МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»

Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

01.03.02 Прикладная математика и информатика
(код и наименование направления подготовки, специальности)

Компьютерные технологии и математическое моделирование
(направленность (профиль)/специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему Разработка интеллектуальных агентов для мобильной игры на платформе Unity

Обучающийся А.А. Ермолаев
(И.О. Фамилия) (личная подпись)

Руководитель к.т.н., доцент Г.А. Тырыгина
(ученая степень, звание, И.О. Фамилия)

Консультант к.п.н., доцент Т.С. Якушева
(ученая степень, звание, И.О. Фамилия)

Тольятти 2023

Аннотация

Тема выпускной квалификационной работы – «Разработка интеллектуальных агентов для мобильной игры на платформе Unity».

Работа посвящена анализу методов разработки и функционирования интеллектуальных агентов, а также созданию прототипа игрового интеллектуального агента при разработке мобильного приложения.

Объект работы – интеллектуальные агенты.

Предмет работы – использование интеллектуальных агентов при разработке мобильной игры.

Цель работы – проанализировать существующие методы создания интеллектуальных агентов и применить их в разработке интеллектуального агента для мобильного приложения.

Данная работа состоит из ведения, трех глав, заключения и списка литературы.

Во введении определяются актуальность темы, цель и задачи, поставленные в работе.

В первой главе описываются общие принципы работы интеллектуальных агентов и различные подходы к их разработке, преимущества и недостатки.

Во второй главе описывается процесс разработки интеллектуального агента, алгоритмы его действий и формулы, задающие принципы его функционирования.

Третья глава посвящена анализу разработанной программы.

Выпускная квалификационная работа содержит пояснительную записку объемом 47 страниц, 13 рисунков, 2 таблицы и список используемой литературы, состоящий из 21 источника.

Abstract

The topic of the graduate qualification work is "Development of intelligent agents for a mobile game on the Unity platform".

The work is devoted to the analysis of methods of development and functioning of intelligent agents, as well as the creation of a prototype of a game intelligent agent in the development of a mobile application.

The object of the work is intelligent agents.

The subject of the work is the use of intelligent agents in the development of a mobile game.

The purpose of the work is to analyze the existing methods of creating intelligent agents and apply them in the development of intelligent agent for mobile application.

This work consists of the introduction, three chapters, conclusion and list of references.

The introduction defines the relevance of the topic, the goal and objectives of the work.

The first chapter describes the general principles of intelligent agents and the different approaches to their development, advantages and disadvantages.

The second chapter describes the process of developing an intelligent agent, the algorithms of its actions and formulas that define the principles of its functioning.

The third chapter is devoted to the analysis of the developed program.

The graduate qualification work contains an explanatory note of 47 pages, 13 figures, 2 tables and a list of references consisting of 21 sources.

Оглавление

Введение

Разработка интеллектуальных агентов для мобильных игр - это одна из ключевых тем в современной игровой индустрии. С каждым годом все больше и больше людей начинают играть в мобильные игры, которые становятся все более популярными. Поэтому важно создавать продукты высокого качества с нестандартным игровым процессом, чтобы удовлетворить растущий спрос. Проработка интеллектуальных агентов позволяет улучшить игровой опыт и общее впечатление о разработанном проекте. Внедрение интеллектуальных агентов также поможет игре выделиться на рынке среди других мобильных игр и привлечь больше пользователей – этим и определяется актуальность данной работы.

Объектом исследования являются интеллектуальные агенты в играх.

Предметом исследования данной работы являются интеллектуальные агенты для мобильной игры на платформе Unity.

Целью данной работы является разработка интеллектуальных агентов для мобильной игры на платформе Unity.

Для достижения поставленной цели в выпускной квалификационной работе были определены следующие задачи:

Анализ существующих методов и подходов к разработке интеллектуальных агентов для мобильных игр.

Разработка прототипа интеллектуального агента для мобильной игры на основе выбранных методов и подходов.

Тестирование разработанного прототипа на соответствие заданным требованиям и оценка его эффективности.

Выпускная квалификационная работа содержит пояснительную записку объемом 47 страниц, 13 рисунков, 2 таблицы и список используемой литературы, состоящий из 21 источника.

Глава 1 Анализ существующих подходов в реализации интеллектуальных агентов

1.1 Принципы работы интеллектуальных агентов

Игровой искусственный интеллект (ИИ) преимущественно занимается выбором действий игровых персонажей в зависимости от текущих условий. Это важная область искусственного интеллекта, находящая широкое применение в различных компьютерных играх. В литературе по искусственному интеллекту это называется управлением "интеллектуальными агентами". Агентом в рамках игры может выступать не только персонаж, но и машина, робот или даже целая группа сущностей, страна или цивилизация.

Действия, которые ИИ выбирает, зависят от текущей ситуации и состояния игрового мира. Игровой ИИ анализирует данные, полученные из игровой среды, включая информацию о персонажах, предметах, препятствиях и других аспектах игры. Используя эту информацию, ИИ может принимать решения о том, какие действия следует совершить, чтобы достичь своей цели или выполнить задание [9].

В любом случае, объект, управляемый ИИ, следит за своим окружением, принимает решения на основе этого анализа и действует соответственно. Этот процесс иногда называют циклом "восприятие-мышление-действие" (Sense/Think/Act) [4].

На первом этапе, агент распознаёт или получает информацию об окружении, которая может влиять на его поведение. Эта информация может быть получена от других агентов, среды или быть замечена самим агентом. Такая информация может включать в себя опасности, находящиеся поблизости, собираемые предметы, важные точки и другие параметры, которые могут повлиять на принятие решения агентом.

На втором этапе, агент использует полученную информацию о своем окружении, чтобы принять решение о том, как поступить в ответ. Например, он может решить, достаточно ли безопасно собрать предметы, стоит ли ему сражаться или лучше сначала спрятаться. Для этого агент может использовать различные алгоритмы принятия решений, такие как искусственные нейронные сети или методы машинного обучения.

На третьем этапе, агент выполняет действия для реализации своих решений. Например, он может начать двигаться по маршруту к врагу или к предмету, который ему нужен. Для этого агент может использовать различные алгоритмы планирования движения, такие как алгоритм A* или метод Монте-Карло.

После выполнения действий, ситуация может измениться в зависимости от действий других агентов и персонажей. Поэтому, цикл начинается заново с новыми данными, которые агент должен учесть в своих действиях.

В реальном мире задачи, связанные с восприятием, являются одними из наиболее сложных для искусственного интеллекта. Например, для беспилотных автомобилей необходимо получать изображения дороги, находящейся перед ними, и комбинировать их с другими данными, такими как радар и лидар, пытаясь интерпретировать то, что они видят. Это означает, что необходимо обрабатывать большой объем данных реального мира, таких как фотографии дороги или несколько кадров видео, для извлечения семантической информации, например, "в 20 метрах передо мной есть ещё одна машина". Эти задачи называются задачами классификации.

Но в играх "восприятие" не является такой сложной задачей, как в реальном мире, потому что оно интегрировано в симуляцию. Например, игра знает, что враг находится перед игроком, и не нужно выполнять алгоритмы распознавания изображений для его обнаружения. Таким образом, "восприятие" в играх упрощено, и основной сложностью является реализация "мышления" и "действий".

Несмотря на то, что "восприятие" в играх упрощено, оно все еще представляет собой важную задачу для искусственного интеллекта. В играх также могут быть сложности, связанные с восприятием окружающей среды. Например, игрок может находиться в темной комнате, где не видно объектов. В этом случае игра должна учитывать это и предоставлять игроку дополнительные средства для "восприятия" окружающей среды, такие как фонарик или карта [3].

В целом, искусственный интеллект является мощным инструментом, который может изменить мир в лучшую сторону. Но для достижения этой цели его необходимо дальше развивать и совершенствовать. Одной из важных задач является усовершенствование "восприятия" окружающей среды, чтобы искусственный интеллект мог более эффективно работать в реальном мире и давать людям больше возможностей для улучшения своей жизни.

1.2 Особенности игровых интеллектуальных агентов

ИИ является программным обеспечением, которое позволяет компьютерным противникам действовать в игре так, как будто они являются настоящими игроками. Однако при создании игрового ИИ необходимо учитывать определенные ограничения, которые могут повлиять на его эффективность и реалистичность [5].

Первым ограничением является то, что игровой ИИ, в отличие от алгоритма машинного обучения, обычно не тренируется заранее. Например, создание нейронной сети для наблюдения за десятками тысяч игроков, чтобы найти наилучший способ играть против них, при разработке игры может быть практически невозможным. Поэтому дизайнеры игры должны определить, какой тип поведения должен иметь ИИ, чтобы игроки ощущали,

что соревнуются с человекоподобными противниками, и игра давала им удовольствие и вызывала интерес.

Второе ограничение заключается в том, что часто к ИИ предъявляют требование реалистичного поведения. Программа AlphaGo оказалась намного лучше, чем люди, но выбранные ей ходы настолько далеки от традиционного понимания игры, что опытные противники говорили о ней как об игре против инопланетянина. Если игра симулирует противника-человека, то обычно это нежелательно, поэтому ИИ должен принимать правдоподобные решения, а не идеальные. Для достижения этой цели могут применяться различные алгоритмы, такие как алгоритмы поиска, машинного обучения и другие [18].

Третье ограничение заключается в том, что ИИ должен выполняться в режиме реального времени. Это означает, что алгоритм не может монополизировать ресурсы процессора на длительное время. Большинство игр имеют ограниченный бюджет времени на выполнение всех операций для следующего кадра графики, например, 16-33 миллисекунды. Если ИИ не может принимать решения в установленные временные рамки, то это может негативно повлиять на игровой процесс и вызвать неудобства у игроков.

Четвертым ограничением является то, что идеальным решением является использование системы, которая зависит от данных, а не жестко заданной. Такой подход позволяет быстрее вносить изменения, что упрощает задачу дизайнерам игры. Благодаря этому ИИ может быстрее адаптироваться к изменению условий игры, что позволяет создавать более эффективные и реалистичные противников.

Учитывая все эти ограничения, можно рассматривать простые подходы к созданию ИИ, которые реализуют весь цикл «восприятие-мышление-действие» способами, обеспечивающими эффективность и позволяющими дизайнерам игры выбирать сложные поведения, похожие на действия человека. Для этого могут использоваться различные алгоритмы, такие как алгоритмы поиска, нейронные сети, генетические алгоритмы и другие.

1.3 Методы реализации интеллектуальных агентов в играх

1.3.1 Жестко заданные условные конструкции

Действия интеллектуального агента могут быть жестко регламентированы единым алгоритмом действий. Данный подход отлично работает в рамках примитивных игр. Для наглядности рассмотрим пример игры «Pong», интерфейс которой представлен на рисунке 1.

В этой игре вашей задачей будет управление движением ракетки, чтобы отбивать мячик и не допускать его пролета мимо ракетки. Правила игры похожи на правила тенниса - вы проигрываете, если пропустили мячик. При этом для компьютерного соперника задача выбора направления движения ракетки остается относительно простой.

