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

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

Допустить к защите
заведующий кафедрой
д-р тех. наук, доцент
_____ А. В. Коваленко
10.06.22 2022 г.

Руководитель ООП
д-р физ.-мат. наук, профессор
_____ М. Х. Уртенов
10.06.22 2022 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТЫ
(МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

РАЗРАБОТКА МНОГОАГЕНТНОЙ СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА ПОСРЕДСТВОМ DATA-ORIENTED DESIGN

Работу выполнил _____ К. Р. Сеидов
(подпись)

Направление подготовки 01.04.02 Прикладная математика и информатика

Направленность (профиль) Математическое и информационное обеспечение экономической деятельности

Научный руководитель
д-р тех. наук, профессор _____ А.А. Халафян
(подпись)

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

Краснодар
2022

РЕФЕРАТ

Выпускная квалификационная работа (магистерская диссертация) 55 с., 32 рис., 8 источников.

РАЗРАБОТКА ИИ, АВТОНОМНЫЙ АГЕНТ, UNITY, DATA ORIENTED DESIGN, UNITY DOTS, ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

Объектом исследование являются применение подхода Data Oriented Design для разработки многоакцентной системы искусственного интеллекта для видеоигры и его влияние на производительность такой системы.

Целью выпускной квалификационной работы является разработка многогенной системы автономных игровых сущностей, способной эффективно использовать ключевые аспекты современных процессоров, такие как многопоточность и процессорная кэш-память.

В результате выпускной квалификационной работы разработана многоакцентная система искусственного интеллекта, спроектированная и реализованная в соответствии с подходом Data Oriented Design, позволяющая осуществлять вычисления параллельно и эффективно использовать кэш-память процессора. Автономные агенты имеют следующие возможности:

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

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

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

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

Одним из наиболее сложных в плане разработки и одним из ключевых аспектов для игрока является реализация системы искусственного интеллекта в игре. Кроме того, что такая система должна обеспечивать правдоподобное поведение игровых сущностей с точки зрения игрока, она так же должна быть спроектирована с учетом дальнейшей расширяемости и наиболее эффективного использования ресурсов системы, и в первую очередь ресурсов ЦПУ.

Целью выпускной квалификационной работы является разработка многогенной системы автономных игровых сущностей, способной эффективно использовать ключевые аспекты современных процессоров, такие как многопоточность и процессорная кэш-память.

В качестве среды разработки была выбрана платформа Unity, являющаяся сегодня наиболее популярным решением при создании видеоигры. Программой код написан на объектно-ориентированном ЯП C#, а архитектура системы построена на базе дизайна, ориентированного на данные DOD (англ. Data Oriented Design), как наиболее оптимальный выбор для достижения поставленной цели в предметной области разработки видеоигр.

Для достижения поставленной цели необходимо решить следующие задачи:

– изучить принципы и особенности разработки с использованием DOD;

– изучить возможности платформы Unity поддержки применения принципа DOD;

– выбрать подходы для реализации различных компонентов системы ИИ, с учетом возможности параллельных вычислений;

– реализовать подсистемы:

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

– сравнить производительность полученной системы с применением DOD относительно системы с применением ООП;

– продемонстрировать возможности разработанной системы.

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

В рамках магистерской работы была разработана система автономных агентов для видеоигры. Под автономных агентом будет пониматься игровая сущностью, как например неигровой персонаж, использующая информацию из игровых данных, определяющая дальнейшее необходимое действие на основе полученной информации и самостоятельно исполняющая это действие. Вычислений для каждого отдельного агента могут выполняться отдельно в собственном потоке, позволяя реализовать параллельные вычисления для всего множества агентов исключая конкуренцию между ними.

Используемые методы для реализации каждой из этих компонент в рамках подхода DOD, как и сам подход, будут рассмотрены более подробно далее.

Выпускная квалификационная работа состоит из трёх разделов. В первом разделе описаны теоретические основы подхода Data Oriented Design и применяемые для его реализации программные средства. Во втором разделе рассматриваются и анализируются основные подходы и техники для реализации системы искусственного интеллекта в видеоиграх. В третьем разделе описывается структура и реализация разработанной системы,

анализируются ее преимущества относительно решения на базе принципов ООП, а также демонстрируется возможность применения полученной системы.

1 Технологии и используемые средства

1.1 Кэш ЦПУ. Проблема «промаха» кэша

Современные процессоры могут работать с тактовой частотой в несколько ГГц и способны выполнять несколько инструкций за такт. Это означает, что процессор может иметь пиковую скорость выполнения нескольких инструкций в наносекунду.

С другой стороны, современная оперативная память работает довольно медленно относительно ЦПУ (рис. 1) [1]. Доступ к ней занимает 50 нс и более, что приводит к зависанию процессора в ожидании поступления данных. Это делает доступ к оперативной памяти одной из самых медленных операций, которые может выполнять процессор.

Поскольку обращения к памяти очень распространены в программах и могут составлять более 25% инструкций, задержки доступа к памяти оказали бы разрушительное влияние на скорость работы процессора, если бы их нельзя было каким-либо образом избежать.

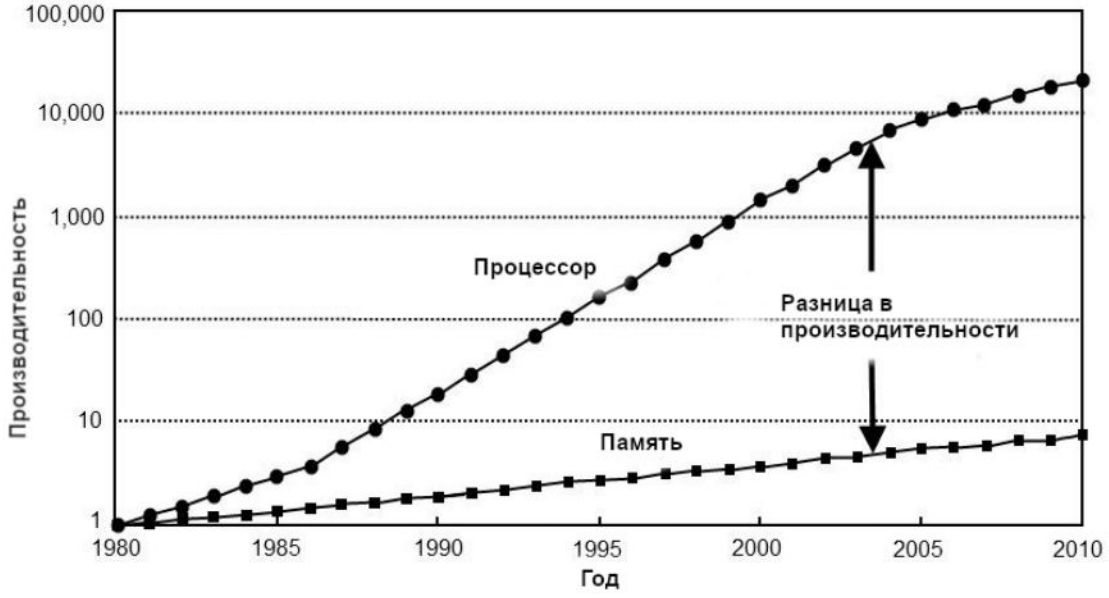

Рисунок 1 – Разница в производительности ОЗУ и ЦПУ

Чтобы решить эту проблему, была введена кэш-память, представляющая собой небольшую, но чрезвычайно быструю память между процессором и медленной основной памятью. Кэш-память, состоящая из более быстрых микросхем статической ОЗУ (SRAM), чем динамическая ОЗУ (DRAM), используемая для основной памяти, позволяет выполнять программные инструкции, считывать и записывать данные с более высокой скоростью. Инструкции и данные передаются из основной памяти в кэш фиксированными блоками, известными как «строки» кэша.

Современные чипы ЦПУ содержат от двух до трех областей кэша [1], начиная с L1, являющегося самым быстрым. Каждый последующий кэш медленнее и больше по объему, чем L1, а инструкции и данные перемещаются из основной памяти в L3, L2, L1 и далее в процессор (рис. 2).

– L1 – это самая быстрая память, которую можно найти в любом потребительском ПК. Она значительно быстрее, чем другие уровни кэш-памяти или оперативной памяти. Тем не менее, она также намного меньше по объему, так как ее производство чрезвычайно дорого. На текущий момент кэш-память L1 варьируется от 256 КБ до 1 МБ [1]. Кэш L1 обычно делится на две части: кэш инструкций и кэш данных. Кэш инструкций имеет дело с информацией об операции, которую должен выполнить ЦП, а кэш данных содержит данные, над которыми должна быть выполнена операция. Также важно отметить, что каждое ядро получает выделенный кэш L1.

– L2 может иметь в несколько раз большую емкость, чем L1 и может достигать 10 МБ. Однако она не такая быстрая, как L1, расположена дальше от ядер. Также выделяется каждому ядру в ЦПУ отдельно.

– L3 значительно больше, чем L1 и даже L2, может достигать в объеме десятков МБ. В отличие от L1, кэш L2 и L3 распределяется между всеми ядрами. Является самой медленной кэш-памятью процессора.

Распределением места в кэше управляет процессор. Когда процессор обращается к части памяти, которой еще нет в кэше, он загружает часть

памяти вокруг адреса, к которому был получен доступ, надеясь, что он скоро будет использован снова.

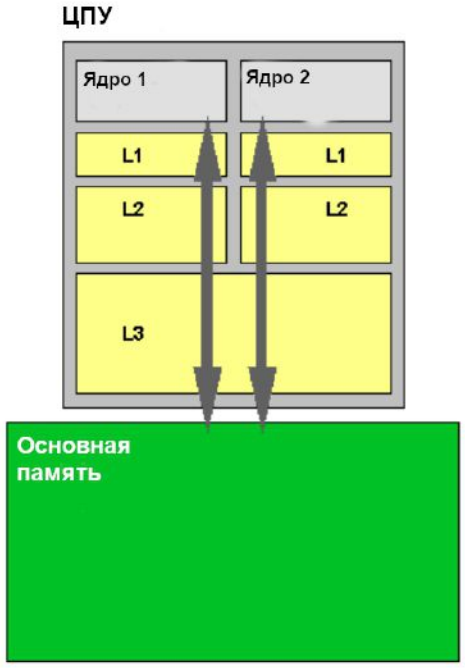

Рисунок 2 – Уровни кэша процессора

Блоки памяти, обрабатываемые кэшем, называются строками кэша. Стандартные размеры строки кэша составляют 32, 64 и 128 байт.

Когда программа обращается к ячейке памяти, которая не находится в кэше, это называется «промахом» кэша [1]. Поскольку затем процессор должен ожидать, пока данные будут извлечены из следующего уровня кэша или из основной памяти, прежде чем он сможет продолжить выполнение, промахи кэша напрямую влияют на производительность приложения.

Кэши работают исходя из предположения, что к данным, к которым обращаются один раз, обычно вскоре будут обращаться снова. Такое поведение известно как локальность данных. Иногда различают два вида локальности [1]:

– временная локальность означает, что программа повторно использует те же самые данные, которые она недавно использовала, и поэтому они, вероятно, находятся в кэше;

– пространственная локальность означает, что программа использует данные рядом с местами, к которым недавно обращались.

Хорошая локализация данных необходима для хорошей производительности приложения. Приложения с плохой локализацией данных снижают эффективность кэша, вызывая длительное время ожидания доступа к памяти. Влияние промаха кэша на производительность зависит от задержки выборки данных из следующего уровня кэша или основной памяти.

1.2 Data Oriented Design

Ориентированный на данные дизайн (англ. Data Oriented Design), далее DOD – подход к программной оптимизации, мотивированный эффективным использованием кэша ЦПУ и широко используемый при разработке видеоигр. Подход фокусируется на выстраивании структуры данных в памяти, их разделении, сортировки и преобразовании. Для сравнения процедурный, функциональные и ООП подходы в большей степени фокусируются на коде программы: процедуры (или функции) в одном случае, сгруппированный код с соответствующими внутренними свойствами в другом [2].

В DOD большое значение имеет понятие «идеальных данных». Идеальные данные – данные, находящиеся в таком формате, что их использование возможно с минимальным количеством усилий и трансформаций, а их входное и выходная структура одинаковы до и после обработки. Чаще всего, идеальные данные представляют в виде смежных блоков однотипных данных, которые возможно обрабатывать последовательно. За счет того, что DOD в своей основе ставит данные на первое место, становится возможным построить программную архитектуру вокруг концепта идеальных данных. Стоит отметить, этот концепт ничто иное как абстракция, которая далеко не всегда достижима в реальных случаях, как зачастую очень редко достигается идеальная ООП-архитектура.

