Аннотация

Abstract

Оглавление

[Введение 4](#_Toc41305764)

[1 Исследование существующих подходов к моделированию поведения интеллектуальных агентов и методов машинного обучения 5](#_Toc41305765)

[1.1 Исследование существующих подходов к моделированию поведения интеллектуальных агентов 5](#_Toc41305766)

[1.1.1 Модель поведения на основе правил 5](#_Toc41305767)

[1.1.2 Модель поведения на основе конечных автоматов 6](#_Toc41305768)

[1.1.3 Модель поведения на базе деревьев поведения 8](#_Toc41305769)

[1.2 Исследование существующих методов машинного обучения 12](#_Toc41305770)

[1.2.1 Обучение с подкреплением 12](#_Toc41305771)

[1.2.2 Глубинное обучение 15](#_Toc41305772)

[1.3 Исследование применимости методов машинного обучения к моделированию поведения интеллектуальных агентов 18](#_Toc41305773)

[1.4 Выбор алгоритмов для реализации 19](#_Toc41305774)

[1.5 Выбор средств реализации 19](#_Toc41305775)

[2 Проектирование проекта проектного проектирования 26](#_Toc41305776)

[2.2 Проектирование тестовой платформы 26](#_Toc41305777)

[2.3 Проектирование модели поведения интеллектуальных агентов 28](#_Toc41305778)

[2.4 Проектирование модели обучения 29](#_Toc41305779)

[2.5 Проектирование графического интерфейса пользователя 29](#_Toc41305780)

[3 Описание программной реализации 35](#_Toc41305781)

[3.1 Разработка тестовой платформы 35](#_Toc41305782)

[3.2 Разработка модели поведения 43](#_Toc41305783)

[3.3 Применение машинного обучения 47](#_Toc41305784)

[3.4 Разработка графического интерфейса пользователя 47](#_Toc41305785)

[3.5 Результаты разработки 49](#_Toc41305786)

[4 Тестирование 51](#_Toc41305787)

[4.1 Описание видов тестирования 51](#_Toc41305788)

[4.1.1 Функциональное тестирование 51](#_Toc41305789)

[4.1.2 Юзабилити-тестирование 51](#_Toc41305790)

[4.1.3 Тестирование стабильности 52](#_Toc41305791)

[4.1.4 Модульное тестирование (юнит-тестирование) 52](#_Toc41305792)

[4.1.5 Тестирование на отказ и восстановление 53](#_Toc41305793)

[4.1.6 Прочие виды тестирования 55](#_Toc41305794)

[4.2 Тестирование программы 55](#_Toc41305795)

[5 Проведение вычислительного эксперимента 56](#_Toc41305796)

[5.1 Постановка задачи 56](#_Toc41305797)

[5.2 Проведение эксперимента 57](#_Toc41305798)

[5.3 Результаты эксперимента 57](#_Toc41305799)

[Заключение 58](#_Toc41305800)

[Список использованной литературы 59](#_Toc41305801)

# Введение

# 1 Исследование существующих подходов к моделированию поведения интеллектуальных агентов и методов машинного обучения

# 1.1 Исследование существующих подходов к моделированию поведения интеллектуальных агентов

# 1.1.1 Модель поведения на основе правил

При такой реализации поведение агента регулируется заранее заданным набором правил, то есть алгоритмов, что ему нужно делать в каждой конкретной ситуации. Такая система стоит дальше всего от настоящего искусственного интеллекта.

Классической игрой, в которой используется такой подход к организации игрового искусственного интеллекта, является Pac-Man. Игрока преследуют четыре привидения. Каждое привидение действует, подчиняясь простому набору правил. Одно привидение всегда поворачивает влево, другое всегда поворачивает вправо, третье поворачивает в произвольном направлении, а четвертое всегда поворачивает в сторону игрока. Если бы на экране привидения появлялись по одному, их поведение было бы очень легко определить, и игрок смог бы без труда от них спасаться. Но поскольку появляется сразу группа из четырех привидений, их движения кажутся сложным и скоординированным выслеживанием игрока. На самом же деле только последнее из четырех привидений учитывает расположение игрока. На рисунке 1 наглядно изображено, каким образом каждое из привидений принимает решения о дальнейшем направлении движения: цветными облачками с номерами изображены привидения, а стрелками – принимаемые ими решения.

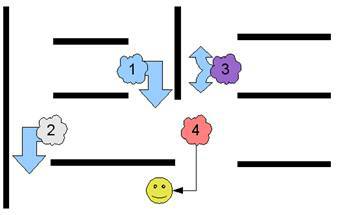


Рисунок 1 – Принятие решений привидениями в Pac-Man

Из этого примера следует, что правила не всегда должны быть жёстко заданы. Правила могут быть заданы также на основе текущего состояния агента. При жёстком задании агент во всех ситуациях одного рода, то есть ситуациях, для которых предусмотрено одно и то же правило, будет вести себя одинаково. При задании правил на основе состояния агента допускается вариативность его поведения: в одинаковых ситуациях агент может повести себя по-разному, например, в зависимости от положения игрока в игровом мире.

Кроме того, правила могут задаваться на основе параметров агентов, таких как уровень агрессии, уровень смелости, дальность обзора, скорость мышления и так далее. Такие параметры позволяют получить более разнообразное поведение объектов даже при использовании систем на основе правил [1].

Системы, использующие такой подход к организации ИИ, далеки от реалистичных, поскольку их агенты всегда следуют только определенным правилам и в случае, если происходит ситуация, не предусмотренная правилами, ИИ будет вести себя непредсказуемо и не удовлетворительно ситуации.

# 1.1.2 Модель поведения на основе конечных автоматов

Конечный автомат (машина с конечным числом состояний) является способом моделирования и реализации объекта, обладающего различными состояниями в течение своей жизни. Каждое «состояние» может представлять физические условия, в которых находится объект, или, например, набор эмоций, выражаемых объектом. Здесь эмоциональные состояния не имеют никакого отношения к эмоциям ИИ, они относятся к заранее заданным поведенческим моделям, вписывающимся в контекст игры. На рисунке 2 изображена схема типичного конечного автомата. На этой иллюстрации кругами изображены возможные состояния агента, а стрелками – переходы из состояния в состояние.



Рисунок 2 – Схема состояний конечного автомата

Одна из наиболее известных реализаций игрового искусственного интеллекта на основе конечного автомата носит название Goal Oriented Action Planning (GOAP) [2]. Такая система впервые появилась в 2005 году и была разработана и использована студией Monolith Productions в игре F.E.A.R. Позже вариации на тему этого подхода были применены в других успешных коммерческих проектах, в том числе S.T.A.L.K.E.R. и Fallout 3.

При такой организации ИИ последовательность действий персонажа определяется не только целью, которую он преследует, но также и текущим состоянием мира и агента. Это означает, что одна и та же цель может быть достигнута несколькими различными методами в зависимости от выполнения определённых условий, что делает ИИ более динамичным и реалистичным.

Для реализации такого подхода необходимо разработать отдельный модуль, называемый планировщиком. Планировщик на основе данных о текущем состоянии агента, таких как его сытость или состояние здоровья, определяет, что агент должен делать. Также планировщик выстраивает последовательность действий, которые нужно совершить агенту, чтобы достигнуть цели. Здесь снова используются данные о состоянии агента и игрового мира – это нужно, чтобы определить, выполняются ли определённые условия, позволяющие агенту достигнуть цели тем или иным способом. Затем планировщик определяет наиболее оптимальный путь для достижения цели в зависимости от «стоимости» действий. В качестве стоимости действия можно принимать, например, количество очков запаса сил, расходуемое на выполнения этого действия, или же время, которое нужно затратить. Таким образом, получается, что агент выбирает последовательность аналогично человеку, руководствуясь тем, как будет проще достичь желаемого результата. Фактически, планировщик не является частью алгоритма GOAP, но тесно связан с ним.

Алгоритмы GOAP значительно упрощают создание ИИ на базе конечных автоматов. Действия, связанные друг с другом в конечном автомате, оказываются не связанными при применении GOAP, что делает код более модульным и простым в обслуживании, а кроме того, в множество действий агента становится гораздо проще добавлять новые действия.

# 1.1.3 Модель поведения на базе деревьев поведения

Деревьями поведения называют математические модели выполнения плана, используемые в информатике, робототехнике, системах управления и видеоиграх, которые описывают переключения в конечном наборе задач по модульному принципу.

В сущности, дерево поведения представляет собой частный случай конечного автомата.

Деревья поведения происходят от индустрии компьютерных игр и являются мощным инструментом моделирования поведения неигровых персонажей (NPC). Такой подход к моделированию поведения NPC был использован в таких играх, как Halo, Bioshock и Spore. Впоследствии деревья поведения стали применяться для решения и других классов задач, таких как управление БПЛА и роботами, роботизированные манипуляции и прочие задачи, однако в данном случае наибольший интерес представляет именно сфера, из которой произошли деревья поведения, а именно – моделирование поведения неигровых персонажей.

Графически дерево поведения представляется как направленное дерево, в котором узлы разделяются на корневые, узлы потока управления или узлы выполнения (задачи). Для каждой пары соединённых узлов исходящий узел называется родительским, а входящий – дочерним. Корневой узел не имеет никаких родителей и имеет ровно один дочерний узел, узлы потока управления имеют одного родителя и хотя бы одного потомка, а узлы выполнения имеют одного родителя и не имеют потомков. Графически дочерние элементы узла потока управления располагаются под ним, упорядоченные слева направо.

Выполнение дерева поведения начинается с корневого узла, который посылает своему потомку сигналы с определённой частотой. Эти сигналы разрешают выполнение дочернего узла. Когда выполнение узла в дереве разрешено, он возвращает родительскому узлу статус выполнения, если ещё выполнение ещё не закончилось, успех, если выполнение узла позволило достигнуть цели, или же провал – в противном случае.

Узел управления потоком нужен для управления подзадачами, из которых он состоит. Узел потока управления может быть либо селекторным (резервным) узлом, либо узлом последовательности. Они выполняют каждую из своих подзадач по очереди. Когда подзадача завершена и возвращает свой статус (успех или неудача), узел потока управления решает, выполнять следующую подзадачу или нет [3].

Деревья поведения могут содержать в себе множество узлов и быть очень глубокими. Так, на рисунке 3 представлен пример дерева поведения, описывающего вход в здание.

