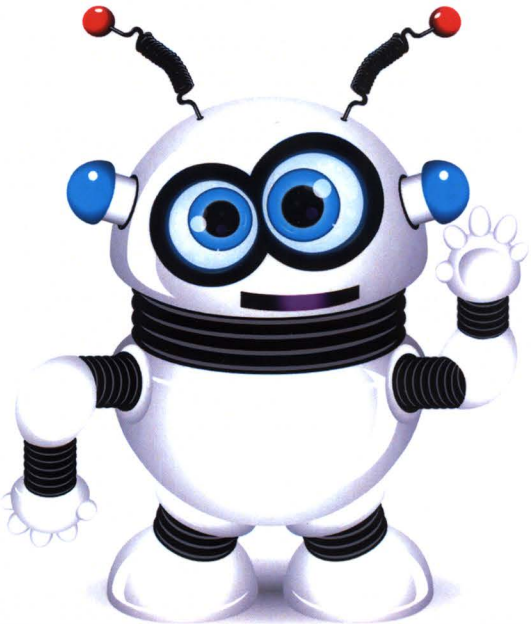
Сильные и мощные инструменты
для создания потрясающих воображение игр!

Unity 5.x Программирование искусственного интеллекта в играх



[PACKT] PUBLISHING

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Хорхе Паласиос

Хорхе Паласиос

Unity 5.x. Программирование искусственного интеллекта в играх

Adam Horton, Ryan Vice

Unity 5.x Game AI Programming Cookbook

BUILD AND CUSTOMIZE A WIDE RANGE OF POWERFUL UNITY AI SYSTEMS WITH OVER 70 HANDS-ON RECIPES AND TECHNIQUES

[PACKT] PUBLISHING open source community experience distilled

Хорхе Паласиос

Unity 5.x. Программирование искусственного интеллекта в играх

70 ПРАКТИЧЕСКИХ РЕЦЕПТОВ И МЕТОДИК СОЗДАНИЯ И НАСТРОЙКИ ШИРОКОГО СПЕКТРА МОЩНЫХ СИСТЕМ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА В UNITY

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2017

УДК 004.4'2Unity3D
ББК 32.972
П14

Паласиос Х.
П14 Unity 5.x. Программирование искусственного интеллекта в играх: пер. с англ. Р. Н. Рагимова. – М.: ДМК Пресс, 2017. – 272 с.: ил.

ISBN 978-5-97060-436-6

Игровой движок Unity 5 включает в себя множество инструментов, помогающих разработчикам создавать потрясающие игры, снабженные мощным искусственным интеллектом. Эти инструменты, вместе с прикладным программным интерфейсом Unity и встроенными средствами, открывают безграничные возможности для создания собственных игровых миров и персонажей. Данная книга охватывает как общие, так специальные методы, позволяющие реализовать эти возможности.

Издание задумывалось как исчерпывающий справочник, помогающий расширить навыки программирования искусственного интеллекта в играх. Рассматриваются основные приемы работы с агентами, программирование перемещений и навигации в игровой среде, принятие решений и координации. Описание построено на практических примерах, в виде легко реализуемых «рецептов».

УДК 004.4'2Unity3D
ББК 32.972

ISBN 978-1-78355-357-0 (анг.) © 2016 Packt Publishing
ISBN 978-5-97060-436-6 (рус.) © Оформление, перевод, ДМК Пресс, 2017

Содержание

Об авторе **8**

О технических рецензентах **9**

Предисловие **10**

Глава 1. Интеллектуальные модели поведения: перемещение **16**

Введение 16
Создание шаблона моделей поведения 17
Преследование и уклонение 21
Достижение цели и уход от погони 23
Поворот объектов 26
Блуждание вокруг 29
Следование по маршруту 31
Уклонение от встреч с агентами 36
Уклонение от стен 39
Смешивание моделей поведения с весовыми коэффициентами 41
Смешивание моделей поведения по приоритету 43
Комбинирование моделей поведения с применением конвейера управления 45
Стрельба снарядами 49
Прогнозирование места падания снаряда 50
Нацеливание снаряда 52
Создание системы прыжков 53

Глава 2. Навигация **60**

Введение 60
Представление игрового мира с помощью сетей 61
Представление игрового мира с помощью областей Дирихле 71
Представление игрового мира с помощью точек видимости 77
Представление игрового мира с помощью навигационного меша 81
Поиск выхода из лабиринта с помощью алгоритма DFS 84
Поиск кратчайшего пути в сети с помощью алгоритма BFS 86
Поиск кратчайшего пути с помощью алгоритма Дейкстры 88
Поиск оптимального пути с помощью алгоритма A* 91

Улучшенный алгоритм A* с меньшим использованием памяти – алгоритм IDA* 95
Планирование навигации на несколько кадров вперед: поиск с квантованием времени 98
Сглаживание маршрута 100

Глава 3. Принятие решений 103
Введение 103
Выбор с помощью дерева принятия решений 104
Работа конечного автомата 107
Усовершенствование конечного автомата: иерархические конечные автоматы 110
Комбинирование конечных автоматов и деревьев принятия решений 112
Реализация деревьев моделей поведения 113
Работа с нечеткой логикой 116
Представление состояний с помощью числовых значений: система Маркова 120
Принятие решений в моделях целенаправленного поведения 123

Глава 4. Координирование и тактика 126
Введение 126
Обработка формирований 127
Расширение алгортма A* для координации: алгоритм A*mbush 132
Выбор удобных точек позиций 136
Анализ точек позиций по их высоте 138
Анализ точек позиций по обзорности и незаметности 140
Оценка точек позиций для принятия решения 142
Карты влияния 143
Улучшение карт влияния путем заполнения 147
Улучшение карт влияния с помощью фильтров свертки 152
Построение боевых кругов 155

Глава 5. Органы чувств агентов 164
Введение 164
Имитации зрения с применением коллайдера 165
Имитация слуха с применением коллайдера 167
Имитация обоняния с применением коллайдера 171
Имитации зрения с применением графа 175
Имитация слуха с применением графа 176
Имитация обоняния с применением графа 179
Реализация органов чувств в стелс-игре 181

Глава 6. Настольные игры с искусственным интеллектом 189
Введение 189
Класс игрового дерева 190
Введение в алгоритм Minimax 192
Алгоритм Negamax 194
Алгоритм AB Negamax 196
Алгоритм Negascouting 199
Подготовка 199
Реализация соперника для игры в крестики-нолики 201
Реализация соперника для игры в шашки 206

Глава 7. Механизмы обучения 217
Введение 217
Предугадывание действий с помощью алгоритма прогнозирования N-Gram 217
Усовершенствованный иерархический алгоритм N-Gram 220
Использование классификаторов Байеса 222
Использование деревьев принятия решений 225
Использование закрепления рефлекса 229
Обучение с помощью искусственных нейронных сетей 234
Создание непредсказуемых частиц с помощью алгоритма поиска гармонии 238

Глава 8. Прочее 242
Введение 242
Улучшенная обработка случайных чисел 242
Соперник для игры в воздушный хоккей 245
Соперник для настольного футбола 251
Программное создание лабиринтов 261
Реализация автопилота для автомобиля 264
Управление гонками с адаптивными ограничениями 265

Предметный указатель 269

Об авторе

Хорхе Паласиос (Jorge Palacios) – профессиональный программист с более чем семилетним опытом. Последние четыре года занимался разработкой игр, работая на различных должностях, от разработчика инструментария до ведущего программиста. Специализируется на программировании искусственного интеллекта и игровой логики. В настоящее время использует в своей работе Unity и HTML5. Также является преподавателем разработки игр, лектором и организатором «геймджемов».

Больше узнать о нем можно на его личном сайте: http://jorge.palacios.co.

О технических рецензентах

Джек Донован (Jack Donovan) – разработчик игр и инженер-программист, работающий с движком Unity3D начиная с третьей версии. Учился в колледже Шамплейн (г. Берлингтон, штат Вермонт), где получил степень бакалавра в области программирования игр.

В настоящее время работает над проектом виртуальной реальности IrisVR в Нью-Йорке, где занимается разработкой программного обеспечения, позволяющего архитекторам создавать модели виртуальной реальности по CAD-моделям. До проекта IrisVR Джек работал в небольшой независимой студенческой команде, занимавшейся разработкой игр, написав при этом книгу «*OUYA Game Development By Example*».

Лорен С. Ферро (Lauren S. Ferro) – консультант по игрофикации (геймификации), дизайнер игр и схожих с играми приложений. Занимается проектированием, консультацией и разработкой стратегий для целого ряда проектов в области профессионального обучения, систем рекомендаций и общеобразовательных игр. Ведет активную исследовательскую работу в области геймификации, профилирования игроков и ориентированного на пользователей проектирования игр. Проводит семинары для специалистов и компаний, занимающихся разработкой игр и приложений с элементами игры, ориентированных на вкусы пользователей. Также является разработчиком методики Gamicards проектирования прототипов игр и приложений с элементами игры.

Предисловие

Стоит подумать об искусственном интеллекте, и в уме возникает множество ассоциаций. От простых моделей поведения, например преследование или убегание от игрока, до игры с искусственным интеллектом в классические шахматы, методов машинного обучения или процедурной генерации контента.

Движок Unity сделал разработку игр намного демократичнее. Благодаря простоте использования, быстрому совершенствованию технологий, постоянно растущему сообществу и новым облачным услугам движок Unity превратился в один из важнейших программных продуктов для игровой индустрии.

С учетом вышесказанного основной целью написания книги была попытка помочь вам, читатель, через описание технологий Unity, знакомство с передовым опытом, изучение теории, разобраться в идеях и методах искусственного интеллекта, что обеспечит вам преимущества как в любительской, так и в профессиональной разработке.

Эта книга рецептов познакомит вас с инструментами создания искусственного интеллекта, например для реализации достойных противников, доведенных до совершенства, и даже для разработки собственного нестандартного движка искусственного интеллекта. Она станет вашим справочным пособием при разработке методов искусственного интеллекта в Unity.

Добро пожаловать в увлекательное путешествие, которое сочетает в себе ряд вещей, имеющих для меня, как профессионала и просто человека, большое значение: программирование, разработка игр, искусственный интеллект и обмен знаниями с другими разработчиками. Не могу не отметить, как я польщен и рад, что вы читаете написанный мной текст прямо сейчас, и я благодарен команде издательства Packt за предоставление такой возможности. Надеюсь, что этот материал поможет вам не только поднять на новый уровень навыки работы с Unity и искусственным интеллектом, но и содержит описание функций, которыми вы заинтересуете пользователей своих игр.

О чем рассказывается в этой книге

Глава 1 «Интеллектуальные модели поведения: перемещение» описывает наиболее интересные алгоритмы перемещения, основанные на

принципах управления поведением, разработанных Крейгом Рейнольдсом (Craig Reynolds) совместно с Яном Миллингтоном (Ian Millington). Они стали основой большинства алгоритмов искусственного интеллекта в современных играх наряду с другими алгоритмами управления перемещением, такими как семейство алгоритмов определения маршрута.

Глава 2 «Навигация» описывает алгоритмы определения маршрута для сложных сценариев перемещения. Она рассматривает несколько способов представления игрового мира с помощью различных графовых структур и алгоритмы поиска путей в различных ситуациях.

Глава 3 «Принятие решений» демонстрирует различные методы принятия решений, достаточно гибкие, чтобы приспосабливаться к разным видам игр, и достаточно надежные, чтобы позволить разрабатывать модульные системы принятия решений.

Глава 4 «Координирование и тактика» содержит ряд рецептов координации действий агентов, превращающих их в целостный организм, например в воинское подразделение, и методов принятия тактических решений на основании графов, таких как точки позиций и карты влияний.

Глава 5 «Органы чувств агентов» описывает различные подходы к моделированию мышления агентов. Для моделирования будут использованы уже известные нам инструменты, такие как коллайдеры и графы.

Глава 6 «Настольные игры с искусственным интеллектом» рассматривает семейство алгоритмов для разработки настольных игр с использованием технологий искусственного интеллекта.

Глава 7 «Механизмы обучения» посвящена области машинного обучения. Она послужит хорошим стартом для изучения и применения методов машинного обучения в играх.