Но тем не менее, в ходе разработки к этой абстракции необходимо стремиться.

Для более детального сравнения ООП и DOD в контексте работы с данными, можно рассмотреть структуру формирования данных в них.

При мысли об объектах, на ум сразу приходят деревья наследования, деревья содержания одного экземпляра в другом или деревья передачи сообщений, и, соответственно, данные в ООП так же и устроены. В результате, когда выполняется операция над объектом, это приводит к тому, что один объект получает доступ к другим объектам далее внизу по дереву. Перебор множества объектов, выполняющих одну и ту же операцию. порождает каскадные совершенно разные операции на каждом из них. Схему можно наблюдать на рисунке 3.

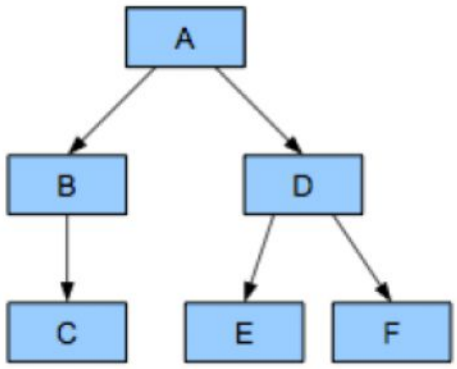

Рисунок 3 – Доступ к данным в классическом ООП

Чтобы добиться наилучшего размещения данных, полезно разбить каждый объект на различные компоненты и группировать компоненты одного типа вместе в памяти, независимо от того, с каким объектом они связаны. Эта организация приводит к большим блокам (рис. 4) однородных данных, которые позволяют обрабатывать данные последовательно. Основная причина, по которой DOO настолько эффективен, это его способность функционировать на больших группах объектов, в отличии от ООП, который чаще всего работает с одним объектом.

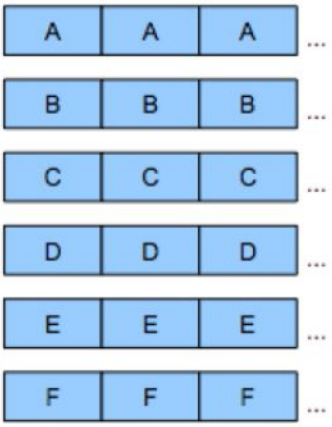

Рисунок 4 – Представление данных в DOD

Итак, при построении архитектуры с применением DOD, становится возможным более эффективное получение доступа к данным и их эффективная обработка. Помимо этого, такой архитектуре характерны следующие преимущества:

– Более естественное применение параллельных вычислений. Существуют входные независимые наборы данных, небольшая функция для их обработки и некоторые выходные данные. Таким образом, возможно легко разделить наборы на несколько потоков с минимальной синхронизацией между ними.

– Эффективное использование кэш-памяти процессора. В дополнение к использованию нескольких ядер, одним из ключей к достижению высокой производительности на современном оборудовании с его глубокими конвейерами инструкций и относительно медленным системы памяти, является эффективное использование кэша процессора. DOD ведет к очень эффективному использованию кэша инструкций ЦПУ, поскольку один и тот же программный код выполняется снова и снова для множества наборов данных. Кроме этого, если данные размещены в больших смежных блоках, становится возможным обрабатывать их последовательно, получая почти идеальное использование кэша и высокую производительность.

– Модульность и расширяемость. Когда программный код написан специально для преобразования данных, получаются компактные функции с

очень небольшим количеством зависимостей от других частей программы. Кодовая база оказывается очень «плоской» с множеством так называемых листовых функций и без большого количества зависимостей. Такой уровень модульности и отсутствие зависимостей значительно упрощает понимание, замену и расширение кода.

– Облегченные тестирование и отладка. Последним важным преимуществом проектирования, ориентированного на данные, является простота тестирования. Написание юнит-тестов для проверки взаимодействия объектов – нетривиальная задача. Нужно настроить макеты и протестировать вещи косвенно. С другой стороны, при работе непосредственно с данными писать модульные тесты становится намного проще: создаются некоторые входные данные, вызывается функцию преобразования и проверяется соответствие выходных данных ожидаемому результату.

1.3 Среда разработки Unity

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

Редактор Unity имеет простой Drag&Drop интерфейс, который легко настраивать, состоящий из различных окон, благодаря чему можно производить отладку игры прямо в редакторе. Среда поддерживает C# в

качестве скриптового языка. Расчёты физики производит физический движок PhysX от NVIDIA.

Проект в Unity делится на уровни, которые представлены отдельными файлами сцен, содержащие свои игровые миры со своим набором объектов, игровых сценариев, так называемые скрипты, и настроек. В концептуальном плане сцена представляет собой некий аналог окна из привычных оконных приложений. Сцены могут содержать в себе как объекты, содержащие модели, 2D-изображения, так и пустые игровые объекты. Они в свою очередь содержат наборы компонентов, с которыми и взаимодействуют скрипты. Также у объектов есть название, им возможно присвоить метку, так называемый тег, и слой, на котором он должен отображаться. Так у любого объекта на сцене обязательно присутствует компонент Transform, который хранит в себе координаты местоположения, поворота и размеров объекта по всем трём осям. У объектов с видимой геометрией также по умолчанию присутствует компонент Mesh Renderer, делающий модель объекта видимой. К объектам можно добавлять такие компоненты как коллайдеры (англ. collider), которые отвечают за параметры сталкивания объектов. В редакторе имеется система наследования объектов; дочерние объекты будут повторять все изменения позиции, поворота и масштаба родительского объекта. Скрипты в редакторе прикрепляются к объектам в виде отдельных компонентов.

Unity предоставляет API (англ. application programming interface) на ЯП C# для написание так называемых скриптов. Скрипт или сценарий – основной компонент разработки в среде Unity. В отличии от ситуаций, когда сценариями называют автономные служебные программы, в Unity сценарии больше напоминают классы в ООП, а присоединенные к объектам сцены скрипты являются экземплярами класса. Следует отметить, что для функционирования кода необходимо чтобы описанный в нем класс был потомком MonoBehavior – базового класса компонентов-сценариев. Этот класс определяет способ присоединения компонентов к игровым объектам.

Наследование от этого класса дает некоторые автоматически запускаемые методы, как Start(), вызываемый при активации игрового объекта, или Update() – метод, вызываемый в каждом кадре. Кроме этого, скрипты используются при разработке пользовательского интерфейса, искусственного интеллекта, различных контроллеров, отвечающих за сетевые и внутрепрограммные функции.

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

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

1.4 Технология Unity DOTS

С точки зрения удобства и предоставляемых возможностей, Unity является лидером относительно своих конкурентов, таких как Unreal Engine, Game Maker или Godot Engine. С другой стороны, для разработки серьезных и масштабных проектов Unity существенно уступает Unreal Engine в производительности, из-за большого ограничения по работе с потоками – основная часть вычислений осуществляется лишь в одном из них. Для решения этой проблемы компанией Unity Technologies разрабатывается набор пакетов технологий, образующих Unity Data Oriented Technology Stack, или сокращенно – Unity DOTS. Он содержит в себе следующие подмодули:

- Unity C# Jobs System;
- Unity Burst Compiler;
- Unity Entities.

1.4.1 Unity Job System

Unity C# Jobs System – система, позволяющая писать многопоточный программный код, который хорошо взаимодействует с остальной частью Unity

Важным аспектом Unity C# Jobs System является то, что она интегрируется с тем, что Unity использует внутри своей системы.

Клиентский код и Unity совместно используют рабочие потоки. Такое сотрудничество позволяет избежать создания большего количества потоков, чем ядер ЦПУ, что могло бы вызвать конкуренцию за ресурсы.

1.4.2 Unity Entities

Unity Entities – пакет, добавляющий в Unity возможности, позволяющие использовать архитектурный паттерн Entity Component System (ECS), основанный на подходе Data Oriented Design. ECS – это ядро стека технологий Unity, ориентированных на данные (Data-Oriented Tech Stack, DOTS). Как видно из названия, ECS состоит из трех основных частей:

– Сущности: объекты, которые заполняют приложение. Сущность не имеет ни поведения, ни данных; вместо этого она определяет, какие фрагменты данных принадлежат друг другу.

– Компоненты: данные, связанные с объектами, но организованные по типу данных, а не по привязке к сущности. Это различие в организации является одним из ключевых различий между объектно-ориентированным и ориентированным на данные дизайном (DOD).

– Системы: логика, которая преобразует данные компонента из его текущего состояния в следующее состояние. Например, система может использовать скорость сущности, умноженную на временной интервал, прошедший с предыдущего кадра, для обновления позиций всех движущихся сущностей.

Архитектура Entity Component System (ECS) разделяет отдельные элементы (сущности), данные (компоненты) и поведение (системы). Архитектура фокусируется на данных. Системы считывают потоки данных компонентов, а затем преобразуют данные из входного состояния в выходное состояние, которое объекты затем индексируют.

На рисунке 4 показано, как эти три основные части работают вместе:

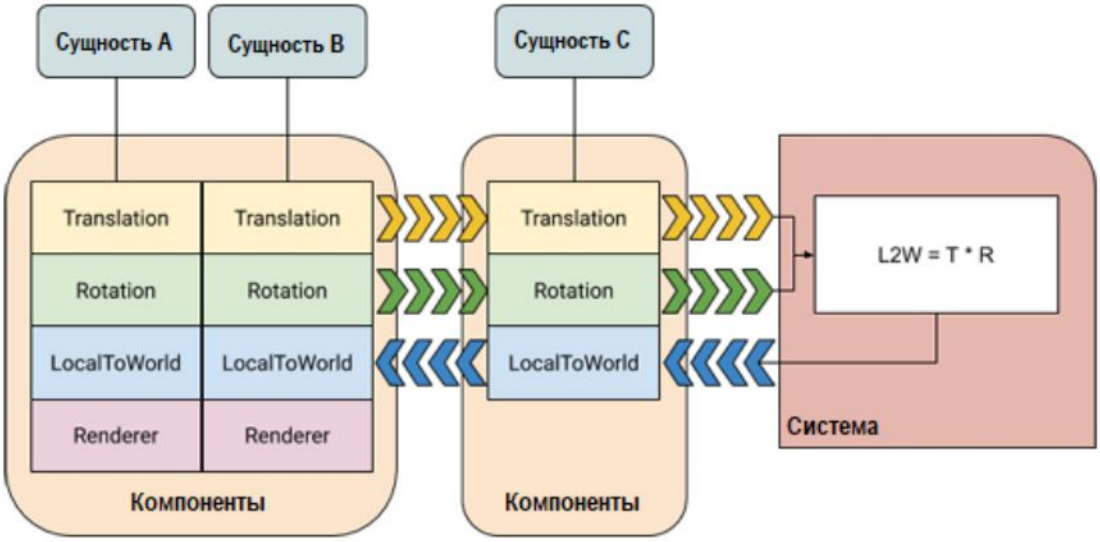

Рисунок 4 – Схема работы ECS

На этой диаграмме система считывает компоненты перемещения и вращения, умножает их и затем обновляет соответствующие компоненты LocalToWorld (L2W = T*R).

Тот факт, что у объектов A и B есть компонент Renderer (отвечает за отрисовку на экране), а у объекта C нет, не влияет на систему, потому что конкретно эта система не работает с подобными компонентами.

Возможно настроить систему так, чтобы ей требовался компонент Renderer, и в этом случае система игнорирует компоненты объекта C; или, в качестве альтернативы, можно настроить систему для исключения объектов с компонентами Renderer, которая затем игнорирует компоненты объектов A и B.

Уникальная комбинация типов компонентов называется архетипом сущности (англ. Entity Archetype). Например, трехмерный объект может иметь компонент для преобразования мира, один для линейного движения, один для вращения и один для визуального представления. Каждый экземпляр одного из этих 3D-объектов соответствует одному объекту, но поскольку они имеют один и тот же набор компонентов, ECS классифицирует их как единый архетип (рис. 5).