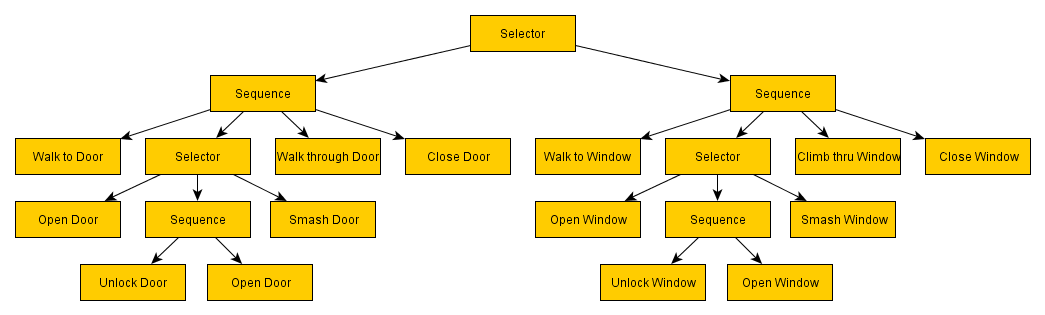


Рисунок 3 – Дерево поведения.

На рисунке видно, как происходит выбор подходящей модели поведения в зависимости от различных внешних факторов. Так, в здание можно войти через дверь или через окно. Чтобы войти через дверь, нужно подойти к ней, затем решить, что делать с самой дверью: просто открыть, разблокировать замок и открыть, или же выбить; затем пройти через саму дверь. Чтобы войти через окно, необходимо подойти к нему, аналогичным образом открыть либо разбить окно, залезть в здание и закрыть окно за собой [4].

Деревья поведения являют собой мощный инструмент для моделирования поведения интеллектуальных агентов.

Реализация деревьев поведения предполагает разработку набора классов, узлов, из которых может быть построено само дерево. Пример реализации дерева поведения приведён в источнике [5]. Будет полезным рассмотреть этот пример подробнее.

Построение дерева поведения начинается с определения базового класса узла – Node. Этот класс объявляется абстрактным, поскольку впоследствии основные его черты будут унаследованы классами-потомками. В классе объявляются несколько методов, в том числе делегат NodeReturn, возвращающий состояние узла. Узел может быть в состояниях Running, Success и Failure. На базе этого класса будут реализованы все типы узлов дерева.

Следующим этапом построения дерева становится расширение существующего функционала узлов до возможностей селекторов. Селектор в дереве поведения – это узел, который имеет один или более дочерних узлов, выполняемых на выбор: если успех действия достигнут при исполнении дочернего узла №1, узел №2 не выполняется. Соответственно, в классе Selector, унаследованном от Node, должна содержаться информация о дочерних узлах в контейнере (в примере эта информация хранится в списке). Кроме того, селектор должен уметь оценивать эффективность применения каждого из дочерних узлов: для этого классу Selector необходим метод Evaluate(), который оценивает каждый узел из списка дочерних.

Следующий шаг в построении дерева – доработка функционала узлов до последовательностей. От класса Node наследуется класс Sequence. Узел Sequence в дереве служит для последовательного исполнения каждого из дочерних узлов. В случае, если какой-то из дочерних узлов вернул состояние Failure, последовательность обрывается, и узел Sequence также возвращает состояние Failure. Аналогичным предыдущему случаю образом, классу Sequence необходим список дочерних узлов. Также классу Sequence необходим механизм, который будет отслеживать текущие состояния дочерних узлов. Этот механизм опрашивает каждый из узлов из списка и проверяет, в каком он состоянии. Если узел перешёл в состояние Failure, вся последовательность также переходит в это состояние. Если узел вернул состояние Success, выполнение последовательности продолжается. Если узел возвращает состояние Running, опрос прекращается, и для всей последовательности также устанавливается состояние Running.

Ещё один необходимый тип узла, также наследуемый от Node – узел Inverter. Задача этого узла – возвращать Failure, когда дочерний узел возвращает Success, и наоборот. В случае, если дочерний узел (единственный) возвращает Running, инвертер также вернёт Running.

Наконец, необходимо реализовать класс ActionNode, экземпляры которого будут листьями дерева, то есть конечными действиями. В теле этого класса, также унаследованного от Node, описывается конкретное конечное действие в виде делегата action. Также классу нужен механизм оценки, который возвращал бы состояние узла.

# 1.2 Исследование существующих методов машинного обучения

Машинное обучение как направление в компьютерной науке берёт своё начало в 50-х годах XX века. Это обширный подраздел искусственного интеллекта, изучающий методы построения алгоритмов, способных обучаться. Выделяют несколько разновидностей методов машинного обучения [5].

Машинное обучение может быть применено к моделированию поведения агентов с целью создания более результативной модели поведения.

# 1.2.1 Обучение с подкреплением

Обучение с подкреплением (reinforcement learning, RL) является одной из разновидностей методов машинного обучения, где испытуемая система (агент) обучается при взаимодействии со средой. Такое обучение является частным случаем обучения с учителем, но в данном случае в роли учителя выступает сама среда. Среда реагирует на воздействия агента посредством откликов, называемых сигналами подкрепления [6].

Агент и среда взаимодействуют друг с другом, то есть воздействуют друг на друга. Алгоритм взаимодействия агента и среды в общем виде можно разделить на следующие шаги:

- агент получает текущее состояние системы S0 (например, информацию о положении объектов вокруг него);

- агент осуществляет действие A0 (например, движение в сторону одного из объектов);

- среда переходит в некоторое новое состояние S1 (положение агента изменилось относительно объектов в среде);

- среда посылает агенту отклик R1 (некоторое вознаграждение).

Цель заключается в максимизации ожидаемого вознаграждения. Эта цель приводит агента к необходимости совершения наилучшего хода или действия.

Различают три подхода к обучению с подкреплением.

Первый подход основывается на значениях. Цель такого подхода – оптимизация функции V(s), то есть стоимостной функции, которая определяет максимально возможное вознаграждение, которое может получить система. Значение каждой позиции – это общая сумма вознаграждения, которое система сможет накопить в будущем, начиная с этой позиции.

Следующий подход основан на политике. При применении такого подхода задача состоит в оптимизации функции политики Pi(s) без использования функции значения.

Политика – это то, что определяет поведение системы в конкретный момент времени. Существует два типа политики: стохастический и детерминированный.

Детерминированный тип политики будет возвращать одно и то же действие.

Стохастический тип выводит вероятность распределения по действиям.

Политика прямо указывает на лучшие действия для каждого шага.

Третий подход основывается на модели. При применении такого подхода моделируется среда, то есть создаётся модель её поведения. Зачастую для одной среды необходимо составлять множество различных моделей, чтобы наиболее полно отразить её поведение [7].

Обучение с подкреплением часто используется для создания игрового искусственного интеллекта. Подтверждений этому можно найти множество: [8], [9].

Отдельным подвидом обучения с подкреплением является Q-обучение. Этот метод обучения применяется при агентном подходе, что и необходимо для решения поставленной задачи.

Суть обучения заключается в следующем.

Агент, получая от среды вознаграждение, формирует функцию полезности Q. В дальнейшем это даёт ему возможность учитывать опыт предыдущего взаимодействия со средой и выбирать стратегию поведения соответствующим образом. В числе преимуществ данного подхода к обучению то, что агент может не формировать модель окружающей среды для того, чтобы сравнить ожидаемую полезность доступных действий.

Алгоритм обучения представлен в виде следующих шагов.

Шаг первый. Инициализация. Функция полезности определённого действия в определённой ситуации задаётся случайным образом.

Шаг второй. Наблюдение. Предыдущие состояния и действия запоминаются, после чего агент посредством сенсоров определяет текущее новое состояние и получает от среды вознаграждение за свои действия.

Шаг третий. Обновление. Агент переопределяет значение полезности Q по формуле, в которую входит предыдущее значение Q, вознаграждение за предыдущее действие r, фактор обучения LF и фактор дисконтирования DF. Фактор обучения определяет, насколько агент доверяет полученной новой информации, а фактор дисконтирования определяет, насколько агент задумывается о выгоде будущих действий.

Шаг четвёртый. Решение. Агент на основе вновь рассчитанной полезности определяет, какое действие в сложившейся ситуации будет наиболее полезным и выполняет его.

Цикл повторяется со второго шага [10].

Q-обучение достаточно часто используется в компьютерных играх.

# 1.2.2 Глубинное обучение

Существуют различные подходы к применению методов машинного обучения в целом и глубокого обучения в частности к играм. Первое, на что стоит обратить внимание – с каким классом игр приходится работать.

Игры с совершенной информацией (с полной информацией) предоставляют игроку возможность получать всю информацию об игре в каждый момент времени. К таким играм относятся классические настольные игры, такие как шашки, шахматы или нарды: в любой момент времени оба игрока видят расположение всех фигур на игровом поле и могут выстраивать стратегию собственного поведения на основе этой информации. Помимо расположения фигур и их наличия, игроки знают, какие ходы до этого совершал оппонент. Это можно экстраполировать на компьютерные игры: в таком случае в качестве игры с полной информацией можно было бы представить стратегию в реальном времени, в которой отсутствует так называемый «туман войны», который обычно закрывает часть карты, которую игрок не исследует своими юнитами в настоящий момент. В таком случае каждый из игроков мог бы наблюдать за действиями оппонента (оппонентов) и знать, доступ к каким ресурсам у него имеется, сколько их есть в запасе, какие здания и юниты могут быть построены в каждый момент времени.

Игры с несовершенной (неполной) информацией, наоборот, характеризуются тем, что игрок не может в любой момент времени знать, чем занимается оппонент. Это создаёт ему дополнительные трудности, связанные с необходимостью предугадывать действия оппонента и тратить больше сил на выстраивание стратегии.

Большинство современных компьютерных игр являются играми с неполной информацией, где существует необходимость предугадывать появление противников и их действия. Соответственно тому, как живому игроку приходится менять свою тактику игры в зависимости от внезапных появлений сил противника на горизонте, интеллектуальному агенту также будет необходимо менять тактику своего поведения. Для этого подходит глубокое обучение.

Глубинное обучение (также глубокое обучение, Deep Learning) – это направление в области Искусственного Интеллекта (Artificial Intelligence) и Машинного Обучения (Machine Learning), основанное на поиске таких моделей и алгоритмов, благодаря которым компьютеры смогут учиться на собственном опыте, формируя в процессе обучения многоуровневые, иерархические представления об окружающем мире, в которых понятия более высокого уровня определяются на основе понятий более низкого уровня. На данный момент основными "глубокими" моделями являются Глубокие Нейронные Сети (Deep Neural Networks) [11].

