МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра технологий программирования**

**Искусственный интеллект в компьютерных играх**

Курсовая работа

|  |  |
| --- | --- |
| Котусева Дмитрия Дмитриевича  студента 3 курса,  специальность  «Прикладная  информатика»  Научный руководитель:  кандидат физико-математических наук,  доцент В. В. Горячкин |  |

Минск - 2019

**РЕФЕРАТ**

**!!!Курсовая работа**, 21 с., 8 рис., 10 источников, 1 приложение.

**Ключевые слова:** АГЕНТ, ИИ.

**Объект исследования** — организация работы искусственного интеллекта в сфере компьютерных игр.

**Предмет исследования** — характеристические особенности методов построения эффективной модели игрового искусственного интеллекта.

**Цель работы** — исследовать различные схемы работы искусственного интеллекта в играх, адаптировать один из методов при реализации собственного приложения, написать работающий прототип.

**Задачи**:

1. аналитический обзор типов компьютерных игр, где применим искусственный интеллект, и способов организации работы искусственного интеллекта;
2. анализ и разбор математических алгоритмов, используемых при разработке искусственного интеллекта;
3. построение собственной модели искусственного интеллекта;
4. разработка работающего прототипа.

**Методы исследования** — изучение теоретических материалов, построение логических схем и моделей, проверка теоретических данных в игровом движке.

**!!!Полученные результаты:**

1. подготовлен общий аналитический обзор сфер и способов применения инерциальных датчиков;
2. построена общая математическая модель трёхосевого МЭМС-акселерометра;
3. программная реализация полученной математической модели.

**Область применения** — компьютерные игры.

**РЕФЕРАТ**

**!!!Курсовой проект**, 21 с., 8 рис., 10 источников, 1 приложение.

**Ключевые слова:** АГЕНТ, ИИ, ИГРОВОЙ ДВИЖОК.

**Объект исследования** — организация работы искусственного интеллекта в сфере компьютерных игр.

**Предмет исследования** — характеристические особенности методов построения эффективной модели игрового искусственного интеллекта.

**Цель работы** — исследовать различные схемы работы искусственного интеллекта в играх, адаптировать один из методов при реализации собственного приложения, написать работающий прототип.

**Задачи**:

1. аналитический обзор типов компьютерных игр, где применим искусственный интеллект, и способов организации работы искусственного интеллекта;
2. анализ и разбор математических алгоритмов, используемых при разработке искусственного интеллекта;
3. построение собственной модели искусственного интеллекта;
4. разработка работающего прототипа.

**Методы исследования** — изучение теоретических материалов, построение логических схем и моделей, проверка теоретических данных в игровом движке.

**!!!Полученные результаты:**

1. подготовлен общий аналитический обзор сфер и способов применения инерциальных датчиков;
2. построена общая математическая модель трёхосевого МЭМС-акселерометра;
3. программная реализация полученной математической модели.

**Область применения** — компьютерные игры.

**РЕФЕРАТ**

**!!!Курсовой проект**, 21 с., 8 рис., 10 источников, 1 приложение.

**Ключевые слова:** АГЕНТ, ИИ.

**Объект исследования** — организация работы искусственного интеллекта в сфере компьютерных игр.

**Предмет исследования** — характеристические особенности методов построения эффективной модели игрового искусственного интеллекта.

**Цель работы** — исследовать различные схемы работы искусственного интеллекта в играх, адаптировать один из методов при реализации собственного приложения, написать работающий прототип.

**Задачи**:

1. аналитический обзор типов компьютерных игр, где применим искусственный интеллект, и способов организации работы искусственного интеллекта;
2. анализ и разбор математических алгоритмов, используемых при разработке искусственного интеллекта;
3. построение собственной модели искусственного интеллекта;
4. разработка работающего прототипа.

**Методы исследования** — изучение теоретических материалов, построение логических схем и моделей, проверка теоретических данных в игровом движке.

**!!!Полученные результаты:**

1. подготовлен общий аналитический обзор сфер и способов применения инерциальных датчиков;
2. построена общая математическая модель трёхосевого МЭМС-акселерометра;
3. программная реализация полученной математической модели.

**Область применения** — компьютерные игры.

Тут должно быть содержание

**Перечень условных обозначений**

ИИ искусственный интеллект

# Введение

В эпоху стремительно развивающихся информационных технологий достижения прогресса проникают во всё большее количество сфер жизнедеятельности человека. Не исключением стала и индустрия развлечений.

Мировая индустрия развлечений аккумулирует миллиарды долларов и является значимой частью экономики большинства стран. Она включает большое количество различных направлений, одним из которых является направление компьютерных игр, получившее развитие именно благодаря укреплению в нашей жизни компьютерных технологий.

Компьютерная игра представляет собой не что иное как обычное приложение, предназначенное для развёртывания на целевой аппаратной платформе. Список этих платформ может варьироваться от обычных в классическом понимании компьютеров до встроенных в автомобили вычислительных устройств и «умных» наручных часов.

Существует огромное количество самых различных жанров компьютерных игр на любой вкус и цвет: шутеры, симуляторы, стратегии, приключения, музыкальные игры, ролевые игры и др. Стоит отметить, что чётких границ в классификации игр на данный момент не существует и зачастую чётко определить принадлежность игры тому или другому жанру крайне затруднительно.

Следует понимать, что почти никто из разработчиков игр не начинает разработку своего творения с абсолютного нуля. Все они используют ту или иную стартовую платформу ― игровой движок. Игровой движок выполняет базовые функции, необходимые в любой игре: трёхмерный рендеринг, обработка сигнала устройств ввода-вывода, обработка звуков, отрисовка элементов индикации и т. д. Также он предоставляет разработчикам необходимые программные интерфейсы для взаимодействия со встроенными возможностями движка и организации собственной логики приложения. Таким образом, конечный продукт является результатом работы как создателей движка, так и разработчиков самой игры.

Как было упомянуто выше, игровые движки берут на себя реализацию базового функционала, без которого никак не обойтись при разработке любой игры. Несмотря на это, на пути создания игры перед разработчиками встаёт немалое количество задач, которые им приходиться решать самим. Одной из таких задач является разработка искусственного интеллекта.

Стоит отметить, что в рамках игровой индустрии под искусственным интеллектом понимается не аналог интеллекта естественного (интеллекта человека), а его симуляция. Основных причин такого подхода можно выделить три:

1. дороговизна разработки полноценного искусственного интеллекта;
2. нехватка вычислительных мощностей для обеспечения должного уровня игрового опыта для игрока;
3. разработка полноценного искусственного интеллекта зачастую является избыточной в рамках поставленной задачи и обойтись можно куда меньшими силами.

То есть при разработке игрового искусственного интеллекта перед разработчиками стоит задача наиболее правдоподобной имитации поведения человека с использованием минимального количества вычислительных ресурсов. Реализация искусственного интеллекта сильно влияет на геймплей, системные требования и бюджет игры, и разработчики балансируют между этими требованиями, стараясь сделать интересный и нетребовательный к ресурсам искусственный интеллект малой ценой. Поэтому подход к игровому искусственному интеллекту серьёзно отличается от подхода к традиционному искусственному интеллекту — широко применяются разного рода упрощения, обманы и эмуляции. Например: с одной стороны, в шутерах от первого лица безошибочное движение и мгновенное прицеливание, присущее ботам, не оставляет ни единого шанса человеку, так что эти способности искусственно снижаются. С другой стороны — боты должны делать засады, действовать командой и т. д., для этого применяются «костыли» в виде контрольных точек, расставленных на уровне [источник 2 (страница википедии про ии)]. В дальнейшем при упоминании искусственного интеллекта будет пониматься именно имитация интеллекта реального, а не его аналог.

