Министерство образования Республики Беларусь

Учреждение образования

Белорусский государственный университет

информатики и радиоэлектроники

УДК\_\_\_\_\_\_\_\_\_\_\_

Тумилович

Сергей Игоревич

Метод оптимизации поведения персонажей в компьютерной игре

**ДИССЕРТАЦИЯ**

на соискание степени магистра \_\_\_\_\_\_\_\_\_\_\_\_ наук

по специальности \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |
| --- |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |
| Научный руководитель |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Минск 2018

Нормоконтроль

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оглавление

[Оглавление 2](#_Toc501440478)

[Условные обозначения и термины 3](#_Toc501440479)

[Общая характеристика работы 4](#_Toc501440480)

[Введение 7](#_Toc501440481)

[Глава 1 9](#_Toc501440482)

[МЕТОДЫ РАЗРАБОТКИ КОМПЬЮТЕРНЫХ ПЕРСОНАЖЕЙ 9](#_Toc501440483)

[1.1 Классификация NPC в видеоиграх по жанрам и сложности 9](#_Toc501440484)

[1.2 Актуальные решения в разработке и оптимизации NPC 17](#_Toc501440485)

[1.3 Функциональные основы поведения NPC 21](#_Toc501440486)

[Глава 2 40](#_Toc501440487)

[Метод оптимизации действий NPC Action First 40](#_Toc501440488)

[2.1 Требования к игре и NPC при применении метода 40](#_Toc501440489)

[2.2 Понятия и алгоритмы метода Action First 42](#_Toc501440490)

[2.3 Алгоритм Action First 51](#_Toc501440491)

[2.4 Пример работы метода Action First 54](#_Toc501440492)

[2.5 Программные основы реализации Action First 58](#_Toc501440493)

[Глава 3 60](#_Toc501440494)

[Разработка и реализация ПО метода 60](#_Toc501440495)

[3.1 Структура, функции, модули продукта Action 60](#_Toc501440496)

[3.2 Методы Action First в библиотеке Action.Core 61](#_Toc501440497)

[3.3 Визуализация очередей действий в Action.Queue 74](#_Toc501440498)

[1.4 Особенности работы Action.Queue в автономном режиме 80](#_Toc501440499)

[Заключение 81](#_Toc501440500)

[Список использованных источников 82](#_Toc501440501)

[ПРИЛОЖЕНИЕ А 83](#_Toc501440502)

[(обязательное) 83](#_Toc501440503)

[Программный модуль отображения маршрутов 83](#_Toc501440504)

[(информационное) 84](#_Toc501440505)

[Описания кодов сообщений для модуля протоколирования 84](#_Toc501440506)

# Условные обозначения и термины

*Геймплей* – совокупность объектов игры, с которыми игроки и компьютерные персонажи могут взаимодействовать, и действий игры, которые игроки и компьютерные персонажи могут выполнять.

*NPC* (None-Played Character) – персонаж игры, который не управляется игроком напрямую и может выступать союзником либо противником игрока.

*Моб* – NPC, который действует, основываясь на предопределенных схемах деятельности (скриптах) вместо использования возможностей виртуального интеллекта для своей деятельности.

*ИИ* (Игровой интеллект) – реализация искусственного интеллекта, применяющаяся в играх для регуляции поведения игровых персонажей.

*Бот* – NPC, который может использовать в своих действиях разнообразные механизмы ИИ, включающие машинное обучение, нейросети, дискретные методы и т.д., имитируя человекоподобное поведение.

*Контекст действий* – относительно изолированная совокупность действий игровых персонажей и игрока Совокупность контекстов образует собой геймплей.

*Действие персонажа* – определенный алгоритм, посредством которого игроки и NPC реализуют своё поведение в игре, достигая своих целей.

*Домен действий персонажа* – совокупность действий, которые вообще могут быть выполнены персонажем в том или ином контексте действий.

*Цепь действий персонажа* – последовательность действий, которые выполняются последовательно или параллельно, согласно свойствам конкретной цепи.

*Конфигурация цепи действий* – схема выполнения действий, определяющая условия и временные интервалы для действий персонажа.

*Очередь действий* – последовательность выполнения действий или цепей действий для персонажа, наделенная средствами количественной и качественной оптимизации, а также методами саморегуляции состояния персонажа в игровом мире.

*Ситуация (событие)* – состояние игрового мира, которое может вызвать действия (цепи действий) персонажей либо прекратить их.

*Цель игрока* – определенное по некоторым признакам состояние игры, при котором игрок достигает победы.

*Цель персонажа* – определенное состояние игры, при котором выполняется вся цепь действий этого персонажа.

# Общая характеристика работы

Целью работы является разработка метода для оптимизации действий компьютерных персонажей (здесь и далее – NPC) в условиях неопределенности действий NPC и игроков-людей при динамически изменяющихся состояниях игры. В методе реализована особая механика распределения действий NPC и их регуляции в зависимости от игрового времени и от конкретной ситуации в локации.

Метод предназначен для оптимизации действий NPC в играх со следующими особенностями:

* наличие у NPC действий, которые могут выполняться не по предопределенным цепочкам, а в зависимости от текущей ситуации;
* динамически изменяющийся игровой мир, в частности быстрое и непредсказуемое изменение решений игрока;
* потребность в конкурентоспособном для человека игровом интеллекте, который может принимать актуальные на каждый момент времени решения;
* условия неопределенности, когда игрок не может точно описать (но может спрогнозировать или предсказать) последующие действия компьютерных оппонентов.

Для выполнения данной цели были исследованы и реализованы следующие задачи:

* ознакомление с особенностями построения NPC в играх различного класса;
* ознакомление с методами разработки NPC из различных предметных областей;
* ознакомление с методами прикладного программирования действий NPC в играх;
* разработка методики определения действий компьютерных персонажей;
* применение положений теории информации для методики оценки и ранжирования действий NPC в условиях неопределенности;
* разработка технологии определения действий компьютерных персонажей использования в игре;
* разработка прикладного ПО для осуществления методики;
* реализация приложения-интерфейса для визуализации очередей действий.

Многие из данных задач напрямую связаны с приоритетными ныне для исследования областями геймдизайна и разработки игр. Диссертация связана, в частности, с предметными областями проектирования NPC, общим геймдизайном, проектированием и разработкой игрового интеллекта. Разработанный в диссертации метод вносит новые приемы и алгоритмы в данные области, которые готовы к апробированию и внедрению в коммерческие игровые проекты.

Некоторые задачи, например, последние три, связаны как с вопросами прикладной разработки игрового и вспомогательного ПО, так и с вопросами внедрения математических моделей в данное ПО. В частности, при разработке прикладного ПО Action First внедрены модели, связанные с положениями теории информации. Благодаря им был достигнут качественный уровень внутренней регуляции алгоритмов NPC, позволяющий их гибкую настройку. Такие нововведения открыли ряд ценных возможностей в настройке сложности NPC, однако повлекли усложнение самой модели продукта и повысили требования к геймдизайну целевых игр.

Таким образом, я разработал метод оптимизации поведения NPC, чем внес посильный вклад в вышеописанные предметные области. Разработанный метод имеет следующие достоинства перед аналогами:

* значительное повышение эффективности действий NPC относительно игрока;
* замена всех мобов игры ботами, умеющими рассчитывать своё поведение, а не только поступать по предопределённым скриптам;
* оптимизация выбора действий NPC, реализованная с возможностью их реорганизации, моментальной отмены и параллельного выполнения;
* использование положений теории информации;
* разработанный комплекс ПО, предоставляющий как отдельный продукт, содержащий средства для реализации подхода, так и средство визуализации поведения NPC;
* гибкая настройка алгоритмов, в частности, уровня сложности и порога отмены действий.

Как и любой другой метод, мой имеет некоторые недостатки:

* повышение сложности разработки игры;
* сложности внедрения в игры в качестве модификации;
* «проклятие размерности», выражающееся в зависимости сложности алгоритма от количества обрабатываемых NPC и размеров домена действий;
* внедрение продукта в качестве внешнего средства отладки, что вызывает проблемы с ограниченной совместимостью с различными играми.

Эти недостатки, будучи актуальными на данный момент, предоставляют возможности совершенствования метода. Например, первый недостаток (повышение сложности разработки), будучи связан с повышенным требованием к качеству правил игры и зависимостей между действиями NPC, можно устранить путем внедрения совместимого единого подхода к проектированию целей (т.н. Goal Design), аналогичного концепции Goal Architecture в области ИИ [1]. Эти и многие другие улучшения будут исследованы и применены в результате соответствующей планируемой мною кандидатской диссертации.

Результаты диссертации представлены в виде разработанного алгоритма метода. Алгоритм, в свою очередь, реализован в виде комплекса ПО Action, представленного динамической библиотекой методов алгоритма Action.Core и приложением визуализации очередей действий Action.Queue. Промежуточные и окончательные результаты исследований представлены на 53-й студенческой конференции БГУИР и на конференции ITIS в виде опубликованных статей.

Продукт Action, являясь практическим результатом реализации метода, успешно продемонстрирован на конкретных примерах тестовой игровой ситуации, а также в самой игре в стандартном режиме работы.

# Введение

Области дизайна и разработки игр относятся к наиболее творческим предметным областям IT. Множество методов и подходов (а в геймдизайне абсолютное большинство) подразумевают творческую и инновационную составляющие, поэтому предлагают уникальные, но разовые возможности по созданию игры. По этой причине, а также в силу молодости предметной области, у разработки и дизайна игр нет универсальной схемы реализации механик игры. Это влечет изобилие методик взаимодействия игрока и NPC, организации геймплея, интерактивности, и т.д.

Все эти приемы обычно индивидуальны для каждой новой игры, хотя крупные фирмы и студии могут придерживаться одних и тех же правил при работе над своими играми, чтобы сэкономить ресурсы и привести свои игры. Поэтому в данный момент ведутся исследования общей методики разработки, которые, несмотря на затраченные усилия [2], пока не привели к результату.

Все подобные исследования и разработки требуют ресурсов, количество которых разнится в зависимости от сложности, класса и жанра игры. Так, проектирование NPC особенно важно для standalone-игр, поскольку NPC в них играют ключевую роль. Для онлайн-игр есть своя специфика настроек NPC, поскольку ключевую роль там играют не они, а сами игроки-люди.

Большую роль играют пока что неудовлетворенные потребности людей в интересных и сравнимых по способностям с игроками-людьми компьютерных персонажах. Увы, абсолютно человекоподобного поведения NPC достичь средствами современных концепций невозможно (поскольку сама природа любого ИИ в нашем традиционном понимании не является человекоподобной), поэтому проблема сводится к максимально правдоподобной имитации человекоподобных алгоритмов.

Таким образом, природа проблемы – создание имитационных алгоритмов для повышения конкурентоспособности NPC по сравнению с человеком в игре. Первопричина потребности людей и команд разработки в методах проектирования NPC состоит в необходимости баланса между развлечением людей и испытанием их при столкновении с могущественным компьютерным оппонентом в игре.

Поведение NPC должно быть, с одной стороны, оптимизировано для эффективного противодействия или содействия игроку-человеку, с другой стороны – быть достаточно гибким, чтобы соответствовать стандартам и сложности игры, позволяя человеку получать от игры положительные эмоции.

Весьма важно, чтобы, решая проблемы в игре, человек испытывал удовольствие и эмоциональный подъем, и это не должно переходить в испытание, непосильное для человека. Игра должна быть сделана так, чтобы сложность поведения компьютерного оппонента способствовала приближению к этому (на данный момент абстрактному [3]) «эмоциональному балансу», а не смещалась в сторону запредельной трудности или крайней лёгкости.

Таким образом, существует необходимость для новых методик, качественно улучшающих функционирование NPC, имитирующее человекоподобную деятельность. Поэтому разработанный в диссертации метод весьма актуален и востребован. Дополнительной стороной вопроса является период активного пользовательского тестирования, применения, согласования вопросов бизнеса и коммерции. Поэтому все подобные продукты внедряются с некоторым «испытательным сроком», в течение которого и подсчитывается их эффективность. Кроме этого, любой разрабатываемый метод может иметь различные реализации, а пользователи и разработчики могут обнаружить уязвимости реализаций метода и перспективы для его развития. Поэтому метод Action First со временем получит развитие в виде своих новых версий, доработанных с учетом обнаруженных перспектив развития.

# Глава 1

# МЕТОДЫ РАЗРАБОТКИ КОМПЬЮТЕРНЫХ ПЕРСОНАЖЕЙ

Для формулирования структуры, понятий и механики регуляции поведения NPC в моём методе, необходимо сформулировать задачу, базируясь на особенностях разработки NPC в играх и на уже существующих методах оптимизации их поведения.

## Классификация NPC в видеоиграх по жанрам и сложности

Ширина проблемы разработки NPC варьируется в зависимости от типа, класса и жанра игры. В зависимости от жанра разработчик сталкивается с различными базовыми задачами разработки, а класс определяет ресурсные ограничения на реализацию.

Проблематика проектирования персонажей в разных жанрах (или комбинациях жанров) игр охватывает разные области знаний. Поэтому в разных жанрах и классах игр существуют разные требования к поведению NPC. Необходимо проанализировать их и определить их ключевые моменты, а уже потом перейти к рассмотрению способов их реализации.

### Поведение NPC в играх жанра Action

Жанр Action включает в себя поджанры Arcade и First-Person Shooter (и Third-Person Shooter, как его вариант). Их особенность - *ориентированная на игрока* динамическая обстановка. NPC в играх жанра Action – это преимущественно противники, с которыми игрок ведет бой. Хотя встречаются и союзники, которые поддерживают игрока в бою с врагами. Обычно временная система таких игр – система реального времени, разбитая на временные такты (элементарный промежуток времени, во время которого рассчитываются все действия в игре). Пример взаимодействия персонажей в игре жанра Action на примере Call of Duty Modern Warfare представлен на рисунке 1.1.



Рисунок 1.1 – NPC в Call of Duty Modern Warfare

В играх жанра Action есть следующие особенности NPC:

* выполнение персонажами действий в зависимости от текущей ситуации и от текущих действий игрока;
* компьютерные персонажи часто не взаимодействуют друг с другом, когда игрок с ними никак не взаимодействует (мир, зависимый от действий игрока);
* каждый NPC (в идеальном варианте) является ботом с индивидуальным набором скриптов либо (в подавляющем большинстве случаев [4]) – мобом, настроенным на типовые действия и потому не выдерживающим конкуренции с игроком;
* индивидуальные для конкретной игры особенности, такие как групповые тактики, использование окружения (например, укрытий), использование персонажами уникальных умений, и т.д.

В данном жанре NPC должны быть способны к быстрому принятию локального решения, оптимального с точки зрения решения игрока. Поэтому, задача оптимизации поведения может быть сведена к задаче поиска локального действия, в каждый момент времени максимально удобного для NPC и максимально неудобного для игрока. Степень приближения фактического критерия к данному характеризует сложность игры (в настоящее время достигаемое благодаря повышению игровых характеристик NPC, а не благодаря его поведению).

### NPC в жанре симуляторов

Жанр симуляторов включает в себя игры-симуляторы (например, Kerbal Space Program) и обучающие игры (например, симуляторы вождения). NPC в таких играх применяются редко, поскольку такие игры ориентированы на прокачку личного навыка игрока (skill), а не на его взаимодействие с другими персонажами. Проблематика оптимизации поведения во многом зависит от того, *что* симулируется (например, для симулятора вождения как NPC могут выступать другие автомобили).

Учитывая тот факт, что симуляторы имитируют работу с какими-то объектами, имеющими отражение в реальном мире, все объекты игры в таком случае создаются с правдоподобной геометрией, физикой, с реальными реакциями на действия игрока. Поэтому NPC, если в них всё же есть необходимость, должны быть максимально приближены к правдоподобному с точки зрения реального мира поведению. Это не означает, что они обязательно должны быть человекоподобны (поскольку в таких играх далеко не всегда NPC отражают субъектов реального мира), поэтому зачастую используются детерминированные последовательности действий, выглядящие в играх этого жанра и без оптимизации уместно.

### Особенности NPC в стратегиях

Стратегии имеют ключевую особенность – все персонажи одной фракции (юниты) в ней управляются **одним** игровым интеллектом. Реальными компьютерными персонажами-NPC в стратегиях являются «фракции»: государства, если игра про управление странами, армии, если это тактическая стратегия, и т.д. Взаимодействие персонажей в стратегиях представлено на рисунке 1.2.

Все стратегии, несмотря на их разделение на пошаговые (Turn-based) и стратегии реального времени (Real-Time Strategy), имеют общую черту с точки зрения теории игр – неравновесную многомерную модель, в которой все участники стремятся к максимальному выигрышу для себя и/или для союзника (а иногда к минимизации выигрыша всех своих противников). При этом, в пошаговых стратегиях ходы соперников неизвестны заранее, в стратегиях реального времени каждый NPC принимает решения одновременно с остальными участниками. Несмотря на удобство описания данной модели с помощью теории игр, на практике оно не применяется из-за «проклятия размерности».



Рисунок 1.2 – Взаимодействие персонажей в Empire Total War

Персонажи стратегий обязаны быть наделены работоспособными глобальными правилами, очень похожими на правила, реализованные в шахматных ИИ [5]. Однако, в зависимости от класса игры (см.п.1.1.6) правила могут изменяться и дополняться, включая в себя новые методы. Широко используются имитационные алгоритмы, имитирующие деятельность игрока-человека с некоторой степенью точности. Благодаря отсутствию в настоящее время возможности правдоподобной имитации такого поведения, широко используются упрощения, связанные с линейными модификаторами («завоеватель», «надежный», «защитник» - эти и иные модификаторы хорошо отражают характер NPC и его склонность к тем или иным действиям в игре). К сожалению, механизм линейных модификаторов является предопределённым, что является основной причиной быстрой адаптации игрока и, в дальнейшем, лёгкой победы игрока над любым таким NPC благодаря новой стратегии человека против неизменной стратегии NPC.

### NPC в жанрах Adventure и Quest

Приключения (Adventure Games) – игры, включающие в себя сюжетно-ориентированные аркады, FPS, фильмы-игры, квесты. Главное их отличие от жанра Action в ориентации **на сюжет**, а не на действия.

Поэтому компьютерные персонажи в них проектируются с **расчетом участия в сюжете** (то есть, воспроизведения какой-либо его части оригинального контента), а не на ответ действиям игрока.

В обоих этих жанрах NPC имеют одинаковую роль и принципы реализации. Дело в том, что оба жанра предполагают вспомогательную детерминированную роль для всех NPC, поскольку ориентированы на самого игрока и на решение им разнородных головоломок. Поэтому, в таких играх (особенно в квестах) используются детерминированные последовательности действий, которые обуславливаются не их необходимостью с точки зрения игровой ситуации, но осуществляются по мере продвижения игрока по сюжету. Поэтому в классических квестах нет необходимости в ботах, которые способны на самостоятельные действия.

Порой NPC в играх Adventure могут приобретать черты NPC из Action – в таком случае, необходимо придерживаться при их реализации тех же правил, что и в Action.

### NPC в ролевых играх

Ролевые игры (RPG) – в настоящий момент ведущий жанр индустрии разработки игр, поскольку может объединять в себе все вышеописанные жанры и даёт массу возможностей как для игроков, так и для разработчиков. Главные особенности жанра таковы:

* подбор для игрока и NPC *ролей*, которые они исполняют в игре;
* динамическая смена ролей и смена степени влияния роли в той или иной ситуации;
* нелинейный и, возможно, не определенный заранее сюжет, часто применяется выбор игрока, последствия которого влияют на развитие сюжета;
* динамично развивающийся и существующий независимо от игрока мир;
* системы прокачки персонажей, такие как навыки, уровни, достижения, и т.д. Это может относиться не только к самому игроку, но и к NPC, что открывает весьма ценные возможности их модификации.
* очень сложные взаимоотношения компьютерных персонажей с игроком и друг с другом, вплоть до «симуляции отношений» [6];
* неопределенность в практически любой игровой ситуации, когда исход и решение ее зависит не от созданных разработчиками условий, а от действий игрока и других персонажей.



