МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**
**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**
**Кафедра анализа данных и искусственного интеллекта**

**КУРСОВАЯ РАБОТА**

# РАЗРАБОТКА СИСТЕМЫ АВТОНОМНЫХ АГЕНТОВ ДЛЯ КОМПЬЮТЕРНОЙ ИГРЫ

Работу выполнил ________________________________________ К.Р. Сеидов
(подпись)

Направление подготовки 01.04.02 «Прикладная математика и информатика»

Направленность (профиль) Математическое и информационное обеспечение экономической деятельности

Научный руководитель
д-р техн. наук, проф _____________________________________ А.А. Халафян
(подпись)

Нормоконтролер
канд. физ.-мат. наук, доцент_____________________________ Г.В. Калайдина
(подпись)

Краснодар
2021

РЕФЕРАТ

Курсовая работа содержит 36 с., 20 рис., 6 источников.

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ, АВТОНОМНЫЙ АГЕНТ, UNITY, ПРОЦЕДУРЕАЯ ГЕНЕРАЦИЯ.

Объектом исследование являются видеоигры, предметом – методы разработки системы искусственного интеллекта для видеоигры.

Целью курсовой работы является разработка системы автономных агентов, существующих и взаимодействующих в виртуальном пространстве.

В результате курсовой работы разработан прототип приложения, реализующего систему автономных агентов, со следующими возможностями:

– корректное перемещение по виртуальному пространству;
– следование определенному набору поведений;
– планирование маршрута по виртуальному окружению;
– реагирование на других агентов системы.

Для отладки системы искусственного интеллекта (ИИ) разработана система процедурной генерации тестового окружения.

Архитектура системы была построена таким образом, чтобы иметь возможность дальнейшего расширения и модификации различных ее компонент независимо друг от друга.

# СОДЕРЖАНИЕ

Введение ..... 4
1 Исследование использованных средств ..... 6
1.1 Среда разработки Unity. ..... 6
1.2 Принятие решений ..... 9
1.3 Передвижени ..... 12
1.3.1 Рулевые силы «Поиск» и «Бегство» ..... 13
1.3.2 Рулевая сила «Преследование» ..... 14
1.3.3 Рулевая сила «Блуждание» ..... 16
1.3.4 Рулевая сила «Избегание стен» ..... 17
1.3.5 Вычисление результирующей силы ..... 18
1.4 Планирование пути ..... 18
1.4.1 Алгоритм поиска пути A* ..... 19
1.4.2 Система переноса окружения в формат графа ..... 18
2 Процедурная генерация тестового окружения ..... 23
2.1 Генерация регулярной сетки ..... 23
2.2 Генерация полигональной сетки ..... 25
3 Программная реализация ..... 27
3.1 Реализация компонента принятия решений ..... 28
3.2 Реализация перемещения ..... 30
3.3 Реализация построения маршрута ..... 30
3.4 Реализация генерации окружения ..... 32
3.5 Результаты работы ..... 34
Заключение ..... 35
Список использованных источников ..... 36

# ВВЕДЕНИЕ

На сегодняшний день индустрия компьютерных игр практически сравнялась с киноиндустрией как по объему средств с продаж, так и по бюджетам, выделяемым на реализацию проектов, аудитория игроков на различных платформах с каждым годом продолжает расти.

Одним из наиболее сложных в плане разработки и одним из ключевых аспектов для игрока является наличие системы искусственного интеллекта в игре.

Целью курсовой работы является разработка системы автономных агентов, существующих и взаимодействующих в виртуальном пространстве.

В качестве среды применения инструментов извлечения была выбрана среда Unity, предоставляющая удобный способ работы с 3D-графикой и поддерживающее множество различных платформ для развертывания разработанных в ней приложений.

Исходя из названной цели, в работе поставлен ряд задач:

– изучить и проанализировать актуальные технологии, применяемые при разработке ИИ в видеоиграх;

– выбрать подходы для реализации различных компонентов системы ИИ;

– реализовать подсистемы:

1) принятия решений для автономных агентов;
2) перемещения для автономных агентов;
3) планирования маршрута для автономных агентов;

– разработать систему генерации окружения для тестирования агентов.

Общую теоретико-методологическую базу курсового проекта составили труды зарубежных исследователей, опубликованные на различных иностранных сайтах.

В качестве среды применения инструментов извлечения была выбрана среда Unity, предоставляющая удобный способ работы с 3D-графикой и поддерживающее множество различных платформ для развертывания разработанных в ней приложений.

В рамках курсовой работы разработана система автономных агентов для видеоигры. Под автономным агентом понимается игровая сущность (например, неигровой персонаж), использующая информацию из данных игры для определения дальнейших действий и, самостоятельно исполняющая эти действия.

В ходе проектирования системы было выделено 3 ее компонента, а именно:

– компонент принятие решений;
– компонент передвижение;
– компонент планирование пути.

Используемые подходы для реализации каждой из этих компонент будут в работе рассмотрены более подробно.

# 1 Исследование использованных средств

## 1.1 Среда разработки Unity

Unity3D или просто Unity – кроссплатформенная среда разработки приложений с поддержкой трехмерной и двухмерной графики. Основной областью применения среды является разработка компьютерных игр. Тем не менее, она так же широко используется при разработке приложений различного рода с применением технологии виртуальной или дополненной реальности. Одним из основных достоинств среды является возможность сборки приложения под различные платформы, начиная от персональных компьютеров на ОС Windows или Mac OS, и заканчивая игровыми консолями текущего поколения и мобильными устройствами под управлением Android или iOS.

Редактор Unity имеет простой Drag&Drop интерфейс, который легко настраивать, состоящий из различных окон, благодаря чему можно производить отладку игры прямо в редакторе. Среда поддерживает C# в качестве скриптового языка. Расчёты физики производит физический движок PhysX от NVIDIA.

Проект в Unity делится на уровни, которые представлены отдельными файлами сцен, содержащие свои игровые миры со своим набором объектов, игровых сценариев, так называемые скрипты, и настроек. В концептуальном плане сцена представляет собой некий аналог окна из привычных оконных приложений. Сцены могут содержать в себе как объекты, содержащие модели, 2D-изображения, так и пустые игровые объекты. Они в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. Также у объектов есть название, им возможно присвоить метку, так называемый тег, и слой, на котором он должен отображаться. Так у любого объекта на сцене обязательно присутствует компонент Transform, который хранит в себе координаты местоположения, поворота и размеров объекта по

всем трём осям. У объектов с видимой геометрией также по умолчанию присутствует компонент Mesh Renderer, делающий модель объекта видимой. К объектам можно добавлять такие компоненты как коллайдеры (англ. collider), которые отвечают за параметры сталкивания объектов. В редакторе имеется система наследования объектов; дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Скрипты в редакторе прикрепляются к объектам в виде отдельных компонентов. Редактор Unity имеет компонент для создания анимации, но также анимацию можно создать предварительно в 3D-редакторе и импортировать вместе с моделью, а затем разбить на файлы. Unity 3D поддерживает систему Level Of Detail (LOD), суть которой заключается в том, что на дальнем расстоянии от игрока высоко детализированные модели заменяются на менее детализированные, и наоборот, а также систему Occlusion culling, суть которой в том, что у объектов, не попадающих в поле зрения камеры, не визуализируется геометрия и коллизия, что снижает нагрузку на центральный и графический процессоры и позволяет оптимизировать проект. При компиляции проекта создается исполняемый файл (*.exe) игры, а в отдельной папке – данные игры, включая все игровые сцены и динамически подключаемые библиотеки. Модели, звуки, текстуры, материалы, скрипты можно запаковывать в формат *.unityassets и передавать другим разработчикам или выкладывать в свободный доступ. Этот же формат используется во внутреннем магазине Unity Asset Store, в котором разработчики могут бесплатно или за деньги выкладывать в общий доступ различные элементы, нужные при создании игр.