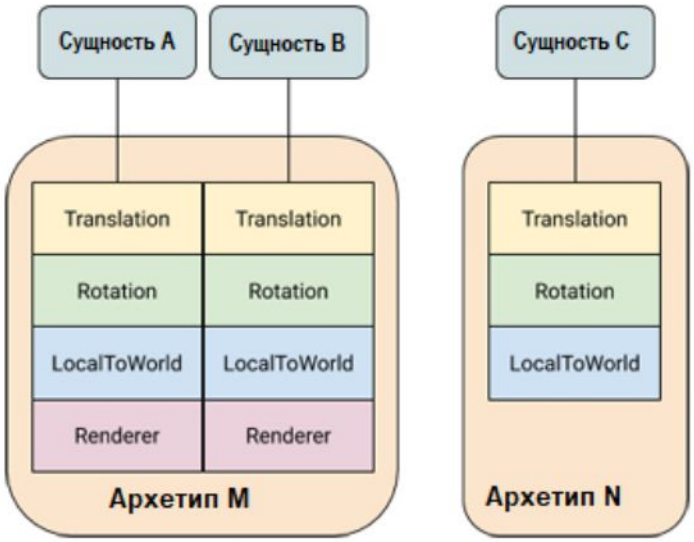

Рисунок 5 – Архетипы сущностей

На этой диаграмме сущности A и B имеют общий архетип M, а сущность C имеет архетип N.

Чтобы изменить архетип сущности, можно добавлять или удалять компоненты во время выполнения. Например, если удалить компонент Renderer из объекта B, он затем переместится в архетип N.

Архетип сущности определяет, где ECS хранит компоненты этой сущности. ECS выделяет память «блоками» (рис. 6). Блок всегда содержит сущности одного архетипа. Когда блок памяти заполняется, ECS выделяет новый блок памяти для любых новых сущностей, созданных с тем же архетипом. Если добавлять или удалять компоненты, которые затем изменяют архетип объекта, ECS перемещает компоненты для этого объекта в другой фрагмент.

Эта организационная схема обеспечивает отношение «один ко многим» между архетипами и блоками. Это также означает, что поиск всех сущностей с заданным набором компонентов требует поиска только среди существующих архетипов, число которых обычно невелико, а не среди всех сущностей, число которых обычно гораздо больше.

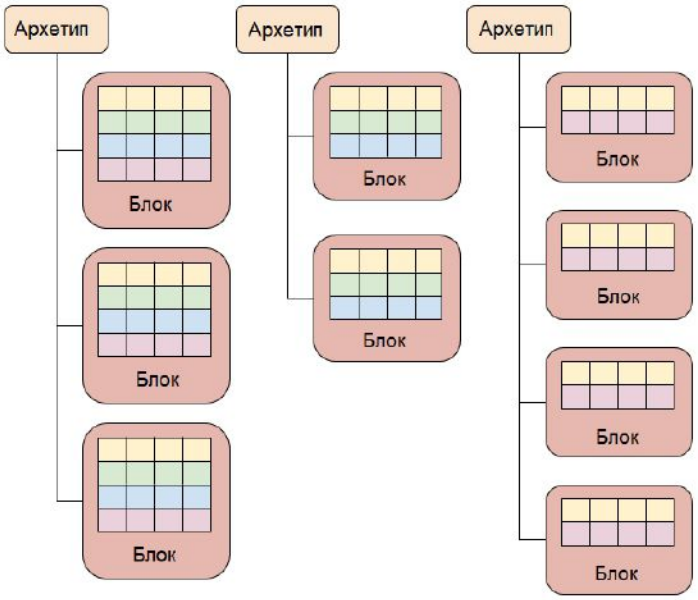

Рисунок 6 – Разбиение памяти на блоки в соответствии с архетипами сущностей

ECS не хранит объекты, находящиеся в блоке, в определенном порядке. Когда объект создается или изменяется на новый архетип, ECS помещает его в первый фрагмент, в котором хранится архетип и в котором есть место. Однако блоки остаются плотно упакованными; когда объект удаляется из архетипа, ECS перемещает компоненты последнего объекта в блоке в только что освободившиеся слоты в массивах компонентов.

1.4.3 Unity Burst Compiler

Unity Burst Compiler - технология компилятора на основе LLVM, работающая с Unity C# Jobs System и гарантирующая высоко оптимизированный машинный код.

Burst поддерживает следующие платформы: Windows, MacOS, Linux, Xbox One, PS4, Android (ARM v7 и v8+), iOS (ARM v7 и v8+).

Burst работает с подмножеством .NET, которое не позволяет использовать какие-либо управляемые объекты/ссылочные типы в коде (такие как классы C#).

2 Методы реализации системы искусственного интеллекта

В ходе проектирования системы было выделено три слоя, а именно:

– слой передвижения;

– слой планирование пути;

– слой принятия решений.

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

2.1 Метод реализации слоя передвижения

Компонент передвижения отвечает за непосредственное перемещение агента в пространстве сцены, основываясь на выбранном на уровне принятия решений действии. Этот компонент в свою очередь можно разделить на два слоя [3]:

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

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

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

Для реализации слоя управления используется подход «Рулевое поведение» (англ. Steering Beheviours), предложенный Крейгом Рейнальдсом в своей публикации «Steering Behaviors for Autonomous Characters» [4]. Суть этого метода заключается в вычислении так называемых Рулевых сил (англ. Steering force), основываясь на внешних и внутренних данных. Каждая из

этих сил моделирует определенное базисное поведение, такое как преследование, бегство или блуждание. После вычисления всех таких базисных сил, из присвоенного агенту набора, происходит вычисление результирующей силы. На этом этапе может быть использован как простой результирующий вектор, ограниченный по длине, так и более продвинутые подходы, один из которых будут рассмотрены далее в этой главе.

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

2.1.1 Рулевые силы «Поиск» и «Бегство»

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

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

Рулевая сила «Поиск» вычисляется как разность желаемой скорости и текущей скорости агента (рис. 7). Сила «Бегство» вычисляется как разность обратного вектора желаемой скорости и текущей скорости агента.

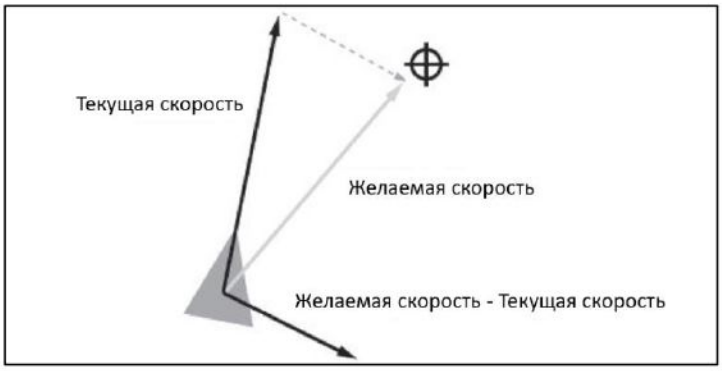

Рисунок 7 – Вычисление силы «Поиск»

2.1.2 Рулевая сила «Преследование»

«Преследования» полезно, когда агенту требуется перехватить движущуюся цель (рис. 8). Конечно, он мог бы продолжать использовать «Поиск» относительно текущей позиции цели, но это мешало бы создать иллюзию интеллекта.

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

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

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

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

где

t – искомое время;

$dist$ – текущее расстояние между агентом и целью;

v_{agent} – текущая скорость агента;

v_{target} – текущая скорость цели.

Итак, вычисление силы «Преследование» осуществляется следующим образом:

– проверяется, находится ли цель перед агентом с углом отклонения от его текущего направления в диапазоне $(-\tau; \tau)$. Если это так – возвращаем

значение «Поиск» до текущего положения цели. Иначе переходим на следующий шаг;

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

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

– возвращаем значение силы «Поиск» относительно вычисленной позиции.

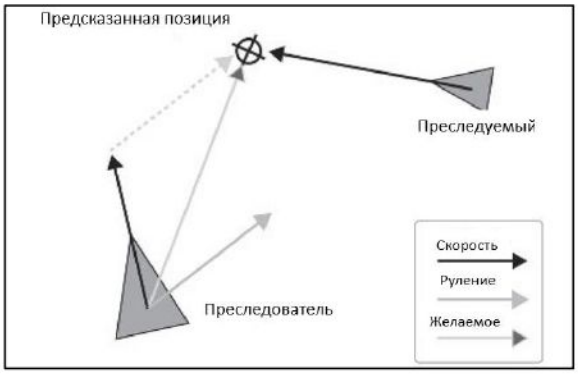

Рисунок 8 – Преследование агентом цели

2.1.3 Рулевая сила «Блуждание»

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

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

Решение, предложенное Рейнольдсом состоит в том, чтобы проецировать окружность (рис. 9) перед агентом и направлять его к цели,

которая вынуждена двигаться по этой окружности. На каждом временном шаге к этой цели добавляется небольшое случайное смещение, и со временем она перемещается вперед и назад, создавая реалистичное чередующееся движение без дрожания. Этот метод может использоваться для создания целого диапазона случайных движений, от очень плавных волнистых поворотов до вихрей и пируэтов, в зависимости от размера круга, расстояния от него до агента и величины случайного смещения в каждом кадре.

Более формальное описание шагов представлено ниже:

– инициализация:

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

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

– блуждание:

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

2) к новой позиции P прибавляется дистанция S.

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

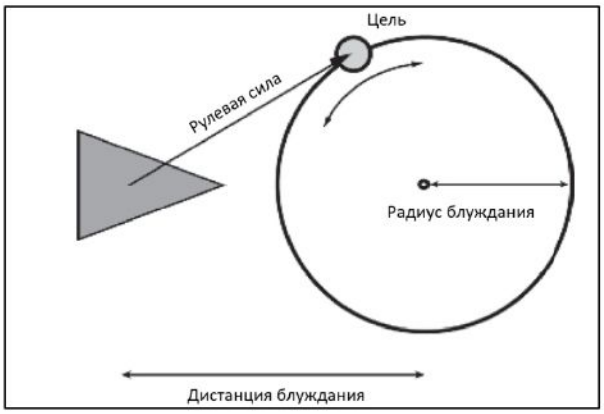

Рисунок 9 – Вычисление силы «Блуждание»

2.1.4 Рулевая сила «Избегание стен»

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

Рассматриваемая рулевая сила возвращает направление призванное повернуть агента во избежание столкновения. Это происходит путем, проецирования трех «щупалец» перед агентом и проверяя, пересекаются ли они с какими-либо стенами в игровом мире (рис. 10). Когда будет найдена ближайшая пересекающаяся стена, происходит вычисление силы путем расчета того, насколько далеко наконечник «щупальца» проник через стенку, и затем создания силы такой величины в направлении нормали стены.

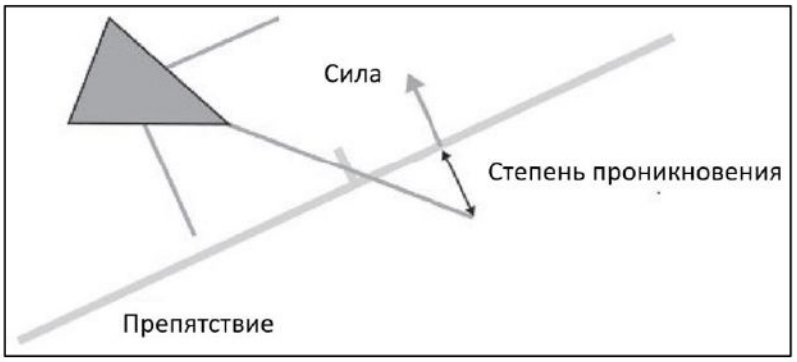

Рисунок 10 – Вычисление силы «Избегание стен»

2.1.5 Вычисление результирующей силы

Финальным этапом расчёта рулевых сил является вычисление результирующего вектора. В программной реализации используется подход приоритетного колебания, предложенный Крейгом Рейнальдсом [4]. Его суть заключается в следующем:

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

– для каждой силы в порядке приоритета вычисляется случайное значение из диапазона [0; 1] и в случае, если оно меньше либо равно

установленной для этой силы вероятности – происходит ее расчет и возвращение этого значения как результирующего, иначе – переход к следующей по приоритету для вычисления;

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

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

2.2 Метод реализации слоя планирования пути

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

– алгоритм поиска кратчайшего пути в графе;

– система переноса виртуального окружения в формат графа.

В качестве систем переноса окружения в формат графа будут рассмотрены два подхода: точки видимости и навигационная полигональная сетка.

2.2.1 Алгоритм поиска пути A*

Для поиска пути используется алгоритм A*, представляющий собой модификацию алгоритма Дейкстры [3].

Алгоритм A* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все информированные алгоритмы поиска, он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению,

его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Составляющая g(x) – это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

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

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

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

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

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

где

h(x) – используемая эвристика;

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

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

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

2.2.2 Точки видимости

Как правило, виртуальное пространство в видеоигре представлено 2D- или 3D-сценой с разной степенью сложности геометрии, будь то город, пещера или набор комнат. Для того, чтобы использовать алгоритм поиска пути, необходимо перевести данные об окружении в формат графа и работать уже с ним. Можно выделить три аспекта этого процесса [5]:

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

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

– квантование, а именно, процесс переноса конкретной точки окружения в соответствующую вершину графа.

