МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РОССИЙСКИЙ ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ Г. В. ПЛЕХАНОВА»

Высшая школа кибертехнологий, математики и статистики
Кафедра цифровой экономики института развития информационного общества

«Допустить к защите»

Заведующий кафедрой
Цифровой экономики
института развития
информационного общества
Уринцов Аркадий
Ильич

_____
(подпись)

«____» _____ 202_ г.

Выпускная квалификационная работа бакалавра

Направление «Математическое обеспечение и администрирование информационных систем»
Профиль «Системное и интернет-программирование»

Разработка алгоритма для управления группой объектов для преследования цели в играх

Выполнил студент Федоров Кирилл Константинович

Группа 15.11Д-МО12/19б

Научный руководитель
выпускной квалификационной работы

_____
(ФИО, звание, степень, должность)

_____
(подпись)

Автор _____
(подпись)

Москва – 2023

Содержание

Оглавление

Введение

Глава I

1.1 Введение в предметную область

Алгоритмы преследования используются в разных областях жизни. Опираясь на диссертацию Ляпина Н.А., искусственный интеллект, а конкретно разные разновидности алгоритмом преследования широко используются при разработке вооружения, таких как БПЛА, БЛА или ПВО. Так, в основу БЛА закладывается алгоритм патрулирования обозначенной территории и уничтожение цели при ее обнаружении. В свою очередь, в ВПК при разработке разного вида вооружения так же используется противоположный алгоритму преследования – алгоритм уклонения. Он используется при разработке разного вида снарядов и ракет в качестве обхода ПВО, которое в свою очередь использует алгоритм преследования.

Так же, алгоритмы преследования используются в геоинформационных системах (ГИС). Они предназначены для управления большим количеством разномасштабной картографической информации, анализа взаимосвязей объектов в пространстве, управления атрибутными характеристиками объектов. Все моделируемые в ГИС объекты и явления имеют пространственную привязку, позволяющую анализировать их во взаимосвязи с другими пространственно-определенными объектами. Модули пространственного анализа и принятия решения средствами ГИС являются ядром геоинформационных технологий. На основе ГИС создаются комплексные программные решения для поддержки объектов инфраструктуры в течение их жизненного цикла: при проектировании, создании и эксплуатации транспортных сетей, энергоснабжения. ГИС уже доказали свою эффективность при поддержке эксплуатации систем энергоснабжения на электротранспорте, в частности, на городском электротранспорте и на железнодорожном транспорте.

Алгоритмов преследования и уклонения нашли широкое применение в разработке компьютерных игр. В настоящее время искусственный интеллект

широко применяется при разработке игр. Игровой искусственный интеллект — это набор программных методов, которые используют в видеоиграх для создания иллюзии разума у NPC (non-player character) через поведение персонажей. Игровой ИИ включает в себя алгоритмы теории управления, робототехники, компьютерной графики и информатики в целом. Зарождение искусственного интеллекта в видеоиграх началось до того, как сама индустрия стала неотъемлемой частью жизни практически каждого человека. Один из наиболее ранних и громких прецедентов использования этой технологии в игре относится к 1950-м годам.

Алгоритмы преследования и уклонения являются базовыми алгоритмами при проектировании центрального процессора принятия решения у игровых объектов в различных игровых жанрах. В основном они используются как одно из состояний конечного автомата игрового ИИ, означающее преследование заданной цели. Например, данная группа алгоритмов используется для ботов-патрульных, которые охраняют определённую территорию и если игрок заходит в поле зрение, то данные боты становятся ищейками, которые будут преследовать игрока. Однако, классический алгоритм преследования не решает множество проблем, связанных с преследованием цели – например, обход полигонов или непроходимых препятствий на карте.

В данной главе будут продемонстрированы различные алгоритмы преследования, способы решения проблемы обхода полигонов в определённой местности и математические модели написания искусственного интеллекта для управления группой объектов. Так же, в этой главе будут выбран и детально проанализирован оптимальный набор алгоритмический и математических моделей для решения поставленной задачи - разработка алгоритма, управляющего группой объектов для преследования цели.

1.2 Анализ математических и алгоритмических методов

Алгоритмы преследования

Метод погони

Этот метод ещё называют чистым преследованием, преследованием по кривой погони. Методом погони называется метод преследования, при котором вектор ПЕ*Vпе в любой момент времени направлен на цель, то есть его курсовой угол α равен нулю. Проанализировав данный метод, можно сделать вывод, что при методе погони, преследователь приближается к цели сзади и кривая погони характеризуется большой кривизной, при любых начальных условиях. Таким образом, даже в условиях отсутствия каких-либо маневров со стороны цели, это приводит к низкой точности преследования, что является недостатком метода. Достоинством этого метода преследования является его помехоустойчивость – для реализации метода в каждый момент времени надо знать только, слева или справа от вектора преследования находится цель, для корректировки курсового угла соответствующим образом.

Метод постоянного угла упреждения

Это метод, при котором курсовой угол в любой момент времени равен некоторой фиксированной величине α0. Величина α0 должна подчиняться условию (Vпе /Vпо)*|sinα0| < 1, в противном случае преследователь начнет описывать вокруг цели бесконечную спираль, так её и не достигнув. Этот метод является модификацией метода погони, но у него есть достоинство в том, что при использовании угла упреждения кривая погони гораздо менее искривлена, чем для метода погони. Кроме этого, метод обладает похожей помехоустойчивостью, что и метод погони. Однако для реализации этого метода необходима информация о пеленге цели, а также о направлении движения цели для выбора правильного угла упреждения.

Метод параллельного сближения

Параллельным сближением называется вид преследования, когда линия визирования всегда смещается параллельно самой себе. Если цель движется

прямолинейно и равномерно, то траектория преследователя есть прямая. При маневрах цели, когда цель получает ускорение, для сохранения условия параллельности линии визирования, ускорение преследователя будет совпадать с нормальным ускорением цели. Это является достоинством данного метода. К недостаткам метода можно отнести большое количество требуемой информации: курсовой угол цели, скорость.

Метод пропорционального наведения

У него нет тех недостатков, которые есть у метода параллельного сближения. Это такой метод, при котором угловая скорость преследователя пропорциональна угловой скорости линии визирования. Назначение этого метода – поражение цели, с учетом тенденции поворота линии визирования. Как следует из принципов действия этого метода, для его реализации необходима лишь информация о пеленге цели.

Оптимальные алгоритмы преследования

Оптимальными называются алгоритмы, который дают возможность получить оптимальное решение задачи управления с четко определенным функционалом (целевой функцией). Решение задач оптимального управления усложняется, при усложнении движения целевого объекта. Когда цель движется равномерно и прямолинейно, возможно аналитическое решение. Что невозможно в случае, когда цель маневрирует, используя сложные пространственные траектории. Подобные задачи называются стохастическими задачами преследования. Использование теории оптимального управления в решении задач наведения является важной составляющей, часто использующейся на практике, например, в частном случае преследования цели, движущейся под острым углом к встречному курсу. В таком случае классические алгоритмы погони и постоянного угла упреждения не будут работать. Теория оптимального управления позволяет получить алгоритмы преследования, которые могут решать и задачу преследования быстродвижущихся целей. Но недостатком данных алгоритмов

является большая сложность в реализации при сложных траекториях искомого объекта.

Дифференциальные игры преследования

Одной из классических дифференциальных игр является задача преследования – уклонения, когда преследователь пытается минимизировать (а цель максимизировать) время, за которое преследователь настигнет цель.

Опираясь на источник, оптимальной стратегией является метод погони. При разных игровых моделях, а также различных управлениях, оптимальными стратегиями могут стать стратегии, реализующиеся аналогами классических алгоритмов со всеми их достоинствами и недостатками.

Учитывая то, что перед данной дипломной работой не стоит задача реализации сложных алгоритмов преследования, которые применяются в военно-промышленной области, например, учитывающие физические нюансы, классический алгоритм преследования, или алгоритм погони, наиболее подходит для поставленной цели. Однако, использование обычного метода погони без каких-либо изменений не вполне рационально, потому что предполагается, что на местности, на которой будет работать данный алгоритм, будут располагаться препятствия в виде полигонов разной величины. Опираясь на источник, использование обычного метода погони, приведет к тому, что объекты-преследователи не будут способны обходить препятствия и просто будут застревать в них, пытаясь пройти через них. К тому же задача дипломной работы заключается в реализации алгоритма, управляющего группой объектов. Из статьи следует, что в играх в основном используется комбинированная модель поведения объектов – они управляются единым интеллектом, однако конкретные функции, например преследование, реализуется на конкретном объекте-преследователе. По сути, единый интеллект распоряжается лишь состояниями подконтрольными объектами.

Для решения задачи обхода полигонов существует множество методов, имеющие принципиально разные концепции и разные входные данные.

Методы решения проблемы обхода полигонов

Проанализировав определённое количество литературы, существует пять основных метода обхода полигонов в играх. У всех есть достоинства и недостатки. В данном разделе будут проанализированы данные пять методов, попутно оценивая их.

Метод с предварительной оценкой местности

Статья предлагает метод для решения проблемы обхода полигонов, основывающийся на предварительной оценке карты местности и последующей корректировки знаний. Карта, в свою очередь, представляется в виде двумерного массива, состоящая из единиц и нулей. Каждая ячейка этого массива представляет собой отдельную клетку, где 0 – клетка проходима, а 1 – не проходима. Так же, данный метод имеет два случая – NPC (Non-Player Character) «знают» местоположение игрока и обратный случай.

В первом, когда NPC «знаю» местоположение игрока, ситуация в понятиях игрового искусственного интеллекта называется cheat-методом. Игрок ставится в неравное положение по отношению к неигровым персонажам, и тем самым повышается сложность игры. Проведенный предварительный анализ проблемы показал, что проблема преследования обычно решается двумя основными способами:

- использование алгоритмов поиска пути;
- комбинация алгоритмов поиска пути и алгоритмов преследования.

С точки зрения практической реализации, первый способ является самым простым. При изменении местоположения игрока пути до него просчитываются заново. Подобный подход имеет серьезные недостатки, связанные с большими временными затратами, а также с большим потреблением памяти.

Если в качестве алгоритма поиска пути выбран алгоритм A*, то временные затраты, в худшем случае, растут экспоненциально, аналогично экспоненциально растут затраты памяти.

Второй способ более сложный. Путь до игрока вычисляется с помощью алгоритмов поиска пути. Если расстояние между игроком и неигровым персонажем становится меньше R, то используется более простой и менее ресурсоемкий алгоритм преследования. Эффективность такого метода определяется, прежде всего, величиной радиуса видимости R и размером карты.

NPC «не знаю» о местоположении игрока. Методы из данной группы добавляют в игру больший реализм. Данные методы требуют, чтобы неигровые персонажи были в состоянии предсказывать местоположение игрока. В простейшем случае они могут просто двигаться в случайных направлениях. Как только игрок попадает в область видимости неигрового персонажа, дальнейшее преследование осуществляется при помощи соответствующих алгоритмов. Таким образом, первоочередной задачей становится задача оценки вероятности появления игрока в той или иной ячейке карты.

К разработанному алгоритму на этапе проектирования были выдвинуты следующие требования:

- результатом его работы должна быть матрица весов, каждый элемент которой–вес конкретной ячейки игровой карты;
- первоначальный расчет должен проводиться с учетом положения входа и выхода и структуры карты. Таким образом, в функционировании алгоритма можно выделить три этапа:
- этап инициализации;

  На данном этапе проводится инициализация матрицы весов

  Вычисления на данном этапе происходят в конце каждой игровой сессии.

- этап корректировок.

В ходе работы алгоритма возникают ситуации, когда необходимо нормировать значения в матрице весов.

Описанный метод был реализован в компьютерной игре Paclight. На произвольных картах размерности 40 на 40 метод показал свою пригодность. Проведенные наблюдения показали, что на картах такого размера время, затраченное на расчет путей, было меньше примерно в 1,2 раз, а по сравнению с применение только A* (в случаях, когда NPC знают местоположение игрока). Ожидается, что с увеличением размерности карт выгода от применения метода предварительной оценки карт возрастет.

Хранение информации о местности

Наиболее простой способ обойти проблему столкновения объектов-преследователей с полигонами состоит в том, чтобы сохранить некоторую "информацию об окружении" (ИО) внутри каждого объекта. Это означает, что если объект утыкается в твердый объект, то он мог бы "спросить" этот объект, куда двигаться, чтобы обойти его.

В зависимости от стороны, с которой произошло столкновение, каждое препятствие возвращает значения пары смещения dx/dy относительно объекта. Правила, для нахождения этих значений:

- Допустим, объект свободно перемещается по экрану, к примеру, из левой верхней к правой нижней его части. И он соприкоснулся с левой стороной препятствия, тогда пускай он движется вниз.
- Если в препятствие ударились по направлению какой-нибудь оси, возвратим противоположное направление по этой же оси. Если препятствие имеет ИО (-1, 1) и получает удар от чего-то движущегося по оси X, следует возвратить 1.

Очевидный недостаток этого приема в том, чтобы сохранить другие два значения для каждого объекта. Возможным решением данной проблемы станет сохранение данной информации, используя всего лишь 4 бита.

Движение объекта-преследователя по выпускаемым лучам

Идея данного алгоритма в выпускании от каждого бота преследователя 2 отрезка сенсора, направленные на игрока. При пересечении полигона лучом, данный луч должен поворачиваться (один поворачивается по часовой стрелке, другой против) до тех пор, пока они не перестанут пересекать полигоны и далее объект-преследователь движется по тому лучу, который быстрее добрался до преследуемой цели и раньше перестал пересекать полигоны.

Однако, у этой идеи есть свои недостатки. Во-первых, она достаточно трудоемкая в силу того, что на каждой итерации надо проверять от каждого объекта-преследователя, лучи пересекаются ли они с каждым ребром полигона, которых потенциально может быть много на местности. Во-вторых, данный алгоритм не всегда приводит к ожидаемым результатам. Если длина луча слишком коротка, то бот может зайти в тупик, из которого придется выбираться.

Например:

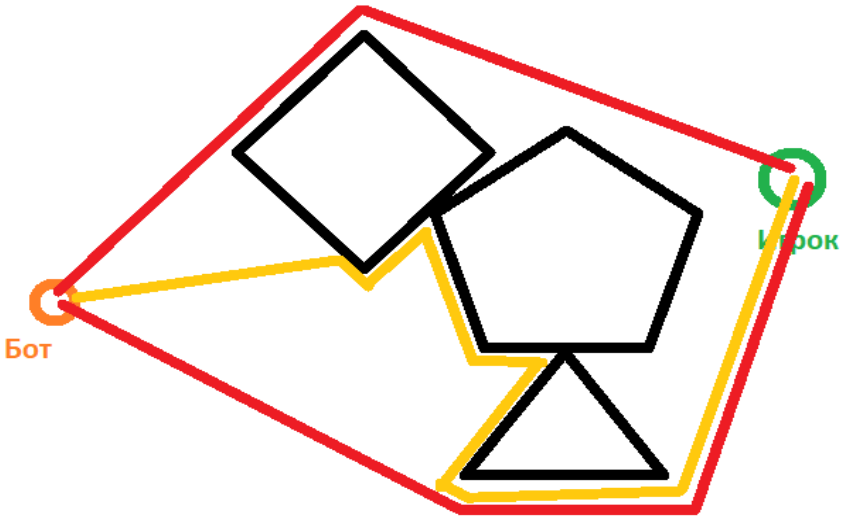


Рисунок 1. Демонстрация работа алгоритма

На Рисунок 1, бот пойдет оранжевым путем, когда красный явно оптимальнее.

Однако, если длина луча слишком длинная, то бот, наоборот, примет узкий проходит за тупит и будет обходить его, как на Рисунок 2.

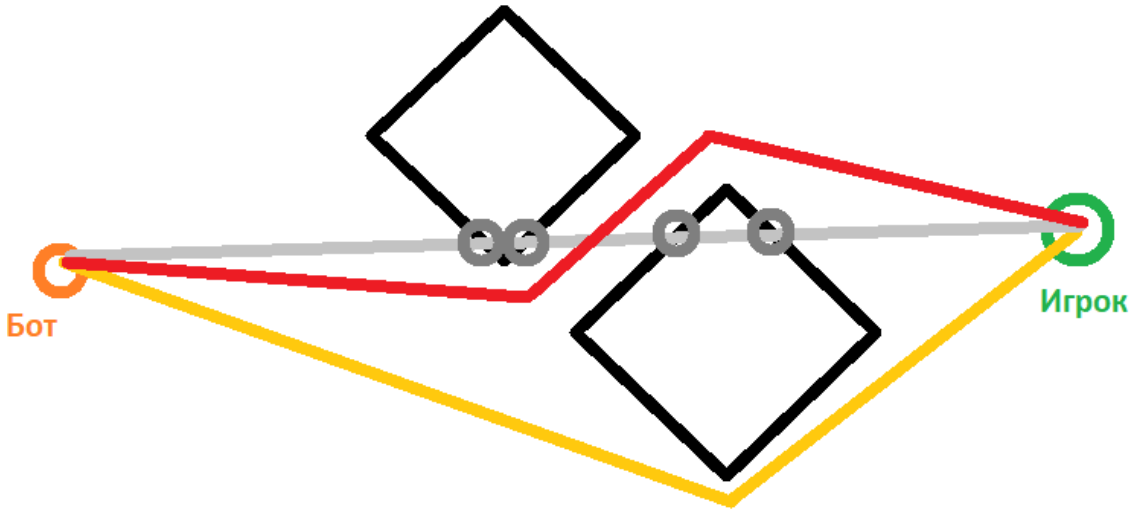


Рисунок 2. Некорректная работа алгоритма

Серый вектор на Рисунок 2 – луч, пересекший полигоны. Здесь бот пойдет по оранжевому пути, хотя красный оптимальнее.

Частично эти проблемы нивелируются, если игрока, объектов-преследователей и вершины полигонов представить в виде вершин графа и совершать поиск наикратчайшего пути, например при помощи алгоритма Дейкстры.

Однако, проблема оптимизации остается прежней. Чем больше карта, полигонов и объектов-преследователей, тем медленнее работает алгоритм, потому что для каждой итерации придется пересчитывать алгоритм Дейкстры на большое дерево и для каждого объекта преследователя.