Рисунок 1.3 – Взаимодействие персонажей в RPG Mass Effect 3

Согласно данным особенностям, разработчик должен знать о следующих особенностях своих NPC в данном жанре:

* система ролей, поведение NPC должно быть различным для каждой его роли;
* сочетание предопределенных сюжетных поворотов и неопределенных линий поведения, которые формируются путем принятия решения игроком или даже самим NPC;
* соответствие NPC существующим и приобретенным способностям игрока, который получает новые уровни и умения, двигаясь по своей системе прокачки;
* система реагирования NPC на взаимные решения и решения игрока в любой игровой ситуации (что и реализует метод Action First).

Разработка персонажей RPG – крайне сложный труд, так как надо предусмотреть множество возможностей: от боевой механики до участия в сюжете и эмпатии со стороны игрока. С точки зрения разработки, в каждом из важных NPC RPG должны сочетаться как методы программирования игр и ИИ, так и результаты гуманитарных разработок (сценарий, сюжет, роль живого актера, кинематография, озвучка, и т.д.).

Одним словом, в качественных RPG большинство (а то и все) персонажи подвержены **индивидуализации**, с точки зрения своих особенностей и способностей. В силу всех этих сложностей вышеописанные задачи на данный момент решаются заменой сложных алгоритмов скриптами и триггерами, которые предоставляют очень приблизительную имитацию решения, зато дешевы и достаточно удобны в использовании [7]. Данная проблема – одна из первопричин потребности в оптимизации поведения NPC моим методом.

Простор экспериментов с параметрами персонажей позволяют внедрять инновационные методы, повышающие их эффективность. Поэтому RPG наиболее часто являются новаторами в разработке и дизайне игр и целью для новых методов оптимизации поведения NPC.

### Задачи NPC в классах игр

В зависимости от сложности игры условно разделяют на следующие **классы** [8]:

* Casual (класс С). Игры данного класса разрабатываются с максимальной простотой, так как делается ставка на простоту геймплея и на посещаемость (привлечение игроков), часто за счет ухудшения качеств геймплея, персонажей и т.д. Некоторые игры (например, игры жанра «три-в-одном») изначально просты, поэтому относятся к этому классу и без дополнительных упрощений. NPC в них либо отсутствуют вообще, либо не нуждаются в оптимизации, благодаря максимально простой и часто детерминированной логике поведения.
* Title – сиквелы, приквелы либо воспроизведения знаменитых фильмов, биографий, и прочих медиапродуктов. Ставка делается на соответствие персонажей живым актерам, ролям, ожиданиям пользователей. Поэтому возможности персонажей ограничены теми, что продемонстрированы персонажами оригинала. Улучшение NPC малоприменимо, поскольку от NPC здесь вряд ли требуется самостоятельность.
* Indie (класс В) – экспериментальные либо некоммерческие игры, отличающиеся малым бюджетом и относительной простотой. Но порой могут содержать нестандартные решения, которые могут привлечь множество игроков (например, симулятор бога Godville). Персонажи могут быть неожиданно сложными для такого класса игры. Поэтому, метод Action First отлично подходит для таких игр.
* стандартный продукт (класс А) – большая часть коммерческих игр, разрабатываемых в стандартных условиях. Такие игры часто разрабатываются с помощью некоторых игровых движков (таких как Unity3D, Unreal Engine, и т.д.), поэтому предоставляют массу возможностей для модификаций и улучшений, в том числе и с помощью Action First. Однако, возникает необходимость адаптации любого алгоритма к применяемому движку (либо есть потребность в cross-engine решении, пригодном к использованию на любом движке).
* Блокбастер (класс ААА) – игры с очень высокой стоимостью и сложностью. Игры ААА-класса обычно имеют свою вселенную, свою фан-зону, обильно модифицируются игроками и обладают высокой степенью свободы действий и новизной. Большая часть «игр будущего», в том числе игр для VR[9], относятся к играм ААА-класса. Благодаря наличию вселенной и фанатской аудитории, персонажи таких игр обладают характером и образом, который им придает сама игра и игроки. Характер и образ являются абстрактными, в основном творческими чертами персонажей, поэтому разработка NPC в данном контексте – наравне техническая и творческая задача. Благодаря этому, внедрение Action First в такие игры может принести массу дополнительных эффектов, как положительных, так и отрицательных. Вдобавок, появляется необходимость модифицируемости любых дополнительных ресурсов, в том числе и для реализации Action First.
* Легенда (класс ААА+) – игра, обладающая легендарными показателями (огромным открытым миром, широким или почти неограниченным выбором игрока, огромным количеством персонажей, геополитикой, сочетанием всех жанров в одном). Все такие игры задают эталон, по которому в дальнейшем развивается индустрия развлечений, наряду с самыми знаменитыми фильмами и саундтреками.

Таким образом, понятно, что жанр и класс игры необходимы для формулирования задач, которые должны быть посильны NPC. Видно, что задачи эти весьма разнообразны, и подойти к ним с помощью единственной «универсальной» программы невозможно технически. Однако, заметно, что основная характеристика данных задач – оптимизированное содействие или противодействие решениям игрока и друг друга. Таким образом, существуют **два** следующих варианта оптимальных действий NPC:

* *Содействие*. NPC действует, исходя из наиболее оптимальной *поддержки действий игрока* в каждый момент времени.
* *Противодействие*. NPC выбирает действия, которые являются *максимально неудобными для игрока* в каждый момент времени.

Вышеописанные два варианта являются частными формулировками задачи оптимизации поведения NPC: **оптимизация поведения NPC есть возможность NPC в каждый момент времени выполнять действия, максимально содействующие или противодействующие действиям игрока**. Оптимизировать поведение NPC значит разработать механизм, позволяющий NPC выбирать действие, **оптимально подходящее** к его назначению, игровой ситуации и времени, либо **ранжировать** таковые для реализации долгосрочных планов NPC.

Данная задача является задачей концептуального уровня, поскольку относится не к реализации самих действий, а к их **организации**. Таким образом, метод Action First предлагает концепцию организации действий NPC, а не функциональную модификацию самих его действий.

## Актуальные решения в разработке и оптимизации NPC

Рассмотрим, каким образом осуществляется реализация поставленной задачи на практике.

### Общие концепции разработки NPC

Общая последовательность действий в разработке NPC состоит из следующих шагов:

* проектирование, на этапе которого осуществляется определение роли, значимости, спектра действий NPC;
* формулировка и реализация правил, где осуществляется разработка правил как элемента интерактивности игры, в частности, разрабатываются правила «причина-следствие», являющиеся парой взаимодействия объектов игры [9];
* применение стилистики игры к персонажу, которая заключается в реализации его внешнего вида, эстетикой, поведенческими механизмами.

Оптимизация поведения, таким образом, относится к двум последним шагам.

Возникает серьёзный и интересный вопрос, каким образом определять сущности и взаимосвязи NPC в играх. Поэтому существует множество концепций, которые представляют игру или как систему, или как совокупность отдельных связей между персонажами.

* Системный подход [10]. Игра рассматривается как система, а персонажи – как элементы системы. Сеть взаимодействий персонажей друг с другом составляет отдельный *контекст* игры, который связан по определенным законам с остальными областями игры.
* Ориентированный на игрока [11]. Игрок и его пожелания ставятся на первое место. Таким образом, вся система правил поведения персонажей строится относительно интерактивности, возможной для игрока. Такой подход распространен в шутерах: если есть игрок – есть и противники, которые должны в него стрелять, и союзники, которые должны помогать игроку либо сами по себе стрелять в противников игрока.
* Ориентированность на персонажей [12]. Такой подход довольно распространен в играх для многих игроков и для игр, ориентированных на сюжет. В нем персонажи (в основном люди) могут задавать правила, а NPC следуют по собственной, не связанной часто с правилами геймплея, сюжетной линии.
* Ориентированность на мир [13]. Если в игре есть вселенная, она может функционировать **независимо** от игроков-людей. При данном подходе, правила ставятся выше игрока, и для того, чтобы не проиграть компьютерным оппонентам, игрок должен соблюдать эти правила. Данный подход распространен в геймдизайне RPG, так как в них игрок далеко не всегда имеет ключевую для игрового мира роль.
* MDA (Models – Developing – Aesthetics). Данный подход имеет четыре (в некоторых источниках – три [14]) повторяющихся итеративно этапа: разработка модели, реализация этой модели, реализация творческой составляющей персонажа и сбор обратной связи от игроков. При разработке современных игр действительно применяется в основном итеративный подход, так как он позволяет постоянно повышать качество продукта, в том числе те параметры, которые нельзя измерить количественно (например, эстетика игры, стиль, удовлетворенность целевой аудитории, и т.д.)

### Использование Game AI[[1]](#footnote-1) при программировании NPC

Игровой интеллект (Game Artificial Intelligence, Game AI) является реализацией концепции искусственного интеллекта в игре. Существует ряд отличий концепции Game AI от концепции искусственного интеллекта, расширяющие стандартное понимание ИИ применительно к играм [15].

В разных играх применяются разные способы регуляции поведения NPC. Существуют, например, довольно распространенные скрипты с механикой if/else [16], которая применяется в реализации мобов либо глобальных компьютерных оппонентов игрока в casual-играх [17].

В большей части современных игр NPC управляются игровым интеллектом, который может варьироваться по сложности от решения отдельных задач игры (game challenges) до симуляции человекоподобного принятия решения (с известной степенью точности, поскольку любая такая симуляция создаёт непредсказуемость принятия решения и ответной реакции игрока).

Эффективность применения Game AI зависит от двух факторов: интеграции с геймдизайн-составляющей игры (то есть, цельность правил, эстетики, роли персонажа и его поведения) и адекватности с точки зрения игроков-людей (насколько поведение персонажа соответствует реальной игровой ситуации с точки зрения человека – в формализованном виде это соответствует сравнению эталонного механизма принятия решений с его имитацией). В попытке оптимизации этих факторов геймдизайнеры и ученые в области искусственного интеллекта создали множество методик:

* *PEM (Player Experience Model)*. Основывается на сборе опыта от игроков относительно применения той или иной модели поведения NPC, и поведение игрока по этой модели является **определяющим** фактором для поведения NPC [18]. Суть заключается в трёх этапах:
* сбор сведений о применяемых игроками стратегиях;
* выработка или поиск стратегий упреждения;
* внедрение стратегий упреждения в NPC.

Благодаря тому, что изобретение разнообразных стратегий прохождения игры является естественным процессом для любого игрока, а число предлагаемых PEM модификаций NPC столь же велико, PEM является итеративным подходом. Его алгоритм можно модифицировать постоянно, по мере возникновения ультимативных стратегий действий.

Анализ поведения пользователя может быть и автоматизированным (то есть, игровой движок сам подстраивается под действия игрока), и непосредственным (от обратной связи, плейтестов, бета-тестирования, etc.). Поведение игрока может анализироваться по-разному: и как реакция на геймплей, и как реакция на персонажей, и как психологический фактор. Так, одно из направлений PEM – анализ поведения пользователя относительно его интерактивности в игре (нажатие клавиш, действия, количество набранных очков, «трудные места», и т.д.) [19].

PEM может иметь разную ориентацию в зависимости от типа игры [20] – на самого игрока (subjective), на окружение/персонажей (objective), на интерактивность (gameplay-based) – но в итоге возможно построение «модели интерактивности игрока», которую уже можно формализовать и выстроить аналогичную модель поведения NPC.

* *PCG (Procedural Content Generation)*. Данное направление популярно в некоторых видах и жанрах игр (например, в MMORPG, 2D, стратегиях), и актуально, если возможно генерировать персонажей и прочий контент вместо левел-дизайна (то есть, вместо ручного заполнения игры контентом) [20]. Game AI персонажей здесь может генерироваться с учетом паттернов, изначально заданных в игре, и некоторых глобальных параметров, например, уровень агрессивности, начальное отношение к игроку, и т.д. На базе этой методики создан фреймворк EDPCG (Experience-Driven Procedural Content Generation) [21], который использует материалы PEM, чтобы генерировать контент, основанный на опыте использования контента игроками. Таким образом, PCG использует актуальные вопросы опыта игроков в создании контента на стыке Game AI, процедурной генерации мира и психологии.
* *Анализ данных ИИ*. Данная методика имеет много общего с методикой PEM, так как тоже подвергает анализу данные об игре (т.н. «метрика игры», *game metrics*) [22]. Сейчас методика особенно актуальна, так как с помощью методов искусственного интеллекта можно внедрять формализованные методы (например, из области Big Data и Machine Learning) в саму игру. Данный подход применяется в стратегиях и PRG, поскольку действия персонажей в них действительно могут основываться на опыте и знаниях персонажа о действиях игрока и друг друга. В самом анализе применяются и количественные, и качественные (семантические) показатели, что соответствует балансу количественных и качественных параметров в игре и (что ещё более важно) – балансу учета как формализованных, так и логико-семантических показателей персонажа.
* *Персонализированные модели NPC*. Здесь могут применяться самые разные алгоритмы из области AI Behavior, чтобы расширить возможности NPC. Самый главный и пока не решенный окончательно вопрос методики состоит в **конкурентоспособности NPC относительно игрока**. Отчасти его можно решить посредством внедрения социальных моделей поведения NPC, отчасти – с помощью внедрения нейросетей, как в шахматных ИИ (попытки машинного обучения в полноценных компьютерных играх происходят и сейчас [23] [24] [25], однако результаты нельзя считать за общее решение данного вопроса). Некоторые алгоритмы, например, предлагаемый в диссертации метод Action First, регулируют определенные стороны NPC, такие как *действия*.

Цель методики – наиболее правдоподобная симуляция человеческой деятельности компьютерным персонажем. Эта цель в итоге никогда не достигается, так как человек может принимать *любые* решения в игре недоступными для машины методами, а поведение NPC всегда регулируется правилами, изначально спроектированными и реализованными человеком. Однако в зависимости от алгоритмов регуляции поведения и от требований к самой игре возможно максимально приблизиться к правдоподобной имитации человеческого поведения в игре. Таким образом, максимально правдоподобная имитация корректного поведения в игре и есть цель любого разрабатываемого метода или фреймворка оптимизации поведения, в том числе и Action First.

Таким образом, существуют методики создания NPC на концептуальном уровне. Здесь же существует задача учета семантических и неформальных требований к NPC, **частично** решаемая с помощью фреймворков PEM и MDA. Однако, данная задача более широка (так как относится к фундаментальной проблеме математического решения np-трудных задач, а именно – к трудности формализации слабоформализуемых факторов явления) [26]. В разработке игр существуют её частные решения, обычно с помощью имитации эталона. Они реализуются на функциональном уровне разработки NPC, в качестве замещающих скриптов. Здесь и находится причина, по которой предпочитают создавать детерминированный функционал скрипта вместо динамического алгоритма имитации действий. Метод Action First позволяет избежать данного подхода, предлагая динамическую замену скриптов реакциями на события.

## Функциональные основы поведения NPC

Существует множество способов улучшить показатели и возможности поведения NPC. Разработчик обычно имеет свободу в выборе способа, учитывая достоинства и недостатки предлагаемых алгоритмов. Рассмотрим несколько таких способов из разных областей знаний.

В широком смысле, функциональные методы оптимизации поведения NPC разделяются по принципу действия в рамках следующих подходов:

* системный подход, в котором игра представляется в качестве системы взаимосвязанных действий и событий;
* программный подход, в котором законы игры описаны статическими схемами-правилами (показано на примере скриптов в предыдущем подразделе);
* имитационный подход, в котором представление семантической информации о персонажах упрощается и переводится (по некоторому алгоритму) в численные значения;
* функциональный подход, в котором поведение персонажей регулируется с помощью учета опыта игроков и планирования поведения NPC разработчиками (PEM);
* прикладной подход, в котором используются математические модели различных предметных областей, связанных с тематикой и сеттингом игры.

Рассмотрим концепции этих подходов подробнее.

### Системный подход в играх

В классическом представлении методов оптимизации с использованием системного подхода игра является совокупностью персонажей, событий и объектов, взаимодействующих по определённым закономерностям.

Таким образом, игра представляется как система S, представленная совокупностью

S={C,E,O},

где С – совокупность персонажей,

Е – совокупность событий игры,

О – совокупность объектов.

Связи в системном подходе к играм обозначаются как пары С→O, C→E, E→C, и т.д. При этом суть связей заключается в отношениях «следует» («персонаж взаимодействует с объектом», «персонаж реагирует на событие», и т.д.). Разнонаправленные связи не тождественны, т.е. E→C ≠ C→E. Могут существовать сколь угодно большие связи по типу С1→С2→С3…→Сn, при этом они разделяются на элементарные связи, а эффекты для персонажей и объектов игры рассчитываются для каждого их участника [27].

Данный подход учитывает всё происходящее в игре максимально комплексно, поскольку в методах оптимизации по системному подходу учитываются **сразу все** персонажи и объекты игры. Поэтому, можно в любое время получить управление над любым событием, персонажем и объектом игры.

Однако, чем больше объектов, тем больше связей между ними присутствует в обрабатываемой системе. Учитывая обилие разнообразных событий с персонажами и объектами в играх, становится сложным поддерживать и реализовывать весь их функционал централизованно. Также, чем больше становятся цепочки взаимодействия между персонажами, тем сложнее персонажи и тем сложнее правильно создать и отследить их поведение. Учитывая тот факт, что между каждым объектом и персонажем игры может существовать связь (а на практике количество таких связей ограничено только желанием разработчика их создавать), налицо «проклятие размерности», вынуждающее все методы в рамках системного подхода работать либо в рамках обработки локальных событий и связей (механизм обработки NPC в Unity [28]), либо обработки по упрощённым схемам (Clausewitz Engine [29]).

Данное правило справедливо и для относительно простых игр: например, игра Super Mario Classic Edition, у которой существует только 6 действий игрока, 4 типа врагов и 14 типов объектов, имеет астрономическое количество вариантов взаимодействия между ними (например, метод А\* для выбора вероятного пути Марио через уровень средней локации игры требует в общем случае перебора 10484 вариантов [28]. Ещё сложнее рассчитать все варианты действий Марио в его прохождении локации, учитывая сотни объектов, с которыми он взаимодействует на локации).

Так, благодаря этому, невозможно предсказать ни конкретные действия игрока в следующем моменте времени игры, ни ответ NPC на эти действия (можно лишь выдвинуть эмпирическое предположение, далеко не всегда верное и исходящее из экспертного опыта наблюдения за действиями NPC в этой игре) [29].

Для того, чтобы уменьшить влияние «проклятия размерности» на продолжительность и сложность вычислений реакций персонажей, применяется практика разделения доменов действий на поддомены (sub-domains) применительно к NPC. Она реализована во множестве игровых движков (unity 3d, Unreal, Cry, etc.), поскольку сами движки реализуют принцип упрощения системы игры и работу человека над отдельными доменами действий NPC.Естественно, пересчет варианта для каждого случая сложен, и потому возникает потребность оптимизации домена вариантов и (как вариант) сохранения примеров успешного выбора в базе знаний для использования в аналогичных ситуациях.

Итак, следует редуцировать влияние «проклятия размерности», сохраняя привязанность NPC к домену правил и настраиваемость персонажа. Это можно реализовать посредством машинного обучения, в частности посредством метода MDP.

MDP (Markov Decision Process) – алгоритм, суть которого в декомпозиции одного исходного домена действий (например, у того же Марио) на несколько доменов, каждый из которых содержит несколько похожих действий. Например, для Марио это могут быть домен движения (Right, Fast Right, Jump, Fast Jump), домен ухода от врагов (действия, позволяющие избежать врага, ловушки или пропасти), домен сбора монет (все действия для их выполнения, в том числе и А\*, выстраивающий кратчайший путь до ближайшей монетки), и т.д. Преимущество такого разделения – **частное решение**, которое принимается исходя из возможности/невозможности тех или иных действий в данный момент. Если в каждом домене принято частное решение, для каждого домена область вариантов итогового решения гораздо меньше, чем исходная, а итоговое действие выполняется после выбора из всех действий, предложенных как частное решение. Согласно исследованию [30], домен вариантов для Марио при таком подходе сократился до 3\*219 [28], рассчитать решение по нему намного проще, чем в исходном варианте. Процесс такого выбора представлен на рисунке 1.4.

Выбор действия в частных доменах может происходить по разной логике, например, случайным образом, или комбинацией экспертных суждений игроков о ситуации из БЗ вместе с долей случайного выбора.

