоровья и уровня страха, также принимается в расчет решение об отступлении каждого члена группы;
- string curState – текущее состояние группы;
- Vector3 spawnedPosition – место создания группы.

3.2. Описание основных классов

На рисунке 13 представлена диаграмма классов, в которой указаны все классы, необходимые для создания противников и управления ими.

Рассмотрим каждый класс подробнее, с описанием всех атрибутов и методов.

Класс Mob – класс, который хранит всю информацию об отдельном противнике.

Публичные атрибуты:

- Player player – хранит информацию о персонаже игрока;
- Animator anim – используется для контроля анимаций;
- NavMeshAgent agent – используется для контроля передвижей. Можно задать скорость передвижения, скорость поворота, и координаты для перемещения;
- EnemyCharacter enemyChar – структура, которая хранит информацию, необходимую для принятия решений
- float attackRange – дистанция, с которой противник может атаковать игрока;
- float agroRange – дистанция, с которой противник замечает игрока;
- int walkSpeed – скорость противника во время патрулирования;
- int actionSpeed – скорость противника во время боевых действий;
- bool isRunawayBlocked – используется для блокировки побега.

Приватные методы:

- void Start() – используется для инициализации переменных;
- void Update() – используется для обновления кадра игрового объекта;
- void Death() – выполняет уничтожение объекта, когда его здоровье достигает нуля;

- void OnMouseOver() – срабатывает, когда игрок наводит мышку на данный объект. Необходим для выделения цели игрока;
- void CalcParams() – высчитывает параметры, такие как соотношение среднего и максимального здоровья, решение атаки и решение отступления.

Публичные методы:

- bool InAgroRange() – проверяет, зашел ли игрок, в зону видимости;
- bool InAttackRange() – проверяет, достаточно ли близко игрок для атаки;
- void SendDamage() – метод, вызывающийся во время анимации атаки. Используется для нанесения урона игроку;
- void Follow(Vector3) – метод, который используется для перемещения противника а указанную точку;
- void EndFollow() – метод, который используется для остановки противника;
- void Restoration() – противник будет постепенно восстанавливать здоровье, если группа находится в безопасной зоне
- bool IsDead() – метод проверяет, достигло ли текущее здоровья противника нуля;
- void GetHit(double damage) – метод, который используется для получения урона от игрока.

Классы War и Mage похожи друг на друга. Они задают основные параметры, такие как здоровье, урон, уровни агрессии и страха, для 2 типов врагов, представленных в игре: воина и мага соответственно.

Класс War

Публичные атрибуты:

- Mob mob – хранит информацию об отдельном противнике. С помощью и будут задаваться основные параметры.

Приватные методы:

- void Start() – используется для инициализации переменных;
- void Update() – используется для обновления кадра игрового объекта;

Публичные методы:

- void Attack() – набор действий, которые выполняет противник во время атаки игрока.

Класс Mage

Публичные атрибуты:

- Mob mob – хранит информацию об отдельном противнике. С помощью и будут задаваться основные параметры.

Приватные методы:

- void Start() – используется для инициализации переменных;
- void Update() – используется для обновления кадра игрового объекта;
- void Heal() – используется во время атаки для восстановления здоровья членам группы;

Публичные методы:

- void Attack() – набор действий, которые выполняет противник во время атаки игрока.
- void RegainHealth() – используется для восстановления здоровья членов группы в безопасной зоне.

Класс Groups – содержит описание структуры Group и список координат безопасных зон

Публичные атрибуты:

- struct Group – описывает структуру данных группы
- static List<Vector3> safezonePoints – содержит координаты безопасных зон

- static Dictionary<Vector3, GameObject> - содержит координаты безопасных зон, с привязкой к игровым объектам, контролирующим группы в соответствующих зонах

Классы PatrolSpawner и SafezoneSpawner отвечают за создание противников и объединение их в группы.

Класс PatrolSpawner – создает патрульные группы.

Приватные атрибуты:

- GameObject enemy – противник, который будет создан;
- float spawnRadius – радиус локации, в которой могут появиться противники;
- Vector3 spawnPos – координаты внутри локации, в которых появится очередной противник;
- int magesNum – текущее количество появившихся противников типа маг
- int magesSize - максимальное количество противников типа маг, которые могут появиться
- int chance – шанс, с которым появляется противник типа маг

Публичные атрибуты:

- GameObject mobWar – игровой объект, содержащий модель и информацию о противнике типа воин;
- GameObject mobMage – игровой объект, содержащий модель и информацию о противнике типа маг;
- Group group – структура, в которую добавляются все созданные противники. После появления всех противников, передается объекту, контролирующему созданную группу;
- Mob mob – хранит информацию, об отдельном противнике. Этот скрипт будет записываться в список членов группы;
- GameObject groupGO – объект, который будет алгоритмы поведения группы;

- PatrolGroup patrolGroup – скрипт, в котором описано поведение патрульной группы;
- int squadSize – количество противников в группе.

Приватные методы:

- void Awake() – служит для инициализации переменных;
- void Spawn() – определяет состав группы и создание противников;
- void InstUnit(GameObject enemy, Vector3 spawnPos) – вызывается внутри метода Spawn() и отвечает непосредственно за создание и размещение противника, а так же добавляет информацию о нем в список членов группы.

Класс SafezoneSpawner – создает охранные группы. Также координаты игрового объекта, к которому привязан этот скрипт, записываются в список координат безопасных зон.

Приватные атрибуты:

- GameObject enemy – противник, который будет создан;
- float spawnRadius – радиус локации, в которой могут появиться противники;
- Vector3 spawnPos – координаты внутри локации, в которых появится очередной противник;
- int magesNum – текущее количество появившихся противников типа маг
- int magesSize - максимальное количество противников типа маг, которые могут появиться
- int chance – шанс, с которым появляется противник типа маг

Публичные атрибуты:

- GameObject mobWar – игровой объект, содержащий модель и информацию о противнике типа воин;

- `GameObject mobMage` – игровой объект, содержащий модель и информацию о противнике типа маг;
- `Group group` – структура, в которую добавляются все созданные противники. После появления всех противников, передается объекту, контролирующему созданную группу;
- `Mob mob` – хранит информацию, об отдельном противнике. Этот скрипт будет записываться в список членов группы;
- `GameObject groupGO` – объект, который будет алгоритмы поведения группы;
- `SafezoneGroup safezoneGroup` – скрипт, в котором описано поведение охранной группы;
- `int squadSize` – количество противников в группе.

Приватные методы:

- `void Awake()` – служит для инициализации переменных;
- `void Spawn()` – определяет состав группы и создание противников;
- `void InstUnit(GameObject enemy, Vector3 spawnPos)` – вызывается внутри метода `Spawn()` и отвечает непосредственно за создание и размещение противника, а так же добавляет информацию о нем в список членов группы.

Класс `ScoutSpawner` – создает разведчиков.

Публичные атрибуты:

- `GameObject mobWar` – игровой объект, содержащий модель и информацию о противнике типа воин;
- `Group group` – структура, в которую добавляется созданный разведчик. Передается объекту, контролирующему противника-разведчика;
- `Mob mob` – хранит информацию, об отдельном противнике. Данный скрипт будет записываться в список членов группы;

- `GameObject scoutGO` – объект, который будет алгоритмы поведения группы;
- `Scout scout` – скрипт, в котором описано поведение охранной группы;

Приватные методы:

- `void Awake()` – служит для инициализации переменных;
- `void InstUnit(GameObject enemy, Vector3 spawnPos)` – отвечает непосредственно за создание и размещение противника.