Целью данной работы является изучение и анализ существующих на данный момент методов организации работы искусственного интеллекта, типичных проблем, с которыми сталкиваются разработчики в процессе создания, а также реализация одного или нескольких из них в виде работающего прототипа.

Актуальность данной работы подкрепляется постоянно растущим спросом игроков на всё более интересные и сложные игры, создание которых требует разработки всё более сложных моделей искусственного интеллекта, а также растущей популярностью компьютерных игр, как способа досуга во всё более широких слоях населения.

# Игровой искусственный интеллект

## Обзор и классификация технологий и методов искусственного интеллекта

В любой игре, где необходимо наличие искусственного интеллекта (далее ИИ), перед этим ИИ ставится задача по управлению «интеллектуальными агентами», где агент является игровым персонажем, транспортным средством, ботом, а иногда и чем-то более абстрактным: целой группой сущностей или даже цивилизацией. Во всех случаях перед агентом стоят следующие задачи:

* Sense: получить информацию о внешней среде;
* Think: на основе полученных данных принять решение о дальнейших действиях;
* Act: собственно выполнение принятого на предыдущем шаге решения.

По завершении выше описанного цикла агент попадёт в новую ситуацию, поэтому цикл придётся проделать заново.

Существует достаточно большое количество методов (алгоритмов) организации работы ИИ.

Одним из простейших алгоритмов является дерево решений. Дерево решений состоит из узлов двух типов:

* узлы принятия решений: выбор между двумя альтернативами на основе проверки некоторого условия, где каждая альтернатива представлена в виде отдельного узла;
* конечные узлы: действие для выполнения, представляющее окончательное решение.

В качестве примера можно провести знаменитую игру Pong, в которой перед игроком и ИИ ставится одинаковая задача — отбить мяч с помощью платформы в сторону оппонента (рисунок 1.1).

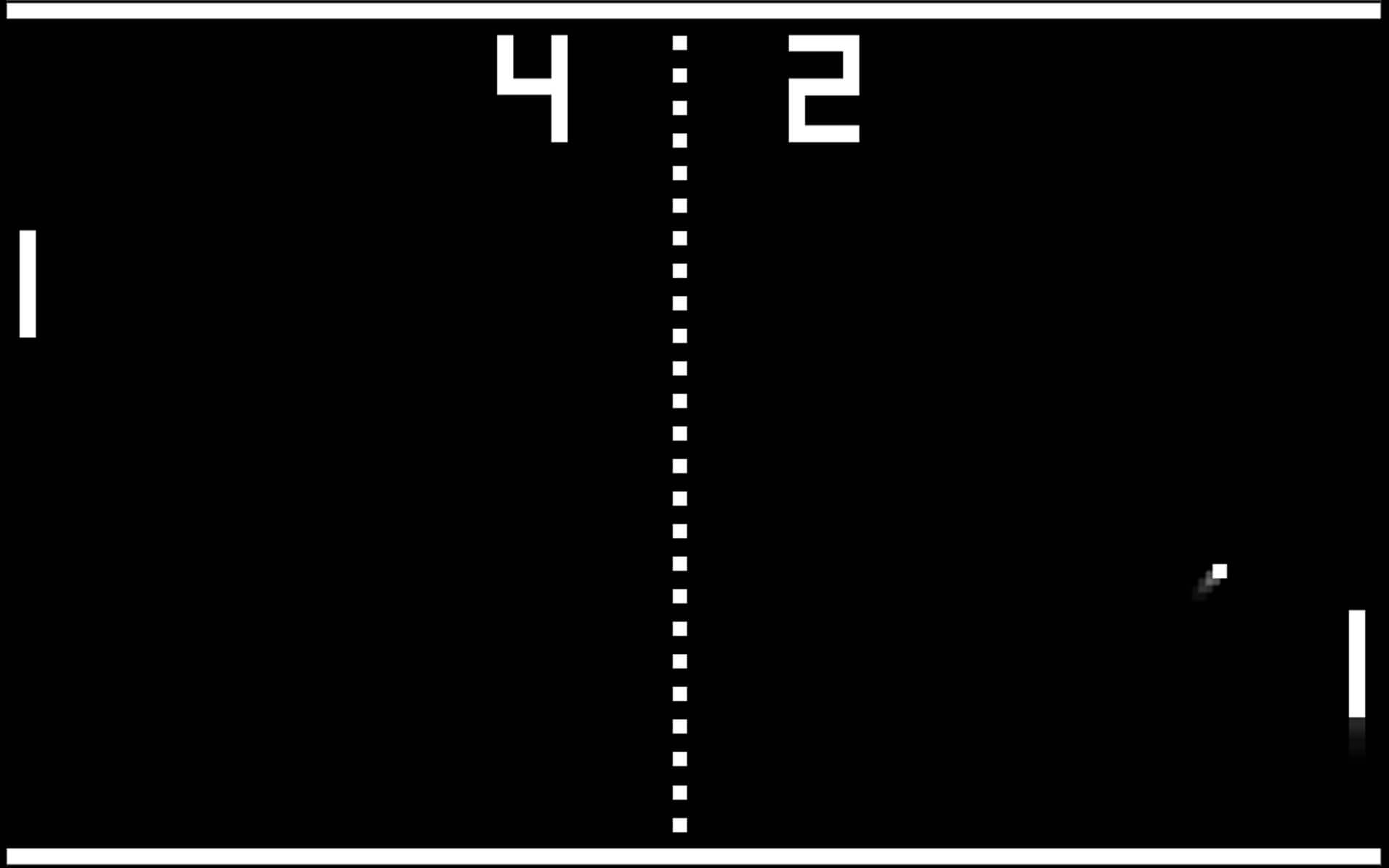


Рисунок . Интерфейс игры Pong

Более точно перед ИИ стоит задача ― решить, в каком направлении переместить платформу, чтобы отбить мяч. Блок-схема дерева решений такого ИИ может выглядеть следующим образом (рисунок 1.2):

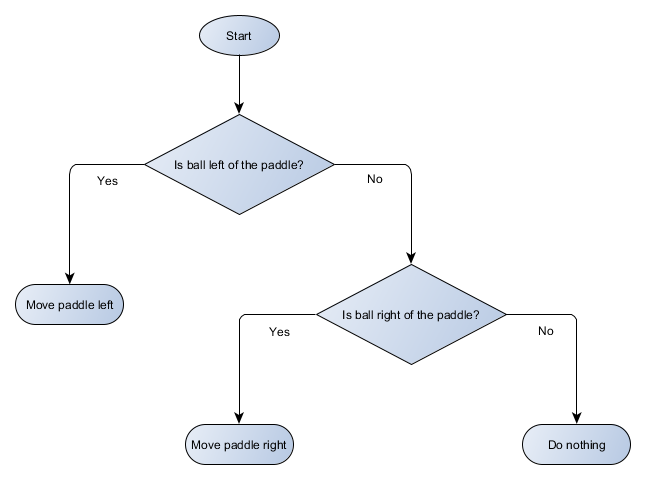


Рисунок . Дерево решений для ИИ игры Pong

В приведённой выше блоке-схеме нетрудно видеть, что получение данных о внешней среде (игра знает где мяч и где платформа, поэтому ИИ обращается к ней за этой информацией) и принятие решений происходит в узлах принятия решений, выполнение же решений происходит в конечных узлах.