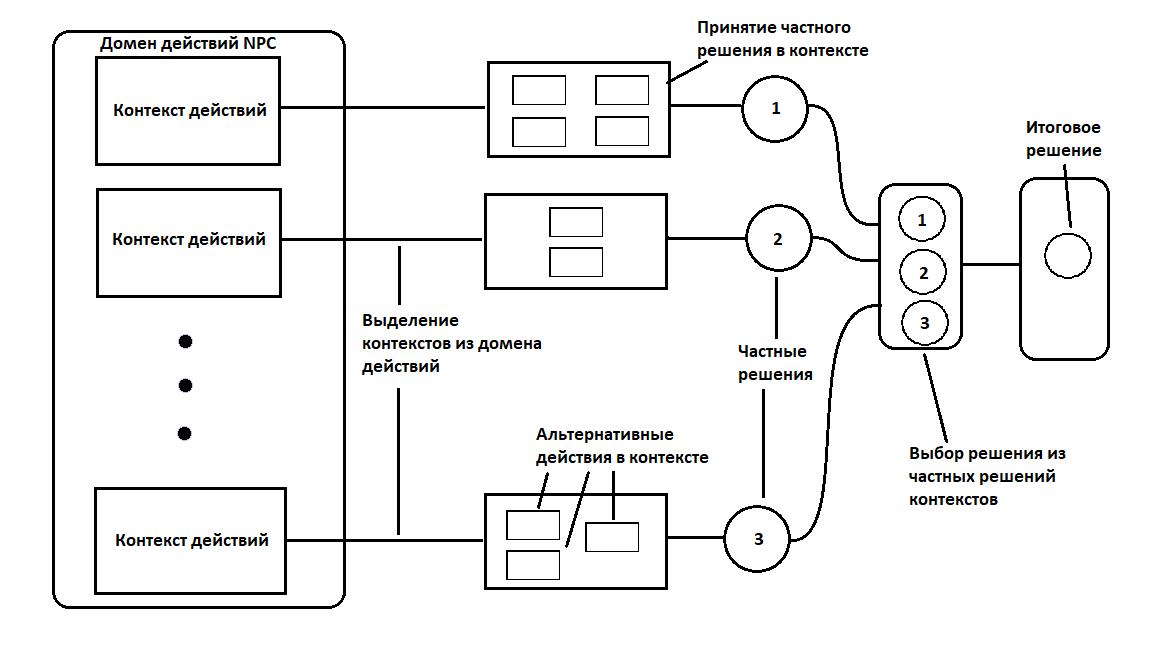


Рисунок 1.4 – Процесс выбора действия по методу MDP.

Данный вопрос рассматривается в частном подходе к выбору. Кроме того, итоговый выбор действия работает по-разному в зависимости от того, возможно ли для NPC выполнение нескольких параллельных действий (теоретически возможно, что и всех предложенных). Подобные случаи возникают в стратегиях, когда NPC **практически всегда** выполняет одновременные действия из разных областей. В случае параллелизма принятия решений, можно сократить домен действий не только за счёт частного решения, но и за счёт отнесения действий, относящихся к разным сферам взаимодействия персонажа, к разным поддоменам.

В методе Action First, разработанном в диссертации, такие поддомены во избежание путаницы с доменами действий именуются контекстами. Кроме того, контексты подразумевают связь с правилами и объектами, а не только действия, как в MDP.

### Семантическая имитация поведения персонажей

Серьезный недостаток многих существующих алгоритмов NPC – учет только количественных факторов, хотя в сути игры, как и в сути человеческих решений в играх, есть и качественные (семантические) факторы, и иррациональные факторы, которые на данном этапе развития технологий возможно только заменить эквивалентными имитационными параметрами. [31].

Семантические параметры относятся к слабоформализуемым величинам (которым необходимо ставить в соответствие определённые формализованные величины, которые и будут в итоге участвовать в расчётах), однако ряд методов позволяет их описать для применения в расчётах. На этом и базируются имитационные методы оптимизации поведения NPC.

Также существует группа методов, основанных на фреймворке Player Experience Model. В них используются не идеи имитационных методов, а модели экспертных систем: для того, чтобы семантические действия NPC были учтены с правильным значением либо весом, используются величины, сформированные разработчиками на основании своих гипотез о поведении NPC и экспертных данных от игроков.

**1.3.1.1** Семантика (в данном случае под этим понятием подразумевается совокупность *иррациональных и качественных критериев* персонажей) может применяться в игре на следующих уровнях:

* *Уровень объектов*. На этом уровне прорабатываются критерии отдельных персонажей как **независимых** объектов. Один из распространенных подходов к семантике на этом уровне – разработанные Калманном и Талманном [32] «смарт-объекты». Эти объекты содержат не только технические характеристики, но и определенные методы, позволяющие предусмотреть процесс всей интерактивности с ними. Обычная коробка в игре может, допустим, содержать информацию о том, кто может его поднимать (и каким уровнем должен обладать), где персонаж должен находится, пока его поднимает и несет, и т.д. Применительно к персонажу так регулируется его внешний вид, анимация, действия, которые персонаж выполняет из-за внешних воздействий или самопроизвольно.

Другой интересный подход на этом уровне – подход «словарей» [33]. Здесь объекты игры имеют свою запись в структуре «словаря» (это может быть, например, база данных), где записывается вся (общая) информация об объекте или персонаже. Данная технология нашла продолжение в технологии шаблонов (templates) во многих распространенных движках, таких как Unreal Engine и Unity. Переход к шаблонам обусловлен попыткой разрешения проблемы семантического описания игровых объектов [34], которая отчасти упрощает левел-дизайн и реализацию (отчасти, поскольку не всегда такая технология является выгодной при проектировании NPC).

* *Уровень отношений между объектами*. Здесь организуется семантика **парных** отношений между объектами или персонажами. Так, по подходу Левиссона [35], можно организовать данные связи иерархически, но тем самым столкнуться с трудностями ранжирования отношений и соотношения между равнозначными связями (например, связи «строитель несет кирпич» и «кирпич, несомый строителем» в данном подходе равнозначны – для разграничения иерархии необходима какая-то дополнительная связь, подключающая дополнительные объекты для описания явления).

Возможна также организация связей, основанных на семантических отношениях, например, отношения «владеет», «вызывает [следует]», «содержит» - данный подход предложен Ганнсом и Сингх [36]. Их подход интересен тем, что такие семантические отношения можно как угодно гибко интерпретировать технически (например, отношение «содержит» в программировании можно реализовать как класс и его поле.

* *Глобальный уровень игрового мира*. Здесь используются семантические параметры, затрагивающие весь игровой мир либо его локацию. Большую роль играют симуляции реальных объектов (например, симуляция погодных условий [37]), а также глобальные параметры, влияющие на все объекты локации. Такие методы имеют особое значение в стратегиях и симуляторах, базирующихся на реальных событиях и реальных моделях поведения (например, исторические стратегии, симуляторы города, симуляторы погоды, и т.д.)

**1.3.1.2** Совместное использование семантических и технических параметров персонажа затруднено разными рамками ограничений для параметров. Например, реакция персонажа на действия другого, как семантический критерий, должна регулироваться на основании «жизненного опыта» решения в таких же ситуациях, но домен возможных действий персонажа часто может не предоставить подходящего действия. Возникает проблема выработки решения, базирующегося на предоставленной информации и существующих семантических и математических ограничениях.

Для решения этой проблемы применяют различные методы, классификация которых представлена на рисунке 1.5.

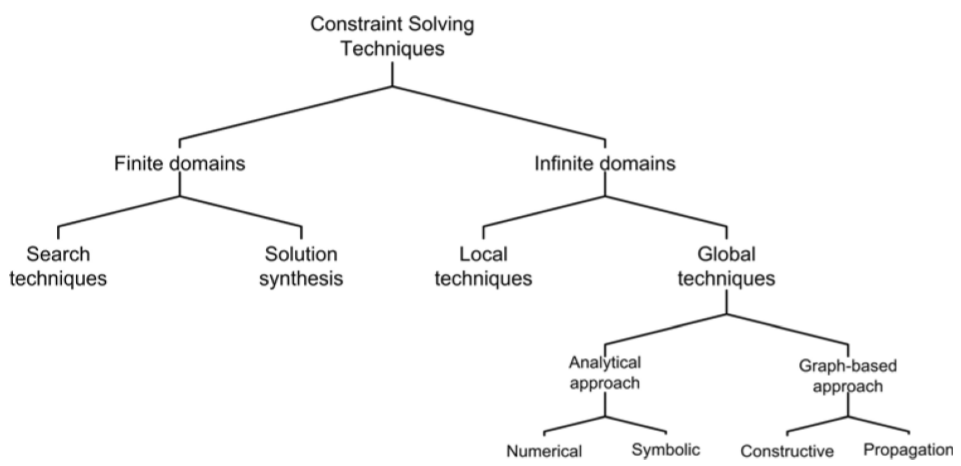


Рисунок 1.5 – Методы выработки решения.

Можно видеть, что главные ветви классификации базируются на свойствах доменов ограничений и объектов (то есть, на совокупностях вообще всех принятых к рассмотрению объектов, персонажей и т.д.). Природа игры со свободным принятием решений NPC имеет неограниченный домен вероятных решений NPC, а потому требует или локальных техник реализации (скрипты и имитационные модели) либо более углубленные методы, предлагающие сложную формализацию и обработку явлений в игре.

Количество вероятных решений персонажа, принятых на основе имеющихся в игре ограничений и действий, измеряется астрономическими числами [38]. Зато домены шаблонов объектов и правил игры гораздо удобнее применять системно. Поэтому, основываясь на конечных доменах, можно применять методы PEM и поисковые техники (относящиеся к области поисковых алгоритмов и Big Data). Относительно регуляции решений в больших доменах можно применять сложные аналитические методы из областей экспертных систем и машинного обучения, однако для упрощения часто используются прикладные математические модели, например модели теории графов для задач нахождения пути NPC (pathfinding; является крайне распространённой задачей для NPC в подавляющем большинстве игр). Например, алгоритм A\* и алгоритм Дийкстры весьма часто применяются [39] для решения проблемы поиска пути персонажем в изменяющейся обстановке на локации.

### Схемы генерирования объектов и поведения персонажей

На этапе левел-дизайна локации семантика применяется для автоматизации добавления контента в игру. Здесь важно не только добавить сам контент, но и его смысл (персонажи должны быть на корректных позициях, деревья и горы не создают недоступных персонажам пространств, и т.д.). Если генерируются новые NPC, необходимо настроить их поведение оптимальным образом.

Для этого могут использоваться системы CAPS (Content Automatic Placement System) [36]. На данный момент существует много таких систем, от созданных вручную для конкретных игр до предусмотренных в игровом движке. Общий их принцип – выборка из домена объектов самих объектов и настройка их в соответствии с правилами из домена правил.

Осуществляется также ведение базы объектов и поддержка шаблонов. Объекты в базе в таком случае могут быть связаны различными семантическими связями (вроде отношений в подходе Ганнса-Сингх). Эти привязки сохраняются (или проверяются) и при расстановке объектов на локации.

Особо распространены данные системы в играх-платформерах и играх с относительно небольшим доменом объектов, как, например, в Binding of Isaac. Действие системы CAPS в данной игре представлено на рисунке 1.7.



Рисунок 1.6 – Сгенерированная локация по CAPS в The binding of Isaac.

Также распространено применение семантики в подходе PСG. Обычно этот подход применяется тогда, когда игровые локации и персонажи формируются в игре динамически, а не предусмотрены разработчиками. Пример применения семантических правил – игра Left for Dead: персонажи (зомби) генерируются отчасти случайным образом в комнатах и на улицах, с учетом зависимости от уровня здоровья и экипировки игроков (а также в зависимости от количества игроков-людей). Правила создания персонажей здесь регулируются одним менеджером персонажей (в самой игре объект называется «режиссер» [37]). Данный подход дает определенную свободу в стиле игры и выборе решений самим игроком, а также уменьшает время работ над игрой.

### Некоторые прикладные подходы к поведению NPC

Для оптимизации поведения NPC могут применяться математические и концептуальные модели. Они настолько разнообразны, что нет смысла описывать их все. Рассмотрим здесь несколько методов решения прикладных проблем, связанных с оптимизацией поведения NPC при применении в вышеописанных подходах.

**1.3.1.4** Достаточно типичная и уже упомянутая ранее задача, для которой выгодно применять математические модели NPC – задача нахождения пути персонажем (Pathfinding). Задача заключается в нахождении кратчайшего пути между точками А и В на локации, учитывая все условия локации и оптимальность на каждом шаге.

Учитывая постановку задачи, проблема поиска пути может быть интерпретирована как задача теории графов о кратчайшем пути или как задача динамического программирования с переменными коэффициентами элементарных участков этого пути между частями игровой локации. В актуальных на данное время методах решения наработки по этим областям объединены в алгоритм A\*.

Основное решение данной задачи в игре – нахождение *условно* кратчайшего пути с учетом всех статических и динамических условий и состояний на локации. Частный случай работы А\* - задача о минимальном покрытии графа [34], благодаря решению которой возможно построить сетку путей на локации (например, в Unity3d). Частный случай данной сетки – цепь точек пути (waypoints), которые создаются как решение задачи о кратчайшем пути и являются детерминированной инструкцией для перемещения NPC по ним. Такая сетка является технической реализацией А\*, поэтому вполне решаема математически.

Однако, у алгоритма A\* существует ряд проблем. Явная проблема – «проклятие размерности», значительно повышающее сложность и время решения в зависимости от размеров локации и количества условий, в которых двигается NPC. Особенно проблема актуальна для Real-Time игр с большим доменом объектов и правил (так как они в задаче поиска пути – условия) и с большими локациями (чем больше возможных точек перемещения – тем больше путей и тем медленнее будет работать алгоритм).

Конечно, есть методы разработки, позволяющие нивелировать данный недостаток: ограничения сетки путей NPC (как в серии игр Dead Space – все NPC имеют свое поле действия и никогда не выходят за его пределы), ручное задание или редактирование сетки при разработке игры, задание явных приоритетов движения (ручное задание сетки waypoints[36]), триангуляция сетки для оптимизации возможных точек перемещения, создание случайных векторов движения (в платформерах и 2D-играх это довольно частый приём), и др. Однако, они могут повлечь странные с точки зрения логики решения NPC по поиску пути (поскольку путь, определенный математически, не всегда означает семантически оптимальный путь). Получается, даже математически оптимальное решение по А\* не гарантирует, что NPC будет двигаться в соответствии с его правильной реакцией на действия игрока или других NPC.

Также, если учесть, что ситуация в играх динамически меняется в условиях неопределенности, возникают дополнительные сложности. Условия неопределенности, когда веса переходов между точками перемещения неизвестны, сводит задачу поиска кратчайшего пути к вычислению кратчайшего пути по количеству точек между заданными A и B, либо вообще делает данную задачу решаемой только с определенной долей вероятности.

Задача динамического программирования подразумевает при этом расчёты, которые будут учитывать всё множество точек перемещения – а если пространство имеет больше, чем три меры (а в играх такое довольно часто происходит) – алгоритмическая сложность задачи динамического программирования не может это реализовать достаточно быстро. Таким образом, данная проблема сохраняется и при таком методе решения.

Семантический подход подразумевает поиск пути персонажем исходя не из *всех возможных* точек перемещения (как в А\*), а исходя из правил игры, коррелируемых с текущими действиями персонажа.

Подход Л. ван Драйл и Р.Бидарра [37] предлагает семантический **доменно-независимый** метод к решению задачи пути. Они выделяют следующие первопричины недостатков алгоритма A\* и графоориентированных методов:

* понимание правил игры и логик персонажей в представлении разработчика или геймдизайнера недоступно *в том же виде* для игрового интеллекта (критерий правдоподобия для разработчика отличается от фактического, поскольку разработчик предполагает человекоподобное поведение NPC, а для самого NPC актуально лишь математически вычисленное правдоподобие – возможно, как оценка его действия);
* *доменнозависимость* всех объектов и правил игры вызывает ограничения в решениях и действиях ИИ, которые противоречат здравому смыслу (то, что выверено математически, не всегда правильно с человеческой точки зрения).

Посему в подходе переходят от метрической навигации (от точки в точке) к семантической (попасть **в область**, где происходит желаемое для персонажа действие). В частности, вместо последовательности шагов персонажа от А до В предлагается ввести понятие *областей*, в которых вообще возможно нахождение данного персонажа. Точки в областях по отдельности не рассматриваются, так как определение конкретного положения можно поручить микрологике персонажа как смарт-объекта (либо имитировать путём придания пути дополнительного параметра «статистического отклонения» в пределах вычисленной области). Подобная микрологика может быть реализована любым другим способом, в том числе и А\*, однако задача о нахождении пути реализуется не в рамках движения из-за этой реализации, а только в рамках областей.

Релевантность областей определяется посредством ввода понятия *контекстов*. Контекст объединяет *подходящие под данную ситуацию* области (домены), из которых принимается решение. Чаще всего при решении задачи поиска пути выделяются следующие контексты:

* *контекст пути-пространства*, который объединяет понятия путевой метрики (точки и пути между ними, геометрия объектов на локации);
* *контекст возможности*, представляющий совокупность правил, предоставляющих или запрещающих возможность перехода персонажа в ту или иную область, например область видимости врага, или наличие пропасти/ловушки в данном месте.
* *контекст места и метрики*, определяющий форму места, показатели места для персонажа, его приоритет, и т.д.

Используя контексты как объединения семантических и метрических критериев, возможно перейти от алгоритма к базированному на Game AI компоненту решений персонажа, а от руководства метрикой при расчете к учету этой метрики. Естественный эффект данного подхода – становится невозможным прямо спроектировать путь персонажа по локации, что ограничивает применение подхода там, где точно знать путь необходимо. Также, «проклятие размерности» все равно влияет на быстродействие (хотя и меньше, чем в А\*), так как широкие концепты из-за больших доменов все равно будут присутствовать, и это все равно затруднит выбор Game AI.

Данный подход применён в играх серии Dead Space. Все NPC имеют ограниченную область действий, за которую они не заходят. Поэтому, можно точно сказать, в какие области могут перемещаться враги игрока – но предсказать, что конкретно они будут делать в этих областях или в какие конкретно координаты локации они переместятся, никак нельзя.

Применение «метода областей» – лишь локальная задача, призванная объяснить действие подхода. Однако, такой подход можно применить для любой другой задачи в рамках поведения NPC. Он показывает, что важно разделение домена действий или решений на области (в Action First они также называются контексты, поскольку используется схожая с подходом Драйл-Бидарра терминология), а затем – решение локальной задачи в пределах области.

У метода областей есть следующий серьезный недостаток: условия и объекты действий NPC в областях определяется только разработчиком. Механизм отбора пути в области также должен программироваться в реализации конкретной логики поведения персонажей, то есть для любой модификации поведения NPC придётся переработать **весь** его функционал. В методе Action First предлагается возможность исправления данного недостатка посредством введения приоритетов для действий.

### Роль понятий теории информации в программировании NPC

При подходе к проблеме проектирования NPC из любой предметной области ясна задача: **приближение поведенческих алгоритмов к человекоподобному в любой игровой ситуации**.

Однако, возникает следующая связанная задача обработки информации об игровых объектах. Для того, чтобы учитывать логико-семантическую структуру объектов игры (последовательности действий, стратегии ИИ, и т.д.) необходимо их сопоставить с их численными характеристиками. Для этого возможно использовать теорию информации.

В теории информации меня интересует раздел прикладной теории информации, в связи с необходимостью решения следующей задачи:

*Минимизировать различие между ожидаемой («человекоподобной») реакцией персонажа на ситуацию и его действительным действием, совершаемым в каждый момент времени.*

Поэтому, мой метод Action First:

* *оперирует действиями NPC*, поскольку они являются непосредственным отражением **поведения** персонажа;
* *сопоставляет действия с их ЧХ,* именуемыми в подходе приоритетами;
* играет большую роль дивергенция Кульбака-Лейблера, поскольку она является мерой *минимизации* различий между ожидаемым действием на данном временном такте и предпринимаемым.

Система действий в игре, согласно данному подходу, является растянутой по времени (представленному совокупностью элементарных промежутков времени T). Допускается, что в каждый момент времени ti может происходить действие xkj (из домена действий персонажа Xk, в котором J действий). В системе одновременно действуют K персонажей.

Непрерывное время в подходе Action First не рассматривается, поскольку для расчета каждой игровой ситуации используется ограниченный (элементарный) промежуток времени. Данное утверждение справедливо даже для Real Time- игры (так как связано с самой механикой расчетов событий в игре), поэтому в дальнейшем все расчеты будут произведены по формулам, учитывающим дискретность, а не непрерывность временного параметра.