Глубинное обучение начало зарождаться ещё в 1980-х, однако значимых результатов получено так и не было. Это было обусловлено низкими вычислительными мощностями ЭВМ тех лет. Интерес к методам глубинного обучения вернулся к середине 2000-х годов, и на данный момент такой подход к построению искусственного интеллекта является одним из передовых, а современные вычислительные машины способны обеспечить необходимую вычислительную мощность.

В настоящее время глубинное обучение используется практически повсеместно. Наиболее часто методы глубокого обучения используются в области обработки изображений: сюда входят задачи распознавания лиц и эмоций [12], задачи цветокоррекции и колоризации [13] и т. д. Помимо работы с изображениями, глубокое обучение может быть использовано и в других областях: например, обработка звуковых сигналов: приложение Magenta [14] умеет создавать музыку, а сервис Google Voice [15] умеет транскрибировать голосовую почту и управлять СМС [16].

Глубинное обучение также может быть использовано в области разработки видеоигр, в частности, в области построения игрового искусственного интеллекта, то есть моделирования поведения интеллектуальных агентов. В материале [17] предлагается подход к реализации поведенческой модели агента на основе глубокого обучения с подкреплением, однако также отмечается, что в настоящий момент в сфере разработки игр машинное обучение в целом и глубинное (и глубинное с подкреплением) обучение в частности используются достаточно мало.

Одни из наиболее значимых успехов в области применения глубокого обучения в области игрового искусственного интеллекта принадлежат компании DeepMind [18]. Эта компания получила наибольшую известность благодаря разработке системы AlphaGo – искусственного игрока в го, который оказался способен обыграть действующего чемпиона мира по этой игре в марте 2016 года [19]. Эта компания занимается созданием и развитием игрового искусственного интеллекта, который способен играть в классические игры 70-х и 80-х. Почти в половине игр, в которые играл этот ИИ (22 игр из 49), ему удалось показать результаты, превосходящие результаты лучших игроков в эти игры.

В основе подхода, используемого этим ИИ, лежит глубинное обучение с подкреплением, также называемое deep Q-network или DQN [20]. DQN представляет собой вариацию обучения с подкреплением без модели с применением Q-обучения.

В случае ИИ, разрабатываемого DeepMind, особенность подхода состоит в том, что функция полезности моделируется с использованием глубинной нейронной сети. В качестве архитектуры нейронной сети была выбрана свёрточная нейронная сеть. В планах команды развить свой ИИ, чтобы он был способен играть в более сложные игры, такие как Doom и гоночные симуляторы [21].

В настоящий момент уже существуют способы научить интеллектуального агента игре в относительно сложные игры. Так, глубинное Q-обучение используется в системе, представленной в материале [22] для игры в Doom.

# 1.3 Исследование применимости методов машинного обучения к моделированию поведения интеллектуальных агентов

Для того, чтобы принять решение о том, какие методы и модели стоит применить к решению задачи, было необходимо провести исследование. Исследование состояло в изучении научных работ, опубликованных за последние несколько лет, с целью установления, какие подходы к применению машинного обучения при моделировании поведения интеллектуальных агентов изучаются исследователями наиболее активно и подробно.

Исследование включало в себя поиск и анализ научных статей, опубликованных в период с 2014 по 2019 год в научных базах, таких как Scopus, Springer и прочие. По результатам исследования был написан литературный обзор. Основная задача обзора сводилась к тому, чтобы дать ответ на вопрос: какие подходы к обучению интеллектуальных агентов наиболее актуальны, то есть изучаются наиболее активно в последнее время.

В ходе написания обзора были получены определённые результаты.

Всего во время написания обзора было проанализировано более 400 статей, опубликованных в период с 2014 по 2019 год. Из них определённым образом была отобрана 51 избранная статья. Получено распределение статей по языкам: русскоязычные – 5 (10%), англоязычные – 46 (90%). Получено распределение статей по источникам: Springer Link – 46 (90%), КИБЕРЛЕНИНКА – 5 (10%).

Из всех рассмотренных источников более половины (53%, 27 источников) описывают применение конкретно обучения с подкреплением. Также вниманием исследователей не обделён нейросетевой подход (21%, 11 источников), а также глубокое обучение и генетические алгоритмы (по 14%, 7 источников). Следом расположилось Q-обучение (12%, 6 источников).

В подавляющем большинстве случаев исследователи предлагают применять обучение конкретно к модели поведения, основанной на деревьях поведения.

Таким образом, в результате проделанной в области изучения научных статей работы был сделан вывод, что наиболее отвечающим требованиям исследования является подход, сочетающий в себе применение обучения с подкреплением в качестве алгоритма обучения и деревьев поведения в качестве модели поведения агентов.

Полный текст обзора приведён в Приложении А.

Таким образом, в результате исследования стало очевидным, что наиболее актуальным подходом к обучению интеллектуальных агентов в настоящее время является подход, основанный на применении обучения с подкреплением к модели, основанной на деревьях поведения.

# 1.4 Выбор алгоритмов для реализации

В рамках проведённого исследования на тему моделей поведения агентов, к которым чаще всего применяются методы машинного обучения, было установлено, что чаще всего исследователями используется модель поведения на базе деревьев поведения.

# 1.5 Выбор средств реализации

Исследование возможностей моделирования поведения интеллектуальных агентов производится исследователями, как правило, в игровых средах. Некоторые исследователи предпочитают использовать для своих работ уже готовые игры, такие как Super Mario (или его модификация для задач машинного обучения MarI/O [?]), Ms. Pac-Man, StarCraft II и прочие. Другие исследователи разрабатывают собственную программную платформу, на базе которой и производится исследование.

Задача настоящего исследования сформулирована таким образом, что подразумевает разработку собственной программной платформы для нужд моделирования.

Рассматривая вариант создания собственной программной платформы, необходимо сделать выбор в пользу одного из подходов к реализации.

Первый подход заключается в программировании окружения с нуля. Такой подход в настоящее время достаточно редко применяется в видеоигровой индустрии, однако обладает как своими преимуществами, так и недостатками.

Среди преимуществ можно отметить то, что созданный собственными силами программный базис для создания собственной игры (или программной платформы) будет полностью отвечать нуждам разработчика, так как при создании этого базиса разработчик будет заранее руководствоваться своими последующими нуждами в создании конкретной программы на его базисе, что позволит учесть необходимые нюансы и облегчить процесс дальнейшей разработки.

Среди недостатков необходимо отметить то, что создание такого программного базиса потребует очень больших трудовых и временных ресурсов.

В случае следования стратегии разработки собственного программного продукта с нуля также необходимо выбрать язык программирования, на котором будет реализована платформа. В качестве требований, предъявляемых к языку, можно отметить высокое быстродействие, так как для проекта важна обработка состояний множества различных агентов в каждом такте работы программы, а также возможность применения объектно-ориентированного подхода.

В таблице 1 продемонстрирована скорость работы различных языков программирования при решении одной и той же задачи.

Таблица 1 – Сравнительная таблица скорости выполнения программы на разных языках программирования

|  |  |
| --- | --- |
| Язык программирования | Время выполнения, с |
| C | 27,52 |
| C++ | 28,60 |
| Go | 30,91 |
| Java | 37,90 |
| Rust | 30,20 |

Из сравнительной таблицы видно, что C++ уступает только языку C, своему предку. При изменении объёма данных C++ может оказаться быстрее, чем C, что связано с особенностями работы оптимизации [9].

Второй подход заключается в использовании игрового движка. У этого подхода также можно выделить существенные преимущества и недостатки.

Среди преимуществ подхода самым важным можно назвать отсутствие необходимости реализовать базовый функционал. Так, в большинстве современных игровых движков достаточно подключить один плагин, чтобы добавить в проект, например, гравитацию, в то время как при собственноручном написании программного базиса было бы необходимо реализовывать её самостоятельно. Также важным преимуществом использования игрового движка можно назвать то, что получившийся программный продукт впоследствии может быть использован в аналогичных проектах на том же движке.

Среди недостатков такого подхода можно отметить узкую специфику результирующей программы: её можно будет использовать только в проектах на том же движке, на котором она была реализована. Кроме того, при реализации проекта на игровом движке необходимо изучить документацию движка и научиться уверенно с ним работать.

В случае следования стратегии использования игрового движка необходимо выбрать один из существующих движков, который наиболее подойдёт для решения поставленной задачи. К настоящему моменту насчитывается большое количество игровых движков, подходящих под различные нужды. Среди требований, предъявляемых к игровому движку, необходимо отметить широкий спектр применения, чтобы разработанную программу было возможно применить в как можно большем количестве проектов, а также проприетарность.

В результате отбора по выделенным критериям на выбор остаётся два наиболее популярных в настоящее время игровых движка: Unity3D [888] и Unreal Engine [889].

Unity3D – это межплатформенная среда для разработки компьютерных игр, которая позволяет вести разработку более чем под двадцать различных операционных систем. В числе основных преимуществ Unity выделяются наличие визуальной среды разработки, межплатформенной поддержки и модульной системы компонентов. В числе недостатков – сложности при работе с многокомпонентными схемами и затруднения при подключении внешних библиотек [900]. Скриптинг в среде Unity производится на языках программирования C#, JavaScript или Boo (диалект Python).

Unreal Engine – это игровой движок от Epic Games, который позволяет разрабатывать игры для разных операционных систем, таких как Windows, Mac OS, Linux и прочие. Актуальной версией движка является Unreal Engine 4. Среди преимуществ можно отметить то, что, в отличие от Unity, Unreal Engine изначально укомплектован всеми необходимыми инструментами разработки. Также стоит отметить, что в Unreal Engine используется преимущественно язык программирования C++, что существенно упрощает работу пользователям, знакомым с этим языком, а для тех, кто не знаком, существует инструментарий Blueprints, позволяющий осуществлять визуальное программирование путём постановки блоков и их связей в нужной последовательности. Самым же главным недостатком этого движка с точки зрения настоящего исследования можно назвать проблемы с обработкой большого количества агентов в сцене, что делает его практически непригодным для реализации требуемой программной платформы. Кроме того, порог вхождения в Unreal Engine гораздо выше, чем в Unity: Unity рассчитан на разработчиков-новичков, а Unreal – на профессионалов [901][902].

По результатам рассмотрения возможных вариантов было принято решение использовать для разработки программной платформы движок Unity. Помимо обозначенных, Unity обладает ещё рядом преимуществ в контексте выполнения диссертации, таких как более эффективная работа с большим числом агентов. Однако самым главным преимуществом можно назвать наличие в движке встроенного фреймворка для машинного обучения Unity ML.

Unity ML – это масштабная библиотека, разрабатываемая разработчиками движка. В комплекте с библиотекой разработчики также поставляют набор реализаций различных алгоритмов обучения на базе TensorFlow.