Unity предоставляет API (*англ.* application programming interface) на ЯП C# для написание так называемых скриптов. Скрипт или сценарий – основной компонент разработки в среде Unity. В отличии от ситуаций, когда сценариями называют автономные служебные программы, в Unity сценарии больше напоминают классы в ООП, а присоединенные к объектам сцены скрипты являются экземплярами класса. Следует отметить, что для

функционирования кода необходимо чтобы описанный в нем класс был потомком MonoBehavior – базового класса компонентов-сценариев. Этот класс определяет способ присоединения компонентов к игровым объектам. Наследование от этого класса дает некоторые автоматически запускаемые методы, как Start(), вызываемый при активации игрового объекта, или Update() – метод, вызываемый в каждом кадре. Кроме этого, скрипты используются при разработке пользовательского интерфейса, искусственного интеллекта, различных контроллеров, отвечающих за сетевые и внутрепрограммные функции.

Как правило игровой движок предоставляет множество функциональных возможностей, позволяющих задействовать их в различных играх. В их число входят моделирование физических сред, карты нормалей, динамические тени и многое другое. В отличие от многих игровых движков, у Unity имеется два основных преимущества: наличие визуальной среды разработки и межплатформенная поддержка. Первый фактор включает не только инструментарий визуального моделирования, но и интегрированную среду, цепочку сборки, что направлено на повышение производительности разработчиков, в частности, этапов создания прототипов и тестирования. Под межплатформенной поддержкой предоставляется не только места развертывания, но и наличие инструментария разработки. Еще одним преимуществом является модульная система компонентов Unity, с помощью которой происходит конструирование игровых объектов, когда последние представляют собой комбинируемые пакеты функциональных элементов. В отличие от механизмов наследования, объекты в Unity создаются посредством объединения функциональных блоков, а не помещения в узлы дерева наследования. Такой подход облегчает создание прототипов, что актуально при разработке игр. В качестве недостатков можно выделить ограничение визуального редактора при работе с многокомпонентными схемами, когда в сложных сценах визуальная работа затрудняется. Вторым недостатком является отсутствие поддержки в Unity ссылок на внешние

библиотеки, работу с которыми программистам приходится настраивать самостоятельно; это также затрудняет командную работу.

Само приложение, разработанное в среде Unity, можно представить как бесконечный цикл графической отрисовки постоянной изменяющейся сцены. Перед тем как построить новый кадр, или в терминологии компьютерной графики «произвести рендер», приложение осуществляет выполнение всех методов Update() в скриптах сцены. Они (скрипты) могут всяческим образом воздействовать на объекты сцены, интерфейс, производить вычисление неких параметров, получать или отправлять данные по сети.

## 1.2 Принятие решений

Компонент принятия решений отвечает за способность агента выбирать подходящее дальнейшее действие, исходя из текущей обстановки. Иными словами, имея некоторый набор входных данных агент должен выработать соответствующие выходные данные – решение о том, какое из возможных действий ему следует предпринять в данный момент.

Сами входные данные или «знания» можно разбит на внешние – сведение об окружении (позиция агента, расположение других агентов относительно ткущего и т.д.); и внутренние – информация о состоянии агента (здоровье, экипировка и т.д.). После осуществления действия эти данные меняются.

Как правило у агента имеется ограниченный набор поведений, который сохраняется до определенного события, после которого идет смена поведения. Таким образом для реализации компонента принятия решений лучше всего подходят конечные автоматы состояний.

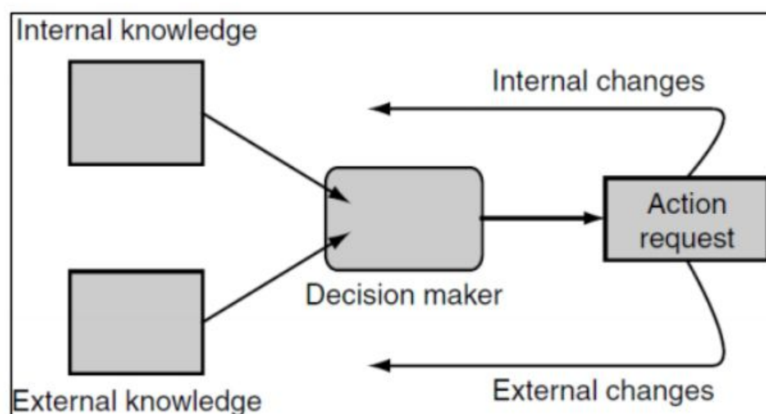


Рисунок 1 – Схема принятия решений

Формальное определение конечного автомата состояний [1] звучит следующе: *конечный автомат – это устройство или модель устройства, которое имеет конечное число состояний, в которых оно может находиться в любой момент времени, и может работать с входными данными, чтобы либо совершать переходы из одного состояния в другое, либо осуществлять выход или необходимое действие. Конечный автомат может находиться только в одном состоянии в любой момент времени*.

Исторически конечный автомат – это жестко формализованное устройство, используемое математиками для решения задач. Самым известным конечным автоматом, вероятно, является гипотетическое устройство Алана Тьюринга: машина Тьюринга, о которой он писал в своей статье 1936 года «О вычислимых числах». Это была машина, предвосхитившая современные программируемые компьютеры, которые могли выполнять любые логические операции путем чтения, записи и стирания символов на бесконечно длинной ленте.

Идея конечного автомата, состоит в том, чтобы разложить поведение объекта на легко управляемые «фрагменты» или состояния. Например, выключатель света на стене – это очень простой конечный автомат. У него есть два состояния: включено и выключено. Переходы между состояниями производятся нажатием на переключатель.

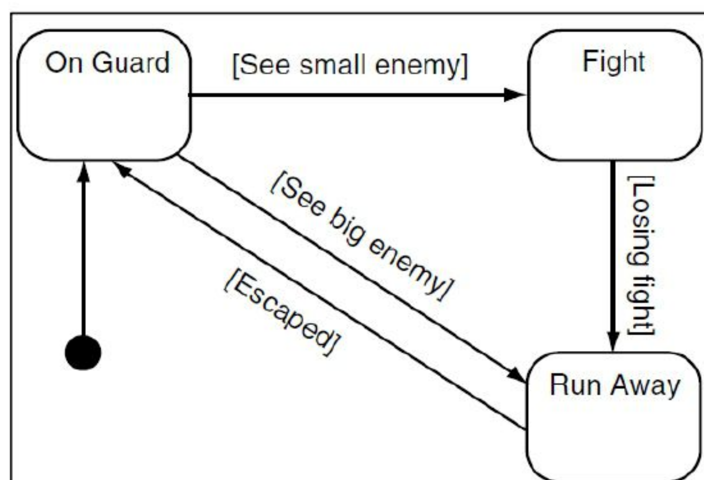


Рисунок 2 – Пример простого конечного автомата состояний для агента

Конечные автоматы в течение многих лет были излюбленным инструментом программистов ИИ, чтобы наполнить игровых агент иллюзией интеллекта. Конечные автоматы того или иного типа можно найти практически в каждой игре, которые появлялись на прилавках с первых дней видеоигр, и, несмотря на растущую популярность более продвинутых архитектур, они будут существовать еще долгое время [1]. Вот лишь некоторые из причин, почему [1]:

– их легко и быстро запрограммировать. Существует много способов программирования конечного автомата, и почти все они достаточно просты в реализации;

– их легко отлаживать. Поскольку поведение игрового агента разбито на легко управляемые фрагменты, если агент начинает вести себя странно, его можно отладить, добавив код трассировки в каждое состояние. Таким образом, программист может легко проследить последовательность событий, предшествующих ошибочному поведению, и принять соответствующие меры;

– у них небольшие вычислительные затраты. Конечные автоматы почти не используют драгоценное процессорное время, потому что они, по сути, следуют жестко запрограммированным правилам. Нет никакого настоящего «мышления», кроме мыслительного процесса «если-то»;

– они интуитивно понятны. Человеческой природе свойственно думать о вещах как о находящихся в том или ином состоянии. Конечно, люди на самом деле не работают как конечные автоматы, но иногда нам полезно думать о своем поведении таким образом. Точно так же довольно легко разбить поведение игрового агента на несколько состояний и создать правила, необходимые для управления ими. По той же причине конечные автоматы также позволяют легко обсуждать дизайн ИИ с «непрограммистами», например, с гейм-дизайнерами, упрощая общение и обмен идеями.