Классы `PatrolGroup` и `SafezoneGroup` описывают алгоритмы поведения созданных групп.

Класс `PatrolGroup` – описывает поведение патрульной группы.

Приватные атрибуты:

- `Group group` – содержит информацию о группе;
- `Vector3 destPoint` – координаты следующего места для патрулирования;
- `Vector3 raPoint` – координаты ближайшей безопасной зоны;
- `int runawayCD` – содержит время, в течение которого отсутствует возможность отступления;
- `bool isMoving` – проверяет, двигается ли группа к заданной патрульной точке;
- `bool isRunawaySet` – проверяет, установлена ли точка отступления.

Публичные атрибуты:

- `List<Vector3> patrolPoint` – список точек для патрулирования.
- Приватные методы:
- `void Start()` – используется для инициализации переменных;
- `void Update()` – используется для обновления кадра игрового объекта;

- `void CalcParams()` – высчитывает параметры, такие как соотношение среднего и максимального здоровья, решение атаки и решение отступления.
- `void Patrol()` – содержит набор действий для состояния патрулирования;
- `void Attack()` – содержит набор действий для состояния атаки;
- `void Runaway()` – содержит набор действий для состояния отступления;
- `void Awaiting()` – содержит набор действий для состояния ожидания;
- `void RunawayCountDown()` – используется как таймер для отсчета времени, в течение которого заблокировано отступление;
- `bool IsPatrolToAttack()` – проверяет условия перехода от состояния патрулирования к состоянию атаки;
- `bool IsPatrolToRunaway()` – проверяет условия перехода от состояния патрулирования к состоянию отступления;
- `bool IsAttackToRunaway()` – проверяет условия перехода от состояния атаки к состоянию отступления;
- `bool IsAttackToPatrol()` – проверяет условия перехода от состояния атаки к состоянию патрулирования;
- `bool IsAttackToAwaiting()` – проверяет условия перехода от состояния атаки к состоянию ожидания;
- `bool IsRunawayToAwaiting()` – проверяет условия перехода от состояния отступления к состоянию ожидания;
- `bool IsAwaitingToAttack()` – проверяет условия перехода от состояния ожидания к состоянию атаки;
- `bool IsAwaitingToPatrol()` – проверяет условия перехода от состояния ожидания к состоянию патрулирования.

Публичные методы:

- `Void SetGroup(Group group)` – используется для передачи информации о группе скрипту, в котором вызывается метод.

Класс `SafezoneGroup` – описывает поведение охранной группы.

Приватные атрибуты:

- `Group group` – содержит информацию о группе;
- `bool isMoving` – проверяет, двигается ли группа к заданной разведчиком точке.

Публичные атрибуты:

- `Vector3` – содержит координаты встречи разведчика с игроком;
- `bool isEntered` – используется для проверки вступления разведчика в группу.

Приватные методы:

- `void Start()` – используется для инициализации переменных;
- `void Update()` – используется для обновления кадра игрового объекта;
- `void CalcParams()` – высчитывает параметры, такие как соотношение среднего и максимального здоровья.
- `void Attack()` – содержит набор действий для состояния атаки;
- `void Awaiting()` – содержит набор действий для состояния ожидания;
- `void Check()` – содержит набор действий для состояния проверки;
- `bool IsAttackToAwaiting()` – проверяет условия перехода от состояния атаки к состоянию ожидания;
- `bool IsAwaitingToAttack()` – проверяет условия перехода от состояния ожидания к состоянию атаки;
- `bool IsAwaitingToCheck()` – проверяет условия перехода от состояния ожидания к состоянию проверки;
- `bool IsCheckToAwaiting()` – проверяет условия перехода от состояния проверки к состоянию ожидания;

- `bool IsCheckToAttack()` – проверяет условия перехода от состояния проверки к состоянию атаки.

Публичные методы:

- `void SetGroup(Group group)` – используется для передачи информации о группе скрипту, вызывающему данный метод.

Класс `Scout` – описывает поведение противников-разведчиков.

Приватные атрибуты:

- `Group group` – содержит информацию о разведчике;
- `Vector3 destPoint` – координаты следующего места для патрулирования;
- `Vector3 curRaPoint` – координаты ближайшей безопасной зоны;
- `int runawayCD` – содержит время, в течение которого отсутствует возможность отступления;
- `bool isMoving` – проверяет, двигается ли группа к заданной патрульной точке;
- `bool isRunawaySet` – проверяет, установлена ли точка отступления.
- `Vector3 checkPoint` – координаты места, где разведчик встретил игрока

Публичные атрибуты:

- `List<Vector3> patrolPoints` – список точек для патрулирования.

Приватные методы:

- `void Start()` – используется для инициализации переменных;
- `void Update()` – используется для обновления кадра игрового объекта;
- `void Patrol()` – содержит набор действий для состояния патрулирования;
- `void Runaway()` – содержит набор действий для состояния отступления;

- void Entering() – содержит набор действий для вступления в группу;
- bool IsPatrolToRunaway() – проверяет условия перехода от состояния патрулирования к состоянию отступления;
- bool IsRunawayToAwaiting() – проверяет условия перехода от состояния отступления к состоянию ожидания;
- bool IsAwaitingToEntering() – проверяет условия перехода от состояния ожидания к состоянию вступления в группу;

Публичные методы:

- Void SetGroup(Group group) – используется для передачи информации о разведчике скрипту, вызывающему данный метод.

3.3. Контрольный пример

В данной работе были созданы алгоритмы, контролирующие группы противников. Общий список алгоритмов выглядит следующим образом:

- Создание противников и объединение их в состав группы;
- Алгоритм поведения охранной группы;
- Алгоритм поведения патрульной группы;
- Алгоритм поведения противника-разведчика

Пример работы алгоритмов представлен ниже:

- 1) Запускаем игру
- 2) На уровне появляются все противники
- 3) Подходим к патрульной группе

Группа проверяет решение об атаке и решение об отступлении. Они равны 1.1 и 0 соответственно. Следовательно, группа нападет на игрока (рисунок 14).