В реализации используется подход, основанный на точках видимости. Он основам на факте, что кратчайший путь через окружение всегда будет иметь точки перегиба [5] в выпуклых вершинных геометрии этого окружения. Эти точки можно использовать в качестве вершин в навигационном графе для воссоздания правдоподобного пути агента. Вершины связаны ребром в том случае, если между соответствующими им точкам можно провести линию, которая не пересекается с каким-либо препятствием – иными словами из одной точки можно увидеть другую (рис. 11).

Одной из особенностей навигационных графов на основе точек видимости является то, что они могут быть легко расширены, чтобы включать в себя узлы с дополнительной информацией, как например различные позиций для стрельбы, укрытия или засады. С другой стороны, в случае сложной структуры окружения потребуется много ручного труда доля корректной расстановки точек видимости.

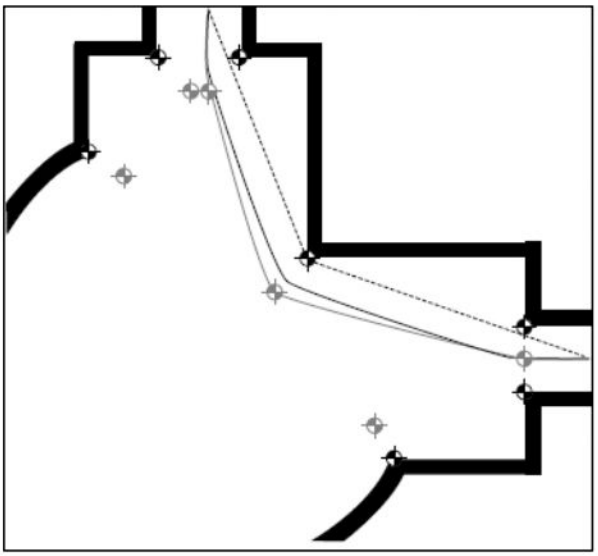

Рисунок 11 – Путь с использованием точек видимости

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

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

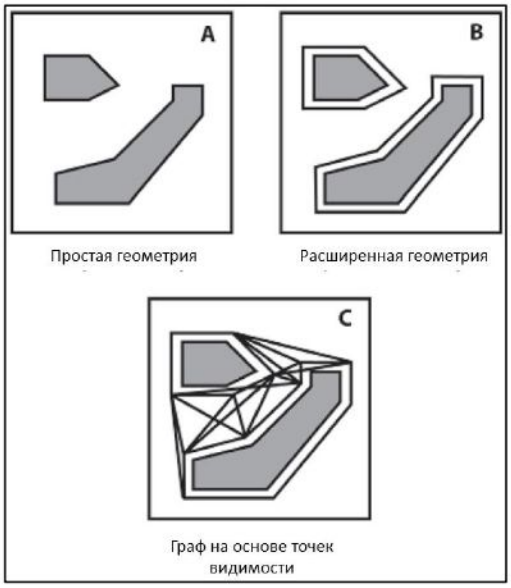

Рисунок 12 – Построение графа на основе точек видимости

2.2.3 Навигационная полигональная сетка

Большинство современных игр используют навигационные полигональные сетки, сокращенно нав-меш (от англ. Mesh - сетка) для поиска пути. Подход навигационной сетки к поиску пути основан на том факте, что сама геометрия сцены состоит из многоугольников, соединенных с другими многоугольниками. Представляется возможным использовать эту графическую структуру как основу для построения графа поиска пути. Каждый многоугольник действует как вершина графа, а узлы соединяются, если соответствующие им полигоны имеют общую границу. Полигоны обычно представляют собой треугольники, но могут быть и четырехугольниками (рис. 13). Таким образом, узлы имеют три или четыре соседних вершины, и низкий коэффициент ветвления. Создание навигационной сетки обычно подразумевает, что 3D-художник помечает

определенные полигоны как, например, «проходимы» и «непроходимые» в своем пакете моделирования. Навигационные сетки требуют меньшего лишнего вмешательства разработчика, чем другие подходы.

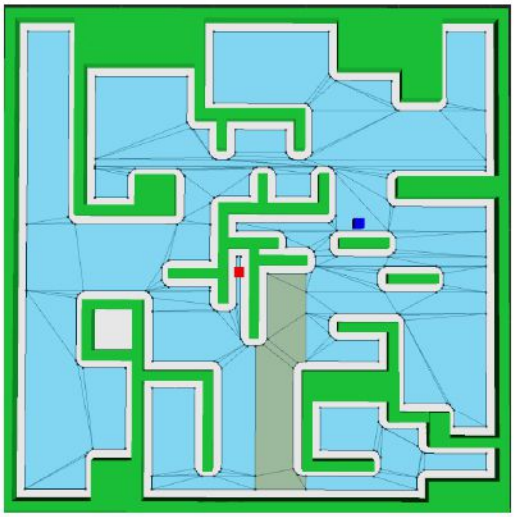

Рисунок 13 – Пример навигационного меша

Для определения, к какому полигону в настоящий момент принадлежит агент, сначала происходит первичное поиск среди всех. В дальнейшем же используется предположение о согласованности. Оно строится на том факте, что, если известно, в каком месте находился агент в предыдущем кадре, то он, скорее всего, будет в том же полигоне или ближайшем соседнем и в следующем кадре. Этот подход полезен во многих схемах разделения, но особенно важен при работе с навигационными мешем.

Единственная сложность возникает, когда агент не касается пола. Не составляет труда найти и использовать первый многоугольник под ним. К сожалению, агент может оказаться в совершенно неподходящем узле при падении или прыжке. Он проецируется в нижнюю часть сцены, хотя на самом деле использует проходы выше (рис. 14). Это может затем заставить систему перепланировать свой маршрут, как если бы агент находился внизу сцены.

При локализации вершины можно использовать любую точку полигона, но обычно используется геометрический центр. Это отлично подходит для треугольников.

Области, созданные с помощью навигационных сеток, могут быть проблематичными с точки зрения связанности. Предполагается, что любая точка, ассоциированная с одной вершине, может быть достигнута непосредственно из любой точки, ассоциированная с соседней вершиной, но это в общем случае может быть не так. На рисунке 15 показан пример, который содержит пару треугольников с областями, в которых прямое перемещение между ними вызывает столкновение. Но поскольку 3D-модель поверхности создаются дизайнером уровней или художником, эту проблему можно решить на этапе производства контента сцены.

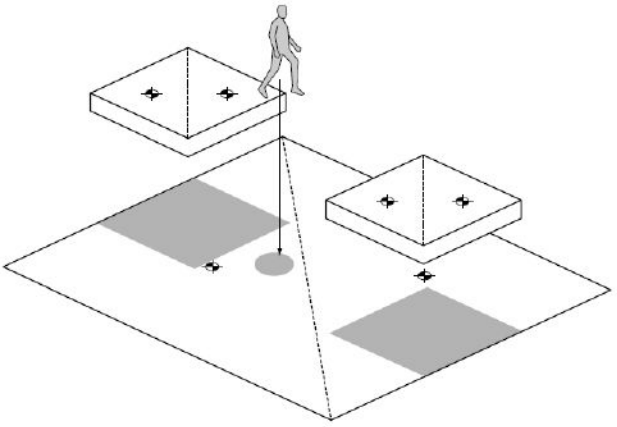

Рисунок 14 – Проблема проекции позиции на некорректный полигон

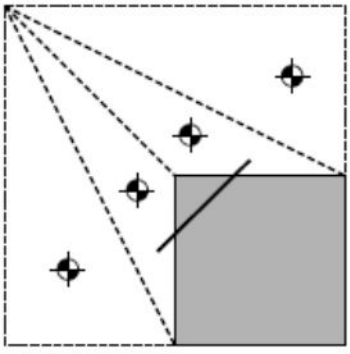

Рисунок 15 – Проблема обеспечения связности

2.3 Методы реализации слоя принятие решений

Слой принятия решений отвечает за способность агента выбирать подходящее дальнейшее действие, исходя из текущей обстановки. Иными словами, имея некоторый набор входных данных агент должен выработать соответствующие выходные данные – решение о том, какое из возможных действий ему следует предпринять в данный момент (рис. 16).

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

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

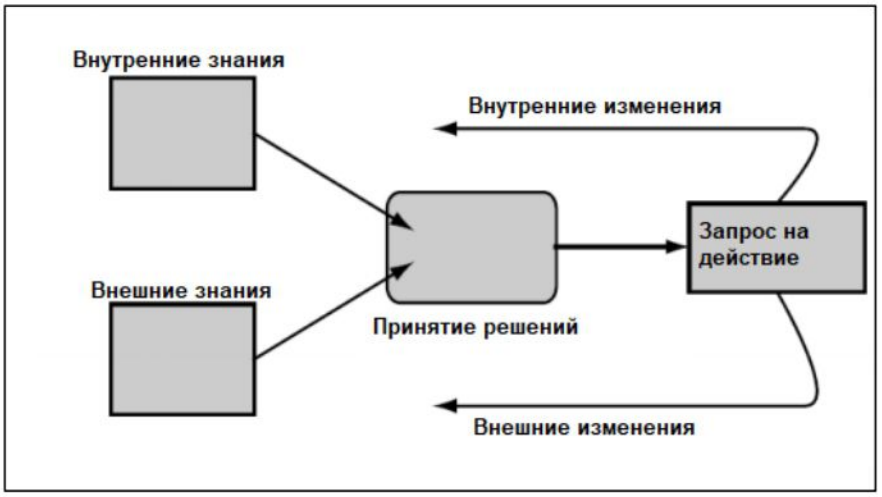

Рисунок 16 – Схема принятия решений

Формальное определение конечного автомата состояний [6] звучит следующе: конечный автомат – это устройство или модель устройства, которое имеет конечное число состояний, в которых оно может находиться в

любой момент времени, и может работать с входными данными, чтобы либо совершать переходы из одного состояния в другое, либо осуществлять выход или необходимое действие. Конечный автомат может находиться только в одном состоянии в любой момент времени.

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

Идея конечного автомата, состоит в том, чтобы разложить поведение объекта на легко управляемые «фрагменты» или состояния (рис. 17). Например, выключатель света на стене – это очень простой конечный автомат. У него есть два состояния: включено и выключено. Переходы между состояниями производятся нажатием на переключатель.

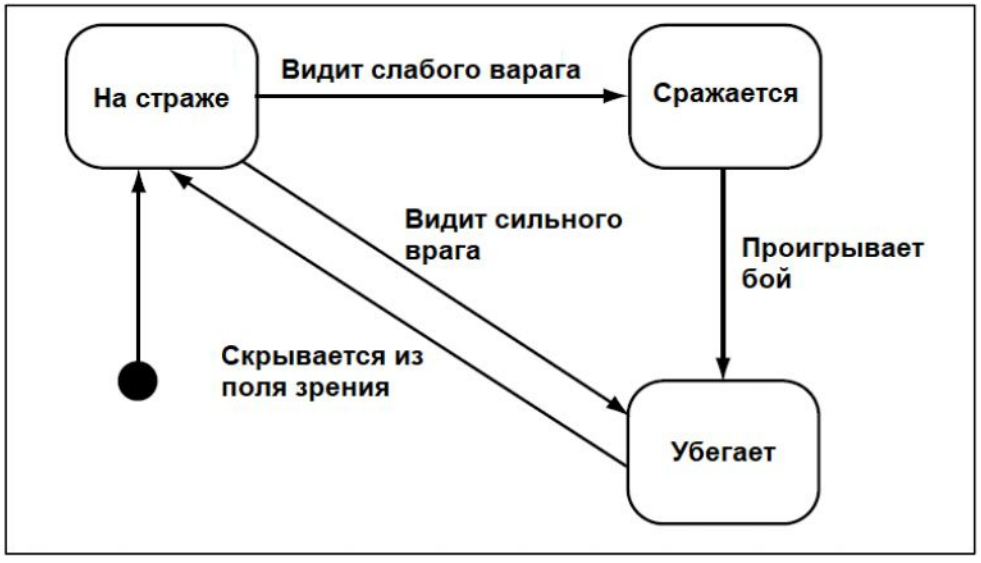

Рисунок 17 – Пример конечного автомата состояний для агента

Конечные автоматы в течение многих лет были излюбленным инструментом программистов ИИ, чтобы наполнить игровых агент иллюзией интеллекта. Конечные автоматы того или иного типа можно найти практически в каждой игре, которые появлялись на прилавках с первых дней видеоигр, и, несмотря на растущую популярность более продвинутых архитектур, они будут существовать еще долгое время [3]. Вот лишь некоторые из причин, почему [3][6]:

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

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

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

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

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

3 Программная реализация

В рамках программной реализации была разработана многоагентная система ИИ на основе подхода Data Oriented Design. Системы поддерживает параллельные вычисления при работе с набором сущностей. Кроме этого, архитектура системы построена на паттерне Entity Component System, является расширяемой, а взаимодействие ее компонент построено так, чтобы обеспечить наименьшую связанность между ними, что обеспечивает высокую степень модульности и простоту дальнейшего сопровождения.