– они гибкие. Конечный автомат игрового агента может быть легко настроен программистом для обеспечения поведения, требуемого разработчиком игры. Также несложно расширить область действия агента, добавив новые состояния и правила. Кроме того, конечные автоматы обеспечивают прочную основу, с которой возможно комбинировать и другие методы, такие как нечеткая логика или нейронные сети.

### 1.3 Передвижение

Компонент передвижения отвечает за непосредственное перемещение агента в пространстве сцены, основываясь на выбранном на уровне принятия решений действии. Этот компонент в свою очередь можно разделить на два слоя [1]:

– управление (англ. *steering*) – этот слой отвечает за расчет желаемых траекторий, необходимых для достижения целей и планов, установленных компонентом принятия решений.

– движение (англ. *locomotion*) – нижний уровень, представляет более механические аспекты перемещения агента. Это способ путешествия из пункта А в пункт Б. Отделив этот слой от уровня управления, можно использоватьодно и то же поведение управления для совершенно разных типов передвижения;

Слой движения в большей степени связан с физическим движком приложения и не связан с системой ИИ. По этой причине, подходы к его реализации не будут рассматриваться в рамках этой работы.

Для реализации слоя управления используется подход «Рулевое поведение» (англ. Steering Beheviours), предложенный Крейгом Рейнальдсом в своей публикации «Steering Behaviors for Autonomous Characters»[2]. Суть этого метода заключается в вычислении так называемых рулевых сил (анг. Steering Force), основываясь на внешних и внутренних данные. Каждая из этих сил моделирует определенное базисное поведение, такое как преследование, бегство или блуждание. После вычисления всех таких базисных сил, из присвоенного агенту набора, происходит вычисление результирующей силы. На этом этапе может быть использован как простой результирующий вектор, ограниченный по длине, так и более продвинутые подходы, один из которых будут рассмотрены далее в этой главе.

В следующих пунктах будет рассмотрено вычисление рулевых сил, использованных в программной реализации.

### 1.3.1 Рулевые силы «Поиск» и «Бегство»

Первыми и самыми основными из рулевых сил являются «Поиск» (англ. «Seek») и «Бегство» (англ. «Flee»).

«Поиск» вычисляет силу, направляющую агента в сторону позиции цели. Сперва вычисляется желаемая скорость – вектор скорости, необходимый агенту для достижения позиции цели в идеальном мире (без учета препятствий и ландшафта). Этот вектор вычисляется разностью между позицией агента и позицией цели, и ограничением полученного вектора по длине на максимальную скорость агента.

Рулевая сила «Поиск» вычисляется как разность желаемой скорости и текущей скорости агента. Сила «Бегство» вычисляется как разность обратного вектора желаемой скорости и текущей скорости агента.

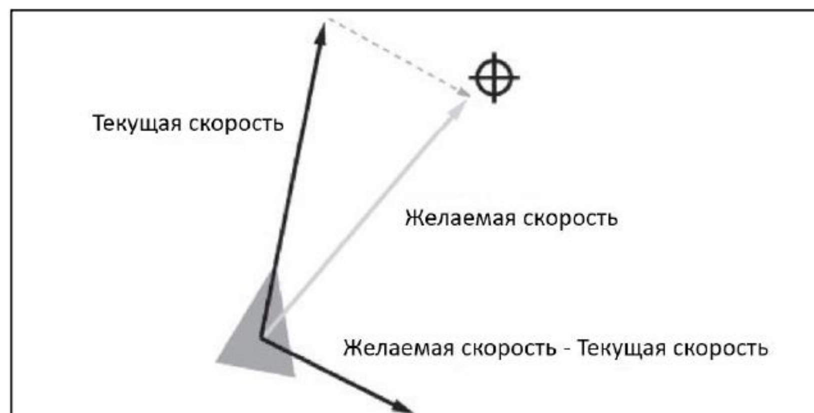


Рисунок 3 – Вычисление силы «Поиск»

### 1.3.2 Рулевая сила «Преследование»

«Преследования» полезно, когда агенту требуется перехватить движущуюся цель (рис. 4). Конечно, он мог бы продолжать использовать «Поиск» относительно текущей позиции цели, но это мешало бы создать иллюзию интеллекта.

Успех функции преследования зависит от того, насколько хорошо преследователь может предсказать траекторию убегающего. Это может оказаться достаточно сложным, поэтому необходимо пойти на компромисс, дабы получить адекватный результат, не потребляя при этом слишком много тактовых циклов процессора.

Стоит отметить, что существует одна краевая ситуация, с которой может столкнуться преследователь: если убегающий находится спереди почти прямо перед агентом, агент должен направиться прямо к текущей позиции убегающего.

Также важным аспектом является то, насколько наперед агент должен рассчитывать будущую позицию цели. Это значение $t$ вычисляется по формуле (1) ниже:

$$t = \frac{dist}{(v_{agent} + v_{target})}, \tag{1}$$

где

$t$ – промежуток времени,

$dist$ – текущее расстояние между агентом и целью,

$v_{agent}$ – текущая скорость агента,

$v_{target}$ – текущая скорость цели.

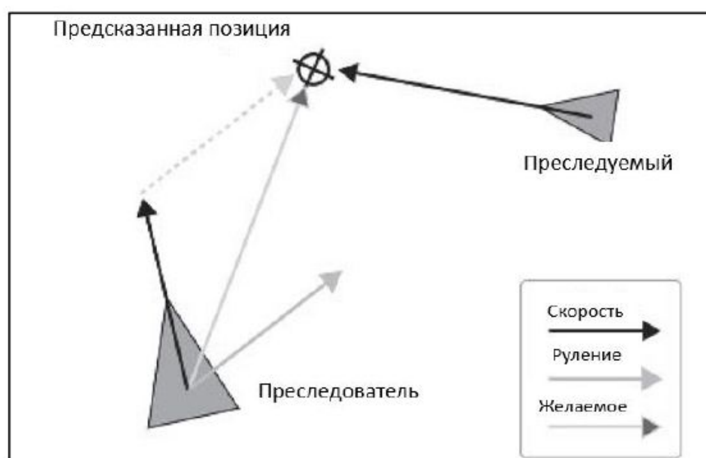


Рисунок 4 – Преследование агентом цели

Итак, вычисление силы «Преследование» осуществляется следующим образом:

– проверяется, находится ли цель перед агентом с углом отклонения от его текущего направления в диапазоне $(-\tau; \tau)$. Если это так – возвращаем значение «Поиск» до текущего положения цели. Иначе переходим на следующий шаг;

– вычисляем период времени $t$ по формуле (1) для расчета позиции цели;

– рассчитываем позицию цели используя ее текущий вектор скорости и значение $t$;

– возвращаем значение силы «Поиск» относительно вычисленной позиции;

### 1.3.3 Рулевая сила «Блуждание»

Часто возникает необходимость в создании у агента иллюзии бесцельного перемещения по окружению. Одним из примеров может служит имитации поведения различных животных в привычной им среде.

Простейшим подходом для достижения этой цели могло бы служить случайное вычисление направления на каждом временном шаге, но это приводит к нестабильному поведению, при котором невозможно достичь длительных устойчивых поворотов. Альтернативой может служить применение таких псевдослучайных функций, как шум Перлина, однако это достаточно затратное по производительности решение, особенно для большого количества агентов.