Глава 8 «Разное» описывает применение новых методов и алгоритмов, рассматривавшихся в предыдущих главах, для создания моделей поведения, которые не вписываются ни в одну из перечисленных выше категорий.

Что потребуется для работы с книгой

Примеры, представленные в книге, были протестированы с последней версией Unity (на момент завершения работы над книгой это была версия Unity 5.3.4f1). Однако, приступая к работе над книгой, я использовал Unity 5.1.2, поэтому ее можно считать минимальной рекомендуемой версией для опробования примеров.

Кому адресована эта книга

Эта книга рассчитана на тех, кто уже знаком с Unity и хочет обзавестись инструментами для создания искусственного интеллекта и реализации игровой логики.

Разделы

В этой книге вы найдете несколько часто встречающихся заголовков («Подготовка», «Как это реализовать», «Как это работает», «Дополнительная информация» и «Полезные ссылки»).

Разделы, начинающиеся с этих заголовков, содержат пояснения к реализации рецептов и используются следующим образом:

Подготовка

Объясняет назначение рецепта и описывает настройку программного обеспечения и другие подготовительные действия, необходимые для осуществления рецепта.

Как это реализовать...

Описывает шаги по реализации рецепта.

Как это работает...

Обычно включает подробное объяснение того, что произошло в предыдущем разделе.

Дополнительная информация...

Содержит дополнительную информацию о рецепте для лучшего понимания читателем.

Полезные ссылки

Содержит ссылки на полезные материалы, содержащие информацию, связанную с рецептом.

Соглашения

В этой книге используется несколько разных стилей оформления текста для выделения разных видов информации. Ниже приводятся примеры этих стилей и объясняется их назначение.

Программный код в тексте, имена таблиц баз данных, имена папок, имена файлов, расширения файлов, фиктивные адреса URL, пользовательский ввод и ссылки в Twitter будут выглядеть так: «AgentBehaviour – это шаблонный класс для определения большинства моделей поведения в главе».

Блоки программного кода оформляются так:

```
using UnityEngine;
using System.Collections;
public class Steering
{
    public float angular;
    public Vector3 linear;
    public Steering ()
    {
        angular = 0.0f;
        linear = new Vector3();
    }
}
```

Когда потребуется привлечь ваше внимание к определенному фрагменту кода, соответствующие строки или элементы будут выделены жирным:

```
using UnityEngine;
using System.Collections;

public class Wander : Face
    {
    public float offset;
    public float radius;
    public float rate;
}
```

Так оформляются предупреждения и важные примечания.

Так оформляются советы и рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и если вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу: http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу: dmkpress@gmail.com.

Поддержка пользователей

Покупателям наших книг мы готовы предоставить дополнительные услуги, чтобы ваша покупка принесла вам максимальную пользу.

Загрузка исходного кода примеров

Загрузить файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф в разделе «Читателям – Файлы к книгам».

Загрузка цветных иллюстраций к книге

Вы также можете получить файл в формате PDF с цветными иллюстрациями и диаграммами к этой книге. Цветные изображения помогут лучше понять содержание книги. Загрузить этот файл можно по адресу: http://www.packtpub.com/sites/default/files/downloads/Unity5xGameAIProgrammingCookbook_ColorImages.pdf.

Список опечаток

Хотя мы приняли все возможные меры, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы нашли опечатку, пожалуйста, сообщите о ней главному редактору по адресу: dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в Интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в Интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли принять меры.

Пожалуйста, свяжитесь с нами по адресу электронной почты dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Вопросы

Вы можете присылать любые вопросы, касающиеся данной книги, по адресу dm@dmk-press.ru или questions@packtpub.com. Мы постараемся решить возникшие проблемы.

Глава 1

Интеллектуальные модели поведения: перемещение

В этой главе рассматриваются алгоритмы искусственного интеллекта для перемещения и охватываются следующие рецепты:

- создание шаблона моделей поведения;
- преследование и уклонение;
- достижение цели и уход от погони;
- поворот объектов;
- блуждание вокруг;
- следование по маршруту;
- уклонение от встреч с агентами;
- уклонение от встреч со стенами;
- смешивание моделей поведения с применением весовых коэффициентов;
- смешивание моделей поведения с применением приоритетов;
- сочетание моделей поведения с применением конвейера управления;
- стрельба снарядами;
- прогнозирование эллипса рассеивания снарядов;
- нацеливание снаряда;
- создание системы прыжков.

Введение

В настоящее время Unity является одним из самых популярных игровых движков, а также основным инструментом создания игр для индивидуальных разработчиков, не только благодаря доступной биз-

нес-модели, но и надежности редактора проектов, ежегодным улучшениям и, что самое главное, простоте использования и постоянно растущему сообществу разработчиков по всему миру.

Благодаря тому что движок Unity сам заботится о сложных закулисных процессах (отображение, физические процессы, интеграция и кросс-платформенное развертывание – лишь некоторые из них), разработчики могут сосредоточиться на создании систем искусственного интеллекта, оживляющих игры, обеспечивающих интерактивное взаимодействие в реальном времени.

Цель книги – познакомить вас с инструментами проектирования искусственного интеллекта, чтобы вы могли создавать достойных противников, доводя их до совершенства, и даже разработать собственный движок искусственного интеллекта.

Эта глава начинается с нескольких наиболее интересных алгоритмов, основанных на принципах управления перемещением, разработанных Крейгом Рейнольдсом (Craig Reynolds) совместно с Яном Миллингтоном (Ian Millington). На этих принципах базируется большинство алгоритмов искусственного интеллекта в современных играх наряду с другими алгоритмами управления перемещениями, такими как семейство алгоритмов определения маршрута.

Создание шаблона моделей поведения

Перед созданием моделей поведения необходимо заложить программный фундамент не только для реализации интеллектуального перемещения, но и для модульной системы, позволяющий изменять и добавлять модели поведения. Здесь создаются пользовательские типы данных и базовые классы для большинства алгоритмов в этой главе.

Подготовка

Для начала вспомним порядок выполнения функций обновления:

- `Update`;
- `LateUpdate`.

Также вспомним о возможности управления порядком выполнения сценариев. В нашей модели поведения сценарии выполняются в следующем порядке:

- сценарии агентов;
- сценарии моделей поведения;
- модели поведения или сценарии, зависимые от предыдущих сценариев.

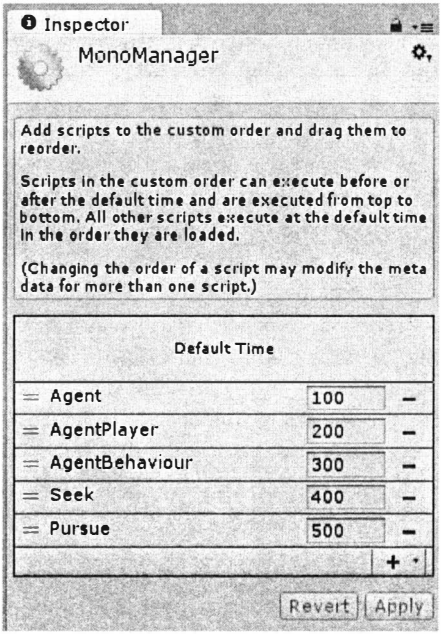


Рис. 1.1 ❖ Пример определения порядка выполнения сценариев перемещения. Сценарий Pursue выполняется после сценария Seek, который выполняется после сценария AgentBehaviour

Как это реализовать...

Нужно создать три класса: `Steering`, `AgentBehaviour` и `Agent`.

1. Класс `Steering` – пользовательский тип данных для хранения значений перемещения и поворота агента:

```
using UnityEngine;
using System.Collections;
public class Steering
{
    public float angular;
    public Vector3 linear;
    public Steering ()
    {
        angular = 0.0f;
        linear = new Vector3();
    }
}
```

2. Создадим класс AgentBehaviour, который станет шаблоном для большей части моделей поведения в этой главе:

```
using UnityEngine;
using System.Collections;
public class AgentBehaviour : MonoBehaviour
{
    public GameObject target;
    protected Agent agent;
    public virtual void Awake ()
    {
        agent = gameObject.GetComponent<Agent>();
    }
    public virtual void Update ()
    {
        agent.SetSteering(GetSteering());
    }
    public virtual Steering GetSteering ()
    {
        return new Steering();
    }
}
```

3. И наконец, класс Agent является основным компонентом, ответственным за реализацию моделей интеллектуального перемещения. Создадим файл с его скелетом:

```
using UnityEngine;
using System.Collections;
public class Agent : MonoBehaviour
{
    public float maxSpeed;
    public float maxAccel;
    public float orientation;
    public float rotation;
    public Vector3 velocity;
    protected Steering steering;
    void Start ()
    {
        velocity = Vector3.zero;
        steering = new Steering();
    }
    public void SetSteering (Steering steering)
    {
        this.steering = steering;
    }
}
```

4. Далее напишем функцию Update, обрабатывающую перемещение в соответствии с текущим значением:

```
public virtual void Update ()
{
    Vector3 displacement = velocity * Time.deltaTime;
    orientation += rotation * Time.deltaTime;
    // необходимо ограничить значение переменной orientation
    // диапазоном (0-360)
    if (orientation < 0.0f)
        orientation += 360.0f;
    else if (orientation > 360.0f)
        orientation -= 360.0f;
    transform.Translate(displacement, Space.World);
    transform.rotation = new Quaternion();
    transform.Rotate(Vector3.up, orientation);
}
```

5. В заключение реализуем функцию LateUpdate, которая подготовит управляющие воздействия для следующего кадра на основании текущего кадра:

```
public virtual void LateUpdate ()
{
    velocity += steering.linear * Time.deltaTime;
    rotation += steering.angular * Time.deltaTime;
    if (velocity.magnitude > maxSpeed)
    {
        velocity.Normalize();
        velocity = velocity * maxSpeed;
    }
    if (steering.angular == 0.0f)
    {
        rotation = 0.0f;
    }
    if (steering.linear.sqrMagnitude == 0.0f)
    {
        velocity = Vector3.zero;
    }
    steering = new Steering();
}
```

Как это работает...

Идея заключается в том, чтобы поместить логику перемещения в функцию GetSteering() модели поведения, которая будет реализо-

вана позже, для упрощения класса агента, предназначенного только для основных расчетов.

Кроме того, это гарантирует установку управляющих значений агента до его использования благодаря настроенному порядку выполнения сценариев и функций в Unity.

Дополнительная информация...

Этот подход на основе компонентов означает, что для обеспечения нужными моделями поведения к объекту класса `GameObject` требуется присоединить сценарий `Agent`.

Полезные ссылки

Дополнительную информацию об игровом цикле в Unity и порядке выполнения функций и сценариев можно найти в официальной документации:

- http://docs.unity3d.com/Manual/ExecutionOrder.html;
- http://docs.unity3d.com/Manual/class-ScriptExecution.html.

Преследование и уклонение

Модели преследования и уклонения отлично подходят для реализации в первую очередь, потому что опираются только на самые основные модели поведения и расширяют их возможностью предсказания следующего шага к цели.

Подготовка

Нам потребуются две базовые модели поведения: `Seek` и `Flee`. Поместим их в порядке выполнения сценариев сразу после класса `Agent`.

Следующий код реализует модель `Seek`:

```
using UnityEngine;
using System.Collections;
public class Seek : AgentBehaviour
{
    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        steering.linear = target.transform.position - transform.position;
        steering.linear.Normalize();
        steering.linear = steering.linear * agent.maxAccel;
```

```
        return steering;
    }
}
```

и модель Flee:

```
using UnityEngine;
using System.Collections;
public class Flee : AgentBehaviour
{
    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        steering.linear = transform.position - target.transform.position;
        steering.linear.Normalize();
        steering.linear = steering.linear * agent.maxAccel;
        return steering;
    }
}
```

Как это реализовать...