Дерево решений неплохо подходит для реализаций простых ИИ, способных выполнять небольшой спектр действий. Но иногда существует слишком много условий, чтобы эффективно представить их в дереве решений или скрипте. Иногда нужно заранее оценивать, как изменится ситуация, прежде чем принимать решение о следующем шаге. Для решения этих проблем нужны более сложные подходы [ссылка на источник 3].

Ещё одним интересным подходом к организации ИИ является конечный автомат (finite state machine или FSM). Конечный автомат предполагает, что наш агент всегда находится в одном из конечного множества состояний. В качестве примера разберём агента (воина) со следующим конечным автоматом (рисунок 1.3):

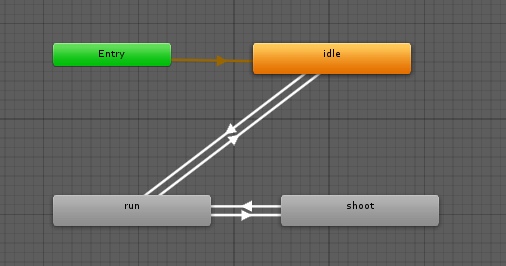


Рисунок . Пример конечного автомата



Рисунок . Игровой агент

На рисунке 1.3 нетрудно различить 3 состояния, в которых может находиться агент:

* состояние покоя (именно в нём он находится на рисунке 1.4);
* состояние бега за игроком;
* состояние стрельбы в игрока.

И снова мы можем посмотреть на эту систему через призму цикла Sense/Think/Act. Sense воплощается в данных, используемых логикой перехода. Think — переходами, доступными в каждом состоянии. А Act осуществляется действиями, совершаемыми периодически в пределах состояния или на переходах между состояниями [ссылка на источник 3]. Основной вопрос при реализации такой логики ИИ вызывает механизм проверки условий переходов между состояниями. Зачастую логику проверок условий переходов выносят за пределы автомата. Автомат, в таком случае, получает данные условий переходов через обработчики событий, генерируемых внешним кодом. Таким образом, этап Sense происходит вне автомата во внешнем коде.

Конечно же, описанный выше пример является крайне простым. В более сложных системах количество состояний и связей между ними может значительно возрасти. В этом случае может возникнуть большое количество одинаковых переходов к схожим по логике состояниям. Например, при организации разных типов состояний атаки (атака ближнего боя и стрельба). Автомат такого типа крайне тяжело расширять и отлаживать из-за неоправданно большого количества схожих связей. Решить данную проблему поможет преобразование автомата из обычного одноуровневого в иерархический конечный автомат (hierarchical finite state machine или HFSM), где каждое состояние может быть оболочкой для группы схожих по логике состояний.

В качестве примера модернизируем автомат нашего воина (рисунки 1.5 и 1.6):

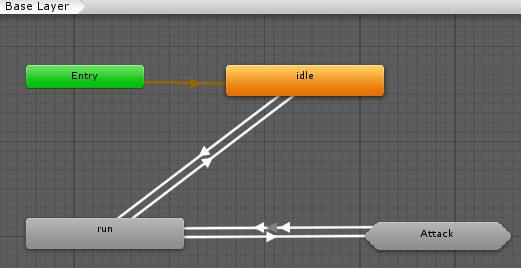


Рисунок . Верхний уровень иерархического конечного автомата

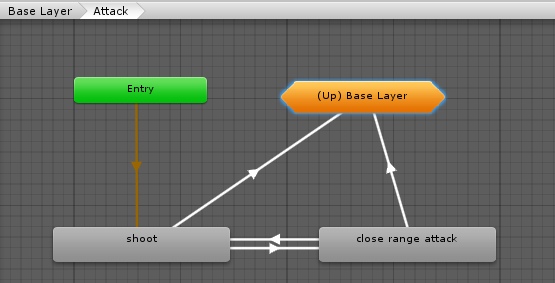


Рисунок . Внутренняя логика состояния атаки

В этом примере мы добавили новый тип атаки (атака ближнего боя) и объединили его вместе с состоянием стрельбы в общее состояние атаки, которое и осталось на верхнем уровне иерархии.

Таким нехитрым способом возможно минимизировать количество дублированных переходов в конечных автоматах, что значительно улучшает общую расширяемость автомата.

Ещё одним достаточно популярным методом реализации логики ИИ является дерево поведений (behavior tree). Данный подход стал популярным после успешного его применения в игре Halo.

В случае дерева поведений логика агента определяется специальным графом, который представляет собой дерево (отсюда первая половина названия метода). Структура графа начинается с единственного корневого узла. В дереве все узлы можно разделить на три типа (рисунок 1.7):

* composite: узлы, имеющие два и более дочерних узлов;
* decorator: узлы, имеющие лишь один дочерний узел;
* leaf: узлы на концах дерева, не имеющие дочерних узлов и обозначающие конкретное действие.

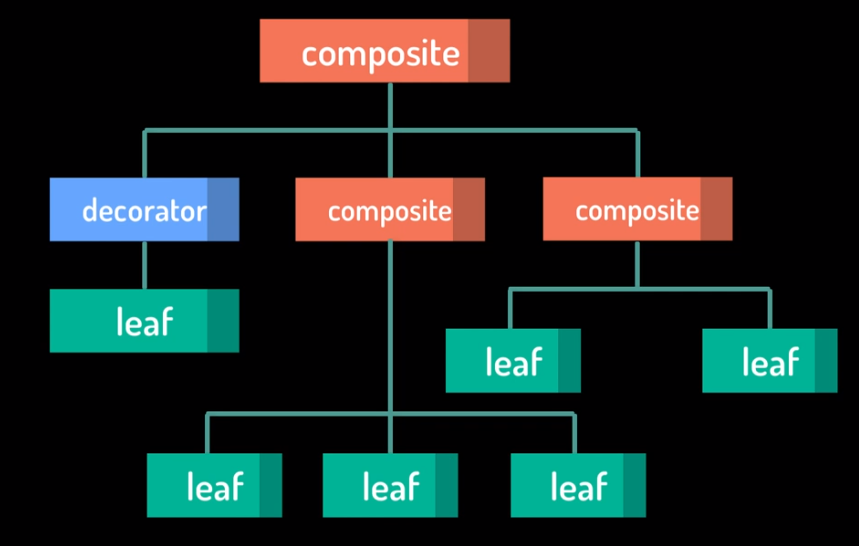


Рисунок . Структура дерева поведений

В свою очередь узлы типа composite делятся на два подтипа:

* selector: сигнализирует о том, что только один дочерний узел будет посещён (рисунок 1.8);
* sequence: обеспечивает, что каждый дочерний узел посещается в порядке следования (рисунок 1.9).