Решение, предложенное Рейнольдсом состоит в том, чтобы проецировать окружность (рис. 5) перед агентом и направлять его к цели, которая вынуждена двигаться по этой окружности. На каждом временном шаге к этой цели добавляется небольшое случайное смещение, и со временем она перемещается вперед и назад, создавая реалистичное чередующееся движение без дрожания. Этот метод может использоваться для создания целого диапазона случайных движений, от очень плавных волнистых поворотов до вихрей и пируэтов, в зависимости от размера круга, расстояния от него до агента и величины случайного смещения в каждом кадре.

Более формальное описание шагов представлено ниже:

1) инициализация:

а) определяются значения констант, такие как радиус R и дистанция S блуждания;

б) высчитывается случайная позиция P на окружности радиусом R вокруг агента (вычисляются случайные координаты x, y со значениями из диапазона [-1; 1] и умножаются на R);

2) блуждание:

а) к позиции P прибавляется некоторое случайное смещение, в результате получаем позицию P'. Затем P' проецируется на окружность радиуса R и полученное значение присваивается P;

б) к новой позиции P прибавляется дистанция S.

Действий из пункта 1) проделываются лишь единожды, в то время как действия из пункта 2) – при каждом вычислении новой точки. На рисунке 5 изображена схема процесса.

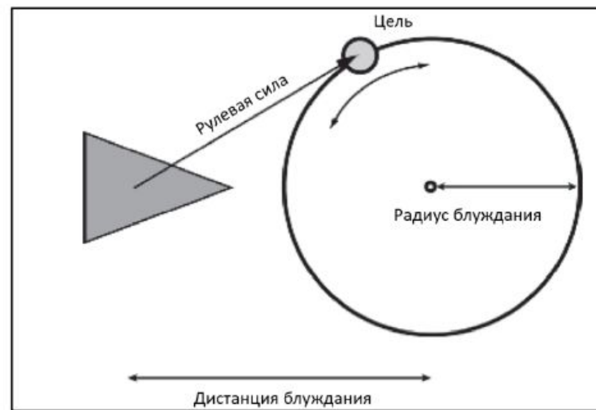


Рисунок 5 – Вычисление силы «Блуждание»

### 1.3.4 Рулевая сила «Избегание стен»

Под стеной будем понимать либо линию в случае двухмерного пространства, либо полигон в случае трех измерений. У такой стены обязательно есть нормаль, указывающая сторону, в которую она обращена.

Рассматриваемая рулевая сила возвращает направление призванное повернуть агента во избежание столкновения. Это происходит путем, проецирования трех «щупалец» перед агентом и проверяя, пересекаются ли они с какими-либо стенами в игровом мире (рис. 6). Когда будет найдена ближайшая пересекающаяся стена, происходит вычисление силы путем расчета того, насколько далеко наконечник «щупальца» проник через стенку, и затем создания силы такой величины в направлении нормали стены.

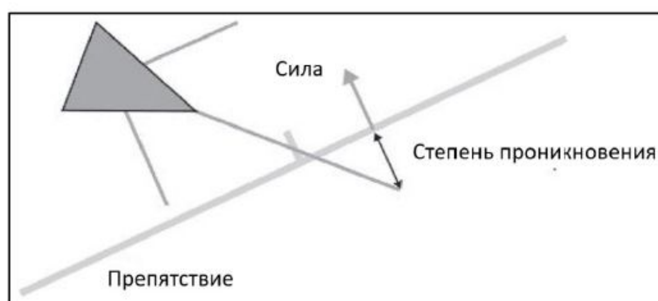


Рисунок 6 – Вычисление силы «Избегание стен»

### 1.3.5 Вычисление результирующей силы

Финальным этапом расчёта рулевых сил является вычисление результирующего вектора. В программной реализации используется подход приоритетного колебания, предложенный Крейгом Рейнальдсом [2]. Его суть заключается в следующем:

– всем рулевым силам присваивается приоритет от низшего в наивысшему и значение вероятности их срабатывания;

– для каждой силы в порядке приоритета вычисляется случайное значение из диапазона [0; 1] и в случае, если оно меньше либо равно установленной для этой силы вероятности – происходит ее расчет и возвращение этого значения как результирующего, иначе – переход к следующей по приоритету для вычисления;

– если для рассматриваемой силы значение равно 0, то рассматривается следующая по приоритету;

Этот метод требует гораздо меньше процессорного времени, чем другие, однако в ущерб точности. Кроме того, он требует детальной настройки вероятностей для каждой силы, прежде чем удастся получить желаемое поведение.

### 1.4 Планирование пути

Компонент планирования пути отвечает за построение агентом маршрута по виртуальному окружению. Его можно разбить на два подкомпонента:

– алгоритм поиска кратчайшего пути в графе;
– система переноса виртуального окружения в формат графа.

#### 1.4.1 Алгоритм поиска пути A*

Для поиска пути используется алгоритм A*, представляющий собой модификацию алгоритма Дейкстры.

A* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая $g(x)$ – это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение $f(x)$, после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа – множеством частных решений, – которое размещается в очереди с приоритетом. Приоритет пути определяется по значению $f(x) = g(x) + h(x)$. Алгоритм продолжает свою работу до тех пор, пока значение $f(x)$ целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Чем меньше эвристика *h(x)*, тем больше приоритет, поэтому для реализации очереди можно использовать сортирующие деревья.

Множество просмотренных вершин хранится в множестве *closed*, а требующие рассмотрения пути – в очереди с приоритетом *open*. Приоритет пути вычисляется с помощью функции *f(x)* внутри реализации очереди с приоритетом.

Временна́я сложность алгоритма A* зависит от сложности эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x)) \quad (2)$$

где $h^*$ – оптимальная эвристика, то есть точная оценка расстояния из вершины $x$ к цели.

Другими словами, ошибка *h(x)* не должна расти быстрее, чем логарифм от оптимальной эвристики.

Но ещё бо́льшую проблему, чем временна́я сложность, представляют собой потребляемые алгоритмом ресурсы памяти. В худшем случае ему приходится помнить экспоненциальное количество узлов.

### 1.4.2 Система переноса окружения в формат графа

Как правило, виртуальное пространство в видеоигре представлено 2D- или 3D-сценой с разной степенью сложности геометрии, будь то город, пещера или набор комнат. Для того, чтобы использовать алгоритм поиска пути, необходимо перевести данные об окружении в формат графа и работать уже с ним. Можно выделить 3 аспекта этого процесса [3]:

– схема разбиения – непосредственный механизм переноса информации окружении в формат графа;

– локализация – процесс переноса вершины графа в некоторую точку или области в пространстве окружения;

– квантование – процесс переноса конкретной точки окружения в соответствующую вершину графа;

В реализации используется подход, основанный на точках видимости. Он основам на факте, что кратчайший путь через окружение всегда будет иметь точки перегиба [3][4] в выпуклых вершинных геометрии этого окружения. Эти точки можно использовать в качестве вершин в навигационном графе для воссоздания правдоподобного пути агента. Вершины связаны ребром в том случае, если между соответствующими им точкам можно провести линию, которая не пересекается с каким-либо препятствием – иными словами из одной точки можно увидеть другую (рис. 7).

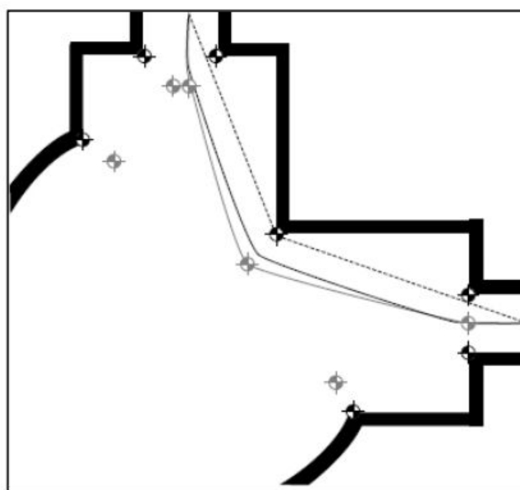

Рисунок 7 – Путь с использованием точек видимости

Одной из особенностей навигационных графов на основе точек видимости является то, что они могут быть легко расширены, чтобы включать в себя узлы с дополнительной информацией, как например различные позиций для стрельбы, укрытия или засады. Обратной стороной является то, что, если игровое окружение достаточно сложная, дизайнер карты может потратить очень много драгоценного времени, позиционируя и настраивая граф.