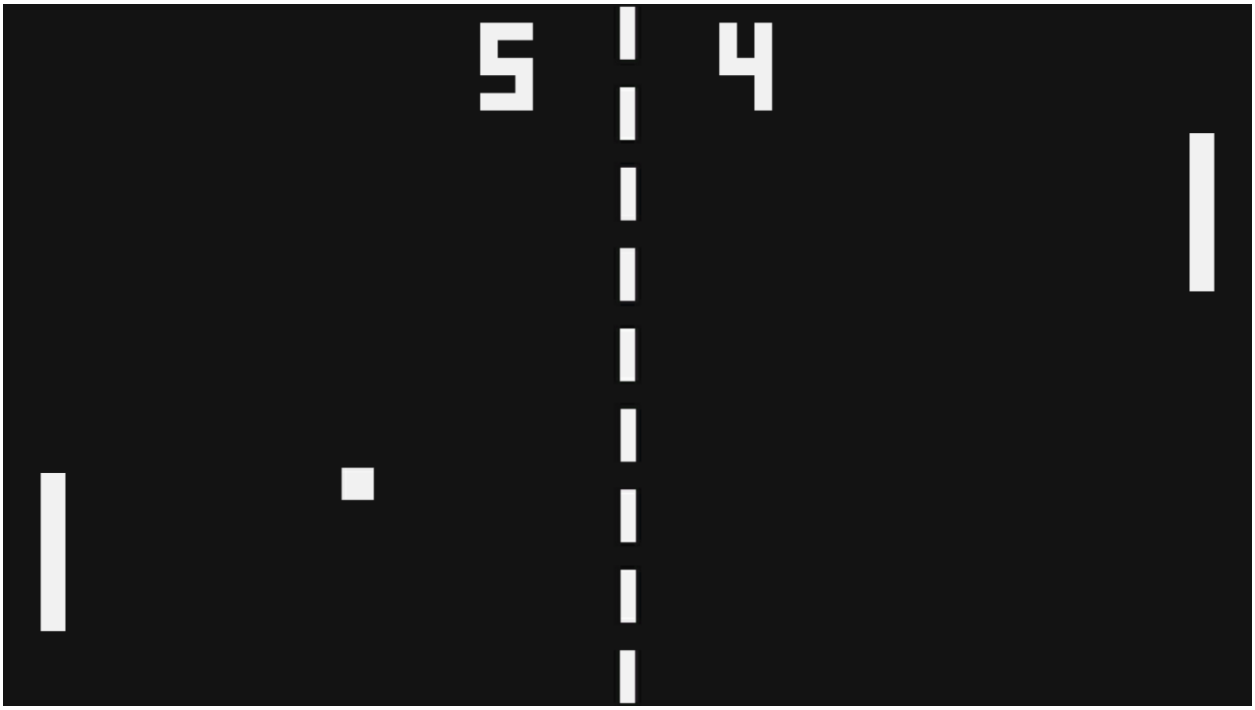


Рисунок 1 – Интерфейс игры «Pong».

Чтобы создать ИИ для управления ракеткой, можно использовать простой алгоритм, который позволит двигать ракетку так, чтобы она находилась под мячом.

Простой алгоритм для реализации этого может выглядеть следующим образом:

```
в каждом кадре пока игра выполняется
  если мячик правее ракетки
        переместить ракетку вправо
  иначе
        переместить ракетку влево
```

Этот алгоритм является идеальным решением для ИИ-игрока в Pong, если считать, что ракетка может двигаться с не меньшей скоростью, чем мяч. Однако, этот подход имеет свои ограничения, т.к. он использует только два оператора: "если" и "иначе" для восприятия мира, движения и принятия решения.

Операторы "если" и "иначе" здесь играют роль "восприятия" мира, т.к. определяют расположение мяча и ракетки и используют эту информацию для принятия решений. В данном случае, два оператора "если" используются для принятия решения о движении ракетки влево или вправо.

Таким образом, вышеназванные операторы в данном случае играют роль "восприятия" и "мышления". "Действие" заключается в движении ракетки. В зависимости от реализации игры, это может быть мгновенное перемещение ракетки или изменение ее скорости и направления.

Подобные подходы часто называются "реактивными", т.к. они используют простой набор правил, который реагирует на состояние мира и мгновенно принимает решение о том, как действовать [7].

1.3.2 Деревья решений

На первый взгляд может показаться, что использование дерева решений не дает явных преимуществ перед операторами "если" из предыдущего раздела. Однако, это не совсем верно.

Это универсальная система, где каждое принятое решение имеет строго одно условие и два возможных результата. Такой подход позволяет разработчикам создавать искусственный интеллект на основе данных, представленных в виде дерева решений, вместо жесткой привязки его к коду [21].

На рисунке 2 представлена иллюстрация древа решений на примере игры «Pong».

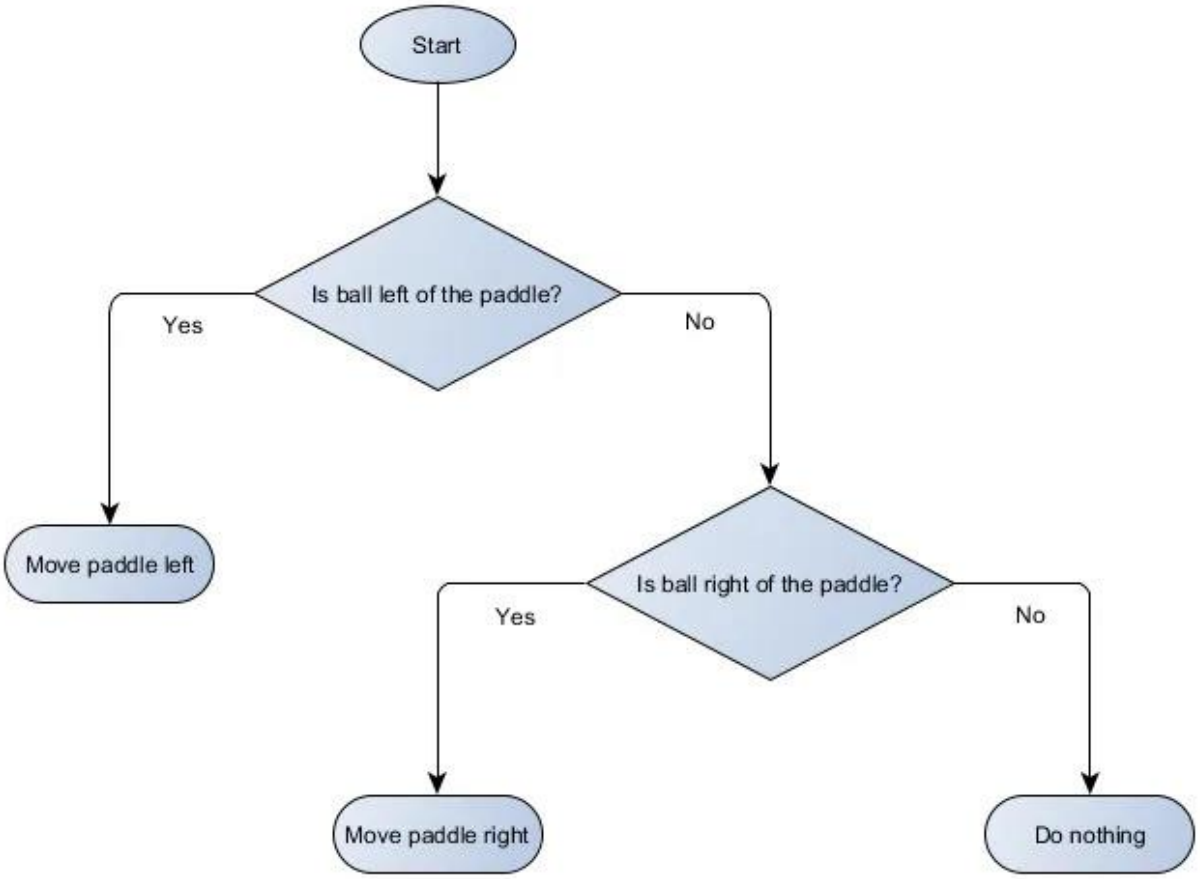


Рисунок 2 – Древо решений на примере игры «Pong».

1.3.3 Конечные автоматы

Возможности простых реактивных систем ограничены, и в некоторых случаях они не могут решить поставленную задачу. К примеру, когда требуется принимать различные решения на основе текущей деятельности агента, условия могут быть неудобными для представления в виде кода. В других случаях, количество условий настолько велико, что даже создание

дерева решений или скрипта неэффективно. Также, необходимо предусмотреть возможные изменения ситуации и оценить их влияние на будущие решения [20]. Для решения таких задач требуются более сложные алгоритмы принятия решений.

Конечный автомат (КА, finite state machine, FSM) — это способ описания состояния объекта и его возможных переходов между этими состояниями [17]. Например, один из наших ИИ-агентов может находиться в одном из нескольких возможных состояний, а количество таких состояний конечно, отсюда и название. Примером конечного автомата из реального мира может служить множество огней светофора, переключающееся с красного на жёлтый, затем на зелёный, и снова обратно. В разных местах может быть разная последовательность огней, но принцип остается одинаковым: каждое состояние имеет свой смысл ("стой", "едь", "стой, если возможно" и т.д.), и в любой момент времени объект находится только в одном состоянии. Переходы между состояниями основаны на простых правилах. Пример конечного автомата в виде графа представлен на рисунке 3.

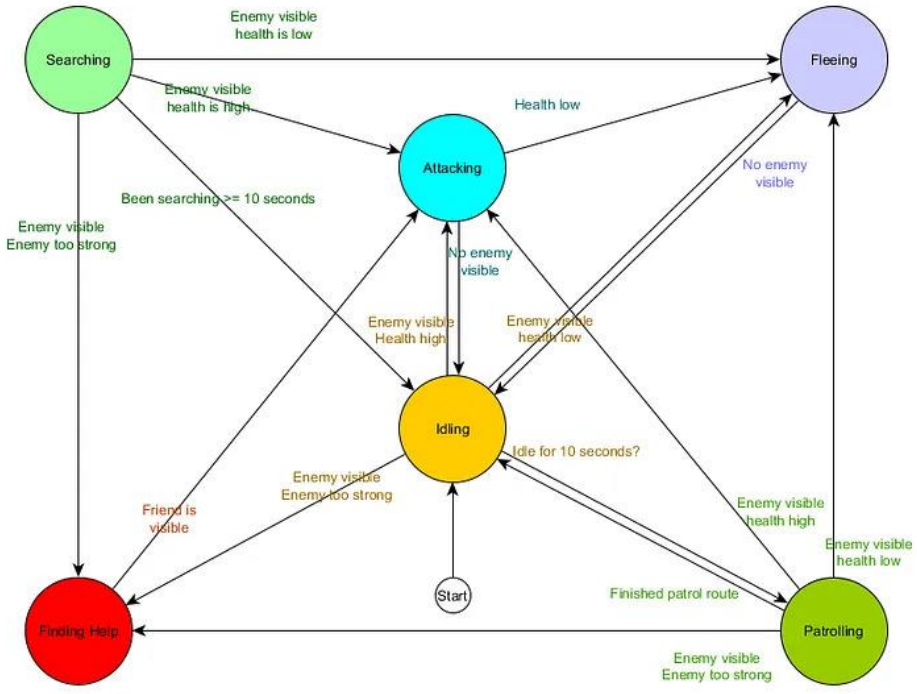


Рисунок 3 - Пример конечного автомата в виде графа.