- 4) Во время боя решение об отступлении поднялось до 0.59 и группа решила отступить, что видно на рисунке 15.
- 5) Охранная группа охраняет свою безопасную зону (рисунок 16)

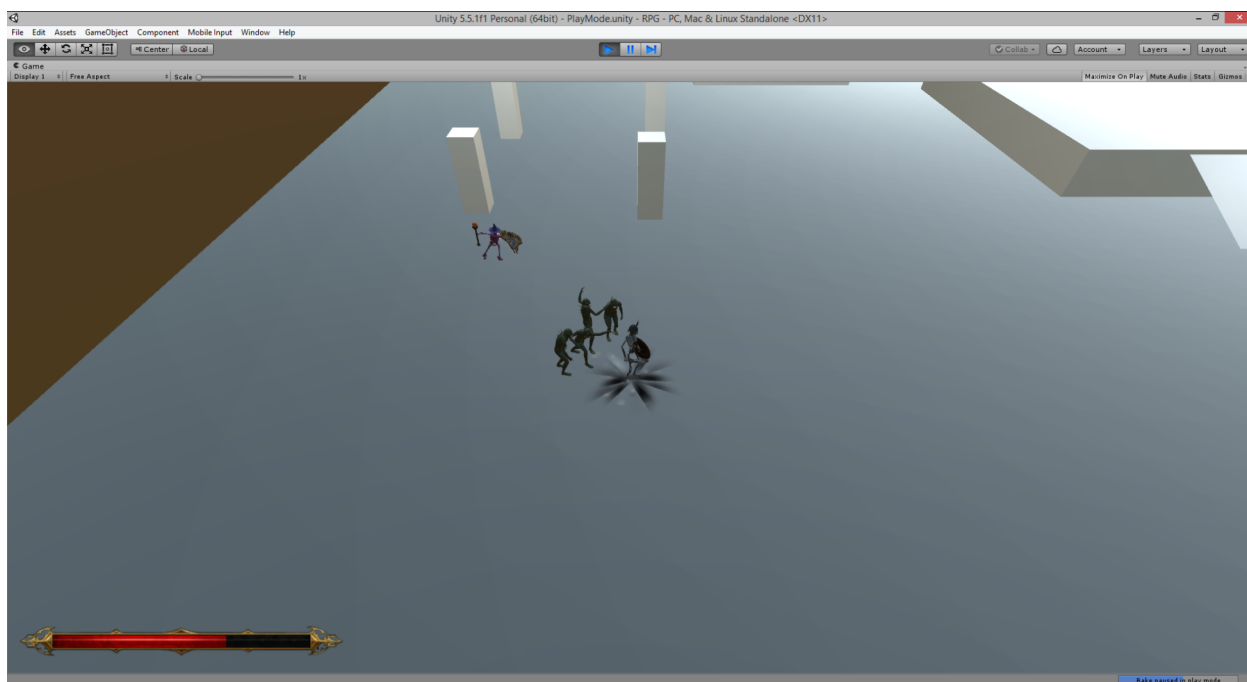


Рисунок 14 – Патрульная группа нападает на игрока

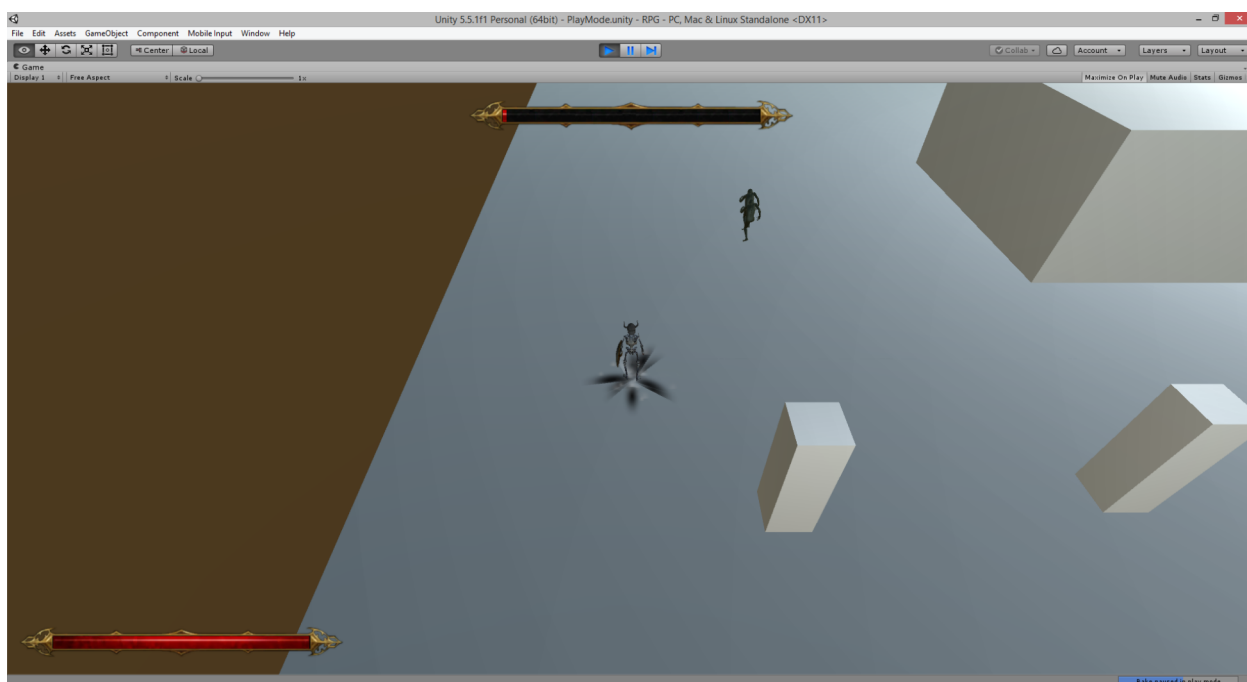


Рисунок 15 – Патрульная группа отступает в безопасную зону

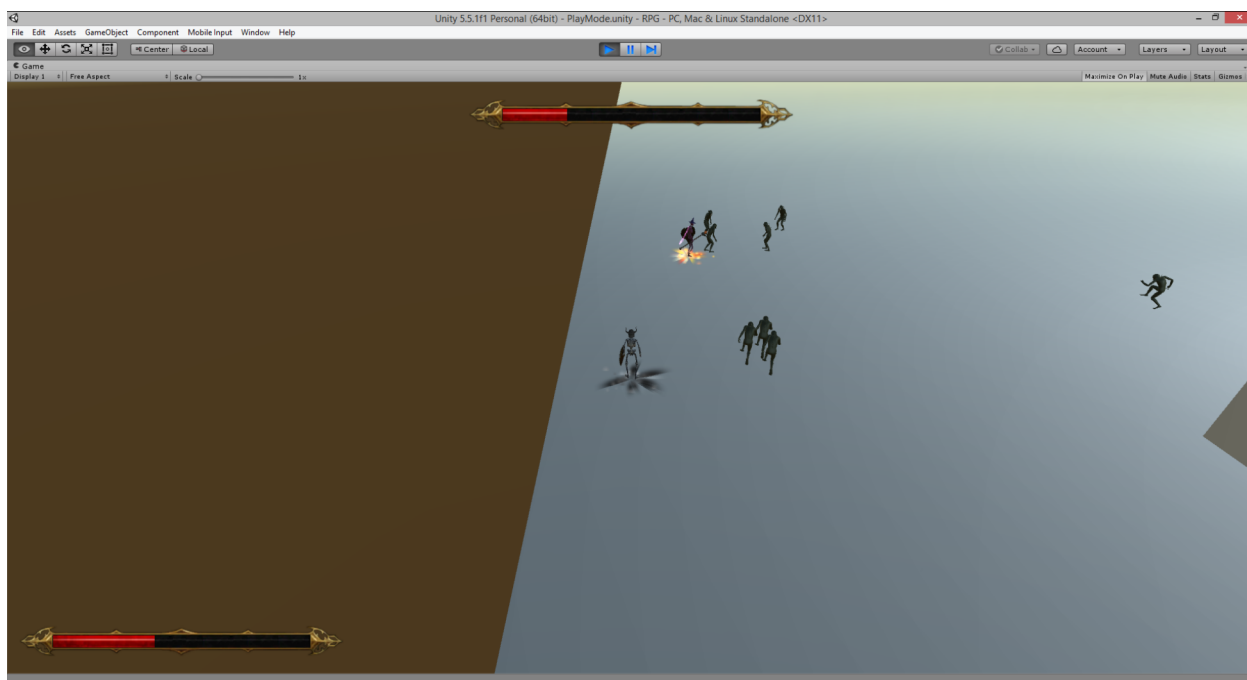


Рисунок 16 – Охранная группа, к которой отступает патрульная группа

6) Если подойти близко к зоне, то обе группы нападут на игрока (рисунок 17)