Другим недостатком является то, что при генерации окружения процедурно, также необходимо задуматься о разработке механизма автоматического размещения точек видимости. Одним из решений этой проблемы является использование методов расширенной геометрии. Если игровая среда построена из многоугольников, можно использовать информацию, представленную в этих фигурах для размещения точек видимости. Это достигается сначала расширением полигонов на величину, пропорциональную радиусу игровых агентов. Вершины, определяющие эту расширенную геометрию, затем добавляются как вершины в навигационный граф. Наконец, запускается алгоритм для проверки прямой видимости между вершинами, и ребра добавляются к графу соответствующим образом. На рисунке 8 изображено построение такого графа.

Рисунок 8 – Построение графа на основе точек видимости

Поскольку многоугольники расширяются на величину не меньше ограничивающего радиуса агента, агент может выполнять поиск в результирующем навигационном графе, чтобы создать пути, которые безопасно пересекают среду, не натыкаясь на стены.

## 2 Процедурная генерация тестового окружения

С целью отладки системы автономных агентов в различных окружениях, в рамках курсовой работы так же разработана система процедурной генерации сцены. Она происходит в два этапа:

– случайная генерация исходных данных регулярной сетки и ее сглаживание с использованием клеточных автоматов;

– построение полигональной сетки для 3D-модели окружения;

Далее рассмотрим подходы для реализации этих этапов.

### 2.1 Генерация регулярной сетки

В первую очередь происходит случайная генерация бинарной матрицы размерности $n \times m$. Эта матрица представляет собой начальные данные будущей регулярной сетки. Для случайно генерации может использоваться как генератор случайных чисел с плавающей точкой от 0 до 1, так и функция шума Перлина.

Далее имея исходную матрицу, необходимо «сгладить» ее с использованием клеточного автомата.

Кле́точный автома́т – дискретная модель, изучаемая в математике, теории вычислимости, физике, теоретической биологии и микромеханике. Включает регулярную решётку ячеек, каждая из которых может находиться в одном из конечного множества состояний, таких как 1 и 0. Решетка может быть любой размерности. Для каждой ячейки определено множество ячеек, называемых окрестностью. К примеру, окрестность может быть определена как все ячейки на расстоянии не более 2 от текущей. Для работы клеточного автомата требуется задание начального состояния всех ячеек и правил перехода ячеек из одного состояния в другое. На каждой итерации, используя правила перехода и состояния соседних ячеек, определяется новое состояние

каждой ячейки. Обычно правила перехода одинаковы для всех ячеек и применяются сразу ко всей решётке.

Один из способов смоделировать двумерный клеточный автомат - использовать лист миллиметровой бумаги и набор правил, которым должны следовать клетки. Каждый квадрат называется «ячейкой», и каждая ячейка имеет два возможных состояния: черное и белое. Окрестность клетки – это соседние, обычно смежные клетки. Двумя наиболее распространенными типами окрестностей являются окрестности фон Неймана и окрестности Мура. [5] «Игра жизни» Конвея - популярная версия этой модели.

Обычно предполагается, что каждая ячейка вселенной начинается в одном и том же состоянии, за исключением конечного числа ячеек в других состояниях; присвоение значений состояния называется конфигурацией. В более общем плане иногда предполагается, что Вселенная изначально покрыта периодическим рисунком, и только конечное число ячеек нарушает этот узор. Последнее предположение является обычным для одномерных клеточных автоматов.

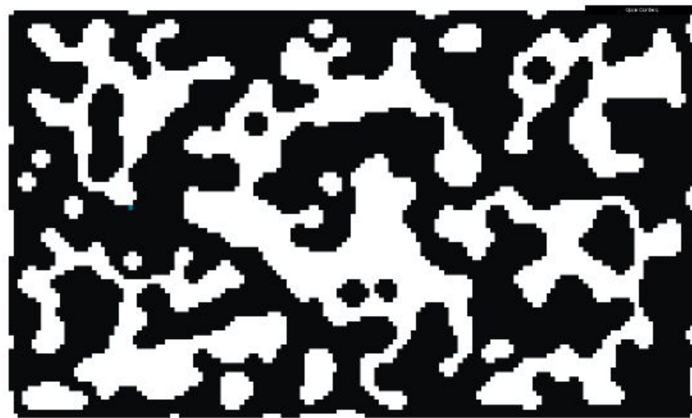

Рисунок 9 – Пример клеточного автомата

Клеточные автоматы часто моделируются на конечной сетке, а не на бесконечной. В двух измерениях Вселенная была бы прямоугольником, а не бесконечной плоскостью. Очевидная проблема с конечными сетками заключается в том, как обрабатывать ячейки на краях. То, как они будут

обрабатываться, повлияет на значения всех ячеек в сетке. Один из возможных способов – оставить значения в этих ячейках постоянными. Другой метод – по-разному определить окрестности для этих ячеек. Можно было бы сказать, что у них меньше соседей, но тогда также нужно было бы определить новые правила для ячеек, расположенных по краям. С этими ячейками обычно обращаются по тороидальному расположению: когда одна выходит сверху, одна входит в соответствующее положение внизу, а когда одна идет слева, одна идет справа. Это можно представить как склеивание левого и правого краев прямоугольника, чтобы сформировать трубу, а затем склейку верхней части. и нижние края трубки, чтобы сформировать тор. Это решает граничные проблемы с окрестностями.

### 2.2 Генерация полигональной сетки

Имея сглаженную регулярную сетку, как на рисунке 9, можно приступить к созданию полигональной сетки и формированию 3D-модели. Для этого будет использоваться алгоритм компьютерной графики «движущиеся квадраты» (англ. *Marching Squares*).

На вход алгоритм получает регулярную сетку, в каждом узле которой известно значение поля. Выходная сетка может иметь меньшее разрешение (в этом случае теряется точность, но уменьшается ступенчатость). Далее для каждого узла выходной сетки проверяется, выше ли значение в нем, чем на изоповерхности. Всем узлам, которые выше, приписывается «+», остальным «-». Далее рассматриваются квадраты выходной сетки, вершины которых лежат в отмеченных узлах. Всего получается 16 различных случаев, которые с учетом симметрий и поворотов можно свести к четырем (рис. 10):

- случай 1: все вершины имеют один знак;
- случай 2: у одной вершины знак отличается;
- случай 3: вершины с одинаковыми знаками имеют общее ребро;

– случай 4: вершины с одинаковыми знаками не имеют общего ребра;

В четвертом случае невозможно однозначно определить форму сегмента изолинии, поэтому дополнительно просматривается значение в центре квадрата.

Для улучшения качества получаемой изолинии применяется линейная интерполяция. В таком случае конец сегмента изолинии на ребре квадрата делит ребро в отношении $\frac{f_1-c}{c-f_2}$, где $f_1, f_2$ – значения на концах ребра квадрата, $c$ – значение изолинии. Фактически, конец сегмента изолинии «подтягивается» к тому концу ребра, который ближе к реальной изолинии.

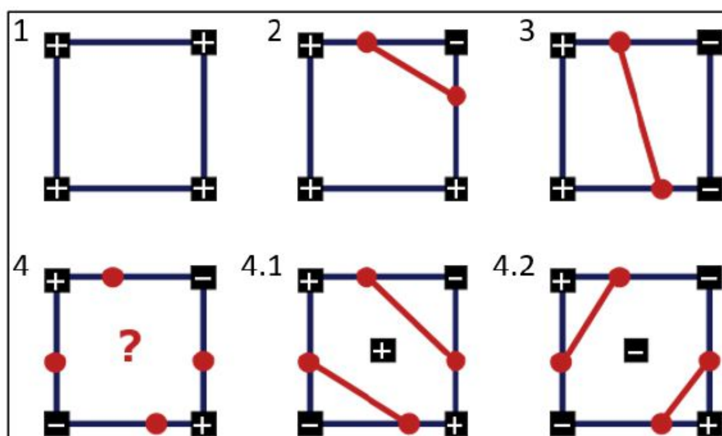


Рисунок 10 – Возможные случаи ячейки выходной сетки

## 3 Программная реализация

В рамках программное реализации разработана система из двух типов агентов – преследователь и жертва. Каждый из них реализует соответствующее поведение.