Можно сказать, что в каждый момент времени t осуществляется выбор действия xkj из множества Хk с определенной вероятностью (определяемой не с помощью зависимости, а на основании статистических данных). На практике, поскольку действия выполняются в определенных условиях, можно говорить о трендовой модели вероятности выполнения данного действия (то есть, видя определенные условия в игровой ситуации, можно предсказать по ним происхождение того или иного действия).

Возникает вопрос: не проще ли учитывать само распределение явлений в игре (если мы говорим о периодически возникающих событиях или выполняющихся действий) вместо трендовой модели? Дело в том, что в играх присутствуют бесконечно разнообразные действия и события, с разной и часто непредсказуемой природой зависимости вероятности их появления от времени. Поэтому нельзя выработать для них какую-то единую схему учета формул распределения, которая могла бы применяться единообразно в каждом конкретном случае. Поэтому в нашем случае лучше описывать действие не формулой его распределения по времени, а его численной характеристикой.

Со статистической точки зрения закономерность происхождения действия xkj по времени является случайным процессом, и в каждом моменте времени происхождения данного процесса состояние выполнения xkj можно описать его численной характеристикой сkj. Природа и характер функции распределения не важен (потому что в каждый момент t нас интересует именно фактическое значение сkj , а не характер её изменения, который в динамически меняющихся условиях может быть непредсказуем на следующих временных тактах).

Учитывая постулат предыдущих пунктов о неопределенности происхождения действия во времени, ЧХ xkj можно описать парным сопоставлением значений (таблица 1):

Таблица 1

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xkj | xkj1 | xkj2 | xkj3 | xkj4 | xkj5 | xkj6 | xkj7 | xkj8 | xkj9 |
| сkj | сkj1 | сkj2 | сkj3 | сkj4 | сkj5 | сkj6 | сkj7 | сkj8 | сkj9 |

Таким образом, для совокупности состояний выполнения действия {xk} есть сопоставленная совокупность их ЧХ {ck}. Данная совокупность является численной характеристикой действия, которая именуется **приоритетом** действия, а её элементы в соответственных промежутках времени являются *значениями* приоритета в момент времени ti.

Приоритет является совокупной численной характеристикой, учитывающей комплексное влияние факторов игры на совершение действия. Именно эта природа позволяет каждому значению приоритета быть непредсказуемой: природа взаимозависимых процессов у других NPC может также варьироваться, а условия – постоянно меняться. Поэтому, механика расчёта приоритета также будет разной, не только в величине значений, но и в самой формуле расчёта могут происходить изменения.

Важно обратить внимание на природу xkj. Вследствие того, что происхождение xkj следует из целой совокупности факторов, как формализованных, так и семантических (имитированное поведение NPC, происхождение событий, величина здоровья или наносимого урона у персонажа, действие игрока, и т.д.) xkj *не является* полностью формализованным математическим понятием. Поэтому, во-первых, его нельзя рассматривать отдельно от его численной характеристики при расчётах (поскольку в таком случае будет учитываться вклад только для формализованных показателей NPC), а во-вторых, мера неопределенности, возникаемая в силу природы xkj, должна быть преобразована из прямого численного значения в меру количества информации (исключающую неопределенность) по формуле Шеннона[30]:

(1)

При этом компонент является составным. Совокупное состояние xkj описывается через сумму слагаемых факторов воздействия на персонажа со стороны объектов и иных зависимостей игры:

(2),

где – мультипликативный коэффициент влияния (если природа влияния на объект игры имеет вес или пропорцию относительно базовой величины параметра, он изменяется. Данный параметр не может принимать значения ниже 1, если существует);

xi – величина параметра персонажа, на который влияют факторы;

-аддитивный коэффициент влияния (изменяется, если используется линейная формула взаимодействия. Если существует, не может принимать значение ниже 1. Если используется только мультипликативная характеристика влияния, равен 0.

Элементарного приоритета недостаточно для комплексного учета характера действия. Дело в самой формуле расчёта факторов воздействия на персонажа в формуле (2): в разные моменты времени значения xi меняются. Также, нельзя судить о приоритете действий только по показателям персонажа, поскольку на решение о применении действия могут оказывать действия других NPC и игрока. Поэтому, для расчета приоритета для всего действия, а не отдельного показателя, необходима более совокупная характеристика. В методе под приоритетом здесь и далее именуется именно данный совокупный приоритет.

Учитывая неопределённость состояний параметров xi, необходимо избавиться от неё, перейдя к количеству информации от числа элементарного приоритета.

Таким образом, ЧХ заменяется на количество информации Ij, рассчитываемое для каждого такта. Значение приоритета как совокупной ЧХ Pkj, рассчитывается по формуле

(4)

Рассмотрим компоненты формулы (3) подробнее.

– коэффициент последовательности. Это число показывает, что действие является элементом последовательности, и данная последовательность определенным образом влияет на его приоритет. В случае, если действие не входит ни в одну последовательность на данном такте – значение коэффициента равно 1. может изменяться по времени – например, если действие станет элементом последовательности на каком-нибудь такте.

– базовое значение приоритета. Показывает, насколько приоритетно действие персонажа в глобальном смысле. То есть, насколько часто, по задумке разработчика, действие должно выполняться. Однако, базовое значение корректируется на каждом такте показателем количества информации:

(5),

где – значение количества информации ЧХ xkj для данного такта (вычисленное по формуле (1)).

– приоритеты действий, *связанных* с текущим и на данный момент присутствующих систем. Это тоже базовые части их приоритетов, с влиянием коэффициентов их последовательностей. Можно сказать, что выражение является совокупностью приоритетов, описывающих связи между всеми связанными действиями системы на данном такте. При этом, заранее неизвестно, сколько именно действий войдут в эту сумму, что обуславливает различное число слагаемых для формулы (2) в различных игровых ситуациях.

Интересно поведение коэффициента : по модулю он совпадает с коэффициентом последовательности своего действия, но по знаку зависит от того, является ли его действие *противодействием* для существующего: если да, знак меняется на отрицательный (поскольку наличие противодействия в системе снижает вероятность выполнения рассчитываемого действия по логическим соображениям). В ином случае знак остается положительным (поскольку наличие связи между действиями должно положительно влиять на приоритет его выполнения).

В формуле дивергенции используются следующие параметры:

и – значения приоритета на предыдущем такте и на существующем. рассчитывается по формуле:

(6),

то есть, фактически, это значение приоритета на такте без его аддитивной части.

δ - дельта-параметр приоритета. Отражает изменения значения приоритета на такте (возрастания или убывания). Его верхний индекс изменяется в зависимости от такта, к которому относится дельта-параметр: относится к **текущему** такту, - к **предыдущему**.

– значение дивергенции Кульбака-Лейблера, которое рассчитывается по формуле

(7),

где, для удобства записи, на заменено выражение , а на – выражение . Видно, что, таким образом, дивергенция Кульбака-Лейблера вычисляется как расстояние от текущего значения приоритета до ожидаемого по каждому действию xkj в данный момент времени.

Отмечу, что неравенство Гиббса в формуле (7) по-прежнему выполняется, то есть D(P||Q) ≠ D(Q||P) [31]. С одной стороны, это обоснованно логически: выполняемое действие P сравнивается с ожидаемым Q, которые, естественно, необязательно эквивалентны; замена местами Q и P в силу отсутствий гарантий эквивалентности, очевидно, не приведет к эквивалентному результату.

Здесь видно, что на такте эталоном для приоритета действия является приоритет того же действия, но в предыдущий такт. Таким образом, реализуется концепция стабильности приоритета, которая устанавливает пороги выполнения и отмены действий:

*Минимизация дивергенции говорит о том, что действие на любом такте игры имеет один и тот же приоритет и, следственно, статус исполнения. При стабильном приоритете оно* ***не может*** *менять своё положение и выполняться или отменяться персонажем. Однако, если приоритет растёт или убывает* ***выше оговоренных пределов*** *(называемых* ***порогом выполнения*** *и* ***порогом отмены****), действие будет выполнено или отменено соответственно.*

Отсюда возникает вопрос определения порогов выполнения и отмены. Достаточно широкие пределы снижают вероятность того, что значение приоритета выйдет за эти пределы, то есть не выполнится соотношение

(8),

а следовательно, что действие не изменит свой статус. А более узкие пределы и делают это более вероятным.

Таким образом, действие на данном такте сравнивается с собственным значением приоритета на предыдущем такте по дивергенции. Сохранять оба значения одновременно помогают дельта-параметры. Минимальное значение дивергенции из всех вычисленных у персонажа говорит о наибольшей «гладкости» изменения приоритета, и, следовательно, о стабильном выборе действия на исполнения, относительно положения значения и лимита исполнения.

Значения приоритета со «скачками» (в ту или иную сторону) должны быть либо немедленно выполнены (если для персонажа это возможно), либо отменены (поскольку скачки приоритета во время выполнения действия в сторону снижения говорят о необходимости немедленной отмены). Величина скачка говорит о степени востребованности действия в данный момент: чем быстрее значение приоритета растёт или падает – тем быстрее необходимо, соответственно, выполнять его или отменять. При внедрении порогов выполнения или отмены выполнение или отмена действия должны происходить, если значение приоритета превысило порог выполнения или оказалось меньше порога отмены.

Таким образом, благодаря описанному методу имеем следующие выводы:

* Минимизация дивергенции Кульбака-Лейблера для значений приоритетов есть мера приближения значения приоритета p к значению приоритета q. Действие, в котором в обрабатываемый момент времени значение приоритета приближается к ожидаемому, должно быть выполнено как самое подходящее в данной игровой ситуации. Однако если в системе существуют действия, приоритеты которых гораздо быстрее возрастают, эти действия
* В методе существует возможность минимизации дивергенции **в известном пределе**, что позволяет приближать выполняемые действия к эталонным в разной степени и, следовательно, по-разному для каждого персонажа регулировать его действия. С помощью этого механизма можно добиться различной степени быстроты принятия решений у NPC и, соответственно, повышения или понижения уровня сложности взаимодействия с ним у игрока. Данный механизм на практике может реализовываться с помощью двух порогов: порог выполнения и порог отмены действия. Если значение приоритета находится в соотношении

(8),

действие может поменять очередность в очереди выполнения, но не меняет статус. Если значение приоритета превосходит порог выполнения, оно либо становится следующим в очереди на выполнение, либо немедленно выполняется. Если оно ниже, чем порог отмены, то такое действие должно быть отменено из выполнения, если оно выполняется. Конкретные свойства присвоения и изменения значений порогов во время игры может характеризовать ширину диапазона допустимых приоритетов, и, как следствие, частоту изменения статусов действия. Персонаж с узким диапазоном значений будет быстрее менять действия, и потому, если данная быстрота неотъемлема для сложных персонажей игры, такой NPC может стать **более умелым** в своих действиях в игре, чем другой персонаж, не имеющий такой модификации. Он будет быстрее менять свои действия в игре, принимая и выполняя оптимальные действия и отменяя неоптимальные на каждом такте. Этим он **приблизится** в определённой степени к игроку-человеку в оптимальном поведении согласно игровой ситуации.

* Действие q может быть выбрано как по соображениям соблюдения последовательности действий, так и относительно продолжительности выполнения по времени (поскольку подход должен предусматривать выполнение действий за большее время, чем один промежуток ti). Поэтому, достигается гибкость приближения значений приоритета не только для *выбора действия для выполнения*, но и для принятия решения об *отмене действия* на данном такте.
* Пороги отмены и выполнения действия задаются в относительных единицах, а не в абсолютных, чтобы учесть изменения приоритета во времени. Они не могут быть равны 0 (поскольку выполняется соотношение ).Значение , приближающееся асимптотически к нулю, говорит об идеальной ситуации мгновенного изменения статуса действия вне зависимости от изменения приоритета; значение, приближенное к 100%, говорит о том, что действие никогда не изменит своего статуса, как бы ни менялся его приоритет (поэтому устанавливать близкое к 100% значение нежелательно).
* В соответствии с вышеописанной механикой, установка, например, значения =-30% и =40% говорит о том, что если рост приоритета составил 40%+, будет осуществлена попытка исполнения; если спад составил больше 30% - будет осуществлена отмена действия, если оно выполнялось.
* Пороги можно задавать асимметрично.
* В каждый момент времени выбираются соответственные ему значения приоритета. Следовательно, для каждого момента времени должен следовать перерасчет этих значений, учитывая изменившиеся значения слагаемых значения приоритета. В самой реализации подхода, для хранения временных и доменных значений используются такие структуры, как пулы персонажа и очереди действий.
* Существует необходимость отслеживания динамики изменения приоритета в соседних временных тактах. Поэтому в вычислении значения приоритета участвуют дельта-параметры текущего и предыдущего тактов.
* На практике, не всегда действия можно выполнять параллельно или отменять, если они уже выполняются. Поэтому действия, которые не выполняются и не отменяются, могут не менять свой статус независимо от их приоритета. Поэтому необходимо предусмотреть возможность ранжирования таких действий по приоритетам, чтобы смещать их соответственно ближе к выполнению или к отмене.
* Реализация возможности ранжирования действий по приоритетам предусматривает организацию их очереди. Однако, для того, чтобы реализовать перемещение действий в очереди согласно изменению их приоритетов, необходимо разграничить величину и знак дельта-параметра текущего такта. Поэтому в реализации метода Action First используются вспомогательные offset-параметры O и абсолютные значения дельта-параметров δ.
* Все компоненты формулы расчета значений (2) и (5) предназначены для характеристик как логико-семантических, так и математических параметров действия. Поэтому приоритет и его элементы (значения) являются *полной численной характеристикой* действия, которой можно оперировать при программных расчетах.

Используя данные выводы, в рамках подхода Action First может быть создана система объектов и алгоритмов, реализующая расчет приоритетов действий. Благодаря минимизации дивергенции Кульбака-Лейблера возможно оптимизировать значения приоритетов действий и, таким образом, приблизить *в известной степени* выполняемое действие к ожидаемому, добившись «относительно человекоподобного» поведения NPC как *наиболее* правдоподобной его имитации. Разработке этих возможностей метода Action First и посвящена моя диссертация.

# Глава 2

# Метод оптимизации действий NPC Action First

Метод Action First предназначен для оптимизации действий персонажей игр и сокращения вариантов альтернативного выбора среди действий. Он основан на модульной обработке действий персонажа в его очереди действий и иных своих структурах, ранжируя их и исполняя (либо прерывая) в оптимальные с точки зрения состояния игровой системы моменты времени.

## 2.1 Требования к игре и NPC при применении метода

Action First использует действия персонажей игры, определенные в самой игре разработчиками, влияя на их организацию по времени игры, а не на функционал персонажей. Поэтому, для реализации не требуется переделывание функций NPC, отвечающих за действия. В применении метода существуют особенности и ограничения, приведенные далее.

Метод Action First применяется для игр, где NPC создаются как индивидуализированные **самостоятельные** участники. Обычно в совокупность таких игр входят RPG, шутеры, естественным образом связанные (см. п. 1.1.3 главы 1) связанные с классами B+. Этим и обуславливается применимость Action First к играм класса В и выше.

С точки зрения структур действий NPC у метода есть следующие требования (как правило, выполняющиеся автоматически в вышеописанных классах и жанрах игр):

* *Все действия персонажей имеют структуру*: начало, конец, основную часть;
* Все действия персонажей *вызываются недетерминированными событиями* в игровом мире;
* У персонажей имеются *независимые друг от друга алгоритмы*, регулирующие их поведение;
* взаимодействие персонажей в игре допускает организацию *последовательностей* действий;
* совокупность всех действий персонажа *поддается логическому и функциональному разграничению*, на основании принадлежности к разным предметным областям.

Метод применим и для игр с реальной временной системой, и для игр с пошаговым режимом. В любом случае, временная система должна базироваться на наборе элементарных отрезков, за которые рассчитываются действия персонажей и объектов игрового мира (то есть, допускать подобное разделение на временные интервалы при перерасчете поведения NPC). В системах с реальным временем может использоваться некоторая оговорённая заранее величина отрезка времени, либо стандартная (для многих движков, например, Unreal Engine) мера Tick.

В использовании метода в пошаговых играх есть своя специфика. Здесь NPC предпринимают действия поочерёдно, что упрощает расчёты приоритетов действий (так как становится очевидным краткосрочное взаимодействие с другими персонажами игры, а их действия на этом ходу уже известны).

Метод Action First допускает широкий круг систем событий в играх. События персонажей могут регулироваться как для отдельных персонажей (например, в играх серии Dark Souls), так и быть организованы по вызовам глобального менеджера событий («режиссёра», концепция которого отлично отражена в Left 4 Dead).

Глобальная система событий является «идеальной» для реализации Action First в игре, поскольку делает возможным организовывать структуры действий персонажей, никак не влияя на механизмы возникновения событий для NPC. В данной системе событие возникает независимо от поведения NPC, зато сам NPC может, применяя Action First, правильно реагировать на эти события.

Сложнее реализуется вариант, когда в игре существует локальная система событий для каждого независимого персонажа. В данном случае, при одинаковой с глобальной системой реализации, имеется «замкнутый круг» действий и событий: локальная система событий создает события, на которые NPC реагирует в соответствии с Action First; но в силу своей системы событий последствием его действий являются новые события, в свою очередь влекущие новые действия, и т.д. В Action First предусмотрены два варианта: либо отказ от поддержки концепции очереди действий для данного, конкретного NPC (как реализовано мной в главе 3), либо разрешение конфликта программными способами по принципу: «Действия, поддерживаемые Action First, исключены из локальной системы событий».

Самое важное требование к системе событий – отсутствие предопределенных заранее последовательностей вызовов, приводящих к возникновению в одно и то же время одних и тех же событий. Дело в противоречии саморегуляции действий персонажа и предопределенных условий его действий – в данной ситуации невозможна организация структур действий, предлагаемых в методе. Поэтому, Action First *не предназначен* для игр с системами событий, основанными на детерминированных последовательностях.

## 2.2 Понятия и алгоритмы метода Action First

Опишем понятия, структуры данных и алгоритмы метода.

### 2.2.1 Общие понятия

**2.2.1.1** *Действие* персонажа – центральное понятие подхода.

Действием в подходе Action First является функция или алгоритм, с помощью которого персонаж меняет состояние или поведение как объектов, в отношении которых действует, так и самого себя. Действие (в системе понятий Action First) обладает следующими универсальными признаками:

* *Завершенность*. Действие имеет начало и конец, представляя собой независимый модуль функционала NPC.
* *Направленность на результат*. Результатом действия считается игровое событие или изменение состояния объектов, с которыми взаимодействовал персонаж.
* *Логическая независимость.* Действие может вызываться в системе событий игры или в зависимости от конкретного состояния персонажа, но не является частью детерминированной последовательности, предопределенной глобальной структурой игры («цепочка скриптов»).
* *Недетерминированность.* Действие происходит не по заранее известным схемам или условиям, но в зависимости от объективной необходимости в нём вследствие действий игрока и NPC.

Сущность и содержание действий определяются разработчиками игры. Для их программной реализации создаются скрипты или программные модули, которые, однако, затрагивают только конкретную реализацию действия. Организацию их выполнения обеспечивает модуль реализации Action First, **не затрагивая** при этом функционал самих действий.

В зависимости от жанра и характера игры для действий и включаемых в это понятие алгоритмов могут существовать разные трактовки. Существует потребность логического, типового и целевого разграничения действий, поэтому в подходе существуют усложнённые структуры – контексты действий, цели, типы и цепочки действий (которые описаны в следующих пунктах).

**2.2.1.2** У действий возможны различные трактовки также с точки зрения начала и завершения, поскольку начало, исполнение, завершение или отмена действий может совершаться по разным причинам и с разными условиями. Для управления состояниями действий введены понятия *статусов* действий. Статусы делятся на текущие и завершающие, в зависимости от критерия их начала или завершения.

Среди статусов текущего типа бывают:

* *Доступен*. Базовый статус для действия в данной группе. Означает, что действие никогда не использовалось персонажем, но может быть выбрано в дальнейшем.
* *Начат.* Персонаж начинает выполнять действие.
* *Недоступен.* Означает, что на данном этапе действие не может быть выбрано персонажем.