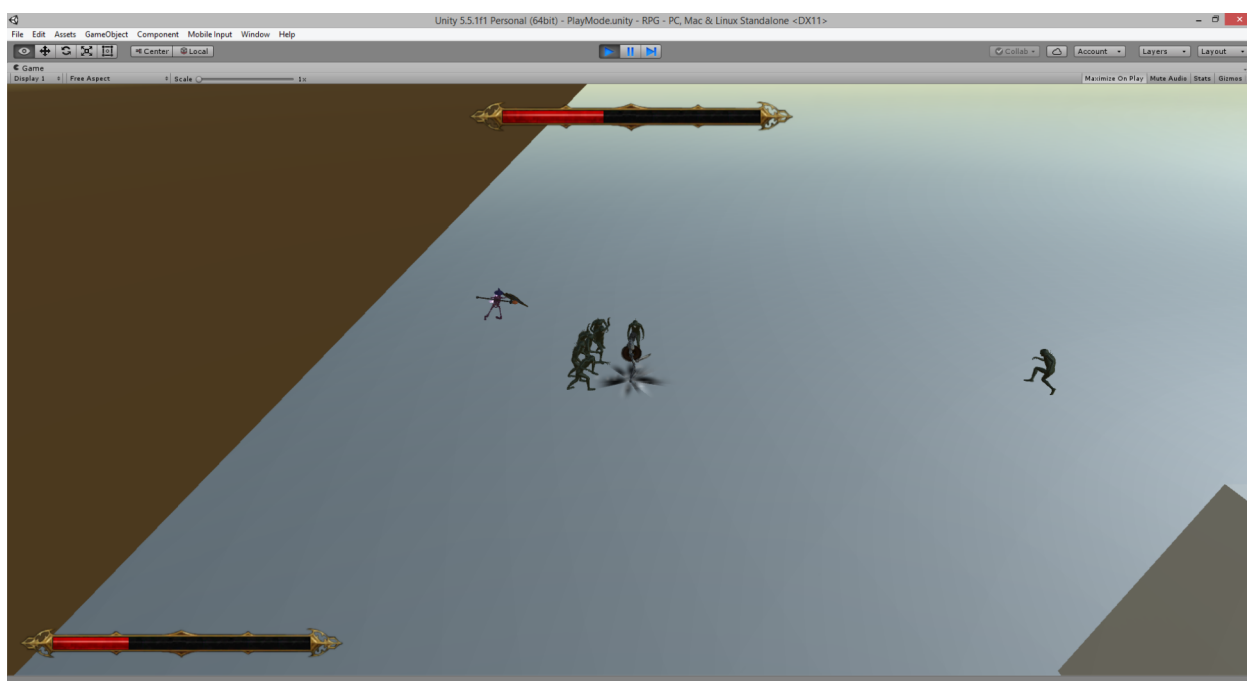


Рисунок 17 – Охранная и отступившая патрульная группы нападают на игрока

7) Патрульная группа через некоторое время вернется к патрулированию

При этом решение об атаке у группы равно 0.34, а решение об отступлении 0.41. Следовательно группа не нападет на игрока сама (рисунок 18).

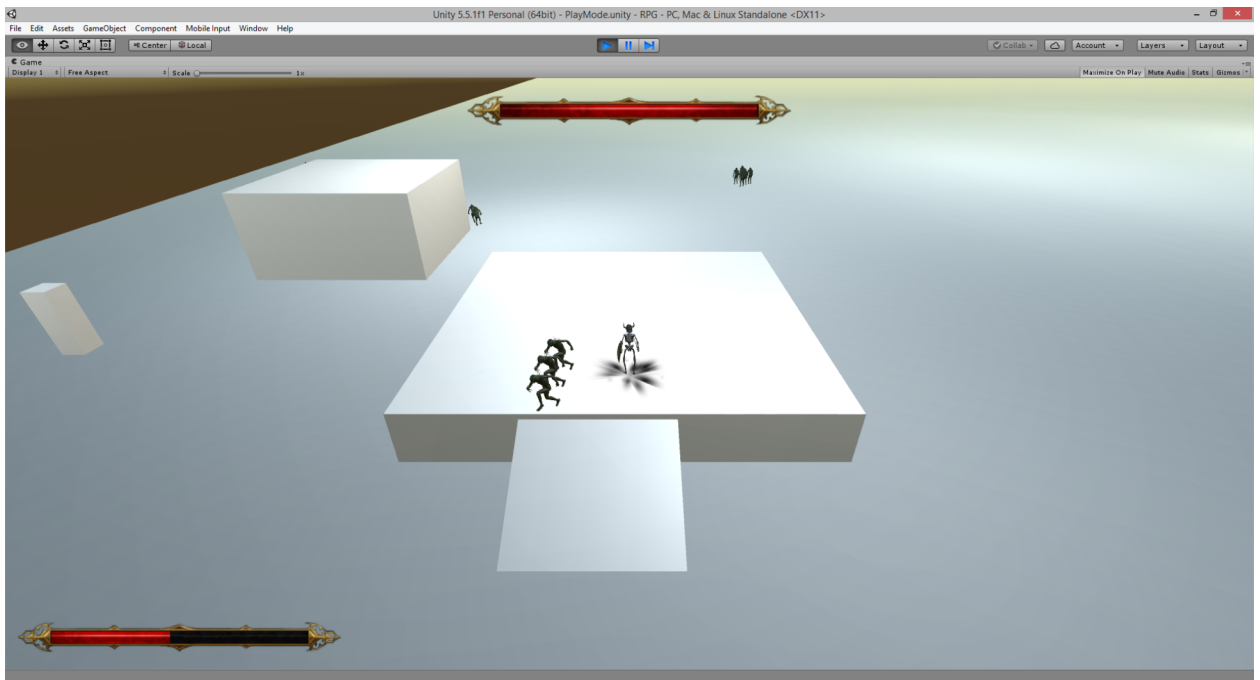


Рисунок 18 – Патрульная группа не нападает на игрока

8) Подходим к противнику-разведчику, он начинает отступать к ближайшей безопасной зоне (рисунок 19)