Метод ищейки

Данный метод основывается на том, что преследуемый объект оставляет следы запаха. Объекты-преследователи, когда не видят преследуемый объект, стараются ориентироваться по оставляемым следам как ищейки. В основу метода закладывается конечный автомат состояний в объект-преследователя. Когда объект видит цель, он просто старается сближаться с ней при помощи классического алгоритма преследования. Когда объект-преследователь не видит цели, то его состояние переключается на ориентирование по оставляемым следам.

Этот метод имеет ряд преимуществ. Во-первых, он не ограничен размерами карты, ее формой, количеством полигонов и формой полигонов. Во-вторых, данный алгоритм энергоемкий по времени и по памяти, опираясь на статью, описывающая данный алгоритм.

*Алгоритм A**

Алгоритм A* находит *оптимальный путь* из начальной в конечную точку, избегая по дороге препятствия. Он реализует это, постепенно расширяя множество *частичных путей*. Каждый частичный путь — это серия шагов от начальной точки до какой-то промежуточной точки на дороге к цели. В процессе работы A* частичные пути становятся всё ближе конечной точке. Алгоритм прекращает работу тогда, когда находит полный путь, который лучше оставшихся вариантов, и это можно доказать.

На каждом шаге алгоритма A* оценивает множество частичных путей и генерирует новые пути, расширяя наиболее многообещающий путь из множества. Для этого A* хранит частичные пути в очереди с приоритетами, отсортированном по *приблизительной длине* — истинной измеренной длине пути плюс примерное оставшееся расстояние до цели. Это приближение должно быть *недооценкой*; то есть приближение может быть меньше истинного расстояния, но не больше него. В большинстве задач поиска пути хорошей преуменьшенной оценкой является геометрическое расстояние по прямой от конца частичного пути до конечной точки. Истинный наилучший путь до цели от конца частичного пути может быть длиннее, чем это расстояние по прямой, но не может быть короче.

Когда A* начинает работу, очередь с приоритетами содержит всего один частичный путь: начальную точку. Алгоритм многократно удаляет из очереди с приоритетами наиболее многообещающий путь, то есть путь с наименьшей приблизительной длиной. Если этот путь завершается в конечной точке, то алгоритм выполнил задачу — очередь с приоритетами гарантирует, что никакой другой путь не может быть лучше. В противном случае, начиная с

конца частичного пути, который он удалил из очереди, A* генерирует ещё несколько новых путей, делая единичные шаги во всех возможных направлениях. Он помещает эти новые пути снова в очередь с приоритетами и начинает процесс заново.

Преимуществом данного алгоритма является его не привязанность к конкретной местности и типу полигонов. Недостатком же является то, что данный алгоритм может вызвать оптимизационные трудности и вызвать замедление работы системы в целом.

Анализ преимуществ и недостатков рассмотренных методов обхода полигонов

Для демонстрации преимуществ и недостатков каждого из алгоритмов обхода полигонов будет использоваться табличный формат.

Как видно из Таблица 1 ниже, единственным алгоритмом, который обладает высокой производительностью, является Метод ищейки. Исходя из требования, что алгоритм должен быть максимально возможно гибким и подходить для любого типа местности и полигонов, то алгоритм ищейки наиболее подходит для данной цели, так как остальные алгоритмы имеют ограничения по поводу размера и типа местности. К тому же он наиболее ресурсоемкий, так остальные алгоритмы либо требуют много времени на начальной инициализации, либо требуют очень много ресурсов в процессе работы.

Однако, метод ищейки не будет работать, если объект-преследователь не видит ни цели, ни следов. Поэтому, нужен алгоритм, который будет работать несмотря на то, что преследуемый объект находится вне видимости. Наилучшими характеристиками среди алгоритмов, которые работает несмотря на то, что объекта нет в зоне видимости, является Алгоритм A*.

Как итог, для решения проблемы обхода полигонов будет использоваться комбинационный вариант двух алгоритмов.

Таблица 1. Табличный вид преимуществ и недостатков, описанный методов обхода полигонов

Название алгоритма	Особенности алгоритмов				
	Не ограничен типом местности	Не ограничен типом полигонов	Высокая производительность	Точный	Объект должен быть в зоне видимости
Метод с предварительной оценкой местности	✓	×	×	✓	×
Хранение информации о местности	×	✓	×	×	×
Движение объекта-преследователя по выпускаемым лучам	×	✓	×	×	×
Метод ищейки	✓	✓	✓	×	✓
Алгоритм A*	✓	✓	×	✓	×

Групповое мышление объектов

Ранее были рассмотрены алгоритмы и математические модели, применяемые для одного конкретного объекта. Задача данной работы разработать алгоритм, управляющий объектами-преследователями. По сути, задача разработать искусственный интеллект, который в зависимости от игровой ситуации будет менять поведение подконтрольных объектов. Искусственный интеллект нужен для имитации разумности NPC, при этом его задача не в том, чтобы обыграть пользователя, а в том, чтобы развлечь его. В

современных играх используются разные подходы для создания ИИ. Опираясь на статью, в основе лежит общий принцип: получение информации → анализ → действие.

Получение информации происходит примерно так же, как и в реальном мире — у ИИ есть специальные сенсоры, при помощи которых он исследует окружение и следит за происходящим. Сенсоры бывают совершенно разными. Это может быть традиционный конус зрения, «уши», которые улавливают громкие звуки, или даже обонятельные рецепторы. Конечно, такие сенсоры — всего лишь имитация реальных органов чувств, которая позволяет сделать игровые ситуации более правдоподобными и интересными. Наличие и реализация сенсоров зависит от геймплея (игровой процесс). Во многих активных шутерах (жанр игр, основанный на высокой концентрации боев) не нужны комплексные рецепторы — достаточно конуса зрения, чтобы реагировать на появление игрока. А в стелс-экшенах (жанр игр, основанный на «бесшумном» прохождении игры) весь геймплей основан на том, чтобы прятаться от противников, поэтому виртуальные органы чувств устроены сложнее.

Когда ИИ получил информацию, он начинает «обдумывать» свои действия, анализируя обстановку. Обычно в этом участвует сразу несколько систем ИИ, отвечающих за разные вещи. Часто разработчики добавляют подобие коллективного интеллекта, который следит за тем, чтобы действия отдельных агентов не противоречили и не мешали друг другу. При этом сами мобы зачастую даже не знают о существовании своих союзников — эта информация им не нужна, потому что за координирование действий отвечает ИИ более высокого уровня.

В играх есть несколько подходов, которые чаще всего используются для принятия решения. Один из самых простых и понятных подходов — это rule-based ИИ. В основе лежит список правил и условий, заранее созданный разработчиками. Такой подход можно эффективно использовать для создания

простого поведения. Например, «если игрок приближается к курице ближе, чем на три метра, то она начинает от него убегать».

Следующий распространённый способ принятия решений — конечные автоматы (КА, finite state machine, FSM). Этот подход позволяет NPC беспроблемно переходить между разными состояниями. Например, есть моб, базовое состояние которого — патрулирование по определённой траектории. Если внезапно появится игрок, NPC перейдёт в новое состояние — начнёт стрелять. Конечные автоматы как раз обеспечивают эти переходы: они принимают информацию с предыдущего состояния и передают в новое.

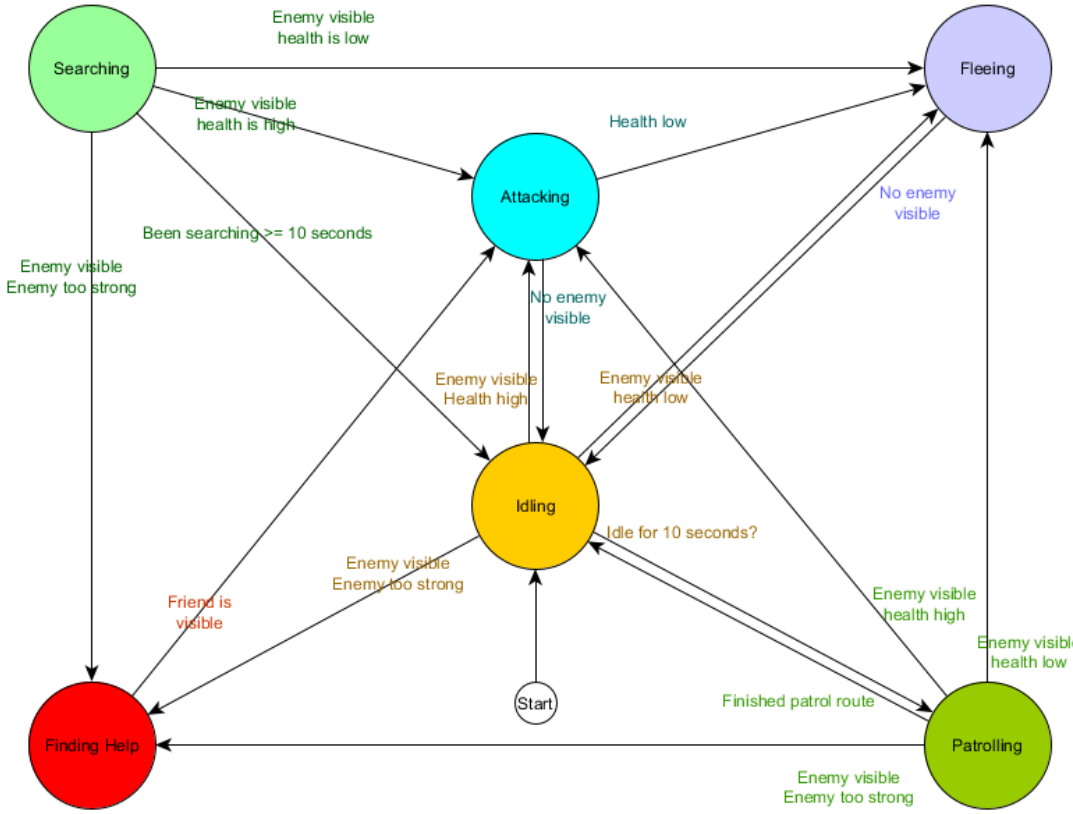


Рисунок 3. Пример конечного автомата принятия решения

Преимущество этого подхода в том, что персонаж всегда будет находиться в каком-то состоянии и не зависнет где-то между ними. Так как разработчик должен прописать все переходы, он точно знает, в каких состояниях может находиться игровой объект. Недостаток метода в том, что с увеличением количества механик значительно разрастается и система

конечных автоматов. Это увеличивает риск появления ошибок, а также может снизить скорость операций.

Дерево поведения — это более формализованный подход построения поведения мобов. Его особенность заключается в том, что все состояния персонажа организованы в виде ветвящейся структуры с понятной иерархией. Дерево поведения содержит в себе все возможные состояния, в которых может оказаться моб. Когда в игре происходит какое-то событие, ИИ проверяет, в каких условиях находится NPC, и перебирает все состояния в поисках того, которое подойдёт для нынешней ситуации.

Дерево поведения отлично подходит для того, чтобы систематизировать состояния NPC в играх, в которых есть множество механик и игровых элементов. В ситуации, когда моб участвует в перестрелке, ему не нужно будет искать подходящее действие в ветке патрулирования. Такой подход помогает сделать поведение NPC отзывчивым и обеспечивает плавный переход между разными состояниями.

Иерархические конечные автоматы объединяют особенности конечных автоматов и дерева поведения. Особенность такого подхода в том, что разные графы внутри логики могут отсылаться друг к другу. Например, нам надо прописать поведение для нескольких мобов. Не обязательно делать для каждого отдельную логику — можно создать общее базовое поведение и просто отсылаться к нему при необходимости.

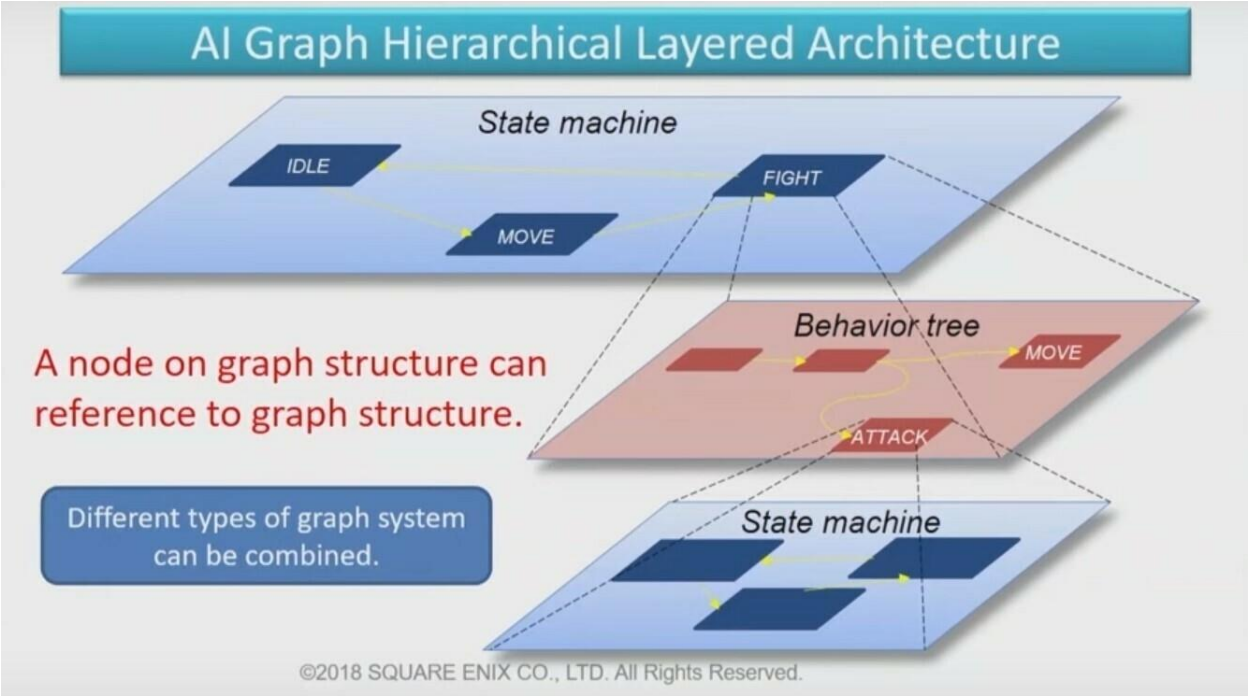


Рисунок 4. Иерархическая архитектура в Final Fantasy XV.

Есть и менее популярные решения, которые не смогли полноценно закрепиться в индустрии. К примеру, в F.E.A.R. использовалась система целеориентированного планирования действий (Goal-Oriented Action Planning, GOAP) — для всех NPC она создаёт план действий, основываясь на информации об игровом мире. Например, если мобу нужно перейти в другую комнату, то система сперва проверяет, какое расстояние нужно пройти до двери, есть ли на пути препятствия, открыта ли дверь и так далее. Когда у системы есть вся информация об окружении, она составляет план, а NPC просто проигрывает последовательность анимаций.
Этот подход работает на основе конечных автоматов, но они отвечают только за воспроизведение анимаций. У автоматов есть всего три состояния, каждое из которых отвечает за свой набор анимаций: движение (бег, ходьба), действия (стрельба, реакции), взаимодействие с объектами (открыть дверь, включить свет).

Очевидно, что выбор определенной модели поведения объектов зависит от целей и задач, которые они должны выполнять. Данная работа преследует

цель создать групповой, алгоритм, который будет способен управлять подконтрольными объектами для преследования цели. В общем случае цепочка принятия решения сводится к обнаружению объекта любым из подконтрольных объектов и началу движения за ним всех остальных подконтрольных объектов. Если объект-жертва стоит на месте, то задача группового алгоритма сводится к окружению данной жертвы так, чтобы она не могла выйти из окружения. Для этой цели подходит модель конечного автомата, так как всего есть три состояния: состояние спокойствия, состояние преследования, состояние окружения. Они будут переключаться в зависимости от игровой ситуации.

Так же, в задачу группового алгоритма войдет предиктивная система оценивания будущей позиции преследуемого объекта. Данная система будет работать, пока алгоритм пытается преследовать цель и будет передавать найденное значения подконтрольным объектам, которые не видят цель, но пытаются дойти до нее методом Алгоритм A*.

В статье 19 представлена магистерская дипломная работа, цель которой являлось предсказание положения объекта в случайный момент времени при помощи линейной регрессии.

Существует множество других математических моделей, которые способны предсказывать положение объекта через како-либо время, например, марковские модели, фильтр Калмана и т. д. Однако, данные модели являются нейронными сетями, требующие обучения. Потенциально, это приведет к потере производительности алгоритма. К тому же внедрение в случайную систему пред обученных моделей намного сложнее и потенциально будет иметь меньший охват, чем алгоритм в текущей конфигурации (без использования нейронный сетей).

1.3 Постановка задачи

В данном параграфе будет сформулирована задача разработки, описана математическая и алгоритмическая модель и озвучены требования к

техническим характеристикам, а также проанализированы аналоги, представленные на рынке.

Прежде, чем переходить к вышеперечисленным планам, стоит рассмотреть существующие на рынке ПО (программное обеспечение), реализующие групповой алгоритм для преследования цели. Так как данный алгоритм в рамках данной выпускной квалификационной работы выполняется в качестве отдельного модуля к существующему ПО для разработки игр (в дальнейшем называемые «Игровые движки»), то стоит рассмотреть существующие на рынке в открытом доступе аналогичные ПО, реализующие похожую технологию.

Аналоги на рынке

Первым игровым движком к рассмотрению будет Unity Engine. Это кроссплатформенная среда разработки компьютерных игр, созданная компанией Unity Technologies. Так как задачей данной ВКР заключается в создании простого искусственного интеллекте, который будет управлять подконтрольными объектами для преследования цели, то стоит рассматривать уже готовые AI (artificial intelligence) модели в игровых движках, в Unity они называются «machine learning agents”. Полагаясь на официальную статью от компании Unity Technologies, внутри данного игрового движка заложены специальные объекты, которые способны обучаться тем задачам, которые в них закладывает разработчик. По сути, это шаблон, который нужно обучать для каких-то конкретных задач. То есть, разработчики данного игрового движка не закладывали готовый функционал алгоритма, который будет детально изучаться в данной ВКР.