Среди статусов завершающего типа существуют следующие:

* *Выполнен.* Действие было успешно завершено.
* *Исключен.* Действие исключено из очереди, не начав выполнение.
* *Отменен.* Действие выполнялось, но было отменено.
* *Выполняется.* Действие выполняется на данный момент.
* *Параллельно выполняется.* Присваивается только действиям, которые выполняются параллельно (если персонаж допускает это).
* *Ожидание.* Означает, что персонаж поставил действие в очередь (оно было ранее отменено; этим данный статус отличается от статуса Доступен).

Таким образом, в методе Action First действие предстает в виде детализированного сложного объекта, образованного сопоставлением реализации действия в виде скрипта или метода, набора его статусов (изменяющегося в соответствии с алгоритмами исполнения), и численным значением приоритета действия.

**2.2.1.3** *Приоритет действия* – численная характеристика, которая сопоставляется с действием. Введение приоритета в систему понятий подхода обусловлена необходимостью сопоставления логико-семантической структуры, которой является действие, с конкретным числом, которое может участвовать в расчётах при исполнении, ожидании или отмене действия.

С концептуальной точки зрения, приоритет действия присваивается всем действиям по мере того, как они исполняются персонажем, и отражает своей величиной два фактора:

1. Насколько актуально данное действие на нынешнем такте игры;
2. Насколько часто данное действие выполняется персонажем.

В обоих случаях чем выше приоритет – тем больше вероятность, что данное действие будет выполнено, а чем ниже – тем больше вероятность, что отменено. В Action First приоритеты имеют единообразный механизм повышения и понижения: значение приоритета повышается, если актуальность и частота использования действия персонажем растёт, и снижается, если падает.

Теория расчёта приоритета действия приведена в пункте 1.3.5 главы 1. Практический пример его применения представлен в подразделе 2.4 настоящей главы.

**2.2.1.4** Цель действия – термин, означающий критерий завершения определенной последовательности действий. Так же, как и действие, он сопоставлен с численной характеристикой, именуемой приоритетом цели. Критерий завершения необходим для прерывания цепочек действий при успешном их исполнении и совпадает по значению с текущим пределом исполнения для действия. При реализации подхода сложно сказать, какая цель на данный момент выполняется, потому данное понятие включено в концепцию действия (если точнее – в его условие *целенаправленности*) и не используется отдельно.

### 2.2.2 Структуры данных Action First

**2.2.2.1**Контекст действий персонажа – некоторая логическая предметная область, к которой относятся те или иные его действия. Контекст персонажей на примере игр-стратегий хорошо представлен в виде разных сторон продвижения игрока к победе. Пример контекстов представлен на рисунке 2.1.



Рисунок 2.1 – Контексты действий персонажей в Crusader Kings II

Роль контекстов заключается в распределении действий по отдельным областям, меньшим, чем весь домен действий. Например, если действие, относящееся к дипломатии, никогда не выполняется в войне, можно отнести его к контексту дипломатии, и тогда оно **не будет** одним из возможных именно в войне.

Таким образом, контексты исполняют роль «доменов» действий (но все они входят в домен действий персонажа), позволяя реализовывать отбор действий из ограниченного набора, что вводит дополнительные возможности для изолированного логического принятия решений в каждом поддомене (допускающего совместное существование решений и их параллелизм [42]).

В каждом контексте процесс выбора и ранжирования действий в очереди не допускает участие иных действий из иных контекстов.

Контексты должны использоваться вместе с их приоритетами. Приоритет контекста – количественный показатель, учитывающий важность данного контекста для персонажа (то есть, объединяет в себе показатели *частоты* использования действий из контекста). Таким образом, приоритет – численное значение, поставленное в соответствие семантической значимости контекста (и так же, как и приоритет действия, не используется отдельно от контекста).

В игре Crusader Kings II на рисунке 2.1 представлены сразу несколько контекстов:

* Геополитика, как слежение за владениями своей и соседних стран;
* Война, то есть сбор войск и бои;
* Дипломатия, как договор о даре союзнику и присоединение к войне союзника;
* управление персонажем-правителем страны.

**2.2.2.2** *Цепь действий* – последовательность взаимосвязанных действий, которые по замыслу разработчика цепи должны выполняться последовательно. Типичный пример цепи действия – «комбо» противников игрока в играх-файтингах.

Действия в цепи связаны взаимными условиями. Первое условие **гарантирует**, что второе действие в цепи начнётся *только после выполнения первого*. Другое – второе действие *не начнётся раньше* первого (это лишь необходимое условие для определения элемента цепи, но не достаточное!). Данные условия введены в подходе, так как необходимо соблюдать не только цепь действий, но и оградить персонажа от выполнения аналогичных отдельных действий во время выполнения цепи с ними.

Цепи действий бывают прерываемыми и непрерывными.

Прерываемые (динамические) цепи действий *не связаны* строгими условиями выполнения для завершения цепи. Если такая цепь во время выполнения становится невыполнимой или невыгодной в следующем фрагменте (в следующем действии, благодаря низкому либо нестабильному значению приоритета), цепь *завершается* текущим действием. Таким образом, данная организация цепи становится выгодной в случае быстро меняющихся условий действий персонажа.

Непрерывные (статические) цепи характеризуются строгим условием завершения. Это означает, что данная цепь действий выполняется персонажем, пока не завершится последнее действие в ней. Данная структура удобна в случаях, когда действия обрабатываются различными программными модулями, но логически неразрывны, либо для сохранения смысла действий персонажа они обязаны выполняться последовательно.

**2.2.2.3***Очередь действий* – одно из центральных понятий метода Action First. Очередь действий является абстрактной структурой, представляющей порядок выполнения действий персонажа по времени.

Очередь действия является упорядоченной структурой, представляющей с программной точки зрения **список**. Элементами списка могут являться и отдельные действия, и их цепи. Таким образом, учитывая цепи действий как структуры, очередь частично поддерживает вложенность: при размещении в очереди цепи в очередь помещаются все действия цепи, если цепь ещё и статическая – её можно тоже рассматривать в широком смысле как вложенную очередь действий.

Очереди действий могут являться как строгими последовательностями, так и допускать **параллелизм** (то есть, состояние, когда на выполнении могут стоять одновременно несколько действий персонажа). Применение данных типов очередей зависит от специфики персонажа и пожеланий разработчика: если образ действий персонажа допускает одновременно выполняющиеся действия, очередь действий может быть организована с учетом параллелизма выполнения.

Элементом очереди является пара из самого действия и сопоставленного с ним приоритета. По фактическим значениям приоритетов идёт упорядочивание очереди и все предусмотренные в подходе операции с действиями. Смещения действий в очереди проводятся также с учетом текущих статусов действий.

Классический последовательный тип очереди действий (c указанием приоритетов и вложенной статической цепью действий) представлен на рисунке 2.2.

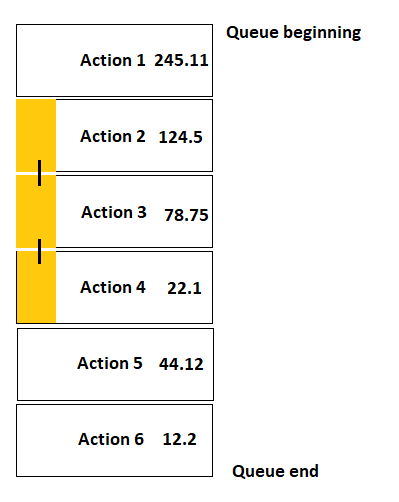


Рисунок 2.2 – последовательная очередь действий

Действие 4 на рисунке 2.2 заслуживает внимания: несмотря на то, что его приоритет ниже, чем у следующего в очереди, оно является элементом неразрывной цепи, а потому будет исполнено раньше, чем действие с большим приоритетом, но вне этой цепи.

По мере близости действия к исполнению оно смещается ближе к началу очереди, по выполнению или прерыванию оно покидает очередь.

Очередь действий формируется в начале жизненного цикла персонажа, но заполняется действиями (или очищается от них) по мере смены игровой ситуации. Поскольку игровая ситуация меняется один раз в элементарный промежуток игрового времени, пересчет и ранжирование действий в очереди происходит один раз в данный промежуток.

Согласно новым рассчитанным приоритетам, таким образом, действиям присваиваются их флаги, которые означают, как должен быть рассчитан приоритет данного действия на следующих шагах (см. подпункт 2.2.1.2 о статусах действия).

В каждый элементарный промежуток времени игры можно получить, таким образом, уникальный «срез» очереди действий, который возможно проанализировать, что даёт возможность глубокого семантического анализа действий NPC, и получить точную формулу расчета приоритета для каждого действия. Пример данного среза представлен на рисунке 2.2 – понятно, что с каждым моментом времени данная картина может измениться, особенно в значениях их приоритетов.

**2.2.2.4** Возникает вопрос о связанности очереди действий как списка. Данный вопрос в методе Action First разрешается, в отличие от стандартной организации списков, и очередей, в зависимости от природы действий.

Действия могут быть связаны друг с другом как логически (посредством, например, цепей действий), так и семантически. Логическая связь представляет собой строгую последовательность, для отображения которой в подходе и существуют цепи действий. Семантическая связь же предусматривается разработчиком, чтобы организовать стратегию действий персонажа как долгосрочную логику его поведения.

Семантическая связь становится важнее, чем логическая, в случае, когда имеется неопределенность или недетерминированность игровой ситуации, а также неопределенность в прогнозировании действий игрока на следующем временном такте или в отдаленном будущем. Семантические связи действий, таким образом, необходимы для замещения неэффективных в условиях неопределенности логических связей цепей.

С точки зрения организации очереди действия связаны друг с другом посредством парного флага «before & after». Однако, в отличие от логической связи (предусматривающей строгую последовательность действий) у данной связи нет условия неразрывности. Так, на рисунке 2.3 действие Action 6, связанное семантической связью с Action 1, может выполниться после всех остальных действий в очереди (но **не раньше**, чем Action 1).

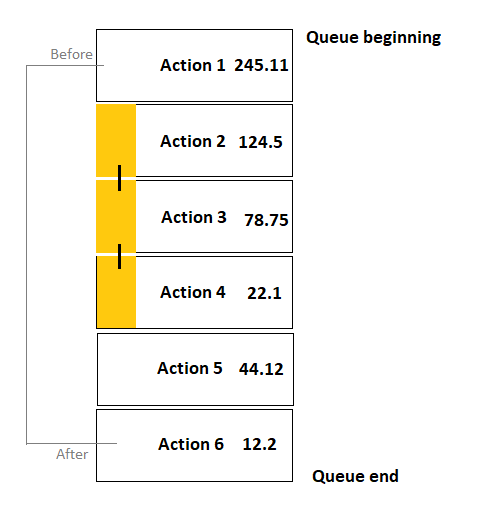


Рисунок 2.3 – семантическая связь действий в очереди

Таким образом, несмотря на связь двух действий, между ними могут выполняться действия, не входящие в данную связь. Данный механизм не противоречит условиям неразрывности связи, поскольку семантическая связь отображает глобальные стратегии поведения, в том числе и растянутые по времени. Однако условия последовательности самих действий по-прежнему выполняется: действие с флагом after будет выполнено только после действия с флагом before, но необязательно *непосредственно* после него. Как и для цепей действий, семантическая связь существует в пределах текущей очереди, и её члены не могут быть исключены из очереди прежде, чем будет исключен самый первый её элемент.

## 2.3 Алгоритм Action First

Алгоритм метода (см.п.1.3.5) состоит из следующих этапов и пунктов:

1. *Инициализация*. Это этап определения действий, которые нужно будет оптимизировать. У него есть следующие этапы:
   * Определение **персонажей**, которых необходимо оптимизировать. Для этого формируется выборка целевых персонажей из общей совокупности персонажей в игре.
   * Определение **контекстов и действий** каждого персонажа в рамках контекстов. Сперва определяются контексты как домены действий в узкой области поведения NPC. Затем определяются действия, которые входят в данный контекст.
   * Создание **очередей действий** для каждого обрабатываемого персонажа.
2. *Присвоение начальных значений*. Это этап установки базовых значений приоритетов для действий и структур действий. Осуществляется реализация этих значений в программе как чисел или структур данных, эквивалентных структурам действий.
   * Определение **формулы влияний** для базовых приоритетов действий. Производится составление формулы

(10),

с определением для каждого действия мультипликативных и аддитивных величин, которыми характеризуются взаимодействия персонажа с другими объектами игры. В примере подраздела 2.4 данный шаг представлен более детально.

* + Определение **базовых приоритетов действий** В для . Эти значения устанавливаются разработчиком, исходя из соотношения между взаимной потребностью в действиях (действие, которое NPC должен применять чаще, должно иметь большее значение ).
  + Определение **логических и семантических связей** между действиями. Цепи действий могут быть организованы как отдельные структуры, например как стек или очередь. Для цепей действий, соответственно, могут назначаться коэффициенты элементов цепи Cj, которые затем будут участвовать в расчётах для своих действий. С их помощью можно установить зависимость между элементами одной цепи (например, можно установить последовательно убывающие коэффициенты , , , … , и таким образом описать, что это нестрогая цепь действий.
  + Определение **противодействий** (логических или семантических связей пары действий). Противодействия определяются как равенство базовых приоритетов, но с различными знаками коэффициентов Cj. Их смысл в том, что два действия, влияющие друг на друга с противоположными коэффициентами Cj, будут иметь следующие приоритеты: чем выше приоритет одного действия, тем ниже приоритет другого (за счёт вклада слагаемого базового приоритета – см. формулу (5)), и наоборот.
  + Установка **статусов** для действий. По умолчанию, используется статус Доступен (Available, cм.пп.2.1.2.2). Хотя, в соответствии с задумкой разработчика конкретной игры, изначально могут быть присвоены и другие статусы. Например, если даже на первом такте игры необходимо выполнить какое-то действие для NPC, необходимо присвоить ему статус Start.

1. *Расчет первичных значений*. Здесь рассчитываются все приоритеты действий на текущем такте игры. Они рассчитываются по формуле

(11)

* + Расчет элементарного приоритета по формуле

(12),

где – коэффициент действия, влияющего на целевое,

– приоритет влияющего действия.

* + Расчет совокупного приоритета действия по формуле

(12),

* + Расчет дивергенции DKL происходит с учетом того, что дельта-параметры пока не определены (поскольку идёт расчёт только первичных значений). Поэтому применяется следующая формула:

(11).

* + Определение дельта-параметров. Дельта-параметры определяются следующим образом: на первом шаге значение дельта-параметра значением всей дивергенции (поскольку именно настолько изменилось расхождение между новым значением и стабильным на данном такте). На первом такте вычисляется параметр , в дальнейшем – его значение присваивается параметру , а у вычисляется новое значение для текущего такта.

1. Вычисление порогов выполнения и отмены действий. Оно может вычисляться следующими двумя способами:
   * Вручную. Разработчик создаёт глобальные переменные порога выполнения и отмены, где хранит их значения (реализовано мной в тестовом примере, поскольку это решение проще реализовать).
   * Автоматически. Пороги могут определяться и таким образом. Например, порог отмены может быть установлен даже в абсолютное значение, равное 0 (таким образом, действия не будут отменяться вообще), если это не противоречит задумке поведения персонажа. Что касается порога выполнения, он может быть автоматически задан как максимальный приоритет на предыдущем такте – если приоритет действия на текущем такте его превысил, это действие, по задумке, будет выполняться.
2. Ранжирование действий. Здесь происходит упорядочение действий в очереди по их приоритетам и присвоение им флагов.
   * Выполняется пересмотр начальных статусов. Если статусы или связи между действиями в очереди не допускает их перестановок (например, действие уже выполняется и не может быть отменено), эти действия сохранят своё положение в очереди.
   * Остальные действия в очереди меняют своё положение в соответствии с приоритетами. Если в очереди есть строгие цепи действий, они перемещаются **совместно**, учитывая перестановку в очереди первого из них.
   * Действия, приоритеты которых выше порога исполнения, могут начать выполняться на данном такте. Однако, учитывая возможность выполнения действий из предыдущих тактов (и невозможность параллельного исполнения, если для этого NPC таковое исполнение недоступно), такие действия могут только сместиться в очереди на второе место, после выполняющегося действия.
   * Действия, приоритеты которых ниже порога отмены, прекращают своё выполнение (если их логика это допускает) с изменением статуса на Revoked. Действия, которые имеют такие приоритеты, но не выполнялись, должны быть исключены из очереди. Исключение из очереди или отмена выполнения происходит с учетом связей, в которых действия участвуют: например, исключение действия, являющегося элементом нестрогой цепи действий, исключит все дальнейшие действия в цепи из очереди.
   * Действия, которые могли бы быть выполнены на данном такте игры, но не участвовали доселе в очереди, добавляются в очередь на места согласно своим приоритетам.
   * В соответствии с вышеописанными правилами для каждого действия изменяется статус. Правила изменения статусов для действия приведены в приложении А.
3. Расчет на последующих тактах. На последующих тактах для расчетов принимаются приоритеты и статусы действий из предыдущего. При этом, на любом промежуточном такте используются уже не один, а **два** дельта-параметра для расчета порогов выполнения и отмены. Таким образом, возможно описывать динамику прироста или снижения приоритета конкретного действия на каждом временном такте.
4. Апостериорный анализ. Поскольку информация о действиях NPC сохраняется на каждом такте, существует возможность её вывода для информирования разработчика о детальном механизме принятия решения NPC. Таким образом, в Action First предусмотрена возможность просмотра и анализа приоритетов действий персонажей, позволяющие уже разработчикам анализировать результаты своей работы и проводить улучшения, если это необходимо. Теоретически, возможность получения данных с каждого такта игры исходит из возможности рассчитывать конкретные значения всех компонентов формул и сами значения приоритетов в каждый момент времени (см.п.1.3.5) – поэтому для каждого значения приоритета в игре будет существовать своя зависимость величины по времени.

## 2.4 Пример работы метода Action First

Для более наглядного описания алгоритма Action First, а также для подготовки к его технической реализации целесообразно рассмотреть весь процесс работы метода на конкретном примере.

### 2.4.1 Постановка задачи рабочего примера

Для тестового внедрения Action First я взял (ранее разработанную мной в рамках работы над диссертацией и другими моими проектами) игру Super Sparty Brothers, которая является 2D-платформером с геймплеем, подразумевающим сбор монеток и вещей игроком и избегание им встреч с врагами. У игрока есть некоторое количество здоровья, которое теряется при каждой встрече с врагом. Игрок может оглушать врагов, спрыгивая на них сверху, но не может их убить (враги спустя некоторое время перестают быть оглушёнными и продолжают искать игрока). Пример локации (используется как рабочий в следующих пунктах) представлен на рисунке 2.4.



Рисунок 2.4 – Локация Super Sparty Brothers.

Игра достаточно проста, чтобы показать механизм Action First максимально прозрачно для читателя диссертации. Метод Action First реализован здесь относительно просто, поскольку в игре существует лишь один тип NPC и 3 типа действий (jump, double-jump, move). Естественно, пример реализации для более сложных игр будет куда менее понятен человеку, чем рассматриваемый, и потребует более сложных расчётов.

На момент создания данной игры (но **до** внедрения туда Action First) у NPC (врагов) реализованы следующие механики:

* детерминированное перемещение между точками пути (механика waypoints) с «патрулированием» означенной точками пути области;
* враги не могут прыгать с платформы на платформу, как игрок (но могут спрыгивать с платформ вниз, если их точки пути это позволяют);
* враги *никогда* не меняют обозначенное точками пути направление движения (например, игрок может проходить некоторые участки, двигаясь за спиной у врага).

Учитывая вышеописанные факты, семантически использование Action First должно привести к следующим эффектам у NPC:

* движение не по точкам пути, а в зависимости от положения относительно игрока (например, сближение с игроком или уход с траектории прыжка игрока, чтобы не быть оглушенным);
* прыжки с платформы на платформу, использование double-jump для достижения слишком высоких точек движения;
* исключение самопроизвольных падений с «земли» внизу локации (ниже существует объект killzone, играющий роль бездонной пропасти и нужный для уничтожения любых объектов, которые упадут за пределы локации).

Для реализации метода в данной игре я использую ПО Action, разработанное как реализация Action First. Подробнее об устройстве Action см. главу 3.

### 2.4.2 Инициализация Action First

Для инициализации метода необходимо обозначить персонажей, которые обрабатываются методом, и их действия. Также, необходимо определить зависимости и контексты действий.