Классы Pursue и Evade, по сути, реализуют один и тот же алгоритм, отличаясь только разными базовыми классами, которые они наследуют:

1. Создадим класс Pursue, наследующий класс Seek, и добавим атрибуты для прогнозирования:

```
using UnityEngine;
using System.Collections;

public class Pursue : Seek
{
    public float maxPrediction;
    private GameObject targetAux;
    private Agent targetAgent;
}
```

2. Реализуем функцию Awake для настройки всего, что связано с реальной целью:

```
public override void Awake()
{
    base.Awake();
    targetAgent = target.GetComponent<Agent>();
    targetAux = target;
    target = new GameObject();
}
```

3. А также реализуем функцию OnDestroy для корректной обработки внутреннего объекта:

```
void OnDestroy ()
{
    Destroy(targetAux);
}
```

4. Наконец, реализуем функцию GetSteering:

```
public override Steering GetSteering()
{
    Vector3 direction = targetAux.transform.position -
                        transform. position;
    float distance = direction.magnitude;
    float speed = agent.velocity.magnitude;
    float prediction;
    if (speed <= distance / maxPrediction)
        prediction = maxPrediction;
    else
        prediction = distance / speed;
    target.transform.position = targetAux.transform.position;
    target.transform.position += targetAgent.velocity * prediction;
    return base.GetSteering();
}
```

5. Реализация модели поведения Evade выглядит точно так же, с той лишь разницей, что она наследует класс Flee:

```
public class Evade : Flee

{

    // все то же самое

}
```

Как это работает...

Эти модели поведения базируются на моделях Seek и Flee, но вдобавок учитывают скорость цели, чтобы предсказать, куда двигаться дальше. Координаты целевой точки при этом сохраняются в дополнительном внутреннем объекте.

Достижение цели и уход от погони

В основу этих моделей заложены те же принципы, что и для моделей Seek и Flee, плюс дополнительно определяется точка, в которой агент

может автоматически остановиться после выполнения определенного условия, либо при приближении к пункту назначения (достижении цели), либо при достаточном удалении от опасной точки (уход от погони).

Подготовка

Для алгоритмов Arrive и Leave необходимо создать отдельные файлы и не забыть настроить порядок их выполнения.

Как это реализовать...

При реализации этих моделей поведения используется один и тот же подход, но они содержат разные свойства и производят разные расчеты в начальной части функции GetSteering:

1. Во-первых, модель Arrive должна определять радиус остановки (достижение цели) и радиус замедления скорости:

```
using UnityEngine;
using System.Collections;

public class Arrive : AgentBehaviour
{
    public float targetRadius;
    public float slowRadius;
    public float timeToTarget = 0.1f;
}
```

2. Создадим функцию GetSteering:

```
public override Steering GetSteering()
{
    // дальнейшая реализация описывается ниже
}
```

3. Первая половина функции GetSteering вычисляет скорость в зависимости от расстояния до цели и радиуса замедления:

```
Steering steering = new Steering();
Vector3 direction = target.transform.position - transform.position;
float distance = direction.magnitude; float targetSpeed;
if (distance < targetRadius)
    return steering;
if (distance > slowRadius)
    targetSpeed = agent.maxSpeed;
else
    targetSpeed = agent.maxSpeed * distance / slowRadius;
```

4. Определим вторую часть функции GetSteering, где устанавливаются управляющие значения, а скорость ограничивается максимальным значением:

```
Vector3 desiredVelocity = direction;
desiredVelocity.Normalize();
desiredVelocity *= targetSpeed;
steering.linear = desiredVelocity - agent.velocity;
steering.linear /= timeToTarget;
if (steering.linear.magnitude > agent.maxAccel)
{
    steering.linear.Normalize();
    steering.linear *= agent.maxAccel;
}
return steering;
```

5. Для реализации модели поведения Leave необходимы другие свойства:

```
using UnityEngine;
using System.Collections;

public class Leave : AgentBehaviour
{
    public float escapeRadius;
    public float dangerRadius;
    public float timeToTarget = 0.1f;
}
```

6. Определим первую половину функции GetSteering:

```
Steering steering = new Steering();
Vector3 direction = transform.position - target.transform.position;
float distance = direction.magnitude;
if (distance > dangerRadius)
    return steering;
float reduce;
if (distance < escapeRadius)
    reduce = 0f;
else
    reduce = distance / dangerRadius * agent.maxSpeed;
float targetSpeed = agent.maxSpeed - reduce;
```

7. Вторая половина функции GetSteering в модели Leave в точности повторяет вторую половину функции GetSteering в модели Arrive.

Как это работает...

После вычисления направления все следующие расчеты основываются на двух радиусах, определяющих, когда двигаться с максимальной скоростью, замедлиться и остановиться. С этой целью используется несколько операторов `if`. В модели `Arrive` на большом расстоянии от агента движение выполняется с максимальной скоростью, после вхождения в область, определяемую соответствующим радиусом, движение замедляется, а на очень близком расстоянии до цели – полностью прекращается. Модель `Leave` действует с точностью до наоборот.

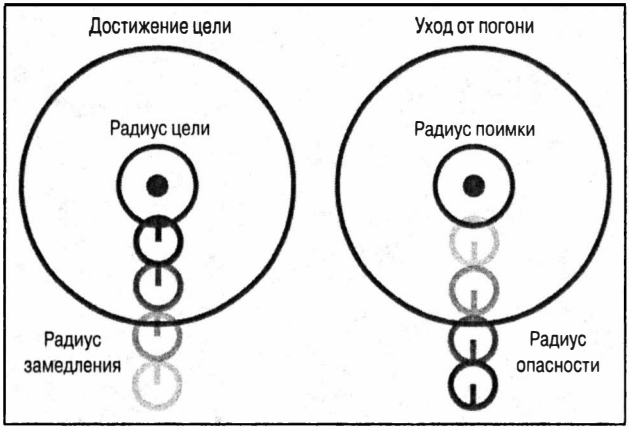


Рис. 1.2 ❖ Визуальное представление работы моделей достижения цели и ухода от погони

Поворот объектов

Прицеливание в реальном мире, так же как в военных тренажерах, выполняется несколько иначе, чем *автоматическое* прицеливание в большинстве игр. Представьте, что требуется реализовать управление башней танка или снайпером. Для этого можно воспользоваться данным рецептом.

Подготовка

Нужно внести некоторые изменения в класс `AgentBehaviour`:

1. Добавим новые свойства ограничений в дополнение к существующим:

```
public float maxSpeed;
public float maxAccel;
public float maxRotation;
public float maxAngularAccel;
```

2. Добавим функцию MapToRange. Она поможет определить направление вращения посредством вычисления разности двух направлений:

```
public float MapToRange (float rotation) {
    rotation %= 360.0f;
    if (Mathf.Abs(rotation) > 180.0f) {
        if (rotation < 0.0f)
            rotation += 360.0f;
        else
            rotation -= 360.0f;
    }
    return rotation;
}
```

3. Также создадим базовую модель поведения Align, являющуюся краеугольным камнем алгоритма поворота. Здесь используется тот же принцип, что и в модели поведения Arrive, но уже к вращению:

```
using UnityEngine;
using System.Collections;

public class Align : AgentBehaviour
{
    public float targetRadius;
    public float slowRadius;
    public float timeToTarget = 0.1f;

    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        float targetOrientation =
           target.GetComponent<Agent>().orientation;
        float rotation = targetOrientation - agent.orientation;
        rotation = MapToRange(rotation);
        float rotationSize = Mathf.Abs(rotation);
        if (rotationSize < targetRadius)
            return steering;
        float targetRotation;
        if (rotationSize > slowRadius)
            targetRotation = agent.maxRotation;
```

```
            else
            targetRotation =
                agent.maxRotation * rotationSize / slowRadius;
            targetRotation *= rotation / rotationSize;
            steering.angular = targetRotation - agent.rotation;
            steering.angular /= timeToTarget;
            float angularAccel = Mathf.Abs(steering.angular);
            if (angularAccel > agent.maxAngularAccel)
            {
                steering.angular /= angularAccel;
                steering.angular *= agent.maxAngularAccel;
            }
            return steering;
        }
    }
```

Как это реализовать...

А теперь приступим к реализации алгоритма поворота, наследующего класс Align:

1. Создадим класс Face с закрытым свойством для ссылки на цель:

```
using UnityEngine;
using System.Collections;

public class Face : Align
{
    protected GameObject targetAux;
}
```

2. Переопределим функцию Awake, в которой настроим ссылки:

```
public override void Awake()
{
    base.Awake();
    targetAux = target;
    target = new GameObject();
    target.AddComponent<Agent>();
}
```

3. Реализуем функцию OnDestroy, в которой освободим ссылки, чтобы избежать проблем с исчерпанием памяти:

```
void OnDestroy ()
{
    Destroy(target);
}
```

4. В заключение реализуем функцию GetSteering:

```
public override Steering GetSteering()
{
    Vector3 direction = targetAux.transform.position -
                        transform.position;
    if (direction.magnitude > 0.0f)
    {
        float targetOrientation = Mathf.Atan2(direction.x, direction.z);
        targetOrientation *= Mathf.Rad2Deg;
        target.GetComponent<Agent>().orientation = targetOrientation;
    }
    return base.GetSteering();
}
```

Как это работает...

Алгоритм вычисляет угол поворота, опираясь на вектор от агента к цели, а затем просто передает управление родительскому классу.

Блуждание вокруг

Этот метод предназначен для моделирования случайных перемещений людей в толпе, животных и практически любых персонажей, не управляемых игроком и осуществляющих хаотичные, случайные перемещения.

Подготовка

В класс AgentBehaviour нужно добавить функцию OriToVec, преобразующую направление в вектор.

```
public Vector3 GetOriAsVec (float orientation) {
    Vector3 vector = Vector3.zero;
    vector.x = Mathf.Sin(orientation*Mathf.Deg2Rad) * 1.0f;
    vector.z = Mathf.Cos(orientation*Mathf.Deg2Rad) * 1.0f;
    return vector.normalized;
}
```

Как это реализовать...

Разобьем реализацию на три этапа: случайный выбор цели, поворот и перемещение к ней:

1. Создадим класс Wander, наследующий класс Face:

```
using UnityEngine;
using System.Collections;

public class Wander : Face
{
    public float offset;
    public float radius;
    public float rate;
}
```

2. Определим функцию Awake, выполняющую настройку на цель:

```
public override void Awake()
{
    target = new GameObject();
    target.transform.position = transform.position;
    base.Awake();
}
```

3. Определим функцию GetSteering:

```
public override Steering GetSteering()
{
    Steering steering = new Steering();
    float wanderOrientation = Random.Range(-1.0f, 1.0f) * rate;
    float targetOrientation = wanderOrientation + agent.orientation;
    Vector3 orientationVec = OriToVec(agent.orientation);
    Vector3 targetPosition = (offset * orientationVec) +
                             transform.position;
    targetPosition = targetPosition +
                     (OriToVec(targetOrientation) * radius);
    targetAux.transform.position = targetPosition;
    steering = base.GetSteering();
    steering.linear = targetAux.transform.position -
                      transform.position;
    steering.linear.Normalize();
    steering.linear *= agent.maxAccel;
    return steering;
}
```

Как это работает...

Модель поведения, учитывая два радиуса, определяет случайную позицию для перехода, осуществляет поворот к ней и преобразует вычисленное направление в вектор.