Вторым для рассмотрения игровым движком будет не менее известный Unreal Engine, который так же находится в открытом доступе, созданный компанией Epic Games. Из официальной документации ясно, что игровой движок предлагает несколько решений для создания собственного искусственного интеллекта. Конкретно, внутри данного движка уже заложен

функционал для создания дерева поведения и автомата состояний, которые рассматривались в параграфе *Групповое мышление объектов*. Так же, данный игровой движок предлагает готовые решения, реализующие навигационную систему в собственной системе координат. Однако готового решения, предлагающее групповой алгоритм, управляющий подконтрольными объектами для преследования цели, в данном игровом движке нет.

Задача разработки

После анализа существующих аналогов на рынке можно сделать вывод, что явной реализации технологии, управляющей группой объектов для преследования цели нет. Задача данной ВКР разработать готовый модуль для собственного игрового движка, которая будет реализовывать данную технологию.

Основная задача – сделать так, чтобы данный модуль работал с максимально возможной эффективностью и на любом типе местности.

После анализа математических и алгоритмических методов в параграфе *1.2 Анализ математических и алгоритмических методов* были выбраны конкретные методы для решения определённых задач. По сути, разработка алгоритма, управляющего группой объектов для преследования цели, сводится к 3 задач: выбор метода преследования, выбор метода реализации группового алгоритма и решение проблемы обхода полигонов на местности.

Так, было установлено, что наилучшим методом преследования цели будет метод погони или классический метод преследования, который характеризуется своей простатой и эффективностью.

Для реализации группового алгоритма будет использоваться модель, основанная на использование конечных автоматов, так как это позволит четко контролировать состояния подконтрольных объектов и самого алгоритма, обеспечивая предсказуемость поведения алгоритма, что гарантирует точность его работы. К тому же автоматы состояний обеспечивают эффективность работы алгоритма.

Для решения проблемы обхода полигонов будут использоваться два разных алгоритма, которые будут сменять друг друга в зависимости от состояния конкретного объекта. В первом случае будет использоваться метод ищейки. Использование данного метода обуславливается его гибкостью и эффективностью. Гибкость обеспечивается за счет того, что данный метод не привязан к локальной системе координат. Он способен работать на любом типе местности с любым количеством непроходимых полигонов. Эффективность же обуславливается, что данный метод построен на автоматах состояний, которые переключается в зависимости от текущей ситуации и компьютеру не придется долго принимать решение. Так же, данный метод, в отличие от остальных, не принуждает к хранению значительного количества информации, такую как сетку местности, координаты конкретных полигонов и т. д., внутри одного игрового объекта. Таким образом, данный легко работает на любом типе местности и при ее изменении так же будет работать без дополнительных изменений внутри самого алгоритма или дополнительной информации.

Во втором случае будет использоваться алгоритм нахождения наикратчайшего пути. Данный метод не обладает конкретными преимуществами и эффективность его спора, однако он необходим в редких игровых ситуациях, так что его использование обязательно и сильно нагружать систему он не будет.

Описание поведения математической модели

Описание поведения мат модели сводится к работе конечного автомата состояний группового алгоритма и подконтрольных алгоритмов. Состояния подконтрольных объектов будут и состояния управляющего ими объекта, реализующий алгоритм группового управления, взаимозависимы. Поэтому, описания автоматов состояния будут происходить параллельно.

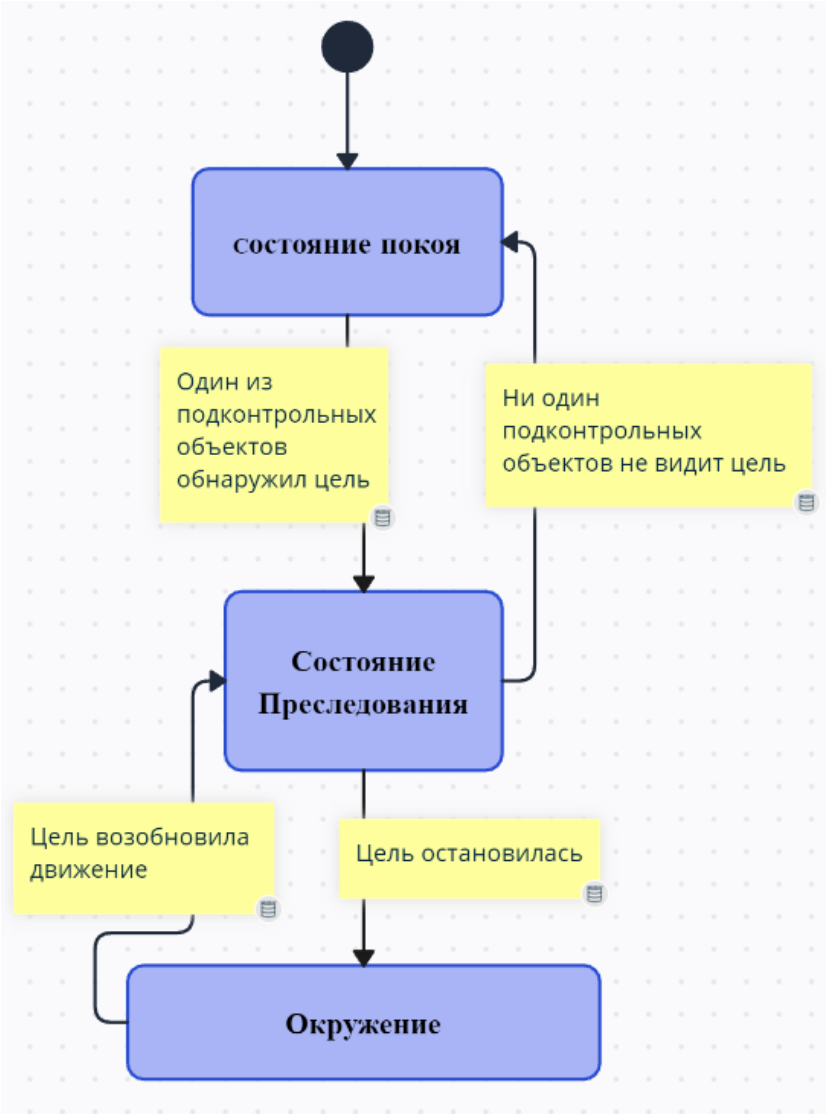


Рисунок 5. Автомат состояний группового алгоритма

Как видно из Рисунок 5, всего существует 3 возможных состояния группового алгоритма.

Первое состояние — это состояние покоя. С начала работы программы алгоритм будет находиться именно в этом состоянии. Оно характеризуется тем, что все подконтрольные объекты не выполняют никаких задач, а сам алгоритм не распределяет никаких задач между подконтрольными объектами.

Второе состояние – это состояние преследования. В него групповой алгоритм может перейти, только если один из подконтрольных объектов «замечает» цель и переходит в состояние преследования, используя метод ищейки, то есть двигаясь по «следам» цели, как показано на Рисунок 6.

Данное состояние характеризуется тем, что групповой алгоритм оповещает все подконтрольные ему объекты о смене состояния покоя на состояние преследование и начинает передавать координаты преследуемой цели тем объектам, которые не двигаются по следам, оставляемые целью.

Групповой алгоритм выходит из этого состояние в случаях, когда:

- Ни один из подконтрольных объектов не видит преследуемую цель
- Цель остановилась

В первом случае групповой алгоритм переходит в состояние покоя и оповещает подконтрольных объектов об этом. Во втором случае алгоритм переходит в состояние окружения. В этом состоянии у группового алгоритма задача окружить преследуемый объект. Алгоритм старается расставить подконтрольные объекты равномерно вокруг цели.

Если преследуемая цель возобновляет движение, то групповой алгоритм обратно переход в состояние преследования и оповещает об этом подконтрольные объекты.

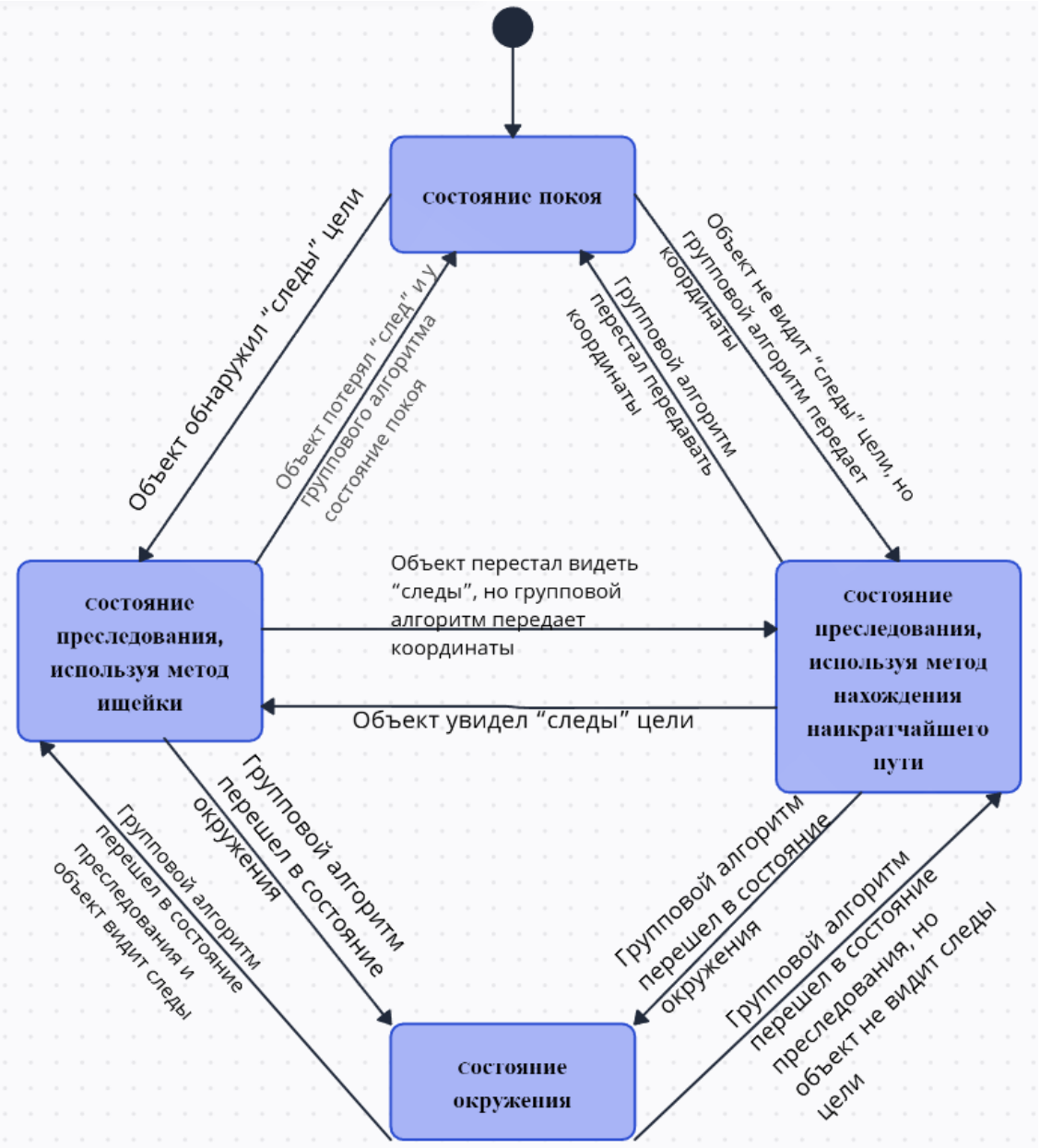


Рисунок 6. Автомат состояний управляемого объекта

На Рисунок 6 продемонстрирован автомат состояний объектов-преследователей или подконтрольного объекта. Он аналогичен автомату состояний группового алгоритма, однако состояние преследования разделена на два разных состояния.

Первым состоянием является состояние покоя. В него входят все объекты-преследователи на этапе инициализации.

Если один из подконтрольных объектов замечает цель, которую надо преследовать, он переходит в состояние преследования и оповещает об объект,

который производит контроль над данным, передавая ему координаты цели. Данный объект оповещает все остальные подконтрольные ему объекты, переводя их в состояние преследования, отсылая им координаты цели. Таким образом, часть объектов-преследователей находится в состоянии преследования, используя метод ищейки, то есть двигаясь по следам, оставляемые целью, а другая часть старается наикратчайшем путем дойти до координат, которые им передает групповой алгоритм. То есть, задача главного алгоритма перевести все ему подконтрольные объекты в состояние преследования по следам.

Подконтрольные объекты и групповой алгоритм находятся в состоянии преследования, пока хотя бы один из подконтрольных объектов двигается по следам цели. Как только все подконтрольные объекты переходят в состояние преследования по координатам, групповой алгоритм переходит в состояние покоя, переводя все подконтрольные объекты в состояние покоя.

Если преследуемый объект останавливается, то групповой алгоритм переводит все подконтрольные ему объекты в состояние окружения, стараясь «окружить» цель. Если преследуемый объект возобновляет движение, то групповой алгоритм переходит в состояние преследования, переводя подконтрольные объекты в одно из двух состояний преследования:

- Если преследуемый объект видит следы, то он переходит в состояние преследования, используя метод ищейки
- Если преследуемый объект не видит следы, то он переходит в состояние преследования, стараясь сократить расстояние с преследуемой целью наикратчайшим путем

Таким образом, групповой алгоритм и подконтрольные ему объекты работают в синергии, переключая друг друга состояния в зависимости от игровой ситуации, стараясь наиболее эффективно догнать преследуемую цель.

Глава II

Вторая глава данной выпускной квалификационной работы посвящена описанию решений, принятых в первой главе и разработке программного средства. В данной главе будет предоставлен и обоснован выбор программных средств для реализации задуманного программного обеспечения, определена цель проектирования рациональной внутри машинной технологии обработки на основе выбранных инструментальных средств, определены функции управляющей программы, определены характеристики, состав, структура входных данных, необходимых для решения задачи, установлены взаимосвязи входных, промежуточных и выходных данных и разработан план тестирования и отладки разработанного программного обеспечения.

2.1 Выбор средств для разработки

Язык программирования

В настоящий момент существует множество языков программирования, подходящих для разработки игр. В основном, опираясь на 13 статью, существует 4 основные платформы для разработки игр: компьютер, консоль, мобильные устройства, веб-приложения. Выбор языка программирования зависит от множества факторов. Первый фактор, который стоит принять во внимание - выбор платформы, под которую будет разрабатываться ПО, реализующее данный алгоритм.

Опираясь на статистику, приведённую в статье 14, наиболее популярной платформой для игр в 2022 стал именно компьютер. Поэтому, ПО, реализующее алгоритм, будет разрабатываться под персональный компьютер.

Теперь стоит рассмотреть популярность языков программирования под компьютерную разработки игр. Исходя из информации, приведенной в статье 15, наиболее популярными языка программирования для разработки компьютерных игр являются: C#, C++ и Java. Однако, из этой тройки язык программирования Java подходит для поставленной задачи наименее всего. Во-первых, код, написанный на Java, не сможет интегрироваться в другие

платформы, в отличие от C++ и C#. Во-вторых, для C++ и C# созданы современные игровые движки, такие как Unreal Engine (для C++) и Unity Engine (для C#). Так как алгоритм, который является целью разработки данной выпускной квалификационной работы, должен быть дополнением к разрабатываемому игровому движку, который будет работать на основе одного из существующих коммерческих игровых движков, то Java, не имеющая за собой ни одного доступного игрового движка, очевидно, хуже подходит для поставленной цели, чем другие два языка программирования в рассмотренном выше списке.

Таким образом, выбор языка программирования сводится к выбору между двумя языка: C# и C++. Изучив определённое количество статей, которые ставят перед собой задачу сравнить эти два языка программирования, можно сделать вывод, что очевидных преимуществ одного языка над другим нет. Сравнение идет по следующим показателям: кроссплатформенность, производительность кода и требовательность к ресурсам, библиотеки, стоимость поддержки, риски и перспективы.

Опираясь на статью 16, с точки зрения кроссплатформенности C++ в более выгодной позиции, чем C#. Автор утверждает, что C# был спроектирован, как кроссплатформенный язык, однако им не оказался, несмотря на существование неофициальных .NET окружений под разными платформами и потенциальную бинарную совместимость между платформами. При этом для разработки на C++ сложилась практически равноценная инфраструктура на большинстве существующих платформ, есть масса библиотек, которые скомпилированы или могут быть скомпилированы под любые существующие платформы.

Производительность кода и требовательность к ресурсам. Возможности по оптимизации *unmanaged* кода куда шире, чем возможности по оптимизации managed кода. Таким образом, пиковая производительность кода достижима только в *unmanaged* исполнении, т. е. в пределе, почти любая задача на C++ может быть решена с меньшими требованиями к ресурсам. Поэтому в тяжелых

задачах, связанных с обработкой большого количества данных, C++ имеет сильные преимущества перед C#. Однако, опираясь на статью 17, целью которой является сравнить C# и C++ по производительности, автор отмечает, что в новых версиях .NET есть возможность код-генерации в runtime. То есть, если pipeline обработки изображений заранее неизвестен (например, задаётся пользователем), то в C++ придётся собирать его из кирпичиков и, возможно, даже использовать виртуальные функции, тогда как в C# можно добиться большей производительности, просто сгенерировав метод. Поэтому, очевидного преимущества C++ над C# с точки зрении оптимизации нет.

Библиотеки. Отличие ассортимента C++ и C# библиотек в том, что C++ библиотек больше, они имеют большую историю, за которую стали неплохо отлажены и оптимизированы, часто кроссплатформенные, многие с открытым кодом. Однако при всех положительных сторонах C++ библиотеки как имеют очень разную, часто даже архаичную архитектуру, часто не объектный, а структурно-процедурный интерфейс. Связано это с тем, что многое C++ библиотеки — это C библиотеки. Другая неприятная особенность C++ библиотек — это создание и переопределение своих базовых типов. Многие C++ библиотеки заводят свои типы строк, контейнеров, переопределяют некоторые базовые типы. Этому есть логичные объяснения (лучшая производительность, поддержка кроссплатформенности, отсутствие подходящих типов на момент написание библиотеки), однако все это не добавляет удобства использования и красоты коду. Базовые же C++ библиотеки дают не так много, как дают стандартные библиотеки C#, поэтому подбор правильных библиотек для проекта C++ — это задача, необходимая даже в сравнительно простых проектах.