Далее будут более подробно рассмотрены реализации отдельных слоев системы.

Кроме этого, на базе построенной системы разработано демонстрационное приложение, реализующее ИИ из двух типов агентов – «Преследователь» и «Жертва». Каждый из них имплементирует соответствующее поведение.

3.1 Реализация слоя расчёта передвижения

Реализация расчета передвижения основана на примени рулевых поведений и рулевых сил, описанных в пункте 2.1. Для начала необходимо рассмотреть структуру данных, используемых в ходе вычислений (рис. 18).

В вычислениях учувствуют два архетипа сущностей: Agent – ассоциирован с внутренними данными агента, такими как вектор скорости, позиция и вращение; Force – ассоциирована с данными рулевой силы, такими как набор компонент для хранения вычисленного значения для каждого конкретного вида силы. Для того, чтобы иметь доступ к данным друг друга, в случаях, когда невозможно построить вычисления линейно, они связаны через данные компонент SteeringForceAgent<T> и SteeringForceHostData, в которых хранятся идентификаторы конкретных сущностей.

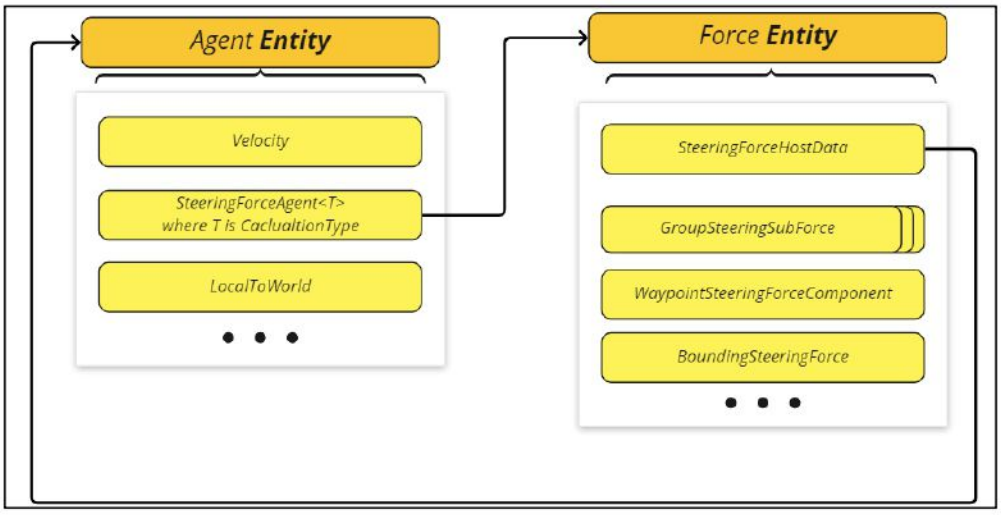

Рисунок 18 – Структура данных рулевых поведений

В компоненте SteeringForceAgent<T> присутствует несколько особенностей, которые необходимо рассмотреть отдельно. Он содержит параметр generic-типа в виде T, который выступает своего рода маркером при определении, каким образом необходимо рассчитывать результирующую рулевую силу, что позволяет передать компонент в соответствующую систему. Кроме этого, он содержит саму величину результирующей силы, что позволяет организовать ее применение к текущей скорости в виде линейных вычислений, что в свою очередь использует кэш-память процессора наиболее эффективно

Перейдем к архитектуре систем, осуществляющих работу с рулевыми силами (рис. 19).

Набор систем логически разбит на три группы: SteeringSubForceSystemGroup, CalculateSteeringForceSystemGroup, ApplyForceSystemGroup. Группы систем обновляются последовательно, но системы в рамках одно группы могут работать параллельно. Рассмотрим каждую из групп более подробно.

SteeringSubForceSystemGroup – набор систем, вычисляющих значения отдельных рулевых сил, таких как «Бегство» и «Преследование». Как правило они читаю внешние данные, как данные других агентов и

окружения, внутренние данные агента; модифицируют и читают данные конкретной рулевой силы, к которой относятся.

CalculateSteeringForceSystemGroup – набор систем, вычисляющих результирующее значение рулевой силы, в зависимости от выбранного метода. Системы имеет доступ к данным агента в режиме запись/чтение и доступ к данным значений рулевых в режиме только чтение.

ApplyForceSystemGroup – набор систем, применяющих результирующие силы к вектору скорости. Представляет собой нижний слой компонента передвижений системы ИИ. Как было упомянуто в пункте 2.1 – в большей степени связан с физическим движком приложения и не связан с системой ИИ. По этой причине, подходы к его реализации далее не рассматриваются.

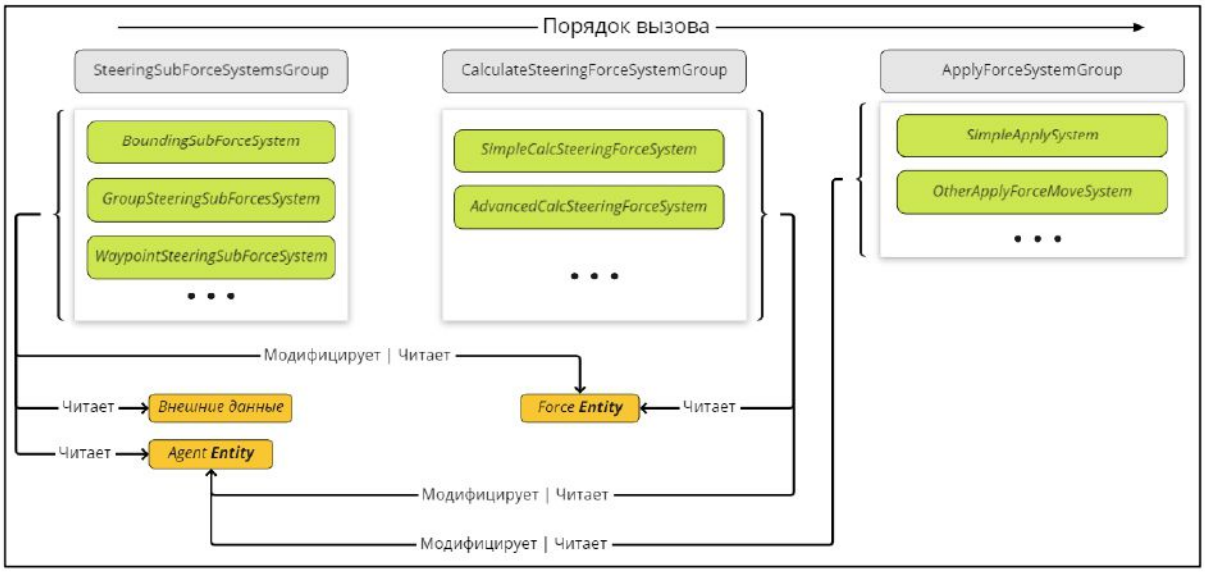

Рисунок 19 – Архитектура вызова систем рулевых сил

Построенная таким образом архитектура работы систем гарантирует, что при произведении вычислений будут модифицироваться только данные одного типа, ассоциированные только с одной сущностью, в то же самое время есть возможность иметь любые данные в режиме «только чтение». Это позволяет избегать конкурентности вычислений и организовывать их параллельно.

3.2 Реализация механизма запроса маршрута

Для начала рассмотрим структуру данных, связанных с механизмом поиска пути (рис 20). В механизме участвует два архетипа сущностей: Agent и Path.

Сущность Agent в контексте поиска пути может содержать один из двух типов данных, в зависимости от текущего состояния процесса поиска пути:

– AgentRequestPathComponent – описание запроса на поиск пути с указанием позиции старта и окончания маршрута.

– FollowPathAgentComponent – описание текущего процесса следования пути в виде: индекса текущего контрольной точки, идентификатора на соответствующую сущность архетипа Path и состояния в виде одного из трёх значений {Starting, Following, DestReached}.

Сущность Path содержит следующие компоненты:

– PathDescComponent – содержит описание прогресса поиска пути: {Idle, InProgress, Done, Failed}.

– PathWaypointBufferElement – буфер позиций контрольных точек;

– PathRequestComponent – компонент дополнительного запроса на поиск маршрута – используется в случаях, когда необходимо произвести поиск в несколько итераций.

Построение данных таким образом позволяет осуществлять поиск в течение нескольких кадров, при этом передавать агенту частичный путь, чтобы он мог начать движение.

Принцип работы механизма поиска маршрута представлен на рисунке 20. Прежде всего некоторой клиентской системой создается запрос в виде компонента и помещается на сущность агента. Далее система PathRequestSytem считывает запрос, создает сущность Path со всеми необходимыми данными и заменяет компонент запроса FollowPathAgentComponent у Agent.

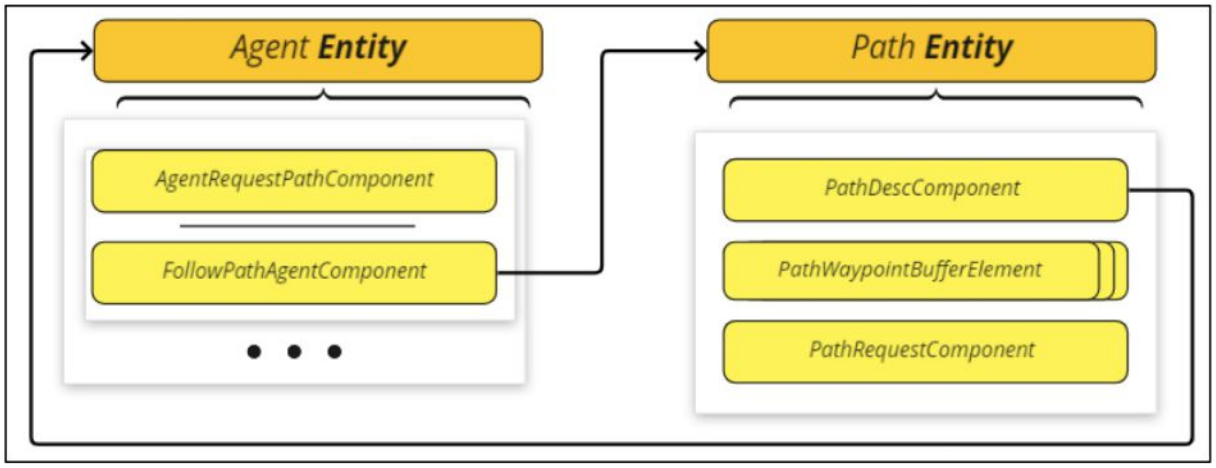

Рисунок 20 – Структура данных запроса поиска пути

Система PathBuildSystem считывает и модифицирует данные Path для поиска пути. Она также хранит в виде кэша все уже найденные пути для повторного использования. В случае, если полный путь не найден за установленное количество итераций, промежуточный результат сохраняется и в рамках следующего кадра возобновляется. Тем не менее даже частичный путь может быть использован системой PathTrackSystem для последующей передачи Agent. Имея данные контрольных точек, Agent может начать движение.