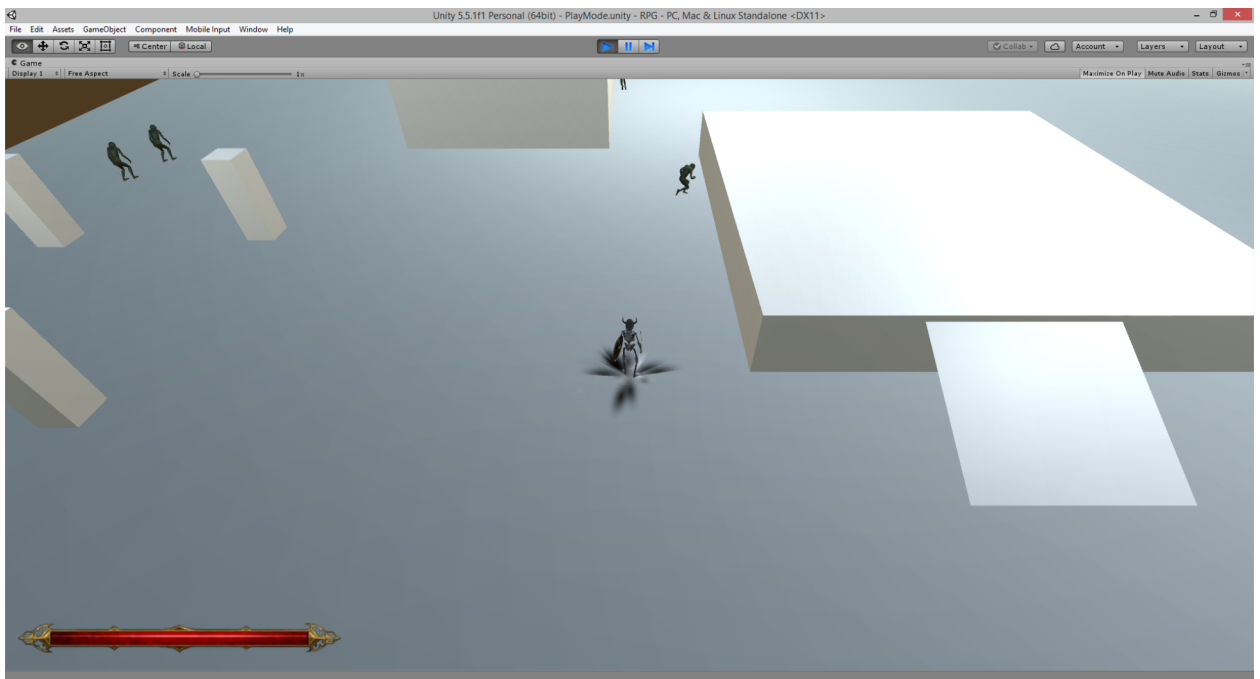


Рисунок 19 – Разведчик отступает к безопасной зоне

9) К месту встречи разведчика с игроком приходит охранная группа
(рисунок 20)

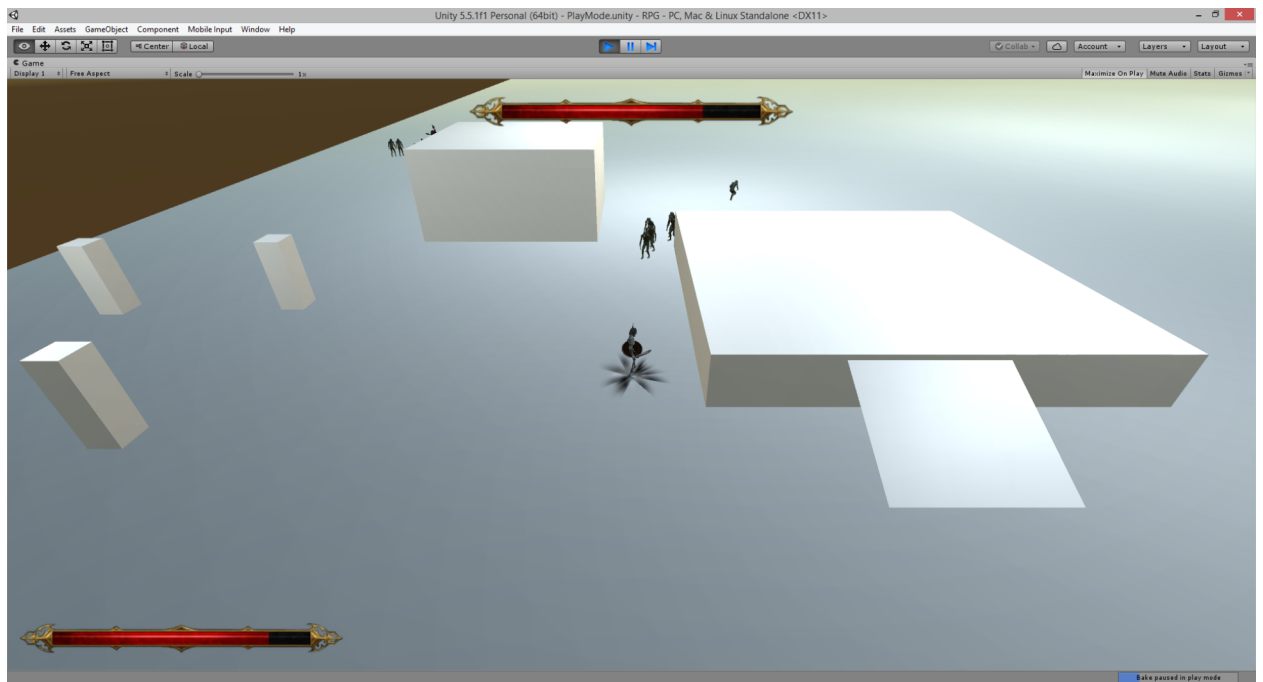


Рисунок 20 – Охранная группа проверяет место встречи

В результате были разработаны и протестированы все спроектированные алгоритмы поведения противников.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы были разработаны алгоритмы группового поведения противников и доказана их работоспособность на прототипе игры в жанре RPG.

Был проведен анализ предметной области и рассмотрены существующие аналоги. При сравнении аналогов были выделены их недостатки.

Разработка игры и алгоритмов велась на игровом движке Unity с помощью языка C# в интеграционной среде разработки Visual Studio 2015.

В результате выполнения выпускной квалификационной работы были спроектированы и разработаны 3 алгоритма поведения, каждый из которых контролирует определенный тип группы. Работа в полном объеме отвечает требованиям технического задания.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Компьютерная игра [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/Компьютерная_игра, свободный, 22.05.2017г – Загл. с экрана.
2. Компьютерная ролевая игра [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/Компьютерная_ролевая_игра, свободный, 22.05.2017г – Загл. с экрана.
3. Diablo 3 [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/Diablo_III, 22.05.2017г – Загл. с экрана.
4. S.T.A.L.K.E.R.: Чистое небо [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/S.T.A.L.K.E.R.:_Чистое_небо, 22.05.2017г – Загл. с экрана.
5. Warcraft 3 [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/Warcraft_III:_Reign_of_Chaos, 22.05.2017г – Загл. с экрана.
6. Unity [Электронный ресурс] – Режим доступа: [https://ru.wikipedia.org/Unity_\(игровой_движок\)](https://ru.wikipedia.org/Unity_(игровой_движок)), 25.05.2017г – Загл. с экрана.
7. Unreal Engine [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/Unreal_Engine, 25.05.2017г – Загл. с экрана.
8. CryEngine 3 [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/CryEngine_3, 25.05.2017г – Загл. с экрана.
9. C# [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/C_Sharp, 25.05.2017г – Загл. с экрана.
10. Объектно-ориентированное программирование [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование, свободный, 22.05.2017г – Загл. с экрана.