В C# перечисленных выше проблем значительно меньше. Огромное количество библиотек с .NET идет в базе, плюс к ним множество свободно доступных библиотек, это покрывает практически все первостепенные задачи разработки под Windows. Наличие большого количества стандартных типов почти избавляет от библиотек, где базовые типы переопределены. И в силу

того, что библиотеки C# сравнительно молодые, - интерфейсы библиотек, как правило, лучше вписываются в те или иные шаблоны проектирования, что часто упрощает их изучение.

Таким образом, C# имеет очевидные преимущества над C++ с точки зрение библиотек.

Стоимость поддержки. В поддержке приложений большой разницы между C++ и C# нет. Хотя стоит понимать, что некоторые ошибки в приложениях, написанных на C#, средствами .NET исправить невозможно и при необходимости их исправить стоимость поддержки может существенно возрасти. Однако если говорить о рефакторинге, то зачастую приложения, написанные на C#, исправлять несколько дешевле.

Перспективы развития. По оценке нескольких специалистов, C# развивается компанией Microsoft намного быстрее, чем C++. Для сравнения стоит рассмотреть несколько статистических данных, приведенных в 18 статье.

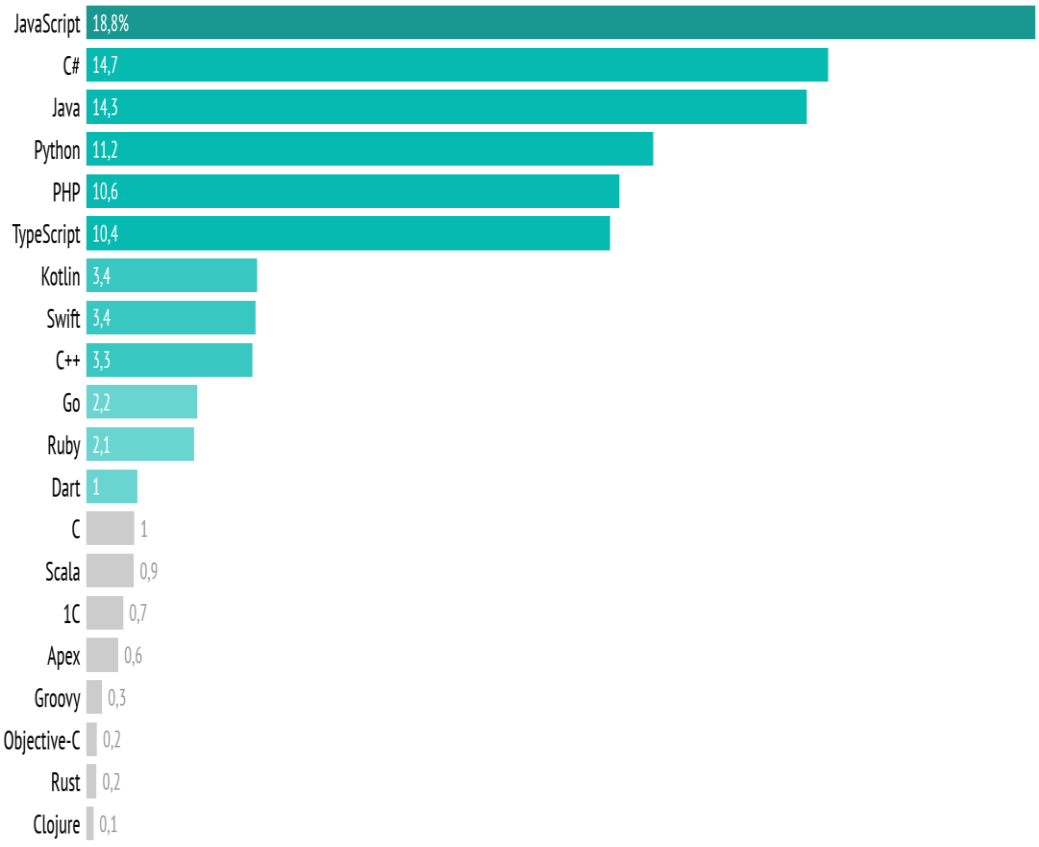


Рисунок 7. Популярность языков программирования сейчас

График, продемонстрированный на Рисунок 7 выше, отображает популярность языков программирования на 2022 год. Очевидно, что C# на несколько позиций популярней, чем C++.

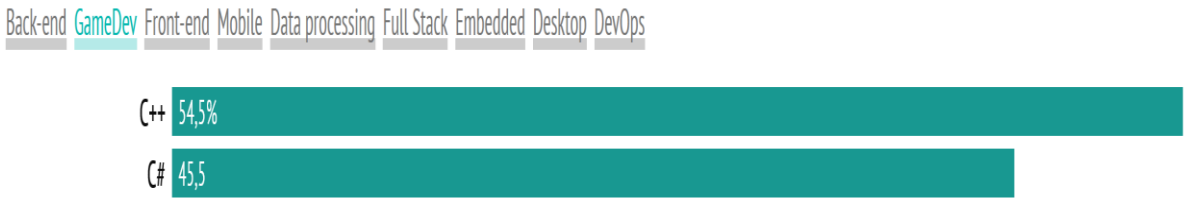


Рисунок 8. Популярность языков в области разработки игр

На Рисунок 8 представлен график, демонстрирующий популярность языков в отдельной области. В данном случае – в области разработки игр. Как видно из графика, C++ немного популярней, чем C#, однако разница не велика.

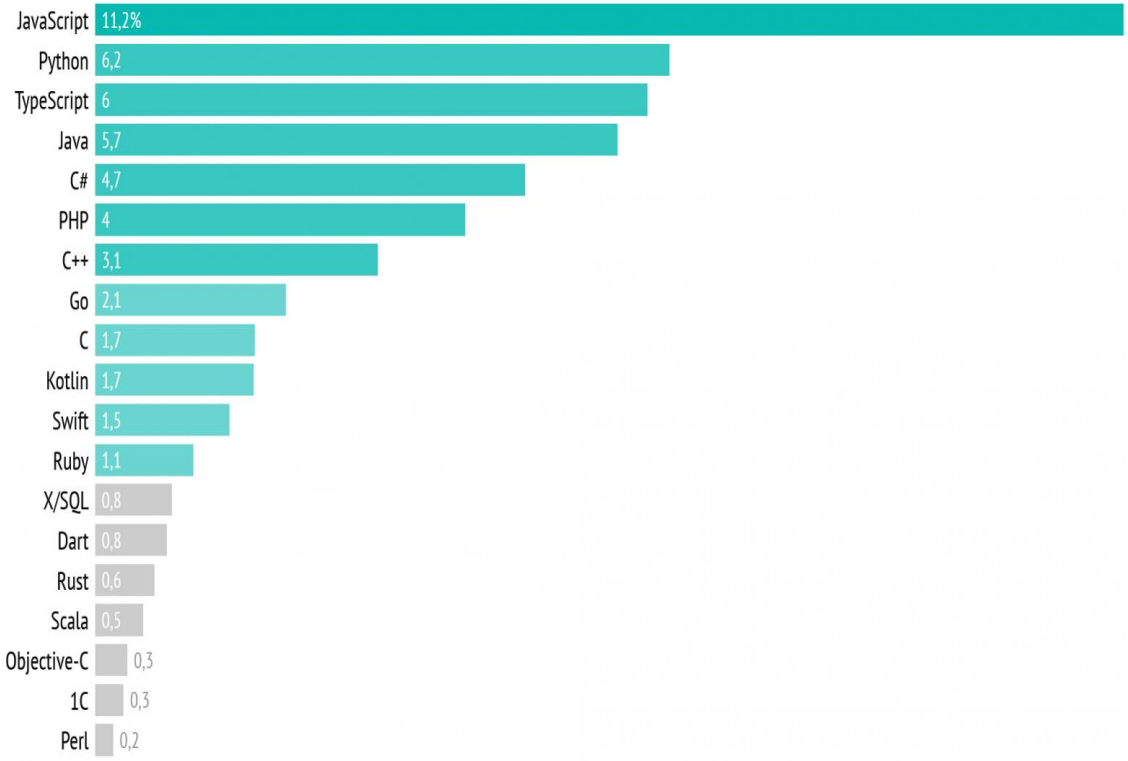


Рисунок 9. Популярность зыков программирования в open source проектах.

Как видно из графика на Рисунок 9 выше, в open source проектах C# обгоняет C++ на 2 позиции и по показателям почти в 1.5 раза.

Вывод. Проанализировав ряд статей и литературы, был произведен сравнительный анализ двух потенциальных языков программирования – C# и C++. На основе этих данных можно построить таблицу – характеристику,

отображающую преимущества и недостатки языков программирования по тем или иным качествам.

Таблица 2. Преимущества и недостатки C# и C++

Я. П.	Сравнительные характеристики				
	Кроссплатформенность	Производительность	Наличие библиотек	Низкая Стоимость поддержки	Перспективы развития
C#	✗	✓	✓	✓	✓
C++	✓	✓	✗	✗	✗

Исходя из Таблица 2 выше, C# является более перспективным выбором под поставленные задачи, чем C++. Таким образом, дальнейшая разработка программного обеспечения будет осуществляться с помощью объектно-ориентированного CLR языка C#.

Аппаратные средства

Программное обеспечение не будет требовать аппаратных средств для запуска, его можно будет запустить с любой машины при помощи любой среды программирования, игрового движка, поддерживающего язык C# или командной строки.

Дальнейшая визуализация работы алгоритма будет осуществляться на ПО для создания игр Unity, а код будет демонстрироваться в IDE Rider.

2.2 Алгоритмы решения задач

В данном параграфе будут проводиться словесно-формульные или графические описания ключевых алгоритмов разрабатываемого программного обеспечения, теоретически разобранные в главе *1.2 Анализ математических и алгоритмических методов*. Сюда войдут:

- Описание работы алгоритмов обхода полигонов, учитывая выбранные методы преследования (*Метод ищейки* и *Алгоритм A**).

- Описание работы автоматов состояния как для группового алгоритма, так и для подконтрольных ему объектов (*Групповое мышление объектов*)
- Описание работы предиктивной системы оценивания положения объекта в момент времени Т.

Описание алгоритмов обхода полигонов

Как было сказано выше, каждый из алгоритмов обхода полигонов реализует принцип работы выбранного метода преследования – Метод погони. Данный метод характеризуется тем, что вектор ПЕ*Vпе в любой момент времени направлен на цель, то есть его курсовой угол α равен нулю. То есть, каждый из алгоритмов обхода полигонов будет стремиться к преследованию цели напрямую.

В главе *Методы решения проблемы обхода полигонов* был сделан вывод, что будут использоваться два основных метода обхода полигонов - Метод ищейки и Алгоритм А* в силу того, что они обладают лучшими характеристиками из рассмотренных.

Описание метода ищейки

Данный алгоритм основывается на том, что он не запоминает расположение полигонов на местности, а старается двигаться так же, как и преследуемый объект, то есть, по «следам», от сюда следует название данного метода.

Реализация метода ищейки сводится к разработке составляющих компонент:

- Реализация следов. То есть, преследуемый объект должен оставлять на своем пути «видимые» следы по мере того, как он движется.
- Реализация области видимости для объекта, реализующего метод ищейки.

Следы могут быть любой формы, цвета, степени прозрачности. Следы могут появляться с разной периодичностью и иметь разный срок жизни. Для наглядности, в данной работе следы будут квадратной формы и иметь серый цвет со степенью прозрачности 1. Периодичность появления следов – раз в 0.1 секунду и время жизни – 1.5 секунды.

Область видимости является область, в которой объекты способны видеть цель или след. Область видимости так же может иметь разные характеристики, такие как: форма обзора, радиус или длина обзора, угол обзора. В данной работе область видимости объектов будет круглой или шарообразной формы, радиус обзора – 10 метров, а угол обзора будет 360 градусов.

Реализация следов

В Листинг 1 представлен скрипт, отвечающий за создание следов. Данную функцию выполняет метод TraceLeaving(), который срабатывает 10 раз в секунду и оставляет след на месте движения объекта.

В Листинг 2 представлен скрипт, отвечающий за время жизни каждого из следа. При появлении следа данный скрипт запускает таймер LifeTimer(), который срабатывает через заданное время, которое хранится в переменной lifeTime.

Реализация области видимости

В Листинг 3 ниже представлен метод FOV_Target() в скрипте FieldOfView.cs. Данный метод реализует области видимости объектов-преследователей. Каждый кадр метод проверяет на наличие цели в области видимости заданного радиуса, который хранится в переменной radius. Если цель в поле зрения, то срабатывает центральное условие функции и булевой переменной canSeeTarget присваивается значение true.

Если одно из условий в ветке if/else не срабатывает, то значение переменной canSeeTraces меняется на false. Если цель находится в поле зрения, то объект преследователь движется к цели путем вызова метода Moving(), как показано в Листинг 4.

Стоит отметить, что движение объекта контролируется не в классе FieldOfView, чтобы не нарушать Single Responsibility Principle из группы принципов программирования SOLID, о которых говорится в статье 20.

Подробней о процессе вызова метода Moving() будет описано в следующих параграфах.

Помимо отслеживания цели в пределах видимости объекта-преследователя, класс FieldOfView.cs так же контролирует попадание следов в поле зрения, путем вызова метода TracesInView() каждый кадр, как показано в Листинг 5 ниже.

Данный метод работает точно так же, как и метод FOV_Target(), который был описан выше. Однако, задача данного метода не только определить нахождения объекта в области видимости, но и вычислить ближайший из найденных. Это выполняется путем вызова метода CalculateTheNearestTrace(), который представлен в Листинг 6.

Так же, если след находится в области видимости, а цель – нет, то объект-преследователь движется к следу, который находится в области видимости и является ближайшим к цели. Движение происходит путем вызова метода Moving(), как показано в Листинг 7 ниже.

Данный метод не является таким же, как и метод Moving() при движении к цели, так как эти методы находятся в разных класса.

Процесс вызова данного метода так же не контролируется классом FieldOfView. Подробнее об этом будет описано ниже.

Описание алгоритма A. Оптимизация алгоритма*

В данном параграфе будет разобрана структура работы алгоритма A* (A star). Поиск A* - алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины к другой. Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость». Для его реализации требуется несколько

скриптов, которые являются отдельными классами, выполняющие свои функции и задачи.

Начальной подготовкой для реализации алгоритма является создание сетки узлов, по котором будет находиться минимальный путь. Данную задачу выполняет два класса: Node и Grid.

Класс Node имеет несколько полей: isWalkable – булевая переменная, отвечающая является ли данный узел (или клетка) проходимой или нет, _heapIndex – численная переменная, отвечающая за номер клетки, Parent – переменная типа Node, которая указывает на родительскую клетку в древовидной структуре.

Класс Grid при инициализации программы вызывает метод CreateGrid(), представленный в Листинг 8 ниже. Данный метод разбивает пространство на узлы (или клетки) итерационно проверяя, является ли поле проходимым. На непроходимость поля указывает особый layerMask, который указывается во входных параметрах класса. Каждая клетка является объектом класса Node, который создается внутри цикла. Данный узлы хранятся в двумерном массиве _grid типа Node.

После завершения работы класса Grid алгоритм знает какие клетки являются проходимыми, а какие – нет. На основе этого в дальнейшим алгоритм сможет искать наикратчайший путь.

Следующим по важности методом в алгоритме является метод нахождения наикратчайшего пути FindPath, который является методом класса PathFindingEngine. Данный метод представлен в Листинг 9 в главе Приложения. Задача метода FindPath – находить кратчайший путь. В качестве входных данных метод принимает параметры startPosition и targetPosition типа Vector3, характеризующие координаты точки отсчета и точки, к которой надо прийти.

В цикле while() метод проверяет стоимость каждой клетки и стоимость пройденного пути, подбирая наикратчайший путь. Конечный путь формируется в переменной closeSet типа HashSet. Начальный путь

формируется в переменной openSet типа Heap. Данный тип является написанным бинарным деревом с методом поиска и сортировки в глубину, чтобы оптимизировать поиск и тратить на него минимальное количество времени.

В стандартной конфигурации в алгоритме не используются особые типы данных. В основном, для openSet используют тип данных List. В данной конфигурации на момент проверки окружающих узлов приходится делать сложные сравнительные проверки стоимости каждой клетки.

Использование бинарного дерева позволяет избежать этого. Каждая вершина бинарного дерева представляет собой отдельный узел, где масса вершины – сумма расстояния от текущего узла к следующему и расстояние до конечного узла. Примерная структура дерева представлена на Рисунок 10 ниже.

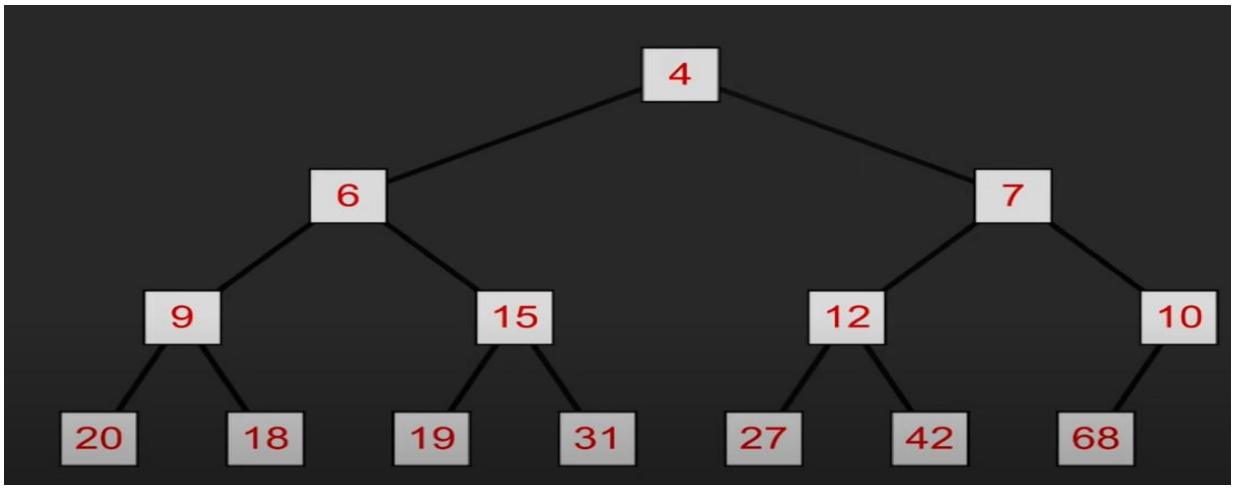


Рисунок 10.Бинарное дерево узлов

Данное дерево построено по правилу, что каждый родительский узел должен быть меньше каждого из его дочерних. Превосходство данной структуры является то, что если нужно добавить новый узел в структуру, как показано на Рисунок 11 ниже, то нужно лишь проверить стоимость добавленного и родительского узла. Если стоимость ниже, то элементы меняются местами. Данная проверка проверятся до тех пор, пока добавленный узел не будет иметь вес больше, чем родительский.