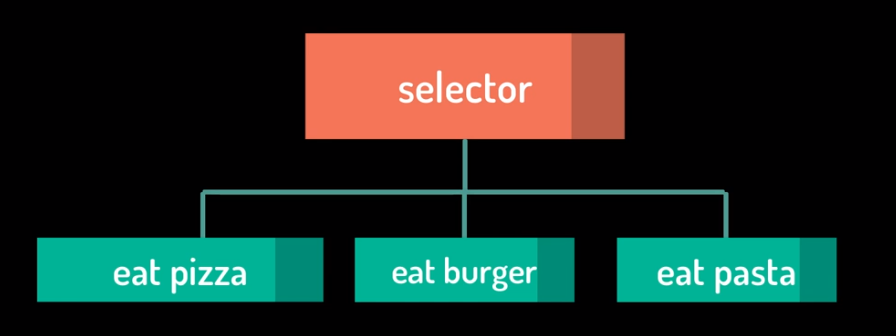


Рисунок . Логика selector’а такова, что он выберет одну из опций в целях утоления голода

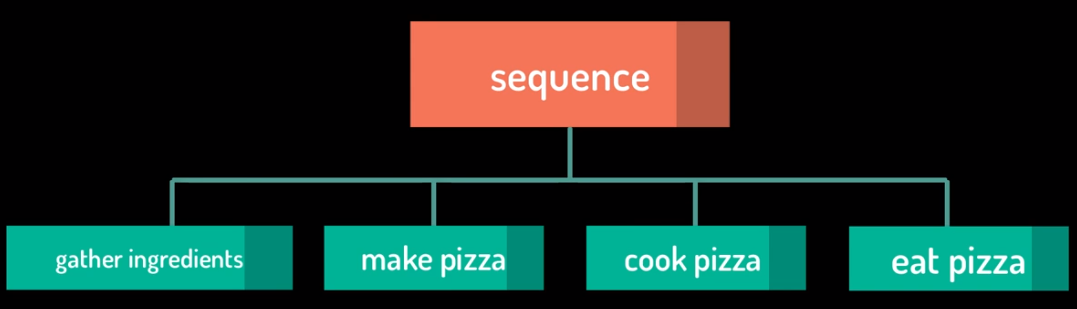


Рисунок . Представлена последовательность действий, которую должен выполнить агент в целях достижения поставленной цели (утоление голода)

Каждый узел возвращает одно из трёх значений:

* Succeded: действие узла завершилось успешно;
* Failed: действие узла завершилось неудачно;
* Running: узел в данный момент в процессе выполнения.

Это лишь общая схема работа дерева поведений. Деревья поведений сложны — есть много способов их составить, да и найти правильное сочетание декораторов и составных узлов может быть проблематично [источник 3]. По этой причине существует большое количество всевозможных реализаций данного метода.

Последним по порядку, но не по значению является алгоритм под общим названием utility-based system (основанная на полезности система). Этот метод предоставляет несколько другой подход к организации выбора решений. Вместо того, чтобы иметь чёткий набор выборов или дерево возможных действий, алгоритм предлагает изучить все действия и выбрать самое подходящее в данный момент на основе какой-либо численной характеристики полезности данного действия, где полезность — произвольная мера того, насколько важно или желательно выполнение этого действия для агента. В общем случае назначается диапазон возможных значений полезности с нижним и верхним пределами. На значение полезностей разных решений могут влиять различные значения внешней среды в данный момент времени. Например, если агент долго не ел и счётчик голода достиг критической отметки, полезность действия «найти пищу» будет стремиться к максимальной из всех возможных. Однако в случае нахождения рядом врагов действие «найти безопасное место» также может заиметь очень высокую меру полезности. Что выберет агент, зависит от реализованной программистом логики, но очевидно, что оба действия в данном примере будут конкурирующими друг с другом и выбор любого из них с точки зрения человека покажется логичным.

Описанные выше подходы не являются единственными и зачастую могут применяться с некоторыми модификациями. Однако при моделировании и реализации нашего собственного ИИ отталкиваться мы будем именно от них.

# Проектирование игрового приложения с искусственным интеллектом

## Обзор современных компьютерных игр и их классификация

Первые компьютерные игры появились лишь в 50-60х годах прошлого столетия, но за это время игровая индустрия совершила огромный скачок в развитии. Очень быстро из каких-то примитивнейших программ игры превратились в произведение искусства [ссылка на источник 4].

Как уже говорилось ранее, на сегодняшний день не существует какой-то единой системы классификации. Помимо этого современные игры, как правило, сочетают в себе сразу несколько разнообразных признаков и их с трудом можно оценить в каких-то определённых рамках.

Естественной обыденностью стала классификация игр по следующим направлениям:

* по платформам, на которые ориентирована игра (игры для настольных компьютеров, для консолей, мобильные игры);
* визуальному представлению (то, как расположена камера, качество графического изображения);
* критериям издателя (инди-игры, free-to-play игры, игры ААА-класса);
* количеству игроков (мультиплеерные и однопользовательские);
* жанрам.

Наибольший интерес представляет классификация именно по жанрам. Нужно понимать, что классификация игр по жанрам также довольно условна. Во-первых, у многих жанров нет четкого, общепринятого, стандартного определения. Во-вторых, разработчики не прекращают своих экспериментов, разбавляя устоявшиеся типы игровых механик новыми элементами. То есть современные игры могут принадлежать к одному или сразу нескольким жанрам, а то и вовсе представлять собой нечто новенькое [источник 4].

Однако выделить несколько основных наиболее распространённых жанров вполне возможно. Список этих жанров выглядит следующим образом:

* action;
* RPG;
* simulator;
* strategy.

Для общего понимания, что из себя представляет каждый из жанров следует более детально разобраться в каждом из них. Пройдёмся по каждому отдельно.

Action

Данный жанр игр характеризуется быстротой происходящих на экране пользователя событий, необходимостью наличия быстроты реакции и грамотной оценки времени. Он объединяет достаточно широкое семейство игр. Поэтому данный жанр обычно разделяют на несколько подгрупп.

Первой подгруппой являются шутеры (от английского shooter, то есть стрелок). Их ещё называют симуляторами стрельбы. Шутеры в основном бывают с видом от первого (first person shooter или FPS, рисунок 2.1) или третьего лица (third person shooter или TPS, рисунок 2.2).



Рисунок . Шутер с видом от первого лица Star Wars Battlefront 2 от Electronic Arts, студия Dice



Рисунок . Шутер с видом от третьего лица The Division 2 от Ubisoft, студия Massive Entertainment

Главной целью игрока в шутере является задача перестрелять своего оппонента, будь тот управляемым ИИ агентом или реальным игроком.

Шутеры являются одним из самых популярных игровых жанров, так как сочетают в себе различные элементы, делающими их привлекательными для практически любой аудитории.

Помимо шутеров в жанр action также входят и файтинги (от английского fighting, то есть сражение). Если в шутерах перед игроком стоит задача расправиться с достаточно обширным количеством оппонентов с помощью огнестрельного оружия, то в файтингах игроку предстоит сражаться, как правило, с одним, реже несколькими оппонентами в ближнем бою (рисунок 2.3).