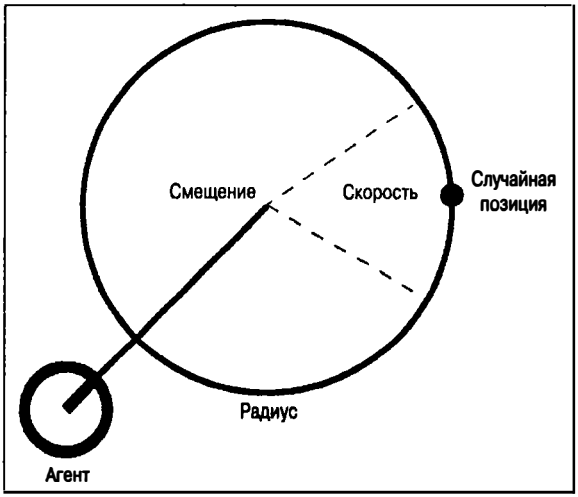


Рис. 1.3 ❖ Визуальное представление определения параметров для модели блуждания

Следование по маршруту

Иногда необходимо реализовать движение по заранее выбранному маршруту, но реализовать это только с помощью кода не всегда просто. Представьте, что вы работаете над игрой в жанре «Стелс»[1]. Станете ли вы писать код, определяющий маршрут для каждого отдельного охранника? Следующий метод обеспечивает гибкое решение для подобных ситуаций:

Подготовка

Нужно определить пользовательский тип данных `PathSegment`:

```
using UnityEngine;
using System.Collections;

public class PathSegment
{
    public Vector3 a;
    public Vector3 b;

    public PathSegment () : this (Vector3.zero, Vector3.zero){}
```

[1] https://ru.wikipedia.org/wiki/Стелс-экшен. – *Прим. ред.*

```
    public PathSegment (Vector3 a, Vector3 b)
    {
        this.a = a;
        this.b = b;
    }
}
```

Как это реализовать...

Разделим этот непростой рецепт на два этапа. На первом этапе определим класс Path, абстрагирующий точки маршрута от конкретных пространственных представлений, а затем реализуем модель PathFollower, использующую эту абстракцию для следования к точкам в пространстве:

1. Определим класс Path с узлами и сегментами, из которых только узлы являются общедоступными и должны задаваться вручную:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Path : MonoBehaviour
{
    public List<GameObject> nodes;
    List<PathSegment> segments;
}
```

2. Определим функцию Start для подготовки сегментов при запуске сцены:

```
void Start()
{
    segments = GetSegments();
}
```

3. Определим функцию GetSegments для создания сегментов из узлов:

```
public List<PathSegment> GetSegments ()
{
    List<PathSegment> segments = new List<PathSegment>();
    int i;
    for (i = 0; i < nodes.Count - 1; i++)
    {
        Vector3 src = nodes[i].transform.position;
        Vector3 dst = nodes[i+1].transform.position;
```

```
            PathSegment segment = new PathSegment(src, dst);
            segments.Add(segment);
        }
        return segments;
    }
```

4. Определим первую функцию для абстрагирования GetParam:

```
public float GetParam(Vector3 position, float lastParam)
{
    // тело функции
}
```

5. Она должна найти ближайший к агенту сегмент:

```
float param = 0f;
PathSegment currentSegment = null;
float tempParam = 0f;
foreach (PathSegment ps in segments)
{
    tempParam += Vector3.Distance(ps.a, ps.b);
    if (lastParam <= tempParam)
    {
        currentSegment = ps;
        break;
    }
}
if (currentSegment == null)
    return 0f;
```

6. Вычислить направление движения из текущей позиции:

```
Vector3 currPos = position - currentSegment.a;
Vector3 segmentDirection = currentSegment.b - currentSegment.a;
segmentDirection.Normalize();
```

7. Найти точку в сегменте с помощью проекции вектора:

```
Vector3 pointInSegment = Vector3.Project(currPos, segmentDirection);
```

8. И наконец, вернуть следующую позицию на линии маршрута:

```
param = tempParam - Vector3.Distance(currentSegment.a,
                                     currentSegment.b);
param += pointInSegment.magnitude;
return param;
```

9. Определим функцию GetPosition:

```
public Vector3 GetPosition(float param)
{
    // тело функции
}
```

10. По текущему местоположению находим соответствующий сегмент:

```
Vector3 position = Vector3.zero;
PathSegment currentSegment = null;
float tempParam = 0f;
foreach (PathSegment ps in segments)
{
    tempParam += Vector3.Distance(ps.a, ps.b);
    if (param <= tempParam)
    {
        currentSegment = ps;
        break;
    }
}
if (currentSegment == null)
    return Vector3.zero;
```

11. Преобразуем параметр в точку пространства и возвращаем ее:

```
Vector3 segmentDirection = currentSegment.b - currentSegment.a;
segmentDirection.Normalize();
tempParam -= Vector3.Distance(currentSegment.a, currentSegment.b);
tempParam = param - tempParam;
position = currentSegment.a + segmentDirection * tempParam;
return position;
```

12. Создадим модель PathFollower, наследующую Seek (не забываем о настройке порядка выполнения):

```
using UnityEngine;
using System.Collections;

public class PathFollower : Seek
{
    public Path path;
    public float pathOffset = 0.0f;
    float currentParam;
}
```

13. Реализуем функцию Awake для определения цели:

```
public override void Awake()
{
```

```
        base.Awake();
        target = new GameObject();
        currentParam = 0f;
    }
```

14. На заключительном этапе определим функцию GetSteering, которая использует абстракцию, созданную классом Path, для определения позиции цели и применяет модель Seek:

```
public override Steering GetSteering()
{
    currentParam = path.GetParam(transform.position, currentParam);
    float targetParam = currentParam + pathOffset;
    target.transform.position = path.GetPosition(targetParam);
    return base.GetSteering();
}
```

Как это работает...

Управление перемещением по заданному маршруту осуществляется классом Path. Он очень важен, поскольку посредством своего метода GetParam отображает смещение точки во внутреннее представление, а с помощью метода GetPosition преобразует внутреннее представление в позицию в трехмерном пространстве.

С помощью функций класса Path алгоритм следования по маршруту получает новую позицию, устанавливает цель и применяет модель преследования Seek.

Дополнительная информация...

Для правильной работы алгоритма необходимо, чтобы узлы были связаны в инспекторе в определенном порядке. Практически это означает, что узлам следует вручную дать имена с соответствующими номерами.

Кроме того, можно определить функцию OnDrawGizmos для визуализации маршрута:

```
void OnDrawGizmos ()
{
    Vector3 direction;
    Color tmp = Gizmos.color;
    Gizmos.color = Color.magenta;//например малиновый
    int i;
    for (i = 0; i < nodes.Count - 1; i++)
    {
```

```
        Vector3 src = nodes[i].transform.position;
        Vector3 dst = nodes[i+1].transform.position;
        direction = dst - src;
        Gizmos.DrawRay(src, direction);
    }
    Gizmos.color = tmp;
}
```

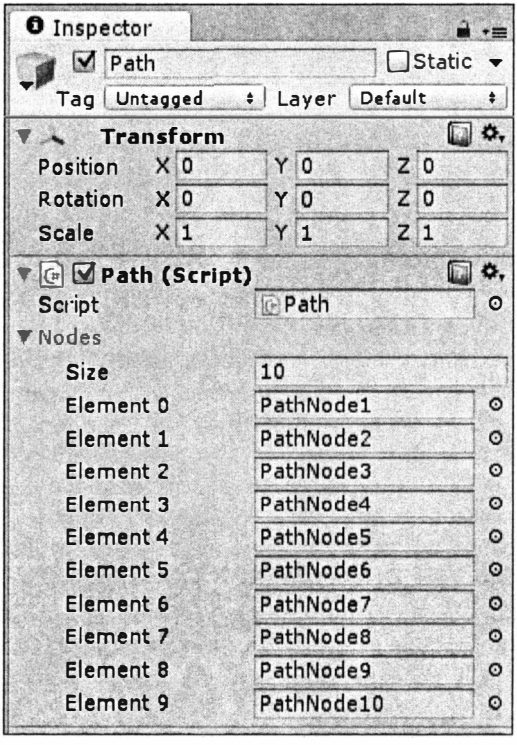


Рис. 1.4 ❖ Пример определения пути в окне инспектора

Уклонение от встреч с агентами

При моделировании толпы в играх поведение агентов, аналогичное поведению физических частиц, выглядит неестественно. Цель этого рецепта – реализовать уклонение агента от столкновений с себе подобными во время движения.

Подготовка

Необходимо создать тег **Agent** и присвоить его объектам, столкновений с которыми требуется избегать, а также присоединить к ним компонент сценария **Agent**.

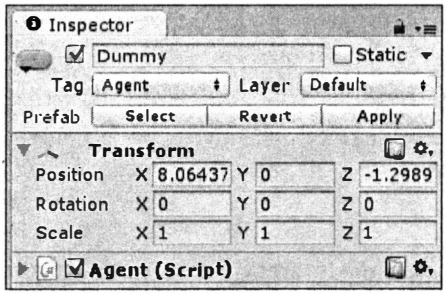


Рис. 1.5 ❖ Окно инспектора с параметрами агента, столкновений с которым следует избегать

Как это реализовать...

Этот рецепт требует создать и обрабатывать единственный файл:

1. Создадим модель `AvoidAgent` с радиусом, определяющим область столкновения, и список агентов для уклонения:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AvoidAgent : AgentBehaviour
{
    public float collisionRadius = 0.4f;
    GameObject[] targets;
}
```

2. Реализуем функцию `Start`, заполняющую список агентов по тегу, созданному ранее:

```
void Start ()
{
    targets = GameObject.FindGameObjectsWithTag("Agent");
}
```

3. Определим функцию `GetSteering`:

```
public override Steering GetSteering()
{
    // тело функции
}
```

4. Добавим переменные для вычисления расстояния и скорости агентов, находящихся поблизости:

```
Steering steering = new Steering();
float shortestTime = Mathf.Infinity;
GameObject firstTarget = null;
float firstMinSeparation = 0.0f;
float firstDistance = 0.0f;
Vector3 firstRelativePos = Vector3.zero;
Vector3 firstRelativeVel = Vector3.zero;
```

5. Найдем ближайшего агента, с которым может столкнуться текущий:

```
foreach (GameObject t in targets)
{
    Vector3 relativePos;
    Agent targetAgent = t.GetComponent<Agent>();
    relativePos = t.transform.position - transform.position;
    Vector3 relativeVel = targetAgent.velocity - agent.velocity;
    float relativeSpeed = relativeVel.magnitude;
    float timeToCollision = Vector3.Dot(relativePos, relativeVel);
    timeToCollision /= relativeSpeed * relativeSpeed * -1;
    float distance = relativePos.magnitude;
    float minSeparation = distance - relativeSpeed * timeToCollision;
    if (minSeparation > 2 * collisionRadius)
        continue;
    if (timeToCollision > 0.0f && timeToCollision < shortestTime)
    {
        shortestTime = timeToCollision;
        firstTarget = t;
        firstMinSeparation = minSeparation;
        firstRelativePos = relativePos;
        firstRelativeVel = relativeVel;
    }
}
```

6. Если такой агент имеется, попытаемся обойти его:

```
if (firstTarget == null)
    return steering;
```

```
if (firstMinSeparation <= 0.0f || firstDistance < 2 * collisionRadius)
    firstRelativePos = firstTarget.transform.position;
else
    firstRelativePos += firstRelativeVel * shortestTime;
firstRelativePos.Normalize();
steering.linear = -firstRelativePos * agent.maxAccel;
return steering;
```

Как это работает...

Из списка агентов извлекается ближайший, и, если он находится достаточно близко, текущий агент покидает маршрут, чтобы избежать столкновения, при этом учитывается текущая скорость встречного агента.

Дополнительная информация...

Данную модель можно использовать в комплексе с другими моделями, применив приемы смешивания (рассматриваются ниже в этой главе), или использовать как отправную точку для реализации ваших собственных алгоритмов предотвращения столкновений.

Уклонение от стен

Цель этого рецепта – реализовать модель уклонения от столкновений со стенами, которая учитывает безопасное расстояние до них и обеспечивает удаление от их поверхностей при чрезмерном приближении.

Подготовка

Здесь используются структура `RaycastHit` и функция `Raycast` физического движка, поэтому обращайтесь к его документации, если эта тема вам недостаточно хорошо знакома.

Как это реализовать...