11. JavaScript [Электронный ресурс] – Режим доступа: <https://ru.wikipedia.org/JavaScript>, 25.05.2017г – Загл. с экрана.
12. Microsoft Visual Studio [Электронный ресурс] – Режим доступа: [https://ru.wikipedia.org/ Microsoft_Visual_Studio](https://ru.wikipedia.org/Microsoft_Visual_Studio), 25.05.2017г – Загл. с экрана.
13. Unity: Руководство [Электронный ресурс] – Режим доступа: <https://docs.unity3d.com/ru/530/Manual/CreatingAndUsingScripts.html>, свободный, 22.05.2017г – Загл. с экрана.
14. Artificial Intelligence For Games/ Ian Millington – Morgan Kaufmann Publishers, 2006. – 895с.

ПРИЛОЖЕНИЕ

ПРИЛОЖЕНИЕ А. ЛИСТИНГ

Алгоритм поведения патрульной группы.

PatrolGroup.cs

```
public class PatrolGroup : MonoBehaviour
{
    //Объявляем переменные
    Groups.Group group = new Groups.Group();

    public void SetGroup(Groups.Group group)
    {
        this.group = group;
    }
    public List<Vector3> patrolPoints;
    Vector3 destPoint;
    Vector3 curRaPoint;
    int runawayCD;
    bool isMoving;
    bool isRunawaySet;

    //Инициализируем необходимые переменные
    void Start()
    {
        patrolPoints = new List<Vector3>()
        {
            new Vector3(27,0,19),
            new Vector3(40,1,26),
            new Vector3(24,0,57),
            new Vector3(50,0,46),
        };
        runawayCD = 0;
        group.curState = "patrol";
        isMoving = false;
        destPoint = transform.position;
        isRunawaySet = false;

        CalcStartParams();
    }

    // Функция вызывается каждый кадр
    void Update()
    {
        //Удаляем из списка группы уничтоженных противников
        for (int i = 0; i < group.charsList.Count; i++)
        {
            if (group.charsList[i].IsDead())
            {
                group.charsList.Remove(group.charsList[i]);
                --i;
            }
        }
        if (group.charsList.Count == 0)
        {
            return;
        }

        CalcParams();
        GroupRestoration();
    }
}
```

```

//Переключение состояний
if (group.curState == "patrol")
{
    Patrol();
}
if (group.curState == "attack")
{
    Attack();
}
if (group.curState == "runaway")
{
    Runaway();
}
if (group.curState == "awaiting")
{
    Awaiting();
}
}
//Вычисление стартовых параметров для группы
void CalcStartParams()
{
    group.maxHealth = 0;
    group.currentHealth = 0;
    group.agroLevel = 0;
    group.fearLevel = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {
        group.maxHealth += group.charsList[i].enemyChar.maxHealth;
        group.currentHealth += group.charsList[i].enemyChar.currentHealth;
        group.agroLevel += group.charsList[i].enemyChar.agroLevel;
        group.fearLevel += group.charsList[i].enemyChar.fearLevel;
        group.charsList[i].SendMessage("SetGroup", group);
    }

    group.currentHealthPerc = group.currentHealth / group.maxHealth;
    group.agroLevel = (group.agroLevel / group.charsList.Count) * (1 + 0.1 *
group.charsList.Count);
    group.fearLevel = (group.agroLevel / group.charsList.Count) * (1 - 0.1 *
group.charsList.Count);
    group.attackDecision = group.currentHealthPerc * group.agroLevel;
    group.runawayDecision = (1 - group.currentHealthPerc) * group.fearLevel;
}

//Вычисление параметров группы, которые будут меняться во время игры
void CalcParams()
{
    group.maxHealth = 0;
    group.currentHealth = 0;
    group.agroLevel = 0;
    group.fearLevel = 0;

    for (int i = 0; i < group.charsList.Count; i++)
    {
        group.maxHealth += group.charsList[i].enemyChar.maxHealth;
        group.currentHealth += group.charsList[i].enemyChar.currentHealth;
        group.agroLevel += group.charsList[i].enemyChar.agroLevel;
        group.fearLevel += group.charsList[i].enemyChar.fearLevel;
    }

    group.currentHealthPerc = group.currentHealth / group.maxHealth;
    group.agroLevel = (group.agroLevel / group.charsList.Count) * (1 + 0.02 *
group.charsList.Count);

```

```

        group.fearLevel = (group.fearLevel / group.charsList.Count) * (1 - 0.02 *
group.charsList.Count);

        group.attackDecision = group.currentHealthPerc * group.agroLevel;
        group.runawayDecision = (1 - group.currentHealthPerc) * group.fearLevel;

        group.size = group.charsList.Count;
    }

    //Восстановление здоровья группы, если они находятся в безопасной зоне
    void GroupRestoration()
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            if (Vector3.Distance(group.charsList[i].transform.position, curRaPoint) <
4 && group.charsList[i].isRestoring == false)
            {
                group.charsList[i].InvokeRepeating("SafezoneHealing", 0f, 1f);
                group.charsList[i].isRestoring = true;
            }
            else if (Vector3.Distance(group.charsList[i].transform.position,
curRaPoint) > 4)
            {
                group.charsList[i].CancelInvoke("SafezoneHealing");
                group.charsList[i].isRestoring = false;
            }
        }
    }

    //Действия группы в состоянии патрулирования
    void Patrol()
    {
        int p;

        if (isMoving == false && Random.Range(0, 300) < 1)
        {
            p = Random.Range(0, patrolPoints.Count - 1);
            destPoint = patrolPoints[p];
            for (int i = 0; i < group.charsList.Count; i++)
            {
                group.charsList[i].Follow(destPoint);
            }
            isMoving = true;
        }

        for (int a = 0; a < group.charsList.Count; a++)
        {
            if (Vector3.Distance(group.charsList[a].transform.position, destPoint) < 3 &&
isMoving)
            {
                for (int i = 0; i < group.charsList.Count; i++)
                {
                    group.charsList[i].EndFollow();
                }
                isMoving = false;
                break;
            }
        }

        //переход на атаку
        if (IsPatrolToAttack())
        {
            isMoving = false;

```

```

        group.curState = "attack";
        return;
    }

    //переход на побег
    if (IsPatrolToRunaway())
    {
        isMoving = false;
        group.curState = "runaway";
        return;
    }
}

//Действия группы в состоянии атаки
void Attack()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        group.charsList[i].SendMessage("Attack");
    }

    //переход на ожидание
    if (IsAttackToAwaiting())
    {
        group.curState = "awaiting";
        return;
    }

    //переход на патрулирование
    if (IsAttackToPatrol())
    {
        group.curState = "patrol";
        return;
    }

    //переход на побег
    if (IsAttackToRunaway())
    {
        group.curState = "runaway";
        return;
    }
}

//Действия группы в состоянии отступления
void Runaway()
{
    Vector3 raPoint = Groups.safezonePoints[0];
    if (isRunawaySet == false)
    {
        for (int i = 0; i < Groups.safezonePoints.Count - 1; i++)
        {
            if (Vector3.Distance(raPoint, group.charsList[0].transform.position) >
                Vector3.Distance(Groups.safezonePoints[i + 1], group.charsList[0].transform.position))
            {
                raPoint = Groups.safezonePoints[i + 1];
            }
        }
        curRaPoint = raPoint;
        isRunawaySet = true;
    }
    for (int i = 0; i < group.charsList.Count; i++)
    {