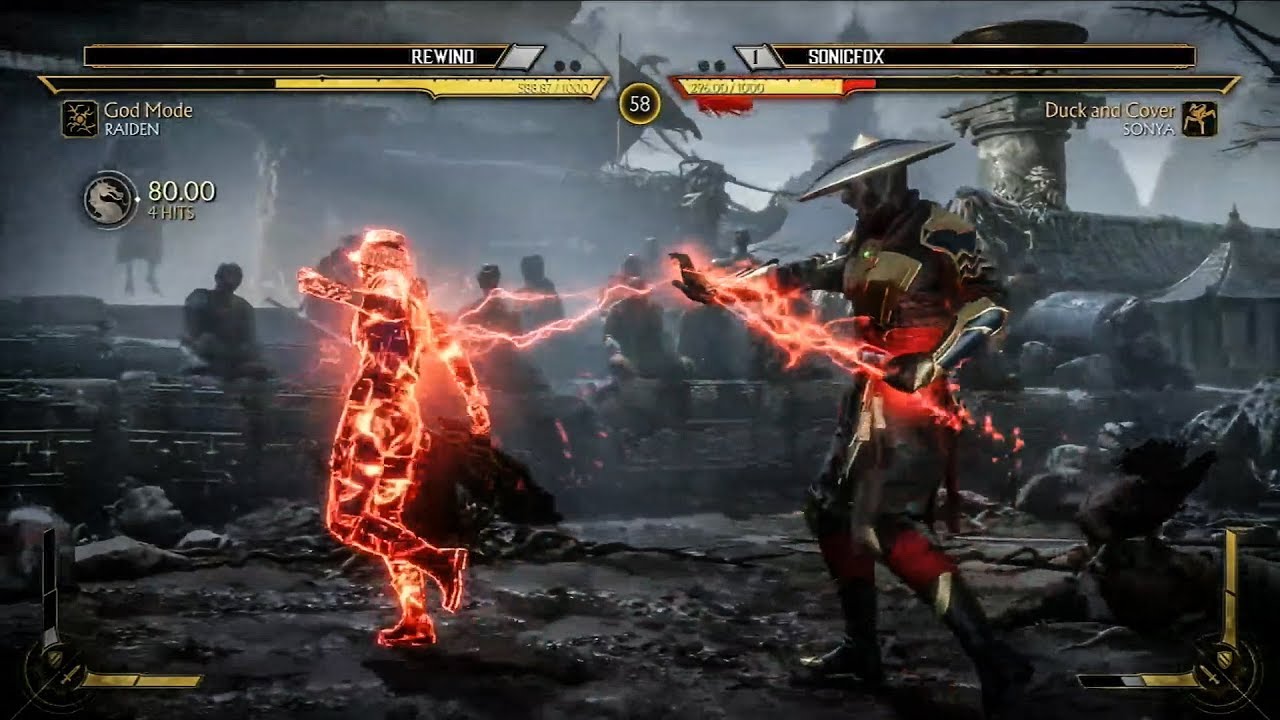


Рисунок . Классический пример файтинга ― серия игр Mortal Combat. Выше представлен скриншот Mortal Combat 11 от WB Games, студия NetherRealm Studios

Ещё одним подтипом жанра action являются слешеры (от английского slasher). Slasher в переводе с английского языка означает «искромсать», «рубить», что в точности описывает основное действо, совершаемое героем слэшера на протяжении всей игры. Основная задача персонажей игр такого жанра заключается в истреблении огромного количества врагов с помощью холодного клинкового оружия (рисунок 2.4) [источник 4].



Рисунок 2.4 Слешер Devil May Cry 5 от Capcom

Последними из основных подтипов жанра action являются платформеры. Из названия вытекает основное отличие данного подтипа ― наличие в нём платформ. Платформы — это самые разнообразные опоры, вроде островков земли, повешенных в воздухе или этажей дома. Цель игрока в платформере ― добраться от начала уровня в его конец, преодолевая препятствия. Как правило, герою платформера необходимо часто совершать прыжки, чтобы преодолевать пропасти, обрывы, двигаясь от платформы к платформе. Помимо преодоления препятствий, от игроков нередко требуется уничтожать врагов (зачастую наскоком) и собирать предметы (рисунок 2.5) [источник 4].



Рисунок . Платформер Sonic Lost World от Sega и Nintendo, студии Sonic Team и Dimps

RPG

RPG или role playing game жанр концентрирует внимание игрока на сюжете игры и развитии одного или нескольких персонажей в контексте этого сюжета. Игрок выбирает путь развития своего персонажа по мере прохождения сюжета, прокачивая его навыки и умения. Нередко в играх такого жанра игроку даётся возможность повлиять на ход сюжета, давая ему возможность выбора в ключевых точках повествования. Зачастую игры RPG-жанра имеют признаки других жанров, например, очень распространены action-RPG, сочетающие в себе черты как RPG, так и action (рисунок 2.6).



Рисунок 2.6 Action-RPG игра Mass Effect 3 от Electronic Arts, студия BioWare

Simulator

Игры жанра «симулятор» симулируют реальность, иначе говоря, пытаются помочь игроку воспроизвести какой-либо реальный опыт [источник 4]. Подкатегорий у симуляторов огромное множество, начиная от симуляторов лётчиков гражданской авиации и заканчивая спортивными и гоночными симуляторами (рисунок 2.7).



Рисунок . Футбольный симулятор FIFA 19 от Electronic Arts

Также существует подкатегория аркадных симуляторов с уменьшенной степенью реализма в угоду более увлекательному игровому опыту (рисунок 2.8).



Рисунок . Аркадный гоночный симулятор Need for Speed: Payback от Electronic Arts, студия Ghost Games

Strategy

Последним жанром в нашем списке являются стратегии. Игры данного жанра характеризуются тем, что игроку для достижения цели необходимо применять стратегическое мышление, и оно противопоставляется быстрым действиям и реакции, которые, как правило, не обязательны для успеха в таких играх (однако есть исключения) [источник 5]. Также в стратегических играх игрок, как правило, как бы отстранён от игрового мира и наблюдает со всем со стороны, отдавая приказы подчинённым ему юнитам.

Стратегии прежде всего принято классифицировать по ходу времени. По этому признаку выделяют следующие типы стратегий:

* стратегии в реальном времени или RTS (Real Time Strategy);
* пошаговые стратегии или TBS (Turn Based Strategy);
* гибридные варианты описанных выше типов.

Начнём разбор с пошаговых стратегий, так они хронологически появились раньше остальных.

Пошаговые стратегии — игры, в которых игроки производят свои действия по очереди. Пошаговые стратегии появились раньше RTS и отличаются значительно большим разнообразием. Разделение игрового процесса на ходы отрывает его от реальной жизни и лишает игру динамизма, в результате чего эти игры не так популярны, как стратегии в реальном времени. С другой стороны, в TBS у игрока гораздо больше времени на размышление, во время совершения хода его ничто не торопит, что позволяет уделять больше времени планированию (рисунок 2.9) [источник 5].



Рисунок . Пошаговая тактическая стратегия XCOM 2 от 2K Games, студия Firaxis Games

Следующими по списку идут гибридные стратегии. Необходимость их появления была обусловлена тем, что в своё время на волне коммерческого успеха пошаговых стратегий (Empire, King`s Bounty, Warlords, SimCity, Civilization, Heroes of Might and Magic) разработчики хотели дать возможность игрокам померяться силами с другими игроками, а не только с ИИ. Код таких игр как Civilization и Heroes of Might and Magic переделывали для возможности игры вдвоем [источник 5]. Но тут всплыла главная проблема концепции пошаговых стратегий: необходимость ожидать окончания хода других игроков. В итоге один ход всех игроков мог растянуться на целый час и более. Ограничение ходов по времени также не сильно помогало, так как, например, даже ограничение хода в 5 минут для четырёх игроков выливалось в среднее время хода около 15-20 минут. Было ясно, что концепция пошаговых стратегий никак не стыкуется с возможностью мультиплеера. Возникновение гибридных стратегий решило эту проблему.