Благодаря проделанной ранее работе этот рецепт получился простым и коротким:

1. Создадим модель `AvoidWall`, наследующую `Seek`:

```
using UnityEngine;
using System.Collections;

public class AvoidWall : Seek
{
    // тело функции
}
```

2. Добавим свойства для определения безопасного удаления и длины отбрасываемого луча:

```
public float avoidDistance;
public float lookAhead;
```

3. Определим функцию Awake для настройки целевого объекта:

```
public override void Awake()
{
    base.Awake();
    target = new GameObject();
}
```

4. Определим функцию GetSteering:

```
public override Steering GetSteering()
{
    // тело функции
}
```

5. Внутри функции объявим и инициализируем переменные, необходимые для отбрасывания луча:

```
Steering steering = new Steering();
Vector3 position = transform.position;
Vector3 rayVector = agent.velocity.normalized * lookAhead;
Vector3 direction = rayVector;
RaycastHit hit;
```

6. Отбросим луч и выполним необходимые расчеты, если он пересекается со стеной:

```
if (Physics.Raycast(position, direction, out hit, lookAhead))
{
    position = hit.point + hit.normal * avoidDistance;
    target.transform.position = position;
    steering = base.GetSteering();
}
return steering;
```

Как это работает...

По направлению движения агента отбрасывается луч. Если луч пересекается со стеной, целевой объект перемещается в новую позицию, с учетом расстояния до стены и заданного безопасного удаления, при этом выполнение самого перемещения передается модели Seek. В результате создается видимость уклонения от стен.

Дополнительная информация...

Модель поведения можно улучшить, добавив дополнительные лучи и расположив их в виде усов. Кроме того, эта модель поведения обычно используется вместе с другими моделями, такими как `Pursue`, путем смешивания.

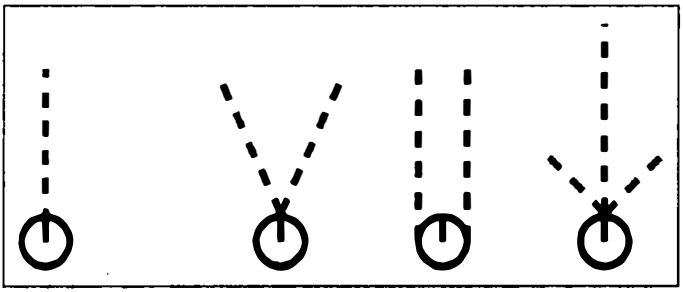


Рис. 1.6 ❖ Оригинальное решение с отбрасыванием луча и возможные усовершенствования

Полезные ссылки

Более подробную информацию о структуре `RaycastHit` и функции `Raycast` можно найти в официальной электронной документации на страницах:

- http://docs.unity3d.com/ScriptReference/RaycastHit.html;
- http://docs.unity3d.com/ScriptReference/Physics.Raycast.html.

Смешивание моделей поведения с весовыми коэффициентами

Приемы смешивания позволяют объединять и смешивать модели поведения без создания новых сценариев, если необходимо придать агенту гибридное поведение.

Это один из самых мощных приемов в этой главе и чаще других используется для смешивания моделей поведения благодаря его возможностям и простоте реализации.

Подготовка

Прежде всего необходимо добавить в класс `AgentBehaviour` свойство `weight` со значением весового коэффициента и, в данном случае, при-

своить ему значение 1.0f. Кроме того, требуется изменить функцию Update, чтобы она передавала свойство weight в функцию SetSteering класса Agent. Таким образом, новый класс AgentBehaviour должен выглядеть примерно так:

```
public class AgentBehaviour : MonoBehaviour
{
    public float weight = 1.0f;

    // ... остальная часть определения класса

    public virtual void Update ()
    {
        agent.SetSteering(GetSteering(), weight);
    }
}
```

Как это реализовать...

Затем нужно изменить сигнатуру и ее определение функции агента SetSteering:

```
public void SetSteering (Steering steering, float weight)
{
    this.steering.linear += (weight * steering.linear);
    this.steering.angular += (weight * steering.angular);
}
```

Как это работает...

Весовые коэффициенты используются для усиления управляющего воздействия steering перед добавлением в основную структуру управления.

Дополнительная информация...

Сумма весовых коэффициентов не должна превышать 1.0f. Параметр weight является ссылкой и определяет значимость управления steering относительно прочих.

Полезные ссылки

В этом проекте имеется пример уклонения от столкновения со стенами, в котором используется смешивание с учетом весовых коэффициентов.

Смешивание моделей поведения по приоритету

Иногда смешивание с учетом весовых коэффициентов не дает нужного результата, поскольку тяжеловесные модели поведения нивелируют влияние легковесных, в то время как эти модели поведения также должны вносить свой вклад в результат. В этом случае применяется смешивание по приоритетам, обеспечивающее каскадный эффект, от высоко- до низкоприоритетных моделей.

Подготовка

Этот подход очень похож на используемый в предыдущем рецепте. Нужно добавить новое свойство в класс `AgentBehaviour` и изменить функцию `Update`, организовав передачу свойства приоритета `priority` в функцию `SetSteering` класса `Agent`. Новый класс `AgentBehaviour` должен выглядеть примерно так:

```
public class AgentBehaviour : MonoBehaviour
{
    public int priority = 1;
    // ... остальная часть определения класса
    public virtual void Update ()
    {
        agent.SetSteering(GetSteering(), priority);
    }
}
```

Как это реализовать...

А теперь внесем изменения в класс `Agent`:

1. Добавим пространство имен еще одной библиотеки:

   ```
   using System.Collections.Generic;
   ```

2. Добавим свойство минимального значения управляющего воздействия для группы моделей поведения:

   ```
   public float priorityThreshold = 0.2f;
   ```

3. Добавим свойство для хранения управляющих воздействий группы моделей:

   ```
   private Dictionary<int, List<Steering>> groups;
   ```

4. Инициализируем переменную в функции Start:

```
groups = new Dictionary<int, List<Steering>>();
```

5. Изменим функцию LateUpdate так, чтобы она присваивала переменной управления значение, возвращаемое функцией GetPrioritySteering:

```
public virtual void LateUpdate ()
{
    // определить управляющее воздействие с учетом приоритетов
    steering = GetPrioritySteering();
    groups.Clear();
    // ... остальные вычисления не изменились
    steering = new Steering();
}
```

6. Изменим сигнатуру и определение функции SetSteering, добавив сохранение значений управления в группах с соответствующими приоритетами:

```
public void SetSteering (Steering steering, int priority)
{
    if (!groups.ContainsKey(priority))
    {
        groups.Add(priority, new List<Steering>());
    }
    groups[priority].Add(steering);
}
```

7. И наконец, реализуем функцию GetPrioritySteering для вычисления группового управляющего воздействия:

```
private Steering GetPrioritySteering ()
{
    Steering steering = new Steering();
    float sqrThreshold = priorityThreshold * priorityThreshold;
    foreach (List<Steering> group in groups.Values)
    {
        steering = new Steering();
        foreach (Steering singleSteering in group)
        {
            steering.linear += singleSteering.linear;
            steering.angular += singleSteering.angular;
        }
        if (steering.linear.sqrMagnitude > sqrThreshold ||
```

```
                Mathf.Abs(steering.angular) > priorityThreshold)
            {
                return steering;
            }
        }
```

Как это работает...

При создании групп по приоритетам смешиваются совместимые модели поведения, а затем выбирается первая группа, в которой величина управляющего воздействия превышает пороговое значение. В противном случае выбирается значение из группы с наименьшим приоритетом.

Дополнительная информация...

Этот подход можно расширить, объединив с весовыми коэффициентами, и получить более надежную архитектуру с повышенной точностью влияния моделей поведения на каждом уровне приоритета:

```
foreach (Steering singleSteering in group)
{
    steering.linear += singleSteering.linear * weight;
    steering.angular += singleSteering.angular * weight;
}
```

Полезные ссылки

В этом проекте имеется пример уклонения от стен, в котором используется смешивание на основе приоритетов.

Комбинирование моделей поведения с применением конвейера управления

Это еще один подход к смешиванию моделей поведения, основанный на целях. Он является золотой серединой между смешением перемещений и планированием без реализации последнего.

Подготовка

При использовании конвейера изменяется сам подход к управлению. Здесь в центре внимания находятся цели и ограничения. То есть вся тяжесть осуществления ложится на базовые и производные от них классы, определяющие модель поведения. Начнем с их реализации.

Ниже приводится определение класса Targeter. Его можно рассматривать как модель поведения, ориентированную на конкретную цель:

```
using UnityEngine;
using System.Collections;

public class Targeter : MonoBehaviour
{
    public virtual Goal GetGoal()
    {
        return new Goal();
    }
}
```

А теперь определим класс Decomposer:

```
using UnityEngine;
using System.Collections;

public class Decomposer : MonoBehaviour
{
    public virtual Goal Decompose (Goal goal)
    {
        return goal;
    }
}
```

класс Constraint:

```
using UnityEngine;
using System.Collections;

public class Constraint : MonoBehaviour
{
    public virtual bool WillViolate (Path path)
    {
        return true;
    }

    public virtual Goal Suggest (Path path) {
        return new Goal();
    }
}
```

и, наконец, класс Actuator:

```
using UnityEngine;
using System.Collections;
```

```
public class Actuator : MonoBehaviour
{
    public virtual Path GetPath (Goal goal)
    {
        return new Path();
    }

    public virtual Steering GetOutput (Path path, Goal goal)
    {
        return new Steering();
    }
}
```

Как это реализовать...

Класс SteeringPipeline использует реализованные ранее классы, обеспечивая управляющий компонентами конвейер, но применяет, как отмечалось выше, совершенно иной подход:

1. Создадим класс SteeringPipeline, наследующий модель поведения Wander и включающий в себя массив компонентов для обработки:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SteeringPipeline : Wander
{
    public int constraintSteps = 3;
    Targeter[] targeters;
    Decomposer[] decomposers;
    Constraint[] constraints;
    Actuator actuator;
}
```

2. Определим функцию Start, инициализирующую ссылки на компоненты, присоединенные к игровому объекту:

```
void Start ()
{
    targeters = GetComponents<Targeter>();
    decomposers = GetComponents<Decomposer>();
    constraints = GetComponents<Constraint>();
    actuator = GetComponent<Actuator>();
}
```

3. Определим функцию GetSteering для работы с целью и управляющим воздействием для ее достижения:

```
public override Steering GetSteering()
{
    Goal goal = new Goal();
    foreach (Targeter targeter in targeters)
        goal.UpdateChannels(targeter.GetGoal());
    foreach (Decomposer decomposer in decomposers)
        goal = decomposer.Decompose(goal);
    for (int i = 0; i < constraintSteps; i++)
    {
        Path path = actuator.GetPath(goal);
        foreach (Constraint constraint in constraints)
        {
            if (constraint.WillViolate(path))
            {
                goal = constraint.Suggest(path);
                break;
            }
            return actuator.GetOutput(path, goal);
        }
    }
    return base.GetSteering();
}
```

Как это работает...

Этот код конструирует составную цель из списка целей *targeters*, создает дочерние цели с помощью деструкторов из списка *decomposers* и перед включением их в конечную цель проверяет соответствие ограничениям из списка *constraints*. Если ни одна из целей не прошла проверку, используется модель поведения по умолчанию `Wander`.

Дополнительная информация...

Попробуйте реализовать несколько рецептов поведения, основанных на целях, деструкторах, ограничениях и исполнителе. Имейте в виду, что исполнитель (экземпляр класса `Actuator`) может быть только один, и только он отвечает за принятие окончательного решения. Например:

- **цели**: поиск, прибытие, поворот и согласование скорости;
- **деструкторы**: алгоритмы поиска маршрута;
- **ограничения**: уклонение от стен/агентов.

Полезные ссылки

Теоретические знания по этой теме можно почерпнуть из книги Яна Миллингтона (Ian Millington) «*Artificial Intelligence for Games*».

Стрельба снарядами

Этот рецепт очень важен для сценариев, управляющих объектами, на которые воздействует сила тяжести, такими как ядра или гранаты, поскольку дает возможность предсказать место падения снаряда или точно послать снаряд к заданной цели.

Подготовка

Этот рецепт несколько отличается от прочих, поскольку в нем не применяется базовый класс `AgentBehaviour`.

Как это реализовать...

1. Создадим класс снаряда `Projectile` со свойствами, которые обрабатываются физической системой:

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour
{
    private bool set = false;
    private Vector3 firePos;
    private Vector3 direction;
    private float speed;
    private float timeElapsed;
}
```

2. Определим функцию `Update`:

```
void Update ()
{
    if (!set)
        return;
    timeElapsed += Time.deltaTime;
    transform.position = firePos + direction * speed * timeElapsed;
    transform.position += Physics.gravity *
        (timeElapsed * timeElapsed) / 2.0f;
    // дополнительные проверки для очистки сцены
    if (transform.position.y < -1.0f)
        Destroy(this.gameObject);// или set = false; и скрыть его
}
```

3. И наконец, добавим функцию `Set`, реализующую метание игрового объекта (например, после инициализации сцены):

```
public void Set (Vector3 firePos, Vector3 direction, float speed)
{
    this.firePos = firePos;
    this.direction = direction.normalized;
    this.speed = speed;
    transform.position = firePos;
    set = true;
}
```

Как это работает...