Определим персонажей посредством присвоения им флага IsActionFirst, который показывает, будут ли взяты эти персонажи для обработки. Функции персонажа, которые считаются действиями (Jump(), Move() и Double-Jump()) обозначены атрибутами [ActionFirst], что позволяет в дальнейшем вычислять для них приоритет.

У данного шага есть программная реализация. В данном примере ПО Action подключено как ресурс для игры, что **не единственный** (но наиболее простой и единообразный) способ. Теоретически, существует способ распознавания персонажа и его действий извне («режим мода»), однако данный способ имеет технические ограничения. Подробнее суть способов подключения и инициализации NPC и действий внутри игры изложена в пп. 3.2.3 главы 3 и в заключении.

В Super Sparty Brothers нет зависимостей действий NPC (поскольку направление и характер их движения, по задумке постановки задачи, меняется динамически, и «комбо» тут не могут быть). Поэтому, здесь не нужно организовывать зависимости действий.

Зато контекстов два: контекст движения, включающий в себя само действие Move(), и контекст прыжков, включающий в себя Jump() и DoubleJump(). В первом контексте необходим выбор не только самого действия, но и его направления (по идее, действие Move() является составным из двух элементарных действий MoveLeft() и MoveRight(), однако из соображений принципа DRY [43] и единообразия программной реализации в самой игре это одно и то же действие, но с разным вектором движения-аргументом).

### 2.4.3 Определение начальных значений действий NPC

На данном шаге необходимо определить базовые приоритеты и статусы каждого действия.

Но вследствие специфики действий NPC в данной игре, необходимо сперва отделить функционал «директивного движения» от функционала самого движения (вопреки концепции Action First, которая говорит о невнесении изменений в функционал действий). На самом деле, действие Move() является в исходной игре не чем иным, как цепью действий Move <Direction> (), поэтому такое разграничение имеет место. Схема данного действия представлена на рисунке 2.5.

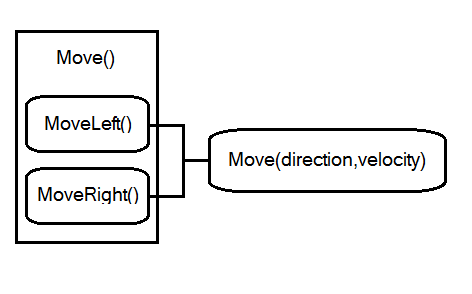


Рисунок 2.5 – Разграничение составного действия Move()

На следующем этапе выполним задание базовых приоритетов действиям Move() (тут уже имеется ввиду функция элементарного движения на рисунке 2.5, а не функция движения оригинальной игры!), Jump() и DoubleJump() (два последних действия, как и предполагалось, остались без изменений, поскольку изначально пригодны для обработки как элементарные).

Определим значения коэффициентов влияния для действий.

Для действия Move существуют следующие компоненты:

* Близость нахождения игрока. Учитывая задумку игры (NPC патрулируют местность и, увидев игрока, нападают на него), NPC должен более упорно следовать за игроком, когда он поблизости, и менее упорно – на более дальней дистанции. Учитывая природу этого явления, его можно охарактеризовать мультипликативным параметром с1=2.5.

На этот коэффициент будет умножаться дистанция между NPC и игроком для расчёта значение формулы влияния; значение 1 указывает на отсутствие данной связи, а иные значения – скорость возрастания приоритета приближения в зависимости от близости персонажа к игроку. С другой стороны, коэффициент не должен быть слишком велик, чтобы другие действия, в случае противодействий игрока, тоже могли выполняться.

* Близость края локации (Killzone Proximity). Не нужно, чтобы персонажи сами падали с локации. Вероятность отмены движения к краю локации должна повышаться, причём должна повышаться вероятность перемещения в **обратном** направлении. Поэтому, здесь необходим мультипликативный параметр с2=-1.3. (отрицательный знак говорит о том, что, если в очереди действий есть аналогичное действие с положительным знаком – будет происходить повышение приоритета одного действия за счёт снижения приоритета другого. Поэтому, чем ближе персонаж к краю локации – тем более вероятно, что он отменит дальнейшее движение и пойдёт в противоположном направлении).
* в случае дилеммы нападения игрока на NPC (игрок находится вверху в прыжке и может упасть на персонажа, оглушив его) необходимо рассуждать так. Исходя из специфики Move(), персонажу нужно либо убежать от прыгающего на него игрока (то есть, двигаться по направлению прыжка), либо – бежать в противоположном от направления прыжка игрока направлении (чтобы игрок в итоге промахнулся и «перелетел» свою цель). Однако учитывая особенности геймплея, в частности возможность игрока «рулить» в полёте (нажимая клавиши движения, можно корректировать траекторию падения, которая, впрочем, никогда не превосходит стандартную дальность прыжка по дальности), я как разработчик выбрал для этой ситуации мультипликативный коэффициент влияния с3=2.3 (NPC будут пытаться убежать от игрока в направлении его прыжка). И также необходим аддитивный параметр M1=90, который показывает угол между текущей траекторией полёта игрока и плоскостью, в которой двигается NPC (горизонталь, поскольку движение Move() рассчитывается только для горизонтального движения вправо или влево).

Таким образом, формула расчета коэффициента влияния для Move следующая:

(12),

где D-дистанция до игрока,

KP-дистанция до края локации,

α – угол траектории игрока относительно персонажа.

Необходимо понимать, что значение коэффициента вычисляется динамически **на каждом шаге** (поскольку значения параметров формулы (8) будут, понятно, меняться). На данном этапе необходимо утвердить именно **сам вид** формулы, а не результаты расчёта по ней.

Аналогичным образом, определим формулу расчета коэффициента влияния для Jump (и необходимо определить аналогичные выражения для всех остальных действий персонажа):

(13),

где – дистанция до платформы, на которую персонаж прыгает,

-угол траектории движения игрока относительно персонажа (коэффициент с5 отрицательный потому, что во время прыжка игрок не может умереть от пересечения с врагом, поэтому есть смысл «ловить» игрока не во время прыжка, а во время приземления),

– высота нахождения героя относительно персонажа (персонажу нужно будет до него допрыгнуть),

– дистанция до края локации (крайне не рекомендуется прыгать в сторону края локации, поскольку нет никаких гарантий корректировки траектории с помощью Move, и возврат из такой ситуации на ближайшую платформу – для NPC весьма сложная задача).

Аналогичная формула справедлива и для DoubleJump(). Данное действие является двойным выполнением прыжка (когда персонаж может прыгнуть, уже находясь в полёте), поэтому у него уже в оригинальном функционале есть проверка, находится ли NPC в прыжке. Только в этом случае он может выполнить данное действие.

Определим приоритеты исходя из относительной важности действий.

На практике, разработчиком определяются некоторые произвольные «границы» значений, ниже которых приоритеты он не задаёт. Приоритет со значением «нижней» границы имеет то действие, которое выполняется реже всего, максимальный приоритет – у самого частого действия. Разработчик может решать вопрос о частоте и актуальности каждого конкретного действия используя как свой экспертный опыт в геймплее игры, так и исходя из статистических собранных данных. Важно именно соотношение между приоритетами, а не их величины, поскольку соотношение, выступая мультипликативной частью для всех вычисленных коэффициентов, определяет совокупное значение приоритета.

Зададим в нашем случае базовый приоритет действия так:

* B(Move())=7; //максимальный приоритет
* B(Jump())=4;
* B(DoubleJump())=1; //минимальный приоритет

Теперь необходимо назначить статусы для действий. В Super Sparty Brothers **нет** ситуаций, когда уже с загрузки локации NPC уже находятся в процессе выполнения действия (хотя теоретически в играх может возникнуть такая ситуация, поэтому и возможно с самого начала назначать действиям любые статусы). В текущей ситуации, достаточно всем действиям назначить статусы Available.

Таким образом, перед обработкой действий на первом такте у каждого NPC есть очередь действий, схематически представленная на рисунке 2.6.

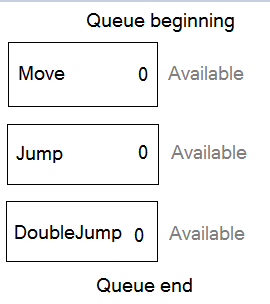


Рисунок 2.6 – Очередь действий NPC Super Sparty Brothers

### 2.4.4 Расчет приоритетов действий

Приступим к расчёту приоритетов. Для наглядности, я проведу этот расчёт для конкретного NPC, представленного на локации на рисунке 2.7.

Для расчёта нам понадобится формула расчёта приоритетов, неоднократно приведённая ранее:

(14)

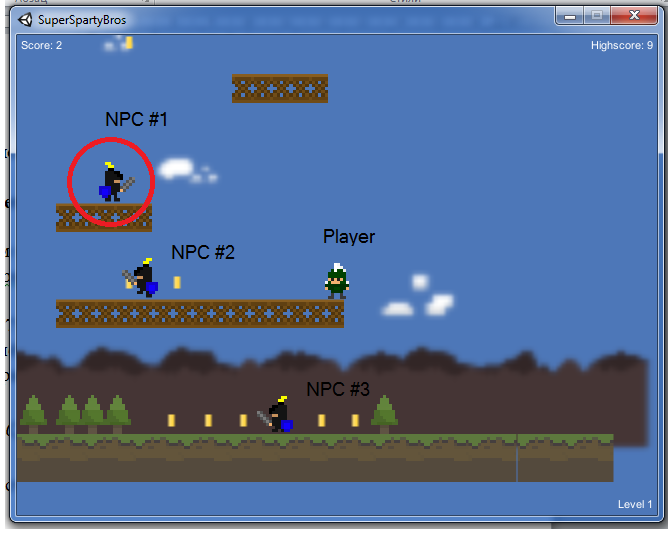


Рисунок 2.7 – Целевой NPC

Начнём расчёт с компонентов формулы.

Сперва подсчитаем базовые значения приоритета Bj, учитывая все его компоненты. Существуют три этапа расчёта:

Найти по формуле (8) значение коэффициента влияния Сm:

Приступим к расчёту приоритетов. Для наглядности, я проведу этот расчёт для конкретного NPC, представленного на локации на рисунке 2.8.

(15)

=-3.089 (16),

(17)

Элементарный приоритет действия DoubleJump равен нулю, поскольку на практике на этом этапе сработала проверка состояния персонажа. Он не находится в прыжке, поэтому DoubleJump в принципе не может быть выполнено.

Рассчитаем по формуле (1) количество информации:

10,92 (18)

2,599 (19)

Рассчитать по формуле (5) результат:

76,44 (20)

Рассчитываем коэффициент зависимости цепи С. Учитывая, что элементом зависимости или цепи действия не являются, а их статус назначен по умолчанию, значение коэффициента равно 1. Иной случай покажем в пункте 2.4.6, когда рассмотрим действие уже со значащимся статусом.

Вычислим теперь аддитивную часть. В аддитивной части, согласно положениям пункта 1.3.5, необходимо включать иные действия из контекста, которые непосредственно влияют на рассчитываемое, с коэффициентом влияния. Однако, в нашем случае таких действий нет (поскольку действие Move() вообще одно в контексте, а Jump() вообще выполняется первый раз), поэтому аддитивная часть формулы равна нулю.

Вычислим дивергенцию приоритета по формуле

(21)

Учитывая замены из-за первого такта

=1 (22)

и

итоговое значение приоритета для Move() будет равно

Формула является частным случаем формулы, поскольку на первом такте за эталонный приоритет берется базовый, исходный приоритет (смысл которого – в предположении разработчиком о вероятности исполнения именно данного действия), а оба дельта-параметра в силу первого такта равны нулю.

Необходимо сразу присвоить значение дивергенции дельта-параметру. Это пригодится на следующем такте.

И, наконец, вычислим по формуле (5) значение приоритета:

Во время аналогичных расчётов получены приоритеты иных действий в очереди действий NPC. На практике значения приоритетов вычисляются значения приоритетов в том порядке, в котором они записываются. Состояние очереди на конец данного этапа представлено на рисунке 2.10.

Рисунок 2.10 – очередь действий с приоритетами

Необходимо заметить, что принятие решения по каждому действию зависит, естественно, не только от детерминированных факторов. Ситуация, приведённая на рисунке 2.9, может быть другой при другом запуске игры, если, допустим, изменятся условия в локации (например, передвинется платформа), или игрок сделает другие действия. Однако, приоритеты действий всегда меняются по очерченным на этапе инициализации зависимостям, что позволяет контролировать правильное поведение NPC при любой неопределенности действий игрока.

### 2.4.5 Ранжирование действий

Перед ранжированием выставим пороги выполнения и отмены действий. Как я уже упоминал в пункте 2.3, я использую в данном примере ручной порядок выставления приоритетов.

В соответствии с задумкой игры и природой действий,

Для ранжирования используем очередь, представленную ранее на рисунке 2.10. Теперь необходимо изменить порядок действий в очереди (если нужно) и назначить им новые статусы, согласно новому порядку действий.

В нашей очереди присутствуют три действия, все имеют один и тот же начальный статус Available и не имеют никаких зависимостей между собой. Поэтому, данные действия можно ранжировать только по значениям приоритетов, не проводя дополнительные манипуляции с элементами зависимостей действий (см. п. 2.2.4) или статусами выполнения (см. п. 2.2.2)

В соответствии с ранжированием, очередь действий будет выглядеть так, как представлена на рисунке 2.11:

Рисунок 2.11 – Очередь действий после ранжирования

Теперь, изменим статусы действий.

В очереди на данный момент нет выполняющихся действий (что понятно, на первом такте такая ситуация бывает редко, а в данной игре – не бывает никогда). Поэтому, выполняться начнёт первое действие в очереди. Ему назначается статус Executing.

Если бы в очереди уже было действие с подобным статусом, решение о выполнении принималось бы в зависимости от того, допускается ли параллелизм выполнения действий. Если да, порядок действий был бы тем же. В ситуации данной игры параллельное исполнение действий невозможно, поэтому статус первого действия в очереди изменяется на Executing только если выполняемое действие было бы отменено или завершено.

В очереди также нет действий, приоритет которых возрос бы выше, чему у текущего выполняемого. Если бы это случилось и выполняемое действие допускало бы свою отмену, оно было бы отменено и сменило бы свой статус на Revoked, а у действия, которое бы превзошло его, статус сменился бы на Executing и оно бы выполнилось.

В очереди нет действий, у которых бы наблюдалось падение приоритета ниже порога отмены (в нашем случае равно нулю). Если бы это случилось,

В случае наличия в очереди действий со статусами, отличными от Available, приоритеты необходимо было бы умножить на их коэффициенты (приведены в приложении А; у самого Available этот коэффициент равен 1). Данный шаг необходимо было бы сделать перед самим ранжированием.

В соответствии с итогами ранжирования на этом такте рассматриваемый NPC примет действия, представленные на рисунке 2.10:



Рисунок 2.10 – NPC и его путь на такте 1

Если учесть всю ситуацию на локации – NPC поблизости выберут решения, как на рисунке 2.11:

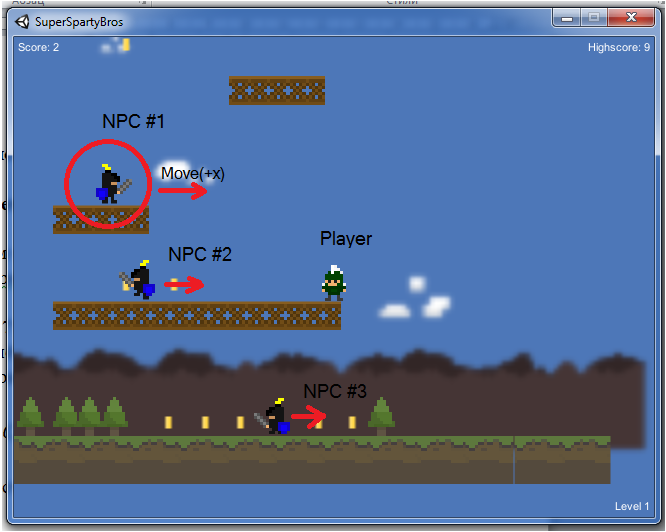


Рисунок 2.11 – Решения всех NPC на такте 1

### 2.4.6 Последующие шаги

На последующих шагах происходит аналогичный расчёт начиная от расчетов приоритета каждого действия NPC и заканчивая ранжированием. Однако, поскольку на следующих шагах необходимо рассчитывать влияние предыдущих, необходимо подключать к расчётам дельта-параметры и учитывать коэффициенты зависимости в расчётах приоритетов.

Рассмотрим ситуацию, представленную на рисунке 2.11.

Рисунок 2.11

Здесь у NPC 2 есть уже выполняющееся действие, которое – с точки зрения геймплея – закончится только после приземления его на землю. Поэтому, данный NPC может иметь любую конфигурацию очереди действий, но всё равно не сможет предпринимать действие Jump() в любых направлениях, пока не приземлится на ближайшую платформу. Математически, это обосновано тем что коэффициент зависимости от статуса Started достаточно высок, что только намного более приоритетное действие (в силу ситуации невозможное – приоритетней прыжка может быть только другой прыжок) может выполниться вместо него. Таким образом, на практике в таком состоянии может выполниться только действие DoubleJump(), поскольку выполнение Jump() является его необходимым условием.

Также, дельта-параметры, рассчитанные на всех предыдущих тактах, будут говорить о постоянном (но замедляющемся с прошедшими тактами) возрастанием приоритета, поэтому они будут также способствовать непрерывному выполнению прыжка.

Однако NPC может гипотетически принимать решения о выполнении действия Move и, подобно игроку, управлять своей траекторией движения в полёте. На нашем примере, такой элемент геймплея можно внедрить, если для очереди будет возможен параллелизм (в примере очередь создается без возможности параллелизма).

## 2.5 Программные основы реализации Action First

Action First применяется для любых персонажей. Однако и сами персонажи, и их действия могут быть реализованы различными способами, в зависимости от выбора разработчика. Поэтому появляется необходимость определять действия персонажа программно.

Метод использует объектно-ориентированный подход к разработке персонажей. По нему, персонаж имеет класс, который определяет все его характеристики и поведение. Поведение, в частности, реализуется посредством методов класса. Некоторые из методов класса персонажа определяют его действия, поэтому они и должны браться в качестве действий на вход алгоритма оптимизации.

Таким образом, рассматриваемые как действия методы персонажа должны быть обозначены (помечены) как действия, участвующие в ранжировании. Подробнее данный механизм рассмотрен в подразделе 3.2.

Контекст действий, с точки зрения своего определения как «поддомена», должен быть определен самим разработчиком при создании персонажей. По умолчанию, в любой игре существует как минимум один контекст (который, если он один, включает в себя все действия NPC).

В случае одного контекста ограничения на выбор действий из их домена отсутствуют, однако в случае нескольких контекстов необходимо разделять какими-то способами действия, принадлежащие к разным контекстам. Например, данную задачу можно реализовать программно с помощью шаблонов либо с помощью типизации (например, Action <Diplomacy> будет обозначать действие, относящееся только к контексту дипломатии).

Очереди действий в программной реализации являются списками действий в упрощённом варианте, и списками сложного типа (список пар «действие – приоритет») в классической ситуации. В зависимости от выбора данной реализации (в главе 3 я реализовал очереди действий в их классическом варианте) могут быть различные методы доступа к элементам вызова и различные методы управления движением действий.

Различного рода зависимости действий, как стратегии, так и цепочки, являются совокупностями действий, связанных друг с другом посредством флагов. Например, цепочка действий combo может быть представлена (в общем виде) как массив действий, у каждого из которых есть флаг combo[i], где i – порядок действия в цепочке. В неразрывном варианте цепочки порядок действия предопределён, поэтому все её действия всегда помещаются в очередь в одном и том же порядке. Также, в зависимости от неразрывности цепочек, первое действие в них имеет разное значение: в неразрывном варианте постановка и выполнение первого действия означает постановку и выполнение в строгом порядке всех действий цепочки. В классическом же варианте нестрогой цепочки данное правило справедливо только для первого действия в цепи; в случае смещений других действий в ранжировании остальные элементы цепи рассматриваются как отдельные действия.

Ранжирование действий может происходить в зависимости от выбранного при реализации метода сортировки. Так же различно можно реализовать и факт исполнения или прерывания действия NPC.