Прежде, чем перейти к рассмотрению реализации компонент системы из главы 1, рассмотрим разработанный фреймворк, на котором строится вся система. На рисунке 11 можно наблюдать его структуру в виде UML-диаграммы.

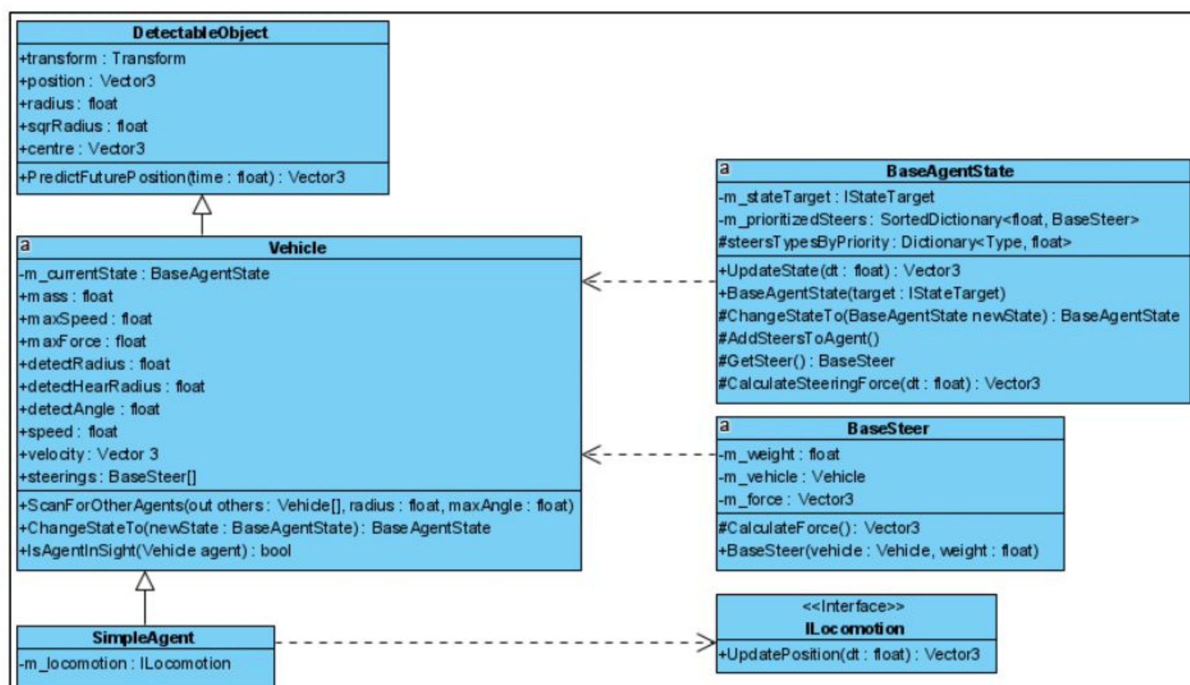


Рисунок 11 – UML-диаграмма иерархии классов DetectableObject

Иерархия классов начинается с DetectablObject – абстракции некоторого неподвижного объекта сцены, для которого характерна позиция, данные о форме и размеры. Он также задает интерфейс предоставления будущей позиции для себя и своих подклассов.

Следующим в иерархии находится класс Vehicle – абстракция передвигающегося объекта сцены. Для него характерны уже такие свойства, как скорость, направление, масса и др.

Класс SimpleAgent представляет собой конкретную реализацию абстрактных классов-предков с конкретной реализацией системы передвижения через интерфейс ILocomotion.

Абстрактные классы BaseAgentState и BaseSteer представляют собой базовые классы для состояния агента и вычисляемой рулевой силы соответственно.

Построенная на представленном фреймворке система обладает достаточной гибкостью и изолированностью различных компонент, а также открыта для расширения.

## 3.1 Реализация компонента принятия решений

Для реализации компонента принятия решений использовались конечные автоматы состояний, описанные в пункте 1.2.

На рисунке 12 изображена UML-диаграмма классов этой подсистемы.

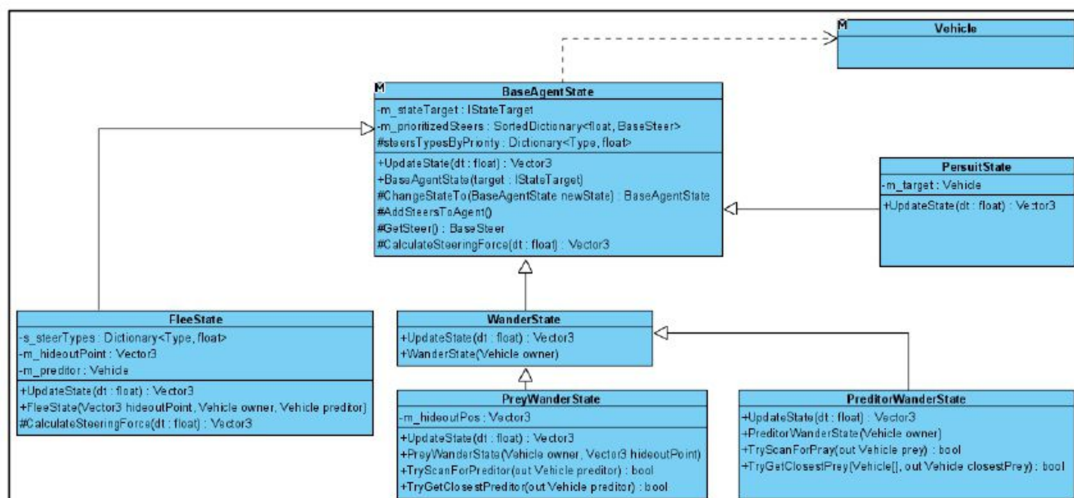


Рисунок 12 – UML-диаграмма классов состояний

Как видно на диаграмме, все конкретные состояния являются подклассами базового класса BaseAgentState. Кроме этого, стоит отметить, то этот класс является «дружественным» для класса Vehicle, т.е. имеет доступ к некоторым его приватным членам, недоступным из вне. Таким образом все суперклассы-состояния могут влиять на экземпляр агента, но лишь в рамках,

предусмотренных в BaseAgentState. Это дает большую гибкость при создании новых видов состояний и не нарушает инкапсуляцию базового класса агента.

На рисунке 13 изображен конечный автомат агента «Хищник». Всего предусмотрено 3 состояния: блуждание-поиск, преследование и атака. Соответственно поведение агента заключается в перемещении по окружению до тех пор, пока он не наткнется на жертву. Далее он начнет ее преследовать до тех пор, пока не сблизится достаточно, чтобы поймать, либо пока цель не отдалится слишком далеко.

Рисунок 13 – Конечный автомат состояний «хищника»

На рисунке 14 изображен конечный автомат агента «жертва». В нем имеется также 3 состояния: блуждание, побег и укрытие. Его поведение можно описать следующим образом: агент перемещается по сцене до тех пор, пока не обнаружит «хищника». Далее он пытается добраться до укрытия не будучи пойманным. В случае, если ему это удается – он перестает находиться в зоне досягаемости и пропадает со сцены.