Эта модель вычисляет параболлическую траекторию движения, опираясь на законы физики, изучаемые в средней школе.

Дополнительная информация...

Можно было выбрать и другой способ: реализовать общедоступные свойства в сценарии или объявить переменные как общедоступные и вместо вызова функции `Set` поместить неактивный по умолчанию сценарий в шаблонный объект (prefab) и активировать его после установки свойств. Это упростило бы применение шаблона «Пул объектов».

Полезные ссылки

За дополнительной информацией о шаблоне «Пул объектов» обращайтесь к статье в Википедии и официальному видеоуроку от Unity Technologies:

- https://ru.wikipedia.org/wiki/Объектный_пул;
- http://unity3d.com/learn/tutorials/modules/beginner/live-training-archive/object-pooling.

Прогнозирование места падания снаряда

После вылета снаряда некоторые агенты должны убегать от него, если речь идет о гранате, или наблюдать за его полетом, когда разрабатывается спортивная игра. В любом случае важно уметь предсказывать место падения снаряда для принятия дальнейшего решения.

Подготовка

Прежде чем пытаться определить место падения, важно узнать время, оставшееся до падения (или до достижения определенной позиции). Поэтому вместо создания новой модели поведения внесем изменения в класс снаряда `Projectile`.

Как это реализовать...

1. Сначала добавим функцию GetLandingTime для вычисления времени падения:

```
public float GetLandingTime (float height = 0.0f)
{
    Vector3 position = transform.position;
    float time = 0.0f;
    float valueInt = (direction.y * direction.y) * (speed * speed);
    valueInt = valueInt - (Physics.gravity.y * 2 *
                           (position.y - height));
    valueInt = Mathf.Sqrt(valueInt);
    float valueAdd = (-direction.y) * speed;
    float valueSub = (-direction.y) * speed;
    valueAdd = (valueAdd + valueInt) / Physics.gravity.y;
    valueSub = (valueSub - valueInt) / Physics.gravity.y;
    if (float.IsNaN(valueAdd) && !float.IsNaN(valueSub))
        return valueSub;
    else if (!float.IsNaN(valueAdd) && float.IsNaN(valueSub))
        return valueAdd;
    else if (float.IsNaN(valueAdd) && float.IsNaN(valueSub))
        return -1.0f;
    time = Mathf.Max(valueAdd, valueSub);
    return time;
}
```

2. Затем добавим функцию GetLandingPos для вычисления предположительного места падения:

```
public Vector3 GetLandingPos (float height = 0.0f)
{
    Vector3 landingPos = Vector3.zero;
    float time = GetLandingTime();
    if (time < 0.0f)
        return landingPos;
    landingPos.y = height;
    landingPos.x = firePos.x + direction.x * speed * time;
    landingPos.z = firePos.z + direction.z * speed * time;
    return landingPos;
}
```

Как это работает...

Здесь решается уравнение из предыдущего рецепта для фиксированной высоты. Зная текущую позицию снаряда и его скорость, можно получить время, за которое снаряд достигнет заданной высоты.

Дополнительная информация...

Что касается проверок на значение NaN, они нужны потому, что уравнение может иметь два, одно или ни одного решения. Кроме того, когда время до падения меньше нуля, это означает, что снаряд не сможет достичь целевой высоты.

Нацеливание снаряда

Для агентов важно не только уметь прогнозировать место падения снаряда, но и направлять их. Будет весело, если агенты, действующие как игроки в регби, не смогут передать мяч.

Подготовка

По аналогии с предыдущим рецептом внесем изменения только в класс снаряда Projectile.

Как это реализовать...

Благодаря проделанной ранее работе этот рецепт получился простым и коротким:

1. Определим функцию GetFireDirection:

```
public static Vector3 GetFireDirection (Vector3 startPos,
    Vector3 endPos, float speed)
{
    // тело функции
}
```

2. Решим соответствующее квадратное уравнение:

```
Vector3 direction = Vector3.zero;
Vector3 delta = endPos - startPos;
float a = Vector3.Dot(Physics.gravity, Physics.gravity);
float b = -4 * (Vector3.Dot(Physics.gravity, delta) + speed * speed);
float c = 4 * Vector3.Dot(delta, delta);
if (4 * a * c > b * b)
    return direction;
float time0 = Mathf.Sqrt((-b + Mathf.Sqrt(b * b - 4 * a * c)) / (2*a));
float time1 = Mathf.Sqrt((-b - Mathf.Sqrt(b * b - 4 * a * c)) / (2*a));
```

3. Если можно послать снаряд в соответствии с заданными параметрами, вернуть ненулевой вектор направления:

```
float time;
if (time0 < 0.0f)
```

```
{
    if (time1 < 0)
        return direction;
    time = time1;
}
else
{
    if (time1 < 0)
        time = time0;
    else
        time = Mathf.Min(time0, time1);
}
direction = 2 * delta - Physics.gravity * (time * time);
direction = direction / (2 * speed * time);
return direction;
```

Как это работает...

Подставляя заданную фиксированную скорость, мы решаем соответствующее квадратное уравнение и получаем направление (при наличии хотя бы одного значения времени), которое не нуждается в нормализации, поскольку уже было нормализовано при настройке снаряда.

Дополнительная информация...

Обратите внимание, что при отрицательном значении времени возвращается *пустое* направление. Это означает, что заданное значение скорости недостаточно велико. Чтобы решить эту проблему, можно, например, определить функцию, которая проверит различные значения скорости до выстрела снаряда.

Еще одно улучшение заключается в добавлении дополнительного параметра типа bool для случаев, когда имеются два допустимых значения времени (что означает две возможные дуги) и требуется послать снаряд над препятствием, например над стеной:

```
if (isWall)
    time = Mathf.Max(time0, time1);
else
    time = Mathf.Min(time0, time1);
```

Создание системы прыжков

Представьте, что в разрабатываемой игре убегающий игрок способен перепрыгивать через скалы или крыши домов. В этом случае про-

тивникам нужно иметь возможность преследовать игрока и обладать достаточным умом, чтобы прыгать и рассчитывать параметры своего прыжка.

Подготовка

Определим базовый алгоритм выбора скорости и введем понятие площадок для прыжка и приземления, чтобы вычислить скорость, необходимую для перемещения между ними.

Кроме того, агенты должны иметь тег `Agent`, а главный объект должен иметь компонент `Collider`, настроенный как триггер. В зависимости от особенностей игры к агенту или к площадкам потребуется присоединить компонент `Rigidbody`.

Следующий код реализует модель поведения `VelocityMatch`:

```
using UnityEngine;
using System.Collections;

public class VelocityMatch : AgentBehaviour {

    public float timeToTarget = 0.1f;

    public override Steering GetSteering()
    {
        Steering steering = new Steering();
        steering.linear = target.GetComponent<Agent>().velocity -
                          agent.velocity;
        steering.linear /= timeToTarget;
        if (steering.linear.magnitude > agent.maxAccel)
            steering.linear = steering.linear.normalized * agent.maxAccel;

        steering.angular = 0.0f;
        return steering;
    }
}
```

Нам потребуется также тип данных `JumpPoint`:

```
using UnityEngine;

public class JumpPoint
{
    public Vector3 jumpLocation; public Vector3 landingLocation;
    // Расстояние от точки прыжка до точки приземления
    public Vector3 deltaPosition;

    public JumpPoint ()
    : this (Vector3.zero, Vector3.zero)
```

```
    {
    }

    public JumpPoint(Vector3 a, Vector3 b)
    {
        this.jumpLocation = a;
        this.landingLocation = b;
        this.deltaPosition = this.landingLocation - this.jumpLocation;
    }
}
```

Как это реализовать...

Рассмотрим реализацию модели прыжка Jump:

1. Определим класс прыжка Jump, наследующий класс VelocityMatch и добавляющий свои свойства:

```
using UnityEngine;
using System.Collections.Generic;

public class Jump : VelocityMatch
{
    public JumpPoint jumpPoint;
    //Признак допустимости прыжка
    bool canAchieve = false;
    //Максимальная вертикальная скорость при прыжке
    public float maxYVelocity;
    public Vector3 gravity = new Vector3(0, -9.8f, 0);
    private Projectile projectile;
    private List<AgentBehaviour> behaviours;

    // дальнейшая реализация описывается ниже
}
```

2. Реализуем метод Isolate. Он отключает все модели поведения агента, за исключением компонента Jump:

```
public void Isolate(bool state)
{
    foreach (AgentBehaviour b in behaviours)
        b.enabled = !state;
    this.enabled = state;
}
```

3. Определим функцию, реализующую прыжок с использованием модели снаряда, рассмотренной ранее:

```
public void DoJump()
{
```

```
    projectile.enabled = true;
    Vector3 direction;
    direction = Projectile.GetFireDirection(jumpPoint.jumpLocation,
                jumpPoint.landingLocation, agent.maxSpeed);
    projectile.Set(jumpPoint.jumpLocation, direction,
                   agent.maxSpeed, false);
}
```

4. Реализуем метод для выбора скорости в зависимости от цели:

```
protected void CalculateTarget()
{
    target = new GameObject();
    target.AddComponent<Agent>();

    //Вычислить время прыжка
    float sqrtTerm = Mathf.Sqrt(2f * gravity.y *
        jumpPoint.deltaPosition.y + maxYVelocity * agent.maxSpeed);
    float time = (maxYVelocity - sqrtTerm) / gravity.y;
    //Проверить допустимость и при необходимости вычислить другое время
    if (!CheckJumpTime(time))
    {
        time = (maxYVelocity + sqrtTerm) / gravity.y;
    }
}
```

5. Реализуем функцию вычисления времени:

```
//Внутренний вспомогательный метод для функции CalculateTarget private
bool CheckJumpTime(float time)
{
    //Вычислить горизонтальную скорость
    float vx = jumpPoint.deltaPosition.x / time;
    float vz = jumpPoint.deltaPosition.z / time;
    float speedSq = vx * vx + vz * vz;

    //Проверить полученное решение на допустимость
    if (speedSq < agent.maxSpeed * agent.maxSpeed)
    {
        target.GetComponent<Agent>().velocity = new Vector3(vx, 0f, vz);
        canAchieve = true;
        return true;
    }
    return false;
}
```

6. Переопределим метод Awake. Самым важным здесь является кэширование ссылок на другие присоединенные модели поведения, что придает смысл функции Isolate:

```
public override void Awake()
{
    base.Awake();
    this.enabled = false;
    projectile = gameObject.AddComponent<Projectile>();
    behaviours = new List<AgentBehaviour>();
    AgentBehaviour[] abs;
    abs = gameObject.GetComponents<AgentBehaviour>();
    foreach (AgentBehaviour b in abs)
    {
        if (b == this)
            continue;
        behaviours.Add(b);
    }
}
```

7. Переопределим метод GetSteering:

```
public override Steering GetSteering()
{
    Steering steering = new Steering();

    // Проверить наличие траектории и, если она отсутствует,
    // создать ее.
    if (jumpPoint != null && target == null)
    {
        CalculateTarget();
    }
    //Проверить равенство траектории нулю.
    //Если нет, ускорение не требуется.
    if (!canAchieve)
    {
        return steering;
    }

    //Проверить попадание в точку прыжка
    if (Mathf.Approximately((transform.position -
            target.transform.position).magnitude, 0f) &&
        Mathf.Approximately((agent.velocity -
            target.GetComponent<Agent>().velocity).magnitude, 0f))
    {
```

```
            DoJump();
            return steering;
        }
        return base.GetSteering();
    }
```

Как это работает...