```

```

        if (group.charsList[i].IsDead() == false)
        {
            group.charsList[i].Follow(curRaPoint);
        }
        else
        {
            group.charsList[i].EndFollow();
        }
    }

    //переход на ожидание
    if (IsRunawayToAwaiting())
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            group.charsList[i].isRunawayBlocked = true;
        }
        //Включение блокировки отступления
        runawayCD = 15;
        InvokeRepeating("RunawayCountDown", 0f, 1f);

        group.curState = "awaiting";
        return;
    }
}
//Действия группы в состоянии ожидания
void Awaiting()
{
    for (int a = 0; a < group.charsList.Count; a++)
    {
        if (Vector3.Distance(group.charsList[a].transform.position, curRaPoint) > 2)
        {
            for (int i = 0; i < group.charsList.Count; i++)
            {
                group.charsList[i].Follow(curRaPoint);
            }
        }
        else
        {
            for (int i = 0; i < group.charsList.Count; i++)
            {
                group.charsList[i].EndFollow();
            }
            break;
        }
    }
}

// проверка перехода на атаку
if (IsAwaitingToAttack())
{
    group.curState = "attack";
    return;
}

// проверка перехода на патрулирование
if (IsAwaitingToPatrol())
{
    group.curState = "patrol";
    return;
}

```

```

}
//Отсчет блокировки отступления
void RunawayCountDown()
{
    if (runawayCD > 0)
    {
        runawayCD -= 1;
    }
    if (runawayCD == 0)
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            group.charsList[i].isRunawayBlocked = false;
        }
        CancelInvoke("RunawayCountDown");
    }
}

#region Проверка переходов

//патруль -> атака
bool IsPatrolToAttack()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].isAttacked == true)
        {
            return true;
        }
    }

    float attackCount = 0;

    for (int a = 0; a < group.charsList.Count; a++)
    {
        if (group.charsList[a].InAgroRange())
        {
            for (int i = 0; i < group.charsList.Count; i++)
            {
                if (group.charsList[i].enemyChar.attackDecision > 0.5)
                {
                    attackCount++;
                }
            }

            attackCount /= group.charsList.Count;

            if (group.attackDecision > 0.5 || attackCount > 0.5)
            {
                return true;
            }
        }
    }
    return false;
}

// патруль -> побег
bool IsPatrolToRunaway()
{
    float runawayCount = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {

```

```

        if (group.charsList[i].enemyChar.runawayDecision > 0.5)
        {
            runawayCount++;
        }
    }

    runawayCount /= group.charsList.Count;

    if (group.runawayDecision > 0.5 || runawayCount > 0.6)
    {
        return true;
    }
    return false;
}

//атака -> патруль
bool IsAttackToPatrol()
{
    int patrolCount = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            patrolCount++;
        }
    }
    if (patrolCount == 0)
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            group.charsList[i].EndFollow();
            group.charsList[i].anim.SetBool("isAttacking", false);
        }
        return true;
    }
    return false;
}

//атака -> побег
bool IsAttackToRunaway()
{
    float runawayCount = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].enemyChar.runawayDecision > 0.5)
        {
            runawayCount++;
        }
    }

    runawayCount /= group.charsList.Count;

    if (group.runawayDecision > 0.5 || runawayCount > 0.7)
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            group.charsList[i].anim.SetBool("isAttacking", false);
        }
        return true;
    }
    return false;
}

```

```

// атака -> ожидание
bool IsAttackToAwaiting()
{
    float attackCount = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            attackCount++;
        }
    }
    for (int a = 0; a < group.charsList.Count; a++)
    {
        if (attackCount == 0 && group.charsList[a].isRunawayBlocked == true)
        {
            for (int i = 0; i < group.charsList.Count; i++)
            {
                group.charsList[i].anim.SetBool("isAttacking", false);
            }
            return true;
        }
    }
    return false;
}

//побег -> ожидание
bool IsRunawayToAwaiting()
{
    for (int a = 0; a < group.charsList.Count; a++)
    {
        if (Vector3.Distance(group.charsList[a].transform.position, curRaPoint) < 2)
        {
            for (int i = 0; i < group.charsList.Count; i++)
            {
                group.charsList[i].EndFollow();
            }
            isRunawaySet = false;
            return true;
        }
    }
    return false;
}

//ожидание -> атака
bool IsAwaitingToAttack()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            return true;
        }
    }
    return false;
}

//ожидание -> патруль
bool IsAwaitingToPatrol()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].isRunawayBlocked == false)
        {

```

```

        return true;
    }
}
return false;
}
#endregion
}

```

Алгоритм поведения охранной группы. SafezoneGroup.cs

```

public class SafezoneGroup : MonoBehaviour
{
    //Объявление переменных
    public Groups.Group group = new Groups.Group();

    public void SetGroup(Groups.Group group)
    {
        this.group = group;
    }
    public bool isEntered;
    public Vector3 checkPoint;
    bool isMoving;

    //Инициализация переменных
    void Start()
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            group.maxHealth += group.charsList[i].enemyChar.maxHealth;
            group.currentHealth += group.charsList[i].enemyChar.currentHealth;
            group.charsList[i].isRunawayBlocked = true;
            group.charsList[i].SendMessage("SetGroup", group);
        }
        group.curState = "awaiting";
        isEntered = false;
        isMoving = false;
    }

    // Вызывается каждый кадр
    void Update()
    {
        //Удаляем из списка группы уничтоженных противников
        for (int i = 0; i < group.charsList.Count; i++)
        {
            if (group.charsList[i].IsDead())
            {
                group.charsList.Remove(group.charsList[i]);
                --i;
            }
        }
        if (group.charsList.Count == 0)
        {
            return;
        }

        CalcParams();

        GroupRestoration();

        //Переключение состояний
        if (group.curState == "attack")
        {

```

```

        Attack();
    }
    if (group.curState == "awaiting")
    {
        Awaiting();
    }
    if (group.curState == "check")
    {
        Check();
    }
}

//Вычисление параметров группы, изменяющихся во время игры
void CalcParams()
{
    group.currentHealth = 0;

    for (int i = 0; i < group.charsList.Count; i++)
    {
        group.currentHealth += group.charsList[i].enemyChar.currentHealth;
    }
}

//Восстановление здоровья группы, если они находятся в безопасной зоне
void GroupRestoration()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (Vector3.Distance(group.charsList[i].transform.position,
group.spawnedPosisition) < 4 && group.charsList[i].isRestoring == false)
        {
            group.charsList[i].InvokeRepeating("SafezoneHealing", 0f, 1f);
            group.charsList[i].isRestoring = true;
        }
        else if (Vector3.Distance(group.charsList[i].transform.position,
group.spawnedPosisition) > 4)
        {
            group.charsList[i].CancelInvoke("SafezoneHealing");
            group.charsList[i].isRestoring = false;
        }
    }
}

//Действия группы в состоянии атаки
void Attack()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        group.charsList[i].SendMessage("Attack");
    }

    // переход на ожидание
    if (IsAttackToAwaiting())
    {
        group.curState = "awaiting";
        return;
    }
}

//Действия группы в состоянии ожидания
void Awaiting()
{