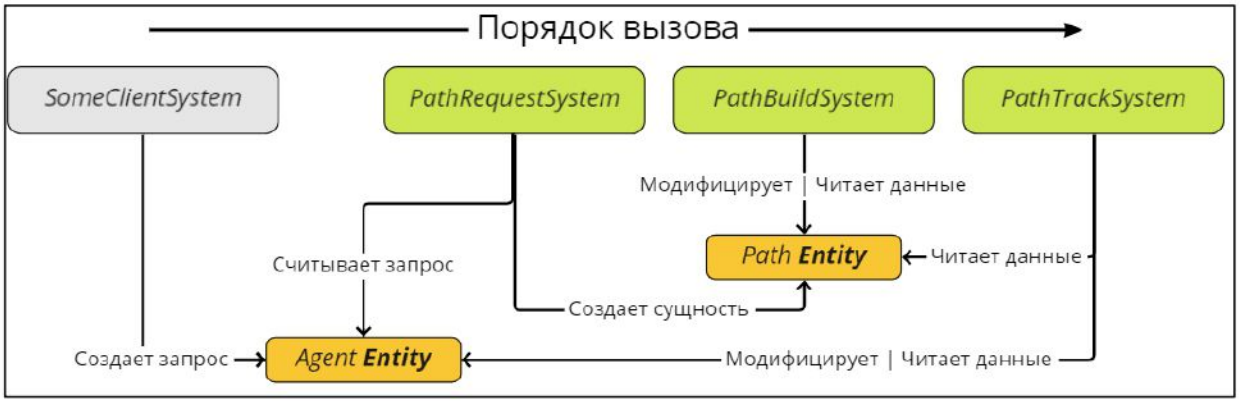

Рисунок 21 – архитектура систем обработки запроса на поиск пути

Реализованные таким образом системы взаимодействуют друг с другом опосредовано через модификацию данных, что снижает связность. Система

не зависти от конкретного алгоритма поиска пути или механизма переноса виртуального окружения в формат графа: возможно функционирование с различными решениями, как на основе нав-меша, так и на основе точек видимости. Кроме этого, появляется возможность проводить операцию поиска пути параллельно для всех агентов. Рассмотрим сам поиск пути более подробно.

3.3 Реализация построения маршрута

В реализации поиск маршрута происходит на навигационном графе, который представляет собой нагруженный не ориентированный и, как правило, разреженный граф. Для осуществления поиска в первую очередь было необходимо создать эффективную структуру для хранения данных графа. Для этого использовалось множество смежности для каждой вершины со следующими модификациями:

– в множестве хранится множество ребер, содержащих эту вершину;

– ребро хранится в виде упорядоченной пары (a, b), где a – всегда та вершина, множество ребер которой мы рассматриваем.

Описанной структуры достаточно для хранения данных связности графа. Для хранения метаинформации, как вес ребер и физическое расположение вершины в трехмерном пространстве, необходимых алгоритма A*, используются хеш-таблицы, предоставляющие доступ к данным за константное время.

Так как граф не ориентированный, то вес ребра (a, b) и (b, a) равны, следовательно в таблицах нужна хеш-функция, которая для приведенного вида ребер вернет одинаковое значение. В качестве такой хеш-функции была выбрана модифицированная функция Шуджика [7] для пары неотрицательных целых чисел, вычисляемая следующим образом:

$$(a, b) \rightarrow y^2 + x = hash\,, \tag{3}$$

где

(a, b) – данное ребро;

$y = min(a, b)$;

$x = max(a, b)$;

hash – результат вычисления в виде целого числа.

Алгоритм A* поиска пути вынесен в отдельный полностью статичный класс. На вход метод получает ссылку на экземпляр графа, координату текущей позиции и координату цели. Далее определяются вершины начала и конца пути. Они представляют собой ближайшие вершины для входных координат, до которых можно пробросить непересекающийся с препятствиями луч. Процесс трассировки лучей использует библиотеку Unity.Physics;

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

Навигационный граф строится на основе точек видимости, которые могут быть как расставлены вручную в редакторе сцены Unity, так и автоматически при генерации тестового окружения основываясь на полученной геометрии. На рисунке 22 представлена визуализация навигационного графа в редакторе Unity. Красными точками обозначены вершины графа, серые лини – ребра полученного графа.

Помимо точек видимости, система поиска пути может работать с навигационной полигональной сеткой, такой, как например Unity NavMesh (рис. 23), так же осуществляя вычисления для множества агентов параллельно.

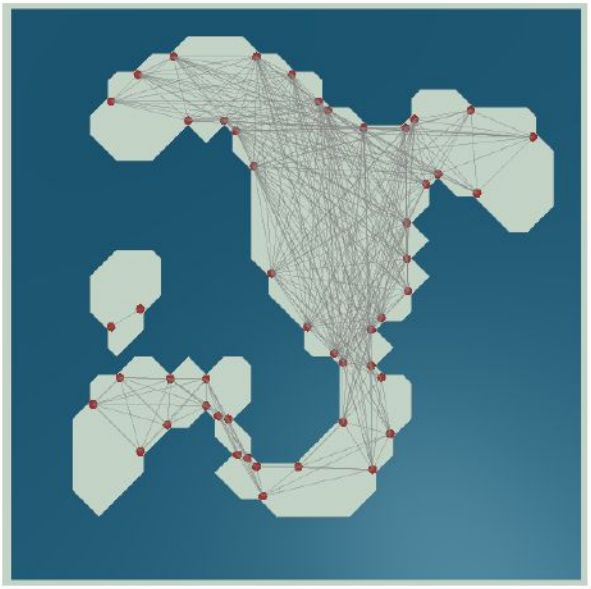

Рисунок 22 – Визуализации навигационного графа

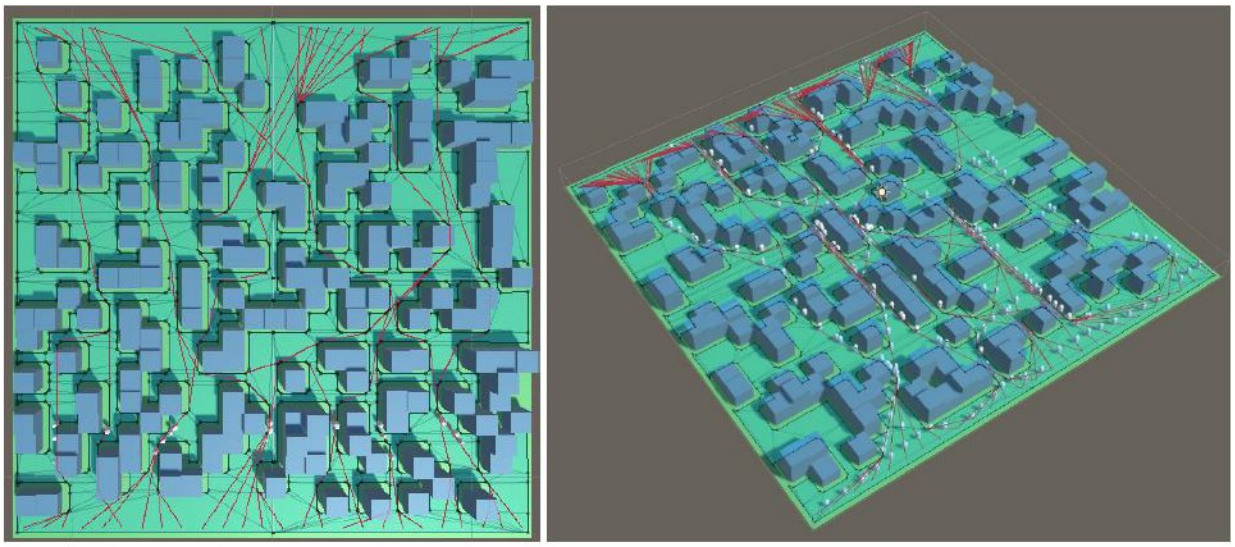

Рисунок 23 – Поиск пути на основе данных нав-меша

3.4 Реализация слоя принятия решений

Для реализации слоя принятия решений использовались конечные автоматы состояний, описанные в пункте 2.3. Схема реализации изображена на рисунке 24.

В контексте реализованного конечного автомата состояний, типы компонент сущности можно разделить на 3 вида:

– постоянные данные, наличие которых не зависит от состояния, такие как позиция и скорость;

– данные, характерные для текущего состояния, например данные преследуемой цели;

– маркеры текущего состояния и маркеры перехода из одного состояния в другое, необходимые для обработки сущности соответствующими системами.

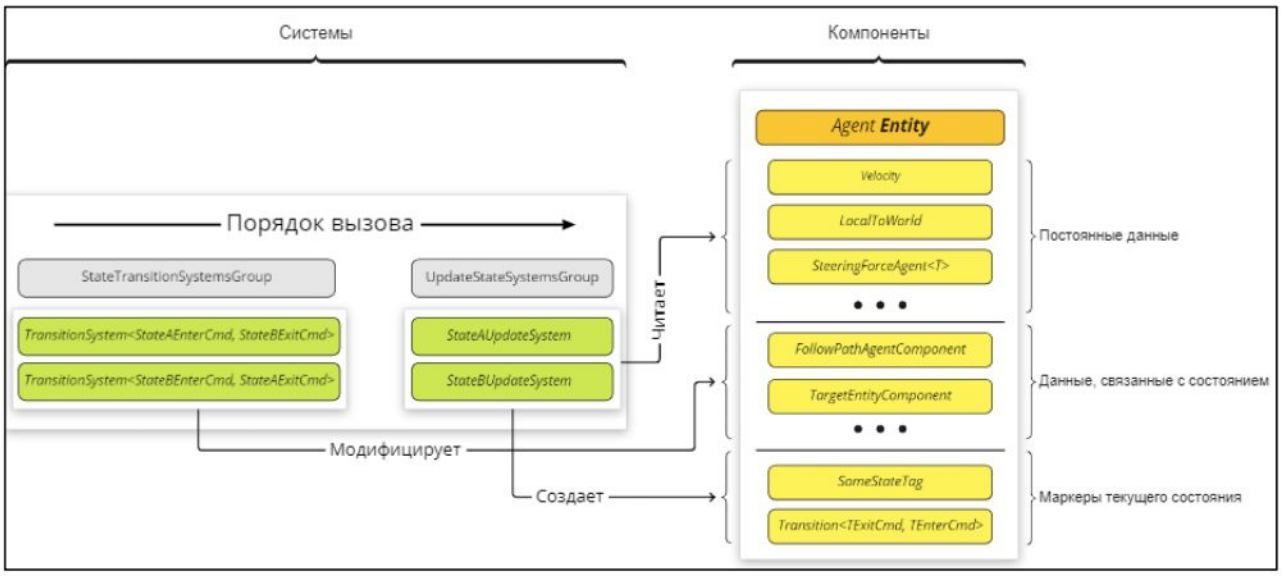

Рисунок 24 – Схема реализации системы принятия решений

Рассмотрим последний вид более подробно. Маркер текущего состояния представляет собой пустой компонент, помещаемый на сущность при завершении перехода из одного состояния к другому. При этом архетип сущности меняется, и она «перемещается» в другой блок памяти.

Маркеры перехода состояний – это компоненты данных, содержащий в себе инкапсулированные данные того, какие структурные изменения необходимо произвести с сущностью при выходе из состояния A и переходе в состояние B, например добавить необходимые для текущего состояния данные. Информация о необходимых изменениях оформлена в виде паттерна «Команда», принимающей в качестве одного из аргументов идентификатор сущности.

В соответствии с ECS, логика работы конечного автомата состояний осуществляется в системах. В системах группы UpdateStateSystemsGroup заключена логика мониторинга данных и запроса перехода между состояниями при достижении заданных условий путем размещения компоненты маркера перехода типа Transition<TExitCmd, TEnterCmd> на сущности. В свою очередь системы группы StateTransitionSystemsGroup считывают такие маркеры и осуществляют хранящиеся в них команды. Поскольку системы работают только с сущностями, имеющими требуемые типы данных (в данном случае компоненты-маркеры), ECS выполняет сортировку этих сущностей таким образом, что сущности в одном и том же состоянии хранятся вместе. Это делает обработку всех сущностей в определенном состоянии более эффективной с точки зрения использования кэша процессора.

На этапе проектирования также рассматривались другие варианты реализации.

Первой альтернативой было использование компонент типа SharedComponent, позволяющих группировать сущности в памяти таким образом, что сущности с одинаковым значением такого компонента хранятся вместе. Такой компонент должен был бы хранить обозначение текущего состояния в виде строки или иным образом закодированным. Таким образом изменение данных в нем влекло бы перемещение в другой блок памяти. Однако это бы не давало никаких преимуществ по сравнению с решением на основе добавления компонент, так как потребовало бы дополнительную фильтрацию по для распределения сущностей по системам. А поскольку запросы с фильтрацией обходятся дороже, чем разделение фрагментов по архетипу, это привело бы к снижению производительности.

Другой альтернативой было простое хранение данных, как в решении выше, но без перемещения данных сущности между блоками памяти и смены архетипа. В этом случае каждая система отслеживания состояний работала бы со всеми сущностями, имеющих этот компонент, но отсеивала бы те,

значения состояния которых ей не соответствуют. Выполнение этой условной логики в каждом кадре представляет из себя слишком много лишней работы для ЦП.