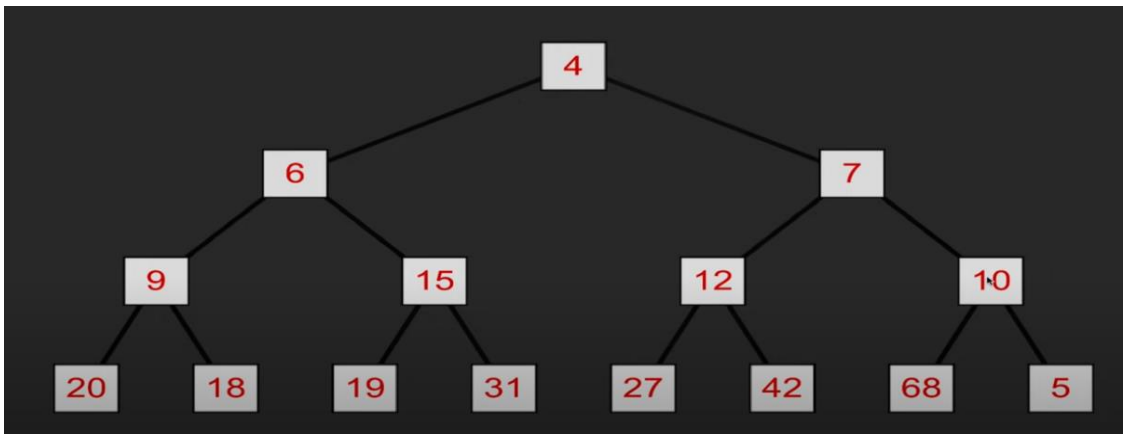


Рисунок 11.Добавление нового узла стоимостью «5» в дерево

Таким образом, если добавить новый элемент весом «5» в структуру, как показано на Рисунок 11 выше, то в конце работы сортировки дерева элемент будет находиться на позиции, как показано на Рисунок 12 ниже.

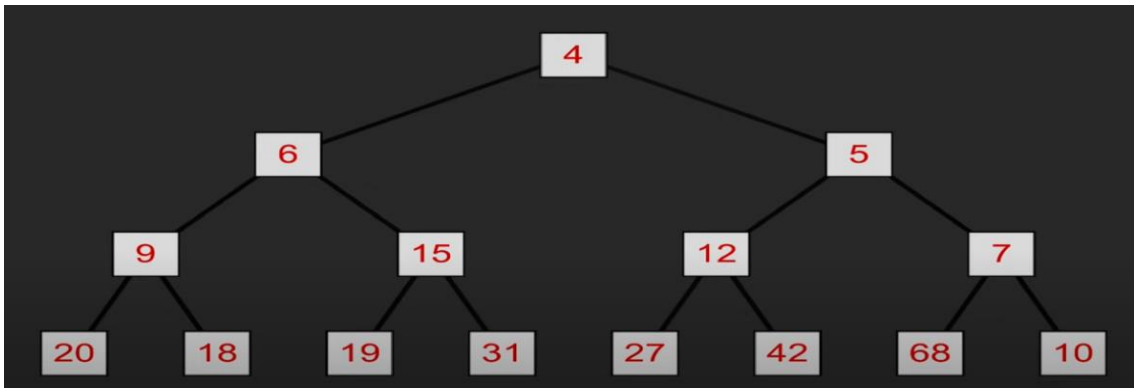


Рисунок 12.Сортировка дерева при добавлении элемента

В итоге, при формировании дерева в процессе нахождения пути, будет строиться бинарное дерево соседних узлов к текущему. И вместо того, чтобы проверять стоимость каждого и сравнивать с текущим, достаточно только достать вершину построенного бинарного дерева, потому что по правилу его построения, узел с наименьшей стоимостью будет всегда на вершине данного бинарного дерева.

Метод, который забирает вершину дерева, находится в классе Heap, который характеризует данное бинарное дерево. Название метода – RemoveFirst (Листинг 10).

Превосходство данного метода над стандартным будет продемонстрировано в дальнейшим.

Следующей важной компонентой данного алгоритма является метод RequestPath в классе PathRequestManager, который возвращает найденный путь (Листинг 11) и вызывает переданный ему метод callback.

В переменную callback передается метод FollowPath (Листинг 12), который следует по найденному пути.

Таким образом, в совокупности разобранные методы реализуют общий алгоритм A*. Данный метод позволяет эффективно искать наикратчайший путь на графе до конечной точки, а оптимизационный модуль, который реализуется благодаря классу Heap, позволяет это делать еще более эффективно, чем алгоритм A* в стандартной конфигурации.

Описание конечных автоматов состояний

В параграфе *Групповое мышление объектов* были разобраны основные алгоритмы, позволяющие реализовать искусственный интеллект. В итоге, была выбрана модель, при которой объекты принимают решения на основе текущего состояния, то есть, задача сводится к написанию конечного автомата состояния.

В статье 21 описан метод разработки автомата состояния в объектно-ориентированной парадигме. По этому принципу была разработана архитектура разрабатываемой машины состояний.

Автомат состояний подконтрольных объектов

У подконтрольных объектов существует 4 состояния:

- Состояние покоя
- Состояние преследования цели
- Состояния преследования следов цели
- Состояния нахождения пути до назначенной точки

Каждое из состояний наследуется от абстрактного класса BotBaseState, представленного в Листинг 13 в главе Приложения. Данный класс обязывает классы наследников реализовывать 3 основных метода: EnterState – метод

входа в состояние, UpdateAction – метод выполнения действия конкретного состояния, UpdateState – метод проверки на переход в следующее состояние.

Метод UpdateAction заставляет двигаться объекты. Методы движения были разобраны в параграфах Описание метода ищейки и Описание алгоритма A*. Оптимизация алгоритма.

Класс BotStateMachine, представленный в Листинг 14 в главе Приложения, является реализацией автомата состояний. Метод UpdateStateWithDelay выполняет метод UpdateState текущего состояния, метод UpdateActionWithDelay выполняет метод UpdateAction текущего состояния, а метод SwitchState выполняет задачу смена текущего состояния.

Автомат состояний группового алгоритма

Реализация автомата состояний группового алгоритма не отличается от реализации автомата состояний подконтрольных объектов.

У группового алгоритма существует два состояний:

- Состояние покоя
- Состояние преследования

Задача состояния покоя заключается в переводе подконтрольных объектов в состояние покоя. Задача состояния преследования заключается в переводе подконтрольных объектов в состояние преследования и передаче им объекта преследования. Так же, пока групповой алгоритм находится в состоянии преследования, запускается предиктивный алгоритм оценки положения объекта через некоторое время, о котором будет идти речь в следующем параграфе.

Описание алгоритма, оценивающего положение объекта

В параграфе *Групповое мышление объектов* было принято решение, что в обязанности группового алгоритма войдет просчет положения объекта через некоторое количество времени.

Данную задачу будет решать класс PredictiveMovementMachine. Данный класс прогнозирует положение объекта через некоторое количество времени. Эта задача решается при помощи линейной регрессии.

Для построение линейной регрессии данный класс нуждается в сборе данных для прогнозирования. Частицами данных являются координаты следов, которые оставляет преследуемый объект. Реализация следов была разобрана в параграфе Описание метода ищейки. Объект-преследователь, находясь в состоянии преследования цели напрямую или по следам (реализация состояний была разобрана в *Описание конечных автоматов состояний*), записывает найденные следы в список и передает их групповому алгоритму.

Групповой алгоритм, в свою очередь, при получении данных отправляет их в метод GetPrediction класса PredictiveMovementMachine. Данный метод вызывает метод LinearRegression (Листинг 15) этого же класса, который строит прямую, являющуюся линейной регрессией. На основе этой прямой можно выбрать точку времени, которая укажет на примерное положение объекта в данный момент времени и вернет переменную типа Vector3, обозначающую координаты объекта.

Таким образом, благодаря данному модулю, групповой алгоритм способен просчитывать примерное положение объекта в пространстве через некоторое время.

2.3 Состав и структура ПО

В параграфе *2.2 Алгоритмы решения задач* были разобраны основные алгоритмы, которые позволяют решить поставленную задачу. Данные алгоритмы можно структурировать по отдельным модулям:

- Модуль группового алгоритма
- Модуль подконтрольных объектов
- Модуль нахождения наикратчайшего пути

Для реализации каждого из модулей используется несколько классов, которые решают отдельные задачи.

В основе реализации модуля группового алгоритма лежат два класса – GroupManagingScript, представленный на Рисунок 17, и GroupAlgorithm_StateMachine, представленные на Рисунок 18.

Задача класса GroupManagingScript состоит в общении с пользователем через конфигурацию начальных данных, которые будут разобраны в параграфе *2.5 Инструкция пользователю*, а также выполнение функции посредника при общении с подконтрольными объектами во время выполнения программы. Ключевыми методами данного класса при общении с подконтрольными объектами являются AnswerForPathfinding, который задает состояние поиска цели для подконтрольных объектов, и метод AnswerForPositionToMove, который фиксирует точку преследования для подконтрольных объектов.

Задача класса GroupAlgorithm_StateMachine заключается в правильной обработке данных в зависимости от внешних условий. Данный класс реализует искусственный интеллект для группового алгоритма. Стоит отметить, что так же в обязанности данного класса входит генерация состояний через абстрактный класс BaseState, представленный на Рисунок 18.

В основе реализации подконтрольных объектов лежат два класса – FieldOfViewScript, представленный на Рисунок 19 и Рисунок 20, и BotStateMachine, представленный на Рисунок 21.

Класс FieldOfView выполняет задачу обнаружения цели или следов и оповещение группового алгоритма через события, представленные на Рисунок 20. Так же, внутри класса FieldOfView есть булевые переменные, которые выполняют задачу триггера для смены состояния внутри класса BotStateMachine.

Класс BotStateMachine, в свою очередь, следит за состоянием булевых переменных внутри класса FieldOfView и в нужный момент переключает

состояние для правильной обработки движения в зависимости от внешних условий.

Модуль нахождения наикратчайшего пути работает как с модулем, отвечающий за работу группового алгоритма, так и с модулем, который отвечает за работу подконтрольных объектов. Данный модуль состоит из пяти классов: Grid, Node, Heap, PathfindingEngine, PathRequestManager. UML диаграмма данного класса представлена на Рисунок 22.

Классы Grid и Node выполняют на этапе инициализации программы и вызываются из класса GroupManagingScript. Задача класса Grid – генерация графа, вершины которого являются объектами класса Node. При инициации строится связный граф, для которого обсчитываются стоимости каждой вершины для дальнейшей корректной работы поиска оптимального пути.

Во время работы программы подконтрольные объекты, находясь в состоянии поиска пути, запрашивают поиск пути через метод RequestPath у класса RequestPathManager. Данный класс переадресует запрос в класс PathfindingEngine, вызывая метод FindPath. Внутри данного метода формируется маршрут и класс Heap выполняет оптимизирующую роль при обработке данных. В итоге метод FindPath формирует оптимальный маршрут, а метод RequestPath декомпозирует его по точкам и возвращает подконтрольному объекту для дальнейшего движения.

Таким образом, разобранные 3 модуля работают в связке друг с другом и выполняют поставленные задачи.

2.4 Тестирование и отладка

Для тестирования разработанного ПО стоит проверить отдельные компоненты в системе, которые наименее надежные и могут потенциально возвращать ошибку. К таким системам относят: автоматы состояний как группового алгоритма, так и подконтрольных ему, вычисление положение объекта при помощи линейной регрессии. Так же, будет рассмотрена производительность алгоритма поиска наикратчайшего пути на графе без

использования сортировки бинарного дерева и с ее помощью. Проверка автоматов состояний и работы линейной регрессии будут проводиться при помощи статического метода тестирования, то есть без запуска программы. Проверка производительности Алгоритм A* будет проводиться во время работы программы, то есть, при помощи динамического метода тестирования.

Тестирование автомата состояний

При написании ПО была разработана отладочная консоль, которая отображает текущее состояние подконтрольных объектов и состояние группового алгоритма. В данном параграфе демонстрацией работы автоматов состояний будет выступать отображение на данной отладочной консоли.

У подконтрольных объектов возможны 4 состояния: состояние покоя, состояние, когда цель в поле зрения, состояние, когда следы цели в поле зрения и состояние нахождения пути. У группового алгоритма имеется 2 состояние: состояние покоя и состояние преследования.

Подконтрольные объекты и групповой алгоритм находятся в состоянии покоя, когда ни один из подконтрольных объектов не видит ни цель, ни ее следов, как показано на Рисунок 31. В таком случае консоль должна отображать состояние idling для подконтрольных объектов и для группового алгоритма, как показано на Рисунок 32.

Если в поле зрения подконтрольного объекта попадает цель, то ее состояние меняется на Target In Vision, а состояние группового алгоритма переводится из idling в chasing, переводя другие подконтрольные объекты в состояние нахождения пути, если они тоже не видят цель. Данная ситуация отображена на Рисунок 33. Как видно из Рисунок 34, объект, в поле зрения которого попала цель, перешел в состояние target in vision, а остальные объекты пытаются найти путь, находясь в состоянии pathfinding. Групповой алгоритм находится в состоянии chasing.

Когда подконтрольный объект теряет цель из поле зрения, он пытается найти следы, которые остаются на пути цели. Если подконтрольный объект их находит, то переходит в состояние traces in vision, а состояние объектов,

которые искали цель, не меняется. Данная ситуация продемонстрирована на Рисунок 35. Как видно из Рисунок 36, подконтрольный объект движется по следам, находясь в состоянии traces in vision, а состояние других подконтрольных объектов и группового алгоритма не поменялось.

Последняя важная ситуация заключается в переходе из состояния pathfinding в состояние target in vision, когда цель добирается до цели. Данная ситуация продемонстрирована на Рисунок 37. Как видно из Рисунок 38, два данный объект входит в состояние target in vision, а другой объект по-прежнему находится в состоянии pathfinding.

Таким образом, можно сделать вывод, что автомат состояний работает стабильно.

Тестирование работы линейной регрессии

Для проверки корректности работы линейной регрессии будут написаны модульный тест, который будет проверять правильность работы метода LinearRegression. Данный тест будет передавать в метод возможные данный и сопоставлять с ожидаемым результатом. Так как линейная регрессия — это линейная функция, то никаких погрешностей быть не может и принимать их во внимание не имеет смысла.

Для написания модульного тестирования будет использоваться встроенные библиотеки C# для проведения модульного тестирования. Код класса, выполняющего роль тестового модуля, продемонстрирован в Листинг 16. Здесь в класс PredictiveMachine передаются начальные данные: набор точек, характеризующие позиции цели, массив пройденного времени и время, для которого надо рассчитать позицию.

При помощи онлайн ресурса Desmos была построена моделируемая линейная регрессия, показанная на Рисунок 39. С помощью нее можно рассчитать предполагаемые координаты объекта через N времени. Эти данные будут использоваться в качестве ожидаемых в классе модульного тестирования.

После запуска теста, можно видеть, что метод линейной регрессии прошел проверку и вернул ожидаемый результат (Рисунок 40).

*Тестирование работы алгоритма A**

Важным фактором проверки данного алгоритма является качество его оптимизации. Для этого нужно измерить время выполнения алгоритма без использования сортировки бинарного дерева и с ее использованием. Процесс и принцип оптимизации алгоритма были разобраны в параграфе *Описание алгоритма A*. Оптимизация алгоритма*.

Для того, чтобы засечь время работы алгоритма в разной конфигурации была использована базовая библиотека C# System.Diagnostics и использованы объект класс Stopwatch и методы Start() и Stop().

Так, при работе алгоритма без использования сортировки бинарного дерева алгоритм искал результат за 24–28 миллисекунды, как показано на Рисунок 41. Это кажется быстрой работой алгоритма, однако стоит учесть, что поиск пути будет происходить несколько раз в секунду и его могут запрашивать бесконечно большое количество подконтрольных объектов.

Однако, при использовании сортировки бинарного дерева, цель которой оптимизировать алгоритм, алгоритм поиск наикратчайшего пути справляется с поставленной задачей в среднем за 5 миллисекунд, что в 5–6 раз быстрее базой конфигурации алгоритма (Рисунок 42).

Таким образом, разобранный метод оптимизации алгоритма существенно сокращает время поиска наикратчайшего пути, что приводит к наилучшим временным показателям.

2.5 Инструкция пользователю

Разработанное ПО используется в качестве инструмента при разработке игр, где необходима система преследования объектов. Оно представляет собой модуль, который самостоятельно наделяет подконтрольные и преследуемый объекты необходимым функционалом на стадии инициализации программы. Использование данного ПО может быть только в Unity подобных системах,

где используется аналогичный tick rate, а в качестве языка программирования выступает объектно-ориентированный CLR язык C#.

Демонстрация работы и использования разработанного ПО будет происходить в игровом движке Unity Engine.

Для начала работы нужно создать объекты, которые будут выступать в роли объектов-преследователей, и объект-цель. Если разработка игры происходит в двухмерном пространстве, то создание объекта происходит как на Рисунок 23, если же разработка идет в трехмерном пространстве, то процесс создания объекта продемонстрирован на Рисунок 24. Фигура объектов не важна – она может быть любой. В итоге должен появиться объект, как показано на Рисунок 25. Опционально возможно создать преграды.

После создания объектов нужно настроить слои, на которых будут находиться объекты-преследователи, цель и препятствия. Для этого нужно добавить три основных слоя (на англ. Layer). В итоге, подконтрольные объекты, цель и преграды должны иметь свои уровни, как показано на Рисунок 27, Рисунок 28 и Рисунок 29. Добавить свои слои можно через инспектора слоев, как показано на Рисунок 26.