1.3.4 Деревья поведений

FSM является инструментом, который позволяет создавать достаточно сложные множества поведений довольно интуитивно понятным способом. Однако, сразу заметно, что принятие решений в виде правил переходов тесно связано с текущим состоянием. Во многих играх и приложениях требуется именно это. А аккуратное использование иерархии состояний позволяет снизить количество дублируемых переходов [10].

Однако, иногда нам нужны правила, применяемые вне зависимости от текущего состояния, или применяемые почти во всех состояниях. В таких случаях идеальным решением становятся деревья поведений. Деревья поведений позволяют нам создавать множества поведений, которые не зависят от текущего состояния и могут быть применены в любой момент времени. Использование иерархии состояний в деревьях поведений позволяет снизить количество дублируемых переходов и сделать код более читаемым и легким для поддержки.

Существует несколько способов реализации деревьев поведений, но их суть для большинства одинакова и очень похожа на упомянутое выше дерево решений. Алгоритм начинает работу с «корневого узла», и в дереве есть узлы, обозначающие решения или действия. Однако, в деревьях поведений есть ключевые отличия.

Узлы теперь возвращают одно из трёх значений: «успешно» (если работа выполнена), «безуспешно» (если выполнить её не удалось), или «выполняется» (если работа всё ещё выполняется и полностью не закончилась успехом или неудачей).

Теперь у нас нет узлов решений, в которых мы выбираем из двух альтернатив, а есть узлы-«декораторы», имеющие единственный дочерний узел. Если они «успешны», то выполняют свой единственный дочерний узел. Узлы-декораторы часто содержат условия, определяющие, окончилось ли выполнение успехом (значит, нужно выполнить их поддерево) или неудачей (тогда делать ничего не нужно). Также они могут возвращать «выполняется».

Выполняющие действия узлы возвращают значение «выполняется», чтобы обозначить происходящие действия. Это позволяет создавать действия, которые могут выполняться параллельно, не блокируя выполнение других действий [15].

Пример древа поведения представлен на рисунке 4.

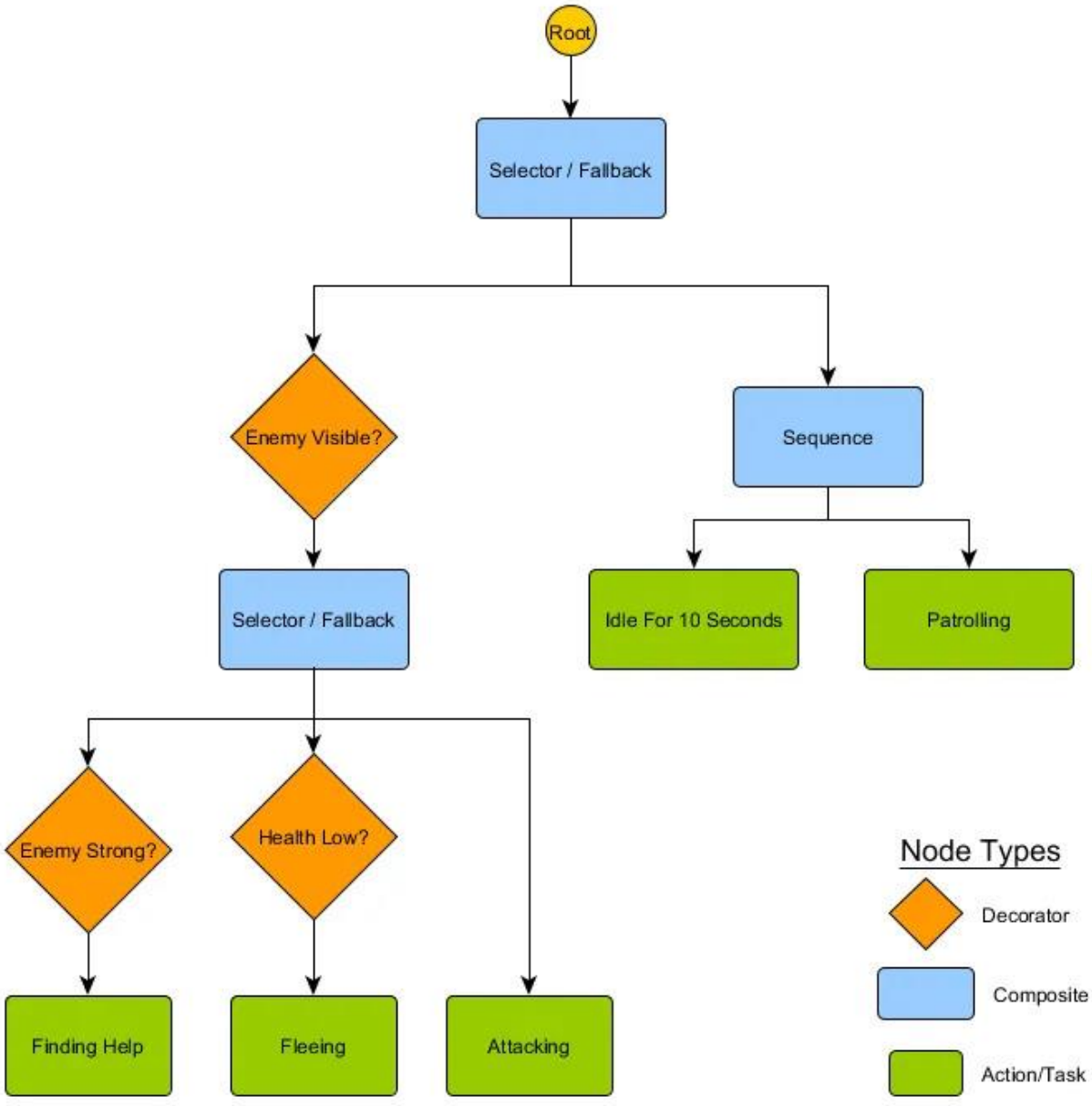


Рисунок 4 – Пример древа поведения.

1.3.5 Системы на основе полезности (Utility)

В некоторых играх, особенно в играх с большим количеством различных действий, создание централизованных правил переходов может быть слишком сложным. В таких случаях системы на основе полезности могут стать более простым и подходящим решением, чем явный набор выборов или дерево потенциальных действий [3].

Системы на основе полезности - это такие системы, в которых агент выбирает действие на основе относительной полезности каждого действия. Полезность здесь определяется как произвольная мера важности или желательности для агента выполнения данного действия.

Для вычисления полезности действия на основании текущего состояния агента и его окружения, агент использует функции полезности. Этот подход напоминает конечный автомат, за исключением того, что переходы определяются оценкой каждого потенциального состояния, включая текущее состояние.

В общем случае, при выборе действия мы предпочитаем переход к наиболее ценному действию. Однако, для большей вариативности и уникальности, мы можем использовать взвешенный случайный выбор, выбирать случайное действие из пяти лучших (или другого количества), и т.д.

Стандартная система на основе полезности назначает допустимый интервал значений полезности для каждого действия, например, от 0 (совершенно нежелательно) до 100 (абсолютно желательно). У каждого действия может быть набор факторов, влияющих на способ вычисления значения полезности [11].

Таким образом, системы на основе полезности предоставляют гибкое решение для выбора действий в играх и других ситуациях, где требуются множественные действия с различной значимостью.

Пример системы на основе полезности представлен в виде таблицы на рисунке 5.

Действие	Вычисление полезности
Поиск помощи	Если враг видим и враг силён, а здоровья мало, то возвращаем 100, в противном случае возвращаем 0
Бегство	Если враг видим и здоровья мало, то возвращаем 90, иначе возвращаем 0
Нападение	Если враг видим, возвращаем 80
Ожидание	Если находимся в состоянии ожидания и уже ждём 10 секунд, возвращаем 0, в противном случае 50
Патрулирование	Если находимся в конце маршрута патрулирования, возвращаем 0, в противном случае 50

Рисунок 5 – Пример системы на основе полезности.

Краткое сравнение всех упомянутых в главе подходов представлено на таблице 1.

Таблица 1 – Краткое сравнение рассмотренных подходов.

Название подхода	Достоинства	Недостатки
Реактивный	Простота и скорость	Ограниченность
Деревья решений	Простота и скорость, отсутствует жесткая привязка к коду	Ограниченность
Конечные автоматы	Возможность работы с состояниями объекта	Невозможность использования правил вне зависимости от текущего состояния

Продолжение таблицы 1

Название подхода	Достоинства	Недостатки
Деревья поведений	Возможность использования правил вне зависимости от текущего состояния, удобство работы с кодом	Сложность работы с большим количеством действий, или при их схожести
Системы на основе полезности	Гибкость в работе с большим количеством действий, или при их схожести	Необходимость работы с точными коэффициентами

Выводы по первой главе

В данной главе были рассмотрены основные подходы к созданию интеллектуальных агентов для компьютерных игр, а также их преимущества и недостатки.

Было выявлено, что существуют различные типы агентов, такие как реактивные, деревья решений и поведения, конечные автоматы, а также системы на основе полезности.

На основе проведенного анализа можно сделать вывод, что для создания интеллектуальных агентов для мобильных игр на платформе Unity необходимо выбрать наиболее подходящие под задачу алгоритмы и методы, а также учитывать ограничения ресурсов мобильных устройств.

Глава 2 Разработка прототипа интеллектуальных агентов для мобильной игры на платформе Unity.

2.1 Создание математической основы игровых механик и алгоритмов действий интеллектуальных агентов.

2.1.1 Описание игровых механик.

Для разработки интеллектуальных агентов для мобильной игры на платформе Unity требуется создание математической основы игровых механик и алгоритмов действий, которые позволят агентам принимать решения на основе данных, полученных из игрового мира. Разработка такой математической модели является важным шагом в создании интеллектуальной системы, которая сможет оптимально выполнять свои задачи, взаимодействуя с другими игровыми объектами и игроком.

Сама сцена представляет собой совокупность объектов, каждый из которых имеет свои уникальные характеристики и умения. Юниты являются игровыми единицами, воплощающими собой различных персонажей. Они разделены на две противоборствующие команды, расположенные на двух сторонах игрового поля. Каждая из сторон поля, в свою очередь, разделена на две линии: переднюю и дальнюю, каждая из которых может вместить в себя 2 юнита. Таким образом, в рамках разрабатываемой сцены может быть задействовано до четырёх юнитов с каждой стороны.