Реализованный же подход позволяет более эффективно использовать кэш процессора за счет разделения сущности по блокам в зависимости от закрепленного за ними маркера состояния. Это так же облегчает процесс фильтрации из-за соответствующего их разбиения по архетипам. Единственным накладным расходом является добавление/удаление компонент при смене состояния, влекущее за собой смену архетипа и перемещение в памяти. Однако поскольку в подобных система смены состояния происходит относительно редко и как правило не для всех сущностей разом, то подобные расходы можно считать приемлемыми.

3.5 Демонстрация работы системы

Для демонстрации взаимодействия всех слоев системы было разработано приложение, в которой агенты ИИ реализуют поведения «Хищник» и «Жертва».

На рисунке 25 изображен конечный автомат агента «Хищник». Всего предусмотрено 3 состояния: блуждание-поиск, преследование и атака. Поведение агента заключается в перемещении по окружению до тех пор, пока он не наткнется на жертву. Далее он начнет ее преследовать до тех пор, пока не сблизится достаточно, чтобы поймать, либо пока цель не отдалится слишком далеко. При этом используется следующий набор рулевых сил для маневрирования: «Блуждание», «Преследование», «Избегание стен».

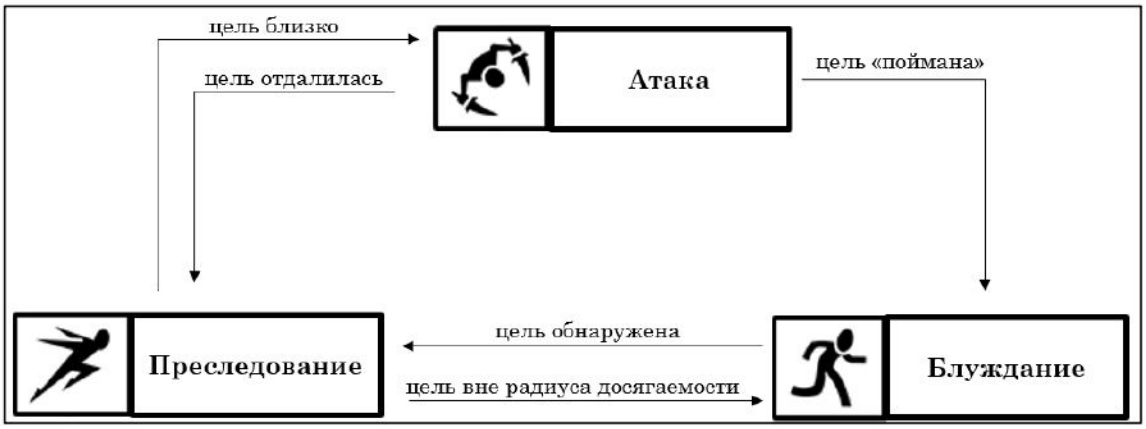

Рисунок 25 – Конечный автомат состояний «хищника»

На рисунке 26 изображен конечный автомат агента «жертва». В нем имеется также 3 состояния: блуждание, побег и укрытие. Его поведение можно описать следующим образом: агент перемещается по сцене до тех пор, пока не обнаружит «хищника». Далее он пытается добраться до укрытия не будучи пойманным. В случае, если ему это удается – он перестает находиться в зоне досягаемости и пропадает со сцены. При этом используется следующий набор рулевых сил для маневрирования: «Блуждание», «Бегство», «Поиск».

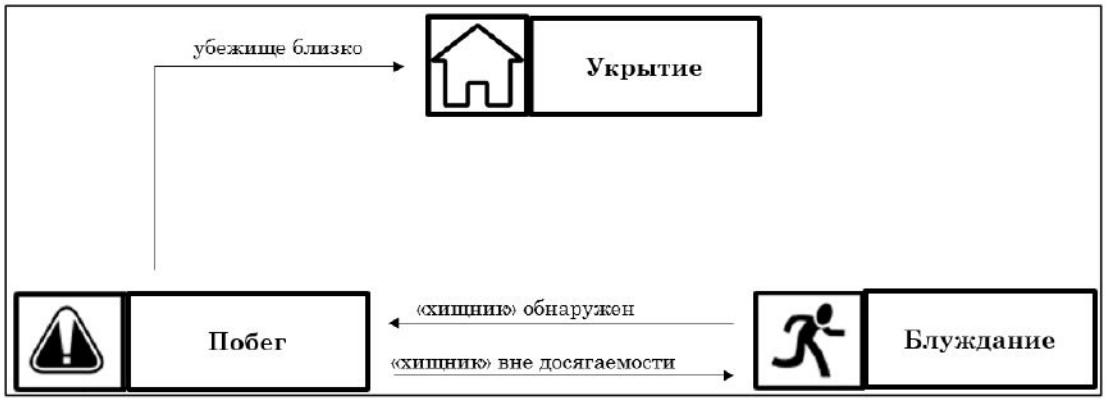

Рисунок 26 – Конечный автомат состояний «жертвы»

Перенос окружения осуществляется через разработанную систему точек видимости.

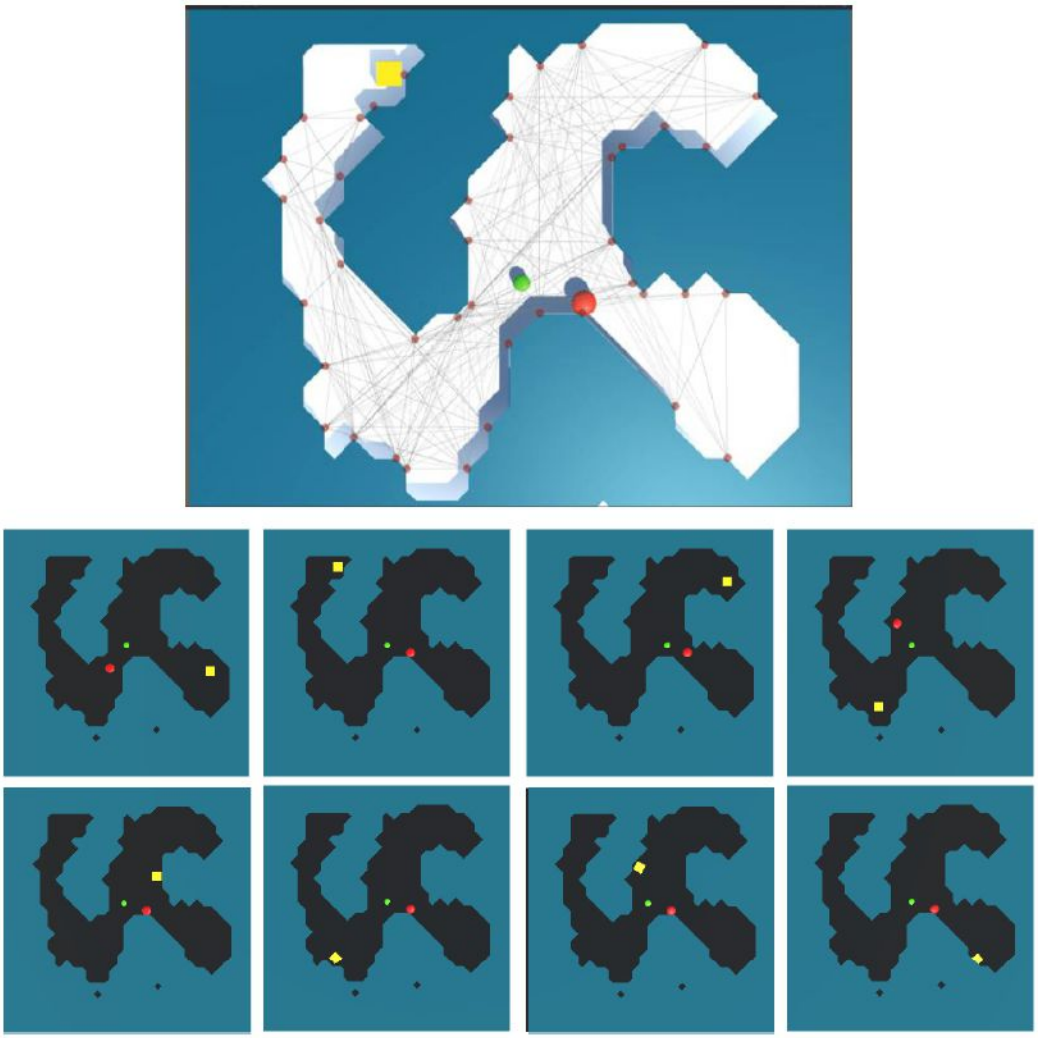

Рисунок 27 – Функционирование системы

3.6 Анализ производительности реализации на основе DOD

Анализ базируется на сравнении эффективности использования подхода рулевого поведения для моделирования поведения стаи. Такой пример был выбран исходя из того, что в рамках него агенту для функционирования необходимы как внутреннего данные состояния (скорость, позиция, ориентация в пространстве) так и внешние данные в виде информации о других агентах.

Первая реализация использует принципы ООП и стандартный подход на базе наследования от Unity MonoBehaviour для построения архитектуры [8] – схематично изображена на рис. 28. Класс BoidBehAgent реализует метод Update, в теле которого происходит вычисление рулевой силы и затем

перемещения агента с применением этой силы и второго закона Ньютона. Стоит отметить, что вычислительная сложность алгоритма вычисления рулевой силы для одного агента равна o(n), где n – общее количество агентов на сцене. Метод Update вызывается последовательно для всех экземпляров BoidBehAgent, тем самым общая вычислительная сложность составляет $o(n^2)$.

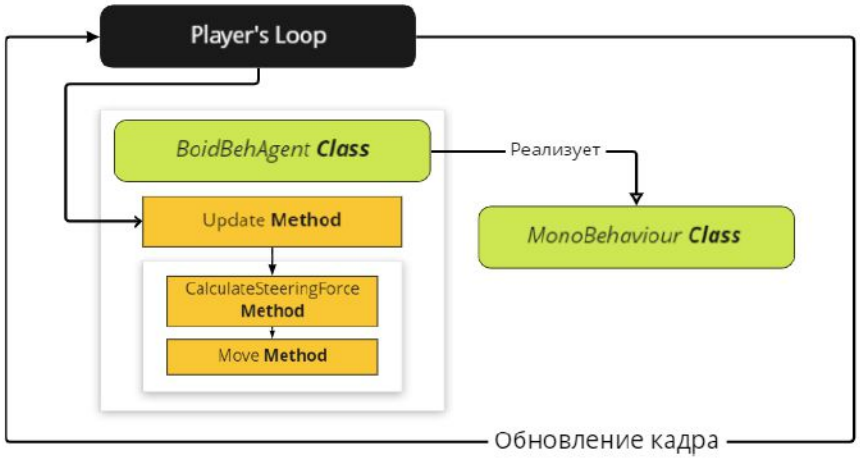

Рисунок 28 – Архитектура на основе MonoBehaviour

Для такого подхода максимальное число агентов на сцене составило 225, сохраняя при этом плавность работы в примерные 30 кадров/с. Данные профайлера предоставлены на рис. 29.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▼ PlayerLoop	92.2%	0.2%	3	1.4 KB	34.00	0.09
▼ Update.ScriptRunBehaviourUpda	85.3%	0.0%	1	0 B	31.45	0.00
▼ BehaviourUpdate	85.3%	1.0%	1	0 B	31.45	0.38
BoidBehAgent.Update()	84.3%	84.3%	225	0 B	31.07	31.07

Рисунок 29 – Данные профайлинга системы на основе MonoBehaviour

Для реализации описанного выше функционала используется разработанная система расчёта рулевых сил. Данные сущностей Agent Force идентичны описанным в пункте 2.1.

Логика преобразования данных оформлена в виде систем, MoveEntitySystem и SteeringForceSytem. Каждая из них считывает в качестве входных данных определенный набор компонент, и модифицирует в качестве

выходного другой, как показано на рисунке 30. Стоит отметить, что каждая операция преобразования входных данных в выходные происходит параллельно и не зависит от других таких операций. Системы обновляются последовательно в рамках построения одного кадра.

При применении ECS при том же количестве агентов на экране, время обработки кадра снизилось с 33 мс до 8 мс (рис. 31). При этом появилась возможность повысить общее количество агентов до 1500, сохранив при этом плавность работы.