Основная идея гибридных стратегий такова, что игроки вместо поочерёдных ходов совершают ходы одновременные. После окончания ходов всех игроков игра их обрабатывает и переходит к следующему. На ходы также накладывается ограничение по времени, поэтому каждый ход всех игроков не превышает какой-то фиксированный величины времени (рисунок 2.10). Стоит отметить, что решение проблемы величины времени ходов, присущее гибридным стратегиям, ― это, по сути, всё, что отличает их от пошаговых аналогов. Все современный игры, использующие пошаговые механики, для мультиплеера использует гибридный подход. Поэтому понятие гибридности для таких игр применяется крайне редко и их, так же как и пошаговые (в классическом смысле) стратегии, называют пошаговыми.



Рисунок 2.10 Гибридная стратегия Age of Wonders 3 от Triumph Studios

Последними в списке идут стратегии в реальном времени. Игроки совершают свои действия одновременно без деления временных промежутков на ходы. Иначе говоря, игровой процесс обладает свойством непрерывности, а не дискретности (как в пошаговом случае). В настоящее время жанр стратегий в реальном времени стал весьма широк и включает игры совершенно разного типажа – от тактических варгеймов до глобальных стратегий (рисунок) [источник 5].

Помимо классификации по времени есть ещё целое множество различных способов классифицировать стратегии, но все они ориентируются уже, как правило, на сравнение игрового сеттинга, мелких особенностей реализации и наполнения игр, а не на принципиальные подходы к построению игровых механик.



Рисунок 2.11 RTS Men of Wat Assault Squad 2 от 1C Company, студия Digitalmindsoft

Заключение

В заключение, хочется ещё раз отметить, что в наше время «чистых» представителей какого-либо жанра отыскать практически невозможно и даже приведенные выше примеры вполне могут соответствовать и другим, помимо основного, жанрам.

Также нельзя не заметить, что помимо упомянутых выше жанров есть и другие менее распространённые, однако от этого не менее популярные, упоминание которых было решено опустить, так как подробный обзор всех существующих жанров не является основной целью данной работы.

## Функциональные и нефункциональные требования

Написать требования…

## Алгоритмы для реализации искусственного интеллекта

Первым вопросом, который стоит перед разработчиками почти любого искусственного интеллекта, является вопрос навигации агента в игровом пространстве.

И вправду игровой опыт пользователя от игры, в которой боты не способны элементарно передвигаться, скорее всего будет негативным, особенно когда речь идёт об играх с открытым миром.

Steering behaviors

В самых простых случаях, когда игровое пространство представляет собой открытую локацию с минимальным количеством препятствий, используют подход, называемый steering behaviors.

Steering behaviors помогают автономным персонажам реалистично двигаться благодаря применению простых сил, сочетание которых создаёт естественно выглядящее и импровизированное движение по окружению [источник 6].

Принцип данного подхода основан на законах векторной математики и во многом сводится именно к сложению и вычитанию векторов.

Концепция steering behaviors подразумевает наличие нескольких стандартных паттернов логики перемещения агента в пространстве, но каждый из них основывается на плавном изменении вектора скорости объекта под действием управляющих сил и последующем применении изменённого вектора скорости при вычислении следующей позиции объекта. Понятно, что каждый агент в такой системе должен обладать параметрами текущей позиции и скорости, оба из который представляют собой векторы (рисунок 2.12).

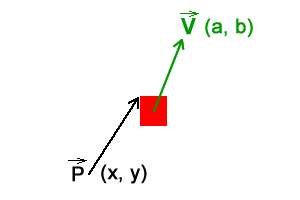


Рисунок . Тут вектор P ― вектор позиции (координаты пространства интерпретируются как векторы), а вектор V ― вектор скорости

Seek

Первым паттерном является паттерн поведения seek (переводится как стремиться). Использование данного паттерна подразумевает наличие цели, к которой движется (стремится) агент.

В случае поведения seek добавление персонажу в каждом кадре управляющих сил заставляет его плавно менять скорость, избегая резких смен маршрута. Если цель сдвигается, то персонаж будет постепенно изменять свой вектор скорости, пытаясь добраться до цели в её новом местоположении.

В поведении seek задействуются два вектора: требуемая скорость и управляющая сила (рисунок 2.13) [источник 6]:

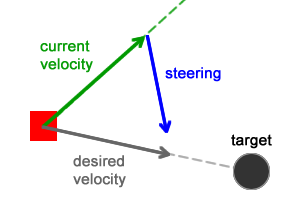


Рисунок . Current velocity ― текущая скорость объекта, desired velocity ― требуемая скорость объекта, steering ― управляющая сила

Требуемая скорость ― это та скорость, к которой стремиться прийти агент. В этом и всех последующих паттернах данное понятие будет присутствовать. По сути своей, требуемая скорость представляет собой вектор той скорости, имея который агент достигнет цели за минимальное время.

Если бы мы просто присваивали требуемую скорость скорости агента, то это бы выглядело нереалистично (агент мгновенно меняет направление) и сказалось бы на игровом опыте пользователя.

Итак, общая логика паттерна seek такова: каждый кадр мы высчитываем вектор управляющей силы:

steering = desired\_velocity - velocity, (2.1)

где

steering ― вектор управляющей силы;

desired\_velocity ― требуемый вектор скорости объекта;

velocity ― текущая скорости объекта.

В процессе игры текущая скорость агента как правило храниться в поле одного из соответствующих агенту классов. А вот требуемую скорость нужно каждый кадр пересчитывать, так как движение объекта в пространстве сводит на ноль актуальность рассчитанной в прошлом кадре требуемой скорости.

Вектор требуемой скорости высчитывается по следующей формуле:

desired\_velocity = (target - position).normalized \* required\_magnitude, (2.2)

где

desired\_velocity ― требуемый вектор скорости объекта;

target ― вектор координат цели;

position ― вектор координат агента;

required\_magnitude ― величина требуемой скорости, скаляр.

В формуле (2.2) проводится операция нормализации разности векторов target и position (запись .normalized) для того, чтобы подсчитанный вектор требуемой скорости не зависел от дистанции между агентом и целью. Этот приём в дальнейшем будет использован не единожды.

С помощью формул (2.1) и (2.2) можно подсчитать вектор управляющей силы, остаётся только приложить его к текущей скорости:

velocity = truncate(velocity + steering \* delta\_time, max\_speed), (2.3)

где

velocity ― скорость агента;

steering ― вектор управляющей силы;

delta\_time ― время, прошедшее между кадрами, скаляр;

max\_speed ― максимальная величина вектора скорости, скаляр.

Операция truncate обеспечивает, что величина вектора velocity не превысит значения max\_speed. Домножение на delta\_time необходимо для обеспечения независимости быстроты изменения скорости от частоты кадров в игре (без этого на высокопроизводительных системах агент будет передвигаться куда быстрее, чем на малопроизводительных). Помимо delta\_time также возможно домножить steering на какой-либо масштабирующий коэффициент (константу), таким образом регулируя скорость поворотов.

После вычисления текущей скорости мы готовы вычислить новую координату агента:

position = position + velocity \* delta\_time, (2.4)

где

position ― координата агента;

velocity ― текущая скорость;

delta\_time ― время, прошедшее между кадрами.

Общая схема движения агента к цели с использованием seek поведения представлена на рисунке 2.14.