Алгоритм, основываясь на скорости агента, определяет, сможет ли агент допрыгнуть до площадки приземления. Задаваемая цель определяет, будет ли выполнен прыжок, и если агент способен его совершить, будет сделана попытка подбора вертикальной скорости для приземления на целевую площадку.

Что дальше

Чтобы получить законченную систему прыжков, необходимы площадки для прыжка и приземления. Площадки должны иметь компоненты `Collider`, настроенные как триггеры. Кроме того, как упоминалось ранее, к площадкам может потребоваться присоединить компонент `Rigidbody`, как показано на рис. 1.7.

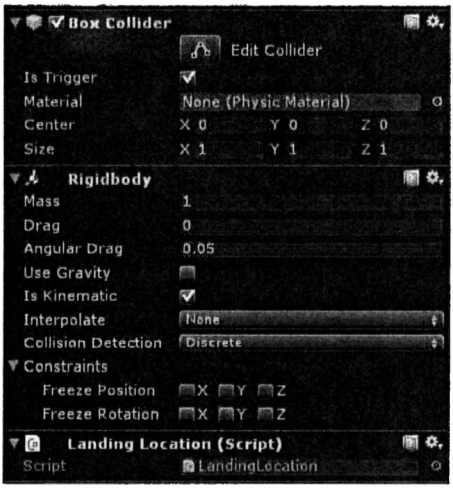


Рис. 1.7

Как поясняется ниже, к площадкам нужно также присоединить сценарии `MonoBehaviour`.

Следующий сценарий нужно присоединить к площадке прыжка:

```
using UnityEngine;

public class JumpLocation : MonoBehaviour
{
    public LandingLocation landingLocation;

    public void OnTriggerEnter(Collider other)
    {
        if (!other.gameObject.CompareTag("Agent"))
            return;
    Agent agent = other.GetComponent<Agent>();
    Jump jump = other.GetComponent<Jump>();
    if (agent == null || jump == null)
        return;
    Vector3 originPos = transform.position;
    Vector3 targetPos = landingLocation.transform.position;
    jump.Isolate(true);
    jump.jumpPoint = new JumpPoint(originPos, targetPos);
    jump.DoJump();
    }
}
```

А следующий – к площадке приземления:

```
using UnityEngine;

public class LandingLocation : MonoBehaviour
{
    public void OnTriggerEnter(Collider other)
    {
        if (!other.gameObject.CompareTag("Agent"))
            return;
        Agent agent = other.GetComponent<Agent>();
        Jump jump = other.GetComponent<Jump>();
        if (agent == null || jump == null)
            return;
        jump.Isolate(false);
        jump.jumpPoint = null;
    }
}
```

Полезные ссылки

- Рецепт *«Стрельба снарядами»*.

Глава 2

Навигация

В этой главе рассматриваются следующие рецепты:

- представление игрового мира с помощью сетей;
- представление игрового мира с помощью областей Дирихле;
- представление игрового мира с помощью точек видимости;
- представление игрового мира с помощью навигационного меша;
- поиск выхода из лабиринта с помощью алгоритма DFS;
- поиск кратчайшего пути в сети с помощью алгоритма BFS;
- поиск кратчайшего пути с помощью алгоритма Дейкстры;
- поиск оптимального пути с помощью алгоритма A*;
- улучшенный алгоритм A* с меньшим использованием памяти – алгоритм IDA*;
- планирование навигации на несколько кадров вперед: поиск с квантованием времени;
- сглаживание маршрута.

Введение

В этой главе рассматриваются алгоритмы поиска для сложных сценариев навигации. Игровые миры обычно представляют собой сложные структуры, будь это лабиринт, открытый игровой мир или что-то промежуточное между ними. Поэтому для решения такого рода задач требуется применение различных методов.

Рассмотрим несколько способов представления игровых миров, использующих разные виды графовых структур, и несколько алгоритмов поиска пути, подходящих для конкретных ситуаций.

Стоит отметить, что алгоритмы поиска опираются на такие методы, как `Seek` и `Arrive`, рассмотренные в предыдущей главе и применяемые для перемещения на карте.

Представление игрового мира с помощью сетей

Сеть zfot других структур используется для представления игровых миров, поскольку она легко реализуется и отображается. Тем не менее их фундаментом являются продвинутые представления, основанные на теории и свойствах графов.

Подготовка

Прежде всего создадим абстрактный класс графов Graph и объявим в нем виртуальные методы, реализующие все графовые представления. Такое решение выбрано потому, что независимо от внутреннего представления вершин и ребер алгоритмы поиска пути должны оперировать сущностями более высокого уровня, что позволит избежать реализации алгоритмов для каждого вида графового представления.

Этот класс послужит родительским классом для различных видов представлений, рассматриваемых в этой главе, и станет хорошей отправной точкой для реализации графовых представлений, не описанных в книге.

Определение класса Graph приводится ниже:

1. Создадим основу со свойствами:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public abstract class Graph : MonoBehaviour
{
    public GameObject vertexPrefab;
    protected List<Vertex> vertices;
    protected List<List<Vertex>> neighbours;
    protected List<List<float>> costs;
    // дальнейшая реализация описывается ниже
}
```

2. Определим функцию Start:

```
public virtual void Start()
{
    Load();
}
```

3. Определим упомянутую функцию Load:

```
public virtual void Load() { }
```

4. Реализуем функцию для получения размера графа:

```
public virtual int GetSize()
{
    if (ReferenceEquals(vertices, null))
        return 0;
    return vertices.Count;
}
```

5. Определим функцию поиска ближайших вершин для заданной позиции:

```
public virtual Vertex GetNearestVertex(Vector3 position)
{
    return null;
}
```

6. Реализуем функцию получения вершины по идентификатору:

```
public virtual Vertex GetVertexObj(int id)
{
    if (ReferenceEquals(vertices, null) || vertices.Count == 0)
        return null;
    if (id < 0 || id >= vertices.Count)
        return null;
    return vertices[id];
}
```

7. Реализуем функцию извлечения соседних вершин:

```
public virtual Vertex[] GetNeighbours(Vertex v)
{
    if (ReferenceEquals(neighbours, null) || neighbours.Count == 0)
        return new Vertex[0];
    if (v.id < 0 || v.id >= neighbours.Count)
        return new Vertex[0];
    return neighbours[v.id].ToArray();
}
```

Нам также потребуется класс Vertex:

```
using UnityEngine;
using System.Collections.Generic;
[System.Serializable]
public class Vertex : MonoBehaviour
```

```
{
    public int id;
    public List<Edge> neighbours;
    [HideInInspector]
    public Vertex prev;
}
```

И класс для хранения соседних вершин с их стоимостями. Назовем его Edge и реализуем:

1. Определим класс Edge, наследующий IComparable:

```
using System;

[System.Serializable]
public class Edge : IComparable<Edge>
{
    public float cost;
    public Vertex vertex;
    // дальнейшая реализация описывается ниже
}
```

2. Реализуем конструктор:

```
public Edge(Vertex vertex = null, float cost = 1f)
{
    this.vertex = vertex;
    this.cost = cost;
}
```

3. Определим метод сравнения:

```
public int CompareTo(Edge other)
{
    float result = cost - other.cost;
    int idA = vertex.GetInstanceID();
    int idB = other.vertex.GetInstanceID();
    if (idA == idB)
        return 0;
    return (int)result;
}
```

4. Реализуем функцию сравнения двух ребер:

```
public bool Equals(Edge other)
{
    return (other.vertex.id == this.vertex.id);
}
```

5. Переопределим функцию сравнения двух объектов:

```
public override bool Equals(object obj)
{
    Edge other = (Edge)obj;
    return (other.vertex.id == this.vertex.id);
}
```

6. Переопределим функцию извлечения хэш-кода. Она необходима для предыдущего метода:

```
public override int GetHashCode()
{
    return this.vertex.GetHashCode();
}
```

Кроме приведенных выше классов, необходимо также определить пару шаблонных объектов (prefab), основанных на кубических примитивах, для визуализации земли (например, с помощью кубического примитива низкой высоты), стен и прочих препятствий. Ссылка на шаблонный объект земли присваивается переменной `vertexPrefab`, а ссылка на шаблонный объект стен – переменной `obstaclePrefab`, которые будут объявлены в следующем разделе.

Наконец, создадим каталог `Maps` для хранения текстовых файлов, определяющих карты.

Как это реализовать...

А теперь углубимся в детали реализации сетевого графа. Сначала реализуем все функции обработки графа, предусмотрев место для размещения текстовых файлов, а в следующем разделе научимся читать файлы с расширением `.map`, часто используемые в играх:

1. Определим класс `GraphGridr`, наследующий класс Graph:

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.IO;

public class GraphGrid : Graph
{
    public GameObject obstaclePrefab;
    public string mapName = "arena.map";
    public bool get8Vicinity = false;
    public float cellSize = 1f;
    [Range(0, Mathf.Infinity)]
```

```
    public float defaultCost = 1f;
    [Range(0, Mathf.Infinity)]
    public float maximumCost = Mathf.Infinity;
    string mapsDir = "Maps";
    int numCols;
    int numRows;
    GameObject[] vertexObjs;
    // это необходимо
    // для чтения из тестового файла
    // в следующем разделе
    bool[,] mapVertices;
    // дальнейшая реализация описывается ниже
}
```

2. Определим функции GridToId и IdToGrid преобразования позиции в сети в индекс вершины и обратно соответственно:

```
private int GridToId(int x, int y)
{
    return Math.Max(numRows, numCols) * y + x;
}

private Vector2 IdToGrid(int id)
{
    Vector2 location = Vector2.zero;
    location.y = Mathf.Floor(id / numCols);
    location.x = Mathf.Floor(id % numCols);
    return location;
}
```

3. Определим функцию LoadMap для чтения текстового файла:

```
private void LoadMap(string filename)
{
    // Здесь будет реализована
    // процедура чтения файлов
    // с использованием
    // vertices[i, j] для логики представления и
    // vertexObjs[i, j] для хранения новых экземпляров
    // шаблонных объектов
}
```

4. Переопределим функцию LoadGraph:

```
public override void LoadGraph()
{
    LoadMap(mapName);
}
```

5. Переопределим функцию GetNearestVertex. Здесь используется традиционный подход, не рассматривающий полученную вершину в качестве препятствия. В дальнейшем она будет улучшена:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    position.x = Mathf.Floor(position.x / cellSize);
    position.y = Mathf.Floor(position.z / cellSize);
    int col = (int)position.x;
    int row = (int)position.z;
    int id = GridToId(col, row);
    return vertices[id];
  }
```

6. Переопределим функцию GetNearestVertex и реализуем в ней алгоритм поиска в ширину (Breadth-First Search, BFS), который будет описан ниже в этой же главе:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    int col = (int)(position.x / cellSize);
    int row = (int)(position.z / cellSize);
    Vector2 p = new Vector2(col, row);
    // дальнейшая реализация описывается ниже
}
```

7. Определим список проверенных позиций (вершин) и очередь позиций, которые еще предстоит проверить:

```
List<Vector2> explored = new List<Vector2>();
Queue<Vector2> queue = new Queue<Vector2>();
queue.Enqueue(p);
```

8. Следующий цикл выполняется, пока в очереди остаются непроверенные элементы. Если все элементы проверены, возвращается null:

```
do
{
    p = queue.Dequeue();
    col = (int)p.x;
    row = (int)p.y;
    int id = GridToId(col, row);
    // дальнейшая реализация описывается ниже
} while (queue.Count != 0);
return null;
```

9. Если это допустимая вершина, немедленно извлекаем ее:

```
if (mapVertices[row, col])
    return vertices[id];
```

10. Добавим позицию в список проверенных, если ее еще там нет:

```
if (!explored.Contains(p))
{
    explored.Add(p);
    int i, j;
    // дальнейшая реализация описывается ниже
}
```

11. Добавим всех допустимых соседей в очередь:

```
for (i = row - 1; i <= row + 1; i++)
{
    for (j = col - 1; j <= col + 1; j++)
    {
        if (i < 0 || j < 0)
            continue;
        if (j >= numCols || i >= numRows)
            continue;
        if (i == row && j == col)
            continue;
        queue.Enqueue(new Vector2(j, i));
    }
}
```

Как это работает...