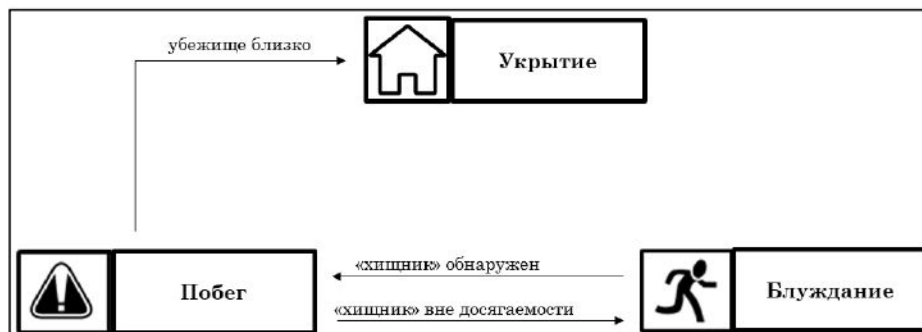


Рисунок 14 – Конечный автомат состояний «жертвы»

## 3.2 Реализация перемещения

При реализации компонента перемещения использовались описанные в главе 1 рулевые силы, а именно «Поиск», «Бегство», «Преследование», «Избегание стен»и «Блуждание». На рисунке 15 изображена UML-диаграмма классов этого компонента.

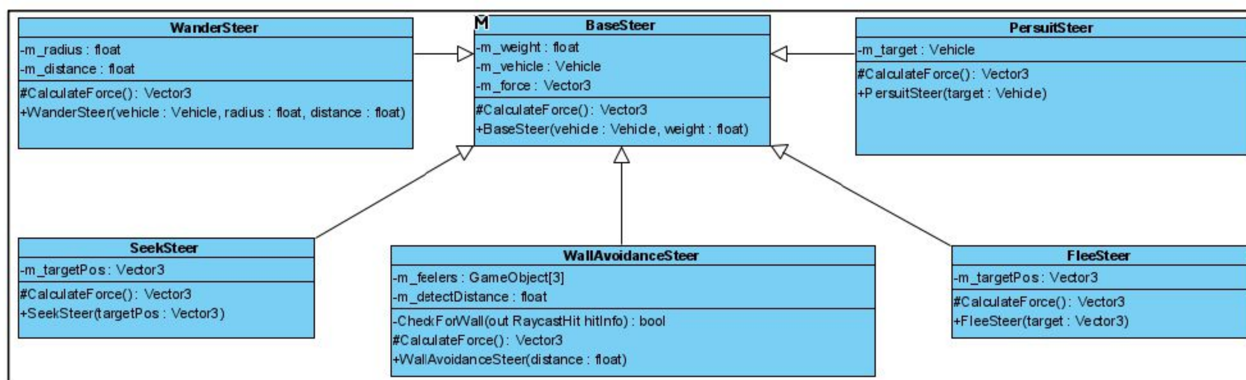


Рисунок 15 – UML-диаграмма иерархии классов BaseSteer

Все классы используют общий интерфейс, в базовом классе BaseSteer. В дальнейшем возможно расширение набора рулевых сил, без необходимости существенных изменений в системе. Приведенная структура предполагает метод вычисления результирующего вектора методом приоритетных колебаний из пункта 1.3.5 и хранит характерный для каждой силы приоритет в поле *m_weight*.

## 3.3 Реализация построения маршрута

Для осуществления поиска маршрута в первую очередь было необходимо создать эффективную структуру для хранения данных навигационного графа. Для этого использовал список смежных вершин в графе. На рисунке 16 изображена UML-диаграмма классов, используемых для представления графа в памяти компьютера.

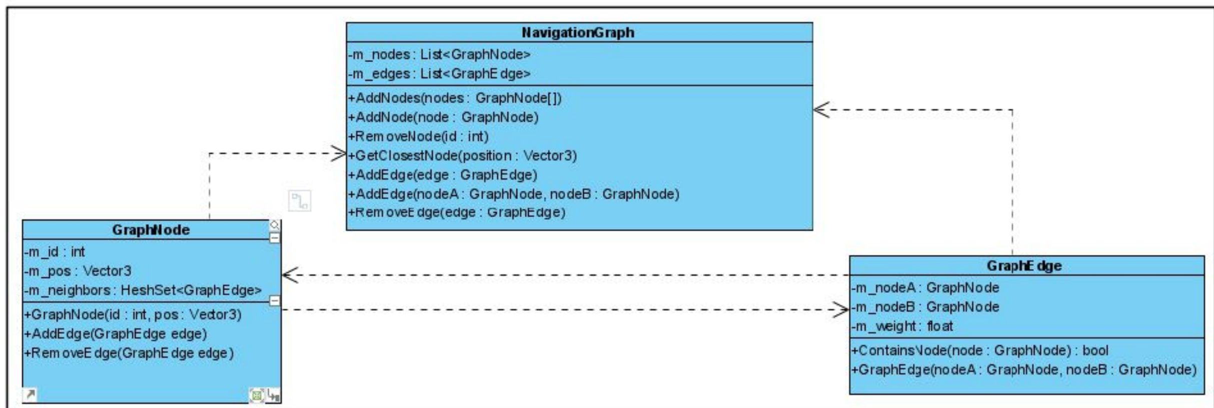


Рисунок 16 – Классы представления графа

В представленной структуре каждая вершина хранит идентификатор, позицию в 3D-сцене в виде координат и список своих соседей. Кроме этого, для экономии процессорного времени при поиске пути, также хранятся нагруженные значением расстояния ребра графа без необходимости повторного вычисления.

Алгоритм A* поиска пути вынесен в отдельный полностью статичный класс. В дальнейшем это позволит производить вычисления для различных агентов полностью параллельно. На вход метод получает ссылку на экземпляр графа, координату текущей позиции и координату цели. Далее определяются вершины начала и конца пути. Они представляют собой ближайшие вершины для входных координат, до которых можно пробросить непересекающийся с препятствиями луч.

В алгоритме присутствует оптимизация для хранения «открытых» вершин в виде приоритетной очереди, основанной на бинарной куче. Это позволяет извлекать следующую для рассмотрения вершину со скорость $O(1)$ вместо $O(n)$, что существенно экономит время при поиске.

Навигационный граф строится на основе точек видимости, которые могут быть как расставлены вручную в редакторе сцены [6] Unity, так и автоматически при генерации тестового окружения основываясь на полученной геометрии. На рисунке 17 представлена визуализация

навигационного графа в редакторе Unity. Красными точками обозначены вершины графа, серые лини – ребра полученного графа.

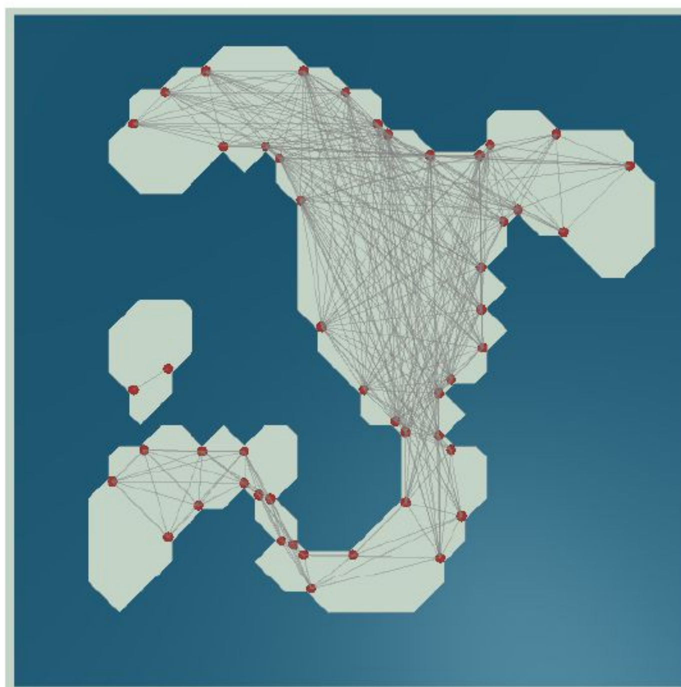

Рисунок 16 – Визуализации навигационного графа

### 3.4 Реализация генерации окружения

Для отладки системы автономных агентов была разработана система процедурной генерации окружения.