Юниты делятся на несколько видов, отличающихся друг от друга различными параметрами, такими как: вероятность успешной атаки, показатель защиты, уровень здоровья и многое другое. Параметры, используемых юнитов представлены на таблице 2.

В рамках противоборства юниты поочередно выполняют различные действия, такие как: нанесение противнику урона, защита союзников, усиление своих характеристик и многое другое. Цель состязания заключается

в том, чтобы уничтожить команду противника, снизив параметр Hlt каждого из юнитов противоположной команды до нуля.

Таким образом, игроки могут выбирать различные стратегии для достижения победы, в зависимости от того, какие юниты они выбрали, как они используют свои умения и как они взаимодействуют со своими персонаж

Для более детального понимания таблицы 2 необходимо проанализировать ее параметры и их значения.

Таблица 2 – Параметры юнитов разрабатываемой игровой сцены.

	U1	U2	U3	U4	U5	U6
Hlt	50	40	30	25	30	30
Dfc	15	10	5	8	10	25
Arm	60	20	0	0	50	0
Int	5	1	4	3	6	2
Atc1	15	15	10	15	12	15
AP1	10	0	100	0	10	100
Dmg1	10-15	8-12	5-10	8-12	8-12	12-17
Frm1	1	1	3	1	1	1
Trg1	1	2	3	1	1	1
Atc2	10	-	-	10	-	-
AP2	30	-	-	0	-	-
Dmg2	15-20	-	-	8-12	-	-
Frm2	1	2	4	1	-	-
Trg2	1	5	4	2	-	-

Первый параметр, «Hlt», отражает уровень урона, который юнит может получить, прежде чем будет уничтожен. Это означает, что при более высоком значении данного параметра, юнит может выдерживать больше урона и продолжать действовать в игре.

Второй параметр, «Dfc», отражает защитные умения персонажа. Данный параметр влияет на то, будет ли атакующее действие противника успешным. Чем выше значение параметра, тем больше вероятность, что атакующее действие будет неудачным.

Третий параметр, «Arm», отображает устойчивость юнита к урону, будь то броня или естественные особенности. Урон по юниту будет понижен на процент, определяемый данной характеристикой. Чем выше значение данного параметра, тем меньше урона будет получен юнитом.

Четвертый параметр, «Int», отображает инициативу юнита – то, насколько быстро он действует. Данная характеристика влияет на то, кто будет действовать раньше в рамках одного хода. Чем выше значение данного параметра, тем быстрее юнит действует.

Пятый параметр, «Atc1», отображает атакующие умения юнита при выполнении действия 1. Он влияет на то, с какой вероятностью атака будет успешна. Прочерк в данном параметре обозначает, что действие при выполнении успешно всегда. Чем выше значение данного параметра, тем больше вероятность успешного выполнения действия.

Шестой параметр, «AP1», отображает способности данной атаки (действия 1) к противодействию броне – параметру «Arm». Значение данного параметра снижает защиту на соответствующий процент. Чем выше значение данного параметра, тем больше уменьшается защита противника.

Седьмой параметр, «Dmg1», отображает разброс урона при выполнении действия 1. При успехе действия урон определяется случайно в рамках заданного диапазона. Чем выше значение данного параметра, тем больше разброс урона.

Восьмой параметр, «Frm1», отображает, какая формула должна использоваться для определения параметра ориентировочной эффективности действия.

Девятый параметр, «Trg1», отображает, какой юнит является целью данного действия:

– 1 – юнит из противоположной команды, находящийся на передней линии,

– 2 – любой юнит из противоположной команды,

– 3 – все юниты из противоположной команды,

– 4 – юнит из своей команды,

– 5 – сам юнит применяющий действие.

2.1.2 Структура игровой сцены

Игровой процесс разделён на раунды, состоящие из ходов. Ход – процесс действия отдельного юнита, по его окончании начинается ход следующего юнита. Когда все юниты выполнили по одному действию, заканчивается раунд, после чего начинается новый.

В начале каждого раунда определяется инициатива – последовательность, в которой юниты будут выполнять свои ходы. Также определяется такой параметр, как ориентировочная длительность сцены – он нужен для долгосрочного планирования интеллектуальными агентами своих действий. Так, если действие будет оказывать эффект на протяжении нескольких раундов, то при скором окончании сражения его эффект не сможет раскрыть свой потенциал. Интеллектуальные агенты должны учитывать данный фактор, т.к. в ином случае, ошибочность принятого решения будет бросаться в глаза игроку.

Теперь мы можем сформировать общую структуру действий игровой сцены, она представлена в виде блок-схемы на рисунке 6.

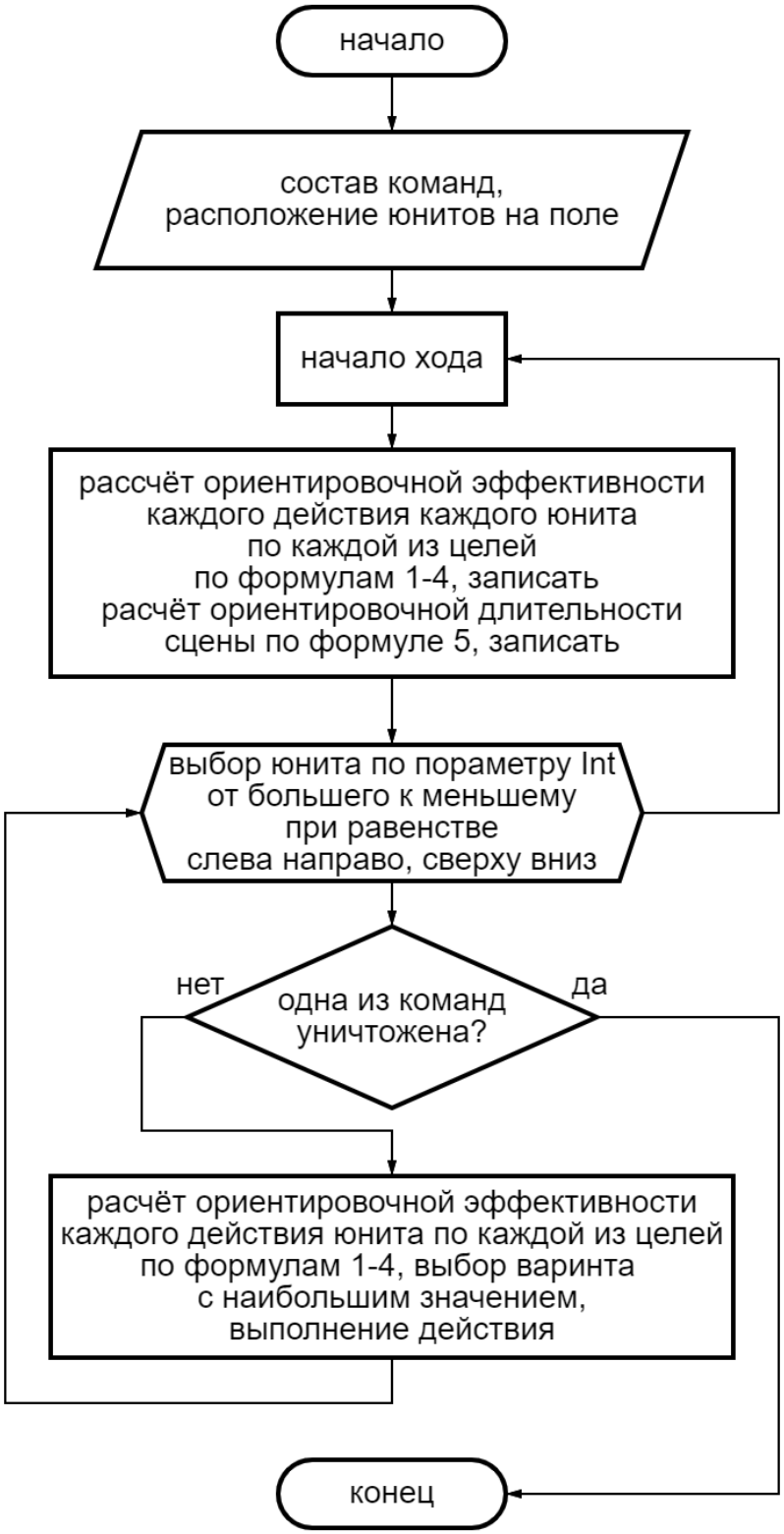


Рисунок 6 – Общая структура игровой сцены

На основе данной структуры будут строиться дальнейшие разработки.

2.1.3 Математическая основа игровых действий

Представим разработку модели для реализации игровой сцены сражения для мобильной игры.

Для того чтобы сделать игровой процесс более глубоким и интересным, нужно продумать большее количество различных действий, которые совершают юниты. В рамках данного прототипа, углубление в данный вопрос может показаться излишним, однако создание основы для дальнейшего развития различных игровых механик является необходимым. Таким образом, действия должны иметь не только прямое влияние на снижение параметра Hlt команды своего противника, что приближает победу, но и способы сделать это косвенно, и при этом не менее эффективно, чтобы разнообразить игровой процесс.

Существует множество способов углубить и усложнить игровой процесс: распределение наносимого урона по большему числу противников, воздействие на наносимый урон своих союзников и многое другое. Различные действия должны влиять на различные цели по-разному, отличаясь эффективностью. Это требует усложнения системы интеллектуальных агентов, так как для каждого типа действий необходима своя формула поиска ориентировочной эффективности - ED. Формулы для различных типов действий представлены ниже.

$$(Arm_b - Apx_a) \geq 0 \rightarrow \frac{Atcx_{Ua}^2}{Atcx_{Ua}^2 + Dfc_{Ub}^2} * \overline{Dmgx_{Ua}} * (Arm_b - Apx_a) = \\ = EDx_{UaUb} \quad (1)$$

$$(Arm_b - Apx_a) < 0 \rightarrow \frac{Atcx_{Ua}^2}{Atcx_{Ua}^2 + Dfc_{Ub}^2} * \overline{Dmgx_{Ua}} = EDx_{UaUb}$$

Первая формула необходима при использовании стандартного атакующего действия по одной цели. Главное в ней – учёт оборонительных способностей цели.

$$(Arm_b - Apx_a) \geq 0 \rightarrow \frac{(Atcx_{Ua} + \mathrm{M}_1)^2 * \frac{1}{2}}{(Atcx_{Ua} + \mathrm{M}_1)^2 + Dfc_{Ub}^2} \cdot \overline{(Dmgx_{Ua} + \mathrm{M}_2)} \cdot$$
$$\cdot (Arm_b - Apx_a) = EDx_{UaUb} \quad (2)$$
$$(Arm_b - Apx_a) < 0 \rightarrow \frac{(Atcx_{Ua} + \mathrm{M}_1)^2 * 1/2}{(Atcx_{Ua} + \mathrm{M}_1)\hat{}2 + Dfc_{Ub}^2} * \overline{(Dmgx_{Ua} + \mathrm{M}_2)}$$
$$= EDx_{UaUb}$$