```

```

        for (int a = 0; a < group.charsList.Count; a++)
        {
            if (Vector3.Distance(group.charsList[a].transform.position,
group.spawnedPosisition) > 2)
            {
                for (int i = 0; i < group.charsList.Count; i++)
                {
                    group.charsList[i].Follow(group.spawnedPosisition);
                }
            }
            else
            {
                for (int i = 0; i < group.charsList.Count; i++)
                {
                    group.charsList[i].EndFollow();
                }
                break;
            }
        }

        //переход на атаку
        if (IsAwaitingToAttack())
        {
            group.curState = "attack";
            return;
        }

        //переход на проверку
        if (IsAwaitingToCheck())
        {
            group.curState = "check";
            return;
        }
    }

    //Действия группы в состоянии проверки
    void Check()
    {
        if (isMoving == false)
        {
            for (int i = 0; i < group.charsList.Count; i++)
            {
                group.charsList[i].Follow(checkPoint);
            }
            isMoving = true;
        }

        for (int a = 0; a < group.charsList.Count; a++)
        {
            if (Vector3.Distance(group.charsList[a].transform.position, checkPoint) < 2
&& isMoving)
            {
                for (int i = 0; i < group.charsList.Count; i++)
                {
                    group.charsList[i].EndFollow();
                }
                isMoving = false;
                break;
            }
        }

        //переход на атаку
        if (IsCheckToAttack())

```

```

    {
        isMoving = false;
        group.curState = "attack";
        return;
    }

    // переход на ожидание
    if (IsCheckToAwaiting())
    {
        group.curState = "awaiting";
        return;
    }
}

#region Проверка переходов

//атака -> ожидание
bool IsAttackToAwaiting()
{
    int attackCount = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            attackCount++;
        }
    }
    if (attackCount == 0)
    {
        for (int i = 0; i < group.charsList.Count; i++)
        {
            group.charsList[i].anim.SetBool("isAttacking", false);
        }
        return true
    }
    return false;
}

//ожидание -> атака
bool IsAwaitingToAttack()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            return true;
        }
    }
    return false;
}

//ожидание -> проверка
bool IsAwaitingToCheck()
{
    if (isEntered)
    {
        isEntered = false;
        return true;
    }
    return false;
}

//проверка -> атака

```

```

bool IsCheckToAttack()
{
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            return true;
        }
    }
    return false;
}

//проверка -> ожидание
bool IsCheckToAwaiting()
{
    int attackCount = 0;
    for (int i = 0; i < group.charsList.Count; i++)
    {
        if (group.charsList[i].InAgroRange())
        {
            attackCount++;
        }
    }
    if (attackCount == 0 && Random.Range(0, 400) < 1 && isMoving == false)
    {
        return true;
    }
    return false;
}
}
#endregion
}

```

Алгоритм поведения противников-разведчиков. Scout.cs

```

public class Scout : MonoBehaviour
{
    //Объявление переменных
    Groups.Group group = new Groups.Group();

    public void SetGroup(Groups.Group group)
    {
        this.group = group;
    }

    public List<Vector3> patrolPoints;

    Vector3 destPoint;
    Vector3 curRaPoint;
    bool isMoving;
    bool isRunawaySet;
    Vector3 checkPoint;

    //Инициализация переменных
    void Start()
    {
        patrolPoints = new List<Vector3>()
        {
            new Vector3(27,0,19),
            new Vector3(40,1,26),
            new Vector3(24,0,57),
            new Vector3(50,0,46),
        };
    }
}

```

```

        group.curState = "patrol";
        isMoving = false;
        destPoint = transform.position;
        isRunawaySet = false;

        group.charsList[0].SendMessage("SetGroup", group);
    }

    // Вызывается каждый кадр
    void Update()
    {
        if (group.charsList[0].IsDead())
        {
            return;
        }

        //Переключение состояний
        if (group.curState == "patrol")
        {
            Patrol();
        }
        if (group.curState == "runaway")
        {
            Runaway();
        }
        if (group.curState == "entering")
        {
            Entering();
        }
    }

    //Действия в состоянии патрулирования
    void Patrol()
    {
        group.charsList[0].agent.speed = 9;

        int p;

        if (isMoving == false)
        {
            p = Random.Range(0, patrolPoints.Count - 1);
            destPoint = patrolPoints[p];
            group.charsList[0].Follow(destPoint);
            isMoving = true;
        }

        if (Vector3.Distance(group.charsList[0].transform.position, destPoint) < 3 &&
isMoving)
        {
            group.charsList[0].EndFollow();
            isMoving = false;
        }

        //переход на побег
        if (IsPatrolToRunaway())
        {
            isMoving = false;
            checkPoint = group.charsList[0].transform.position;
            group.curState = "runaway";
            return;
        }
    }
}

```

```

//Действия в состоянии отступления
void Runaway()
{
    Vector3 raPoint = Groups.safezonePoints[0];
    if (isRunawaySet == false)
    {
        for (int i = 0; i < Groups.safezonePoints.Count - 1; i++)
        {
            if (Vector3.Distance(raPoint, group.charsList[0].transform.position) >
                Vector3.Distance(Groups.safezonePoints[i + 1], group.charsList[0].transform.position))
            {
                raPoint = Groups.safezonePoints[i + 1];
            }
        }
        curRaPoint = raPoint;
        isRunawaySet = true;
    }

    if (group.charsList[0].IsDead() == false)
    {
        group.charsList[0].Follow(curRaPoint);
    }
    else
    {
        group.charsList[0].EndFollow();
    }

    //переход на вступление в группу
    if (IsRunawayToEntering())
    {
        group.curState = "entering";
        return;
    }
}

//Действия при вступлении в группу
void Entering()
{
    group.charsList[0].agent.speed = group.charsList[0].walkSpeed;
    GameObject newGroupGO = Groups.safezonePointsDict[curRaPoint];
    SafezoneGroup newGroup = newGroupGO.GetComponent<SafezoneGroup>();
    if (newGroup.group.charsList.Count != 0)
    {
        if (Vector3.Distance(group.charsList[0].transform.position,
            newGroup.group.charsList[0].transform.position) < 4)
        {
            newGroup.group.charsList.Add(group.charsList[0]);
            newGroup.isEntered = true;
            newGroup.checkPoint = checkPoint;
            Destroy(gameObject);
        }
    }
    else
    {
        newGroup.group.charsList.Add(group.charsList[0]);
        newGroup.isEntered = true;
        newGroup.checkPoint = checkPoint;
        Destroy(gameObject);
    }
}

#region Проверка переходов

```

```

// патруль -> побег
bool IsPatrolToRunaway()
{
    if (group.charsList[0].InAgroRange())
    {
        return true;
    }
    return false;
}

//побег -> вступление
bool IsRunawayToEntering()
{
    if (Vector3.Distance(group.charsList[0].transform.position, curRaPoint) < 2)
    {
        group.charsList[0].EndFollow();
        isRunawaySet = false;
        return true;
    }
    return false;
}
#endregion
}

```