По результатам работы над проектом разработчиками был написан и опубликован научный материал, который доступен по ссылке [12]. В нём описывается использование библиотеки в практических целях. Так, описывается и последовательность действий по внедрению SDK ML-Agents в свой проект.

Набор ML-Agents предоставляет разработчику игры на Unity всё необходимое для создания самообучающегося агента и его взаимодействия с симуляцией.

Как только SDK импортирован в проект Unity, сцены на движке могут быть превращены в обучающие площадки для агентов. Это достигается путём использования трёх сущностей, описанных в SDK: Agent, Brain и Academy. Схема взаимодействия приведена на рисунке 3.

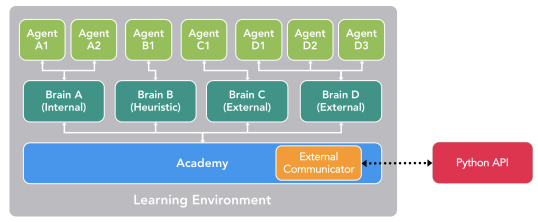


Рисунок 3 – Схема взаимодействия Agent, Brain и Academy

Сущность Agent отвечает за восприятие мира агентом и выполнение им каких-либо действий. Сущность Brain контролирует поведение группы агентов, привязанных к ней. Сущность Academy отвечает за глобальную координацию действий агентов и общее состояние симуляции.

Для функционирования этого набора после импорта SDK в проект необходимо определить, какие объекты представляют сущности Agent, Brain и Academy. Объекты Agent – это агенты внутри сцены, к которым добавляется компонент Agent из SDK. Объекты Brain и Academy абстрактны, но имеют соответствующие компоненты, связанные с компонентами объектов уровнем ниже: так, объект Academy связан с несколькими объектами Brain, каждый из которых связан с несколькими объектами Agent. Благодаря такой иерархии удаётся добиться построения мультиагентной системы с различными популяциями: общие шаблоны поведения регулируются объектом Brain, к которому может быть присоединено сколько угодно конечных объектов Agent, формирующих популяцию агентов, разделяющих общую модель поведения.

Механизм взаимодействия Brain и Agent может быть представлен следующим образом. Agent опрашивает Brain с некоторой заданной частотой и получает от него инструкции поведения. При этом функция вознаграждения, необходимая для обеспечения непосредственно обучения, может быть обновлена или задана в каждый момент времени с помощью соответствующих скриптов.

# 2 Проектирование проекта проектного проектирования

# 2.2 Проектирование тестовой платформы

Для функционирования агентов и возможности демонстрировать их способности необходимо разработать тестовую платформу, в пределах которой агенты будут способны свободно перемещаться и контактировать с окружением.

Первая задача в рамках проектирования тестовой платформы – выделить необходимые основные структурные особенности. Платформа должна быть достаточно просторной, чтобы вмещать две популяции по N агентов. Для удобства пол платформы будет плоским, а края огорожены непроницаемой стеной, которую не смогут преодолеть агенты.

Следующая задача состоит в наполнении полученной платформы. Поскольку показатели сытости и жажды агентов могут изменяться с течением времени, в пределах арены необходимо разместить объекты, которыми агенты смогут пополнять свои запасы пищи и питья. Для разнообразия и усложнения поведения агентов внутри каждой из категорий будут выделены несколько различных видов объектов:

- пища:

- яблоко: восстанавливает K сытости и P здоровья;

- хот-дог: восстанавливает Z сытости и L здоровья;

- питьё:

- вода: убирает Z жажды и добавляет M здоровья.

Помимо этого, в пределах платформы необходимо расположить объекты-аптечки, которые не влияют на жажду и сытость, а только восстанавливают здоровье агента. Аптечками агент сможет пользоваться в тех случаях, когда нет потребности тратить пищу на пополнение здоровья, или когда в распоряжении нет пищи.

В долгосрочной перспективе задача агента сводится в максимизации набранных очков, поэтому в пределах платформы нужно расположить объекты, которые будут приносить агенту очки. Таких объектов тоже можно выделить несколько:

- малый: приносит 1 очко;

- средний: приносит 3 очка;

- большой: приносит 5 очков;

- огромный: приносит 10 очков.

Расположение объектов на платформе должно подчиняться определённым законам. Для каждого вида объектов задаётся своя политика размещения.

Вся пища будет размещена случайно в пределах платформы, однако размещение зависит от ценности конкретного вида пищи. Пища, которая восстанавливает больше очков сытости и здоровья, будет распределена по такому закону, чтобы чаще всего её можно было найти ближе к центру платформы, где за неё будет наиболее высокая конкуренция. Менее ценная пища будет, наоборот, расположена ближе к краям платформы, где её будет проще добыть. Пополнение запасов пищи будет происходить следующим образом: в случайные промежутки времени в случайных местах в соответствии с описанным выше законом размещения будут появляться объекты класса «пища», причём вероятность появления более ценного вида пищи ниже, чем менее ценного.

Питьё будет размещаться в пределах платформы по аналогичному принципу.

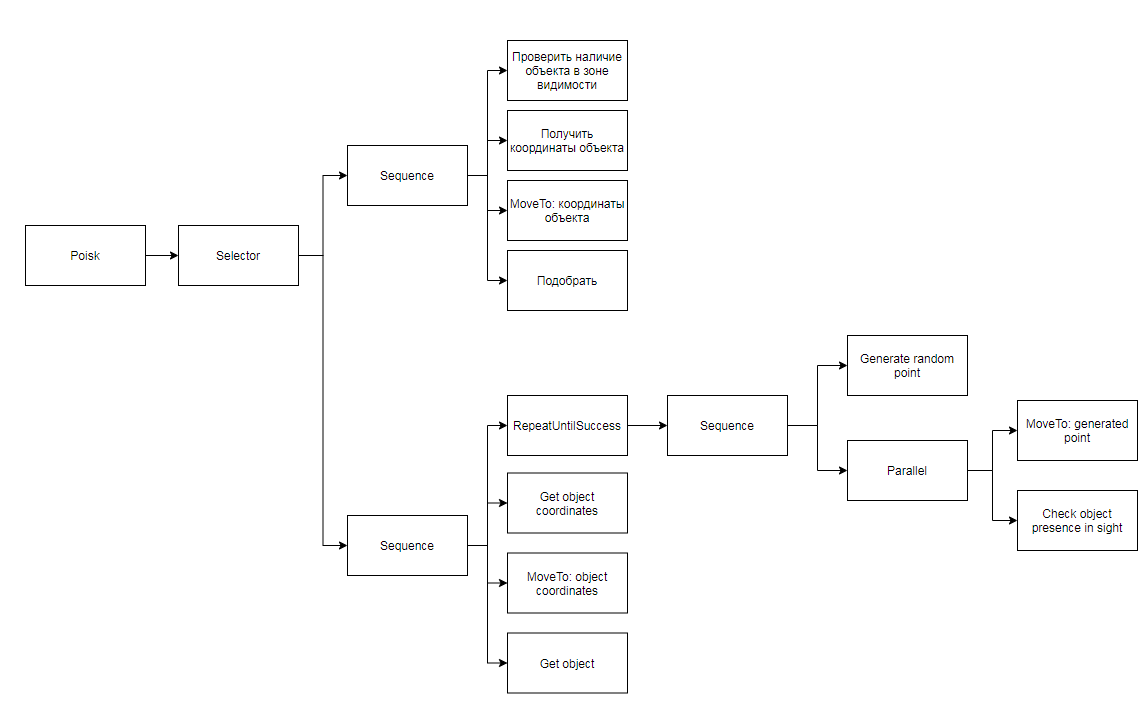
Объекты, которые приносят агентам очки, будут размещаться по другой политике. Для каждого вида таких объектов будет установлен отдельный таймер, по истечении которого новый объект этого подвида будет появляться в случайном месте в пределах платформы. Принцип размещения объектов аналогичен случаю с пищей, однако, если в том случае существовала минимальная вероятность, что более ценный вид пищи появится на границе платформы, в данном случае такой вероятности нет: для каждого подвида объектов будут установлены рамки, в пределах которых он может появиться. Так, наименее ценные объекты, приносящие одно очко, могут быть расположены хаотично везде в пределах платформы, объекты, приносящие три очка, уже не появятся на границе платформы и будут сосредоточены ближе к центру. Самый ценный объект, приносящий 10 очков, будет появляться в малом радиусе от центра платформы. Соответствующим образом будут установлены и таймеры: менее ценные объекты будут появляться чаще, более ценные – реже. Количество объектов, находящихся на платформе, также будет регулироваться схожим правилом: более ценные объекты встречаются реже ввиду меньшего их количества. Также, по достижении определённого количества объектов какого-либо из подвидов такие объекты перестают появляться до тех пор, пока их количество на платформе не сократится.

Чтобы внести ещё больше разнообразия в поведение агентов, необходимо добавить в платформу опасные зоны. Пространство в определённом радиусе от центра платформы будет обладать постоянным эффектом, который отнимает у агента фиксированное количество здоровья в секунду.

# 2.3 Проектирование модели поведения интеллектуальных агентов

Таким образом можно реализовать весь функционал деревьев поведения своими руками, не прибегая к использованию ассетов из Unity Asset Store.

Задачу поиска объекта было решено выделить в отдельное дерево поведения.



# 2.4 Проектирование модели обучения

# 2.5 Проектирование графического интерфейса пользователя

В рамках проектирования графического интерфейса пользователя для разрабатываемой программы были созданы макеты всех необходимых окон. По разработанным макетам впоследствии будет произведена разработка окон программы.

Стартовое окно программы должно включать в себя возможности настройки создаваемой симуляции в соответствии с техническим заданием на программу.

В соответствии с пунктом 4.1.2 Технического задания (см. Приложение Б), входные данные должны быть представлены следующим набором:

- две выбранные популяции агентов: «умная», «рискованная», «сбалансированная» и «осторожная»;

- время симуляции: от 2 до 5 минут;

- частота спавна объектов: от 0,5 до 2;

- количество объектов сбора: от 10 до 100;

- количество пищи: от 10 до 100;

- количество воды: от 10 до 100.

Таким образом, в пользовательском интерфейсе необходимы виджеты, с помощью которых можно было бы устанавливать перечисленные параметры. Для разных параметров были выбраны разные способы их установки.

Так, выбор популяций агентов-соперников пользователю предлагается осуществлять с помощью выпадающих списков, наполненных доступными значениями. Установка оставшихся параметров может быть удобно осуществлена посредством использования слайдеров с заранее установленными пограничными значениями.