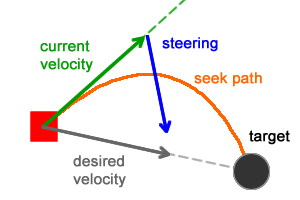


Рисунок . Оранжевым цветом помечена траектория движения агента к цели (seek path)

Flee

Поведение flee (убегание) является противоположным по отношению к seek. Тут агент уже не стремится достигнуть цели, а наоборот отдалиться от неё как можно дальше. Иными словами, если в случае seek вектор desired\_velocity был направлен в сторону цели, то теперь он направлен от неё (рисунок 2.15).

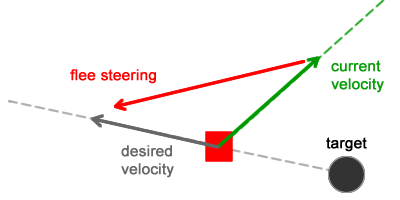


Рисунок . Логика паттерна flee

В виде формуле это свойство можно записать так:

flee\_desired\_velocity = -seek\_desired\_velocity, (2.5)

где

flee\_desired\_velocity ― вектор требуемой скорости убегания;

seek\_desired\_velocity ― вектор требуемой скорости стремления к цели.

Учитывая, что seek\_desired\_velocity вычисляется по формуле (2.2), формулу (2.5) можно переписать:

desired\_velocity = (position - target).normalized \* required\_magnitude, (2.6)

где

desired\_velocity ― требуемый вектор скорости убегания объекта;

target ― вектор координат цели;

position ― вектор координат агента;

required\_magnitude ― величина требуемой скорости, скаляр.

Все остальные формулы паттерна flee полностью совпадают с flee.

Arrival

Поведение seek заставляет персонаж двигаться к цели. Когда он достигает своей цели, управляющая сила продолжает на него воздействовать в соответствии с теми же правилами, заставляя персонаж «скакать» вперёд и назад вокруг цели. Поведение же arrival помогает успешно бороться с этой проблемой. При приближении к конечной точке оно позволяет плавно останавливаться вплоть до полной остановки в точке назначения.

Общая схема поведения состоит из двух этапов:

1. Сначала, когда агент находится на большом расстоянии от цели, он движется по паттерну seek.
2. Когда агент подходит достаточно близко к цели (внутри области замедления), он начинает снижать скорость до полной остановки (рисунок 2.16).

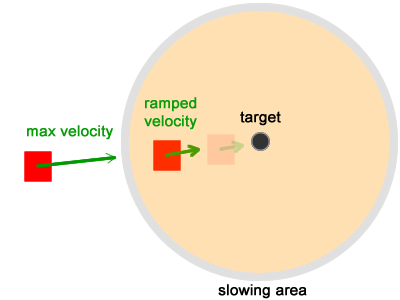


Рисунок . Длина зелёного вектора скорости отражает величину скорости агента в каждой точке

То есть, по сути, перед нами стоит задача высчитать вектор управляющей силы замедления, который должен зависеть от дистанции до цели в радиусе замедления. Логично, что, находясь в позиции цели, агент должен иметь скорость ноль, то есть управляющая сила должна по модулю быть равной его текущей скорости, чтобы при очередном вычислении скорости обнулить её.

desired\_velocity = (target – position).normalized \* required\_magnitude \* (distance / slowing\_radius), (2.7)

где

desired\_velocity ― вектор требуемой скорости;

target ― вектор позиции цели;

position ― вектор позиции агента;

required\_magnitude ― величина требуемой скорости без учёта замедления, скаляр;

distance ― расстояние от агента до цели;

slowing\_radius ― дистанция замедления.

Данная формула применяется тогда, когда агент находится в зоне замедления, то есть подошёл достаточно близко (distance <= slowing\_radius). Иначе используется стандартная формула поведения seek.

Далее вычисляется значение управляющей силы по уже описанной формуле (2.1).

Нетрудно видеть, что величина управляющей силы в случае прибытия агента в точку назначения будет равна величине скорости по модулю и противоположно ей направлена, то есть итоговая скорость после воздействия на неё управляющей силы будет равна нулю. Именно этого мы и добивались.

В конечном итоге, паттерн arrival можно рассматривать как более универсальное продолжение паттерна seek.

Wander

Паттерн wander (блуждать) подразумевает случайное блуждание персонажа по какой-либо территории. Если игрок увидит чётко заданные маршруты или нереалистичное перемещение, то это приведёт к раздражению. В самом худшем случае игрок поймёт, как предсказывать движения персонажа, и тогда игровой процесс станет для него скучным. Управляющее поведение wander предназначено для создания реалистичного «естественного» движения, которое убедит игрока, что персонаж на самом деле живой и самостоятельно ходит [источник 6].

Существует достаточно большое множество различных реализаций этого паттерна. Самым простым из них является использование паттерна seek в связке с периодическим заданием новой точки назначения.

Pursuit

При использовании паттерна pursuit агент стремиться догнать движущуюся в пространстве цель. Его отличие от паттерна seek состоит в том, что агент пытается предугадать будущую позицию цели и направляется именно к ней (рисунок 2.17).

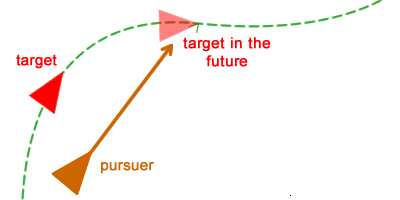


Рисунок .

Для предугадывания будущей позиции цели используется немного изменённая формула (2.4):

future\_position = position + velocity \* T, (2.8)

где

future\_position ― предполагаемая координата цели в будущем;

position ― текущая координата цели;

velocity ― текущая скорость цели;

T ― время от текущего момента времени, на которое производится предугадывание.

Параметр T в формуле (2.8) обычно подсчитывают динамически, при этом он должен зависеть от скорости цели и дистанции от цели до агента:

T = distance\_between\_target\_and\_pursuer / max\_velocity, (2.9)

где

T ― время от текущего момента времени;

distance\_between\_target\_and\_pursuer ― дистанция между целью и агентом;

max\_velocity ― максимальная скорость цели.

Данный трюк необходим для актуализации вычисляемой координаты цели в будущем (абсурдно считать позицию цели в далёком будущем, находясь к ней очень близко).

В остальном же движение к вычисленной будущей позиции цели происходит по паттерну seek.

Evade

Поведение Evade противоположно поведению Pursuit. Вместо стремления к будущей позиции цели при поведении Evade будет убегать от этой позиции (рисунок 2.18) [источник 6]:

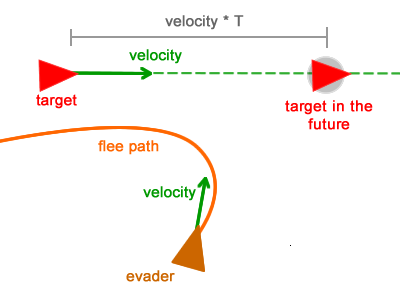


Рисунок .

Для удаления от цели в этом случае используется паттерн flee.

Комбинирование паттернов

Элегантность steering behaviors заключается в том, что все выше описанные паттерны можно легко комбинировать. Например, если агент должен одновременно достичь какой-либо цели, но в то же время убежать от другого агента. В этом случае напрашиваются паттерны seek и evade. Для применения обоих паттернов одновременно нужно просто вычислить значения управляющих сил для каждого из паттернов и сложить их. Полученный вектор необходимо промасштабировать и применить к скорости по уже известной формуле (2.3).