Далее нужно добавить объект, который отвечает за менеджмент подконтрольных объектов в иерархию объектов. Данный объект находится в папке prefabs. Добавлении происходит, как на Рисунок 30. Объект называется GroupManager.

Дальнейшая настройка происходит исключительно внутри объекта GroupManager. Сначала, нужно добавить созданные объекты в соответствующие поля. Объекты-преследователи в список подконтрольных объектов, а цель – в поле для цели, как показано на Рисунок 13.

Далее, нужно групповому алгоритму дать понять на каких слоях считать цель, препятствия и подконтрольные объекты. Демонстрация настройка продемонстрирована на Рисунок 15. В поле target Mask – слой цели, в поле obstacle Mask – слой препятствий, в поле bot Mask – слой подконтрольных объектов.

Теперь, в объекте GroupManager можно настроить время жизни оставляемых целью следов. Данная настройка происходит через конфигурацию поля trace lifeTime, как показано на Рисунок 14. Число указывается в секундах.

Так же, опционально можно настроить отладочную консоль, которая отображает текущее состояние подконтрольных объектов и состояние группового алгоритма. У пользователя или разработчика есть выбор включать данную консоль или нет. Данная настройка происходит через переключение булевой переменной console Active, как показано на Рисунок 16. Если значение false – консоль будет невидимая. В обратном случае консоль будет включена.

После настройки всех параметров и этапов подготовки, описанных выше, алгоритм готов к эксплуатации.

Список литературы

1. https://cyberleninka.ru/article/n/sravnenie-algoritmov-presledovaniya-obektov/viewer
2. https://programcpp.ucoz.ru/publ/4-1-0-6
3. https://intuit.ru/studies/courses/1104/251/lecture/6456
4. https://cyberleninka.ru/article/n/problema-presledovaniya-v-igrah-snizhenie-resursoemkosti-resheniya-s-pomoschyu-metoda-s-predvaritelnoy-otsenkoy-kart/viewer
5. https://qna.habr.com/q/870001
6. https://habr.com/ru/post/496878/
7. https://habr.com/ru/company/netologyru/blog/598489/
8. https://habr.com/ru/post/420219/
9. https://www.dissercat.com/content/razrabotka-i-issledovanie-algoritma-garantiruyushchego-upravleniya-traektoriei-bespilotnogo
10. https://unity.com/products/machine-learning-agents
11. https://docs.unrealengine.com/5.0/en-US/artificial-intelligence-in-unreal-engine/

12. https://habr.com/ru/articles/451642/
13. https://sky.pro/media/na-kakom-yazyke-programmirovaniya-pishut-igry/
14. https://shazoo.ru/2023/01/01/137457/stali-izvestny-samye-populiarnye-igrovye-platformy-2022-goda
15. https://mmoglobus.ru/10-luchshih-yazykov-programmirovaniya-dlya-razrabotki-igr
16. https://habr.com/ru/articles/262461/
17. https://habr.com/ru/articles/532442/
18. https://habr.com/ru/articles/651585/
19. https://medium.com/analytics-vidhya/predicting-common-movement-b225569bee91
20. https://web-creator.ru/articles/solid
21. http://www.softcraft.ru/auto/switch/statemachine/statemachine.pdf
22. https://github.com/AplyQ8/Thesis_Code
23.

Приложения

```
public class TraceMaker : MonoBehaviour
{
    public GameObject tracePrefab;
    public GameObject traceKeeper;

    public void Start()
    {
        StartCoroutine(TraceLeaving());
    }

    IEnumerator TraceLeaving()
    {
        while (true)
        {
            var trace = Instantiate(tracePrefab, transform.position,
Quaternion.identity);
            trace.transform.SetParent(traceKeeper.transform);
            yield return new WaitForSeconds(.1f);
        }
    }
}
```

Листинг 1. Класс TraceMaker

```
public class TraceScript : MonoBehaviour
{
    public float lifeTime;
    private void Awake()
    {
        StartCoroutine(LifeTimer());
    }
    IEnumerator LifeTimer()
    {
        while (true)
        {
            yield return new WaitForSeconds(lifeTime);
            Destroy(gameObject);
        }
    }
}
```

Листинг 2. Класс TraceScript

```
private void FOV_Target()
    {
        Collider2D[] rangeCheck = Physics2D.OverlapCircleAll(
            transform.position,
            radius,
            _targetMask
            );
        if (rangeCheck.Length > 0)
        {
            Transform targetTransform = rangeCheck[0].transform;
            Vector2 directionToTarget = (targetTransform.position -
transform.position).normalized;
            if (Vector2.Angle(transform.up, directionToTarget) < angle / 2)
            {
                float distanceToTarget = Vector2.Distance(transform.position,
targetTransform.position);
                if (!Physics2D.Raycast(transform.position, directionToTarget,
distanceToTarget, _obstacleMask))
                {
                    canSeePlayer = true;
                    EnemyDetection.Invoke(gameObject);
                    canSeeTraces = false;
                }
                else
                    canSeePlayer = false;
            }
            else
                canSeePlayer = false;

        }
        else if (canSeePlayer)
            canSeePlayer = false;
    }
```

Листинг 3. Метод FOV_Target

```
IEnumerator Moving()
    {
        WaitForEndOfFrame delay = new WaitForEndOfFrame();
```

```
        while (true)
        {
            transform.position = Vector3.MoveTowards(
                transform.position,
                _viewScript.target.transform.position,
                speed * Time.deltaTime);
            yield return delay;
        }
    }
```

Листинг 4. Метод Moving. Движение к цели

```
private void TracesInView()
    {
        tracesInVision.Clear();
        if (canSeePlayer) return;
        Collider2D[] arrayOfTraceColliders = Physics2D.OverlapCircleAll(
            transform.position,
            radius,
            _traceMask
        );
        foreach (var traceCollider in arrayOfTraceColliders)
        {
            Vector2 directionToTrace = (traceCollider.transform.position -
transform.position).normalized;
            if (Vector2.Angle(transform.up, directionToTrace) < angle / 2)
            {
                float distanceToTrace = Vector2.Distance(transform.position,
traceCollider.transform.position);
                if (!Physics2D.Raycast(transform.position, directionToTrace,
distanceToTrace, _obstacleMask))
                {
                    tracesInVision.Add(traceCollider);
                    RecordTrace.Invoke(traceCollider.transform.position);
                }
            }
        }
        if (tracesInVision.Count == 0)
        {
            canSeeTraces = false;
            return;
        }
        canSeeTraces = true;
        trace = CalculateTheNearestTrace(tracesInVision,
target.transform.position).gameObject;

    }
```

Листинг 5. Метод TracesInView

```
private Transform CalculateTheNearestTrace(List<Collider2D> collider2Ds, Vector3
targetPos)
    {
        Transform objectToReturn = collider2Ds[0].transform;
        float distance = float.MaxValue;
        foreach (var collider2D1 in collider2Ds)
        {
            var localDist = Vector3.Distance(collider2D1.transform.position,
targetPos);
            if (localDist < distance)
```

```
            {
                distance = localDist;
                objectToReturn = collider2D1.transform;
            }
        }

        return objectToReturn;
    }
```

Листинг 6. Метод подсчета ближайшего следа

```
IEnumerator Moving()
    {
        WaitForEndOfFrame delay = new WaitForEndOfFrame();
        while (true)
        {
            try
            {
                //Движение к текущему следу
                transform.position = Vector3.MoveTowards(
                    transform.position,
                    _viewScript.trace.transform.position,
                    speed * Time.deltaTime);
            }
            catch (Exception)
            {
                // ignored
            }
            yield return delay;
        }
    }
```

Листинг 7. Метод Moving. Движение к следу

```
void CreateGrid()
    {
        _grid = new Node[_gridSizeX, _gridSizeY];
        Vector3 worldBottomLeft = transform.position - Vector3.right *
gridWorldSize.x / 2 -
                                  Vector3.up * gridWorldSize.y / 2;
        for (int x = 0; x < _gridSizeX; x++)
        {
            for (int y = 0; y < _gridSizeY; y++)
            {
                Vector3 worldPoint = worldBottomLeft + Vector3.right * (x *
_nodeDiameter + nodeRadius) +
                                     Vector3.up * (y * _nodeDiameter + nodeRadius);
                bool isWalkable = !(Physics2D.OverlapCircle(worldPoint, nodeRadius,
obstacleMask));
                _grid[x, y] = new Node(isWalkable, worldPoint, x , y);
            }
        }
    }
```

Листинг 8. Метод CreateGrid. Создание графа

```
private IEnumerator FindPath(Vector3 startPosition, Vector3 targetPosition)
    {
```

```
        Vector3[] wayPoints = new Vector3[0];
        bool pathSuccess = false;

        Node startNode = _grid.NodeFromWorldPoint(startPosition);
        Node targetNode = _grid.NodeFromWorldPoint(targetPosition);

        if (!startNode.IsWalkable && !targetNode.IsWalkable)
            yield return null;

        Heap<Node> openSet = new Heap<Node>(_grid.MaxSize);
        HashSet<Node> closeSet = new HashSet<Node>();
        openSet.Add(startNode);
        while (openSet.Count > 0)
        {
            Node currentNode = openSet.RemoveFirst();

            closeSet.Add(currentNode);

            if (currentNode == targetNode)
            {
                pathSuccess = true;
                break;
            }

            foreach (Node neighbour in _grid.GetNeighbours(currentNode))
            {
                if(!neighbour.IsWalkable || closeSet.Contains(neighbour))
                    continue;
                int newMovementCostToNeighbour = currentNode.GCost +
GetDistance(currentNode, neighbour);
                if (newMovementCostToNeighbour < neighbour.GCost ||
!openSet.Contains(neighbour))
                {
                    neighbour.GCost = newMovementCostToNeighbour;
                    neighbour.HCost = GetDistance(neighbour, targetNode);
                    neighbour.Parent = currentNode;

                    if(!openSet.Contains(neighbour))
                        openSet.Add(neighbour);
                    else
                        openSet.UpdateItem(neighbour);
                }
            }
        }

        yield return null;
        if (pathSuccess)
        {
            wayPoints = RetracePath(startNode, targetNode);
        }
        _requestManager.FinishedProcessingPath(wayPoints, true);

    }
```

Листинг 9. Метод нахождения пути FindPath

```
public T RemoveFirst() {
            T firstItem = items[0];
            currentItemCount--;
            items[0] = items[currentItemCount];
            items[0].HeapIndex = 0;
            SortDown(items[0]);
```

```
            return firstItem;
        }
```

Листинг 10. Метод удаления вершины дерева RemoveFirst

```
public static void RequestPath(Vector3 pathStart, Vector3 pathEnd,
Action<Vector3[], bool> callback)
    {
        PathRequest newRequest = new PathRequest(pathStart, pathEnd, callback);
        _instance._pathRequestsQueue.Enqueue(newRequest);
        _instance.TryProcessNext();
    }
```

Листинг 11. Метод RequestPath

```
IEnumerator FollowPath()
    {
        try
        {
            Vector3 currentWaypoints = _path[0];
        }
        catch (Exception)
        {
            yield break;
        }
        Vector3 currentWaypoint = _path[0];

        while (true)
        {
            if (transform.position == currentWaypoint)
            {
                _targetIndex++;
                if (_targetIndex >= _path.Length)
                {
                    yield break;
                }

                currentWaypoint = _path[_targetIndex];
            }

            transform.position = Vector3.MoveTowards(
                transform.position,
                currentWaypoint,
                speed * Time.deltaTime);
            yield return null;
        }
    }
```

Листинг 12. Метод FollowPath

```
public abstract class BotBaseState: MonoBehaviour
{
    //Поле для имени состояния
    public string StateName { get; protected set; }
    //Функция, срабатывающая при захождении в состояние
    public abstract void EnterState(BotStateMachine botStateMachine);
    //Функция, характеризующая выполняемые действия состояния
    public abstract void UpdateAction();
```

```
    //Функция, проверяющая условие на выход из текущего состояния
    public abstract void UpdateState(BotStateMachine botStateMachine);

}
```

Листинг 13. Абстрактный класс BotBaseState

```
IEnumerator UpdateStateWithDelay()
    {
        WaitForSeconds delay = new WaitForSeconds(.2f);
        while (true)
        {
            currentState.UpdateState(this);
            yield return delay;
        }
    }
    //Выполнение действия состояния
    IEnumerator UpdateActionWithDelay()
    {
        //WaitForEndOfFrame delay = new WaitForEndOfFrame();
        WaitForSeconds delay = new WaitForSeconds(0.01f);

        while (true)
        {
            currentState.UpdateAction();
            yield return delay;
        }
    }
    //Переключатель состояния
    public void SwitchState(BotBaseState state)
    {
        currentState = state;
        currentState.EnterState(this);
    }
```

Листинг 14. Основные методы класса BotStateMachine

```
private void LinearRegression(
        float[] xValues,
        float[] yValues,
        out float rSquared,
        out float intercept,
        out float slope)
    {
        if (xValues.Length != yValues.Length)
        {
            throw new Exception("Input values should be with the same length.");
        }

        double sumOfX = 0;
        double sumOfY = 0;
        double sumOfXSq = 0;
        double sumOfYSq = 0;
        double sumCodeviates = 0;

        for (var i = 0; i < xValues.Length; i++)
        {
            var x = xValues[i];
            var y = yValues[i];
            sumCodeviates += x * y;
```

```
            sumOfX += x;
            sumOfY += y;
            sumOfXSq += x * x;
            sumOfYSq += y * y;
        }

        var count = xValues.Length;
        var ssX = sumOfXSq - ((sumOfX * sumOfX) / count);
        var ssY = sumOfYSq - ((sumOfY * sumOfY) / count);

        var rNumerator = (count * sumCodeviates) - (sumOfX * sumOfY);
        var rDenom = (count * sumOfXSq - (sumOfX * sumOfX)) * (count * sumOfYSq -
(sumOfY * sumOfY));
        var sCo = sumCodeviates - ((sumOfX * sumOfY) / count);

        var meanX = sumOfX / count;
        var meanY = sumOfY / count;
        var dblR = rNumerator / Math.Sqrt(rDenom);

        rSquared = (float)(dblR * dblR);
        intercept = (float)(meanY - ((sCo / ssX) * meanX));
        slope = (float)(sCo / ssX);
    }
```

Листинг 15. Метод LinearRegression. Высчитывание линейной регрессии

```
public class UnitTest1

{
      [TestMethod]
       public void TestMethod1()
       {
            PredictiveMachine machine = new PredictiveMachine();

            var expectedXPos = 12;
            var expectedYPos = 15.707f;

            List<Vector3> vector3s = new List<Vector3>(){
               new Vector3(1, 0, 0),
               new Vector3(2, 1, 0),
               new Vector3(3, 3, 0),
               new Vector3(3, 4, 0),
               new Vector3(3, 5, 0),
               new Vector3(4, 4, 0),
               new Vector3(5, 6, 0),
               new Vector3(7, 8, 0),
               new Vector3(8, 10, 0),
               new Vector3(9, 12, 0)
            };
            float[] timeArray = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
            int time = 13;
            machine.SetArrayInfo(vector3s, timeArray, time);

            machine.LinearRegression();

            float actualXPos = machine.predictedValueX;
            float actualYPos = machine.predictedValueY;

            Assert.AreEqual(expectedXPos, actualXPos);
            Assert.AreEqual(expectedYPos, actualYPos);
        }
    }
}
```

Листинг 16. Класс модульного тестирования

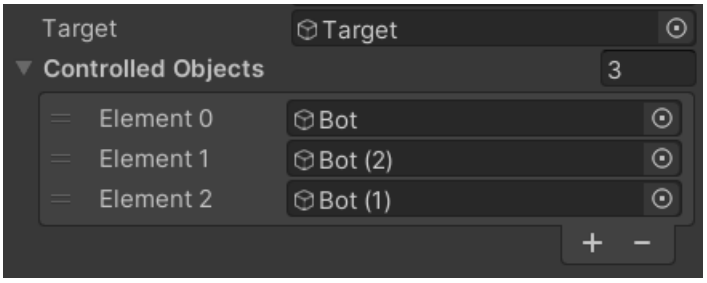


Рисунок 13.GroupManagingScript. Настройка цели и подконтрольных объектов

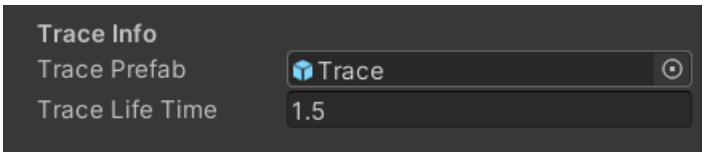


Рисунок 14.GroupManagingScript. Настройка следов

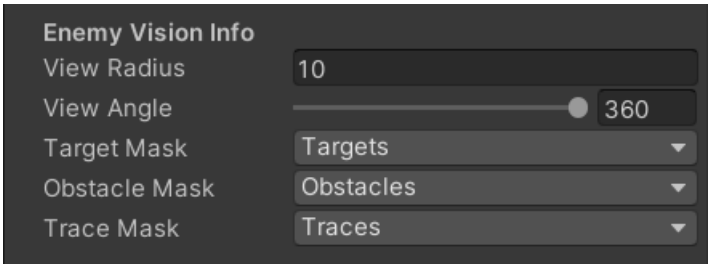


Рисунок 15.GroupManagingScript. Настройка поля зрения

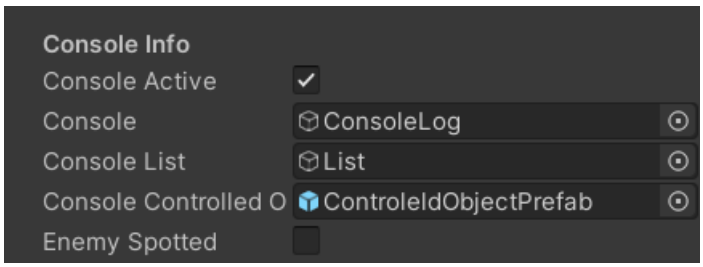
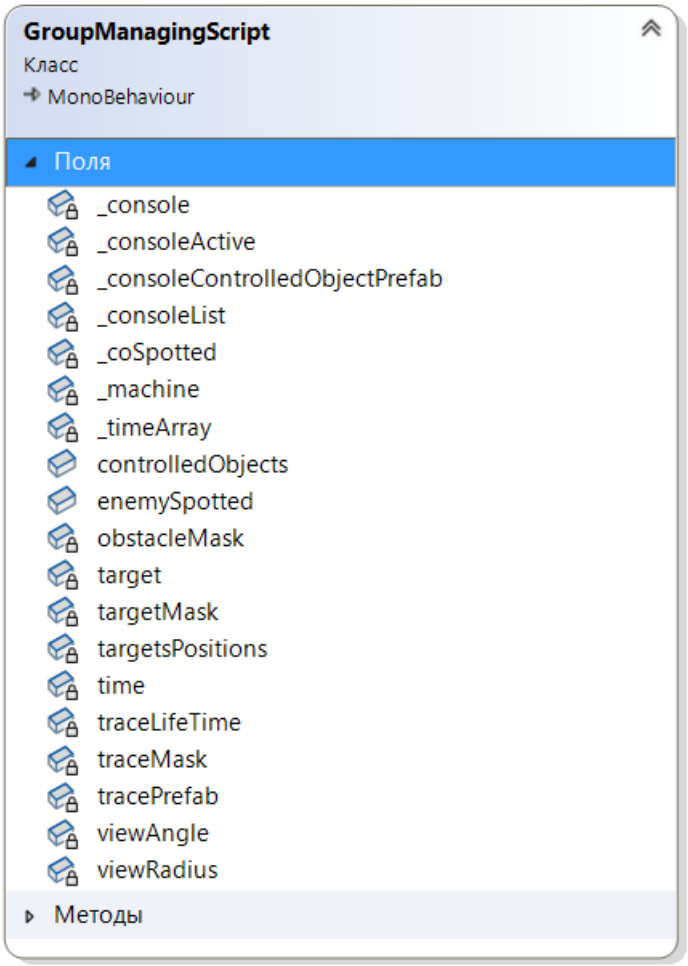