Исходя из этого, можно перечислить виджеты, необходимые для корректной работы стартового окна:

- выпадающий список для выбора первой популяции;

- выпадающий список для выбора второй популяции;

- слайдер для установки количества пищи;

- слайдер для установки количества воды;

- слайдер для установки количества объектов сбора;

- слайдер для установки частоты спавна объектов;

- слайдер для установки времени симуляции.

Такой выбор виджетов позволяет решить серьёзную проблему, связанную с валидацией, без которой пользователь смог бы ввести в программу неподходящие значения. В случае использования выпадающих списков и слайдеров такой проблемы не возникает, поскольку у таких виджетов заранее установлены возможные значения. В случае указанных параметров все ограничения обозначены со включением граничных значений: так, для параметра «время симуляции» и соответствующего ему слайдера устанавливается диапазон значений «от 2 до 5», что переводится на язык строгих выражений в виде [2; 5].

В качестве пограничных значений для слайдеров были установлены значения из пункта «входные данные» Технического задания. Выпадающие списки также были заполнены только опциями из этого пункта.

Заключительным элементом пользовательского интерфейса стартового окна является кнопка, которая необходима для сохранения введённых данных и перехода к главному окну программы. В качестве кнопки предполагается использования стандартного виджета Push Button.

Полученный в результате проектирования макет экранной формы представлен на рисунке МНОГО.

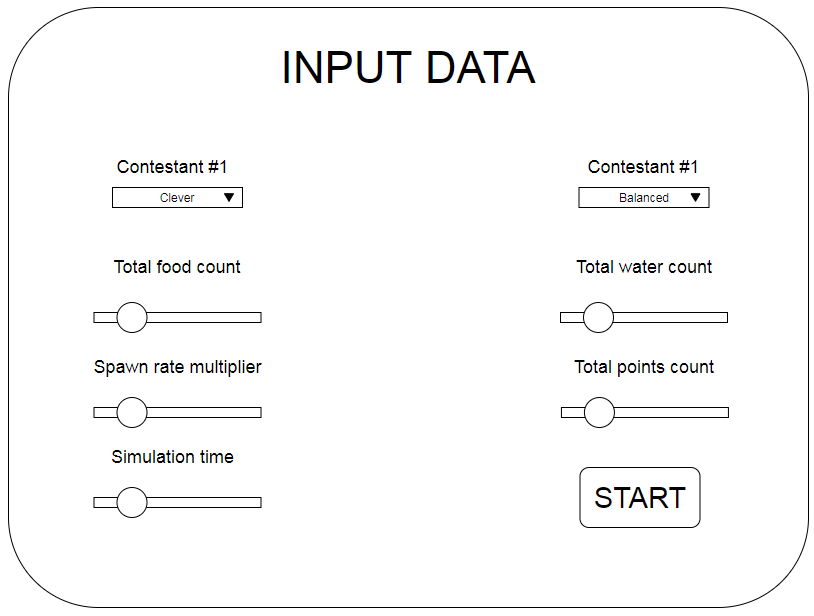


Рисунок МНОГО – Макет экранной формы стартового окна.

По нажатии на кнопку создания симуляции должно происходить сохранение введённых данных и переход к основному окну программы.

Основное окно программы предназначено для предоставления пользователю моментальной информации о текущем состоянии арены и её обитателей. В этом окне пользователю должна быть представлена следующая информация:

- количество и положение объектов сбора;

- количество и положение пищи;

- количество и положение воды;

- положение агентов;

- текущие показатели состояния агентов;

- оставшееся время симуляции.

Основная часть перечисленных параметров может быть представлена визуально: положение всех объектов и их количество видно на изображении самой арены. Для этого внутри арены устанавливается камера, которая следит за состоянием арены. Картинка с камеры предоставляется пользователю, а поверх этой картинки добавляются элементы графического интерфейса.

Таким образом, графический интерфейс главного окна должен дополнять картинку с камеры и позволять пользователю видеть оставшиеся неохваченными параметры, а именно:

- оставшееся время симуляции;

- показатели состояния агентов.

Было принято решение визуально разграничить агентов разных популяций (команд). Так, агенты популяции №1, выбранной в стартовом окне, будут отображены в специальном боксе в левой части экрана, агенты популяции №2 – в специальном боксе в правой части экрана. Такие боксы принято называть «ушами».

В «ушах» в виде списков выводится информация о состоянии каждого агента. К этой информации относятся:

- данные о текущем здоровье агента;

- данные о текущей сытости агента;

- данные о текущей жажде агента;

- данные о текущем количестве набранных агентом очков;

- данные о количестве каждого из видов объектов сбора, поднятых агентом.

Время, оставшееся до конца симуляции, будет отображаться в специальном боксе в нижней части экрана.

Полученный в результате проектирования макет экранной формы основного окна программы представлен на рисунке МНОГО+1.

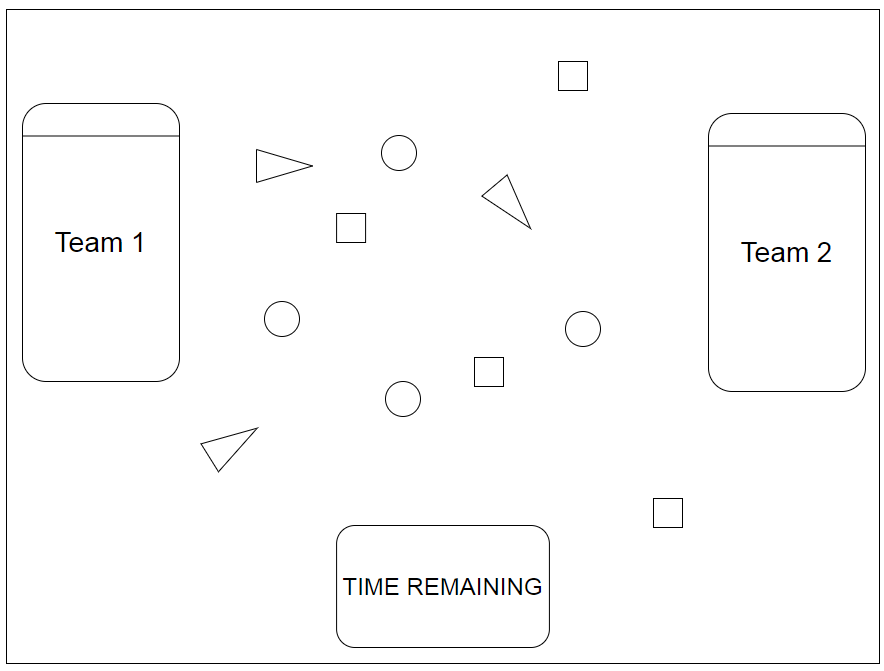


Рисунок МНОГО+1 – Макет экранной формы основного окна.

Третье окно программы должно быть предоставлено пользователю после основного. Основное окно программы закрывается по истечении таймера, после чего собранная информация сохраняется и переносится в следующее, заключительное окно программы.

Заключительное окно программы служит для предоставления пользователю информации о том, какая из команд одержала победу, а также для отображения в виде графиков данных о том, как происходил набор очков каждым из агентов в отдельности, всей командой в целом, как менялся набор в динамике и прочих.

Полученный в результате проектирования макет экранной формы заключительного окна представлен на рисунке МНОГО+3.

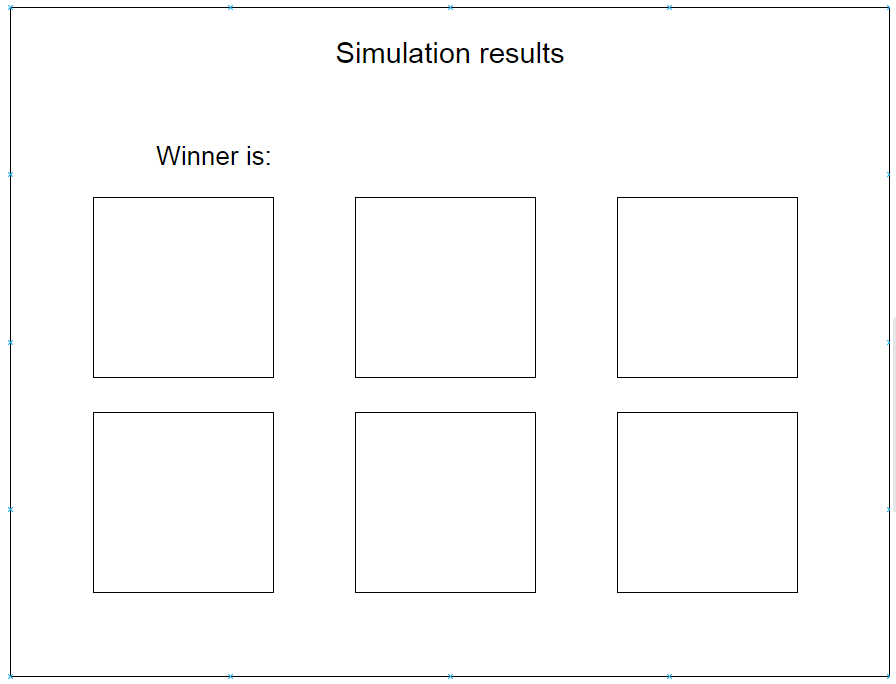


Рисунок МНОГО+3 – Макет экранной формы заключительного окна.

По спроектированным экранным формам представляется возможным произвести разработку непосредственно окон программы, однако сами окна могут претерпеть незначительные изменения относительно своих макетов в пользу повышения удобства пользования ими.

# 3 Описание программной реализации

# 3.1 Разработка тестовой платформы

Тестовая платформа должна обеспечивать возможность корректной симуляции поведения популяций интеллектуальных агентов. Из такой постановки можно выделить основные качества, которые должны быть присущи платформе:

- производительность: платформа должна корректно обрабатывать одновременные действия как минимум двух популяций по 8 агентов (16 агентов в сумме);

- достаточное пространство: в пределах арены должно быть достаточно места для автономного функционирования минимум 16 агентов, чтобы им не пришлось сталкиваться слишком часто или слишком агрессивно конкурировать за ресурсы;

- восполнение: платформа должна обладать свойством восполнять ресурсы, находящиеся на платформе, после того, как они были собраны кем-либо из агентов.

Первая задача, требующая решения – создание непосредственно арены необходимой формы и площади, чтобы обеспечить достаточно места для функционирования всех агентов. Размер каждого агента был выбран таким образом, чтобы с ним было удобно работать в редакторе Unity, и составил 2х2х2 условные единицы размера. В качестве формы агенты была принята капсула. Таким образом, горизонтальная площадь агента на платформу составила 4 единицы квадратные.