Где $M_1 = 10; M_2 = 5$

Данная формула рассчитана на усиление последующей атаки. В случае, если защита противника слишком хороша – вместо нескольких малоэффективных действий, может быть лучше подготовить одно более эффективное. Именно здесь и начинает проявляться необходимость учёта долговременного планирования. При конфронтации одного юниты с одной стороны, и одного – с другой, имеющийся подход будет удобен, однако когда появляется необходимость учёта действий союзников – использование параметра ориентировочной длительности сцены становится необходимым. Так, при использовании данной формулы необходима проверка данного параметра, позволяющая определить, необходимо ли создавать фундамент для дальнейших действий, или данных подход несвоевременен.

$$(Arm_b - Apx_a) \geq 0 \rightarrow \sum_{i=1}^{N} \frac{Atcx_{Ua}^2}{Atcx_{Ua}^2 + Dfc_{Ub}^2} \cdot \overline{Dmgx_{Ua}} \cdot$$
$$\cdot (Arm_b - Apx_a) = EDx_{UaUb} \quad (3)$$
$$(Arm_b - Apx_a) < 0 \rightarrow \sum_{i=1}^{N} \frac{Atcx_{Ua}^2}{Atcx_{Ua}^2 + Dfc_{Ub}^2} \cdot \overline{Dmgx_{Ua}} = EDx_{UaUb}$$

Где $N = (1,2,3,4)$

Данная формула необходима, чтобы учесть действия, затрагивающие несколько целей. Когда бой идёт против большого количества юнитов – такое действие с высокой долей вероятности будет эффективно, с другой стороны, если противник будет лишь один – смысла в его применении будет куда меньше. Интеллектуальные агенты должны учитывать эти факторы.

$$(Arm_b - Apx_a) \geq 0 \rightarrow$$

$$\rightarrow \left(\begin{array}{c} \dfrac{(Atcx_{Ua} + \mathrm{M}_1)^2}{(Atcx_{Ua} + \mathrm{M}_1)^2 + Dfc_{Ub}^2} \cdot \overline{(Dmgx_{Ua} + \mathrm{M}_2)} \cdot \\ \cdot (Arm_b - Apx_a) - \dfrac{Atcx_{Ua}^2}{Atcx_{Ua}^2 + Dfc_{Ub}^2} \cdot \overline{Dmgx_{Ua}} \cdot (Arm_b - Apx_a) \end{array} \right) \cdot 3 =$$

$$= EDx_{UaUb} \quad (4)$$

$$(Arm_b - Apx_a) < 0 \rightarrow \left(\begin{array}{c} \dfrac{(Atcx_{Ua} + \mathrm{M}_1)^2}{(Atcx_{Ua} + \mathrm{M}_1)^2 + Dfc_{Ub}^2} \cdot \\ \cdot \overline{(Dmgx_{Ua} + \mathrm{M}_2)} - \dfrac{Atcx_{Ua}^2}{Atcx_{Ua}^2 + Dfc_{Ub}^2} \cdot \overline{Dmgx_{Ua}} \end{array} \right) \cdot$$

$$\cdot 3 = EDx_{UaUb}$$

Где $M_1 = 5;\ M_2 = 5$

Эта формула позволяет расширить возможности формулы 2), увеличивая длительность воздействия эффекта. В данном случае эффект длится 3 раунда. Разница может показаться незначительной, однако спектр возможностей расширяется крайне сильно. С одной стороны, согласно таблице 2, действие, соответствующее данной формуле, может быть применено не только к действующему юниту, но и к любому из его союзников, что даёт простор для различных комбинаций, с другой – увеличение длительности воздействия расширяет горизонт планирования, что усложняет грамотный расчёт эффективности.

Проблему расширения горизонта планирования можно решать различными способами. Вопрос заключается в том, насколько много факторов можно и необходимо учесть. Так, различные комбинации действий могут быть крайне эффективны, однако чтобы учесть такую возможность, необходимо позволить интеллектуальным агентам учитывать не только средние показатели других юнитов, но и все действия, которые могут предпринимать остальные участники сцены по воздействию друг на друга. Так, в развитой системе интеллектуальных агентов, рассчитываться должны не актуальные параметры, а такие, какими они могут стать после воздействия на них остальных юнитов. И хотя такой подход позволяет сделать интеллектуальные агенты куда более эффективными, математическая модель, как и её программная реализация вследствие данного усложнения становятся крайне громоздкими, что негативно влияет на динамику игрового процесса и, как следствие, но положительное восприятие итогового продукта [16]. В таком случае необходим подход более простой.

Если прерогативу на сложные комбинации оставить пользователю, то учёт времени воздействия просто необходим, по прежде описанным причинам. Так мы подходим к потребности создания формулы, которая поможет нам определить, ориентировочную длительность сцены.

$$\frac{TH_a}{TE_b} \geq \frac{TH_b}{TE_a} \rightarrow AD = \frac{TH_b}{TE_a} \qquad (5)$$

$$\frac{TH_a}{TE_b} < \frac{TH_b}{TE_a} \rightarrow AD = \frac{TH_a}{TE_b}$$

Где: TH – сумма параметра Hlt юнитов команды

TE – сумма ориентировочной эффективности ED юнитов команды

AD – ориентировочная длительность сцены

Основной вопрос принятия решений проработан, однако остаётся выполнить само действие. Формулы планирования действия и самого действия напрямую зависят друг от друга, потому уже на этапе принятия решений используются фрагменты формулы их реализации. Ключевой же разницей является элемент случайности.

Первоочередной задачей в данном вопросе является определение того, будет ли действие выполнено успешно. Вероятность успеха определяется формулой 6.

$$\frac{Atcx_{Ua}^2}{Atcx_{Ua}^2+Dfc_{Ub}^2} = S \quad (6)$$

Где S – вероятность успешного действия

Далее необходимо определить итоговый урон от атаки, формула которого представлена далее.

$$(Arm_b - Apx_a) \geq 0 \rightarrow \left(1 - (Arm_b - Apx_a)\right) \cdot Dmgx_a = TD_{ab} \quad (7)$$

$$(Arm_b - Apx_a) < 0 \rightarrow Dmgx_a = TD_{ab}$$

Где: $Dmgx_a$ – случайное значение в диапазона параметра,

TD_{ab} - итоговый урон от действия юнита a, вычитаемый из параметра Hlt юнита b

Все эти формулы послужат основой для дальнейшей программной реализации интеллектуальных агентов.

2.2 Программная реализация интеллектуальных агентов на платформе «Unity»

2.2.1 Особенности среды разработки «Unity»

Разработка мобильных приложений – это процесс создания программных приложений для мобильных устройств, таких как смартфоны, планшеты и носимые гаджеты. С ростом мобильной индустрии разработка мобильных приложений стала важной областью для разработчиков программного обеспечения и бизнеса.

Разработка мобильных приложений требует различных навыков, включая программирование, дизайн пользовательского интерфейса и программирование мобильных устройств. Есть множество языков программирования, фреймворков и инструментов, которые можно использовать в разработке мобильных приложений. Одним из популярных инструментов является Unity - игровой движок, который используется для создания мобильных игр для разных платформ, таких как iOS, Android и другие. Unity известна своей простотой использования и кроссплатформенными возможностями, которые делают ее популярным выбором для разработчиков игр.

Процесс разработки мобильных приложений включает несколько этапов: создание идеи, дизайн, разработку, тестирование и развертывание. На стадии создания идеи разработчик определяет цель приложения и целевую аудиторию. На стадии дизайна создаются каркасы и разрабатывается пользовательский интерфейс. На этапе разработки происходит кодирование приложения, а на стадии тестирования разработчик проверяет наличие ошибок и гарантирует, что приложение работает должным образом. Наконец, на стадии развертывания приложение становится доступным для использования.

Unity - это игровой движок с мощным инструментарием, который позволяет создавать высококачественные игры для различных платформ. Ниже перечислены некоторые инструменты, которые предоставляет Unity:

Редактор сцен

Unity предоставляет интуитивно понятный редактор сцен, который позволяет создавать и редактировать игровые уровни, добавлять объекты, настраивать свойства объектов и создавать сложные игровые механики. Редактор сцен имеет большой набор инструментов и возможностей, таких как добавление света, тумана, тени, и многих других элементов, которые позволяют создавать уникальные игровые миры [6].

Редактор анимаций

Unity имеет встроенный редактор анимаций, который позволяет создавать и редактировать анимации объектов. Редактор анимаций имеет набор инструментов для создания и редактирования анимаций, а также возможность просмотра анимаций в режиме реального времени. Редактор анимаций также позволяет создавать сложные системы анимации, такие как Blending, Layering и State Machines, которые позволяют создавать более сложные и интересные анимации [19].

Редактор скриптов

Unity также предоставляет редактор скриптов, который позволяет создавать и редактировать скрипты, используемые для создания игровых механик и поведения объектов. Редактор скриптов имеет интегрированную среду разработки и поддерживает несколько языков программирования, включая C#, JavaScript и Boo. Редактор скриптов также имеет большой набор библиотек и инструментов, которые упрощают процесс разработки и позволяют создавать более сложные игровые системы [12].

Редактор шейдеров

Unity имеет встроенный редактор шейдеров, который позволяет создавать и редактировать шейдеры, используемые для создания реалистичных и красивых графических эффектов. Редактор шейдеров имеет

интуитивно понятный интерфейс и поддерживает несколько языков программирования, включая Cg и HLSL. Редактор шейдеров также позволяет создавать сложные системы шейдеров, такие как параллакс-эффекты, блики и тени, которые позволяют создавать более реалистичную графику.

Редактор аудио

Unity предоставляет редактор аудио, который позволяет создавать и редактировать звуковые эффекты, музыку и диалоги. Редактор аудио имеет набор инструментов для создания и редактирования звуков, а также возможность просмотра и редактирования звуков в режиме реального времени. Редактор аудио также позволяет создавать сложные системы звуков, такие как 3D звук и Spatial Audio, которые позволяют создавать более реалистичную звуковую среду.

Инструменты разработки мобильных игр

Unity предоставляет набор инструментов для разработки мобильных игр, включая интеграцию с магазинами приложений, возможность создания адаптивного интерфейса и оптимизации игрового процесса для мобильных устройств. Инструменты разработки мобильных игр также включают в себя создание многопользовательских игр, интеграцию с социальными сетями и многое другое.

Инструменты разработки виртуальной реальности

Unity также предоставляет инструменты для разработки виртуальной реальности, включая интеграцию с VR-устройствами, возможность создания интерактивных объектов и оптимизации игрового процесса для VR-устройств. Инструменты разработки виртуальной реальности также включают в себя создание многопользовательских VR-игр, интеграцию с социальными сетями и многое другое.

2.2.2 Этапы разработки мобильного приложения

Разработка мобильного приложения на Unity - это сложный процесс, который требует умения работать с различными инструментами и технологиями. Понимание общей структуры и этапов разработки мобильного

приложения на Unity позволяет существенно упростить процесс создания приложения.