Рисунок 16.GroupManagingScript. Настройка отладочной консоли



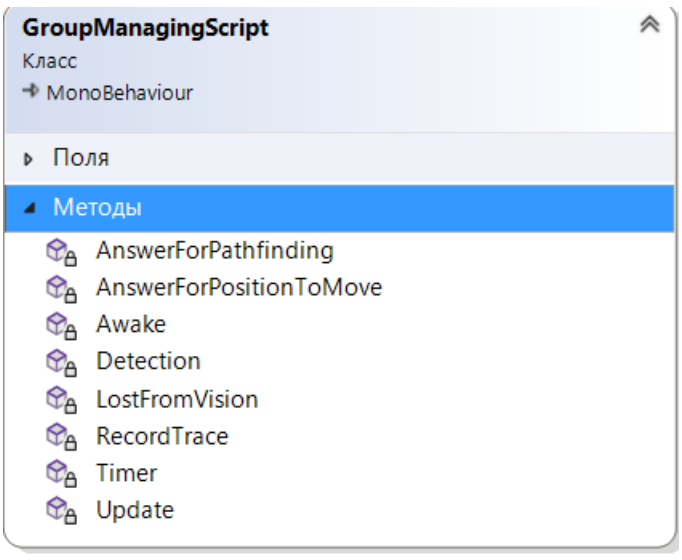


Рисунок 17.Поля и методы класса GroupManagingScript

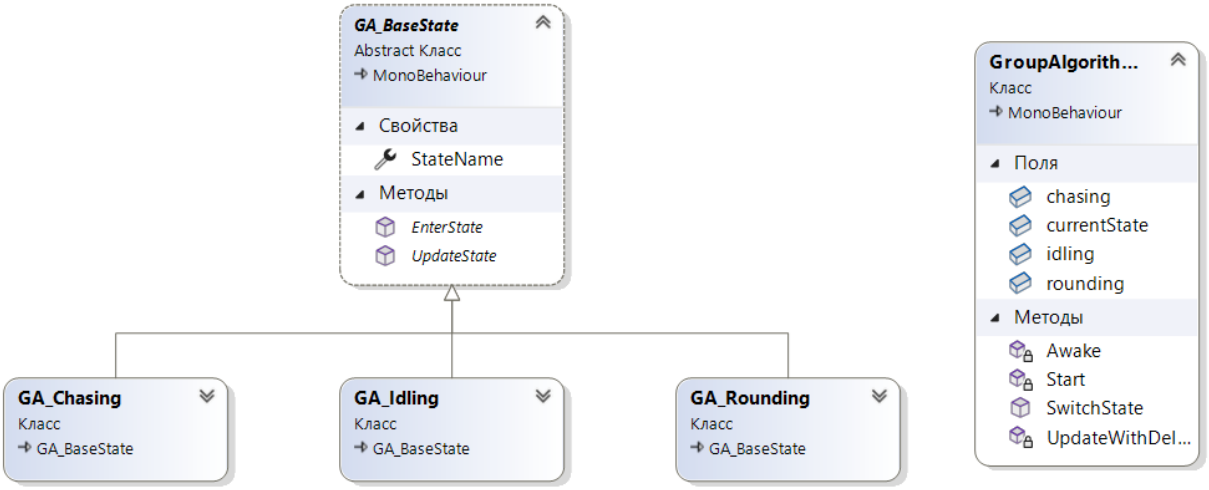


Рисунок 18.UML диаграмма класса GroupAlgorithm_StateMachine

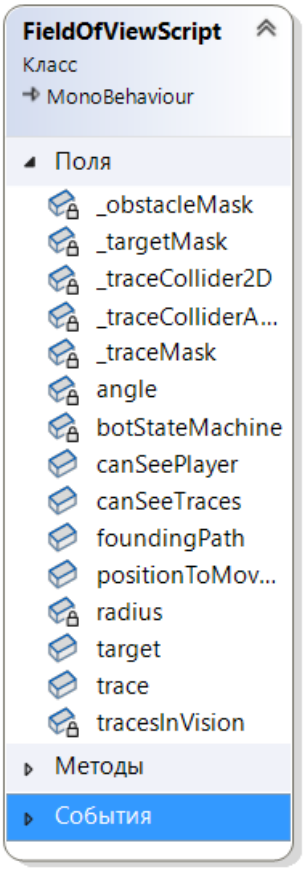


Рисунок 19.Поля класса FieldOfView

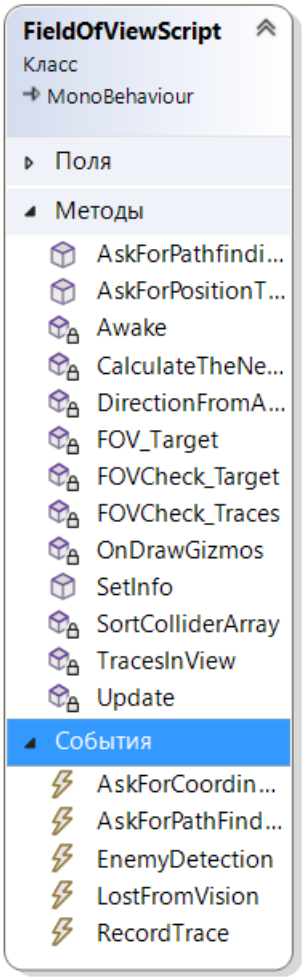


Рисунок 20.Методы и события класса FieldOfView

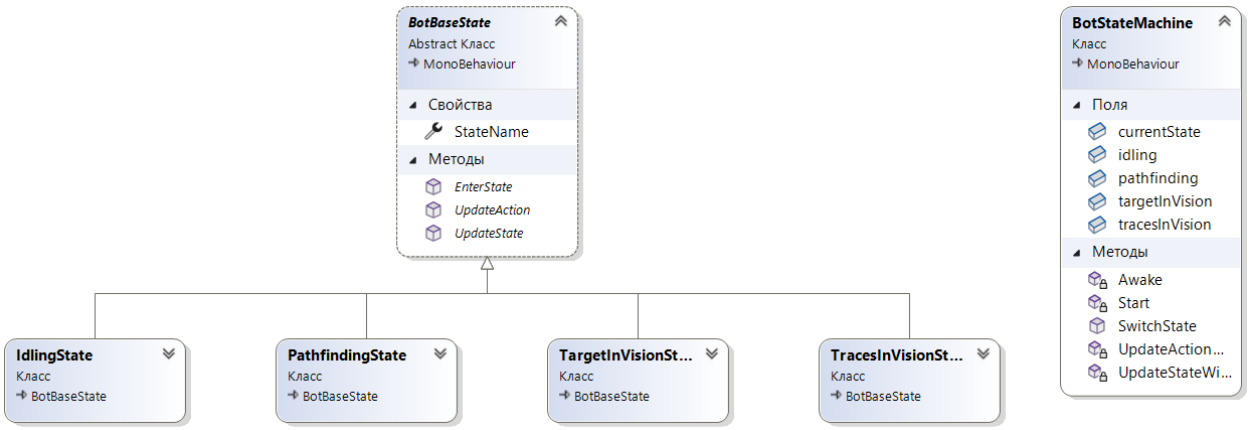


Рисунок 21.UML диаграмма класса BotStateMachine

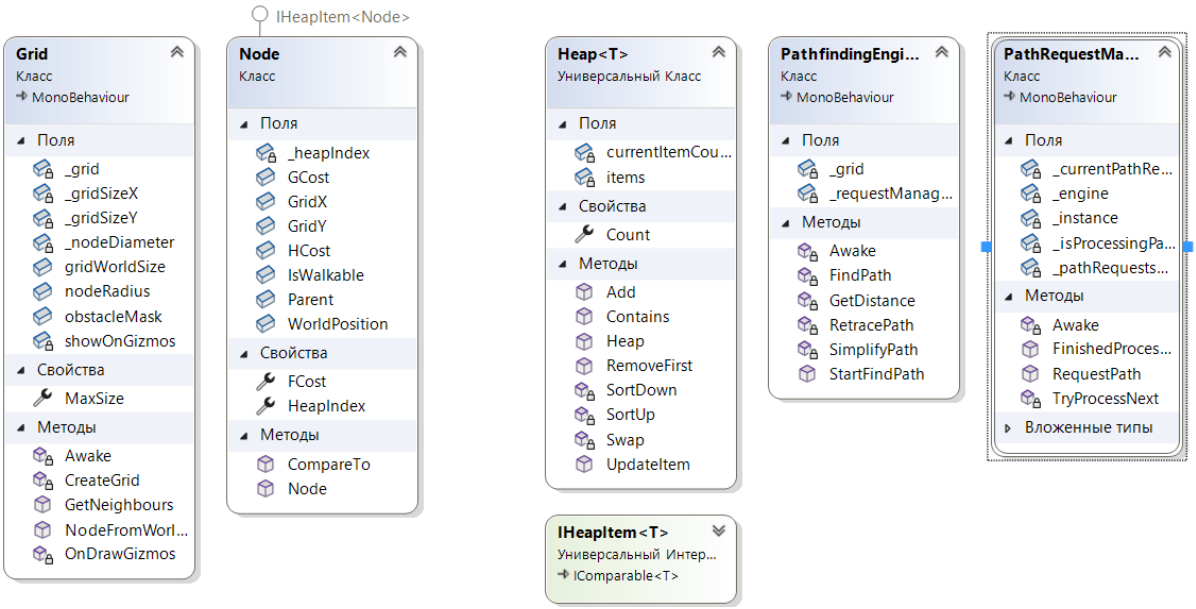


Рисунок 22.UML диаграмма модуля нахождения наикратчайшего пути

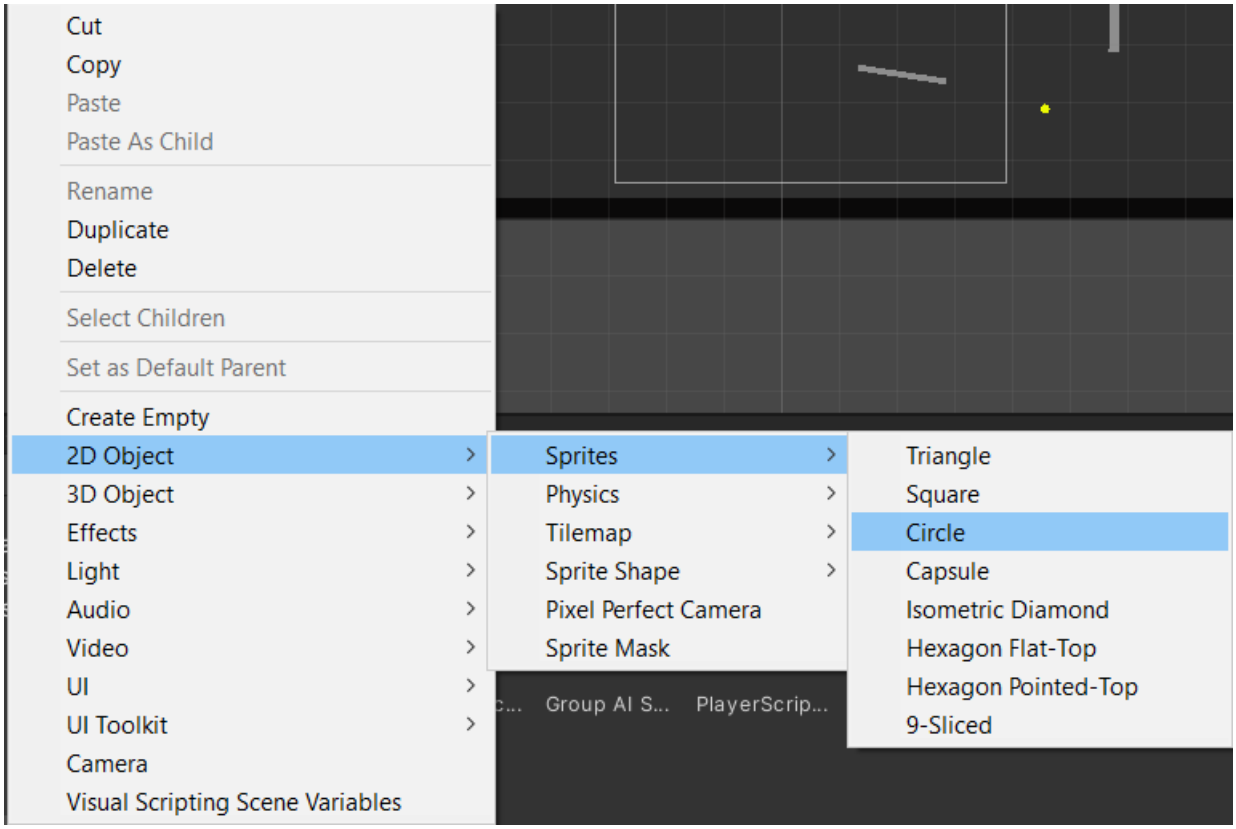


Рисунок 23. Создание объекта в двухмерном пространстве

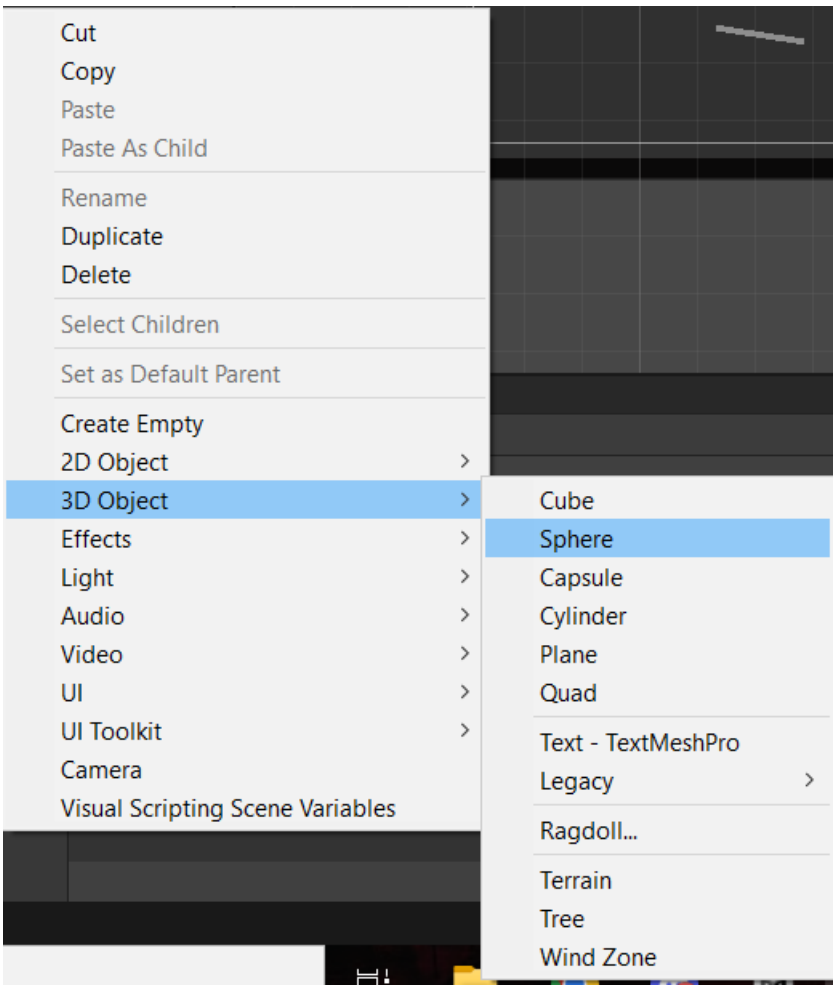


Рисунок 24. Создание объекта в трехмерном пространстве

Рисунок 25. Созданный объект

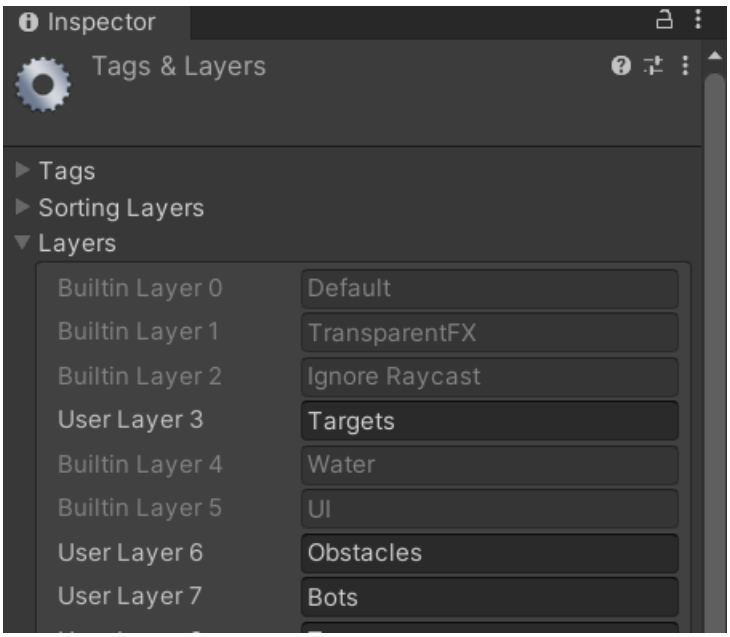


Рисунок 26. Добавление слоев

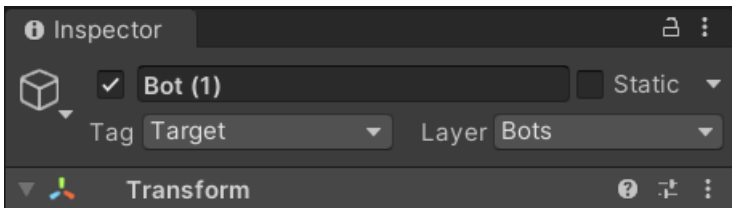


Рисунок 27. Слой подконтрольных объектов

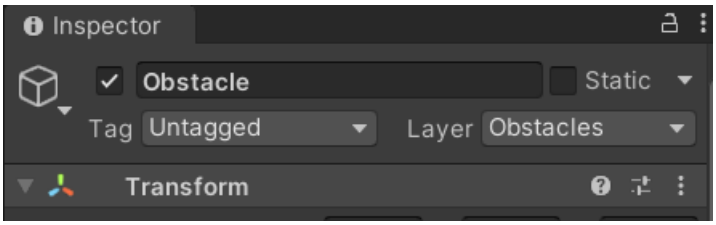


Рисунок 28. Слой преград

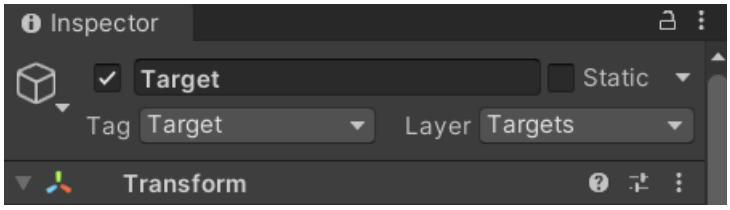