Было решено выбрать размеры арены равными 30х30 условных единиц. Форма арены – квадрат, таким образом, её площадь составляет 900 условных единиц, однако, при пересчёте единиц стоит учесть, что линейный размер капсулы в 10 раз меньше линейного размера плоскости, то есть объекта, являющегося полом. Таким образом, если принять размер капсулы за базовый, площадь арены составит 300х300 единиц, что равно 90000 единицам квадратным.

Исходя из таких соображений, можно вычислить, что на одного агента приходится более 1400 единиц квадратных, чего более чем достаточно для комфортного сосуществования.

Внешние границы арены ограждены непроницаемыми стенами, через которые агенты проходить не могут. Вид платформы, на котором можно увидеть размеры арены и агентов, представлен на рисунке \*\*\*.

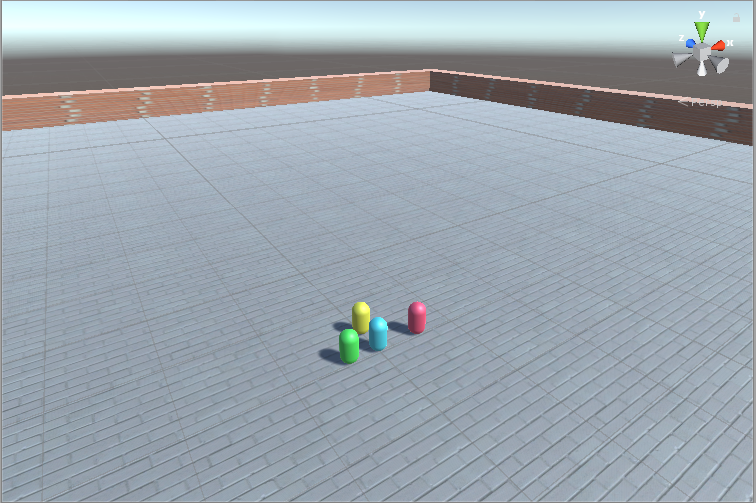


Рисунок \*\*\* - Вид арены с агентами.

После того, как арена физически существует, необходимо наполнить её объектами.

На этапе проектирования платформы было решено ограничиться следующими типами объектов:

- пища двух видов: яблоко и хот-дог;

- жидкость одного вида: вода;

- аптечки;

- объекты сбора четырёх видов.

Для каждого вида объектов были подобраны необходимые модели, которые будут визуально представлять эти объекты в сцене. Все модели, использованные в проекте, были получены с сайта TurboSquid [???], где находятся в свободном доступе.

Модель яблока представлена на рисунке 100500.



Рисунок 100500 – Модель яблока.

Модель хот-дога представлена на рисунке 100501.



Рисунок 100501 – Модель хот-дога.

Модель бутылки с водой представлена на рисунке 100502.



Рисунок 100502 – Модель бутылки с водой.

Модель аптечки представлена на рисунке 100503.

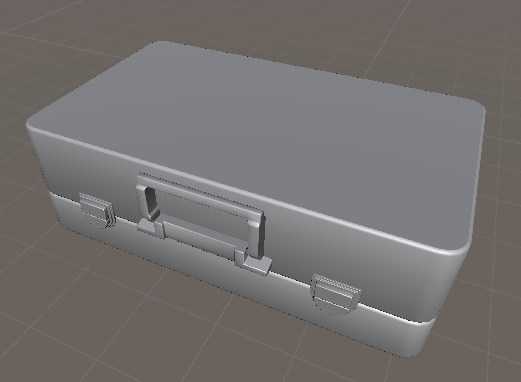


Рисунок 100503 – Модель аптечки.

В качестве модели объекта сбора выступила обычная сфера – встроенный объект Unity. Эта модель представлена на рисунке 100504.

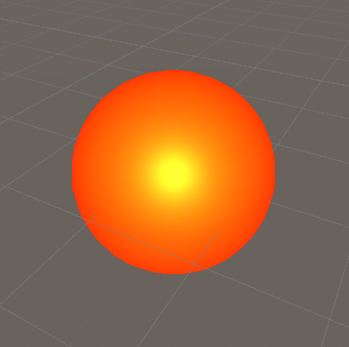


Рисунок 100504 – Модель объекта сбора.

Чтобы различать объекты сбора разной ценности, было принято решение написать отдельный скрипт, который будет менять размер модели в зависимости от ценности объекта, который она представляет.

На основе полученных объектов были сконструированы необходимые префабы. Префабом (prefab от pre-fabricated) в Unity называется игровой объект, который заранее подготовлен для последующей (возможно, многократной) репликации (инстанцирования). Функционал префабов приходится очень кстати в тех ситуациях, когда необходимо продуцировать многие одинаковые объекты во время игры (например, с помощью префабов часто реализуется стрельба в играх: при каждом выстреле инстанцируется новая копия объекта «пуля»).

Префабы необходимы для корректной работы спавнеров, так как алгоритм их работы включает в себя логику, согласно которой количество объектов должно восполняться со временем: как только какой-то из агентов подбирает объект, запускается таймер, по истечении которого должен быть сгенерирован и размещён новый объект на замену исчезнувшему.

Для дальнейшей работы необходимы префабы всех объектов, которые будут размещаться спавнерами. Префабы будут хранить состояние объекта, а также его параметры, с которыми он будет размещён в сцене в случае необходимости, включая модель, материал и все скрипты, присоединённые к игровому объекту.

После того, как все префабы подготовлены, необходим объект, который будет следить за наполнением сцены объектами, осуществлять их расстановку и вычислять, в какие моменты времени необходимо добавить к уже существующим объектам новые. Для этого был создан первый необходимый объект Spawner.

Spawner представляет собой пустой игровой объект (Empty GameObject), который расположен в геометрическом центре арены и не имеет модели и внешнего представления. Этот объект служит только для того, чтобы привязать к нему скрипты, необходимые для регулирования числа и расстановки объектов.

По количеству объектов было создано необходимое количество скриптов:

- скрипт collectableSpawnController служит для осуществления расстановки и контроля количества объектов сбора;

- скрипт foodSpawnController служит для осуществления расстановки и контроля количества пищи;

- скрипт waterSpawnController служит для осуществления расстановки и контроля количества воды;

- скрипт medkitSpawnController служит для осуществления расстановки и контроля количества аптечек;

- скрипт agentSpawnController служит для первоначальной расстановки агентов внутри арены.

Кроме agentSpawnController, все скрипты реализуют одну и ту же логику. Алгоритм работы этих скриптов представлен на схеме на рисунке 100505.

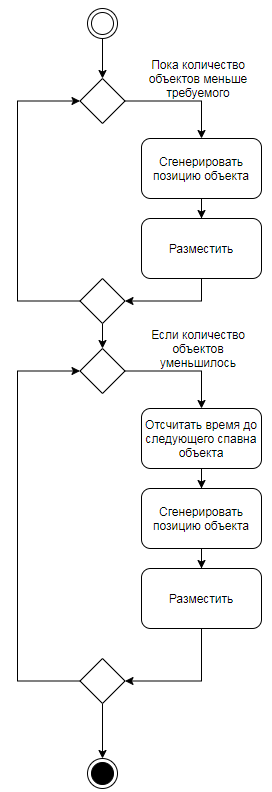


Рисунок 100505 – Схема работы спавнеров.

Таким образом, логика каждого скрипта представлена определёнными блоками:

- генерация позиции: в зависимости от типа объекта, который должен быть размещён на арене, для него случайным образом генерируется позиция: так, наиболее ценные объекты сбора появляются на случайных позициях около центра арены, а наименее ценные могут быть разбросаны по всей её площади;

- размещение объекта: в качестве параметра в скрипт подаётся префаб объекта, который должен быть размещён в сцене;

- слежение за общим количеством объектов в сцене;

- отсчёт времени до следующего спавна объекта.

# 3.2 Разработка модели поведения

3.2.1 Разработка базового функционала агента

Базовый функционал агента включает в себя наличие нескольких основных модулей для решения поставленных перед агентом задач. В числе таких задач можно выделить:

- навигация: агент должен уметь находить путь в требуемую точку в пределах платформы;

- контроль состояния: агент должен быть способен отслеживать критические показатели своего состояния, которые могут повлиять на его дальнейшее поведение;

- сенсорика: агент должен быть способен видеть объекты в радиусе своего зрения.

Специфика реализации программы на Unity позволяет выносить отдельные компоненты в отдельные классы, которые, будучи подключёнными к необходимому объекту, будут способны обеспечить требуемый функционал. Таким образом, для каждого из требуемых модулей можно создать отдельный класс, в котором будет описан его функционал.

Так, для реализации зрения агента был запрограммирован класс InteractSystem. Этот класс реализует функционал, связанный с опознаванием объекта агентом и определением необходимости найденного объекта. На рисунке 999 изображено поле зрения агента. Агент изображён капсулой, поле зрения – параллелепипедом перед ним.

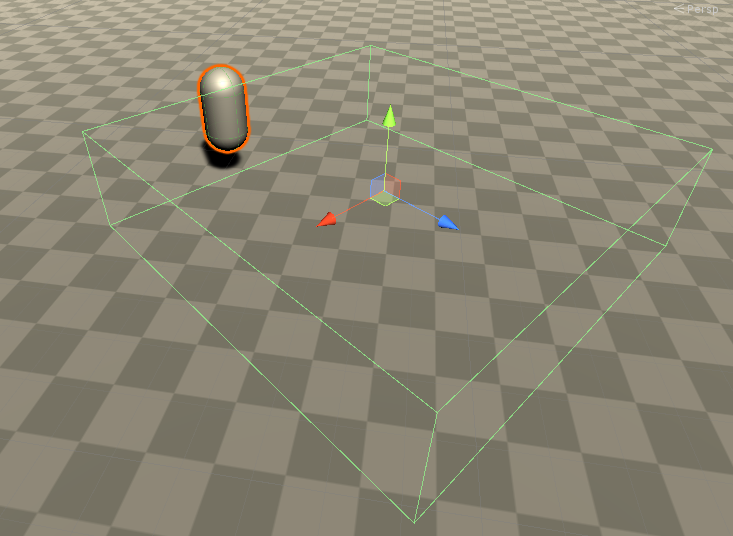


Рисунок 999 – Поле зрения агента

Объект, попавший в поле зрения агента, регистрируется и заносится в список. За это действие отвечает метод OnTriggerEnter. Поле зрения агента с точки зрения Unity является триггером. Как только объект попадает в пределы параллелепипеда, срабатывает функция вхождения объекта в зону действия триггера. На рисунке 1000 приведён фрагмент кода, отвечающий за это действие.