Первым шагом является установка Unity на ваш компьютер. Это можно сделать, перейдя на официальный сайт Unity и загрузив последнюю версию программы. Unity - это мощный инструмент для создания игр, и его использование будет важным для создания вашей будущей игры [14].

После установки Unity необходимо выбрать платформы, на которые будет выпущена игра. Выбор целевой платформы важен для оптимизации и тестирования игры. На данный момент поддерживаются iOS, Android и другие популярные мобильные ОС.

После создания проекта, можно начать разрабатывать игровой процесс. Различные инструменты в Unity помогут вам создать интересный и захватывающий игровой процесс. Это включает в себя создание игровых объектов, настройку игровых механик и взаимодействий между ними.

После завершения разработки игрового процесса, необходимо протестировать игру на разных устройствах, чтобы убедиться, что она работает корректно на всех платформах. Это важный шаг для того, чтобы убедиться, что игра будет запущена и будет работать корректно на всех устройствах.

И последним шагом является оптимизация игры, чтобы она работала быстро и без сбоев на всех устройствах. Это включает в себя оптимизацию графики, звука и других элементов игры. При этом необходимо учитывать ограничения технических возможностей устройств и платформ, на которых будет выпущена игра [2].

Понимание общей структуры и этапов разработки мобильного приложения на Unity позволяет разработчикам оптимизировать свой рабочий процесс и существенно ускорить создание приложения. Более того, это знание позволяет улучшить качество приложения и сделать его более профессиональным [13].

2.2.3 Создание сцены и объектов

При разработке мобильной игры на Unity, создание сцены и добавление объектов является одним из ключевых этапов. Для данного прототипа началом стало создание пустой сцены и определения ее размера и разрешения. Размер и разрешение можно настроить в окне "Game" во вкладке "Resolution and Presentation". Было выбрано разрешение 1080x1920, так как разрабатываемая игра предназначена для мобильных устройств.

Далее, на сцену был добавлен объект "Main Camera", который будет следить за игровым полем. Таким образом, установлена камера, чтобы игровое поле было видно на экране устройства пользователя. Кроме того, были настроены параметры камеры, чтобы она могла следить за юнитами на поле боя.

Для создания поля боя, был выбран объект "Sprite", после чего был добавлен на сцену. Затем проводилась настройка его размера и позиции, чтобы он занимал всю область экрана. Настройки можно изменить в окне "Inspector".

Далее был создан объект "Canvas" для отображения клеток поля боя. Для этого нужно было воспользоваться функцией "Canvas " в меню "GameObject" и настроить его размер и шаг. Каждая клетка была отдельным объектом "Sprite", который был добавлен на сцену. Это позволило создать сетку для игры, на которой можно было бы разместить юниты. Изображение игрового пространства представлено на рисунке 7.

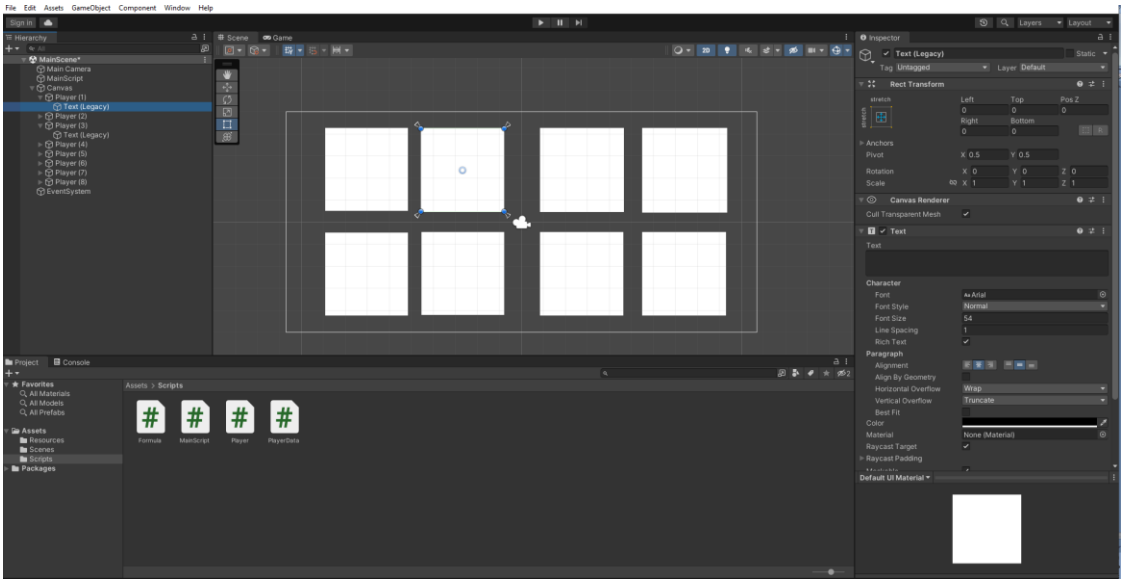


Рисунок 7 – Игровая сцена

Для добавления юнитов на поле боя, был создан новый объект "Sprite" и настроен его размер и внешний вид – на этапе прототипа они представляют собой лишь цифры, отображающие параметр Hlt. Затем были добавлены на сцену несколько копий этого объекта, чтобы создать несколько юнитов. Каждый юнит был отдельным объектом. Было реализовано несколько типов юнитов, для того чтобы одинаковые типы функционировали одинаково и не приходилось прописывать отдельную логику поведения для каждого из них.

Запущенная сцена с отображающимися показателями Hlt представлена на рисунке 8. Зелёным цветом отображен действующий юнит, в то время как красным – цель его действия.

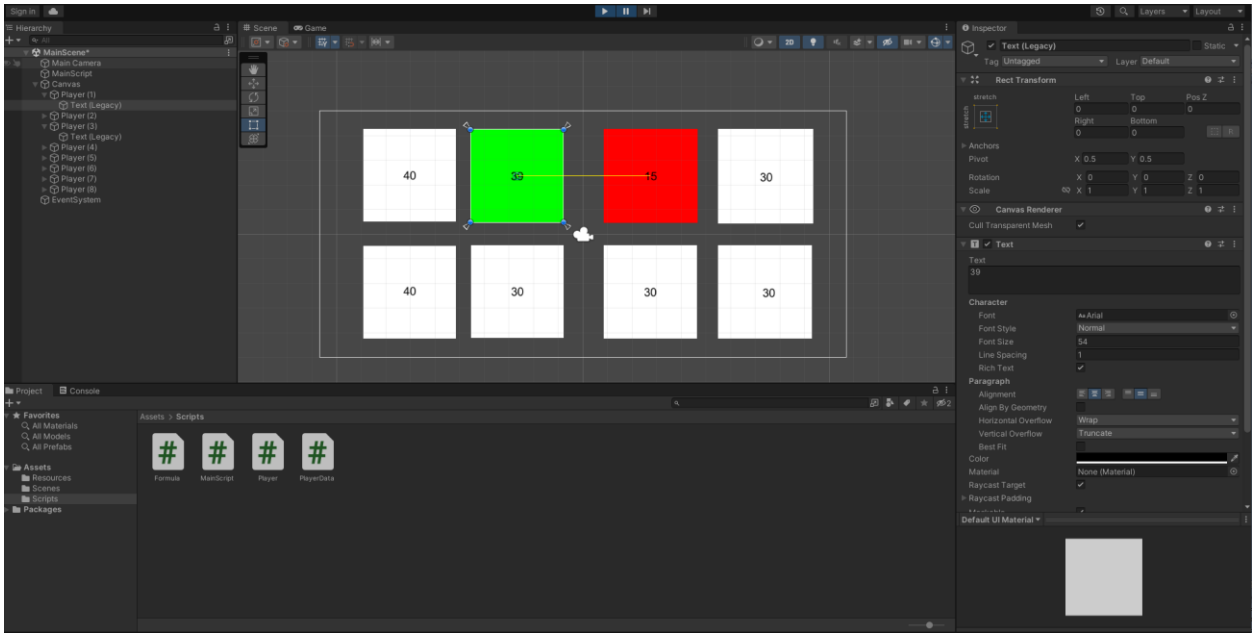


Рисунок 8 – Запущенная сцена

2.2.4 Создание логики поведения интеллектуальных агентов

Когда сцена готова, а объекты созданы и настроены, остаётся главное – выработать логику, согласно которой объекты будут взаимодействовать. Благодаря уже заложенной в первом параграфе данной главы основе, создать скрипты, которым будут следовать интеллектуальные агенты, становится куда проще. Итоговый вариант кода скриптов, описывающих общий сценарий игровой сцены представлен на рисунках 9, 10, 11, 12 и 13.

MainScript.cs

Прочие файлы

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using static UnityEngine.Mathf;

public class MainScript : MonoBehaviour
{
    [SerializeField] private List<Player> leftPlayers;
    [SerializeField] private List<Player> rightPlayers;
    private List<Player> _allPlayers = new();
    public int currentPlayer = 0;
    private void Awake()
    {
        foreach (var leftPlayer in leftPlayers)
        {
            _allPlayers.Add(leftPlayer);
        }
        foreach (var rightPlayer in rightPlayers)
        {
            _allPlayers.Add(rightPlayer);
        }
        foreach (var allPlayer in _allPlayers)
        {
            allPlayer.hp = allPlayer.data.hlt;
            allPlayer.text.text = allPlayer.hp.ToString();
        }
        _allPlayers = _allPlayers.OrderByDescending(player => player.data.@int).ToList();
    }

    void Start()
    {
        var aliveLeftPlayers = 0;
        var aliveRightPlayers = 0;

        foreach (var leftPlayer in leftPlayers)
        {
            if (!leftPlayer.isDead) aliveLeftPlayers++;
        }
        foreach (var rightPlayer in rightPlayers)
        {
            if (!rightPlayer.isDead) aliveRightPlayers++;
        }

        if (aliveLeftPlayers == 0)
        {
            Debug.LogWarning("MATCH END, right win!");
            return;
        }

        if (aliveRightPlayers == 0)
        {
            Debug.LogWarning("MATCH END, left win!");
            return;
        }

        StartCoroutine(StartCalculate());
    }

```

Рисунок 9 – Фрагмент кода скрипа основы структуры игрового раунда

```
IEnumerator StartCalculate()
{
    if (_allPlayers[currentPlayer].isDead)
    {
        currentPlayer++;
        StartCoroutine(StartCalculate());
        yield break;
    }
    _allPlayers[currentPlayer].render.color = Color.green;

    //yield return new WaitForSeconds(1);