Рисунок 29. Слой цели

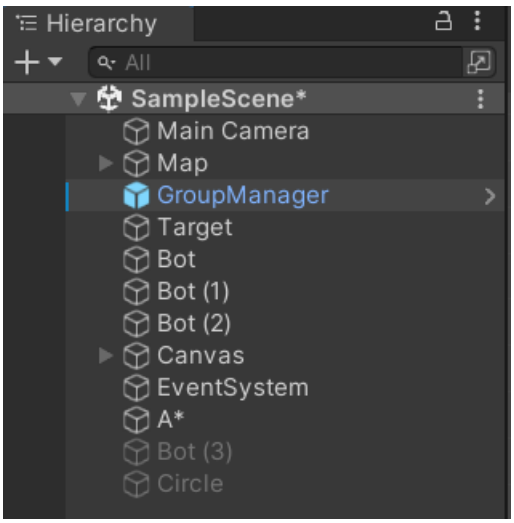


Рисунок 30. Иерархия объектов

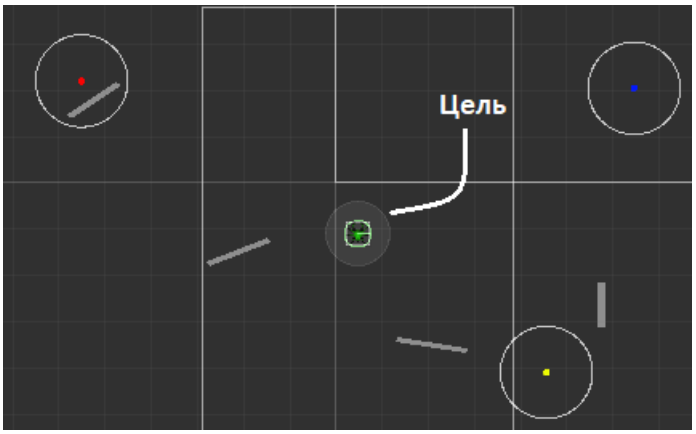


Рисунок 31. Состояние покоя на сцене

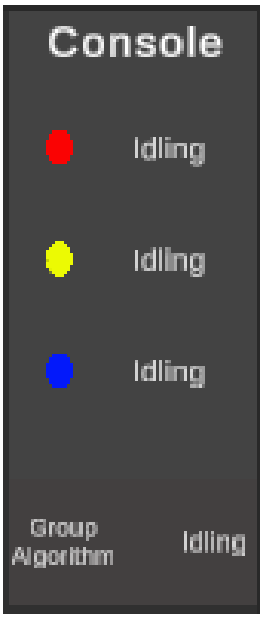


Рисунок 32. Состояние покоя на консоли

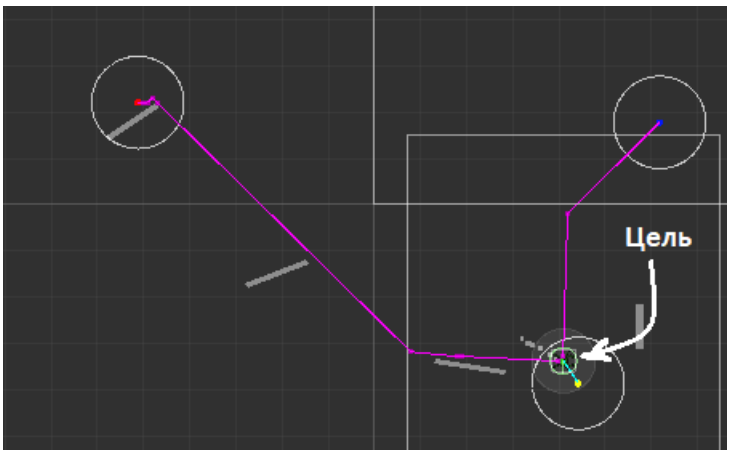


Рисунок 33. Target In Vision для одного подконтрольного объекта

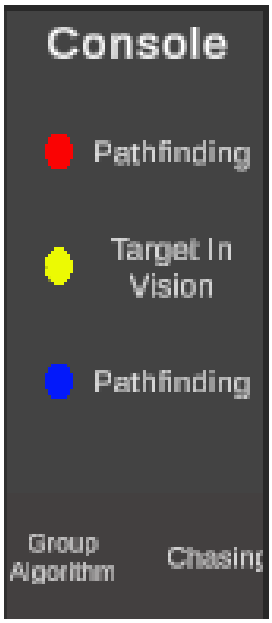


Рисунок 34. Target In Vision на консоли

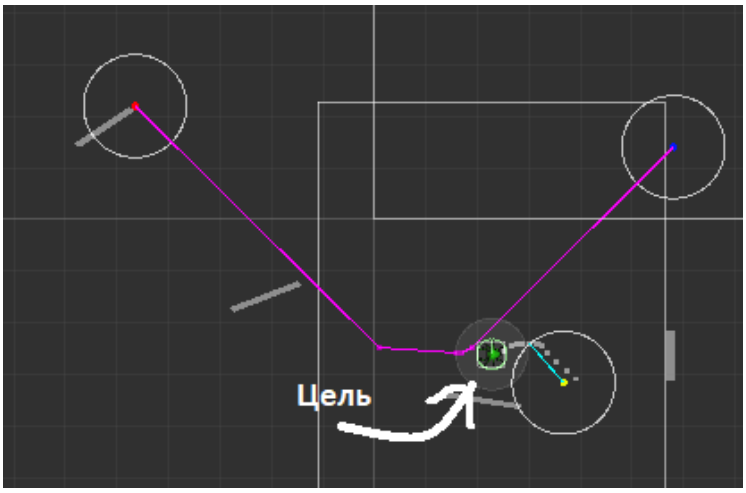


Рисунок 35. Traces in vision на сцене

Рисунок 36. Traces in vision на консоли

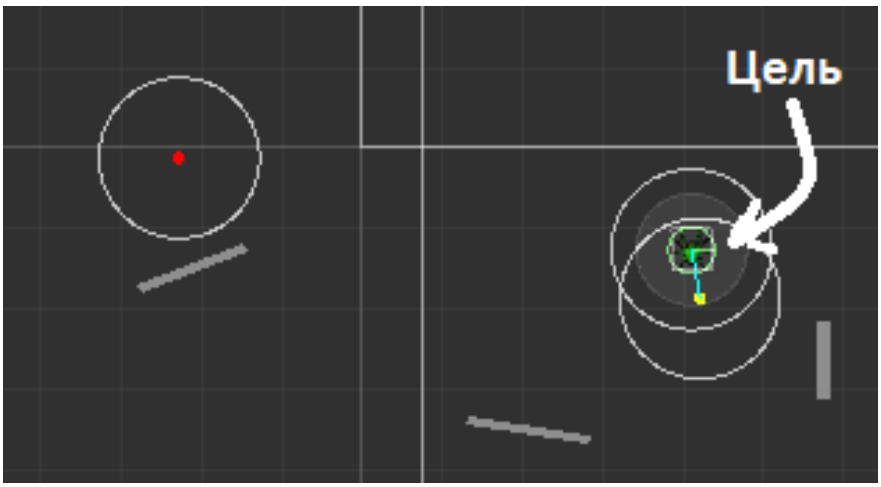


Рисунок 37. Объект нашел путь

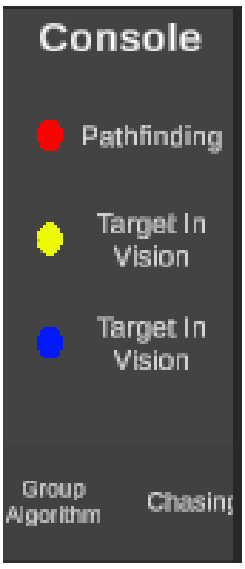


Рисунок 38. Два объекта находятся в состоянии target in vision

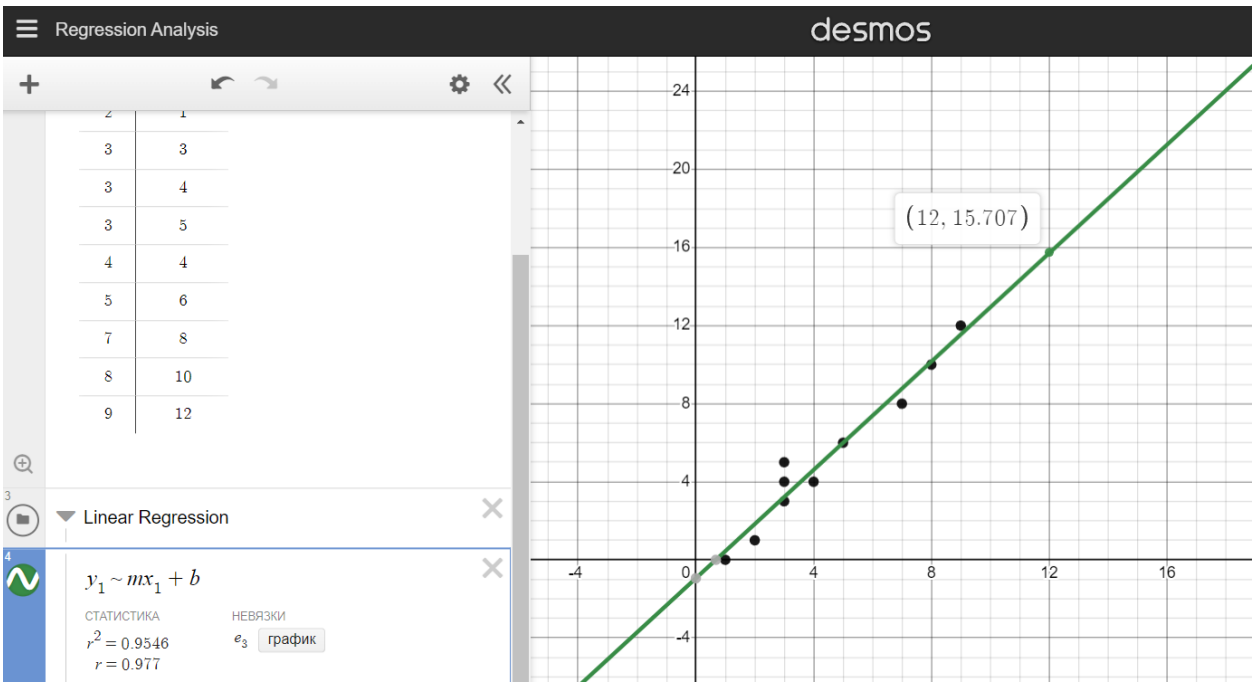


Рисунок 39. Моделируемая линейная регрессия

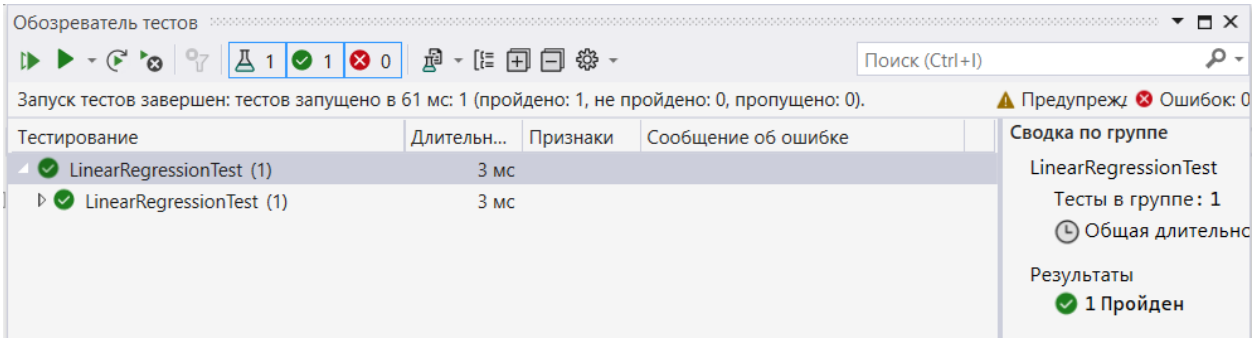


Рисунок 40. Пройденные тесты

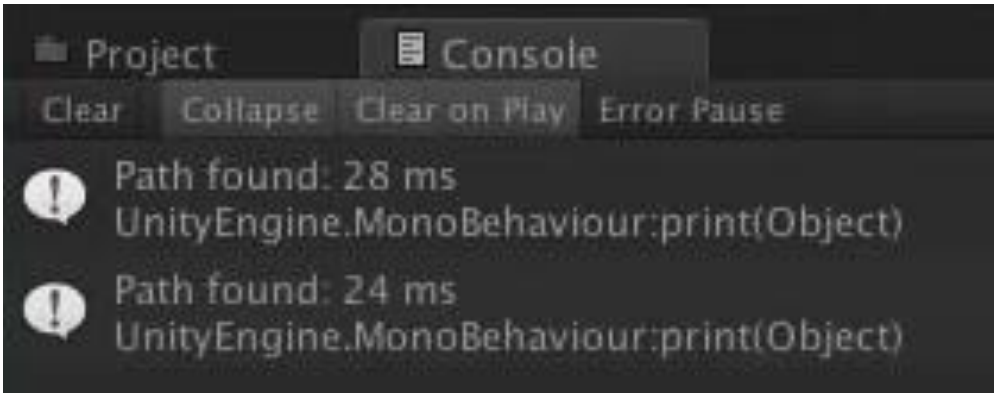


Рисунок 41.Поиск пути без использования сортировки бинарного дерева

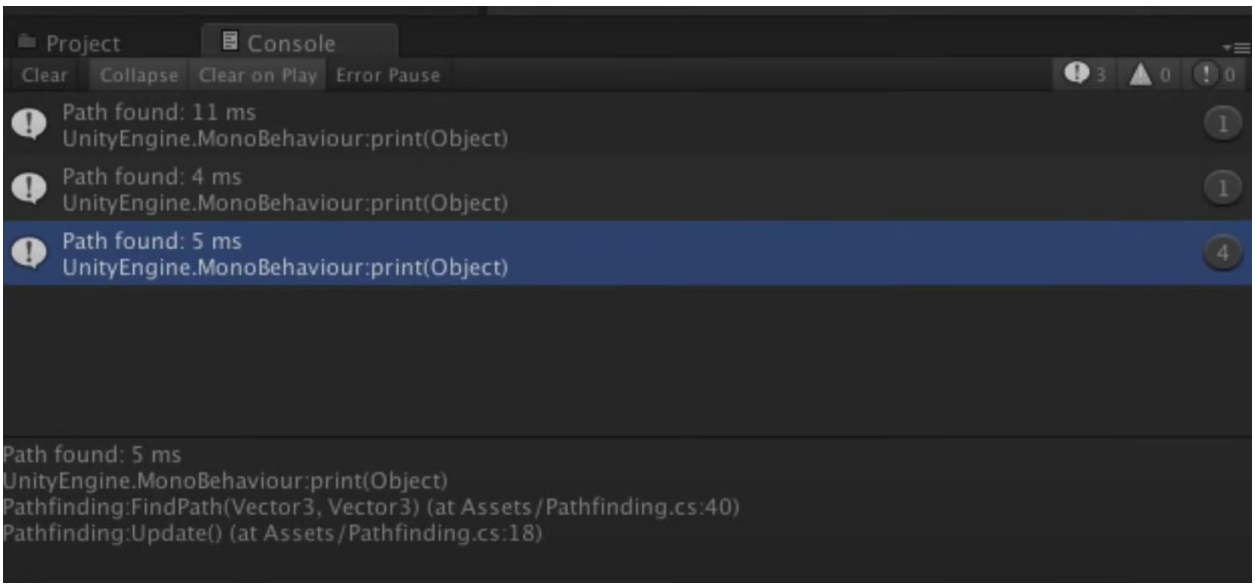


Рисунок 42. Оптимизированный поиск пути