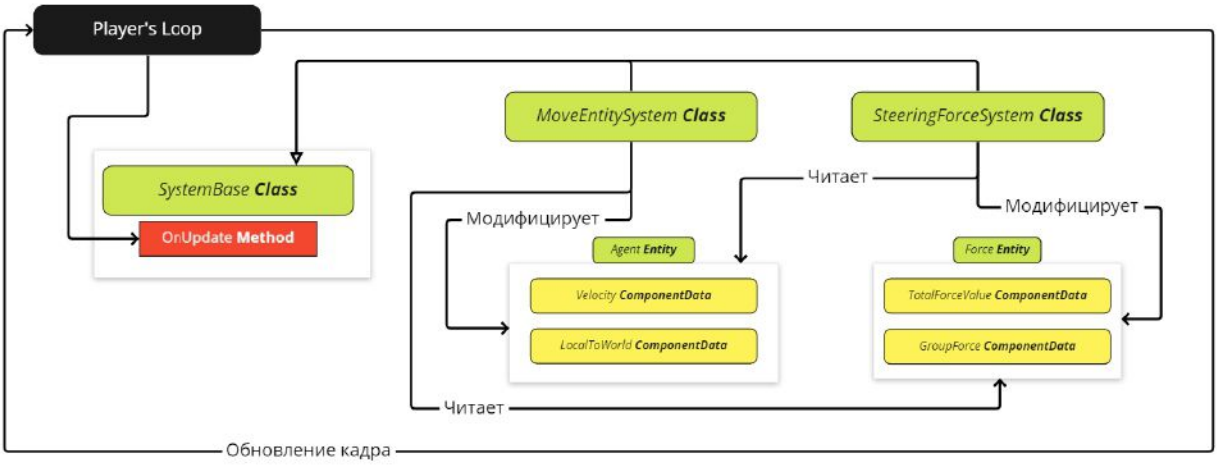

Рисунок 30 – Архитектура на основе ECS

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
▶ PlayerLoop	75.1%	0.7%	3	1.4 KB	8.16	0.08

Рисунок 31 – Данные профайлинга системы на основе ECS

Таким образом благодаря применению технологий Unity DOTS получилось повысить производительность более чем на 300%, при этом общее количество сущностей на экране возросло более чем в шесть раз. Наглядно разницу можно увидеть на рисунке 32.

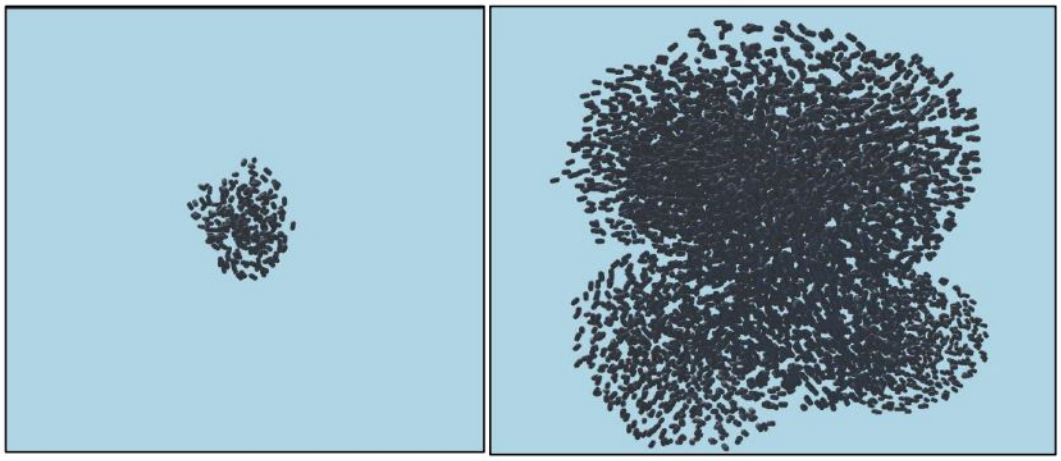

Рисунок 32 – Наглядное сравнение количества агентов

ЗАКЛЮЧЕНИЕ

В результате выполнения выпускной квалификационной работы решены задачи, сформированные во введении:

– изучены принципы и особенности разработки с использованием DOD;

– изучены возможности платформы Unity поддержки применения принципа DOD;

– выбраны подходы для реализации различных компонентов системы ИИ, с учетом возможности параллельных вычислений;

– реализованы подсистемы принятия решений для автономных агентов, расчёта перемещения для автономных агентов, планирования маршрута для автономных агентов;

– сделан сравнительный анализ производительности полученной системы с применением DOD относительно системы с применением ООП;

– продемонстрированы возможности разработанной системы.

Результатом работы является система ИИ, спроектированная и разработанная в соответствии с подходом Data Oriented Design, имеющая возможность дальнейшего расширения, позволяющая осуществлять вычисления параллельно и эффективно использовать кэш-память процессора. Возможности разработанной системы были продемонстрированы на примере прототипа.

Проведен анализ производительности разработанной системы посредством ее сравнения с аналогичной, но построенной на принципе ООП.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Data Locality – Optimization Game Programming Pattern [Электронный ресурс]. – Режим доступа: https://gameprogrammingpatterns.com/data-locality.html (дата обращения: 03.05.2021).

2. Lopis N. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). Game Developers Journal. 14 September 2009 – URL: https://gamesfromwithin.com/data-oriented-design.

3. Buckland, M. Programming Game AI by Example 1st Edition / M. Buckland. – Плано, Техас, США: Wordware Publishing, Inc., 2005. – 85 с.

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

5. Millington, I. Artificial Intelligence for Games / I. Millington, J. Funge. – Берлингтон, США: Morgan Kaufmann Publishers, 2009. – 244 с.

6. AI Game Programming Wisdom 4 / ред. S. Rabin. – Бостон, США: Charles River Media, 2008. – 212 с.

7. Szudzik M. An Elegant Pairing Function. // Wolfram Research, Inc., 2015. (Engl.) – URL: http://szudzik.com/ElegantPairing.pdf [15 марта 2022].

8. Unity User Manual (2018.04): официальный сайт / США, Сан-Франциско – URL: www.docs.unity3d.com/2018.4/Documentation/Manual/index.html (дата обращения 05.04.2020).

АНТИПЛАГИАТ
ОБНАРУЖЕНИЕ ЗАИМСТВОВАНИЙ

СПРАВКА

Кубанский Государственный университет

о результатах проверки текстового документа
на наличие заимствований

ПРОВЕРКА ВЫПОЛНЕНА В СИСТЕМЕ АНТИПЛАГИАТ.ВУЗ

Автор работы: Сеидов Кирилл Рустемович
Самоцитирование рассчитано для: Сеидов Кирилл Рустемович
Название работы: Разработка многоагентной системы искусственного интеллекта посредством data-oriented design
Тип работы: Магистерская диссертация
Подразделение: ФКТиПМ, кафедра анализа данных и исскуственного интеллекта

РЕЗУЛЬТАТЫ

■ ОТЧЕТ О ПРОВЕРКЕ КОРРЕКТИРОВАЛСЯ: НИЖЕ ПРЕДСТАВЛЕНЫ РЕЗУЛЬТАТЫ ПРОВЕРКИ ДО КОРРЕКТИРОВКИ

ЗАИМСТВОВАНИЯ	▬	14.67%	ЗАИМСТВОВАНИЯ	▬	14.67%
ОРИГИНАЛЬНОСТЬ	▬▬▬▬▬▬	81.91%	ОРИГИНАЛЬНОСТЬ	▬▬▬▬▬▬	81.91%
ЦИТИРОВАНИЯ	▪	3.42%	ЦИТИРОВАНИЯ	▪	3.42%
САМОЦИТИРОВАНИЯ		0%	САМОЦИТИРОВАНИЯ		0%

ДАТА ПОСЛЕДНЕЙ ПРОВЕРКИ: 05.06.2022

ДАТА И ВРЕМЯ КОРРЕКТИРОВКИ: 05.06.2022 18:05

Модули поиска: ИПС Адилет; Библиография; Сводная коллекция ЭБС; Интернет Плюс; Сводная коллекция РГБ; Цитирование; Переводные заимствования (RuEn); Переводные заимствования по eLIBRARY.RU (EnRu); Переводные заимствования по Интернету (EnRu); Переводные заимствования издательства Wiley (RuEn); eLIBRARY.RU; СПС ГАРАНТ; Модуль поиска "КубГУ"; Медицина; Диссертации НББ; Перефразирования по eLIBRARY.RU; Перефразирования по Интернету; Перефразирования по коллекции издательства Wiley; Патенты СССР, РФ, СНГ; СМИ России и СНГ; Шаблонные фразы; Кольцо вузов; Издательство Wiley; Переводные заимствования

Работу проверил: Калайдина Г В
ФИО проверяющего

Дата подписи: 9 июня 2022

Подпись проверяющего

Чтобы убедиться
в подлинности справки, используйте QR-код,
который содержит ссылку на отчет.

Ответ на вопрос, является ли обнаруженное заимствование корректным, система оставляет на усмотрение проверяющего. Предоставленная информация не подлежит использованию в коммерческих целях.

Отзыв
на выпускную квалификационную работу магистра 01.04.02 «Прикладная математика и информатика» К.Р. Сеидова

РАЗРАБОТКА МНОГОАГЕНТНОЙ СИСТЕМЫ ИСКУСТВЕННОГО ИНТЕЛЕКТА СРЕДСТВАМИ DATA-ORIENTED DESIGN

Актуальность работы Сеидова К.Р. определяется тем, что на сегодняшний день индустрия компьютерных игр практически сравнялась с киноиндустрией, как по объему средств с продаж, так и по бюджетам, выделяемым на реализацию проектов, а аудитория игроков на различных платформах с каждым годом продолжает расти. При этом одним из ключевых аспектов является реализация системы искусственного интеллекта в игре для обеспечения правдоподобного поведения игровых сущностей с точки зрения игрока.

Цель дипломной работы – разработка современной функциональной системы автономных игровых сущностей достигнута.

При этом были решены поставленные в работе задачи:

– изучены принципы и особенности разработки с применением DOD, возможности платформы Unity поддержки DOD;

– выбраны подходы для реализации компонентов системы ИИ;

– программно реализованы подсистемы принятия решений для автономных агентов, расчёта перемещения для автономных агентов, планирования маршрута для автономных агентов;

– проведено сравнение производительности полученной системы с применением DOD относительно системы с применением ООП;

– продемонстрированы возможности разработанной системы.

При выполнении дипломной работы Сеидов К.Р. проявил способности к самостоятельному изучению современных сред программирования, к разработке программных продуктов с элементами ИИ. Разработал систему автономных игровых сущностей, пригодную для практического применения при создании видеоигр.

Считаю, что дипломная работа Сеидова К.Р. является целостным законченным исследованием актуальной темы, выполнена в соответствии с требованиями, предъявляемыми к квалификационным работам и, заслуживает оценки «отлично»

Руководитель квалификационной работы,
д.т.н., профессор кафедры анализа данных и искусственного интеллекта КубГУ А.А. Халафян

РЕЦЕНЗИЯ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(МАГИСТЕРСКУЮ ДИССЕРТАЦИЮ)

обучающегося ФГБОУ ВО КубГУ факультета компьютерных технологий и прикладной математики, направление подготовки 01.04.02 Прикладная математика и информатика, направленность «Математическое и информационное обеспечение экономической деятельности», Сеидова Кирилла Рустемовича
на тему «Разработка многоагентной системы искусственного интеллекта посредствам data-oriented design»

Представленная для рецензирования магистерская диссертация посвящена вопросу разработки ресурсно-эффективной системы искусственного интеллекта в контексте видеоигр.

Автор работы, Сеидов К. Р. провел анализ предметной области, изучил необходимую теорию по актуальным методам разработки видеоигрового искусственного интеллекта и принципы подхода data-oriented design, которые необходимы для разработки программы. Автором разработана расширяемая программная система, позволяющая реализовать широкий спектр поведений для внутриигровых сущностей.

Отличительной особенностью данной работы является то, что за счет применения подхода data-oriented design, многопоточных вычислений и эффективного использования кэш-памяти ЦПУ удалось добиться существенного прироста производительности относительно решения на базе объектно-ориентированного подхода.

По результатам рецензирования магистерской диссертации можно сделать вывод о том, что автор в ходе ее выполнения продемонстрировал наличие аналитических навыков, исследовательских способностей, а также навыков проектирования и разработки программных средств.

Работа представляет собой законченное исследование; цель работы достигнута, задачи решены. Полученные результаты могут в дальнейшем использоваться на практике.

Оценив степень сформированности компетенций в результате освоения образовательной программы, можно сделать вывод о готовности выпускника к профессиональной деятельности, предусмотренной ОПОП ВО по направлению 01.04.02 Прикладная математика и информатика. Магистерская диссертация выполнена на достаточно высоком профессиональном уровне и заслуживает оценки «отлично», а ее автор присвоения квалификации «Магистр» по направлению 01.04.02 Прикладная математика и информатика.

канд. пед. наук, доцент,
доцент кафедры информационных
технологий ФГБОУ ВО КубГУ _____ Е.О. Добровольская