# Глава 3

# Разработка и реализация ПО метода

## 3.1 Структура, функции, модули продукта Action

Программный продукт Action (здесь и далее – ПП Action) – разработанное мной программное средство реализации метода Action First. Action выполняет следующие функции:

* Организация действий NPC в очереди действий;
* Ранжирование и упорядочивание действий в очереди;
* Постановка на исполнение, прерывание и отмена действий NPC;
* Визуализация очередей действий и порядка их исполнения для выбранных персонажей;
* Ведение истории исполнения действий персонажами.

Action работает во внутреннем режиме. В нём библиотека ядра продукта Action.Core используется **неотделимо** от игры, подключаясь внутри неё. Поэтому все алгоритмы и методы подхода могут быть использованы непосредственно. В случае необходимости отладки визуализатор Action.Queue может получать отладочные данные непосредственно из Core. Данный режим удобен, когда необходимо применять подход Action First при разработке игры с самого начала.

Action состоит из трех приложений:

* Action.Core. Является основным приложением Action. Action.Core реализован в виде динамической библиотеки, содержащей все классы-реализации объектов Action First и все методы и алгоритмы ранжирования, определения и сопоставления действий. Поэтому Action.Core предназначено для непосредственной обработки действий NPC.
* Action.Queue. Данное приложение является интерфейсом ПП Action, поскольку визуализирует очереди действий для персонажей и сохраняет историю их исполнения в log-файлы. Action.Queue обрабатывает результаты работы Action.Core, визуализируя их на форме. Таким образом, Action.Queue является как средством визуализации работы продукта, так и средством отладки при проектировании поведения NPC.
* Action.Patch. Предоставляет данные для отображения приложению-визуализатору и дает возможность библиотеке ядра получать действия персонажей из ресурсов игры и обрабатывать их.

Общая структура ПП Action представлена на рисунке 1.

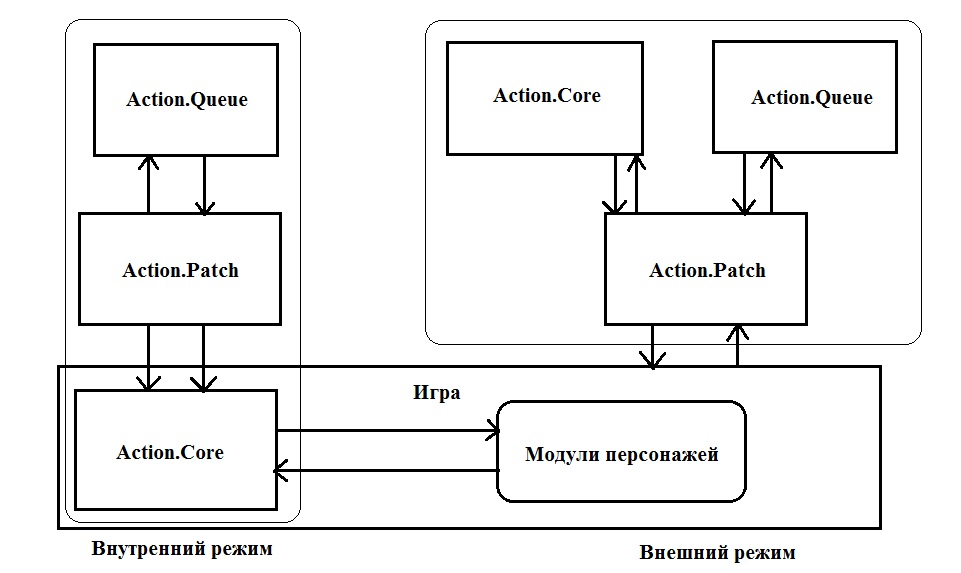


Рисунок 1 – Общая структура ПП Action

Для реализации данного продукта мною был выбран язык C# в среде .Net по следующим причинам:

* необходимость в универсальной среде разработки локальных приложений;
* необходимость высокоуровневого программирования организационных процессов Action First;
* поддержка единообразного межпрограммного взаимодействия;
* кроссплатформенность, универсальность;
* возможность организации автономных программных модулей для доступа к ним напрямую во внутреннем режиме;
* универсальность использования методов в любых обрабатываемых играх.

## 3.2 Методы Action First в библиотеке Action.Core

Action.Core – библиотека, предназначенная для реализации объектов, методов и алгоритмов Action First. В данном разделе описаны принципы их реализации.

**3.2.1 Действия**

Действие представлено классом Action. Он имеет следующий вид:

//character's action

public class Action

{

//action's method

object handler;

//context

public Context context=new GlobalContext();

//status

public Status status=Status.Idle;

//priority

public Priority priority;

//last priority

public Priority lastPriority;

}

Его свойства имеют следующие значения:

* handler – поле метода, который является функциональной реализацией данного действия. Например, если действие обрабатывает «шаг влево» для некоторого NPC, в поле Handler будет содержаться обработчик данного действия. При инициализации класса Action это обязательное поле.
* context – значение контекста, к которому принадлежит действие. По умолчанию содержится в контексте GlobalContext, объединяющем все действия игры (подробнее о контекстах см. подраздел 2.2).
* status – перечисление, из допустимых значений которого Statuses выбирается текущий статус данного действия. Статус изменяется извне действия, при инициализации класса он всегда равен Idle (то есть, действие не участвует в очереди и не выполняется, но доступно для любых манипуляций).
* priority – поле приоритета действий. Это поле имеет класс Priority, предназначенный для хранения значения приоритета и значений его составляющих.
* lastPriority – поле приоритета действий, в котором сохраняется приоритет предыдущего такта игры. Необходим для расчета дельта-параметра приоритета и, таким образом, расчёта дивергенции при пороге перемещения данного действия в очереди.

Для статуса Action применяется следующее перечисление:

//Action's statuses

public enum Status: byte

{

Available, //default -can be used in queue

Initiated, //standed ints queue

Denied, //cannot be used in queue

Completed, //completed and revoked

Processed, //processing now

Revoked, //revoked, implementation denied for this tick

Canceled, //canceled

ParProcessed //processing if parallel implementation is allowed

}

Таким образом, статусы имеют следующий смысл:

* Available – действие доступно;
* Initiated – поставлено в очередь;
* Denied – исключено из очереди;
* Completed – выполнено (и возвращено в очередь);
* Processed – выполняется (на данном такте);
* Revoked – выполнено, но исключено из очереди;
* Canceled – выполнялось ранее, но на данном такте отозвано;
* ParProcessed – выполняется параллельно (если доступно параллельное исполнение);

**3.2.2 Контексты**

Контексты в Action реализованы классом Context. Данный класс необходим для того, чтобы путём использования данного шаблона ограничить круг выбора действий из всех имеющихся в обработке.

public abstract class Context

{

public int Value { get; set; }

}

У класса Context есть один наследник GlobalContext; его инициализация происходит при инициализации всех объектов Action First. Данный класс обозначает один глобальный контекст, включающий все действия игры. Поэтому, если нет необходимости или возможности создать новые контексты, GlobalContext используется как шаблон во всех действиях и операциях с ними. GlobalContext имеет следующую структуру:

public class GlobalContext:Context

{

public GlobalContext()

{

//Global Context has his value 0 as a default context

//Increase it for your custom contexts

Value = 0;

}

}

При работе во внутреннем режиме, есть возможность создания дополнительных контекстов. Для этого разработчик должен предпринять следующие действия:

* Определить, какие действия входят в данный контекст;
* Создать класс контекста по образу GlobalContext, указав класс Context в качестве его родителя;
* Использовать класс контекста в качестве шаблона для всех действий, которые в него входят.

Подробнее данный механизм иллюстрирован в подразделах 3.2.6 и 3.2.7.

**3.2.3 Приоритеты**

Приоритеты реализованы в качестве класса Priority.

//Priority

public class Priority

{

//value of priority

public float OriginalValue { get; set; }

//base value depending on action

float baseValue { get; set; }

//status effects

float StatusEffect { get; set{

value=0;

} }

//offset effect

float offsetValue { get; set; }

//divergency value

float divergency { get; set{

value = 0;

}}

//delta parameter

float deltaParam { get; set; }

}

Здесь использованы следующие значения:

* baseValue – базовое значение приоритета для действия. Инициализируется в самом начале. Необходимо установить данное значение для каждого обрабатываемого действия, чтобы получить его для приоритета (подробнее см. пункт 3.2.6).
* StatusEffect – эффект от статуса, влияющий на приоритет. Разные статусы оказывают различный эффект на приоритет действия. Для значения по умолчанию (статус Idle) значение StatusEffect равно нулю.
* offsetValue – значение, указывающее на смещение данного действия в своей очереди. Если действие исключено из очереди, равно нулю, так же, как и при стабильном его положении. Положительное значение offsetValue указывает на смещение действия в очереди ближе к исполнению, отрицательное – дальше от исполнения. Таким образом, положительное значение offsetValue повышает приоритет действия, тогда как отрицательное его снижает.
* divergency - значение, отражающее рассчитанный показатель дивергенции Кульбака на данном и предыдущем такте игры.
* deltaParam – дельта-параметр приоритета.

**3.2.4 Очереди действий**

Очереди действий описываются классом Queue.

//a character queue

public class Queue

{

//All actions in the queue

List<Action> Actions;

//limit for priority value

float deltaLimit{ get; set; }

//allows parallel implementation

bool allowParallel;

}

У класса Queue есть следующие свойства:

* Actions – основное свойство очереди, являющееся списком действий. Поведение данного списка отличается от изначально существующего в C# и регулируется иными алгоритмами Action.Core.
* deltaLimit – текущий порог приоритета действий в очереди. В отличие от естественного нижнего порога приоритета (равного 0), текущий порог может варьироваться в зависимости от того, какие действия стоят в очереди и допускает ли очередь параллельное исполнение.
* allowParallel – флаг, показывающий, допускает ли очередь параллельное исполнение. В случае, если флаг установлен, в очереди могут присутствовать несколько действий со статусом Processed или Completed. Если данный флаг не установлен, может существовать только одно действие с данным статусом.

**3.2.5 Цепи и стратегии действий**

Зависимости действий, как уже было сказано в подразделе 2.2, определяются взаимосвязями на уровне статуса. С программной точки зрения, действия, являющиеся элементами той или иной зависимости, ничем по функционалу не отличаются от остальных действий, однако необходимо сохранить данную зависимость. Данная задача в Action.Core решена с помощью атрибутов.

В Action.Core реализованы атрибуты Chain(id, number) и Strategy(id, number). Данные атрибуты реализованы следующим образом:

[System.AttributeUsage(AttributeTargets.Method | AttributeTargets.Event, AllowMultiple=true)]

public class Chain : System.Attribute

{

//id of chain

public int id;

//true if this cain can be interrupt; otherwise false

public bool isInterruptable;

//number of target action in chain

public int number;

public Chain(int id, int num, bool interruptable)

{

this.id = id;

this.number = num;

this.isInterruptable = interruptable;

}

}

В данной реализации поле id хранит номер цепи действий (назначается на усмотрение разработчика, т.к. общее число типов цепей действий в игре неизвестно заранее). Поле number содержит номер действия в цепи. Флаг isInterruptable показывает, может данная цепь действий прерываться (флаг установлен) или нет (не установлен).

Стратегия действий имеет тот же вид атрибута Strategy, что и для цепи действий, однако в ней отсутствует флаг isInterruptable. Это обусловлено тем, что стратегия может быть прервана в любой момент, когда это будет нужно.

Разработчик игры может использовать данные атрибуты для методов и событий, которые обозначены как действия. Для их работы необходимо, чтобы целевое действие имело атрибут Action (подробнее о причинах см. в п. 2.7). Данные атрибуты можно применять несколько раз к одному и тому же действию, так как одно и то же действие может быть элементом нескольких цепей и стратегий одновременно.

**3.2.6 Персонажи**

Несмотря на то, что Action First рассматривает методы без привязки к конкретным NPC, вследствие объектно-ориентированной модели реализации персонажей в играх необходимо реализовать понятие персонажа в Action.Core. Данное понятие реализовано с помощью класса Character.

//a character

class Character

{

//activity flag

public bool isActive;

//a character's queue

public Queue queue;}

У класса Character есть следующие поля:

* Queue – очередь действий персонажа;
* isActive – флаг, который отражает необходимость обработки действий данного персонажа с помощью Action.Core. Если он установлен, обработка действий в очереди происходит в установленном порядке; в противном случае (например, вследствие смерти персонажа или переключения его на управление игроком) обработка действий прекращается.

Таким образом, видно, что данный класс является контейнером для очереди. Справедливо, что очередь не идентифицирует персонажа, хоть и недопустимо, чтобы персонаж имел несколько очередей действий. Кроме того, существование класса персонажа позволяет визуализатору Action.Patch обращаться к очереди действий NPC напрямую, а не опосредованно.

**3.2.7 Подключение библиотеки во внутреннем режиме**

Для обработки персонажей и их действий во внутреннем режиме необходимо обозначить их как пригодных для Action.Core. Для этого существуют следующие атрибуты:

* [Action] – атрибут для действия. Устанавливается только для метода или функции персонажа. Означает, что данное действие может быть поставлено в очередь и исполнено в порядке, определяемом Action.Queue.
* [Сharacter] – атрибут для персонажа. Позволяет считать помеченный данным атрибутом класс как NPC, для которого существует очередь из помеченных [Action] действий. В Action.Queue становится возможным вести журнал действий данного NPC и отслеживать его поведение в интерфейсе приложения.

Атрибуты Action и Character имеют схожую структуру и реализованы следующим образом:

[System.AttributeUsage(System.AttributeTargets.Method | AttributeTargets.Event)]

public class Action : System.Attribute

{ }

[System.AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]

public class CharacterAttribute: System.Attribute

{

public string name;

public CharacterAttribute(string name)

{

this.name = name;

} }

Видно, что атрибут Action не имеет параметров, потому что служит атрибутом-флагом, показывающим, что помеченный им метод является действием. Для атрибута Character предусмотрено поле name, чтобы идентифицировать персонажа понятным для пользователя именем (в отличие от действий, которые не уникальны, при отслеживании персонажа необходимо понимать, какой именно персонаж действует).

Упомянутые выше (в подразделе 2.5) атрибуты [Chain] и [Strategy] также необходимы для того, чтобы обозначить цепи и стратегии действий. Вследствие возможности указания данных атрибутов с параметрами номера зависимости и порядком в зависимости, а также (для атрибута Chain) прерываемости цепи, разработчик может конфигурировать с их помощью зависимости действий.

Действия и персонажи, атрибуты для которых не указаны вышеописанным способом, во внутреннем режиме не обрабатываются Action.Core. Это позволяет как отдельное тестирование поведения персонажей с подходом Action First, без необходимости повторной сборки игры или внесения в неё глобальных изменений, так и организует «гибридный» режим работы подхода (поскольку, как уже упоминалось в главе 2, некоторые персонажи могут и не подвергаться модификации).

Действия, которые помечены атрибутами Chain и/или Strategy, но не помечены атрибутом Action, также не воспринимаются как действие и не могут быть обработаны в данном случае. Это необходимо учитывать при принятии решения, из каких действий нужно конфигурировать цепи и стратегии.

**3.2.8 Внутренние операции с приоритетами действий**

Значения приоритетов действий изменяются с каждым тактом игры. Поэтому необходимо за каждый такт вести пересчет приоритетов.

Для того, чтобы реализовать такое поведение, необходим пересчет значения OriginalValue приоритета не при обращении к его действию, а при изменении его компонент. Таким образом, изменять значения компонент приоритета могут различные функции библиотеки, а сам пересчет организован с помощью событий.

В классе Priority имеются следующие события:

* onValueChanged – событие изменения свойства OriginalValue приоритета. Возникает при общем пересчете значения.
* onOffsetChanged – событие изменения компонента смещения у приоритета. Наиболее часто вызывается при ранжировании действий и смещении некоторых из них относительно начала или конца очереди. У события есть также вспомогательная функция контроля значения компонента: поскольку он принимает целочисленные значения от 1 до -1 (включая ноль). Событие вызывает метод, который правит данное значение, оставляя лишь его знак.
* onStatusChanged -событие изменения статуса действия. Оно предназначено для пересчета эффектов статуса для аддитивной части приоритета.
* onDeltaParamChanged – событие изменения дельта-параметров. Вызывает собой перерасчет всей аддитивной части, включающей в себя пересчет дивергенции Кульбака с учетом нового значения дельта-параметров на текущем шаге [44].
* onDeltaLimitChanged – событие изменения предела дельта-параметров. Вызывает перерасчет приоритетов с точки зрения их лимитов.

У класса Priority имеются следующие методы:

* Calculate()– расчет значения приоритета по формуле приоритета и помещение результата в свойство OriginalValue. Используется всегда, когда необходим пересчет приоритета, независимо от того, происходило ли обращение к действию.
* AddFlag() – статический метод управления статусом действия. Если из-за пересчета приоритета действие потеряло свой последний статус (например, его необходимо отозвать с выполнения, и статус Processed должен измениться на Revoked), необходимо использовать данный метод.
* Offset() – метод, осуществляющий независимое от вызова действия ранжирование в очереди по новому вычисленному приоритету. Подробнее о нем см. в подразделе 2.12.
* CalculateAdditional() – метод, необходимый для расчета аддитивной части приоритета. Является вспомогательным для метода Calculate() и включает в себя расчет новых дельта-параметров.

**2.10 Пулы действий**

Пулы действий предназначены для иллюстрации доменов действий персонажей. Пулы, так же как и очереди, являются списками, куда помещаются все действия, не участвующие на данный момент времени в очереди действий.

Пулы действий реализованы как глобальные объекты типа List<Action>, похожие на очереди действий как поля персонажа, однако с ними не проводятся операции на исполнение.

К пулу обращаются некоторые операции с действиями, например, операция исключения из очереди добавляет в пул удаленное из очереди действие, а операция постановки в очередь – наоборот, принимает его.

**2.11 Операции с очередью действий**

Методы, относящиеся к классу Queue, предоставляют функционал для постановки действий в очередь, удаления из очереди, выполнения и смещения действий.

Реализованы следующие методы постановки действий:

* AddAction() – метод постановки действия из пула в очередь. Присваивает флаг Initiated действию.
* CancelAction() – обратная операция перемещения действия в пул, если оно было отозвано.
* Start() – функция начала выполнения действия в очереди.
* Update() – статическая функция обновления состояния очереди, когда действия ранжируются или меняют свои статусы
* Revoke() – функция отзыва действия от исполнения, с отзывом из очереди или без него.
* ChangeFlag() – функция изменения статуса действия. Вызывается как вспомогательная для предыдущих, так как изменение статуса действия является последствием от операций с действиями.
* Validation() – функция проверки состояния очереди. Является служебной, предназначена для разрешения исключительных ситуаций.

**2.12 Ранжирование действий**

Ранжирование действий выполняется сразу после пересчета приоритета для какого-либо из них. Для этого в классе Queue существует метод Offset(action), сдвигающий действие-аргумент на позицию согласно приоритету. Важно понимать, что несмотря на эту перестановку, данное смещение никогда не влияет на выполняющиеся действия на данном такте.

Метод Offset имеет следующие шаги к выполнению:

1. Сравнение приоритета выбранного действия с соседними. Если приоритет оказался больше чем у предыдущего действия в очереди, будет сдвиг ближе к началу очереди; если меньше следующего – сдвиг к концу.
2. Алгоритм сравнивает приоритет выбранного действия с приоритетами всех действий в выбранном направлении. Если окажется, что в стороне увеличения приоритет некоторого действия оказался больше (в стороне уменьшения – меньше) – алгоритм запоминает предшествующую позицию.
3. Действие вставляется в очередь на данную позицию. Все действия, которые были смещены со своих позиций, пересчитывают коэффициент смещения со знаком в зависимости от направления перестановки.

Функция Offset имеет следующий вид:

public static int Offset(List<Action> queue, Action action)

{

try

{

float argValue = action.priority.OriginalValue;

int position = queue.IndexOf(action);

int i = 1; int designatedposition = 0;

//choose direction

if (queue[position - 1].priority.OriginalValue < argValue)//to top

{

while (queue[position - i].priority.OriginalValue < argValue)

{

i++;

}

designatedposition = position - i + 1;

i = 0;

//change offset parameters for other actions

for (int j = position; j < designatedposition; j++)

{

queue[j].priority.offsetValue = 1;

}

}

else if(queue[position+1].priority.OriginalValue>argValue)//to bottom

{

while (queue[position + i].priority.OriginalValue < argValue)

{

i++;

}

designatedposition = position + i - 1;

i = 0;

//change offset parameters for other actions

for (int j = designatedposition; j > position; j--)

{

queue[j].priority.offsetValue = -1;

}

}

//insert offset action

queue.Remove(action);

queue.Insert(designatedposition, action);

return 0;

}

catch

{

return -1; }

**2.13 Асинхронные операции с действиями**

Операции отзыва, постановки на исполнение или отмены действий являются задачами асинхронного программирования. Необходимо иметь в виду, что часто методы персонажей в игре не предусматривают манипуляции [5] с их внутренней структурой во время выполнения. Поэтому, данные операции нуждаются в оболочках, предусматривающих решение данной проблемы.