В первую очередь в ней задается размерность окружения в условных единицах в формате 2d-вектора, плотность «стен» в диапазоне (0, 1). Далее идет генерация исходной бинарной регулярной сетке, как на рисунке 17.

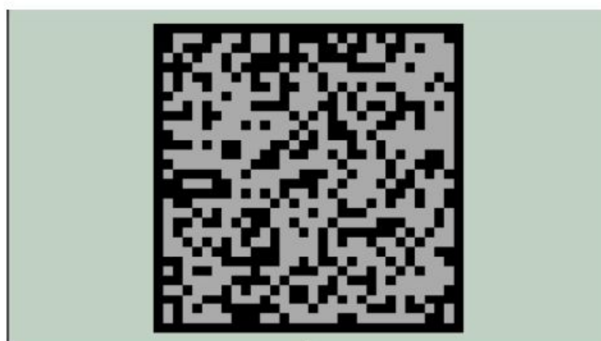

Рисунок 17 – Пример исходной регулярной сетки

Далее методом клеточного автомата идет сглаживание регулярной сетки. Для каждой не краевой клетки используется следующее правило: если количество соседних клеток со значением 1 больше либо равно 5, то значение рассматриваемой клетки так же равно 1, иначе 0. Осуществляя сглаживание в пять итераций, получаем результат, как на рисунке 18.

Рисунок 18 – Результат сглаживания клеточным автоматом

Далее на основе сглаженной регулярной сетки методом движущихся квадратов строится плоская полигональная сетка (рис. 19). Внутренняя область считается свободной. По ее контуру добавляются полигоны, формируя стены. После этого вычисляются выпуклые углы в полученной геометрии и размещаются точки видимости. Для этого рассматриваются вершины контура внутренней области по часовой стрелки. В случае, если угол между нормалями стен соединенной точкой контура является больше 180 градусов – точка является выпуклой. В таких точках методом расширенной геометрии размещаются точки видимости с учетом радиуса агента.

Навигационный граф строится на последнем этапе при перебрасывании лучей между всеми точками видимости. На рисунке 20 изображён итоговый вид сцены. На ней зеленый цилиндр – агент-жертва, красный – агент-хищник, желтый прямоугольник – область убежища жертвы. Линиями и красными точками отображен навигационный граф.

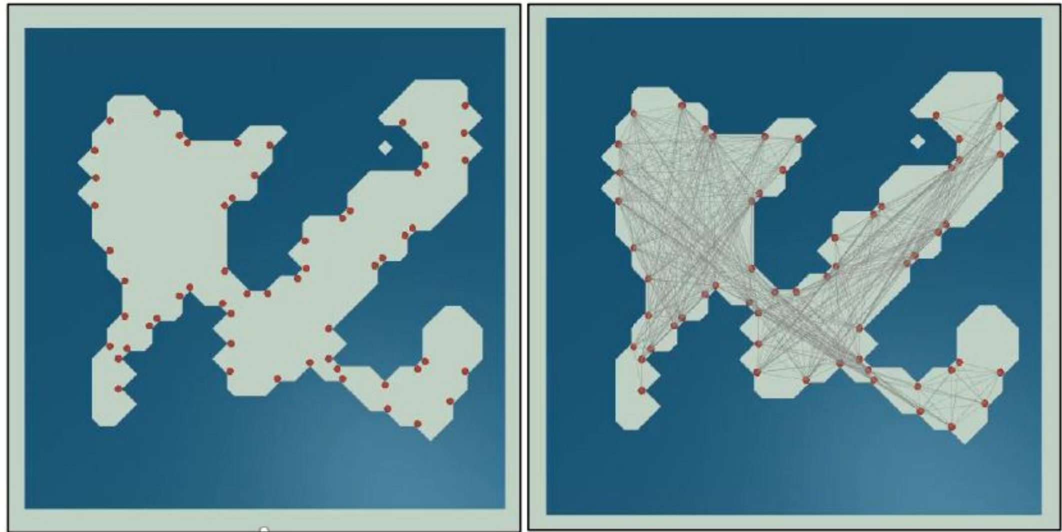

Рисунок 19 – Итог процедурной генерации

### 3.5 Результаты работы

При запуске приложения агенты перемещаются по сцене до тех пор, пока не обнаружат друг друга. После этого их поведение меняется: зеленый агент пытается отдалиться от красного и по возможности добраться вдоль построенного маршрута до убежища. Красный агент преследует зеленного до тех пор, пока расстояние между ними не сократиться до определенного значения.

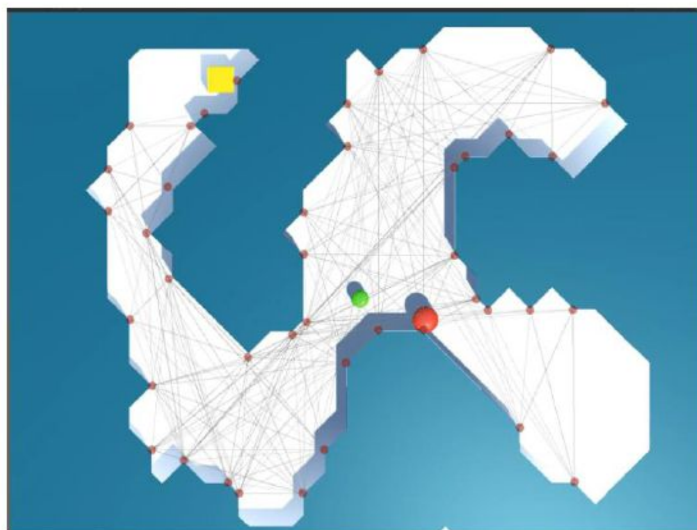

Рисунок 20 – Взаимодействие агентов

# ЗАКЛЮЧЕНИЕ

В итоге в рамках курсовой работы были изучены актуальные подходы и методы реализации искусственного интеллекта, также системы автономных агентов для видеоигры и технологий процедурной генерации окружения.

Разработан прототип приложения системы автономных агентов, со следующими возможностями:

- корректное перемещение по виртуальному пространству;
- следование определенному набору поведений;
- планирование маршрута по виртуальному окружению;
- реагирование на других агентов системы.

Архитектура системы построена таким образом, чтобы иметь возможность дальнейшего расширения и модификации различных ее компонент независимо друг от друга.

Разработана система процедурной генерации тестового окружения для отладки системы искусственного интеллекта.

Дальнейшее развитие системы включает:

- расширение набора Steering Behaviors;
- переход от точек видимости к навигационному мешу при создании навигационного графа окружения;
- внедрение параллельных вычислений и перенос всей системы на Unity DOTS;
- исследование альтернатив FSM, таких как нечеткая логика и гибридные подходы.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Buckland, M. Programming Game AI by Example (Wordware Game Developers Library) 1st Edition / Wordware Publishing, Inc., Texas, USA, 2005. – с. 172.

2. Reynolds, C. Steering Behaviors For Autonomous Characters / Sony Computer Entertainment America, 2002 // (Eng.) – URL: https://www.researchgate.net/publication/2495826_Steering_Behaviors_For_Autonomous_Characters (дата обращения: 03.05.2021).

3. Millington, I. Artificial Intelligence for Games / I. Millington, J. Funge. – Morgan Kaufmann Publishers, Burlington, USA, 2009. – с. 265.

4. Rabin, S. AI Game Programming Wisdom / Charles River Media, Boston, USA, 2002. – с. 201.

5. Bialynicki-Birula, I. Modeling Reality: How Computers Mirror Life / Oxford University Press, 2004. – с. 276.

6. Unity User Manual (2018.04): официальный сайт / США, Сан-Франциско – URL: www.docs.unity3d.com/2018.4/Documentation/Manual/index.html (дата обращения 05.04.2021).