Алгоритмы поиска наикратчайшего пути

Описанные выше паттерны позволяют агентам реалистично передвигаться в пространстве, используя простые правила векторной математики. Однако для более сложных систем такого подхода будет недостаточно. Например, пользователь, приказав агенту двигаться в определённую точку локации, на пути к которой будет много препятствий, ожидает нахождения агентом минимального по длине маршрута к этой цели, чего простые steering behaviors в чистом виде обеспечить не могут. Тут нам на помощь приходят алгоритмы поиска кратчайшего пути на графах.

Для использования алгоритмов поиска на графах на игровые локации обычно помещает сетки возможных путей (путей, по которым может пройти агент). Они визуально представляют собой ни что иное как граф (рисунок 2.19).

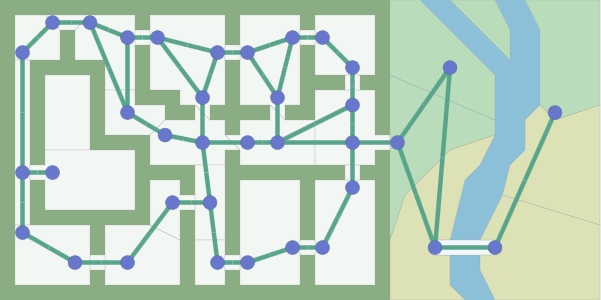


Рисунок . Простейший пример путевой сетки (визуализирована для наглядности, в процессе игры игроки её, как правило, не видят)

Таким образом, агент при ориентации в пространстве видит карту именно как набор узлов, связанных между собой рёбрами. В английской литературе такая организация взаимодействия агентов с миром называется просто waypoints (путевые точки, рисунок 2.20).



Рисунок . Красные шары – визуализированные waypoint’ы

Более продвинутой версией путевых точек является так называемый навигационный меш (navigation mesh). Навигационный меш представляет собой набор двухмерных полигонов, обозначающих зоны, по которым агент может передвигаться. Переходя на язык графов, по сути, каждый полигон является узлом графа, в котором производится поиск пути (рисунок 2.21).



Рисунок . Синим цветом нарисован навигационный меш, в котором легко различимы полигоны ― составные части меша

Мы определились, каким образом представить игровое окружение для агента в понятном ему виде. Но теперь возникает вопрос, каким же образом найти кратчайший путь.

Конечно, при поиске кратчайшего пути в графе первыми приходят на ум:

1. Алгоритм поиска в ширину
2. Алгоритм Дейкстры

Каждый из них неплохо себя показывает в определённых ситуациях, однако алгоритм поиска в ширину хорош тогда, когда не нужно учитывать стоимость перехода из узла в узел, а алгоритм Дейкстры в чистом виде редко когда применяется. Применяется именно модифицированная версия алгоритма Дейкстры, называемая A\*.

Разберём принцип работы алгоритма A\*.

Напишем функцию, реализующую алгоритм A\*, с помощью псевдокода.

// Функция принимает в качестве параметров текущее положение агента

// и конечное положение цели, а также сам граф поиска

function AStarAlgorithm(start, finish, graph):

// Приоритетная очередь ещё не посещённых вершин

PriorityQueue frontier = new PriorityQueue()

// В приоритетную очередь кладётся стартовая позиция агента

// Вторым параметром является вес данной позиции в очереди

// Чем ниже вес, тем выше приоритет

Frontier.Put(start, 0 + HeuristicFunc(start, finish))

// Контейнер для хранения узлов, из которых был достигнут

// данный узел

Map cameFrom = newMap()

cameFrom[start] = null

// Контейнер для хранения цен, за которые были достигнуты

// узлы

Map costSoFar = newMap()

costSoFar[start] = 0

while (frontier.NotEmpty())

current = frontier.Get()

frontier.Pop()

if (current == finish)

break

for neighbor in graph.Neighbors(current)

newCost = costSoFar[current] + graph.Cost(current, neighbor)

if neighbor not in costSoFar or newCost < costSoFar[neighbor]:

costSoFar[neighbor] = newCost

priority = newCost + HeuristicFunc(finish, neighbor)

frontier.Put(neighbor, priority)

cameFrom[neighbor] = current

// Если конечного узла нет в этой коллекции, значит конечный узел

// не достижим из заданной начальной точки

if (finish not in cameFrom)

return null

// Осталось построить путь к цели, который и будет возвращён

// в качестве результата. Для этого используется коллекция

// cameFrom

List resultPath = new List()

current = finish

while (current != start)

resultPath.PushBack(current)

current = cameFrom[current]

return resultPath

Алгоритм A\* очень похож на алгоритм Дейкстры и почти вся его суть в точности копирует алгоритм Дейкстры, поэтому его можно смело назвать модификацией алгоритма Дейкстры. Основное же отличие от алгоритма Дейкстры состоит в том, что нужно найти один путь до одной конкретной вершины, в то время как алгоритм Дейкстры ищет кратчайшие пути сразу ко всем вершинам графа. Конечно, можно, используя алгоритм Дейкстры, останавливать поиск, когда мы дойдём до нужной нам вершины, после чего восстановить путь именно до неё, и получить тот же самый результат, что и при использовании алгоритма A\*, однако тут сказывается второе отличие алгоритма A\* от алгоритма Дейкстры ― наличие эвристической функции (в псевдокоде называется HeuristicFunc).

Эвристическая функция, так сказать, помогает задать направление поиска. Реализация эвристической функции не фиксирована строго. Однако зачастую эвристическая функция оценивает расстояние до цели:

function HeuristicFunc (a, b):

return Sqrt(Pow(a.x - b.x, 2) + Pow(a.y - b.y, 2) + Pow(a.z - b.z, 2))

На рисунках 2.22 и 2.23 наглядно показана разница результатов работы алгоритмов Дейкстры и A\*.

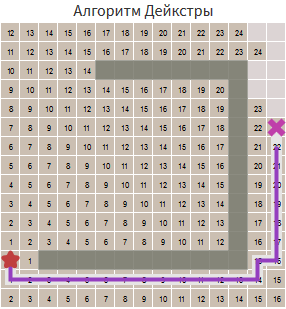


Рисунок .

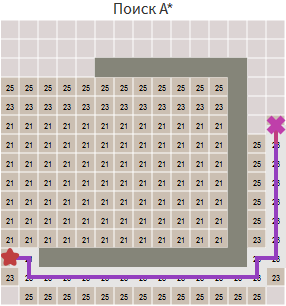


Рисунок .

Нетрудно видеть, что, обеспечивая тот же результат что и алгоритм Дейкстры, алгоритм A\*, перебирает куда меньше узлов, таким образом ускоряя общую скорость работы алгоритма, что может быть критически важно, когда в игровом пространстве находится сразу множество агентов, для каждого из которых нужно быстро посчитать путь до цели.

## Архитектура приложения

Расписать архитектуру…

# Реализация искусственного интеллекта в компьютерной стратегии в реальном времени

## Используемые технологии для реализации компьютерной игры

Для реализации компьютерной стратегии в реальном времени было принято решение использовать игровой движок Unity, который можно легко скачать с официального сайта бесплатно.

Целевой платформой для игры будут персональные компьютеры, так как именно они обеспечивают достаточное количество устройств ввода для приемлемого уровня управления в игре в отличие от консолей или мобильных устройств.

## Создание прототипа