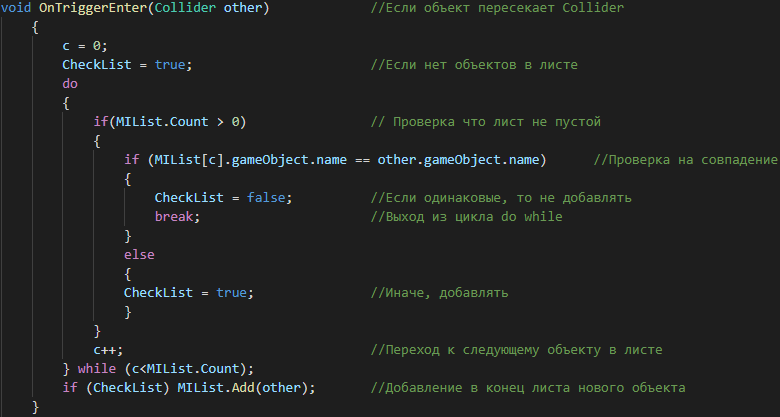


Рисунок 1000 – Фрагмент кода

Аналогичным образом реализована обратная функция. Как только объект выходит из поля зрения агента, срабатывает метод OnTriggerExit. Объект, пропавший из поля зрения, более не должен храниться в памяти агента, поэтому он удаляется из списка найденных объектов. Соответствующий фрагмент кода приведён на рисунке 1001.

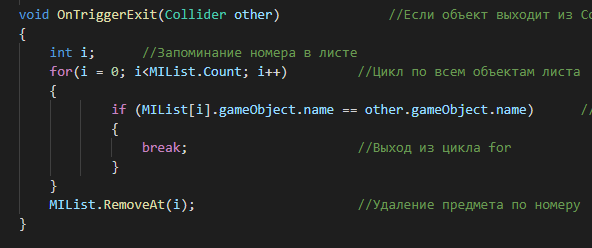


Рисунок 1001 – Фрагмент кода

На рисунке 8 можно увидеть, что на поверхности присутствует голубой слой, огибающий стены. Это и есть карта пространства, где агентам разрешено ходить.

Отдельный модуль – контроль состояния агента. Для его реализации также был создан отдельный компонент, который впоследствии был подключён к каждому агенту. Этот компонент отслеживает критические показатели состояния агента, такие как жажда, голод и показатель здоровья.

3.2.2 Разработка дерева поведения агента

Программная реализация дерева поведения агента может быть выполнена только после определения всех возможных последовательностей действий агента и связей между ними.

На данный момент агент обладает следующими возможностями:

- перемещаться в пределах арены;

- подбирать объекты;

- добавлять объекты (еду, питьё и аптечки) в свой инвентарь;

- доставать объекты из инвентаря и поглощать;

- осуществлять поиск нужного объекта;

-

Для построения деревьев поведения в Unity существуют специальные средства, которые могут упростить этот процесс. Существует ассет Behavior Designer [3], с помощью которого построение дерева поведения можно осуществлять наглядно в специальном редакторе. Однако, эта утилита на данный момент стоит $40, что делает её использование в рамках данной работы непрактичным.

Однако, есть и другие ассеты. Так, ассет Behaviour Machine [4] также предоставляет возможность построения деревьев поведения, но при этом предоставляет бесплатную версию.

3.2.2.1 Разработка модуля принятия решений

Модуль принятия решений служит для того, чтобы на основе текущего состояния агента и мира определить, какие действия необходимо совершить в первую очередь.

Модуль принятия решений было решено выделить в отдельный скрипт на языке C# и присоединить его в качестве компонента к агенту.

Для работы этого скрипта в первую очередь необходимо обратиться к другому компоненту – контроллеру состояния агента. Из этого компонента планировщик решений должен получить данные о текущем состоянии агента, а точнее – о том, находится ли агент в одном из критических состояний.

К числу критических состояний относятся состояния:

- голоден: показатель сытости агента упал ниже порогового значения;

- испытывает жажду: показатель насыщения жидкостью агента упал ниже порогового значения;

- умирает: показатель здоровья агента опустился ниже порогового значения.

На основе полученных данных планировщик решений должен определить, какие действия нужно совершать.

3.2.2.2 Разработка дерева поведения

# 3.3 Применение машинного обучения

# 3.4 Разработка графического интерфейса пользователя

Вид главного меню (стартового окна) программы представлен на рисунке 0.

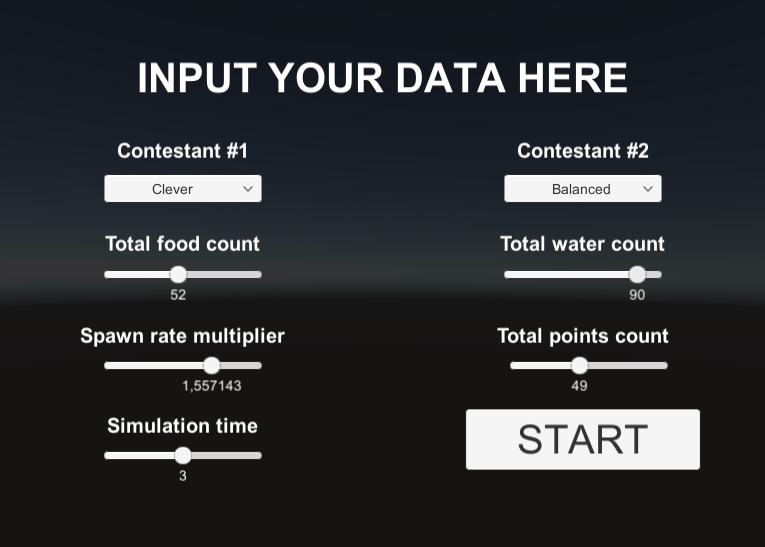


Рисунок 0 – Вид стартового окна программы.

На рисунке 0 хорошо видно, что все элементы графического интерфейса подписаны и интуитивно понятны.

В верхней части стартового окна располагаются два выпадающих списка, из каждого из которых можно выбрать одно из четырёх значений. Выбранные значения определяют, какие именно популяции агентов будут соревноваться в наборе очков.

На рисунке 00 представлено наполнение выпадающих списков.

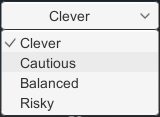


Рисунок 00 – Наполнение выпадающего списка.

Оставшиеся элементы – слайдеры.

Слайдер Total food count определяет, насколько сцена может быть заполнена объектами класса «еда», значение этого слайдера – целое число, которое может изменяться в границах от 10 до 100 единиц.

Слайдер Total water count действует по аналогичному принципу и определяет, сколько объектов класса «вода» может присутствовать одновременно в сцене. Значение этого слайдера тоже представлено целым числом в диапазоне от 10 до 100.

Слайдер Spawn rate multiplier определяет, насколько часто новые объекты появляются в сцене. Этот показатель представлен числом с плавающей запятой и может изменяться от 0,5 до 2. Показатель 2 будет означать, что объекты появляются вдвое чаще, чем обычно, показатель 0,5 – вдвое реже, чем обычно.

Слайдер Total points count представляет целое число в диапазоне от 10 до 100. Значение этого слайдера определяет, сколько объектов сбора может быть одновременно представлено в сцене. Исходя из этого значения, программно вычисляется количество каждого вида объектов сбора. <тут бы формулы, но я их ещё ведь не исправил>

Наконец, кнопка Start переводит пользователя в следующее окно программы. По нажатии на эту кнопку запускается особый скрипт, который необходимо рассмотреть подробнее.

тут описание скрипта

# 3.5 Результаты разработки

В результате разработки программной платформы на движке Unity и разработки интеллектуальных агентов для неё были

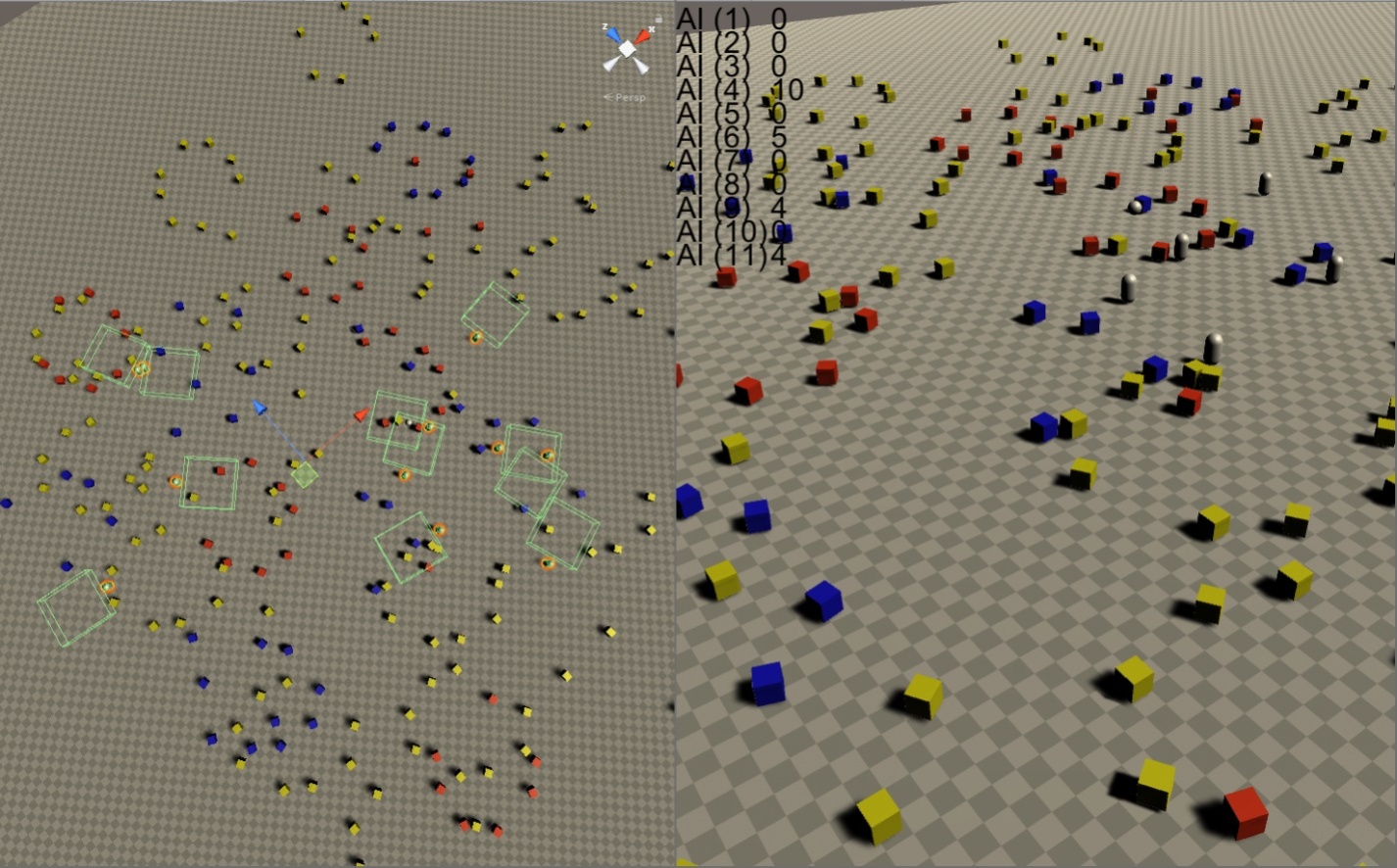


Рисунок 2000 – Вид программной платформы и агентов на ней

# 4 Тестирование

# 4.1 Описание видов тестирования

# 4.1.1 Функциональное тестирование

Суть функционального тестирования [1] заключается в установлении соответствия разработанного ПО исходным функциональным требованиям заказчика. Таким образом, в рамках функционального тестирования проверяется способность разработанной программы решать требуемые задачи в заданных условиях использования.