    Player attackedPlayer;

    if (!IsPlayerFromLeftSide(_allPlayers[currentPlayer]))
    {
        attackedPlayer = CalculateEffectiveness(_allPlayers[currentPlayer], leftPlayers);
    }
    else
    {
        attackedPlayer = CalculateEffectiveness(_allPlayers[currentPlayer], rightPlayers);
    }
    attackedPlayer.render.color = Color.red;
    Debug.DrawLine(_allPlayers[currentPlayer].transform.position, attackedPlayer.transform.position, Color.yellow, 0.5f);
    AttackPlayer(_allPlayers[currentPlayer], attackedPlayer);
    yield return new WaitForSeconds(0.5f);
    _allPlayers[currentPlayer].render.color = Color.white;
    if (!attackedPlayer.isDead)
        attackedPlayer.render.color = Color.white;
    currentPlayer++;
    if (currentPlayer >= _allPlayers.Count - 1)
    {
        currentPlayer = 0;
    }
    Start();
}

struct Effectiveness
{
    public Player player;
    public float effectiveness;

    public Effectiveness(Player player, float effectiveness)
    {
        this.player = player;
        this.effectiveness = effectiveness;
    }
}
```

Рисунок 10 – Фрагмент скрипта, отвечающего за визуализацию

```
private float Formula1(PlayerData ua, PlayerData ub, bool isFirst)
{
    if (isFirst)
    {
        if (ub.arm - ua.ap1 >= 0)
        {
            return (Pow(ua.atc1, 2) / Pow(ua.atc1, 2) + Pow(ub.dfc, 2)) * ua.GetAverageDamage1() * (ub.arm - ua.ap1);
        }
        else
        {
            return (Pow(ua.atc1, 2) / Pow(ua.atc1, 2) + Pow(ub.dfc, 2)) * ua.GetAverageDamage1();
        }
    }
    else
    {
        if (ub.arm - ua.ap2 >= 0)
        {
            return (Pow(ua.atc2, 2) / Pow(ua.atc2, 2) + Pow(ub.dfc, 2)) * ua.GetAverageDamage2() * (ub.arm - ua.ap2);
        }
        else
        {
            return (Pow(ua.atc2, 2) / Pow(ua.atc2, 2) + Pow(ub.dfc, 2)) * ua.GetAverageDamage2();
        }
    }
}
private float Formula2(PlayerData ua, PlayerData ub, bool isFirst)
{
    var m1 = 10;
    var m2 = 5;
    if (isFirst)
    {
        if (ub.arm - ua.ap1 >= 0)
        {
            return (Pow(ua.atc1+m1,2)*0.5f)/Pow((ua.atc1+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage1()+m2)*(ub.arm-ua.ap1);
        }
        else
        {
            return (Pow(ua.atc1+m1,2)*0.5f)/Pow((ua.atc1+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage1()+m2);
        }
    }
    else
    {
        if (ub.arm - ua.ap2 >= 0)
        {
            return (Pow(ua.atc2+m1,2)*0.5f)/Pow((ua.atc2+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage2()+m2)*(ub.arm-ua.ap2);
        }
        else
        {
            return (Pow(ua.atc2+m1,2)*0.5f)/Pow((ua.atc2+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage2()+m2);
        }
    }
}
```

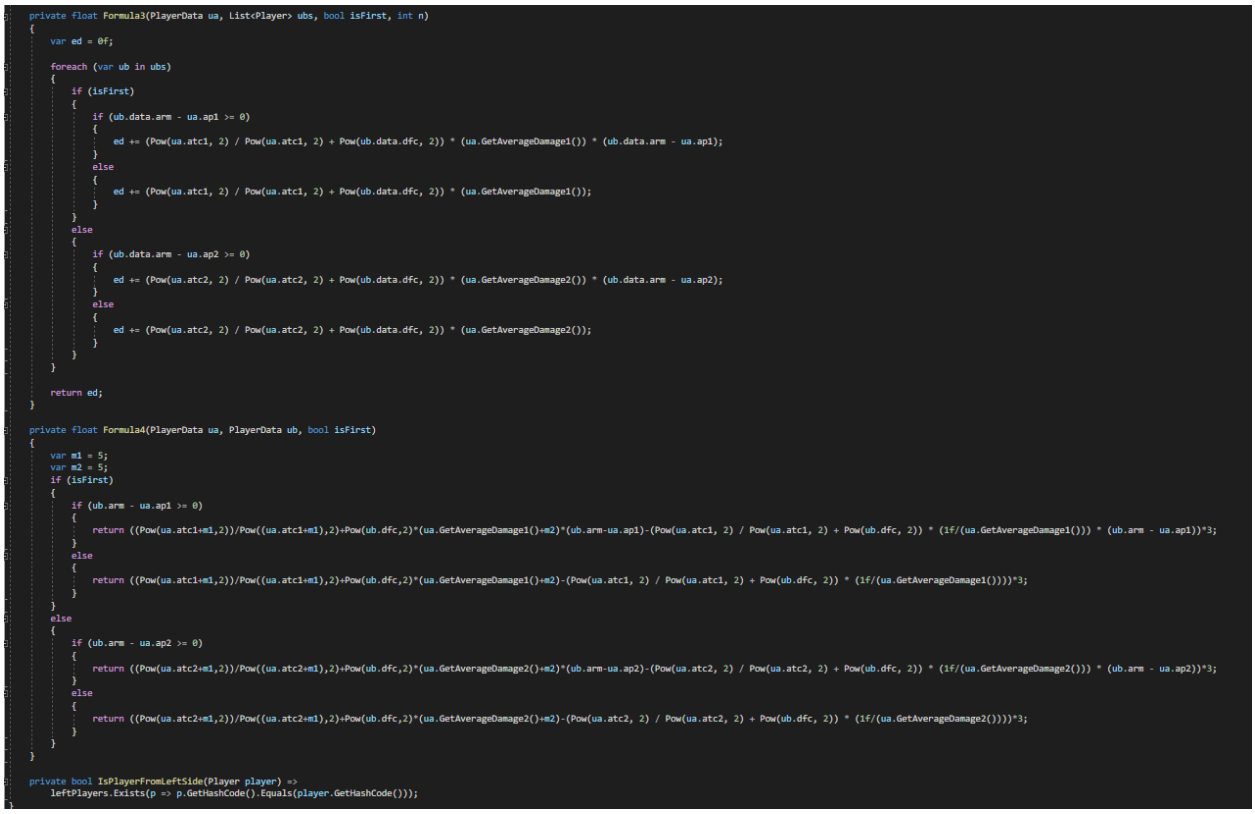
Рисунок 11 – Код скрипта, рассчитывающего ориентировочную эффективность юнитов

```
Player CalculateEffectiveness(Player u1, List<Player> u2s)
{
    List<Effectiveness> topEffectiveness = new();
    switch (u1.data.frm1)
    {
        case Formula.one:
            foreach (var u2 in u2s)
            {
                if (u2.isDead) continue;
                var ef = new Effectiveness(u2, Formula1(u1.data, u2.data, true));
                topEffectiveness.Add(ef);
                Debug.Log(ef.effectiveness);
            }
            break;
        case Formula.two:
            foreach (var u2 in u2s)
            {
                if (u2.isDead) continue;
                var ef = new Effectiveness(u2, Formula2(u1.data, u2.data, true));
                topEffectiveness.Add(ef);
                Debug.Log(ef.effectiveness);
            }
            break;
        case Formula.three:
            Debug.Log(Formula3(u1.data, u2s, true, 1));
            break;
        case Formula.four:
            foreach (var u2 in u2s)
            {
                if (u2.isDead) continue;
                var ef = new Effectiveness(u2, Formula4(u1.data, u2.data, true));
                topEffectiveness.Add(ef);
                Debug.Log(ef.effectiveness);
            }
            break;
        default:
            throw new ArgumentOutOfRangeException();
    }

    topEffectiveness.OrderByDescending(effectiveness => effectiveness.effectiveness);
    return topEffectiveness[0].player;
}

private void AttackPlayer(Player player1, Player player2)
{
    var dmg = player1.data.GetRandomDamage1();
    Debug.LogError(dmg);
    player2.hp -= dmg;
    if (player2.hp <= 0)
    {
        player2.isDead = true;
        player2.text.text = "D";
        player2.text.color = Color.white;
        player2.render.color = Color.black;
        return;
    }
    player2.text.text = player2.hp.ToString();
}
```

Рисунок 12 – Код скрипта, задающего состояния объектов и рассчитывающего принятие решений



```
private float Formula3(PlayerData ua, List<Player> ubs, bool isFirst, int n)
{
    var ed = 0f;

    foreach (var ub in ubs)
    {
        if (isFirst)
        {
            if (ub.data.arm - ua.ap1 >= 0)
            {
                ed += (Pow(ua.atc1, 2) / Pow(ua.atc1, 2) + Pow(ub.data.dfc, 2)) * (ua.GetAverageDamage1()) * (ub.data.arm - ua.ap1);
            }
            else
            {
                ed += (Pow(ua.atc1, 2) / Pow(ua.atc1, 2) + Pow(ub.data.dfc, 2)) * (ua.GetAverageDamage1());
            }
        }
        else
        {
            if (ub.data.arm - ua.ap2 >= 0)
            {
                ed += (Pow(ua.atc2, 2) / Pow(ua.atc2, 2) + Pow(ub.data.dfc, 2)) * (ua.GetAverageDamage2()) * (ub.data.arm - ua.ap2);
            }
            else
            {
                ed += (Pow(ua.atc2, 2) / Pow(ua.atc2, 2) + Pow(ub.data.dfc, 2)) * (ua.GetAverageDamage2());
            }
        }
    }

    return ed;
}

private float Formula4(PlayerData ua, PlayerData ub, bool isFirst)
{
    var m1 = 5;
    var m2 = 5;
    if (isFirst)
    {
        if (ub.arm - ua.ap1 >= 0)
        {
            return ((Pow(ua.atc1+m1,2))/Pow((ua.atc1+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage1()+m2)*(ub.arm-ua.ap1)-(Pow(ua.atc1, 2) / Pow(ua.atc1, 2) + Pow(ub.dfc, 2)) * (1f/(ua.GetAverageDamage1())) * (ub.arm - ua.ap1))*3;
        }
        else
        {
            return ((Pow(ua.atc1+m1,2))/Pow((ua.atc1+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage1()+m2)-(Pow(ua.atc1, 2) / Pow(ua.atc1, 2) + Pow(ub.dfc, 2)) * (1f/(ua.GetAverageDamage1())))*3;
        }
    }
    else
    {
        if (ub.arm - ua.ap2 >= 0)
        {
            return ((Pow(ua.atc2+m1,2))/Pow((ua.atc2+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage2()+m2)*(ub.arm-ua.ap2)-(Pow(ua.atc2, 2) / Pow(ua.atc2, 2) + Pow(ub.dfc, 2)) * (1f/(ua.GetAverageDamage2())) * (ub.arm - ua.ap2))*3;
        }
        else
        {
            return ((Pow(ua.atc2+m1,2))/Pow((ua.atc2+m1),2)+Pow(ub.dfc,2)*(ua.GetAverageDamage2()+m2)-(Pow(ua.atc2, 2) / Pow(ua.atc2, 2) + Pow(ub.dfc, 2)) * (1f/(ua.GetAverageDamage2())))*3;
        }
    }
}