Для решения данной задачи использована концепция структуры Coroutine из движка Unity 3D [45]. Они подразумевают разделение времени выполнения на элементарные промежутки, во время которых возможна пауза или отмена выполнения функций.

Данная концепция реализована в методах Start(), Revoke(), Update(). Каждый из них имеет оболочку следующего вида [46]:

//asynс check for proximity of executing priorities

IEnumerator DoCheck() {

for(;;) {

ProximityCheck;

yield return new WaitForSeconds(.1f);

}

}

//allows start, stop & update this function

function ProximityCheck() {

for (int i = 0; i < enemies.Length; i++) {

if (Vector3.Distance(transform.position, enemies[i].transform.position) < dangerDistance) {

return true;

}

}

return false;

}

## 3.3 Визуализация очередей действий в Action.Queue

Action.Queue – приложение, являющееся интерфейсом продукта Action. Данное приложение отображает персонажей, очереди действий и значения их приоритетов во временном интервале своей сессии, а также ведет историю использования действий. Исходя из этого, разработчик может видеть причины (в виде приоритетов и их изменений), по которым конкретный персонаж предпринял то или иное действие.

Большая часть логики Action First реализована в Action.Core, поэтому Action.Queue отображает лишь результаты обработки действий и очередей. Однако, приложение также работает как средство отладки, сохраняя телеметрические данные о действиях и предоставляя их разработчику в автономном режиме.

Action.Queue не является обязательной частью продукта Action, без которой продукт не сможет функционировать. Однако без него понять, отследить и проанализировать работу Action.Core не представляется возможным.

Интерфейс Action.Queue представлен на рисунке 3.

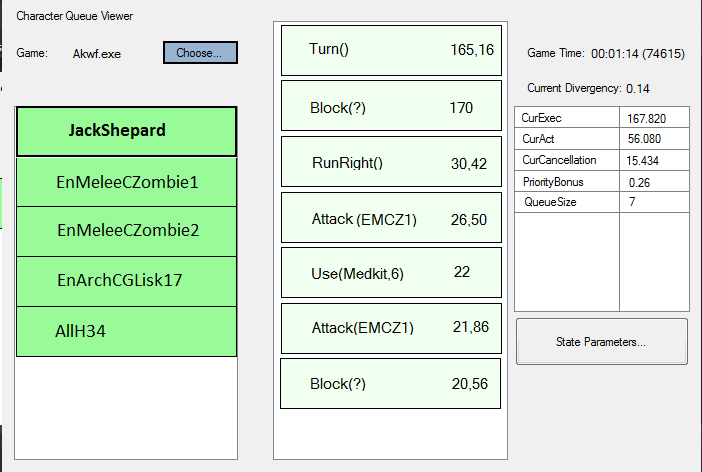


Рисунок 3 – Интерфейс Action.Queue

В текущей версии Action.Queue не предусмотрена возможность редактирования, а также дозагрузки персонажей, очередей, действий и приоритетов (в том числе и из внешнего источника). Причиной этого является непредсказуемость последствий редактирования информации внутри процесса игры: её изменение может повлечь за собой последствия разной степени опасности и для игры, и для всей операционной системы пользователя. Однако, приложение отлично работает с остальными модификациями игры (если они есть), обрабатывая их данные наряду с оригинальными данными самой игры.

**3.1 Режимы работы Action.Queue**

Приложение Action.Queue работает в двух режимах, независимо от того, во внешнем или внутреннем режиме работает весь продукт.

* Фоновый режим. В нём осуществляется запись истории действий и общее отслеживание персонажей и очередей. Фоновый режим является основным режимом работы приложения. Он применяется одновременно с запуском игры; если оконные настройки игры допускают переключение между окнами в текущей операционной системе, пользователь может одновременно играть и просматривать отладочную информацию. В случае, если игра не позволяет оперировать иными приложениями в свой сеанс, пользователь может просмотреть отладочную информацию после окончания игры в автономном режиме работы.
* Автономный режим. В нём Action.Queue не требует включенной игры. Поэтому сбор отладочной информации в данном режиме невозможен. Автономный режим предназначен для просмотра ранее собранной информации и загрузки сохраненных ранее журналов действий.

Action.Queue меняет режим работы в зависимости от состояния игры, персонажи которой отслеживаются. Например, если во время сеанса игры приложение работало в фоновом режиме, по окончанию игры сбор данных о действиях становится невозможным, и пользователь может использовать Action.Queue только в автономном режиме. Например, для просмотра собранных данных. Обратная ситуация – переход в фоновый режим из автономного – случается при запуске отслеживаемой игры; однако во внешнем режиме функционирования Action необходима работа Action.Patch, чтобы приложение получило доступ к отслеживаемой игре.

**3.2 Отображение персонажей и действий**

Для того, чтобы получить доступ к персонажам, их действиям и приоритетам, необходимо выбрать игру, к которой необходимо получить доступ. Для этого необходимо нажать кнопку Choose в левой верхней части интерфейса. Откроется стандартное диалоговое окно, представленное на рисунке. Необходимо выбрать приложение или просто написать имя процесса в диалоговой строке.

Выбор игры реализован в виде поиска одноименного процесса в операционной системе. В случае, если данный процесс существует (игра уже запущена), приложение будет подключено к ней и инициирует работу прослушивателя событий (Event Listener, здесь и далее EL). Если процесс отсутствует в системе, Action.Queue также инициирует EL, но останется в автономном режиме до тех пор, пока в системе не появится процесс с указанным именем и не будет совершено подключение к нему.

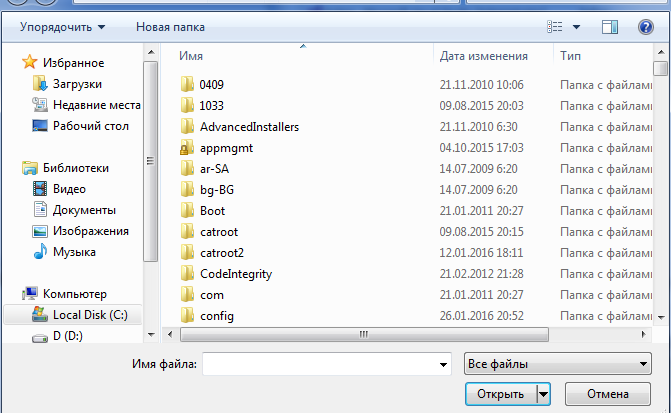


Рисунок 3 – Окно выбора игры

Пользователь всегда может изменить свой выбор игры путем повторного вызова данного окна и изменения имени процесса игры. При подобной операции в фоновом режиме, уже накопленная информация будет сохранена в соответствующий лог-файл, а отображение данных будет обновлено в соответствии с полученными данными из нового процесса.

Возможно также удаление имени: в таком случае, приложение будет всегда запускаться в автономном режиме. До тех пор, пока новое имя процесса не будет введено, будет доступен только просмотр истории действий, но не её сбор.

Персонажи отображаются на вкладках формы. На каждой вкладке размещается список, представляющий собой очередь действий.

При нажатии на элемент списка (действие) в правой части интерфейса появляется диаграмма приоритета для данного действия.

**3.3 Отображение диаграммы приоритетов**

Диаграмма приоритетов предназначена для отслеживания приоритетов действий в очереди по времени. Используя данную диаграмму, возможно определить поведение персонажа и, таким образом, судить об эффективности его программирования и настройки.

Пример диаграммы приоритетов представлен на рисунке.

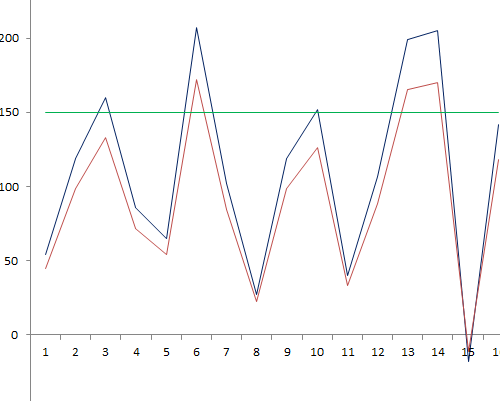


Рисунок 4– Пример диаграммы приоритетов

Обозначенные цифрами участки имеют следующее значение:

1. Синяя ломаная линия является зависимостью величины приоритета от времени.
2. Красная ломаная линия означает отклонение дельта-параметров на каждом такте (характеризует скорость увеличения или снижения приоритета действия).
3. Зеленые линии означают минимальный и максимальный лимиты приоритетов.

По диаграмме приоритетов можно отследить следующие факты о действии:

* Действие было поставлено в очередь с первым же тактом отслеживаемого периода в игре. Его приоритет возрос, и действие могло бы быть поставлено на выполнение – однако рост дельта-параметра оказался недостаточным для того, чтобы это произошло (показатель дельта-параметра не превысил верхний лимит приоритета). Поэтому на тактах 4-6 действие было отклонено (вероятно, за счет выполнения других действий с более высокими приоритетами).
* Первый пик превышения означает, что действие было поставлено на выполнение на такте 6. Однако быстрое уменьшение его приоритета на последующих тактах говорит об отзыве действия от исполнения. По природе диаграммы, чем более «плоский» пик превышения лимита – тем дольше выполнялось действие в очереди.
* Максимум приоритета достигнут на такте 14. Вершина данного пика пологая, что говорит о выполнении действия на тактах 13 и 14.
* На такте 15 значение приоритета ниже, чем лимит отзыва, равный нулю. Это говорит о том, что аддитивный параметр имел отрицательный знак и значительно превысил базовое значение. Такая ситуация говорит о том, что действие было отозвано сразу после окончания выполнения (на такте 14).

На практике, особенно для real-time игр, диаграммы приоритетов содержат намного больше тактов, чем в данном примере (1 секунда=300 Tick() в Unity 3D). Поэтому, удобнее смотреть подобные диаграммы с помощью Action.Queue (производящего по данным построение диаграмм), а не вручную.

**3.4 Сохранение, просмотр и загрузка историй действий**

Если рабочий процесс игры, отслеживаемый Action.Queue, завершается (осуществляется выход из игры), в приложении происходит автосохранение собранной истории действий в файл с расширением .csv. Данный файл является текстовой таблицей, хранящей данные о персонажах, очередях действий и приоритетах. Структура .лог-файла представлена на рисунке.

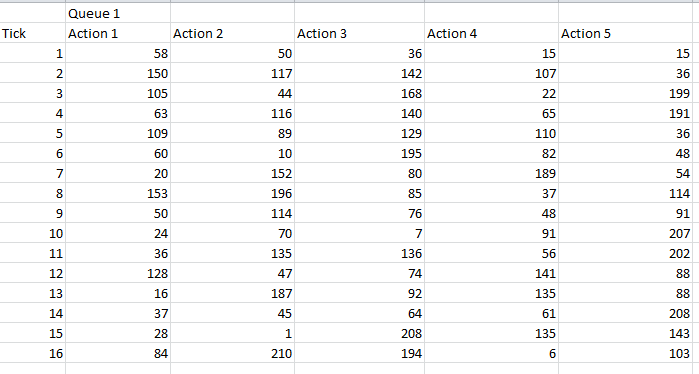


Рисунок – Структура лог-файла Action.Queue

Даже по чистому лог-файлу можно судить о том, как по времени были распределены действия в очереди. Например, на втором такте игры действие 5 было на первом месте очереди (но, судя по третьему такту, было на нём отменено).

Ранее созданные лог-файлы можно загружать в приложение посредством нажатия кнопки Load Action History. При этом происходит загрузка данных и формируются вкладки персонажей со списками очередей действий. При нажатии на действие из файла извлекается информация о его приоритетах и строится диаграмма приоритетов.

Существует возможность выбора директории, куда приложение сохраняет лог-файлы. Для этого необходимо нажать кнопку Log Files Into… и выбрать желаемую директорию.

# Заключение

Разработанный в диссертации метод имеет следующие преимущества перед уже используемыми в оптимизации NPC:

* использование принципов системного подхода, позволяющее использовать централизованную архитектуру продуктов Action First и удобство подключения к игре;
* исключение модификации скриптов и рабочего функционала NPC при внедрении (хотя, если случай конкретной игры того требует, может потребоваться разграничение составных действий на несколько элементарных);
* гибкость описания действий и взаимодействий персонажа с помощью численных характеристик;
* частичная автоматизация проектирования поведения персонажей, когда разработчику не нужно проектировать всю схему поведения NPC по времени, ограничившись функционалом зависимостей и действий;
* гибкость настройки поведения персонажа, путем настройки коэффициентов статусов, зависимостей и взаимодействий в игре;
* гибкость настройки, независимость от принятой в игре структуры рабочего проекта, объектной и временной системы;
* предусмотренная возможность для сбора и анализа данных по применениям действий NPC, что даёт разработчику возможность оценивать поведение персонажей экспериментально, а не эмпирически.

Метод также имеет средство реализации в виде продукта Action, который можно использовать непосредственно в игре. Несмотря на то, что сам продукт может требовать дополнительной доработки и настройки для каждой конкретной игры (в зависимости от того, насколько принципы Action First сопоставимы с принципами реализации этой игры), данный продукт является полезной составляющей модуля управления NPC.

Также, существуют своего рода перспективы для дальнейшего развития метода:

* Наличие возможностей улучшения описания факторов влияния действия. В настоящий момент для описания нечисловых характеристик факторов используются соответственные численные характеристики. Однако, данный подход внедрён из соображений простоты и невозможности полного описания семантических параметров. Следовательно, одна из перспектив состоит в нахождении подобных способов (возможно, в разработке собственного символьного алфавита описания, как вариант).
* Наличие возможности использования теории информации для более глубокой настройки поведения персонажа. Дивергенция Кульбака-Лейблера используется в методе как мера различия между ожидаемым и реальным приоритетом действия, и выступает, таким образом, мерой гладкости функции приоритета по времени. Задача о минимизации дивергенции, таким образом, есть задача сглаживания различий, а сама величина дивергенции может по-разному влиять на расчёт приоритета в методе, и, как итог – по-разному влиять на поведение персонажа.
* Возможность использования «гибридной» модели метода. В играх часто применяются глобальные объекты-регулировщики событий, помимо объектов, предлагаемых в методе. Возникает закономерная проблема их совместного использования. На данный момент, если организационная модель и родного для метода объекта, и стороннего объекта использует один и тот же функционал персонажа, возможны сбои в работе обоих. Следовательно, необходимы дополнительные работы в поддержании режимов совместимости между глобальными объектами игры (например, введение стандарта совместимости, при котором действия, которые регулируются Action First, не применяются одновременно в других модулях, которые инициируют или отменяют их выполнение)
* Возможность использования внешнего режима работы. Игры имеют широкое разнообразие способа сборки и размещения на пользовательском устройстве. Поэтому модули, с помощью которых они действуют, могут быть по-разному организованы при разных способах размещения и в разных операционных системах. Максимально кроссплатформенное решение, таким образом – внутреннее использование библиотеки методов Action внутри самой игры, как дополнительного ресурса игры. В течении дальнейших работ над продуктом одной из основных задач станет возможность внешнего режима работы, в качестве модификации.
* Возможность комплексного сбора информации. Благодаря тому, что в лог-файлах и в приложении визуализации выводится только информация о приоритетах, эти данные можно сделать более понятными более широкому кругу разработчиков, не имеющих непосредственную квалификацию разработчиков ИИ NPC. Например, можно рассматривать в ходе работ над проектом перспективу сбора всех метаданных персонажа, включая полный журнал расчёта его параметров в игре.

Учитывая все преимущества и перспективы Action First, данный метод вносит свой ощутимый вклад в решение вопроса о регуляции глобально поведения NPC и в частности – регуляции его действий.

# Список использованных источников

[1] Eliezer Yudkowski. Creating Friendly AI 1.0: The Analysis and Design of Benevolent Goal Architectures/Eliezer Yudkowski – .The Singularity Institute, San Francisco, CA, 2001 – 278p.

[2] Saiqa Aleem, Luis Fernando Capretz, Faheem Ahmed. Journal of Software Engineering Research and Development/S.Aleem, L.F.Capretz, F.Ahmed - Journal of Software Engineering Research and Development, 2016 – 78p.

[3] Dan Felder. Design 101: Balancing Games [Электронный ресурс]. – Режим доступа: https://www.gamasutra.com/blogs/DanFelder/20151012/251443/Design\_101\_Balancing \_Games.php

[4]

[5]

[6]

[7]

[8]

[9]

[10]

[11]

[12]

[13]

[14]

[15]

[16]

[17]

[18]

[19]

[20]

[21]

[22]

[23]

[24]

[25]

[26]

[27]

[28]

[29]

[30]

[31]

[32]

[33]

[34]

[35]

[36]

[37]

[38]

[39]

[40]

[41]

[42]

[43]

[44]

[45]

[46]

[47]

[9] Палицын, В. А. Технико-экономическое обоснование дипломных проектов. Метод. пособие для студ. всех спец. БГУИР. В 4-х ч. Ч.4: проекты программного обеспечения/В. А. Палицын. – Мн.: БГУИР, 2006. – 76 с.

[10] Delphi 7. Обоснование выбора [Электронный ресурс]. – Режим доступа: http://barsicne.narod.ru/publ/delphi\_7\_obosnovanie\_vybora/1-1-0-1

## ПРИЛОЖЕНИЕ А

## (обязательное)

## Программный модуль отображения маршрутов

public void ShowData()

{try

{

//getting an application type whish represents registry type

ПРИЛОЖЕНИЕ **Г**

## (информационное)

## Описания кодов сообщений для модуля протоколирования

| Код сообщения | Описание | Состояния | |
| --- | --- | --- | --- |
| Тип | Описание |
| 0 | Запуск приложения | 0 | Обычный режим |
| 0 | Режим просмотра |
| 1 | ЦВК недоступен |
| 1 | Выход из приложения | 0 | Отмена запуска |
| 0 | Выход пользователя |
| 3 | Аварийный выход: недоступен реестр |
| 3 | Аварийный выход: отсутствие запрашиваемого ПО |
| 2 | Сохранение параметров маршрутов в реестр | 0 | Параметры маршрута сохранены |
| 3 | Отмена сохранения |
| 3 | Запись значений параметров в реестр | 3 | Не удалось записать параметр в реестр |
| 4 | Чтение данных маршрутов из внешнего источника | 3 | Некорректная запись в ресурсе |
| 5 | Создание маршрута | 0 | Маршрут создан |
| 3 | Не удалось создать маршрут |
| 0 | Дубликат маршрута создан |
| 3 | Не удалось получить дубликат текущего маршрута |
| 6 | Удаление маршрута | 0 | Успешное удаление |
| 3 | Отмена удаления |
| 7 | Установка доступа к ЦВК | 0 | Доступ к ЦВК разрешен |
| 1 | Доступ запрещен или отсутствует его возможность, работа разрешена только с реестром |

Продолжение приложения Г

| Код сообщения | Описание | Состояния | |
| --- | --- | --- | --- |
| Тип сообщения | Описание |
| 8 | Установка прав доступа к реестру | 3 | Не удалось установить права доступа к реестру |
| 9 | Исправление ошибок настройки доступа | 3 | Не удалось исправить или откатить настройки доступа: недоступно или не установлено требуемое ПО |
| 3 | Не удалось исправить или откатить настройки доступа из-за реестра |
| 10 | Конверсия типов при доступе к реестру | 3 | Ошибка извлечения при доступе к реестру |
| 11 | Отображение параметров маршрутов | 3 | Не удалось отобразить параметр |
| 12 | Изменение параметров маршрутизатора | 3 | Не удалось сохранить параметры маршрутизатора |

1. Game Artificial Intelligence – в русском языке аналогом является понятие «игровой интеллект». [↑](#footnote-ref-1)