Различают два подхода к проведению функционального тестирования в зависимости от степени доступа к коду тестируемой программы:

- тестирование чёрного ящика проводится без доступа к коду программы;

- тестирование белого ящика проводится с доступом к коду программы.

В данном случае проводится тестирование белого ящика, поскольку есть доступ к коду программы.

# 4.1.2 Юзабилити-тестирование

Юзабилити-тестирование [2] (или проверка эргономичности) – это исследование, проводимое с целью определения, удобен ли объект для пользования. В качестве тестировщиков для этого вида тестирования привлекаются обычные пользователи.

В контексте юзабилити-тестирования программы проверяется, насколько удобен пользовательский интерфейс.

Процесс тестирования эргономики включает в себя несколько этапов.

Пользователю предлагается решить основные задачи, для выполнения которых был разработан продукт. Пользователь должен высказывать свои впечатления в процессе тестирования.

Весь процесс тестирования протоколируется с использованием видео или аудиоустройств. После проведения самого тестирования собранные данные структурируются и анализируются. Важно отдельно зафиксировать отдельные моменты:

- речь пользователя;

- выражение лица пользователя;

- изображение экрана компьютера, за которым сидит пользователь;

- события, происходящие на мониторе, такие как перемещения курсора, нажатия клавиш, переходы между экранами.

# 4.1.3 Тестирование стабильности

Тестирование стабильности [3] (или тестирование надёжности) предполагает тестирование системы со значительной нагрузкой, распределённой на значительный период времени. Этот вид тестирования проводится, чтобы определить, как будет вести себя система под нагрузкой.

Программное обеспечение может вести себя при работе в течение так, как предполагал разработчик, однако если испытывать приложение в течение нескольких часов или суток, могут возникнуть непредвиденные проблемы в виде утечки памяти или зацикливания, что приведёт к сбою системы и/или её непрогнозируемому поведению.

# 4.1.4 Модульное тестирование (юнит-тестирование)

Модульное тестирование, иногда блочное тестирование или юнит-тестирование — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок [4].

# 4.1.5 Тестирование на отказ и восстановление

Тестирование на отказ и восстановление (Failover and Recovery Testing) проверяет тестируемый продукт с точки зрения способности противостоять и успешно восстанавливаться после возможных сбоев, возникших в связи с ошибками программного обеспечения, отказами оборудования или проблемами связи (например, отказ сети). Целью данного вида тестирования является проверка систем восстановления (или дублирующих основной функционал систем), которые, в случае возникновения сбоев, обеспечат сохранность и целостность данных тестируемого продукта.

Тестирование на отказ и восстановление очень важно для систем, работающих по принципу “24x7”. Если Вы создаете продукт, который будет работать, например, в интернете, то без проведения данного вида тестирования Вам просто не обойтись. Т.к. каждая минута простоя или потеря данных в случае отказа оборудования, может стоить вам денег, потери клиентов и репутации на рынке.

Методика подобного тестирования заключается в симулировании различных условий сбоя и последующем изучении и оценке реакции защитных систем. В процессе подобных проверок выясняется, была ли достигнута требуемая степень восстановления системы после возникновения сбоя.

Для наглядности рассмотрим некоторые варианты подобного тестирования и общие методы их проведения. Объектом тестирования в большинстве случаев являются весьма вероятные эксплуатационные проблемы, такие как:

- отказ электричества на компьютере-сервере;

- отказ электричества на компьютере-клиенте;

- незавершенные циклы обработки данных (прерывание работы фильтров данных, прерывание синхронизации);

- объявление или внесение в массивы данных невозможных или ошибочных элементов;

- отказ носителей данных.

Данные ситуации могут быть воспроизведены, как только достигнута некоторая точка в разработке, когда все системы восстановления или дублирования готовы выполнять свои функции. Технически реализовать тесты можно следующими путями:

- симулировать внезапный отказ электричества на компьютере (обесточить компьютер);

- симулировать потерю связи с сетью (выключить сетевой кабель, обесточить сетевое устройство);

- симулировать отказ носителей (обесточить внешний носитель данных);

- симулировать ситуацию наличия в системе неверных данных (специальный тестовый набор или база данных).

При достижении соответствующих условий сбоя и по результатам работы систем восстановления, можно оценить продукт с точки зрения тестирования на отказ. Во всех вышеперечисленных случаях, по завершении процедур восстановления, должно быть достигнуто определенное требуемое состояние данных продукта:

- потеря или порча данных в допустимых пределах;

- отчет или система отчетов с указанием процессов или транзакций, которые не были завершены в результате сбоя.

Тестирование на отказ и восстановление – это продукт-специфичное тестирование. Разработка тестовых сценариев должна производиться с учетом всех особенностей тестируемой системы. Принимая во внимание довольно жесткие методы воздействия, стоит также оценить целесообразность проведения данного вида тестирования для конкретного программного продукта [5].

# 4.1.6 Прочие виды тестирования

Помимо обозначенных, существуют также такие виды тестирования, как:

- тестирование безопасности;

- тестирование взаимодействия;

- приёмочное тестирование;

- интеграционное тестирование;

- системное тестирование;

- стрессовое тестирование;

- нагрузочное тестирование;

- объёмное тестирование;

- тестирование установки;

- конфигурационное тестирование;

- дымовое тестирование;

- регрессионное тестирование;

- тестирование сборки;

- санитарное тестирование или проверка согласованности/исправности [6].

# 4.2 Тестирование программы

# 5 Проведение вычислительного эксперимента

# 5.1 Постановка задачи

Основная задача настоящей диссертационной работы состоит в том, чтобы определить, насколько целесообразно применять методы машинного обучения к моделированию поведения интеллектуальных агентов.

Для того, чтобы установить факт целесообразности или нецелесообразности применения методов машинного обучения, необходимо сравнить по объективным критериям поведение двух различных популяций агентов. Одна популяция снабжается моделью поведения, не обезображенной интеллектом. Поведение этой популяции агентов будет регулироваться исключительно самим механизмом деревьев поведения, по которым агенты будут выбирать наиболее подходящие для каждой ситуации действия в зависимости от текущего состояния самого себя и мира.

Вторая популяция агентов снабжается моделью поведения с применением обучения. Поведение агентов этой популяции уже не будет жёстко зависеть от правил, изначально заданных разработчиком, оно не будет таким детерминированным.

Для сравнения эффективности работы каждой из популяций необходимо сформулировать объективные критерии оценки. В качестве таких критериев оценки предлагается использовать:

- общий счёт очков: во время своего функционирования агент путешествует по платформе и собирает объекты, приносящие ему очки. Количество очков, полученных к определённому моменту, будет считаться показателем эффективности деятельности агента;

- время жизни: внутри платформы объявляется фактор воздействия, приводящий агентов к состоянию смерти. Популяции будут сравниваться по показателям, связанным со временем жизни: максимальная продолжительность, средняя, мода продолжительности жизни, минимальная продолжительность и т. д.;

- комбинированный подход: наиболее эффективной будет считаться популяция, агенты которой сумели достичь максимального (максимального среднего, максимальной моды) показателя, вычисленного на основе собранных очков и продолжительности жизни.

# 5.2 Проведение эксперимента

# 5.3 Результаты эксперимента

В результате проведения эксперимента были достигнуты следующие результаты.

<тут много графиков по разным показателям с разной сортировкой>

По результатам эксперимента можно сделать вывод, что наиболее успешной по совокупности различных показателей стала популяция . . .

# Заключение

# Список использованной литературы

888 <https://unity.com/ru>

889 <https://www.unrealengine.com/en-US/>

900 <https://ru.wikipedia.org/wiki/Unity_(%D0%B8%D0%B3%D1%80%D0%BE%D0%B2%D0%BE%D0%B9_%D0%B4%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA)>

??? <https://www.turbosquid.com/>

901 <https://ru.wikipedia.org/wiki/Unreal_Engine>

902 <https://cubiq.ru/dvizhok-unreal-engine/>

<https://github.com/GrymmyD/UnityBehaviourTree>

1 Функциональное тестирование программного обеспечения [Электронный ресурс] / aplana.ru . – Режим доступа : <http://aplana.ru/services/testing/functionalnoe-testirovanie> (дата обращ. 28.05.18).

2 Юзабилити-тестирование [Электронный ресурс] / Wikipedia.org . – Режим доступа : <https://ru.wikipedia.org/wiki/Юзабилити-тестирование> (дата обращ. 28.05.18).

3 Тестирование стабильности [Электронный ресурс] / devopswiki.net . – Режим доступа : <http://devopswiki.net/index.php/Тестирование_стабильности> (дата обращ. 28.05.18).

4 Модульное тестирование // Википедия URL: <https://ru.wikipedia.org/wiki/Модульное_тестирование> (дата обращения: 26.11.2019).

5 Тестирование на отказ и восстановление (Failover and Recovery Testing) // Про Тестинг URL: <http://www.protesting.ru/testing/types/failover.html> (дата обращения: 26.11.2019).

6 Виды тестирования // Про Тестинг URL: <http://www.protesting.ru/testing/testtypes.html> (дата обращения: 26.11.2019).

Приложение А – Полный текст обзора литературы

Приложение Б – Техническое задание

Приложение В – Руководство пользователя