private bool IsPlayerFromLeftSide(Player player) =>
    leftPlayers.Exists(p => p.GetHashCode().Equals(player.GetHashCode()));
```

Рисунок 13 - Фрагмент кода скрипа основы структуры игрового раунда

Выводы по второй главе

В данной главе была выработана математическая основа для реализации прототипа интеллектуальных агентов для мобильной игры на платформе Unity, на основе которой была разработана игровая сцена. Разработанная программная составляющая является основой для дальнейшего анализа и развития рассматриваемых интеллектуальных агентов.

Глава 3 Тестирование и анализ итогового продукта

3.1 Тестирование программы

В рамках данной работы проводилось тестирование разработанных интеллектуальных агентов для мобильной игры на платформе Unity. Основная цель тестирования заключалась в проверке функциональности программы, выявлении возможных ошибок и недоработок [1].

Планирование процесса тестирования

Первым этапом планирования процесса тестирования было определение целей и задач тестирования. На основании этого был составлен тест-план, включающий в себя список тестовых сценариев и тест-кейсов, необходимых для проведения тестирования. В качестве основы для составления тест-плана использовались требования, предоставленные руководителями от предприятия.

Сценарий 1. Проверка работоспособности прототипа

Цель: проверка работоспособности прототипа игры

Шаги:

-запустить прототип игры,

-проверить, что игра запускается без ошибок,

-проверить, что игровая сцена отображается корректно,

-проверить, что игровой процесс работает без ошибок,

Ожидаемый результат: прототип игры запускается без ошибок, игровой процесс работает корректно.

Сценарий 2. Проверка работы интеллектуальных агентов

Цель: проверка работы интеллектуальных агентов

Шаги:

- запустить прототип игры,

- проверить, что агенты корректно выполняют свои функции,

- проверить, что агенты принимают корректные решения,

- проверить, что агенты реагируют на изменения в игровой ситуации,

Ожидаемый результат: интеллектуальные агенты работают корректно и выполняют свои функции.

Проектирование тестов

На этапе проектирования тестов были использованы различные техники тестирования, такие как тестирование граничных значений, тестирование внутренней логики игры, тестирование управления и реакции игры на действия пользователя. Было определено, что тестирование будет проводиться вручную, так как задача была проверить игру на реальных устройствах и обнаружить возможные ошибки, которые не могут быть выявлены автоматизированными тестами.

Подготовка тестового окружения

На этапе подготовки тестового окружения были созданы условия для проведения тестирования. В условиях создания мобильного приложения также происходит выбор различных устройств на которых будут проводиться тестирование, включая смартфоны и планшеты, с разными версиями операционных систем iOS и Android, создаются данные для тестирования, такие как учетные записи пользователей, и подготовлены инструменты для автоматизации тестирования. Однако в рамках разработки прототипа интеллектуальных агентов данную часть можно сократить до тестирования на одном устройстве. Инструментом тестирования выступал Unity Test Runner.

Unity Test Runner - это инструмент, используемый для автоматического тестирования проектов в Unity. Он предоставляет возможность тестировать игры на различных устройствах, используя различные сценарии и наборы тестовых данных. В Unity Test Runner можно создать различные тесты, такие как модульные тесты, интеграционные тесты и тесты на производительность. Тесты в Unity Test Runner могут быть запущены как локально, так и на удаленных устройствах.

Выполнение тестов

На этапе выполнения тестов было проведено тестирование вручную, чтобы убедиться в корректной работе игры и обнаружить возможные ошибки, не выявленные автоматически.

3.2 Анализ результатов тестирования

Ключевым элементом анализа работы интеллектуальных агентов является проверка соответствия требованиям.

Адекватность поведения.

Игроки ожидают, что противники и союзники будут вести себя соответствующим образом в соответствии с их типом и имеющимися действиями в игре. Например, при использовании атакующего действия агент должен нападать на более слабых юнитов, когда используется действие, повышающее параметры – необходимо оказывать поддержку более эффективным союзным юнитам.

Данное требование было выполнено полностью.

Правильное принятие решений.

Интеллектуальный агент должен уметь принимать правильные решения в соответствии с текущей ситуацией на поле боя. Например, если у противника осталось мало здоровья, то агент должен выбрать наиболее эффективную атаку для его устранения.

Проведя сравнительный анализ данного требования, был сделан следующий вывод: несмотря на правильность тактики, с позиции нанесения наибольшего урона противнику, данный подход не является оптимальным, так как не учитывается различный атакующий потенциал юнитов и их актуальное состояние. Так, выбор более защищённого противника может быть эффективнее, если его влияние на состязание существенно больше, чем у альтернатив [8]. Для расчёта данного фактора нужно добавить

модификатор отношения ориентировочной эффективности цели к её показателю здоровья в данный момент. Таким образом, на этапе определении цели, нужно ориентироваться не на параметр ED, а на значение ED/Hlt.

Стабильность работы.

Интеллектуальные агенты должны работать стабильно и безошибочно, чтобы не нарушать работу приложения и не вызывать ошибок. Они должны быть протестированы на различных устройствах и в различных условиях, чтобы убедиться в их стабильности.

По результатам тестирований программной составляющей разрабатываемого прототипа, были обнаружены и исправлены различные ошибки, нарушавшие как работу самого приложения, так и внутреннюю логику действий интеллектуальных агентов.

Выводы по третьей главе

В данной главе было проведено тестирование разработанных интеллектуальных агентов для мобильной игры на платформе Unity. В результате тестирования были выявлены некоторые ошибки и недочеты в работе агентов, которые были исправлены.

Также был проведен анализ итогового продукта, который показал высокое качество разработанных агентов и их эффективность в игровом процессе.

Тестирование и анализ итогового продукта позволили убедиться в правильности выбранного подхода к разработке интеллектуальных агентов и подтвердили их функциональность и удобство использования в игровом процессе.

Заключение

В рамках данной работы были разработаны интеллектуальные агенты для мобильной игры на платформе Unity. Для создания этих агентов были рассмотрены основные понятия и методы, применяемые в игровой индустрии для разработки интеллектуальных агентов. Были проведены анализ и сравнение этих методов, что помогло определить наиболее эффективные из них и использовать их для создания интеллектуальных агентов в данной работе.

В ходе работы были реализованы интеллектуальные агенты, способные принимать решения на основе имеющейся информации и взаимодействовать друг с другом. Разработанные математические формулы позволили сделать принимаемые решения эффективными, позволяющими создать у пользователя впечатление разумности. Устройство этих формул позволяет, как менять отдельные значения характеристик в угоду игровому балансу, так и добавлять новые структурные элементы, создавая новые игровые механики.

Был проведен анализ эффективности агентов в игровой среде и определены преимущества и недостатки их использования. На основе проведённого анализа были сделаны корректировки и дополнения.

Результаты исследования показали, что использование интеллектуальных агентов в мобильных играх на платформе Unity может значительно улучшить игровой процесс и повысить уровень интерактивности для пользователя.

В дальнейшем планируется улучшить работу созданного прототипа и расширить функционал игры, чтобы еще больше улучшить игровой процесс и увеличить количество довольных пользователей. Благодаря использованию интеллектуальных агентов, пользователи получат больше возможностей для взаимодействия с игрой, что сделает геймплей более увлекательным и захватывающим.

Список используемой литературы

1. Библиотека алгоритмов искусственного интеллекта для игр. Часть 1. (исходники) // Интерфейс URL: https://www.interface.ru/home.asp?artId=1321 (дата обращения: 02.03.2023).

2. Джон М. Unity для разработчика. Мобильные мультиплатформенные игры/ Джон М., Пэрис Б-Э. - 1-е изд. - СПБ: Питер, 2018. - 352 с.

3. Как создать игровой ИИ: гайд для начинающих // Хабр URL: https://habr.com/ru/companies/pixonic/articles/428892/ (дата обращения: 02.03.2023).

4. Обзор техник реализации игрового ИИ // Хабр URL: https://habr.com/ru/articles/420219/ (дата обращения: 02.03.2023).

5. Практический геймдизайн: создание ИИ для своей игры // DTF URL: https://dtf.ru/gamedev/41108-prakticheskiy-geymdizayn-sozdanie-ii-dlya-svoey-igry (дата обращения: 02.03.2023).

6. Разработка искусственного интеллекта из искусственного идиота в пошаговой тактической игре // DTF URL: https://dtf.ru/gamedev/91889-razrabotka-iskusstvennogo-intellekta-iz-iskusstvennogo-idiota-v-poshagovoy-takticheskoy-igre (дата обращения: 02.03.2023).

7. Создание искусственного интеллекта для игр — от проектирования до оптимизации // SavePearlHarbor URL: https://savepearlharbor.com/?p=264682 (дата обращения: 02.03.2023).

8. Шампандар А.Д. Искусственный интеллект в компьютерных играх: как обучить виртуальные персонажи реагировать на внешние воздействия. - 1-е изд. - М: Вильямс, 2007. - 768 с.

9. 7 методов тестирования игр // Хабр URL: https://habr.com/ru/companies/otus/articles/557832/ (дата обращения: 02.03.2023).

10. AI: Практическое руководство по построению полного игрового ИИ: Часть I // Библиотека программиста URL: https://masandilov.ru/ai/practical_guide_to_building_a_complete_game_ai_vol_i (дата обращения: 02.03.2023).

11. AI: Практическое руководство по построению полного игрового ИИ: Часть II // Библиотека программиста URL: https://masandilov.ru/ai/practical_guide_to_building_a_complete_game_ai_vol_ii (дата обращения: 02.03.2023).

12. Bourg D.M., Seemann G. AI for Game Developers: Creating Intelligent Behavior in Games. - 1-е изд. - O'Reilly Media, 2004. - 392 с.

13. Buckland M. Programming Game AI by Example. - 1-е изд. - Jones & Bartlett Learning, 2004. - 495 с.

14. Designing AI Algorithms For Turn-Based Strategy Games // Game Developer URL: https://www.gamedeveloper.com/design/designing-ai-algorithms-for-turn-based-strategy-games#close-modal (дата обращения: 02.03.2023).

15. Gibson J.B. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. - 1-е изд. - 2014. - 1614 с.

16. Hex Map 1 Creating a Hexagonal Grid // Catlike Coding URL: https://catlikecoding.com/unity/tutorials/hex-map/part-1/ (дата обращения: 02.03.2023).

17. Hocking J. Unity in Action, Second Edition Multiplatform game development in C#. - 2-е изд. - Shelter Island: Manning, 2018. - 402 с.

18. Millington I. AI for Games AI for Everything. - 1-е изд. - 2022. - 76 с.

19. Penny B. Mathematics for Game Programming and Computer Graphics: Explore the essential mathematics for creating, rendering, and manipulating 3D virtual environments. - 1-е изд. - Birmingham: Packt Publishing, 2022. - 445 с.

20. Russel S, Artificial Intelligence: A Modern Approach/ Russel S., Norvig P. - 3-е изд. - Boston: 2009. - 1152 с.

21. Sadler M. Game Changer: AlphaZero's Groundbreaking Chess Strategies and the Promise of AI/ Sadler M., Regan N.. - 1-е изд. - New in Chess, 2019. - 937 с.