Рис. 2.1 ❖ Преобразование позиций из двумерного в одномерное представление

Алгоритм использует свои внутренние функции для адаптации общедоступных функций, унаследованных от родительского класса, и преобразует позиции из двумерного в одномерное представление (индекс вершин) с применением простых математических функций.

Функцию LoadMap вы должны реализовать сами, но в следующем разделе будет представлен один из возможных ее вариантов для чтения определенных видов текстовых файлов с картами, основанными на сетях.

Дополнительная информация...

Рассмотрим один из вариантов реализации функции LoadMap, использовав в качестве примера файл в формате .map:

1. Определим эту функцию и создадим объект StreamReader для чтения файла:

```
private void LoadMap(string filename)
{
    string path = Application.dataPath + "/" + mapsDir + "/" + filename;
    try
    {
        StreamReader strmRdr = new StreamReader(path);
        using (strmRdr)
        {
            // дальнейшая реализация описывается ниже
        }
    }
    catch (Exception e)
    {
        Debug.LogException(e);
    }
}
```

2. Объявим и инициализируем необходимые переменные:

```
int j = 0;
int i = 0;
int id = 0;
string line;
Vector3 position = Vector3.zero;
Vector3 scale = Vector3.zero;
```

3. Прочитаем заголовок файла, содержащий высоту и ширину:

```
line = strmRdr.ReadLine();// строка, не содержащая важной информации
line = strmRdr.ReadLine();// высота
numRows = int.Parse(line.Split(' ')[1]);
```

```
line = strmRdr.ReadLine();// ширина
numCols = int.Parse(line.Split(' ')[1]);
line = strmRdr.ReadLine();// строка "map" в файле
```

4. Инициализируем свойства, выделив для них память:

```
vertices = new List<Vertex>(numRows * numCols);
neighbours = new List<List<Vertex>>(numRows * numCols);
costs = new List<List<float>>(numRows * numCols);
vertexObjs = new GameObject[numRows * numCols];
    mapVertices = new bool[numRows, numCols];
```

5. Определим цикл for для обхода символов в следующих строках:

```
for (i = 0; i < numRows; i++)
{
    line = strmRdr.ReadLine();
    for (j = 0; j < numCols; j++)
    {
        // дальнейшая реализация описывается ниже
    }
}
```

6. Присвоим логическому представлению символа значение true или false:

```
bool isGround = true;
if (line[j] != '.')
    isGround = false;
mapVertices[i, j] = isGround;
```

7. Создадим экземпляр шаблонного объекта:

```
position.x = j * cellSize;
position.z = i * cellSize;
id = GridToId(j, i);
if (isGround)
    vertexObjs[id] = Instantiate(vertexPrefab, position,
        Quaternion.identity) as GameObject;
else
    vertexObjs[id] = Instantiate(obstaclePrefab, position,
        Quaternion.identity) as GameObject;
```

8. Добавим новый игровой объект в граф и дадим ему соответствующее имя:

```
vertexObjs[id].name = vertexObjs[id].name.Replace("(Clone)",
    id.ToString());
Vertex v = vertexObjs[id].AddComponent<Vertex>();
v.id = id;
```

```
vertices.Add(v);
neighbours.Add(new List<Vertex>());
costs.Add(new List<float>());
float y = vertexObjs[id].transform.localScale.y;
scale = new Vector3(cellSize, y, cellSize);
vertexObjs[id].transform.localScale = scale;
vertexObjs[id].transform.parent = gameObject.transform;
```

9. Определим два вложенных цикла сразу после предыдущего, чтобы настроить соседей для каждой вершины:

```
for (i = 0; i < numRows; i++)
{
    for (j = 0; j < numCols; j++)
    {
        SetNeighbours(j, i);
    }
}
```

10. Определим функцию SetNeighbours, используемую выше:

```
protected void SetNeighbours(int x, int y, bool get8 = false)
{
    int col = x;
    int row = y;
    int i, j;
    int vertexId = GridToId(x, y);
    neighbours[vertexId] = new List<Vertex>();
    costs[vertexId] = new List<float>();
    Vector2[] pos = new Vector2[0];
    // дальнейшая реализация описывается ниже
}
```

11. Вычислим значения, соответствующие ближайшим восьми вершинам (сверху, снизу, справа, слева и по углам):

```
if (get8)
{
    pos = new Vector2[8]; int c = 0;
    for (i = row - 1; i <= row + 1; i++)
    {
        for (j = col -1; j <= col; j++)
        {
            pos[c] = new Vector2(j, i);
            c++;
        }
    }
}
```

12. Настроим ближайшие четыре вершины (кроме угловых):

```
else
{
    pos = new Vector2[4];
    pos[0] = new Vector2(col, row - 1);
    pos[1] = new Vector2(col - 1, row);
    pos[2] = new Vector2(col + 1, row);
    pos[3] = new Vector2(col, row + 1);
}
```

13. Добавим соседние вершины в списки. Это та же процедура, что и для ближайших:

```
foreach (Vector2 p in pos)
{
    i = (int)p.y;
    j = (int)p.x;
    if (i < 0 || j < 0)
        continue;
    if (i >= numRows || j >= numCols)
        continue;
    if (i == row && j == col)
        continue;
    if (!mapVertices[i, j])
        continue;
    int id = GridToId(j, i);
    neighbours[vertexId].Add(vertices[id]);
    costs[vertexId].Add(defaultCost);
}
```

Полезные ссылки

За дополнительной информацией о форматах карт обращайтесь на сайт *лаборатории Moving AI*, возглавляемой профессором Стертевантом (Sturtevant): http://movingai.com/benchmarks/, где также можно загрузить некоторые бесплатные карты.

Представление игрового мира с помощью областей Дирихле

Области Дирихле, называемые по-другому диаграммой Вороного, – один из способов разделения пространства на регионы, состоящие из наборов точек, расположенных к заданной точке ближе, чем все про-

чие. Такое графическое представление помогает разделить пространство с помощью примитивов Unity или мешей, что не полностью соответствует определению, а является просто использованием идеи в качестве средства достижения конечной цели. Области Дирихле обычно отображаются в виде конусов, ограничивающих область для данной вершины, но здесь этот принцип адаптирован под конкретные нужды и имеющиеся инструменты.

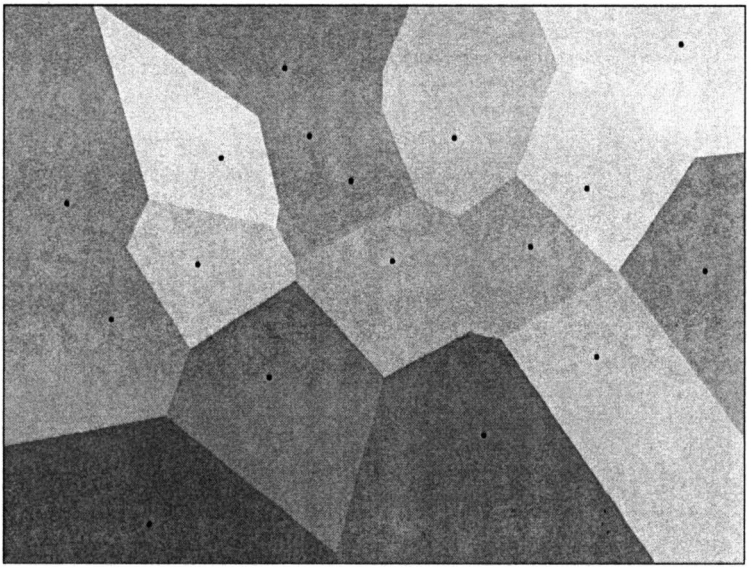


Рис. 2.2 ❖ Пример диаграммы, или полигона Вороного

Подготовка

Перед определением нового класса графа `Graph` создадим класс `VertexReport`, затем внесем некоторые изменения в класс `Graph` и добавим в проект тег `Vertex`:

1. Поместим определение класса `VertexReport` в тот же файл, перед классом `Graph`:

```
public class VertexReport
{
    public int vertex;
    public GameObject obj;
    public VertexReport(int vertexId, GameObject obj)
```

```
    {
        vertex = vertexId;
        this.obj = obj;
    }
}
```

Обратите внимание, что к объектам вершин в сцене необходимо присоединить компонент коллайдера и добавить тег `Vertex`. Эти объекты могут быть примитивами или мешами, ограничивающими максимальный размер области, рассматриваемой как узел для вершины.

Как это реализовать...

Разобьем этот рецепт на две части. Сначала определим реализацию вершин, а затем перейдем к графу:

1. Определим класс `VertexDirichlet`, наследующий класс `Vertex`:

```
using UnityEngine;
public class VertexDirichlet : Vertex
{
    // дальнейшая реализация описывается ниже
}
```

2. Определим функцию `OnTriggerEnter` для регистрации объекта в текущей вершине:

```
public void OnTriggerEnter(Collider col)
{
    string objName = col.gameObject.name;
    if (objName.Equals("Agent") || objName.Equals("Player"))
    {
        VertexReport report = new VertexReport(id, col.gameObject);
        SendMessageUpwards("AddLocation", report);
    }
}
```

3. Определим функцию `OnTriggerExit` для обратной процедуры:

```
public void OnTriggerExit(Collider col)
{
    string objName = col.gameObject.name;
    if (objName.Equals("Agent") || objName.Equals("Player"))
    {
        VertexReport report = new VertexReport(id, col.gameObject);
        SendMessageUpwards("RemoveLocation", report);
    }
}
```

4. Определим класс GraphDirichlet, наследующий класс Graph:

```
using UnityEngine;
using System.Collections.Generic;

public class GraphDirichlet : Graph
{
    Dictionary<int, List<int>> objToVertex;
}
```

5. Реализуем функцию AddLocation, используемую выше:

```
public void AddLocation(VertexReport report)
{
    int objId = report.obj.GetInstanceID();
    if (!objToVertex.ContainsKey(objId))
    {
        objToVertex.Add(objId, new List<int>());
    }
    objToVertex[objId].Add(report.vertex);
}
```

6. Определим также функцию RemoveLocation:

```
public void RemoveLocation(VertexReport report)
{
    int objId = report.obj.GetInstanceID();
    objToVertex[objId].Remove(report.vertex);
}
```

7. Переопределим функцию Start для инициализации свойств:

```
public override void Start()
{
    base.Start();
    objToVertex = new Dictionary<int, List<int>>();
}
```

8. Реализуем функцию Load, которая подключит все, что необходимо:

```
public override void Load()
{
    Vertex[] verts = GameObject.FindObjectsOfType<Vertex>();
    vertices = new List<Vertex>(verts);
    for (int i = 0; i < vertices.Count; i++)
    {
        VertexVisibility vv = vertices[i] as VertexVisibility;
```

```
            vv.id = i;
            vv.FindNeighbours(vertices);
        }
    }
```

9. Переопределим функцию GetNearestVertex:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    Vertex vertex = null;
    float dist = Mathf.Infinity;
    float distNear = dist;
    Vector3 posVertex = Vector3.zero;
    for (int i = 0; i < vertices.Count; i++)
    {
        posVertex = vertices[i].transform.position;
        dist = Vector3.Distance(position, posVertex);
        if (dist < distNear)
        {
            distNear = dist;
            vertex = vertices[i];
        }
    }
    return vertex;
}
```

10. Определим перегруженную версию функции GetNearestVertex, принимающую игровой объект:

```
public Vertex GetNearestVertex(GameObject obj)
{
    int objId = obj.GetInstanceID();
    Vector3 objPos = obj.transform.position;
    if (!objToVertex.ContainsKey(objId))
        return null;
    List<int> vertIds = objToVertex[objId];
    Vertex vertex = null;
    float dist = Mathf.Infinity;
    for (int i = 0; i < vertIds.Count; i++)
    {
        int id = vertIds[i];
        Vertex v = vertices[id];
        Vector3 vPos = v.transform.position;
        float d = Vector3.Distance(objPos, vPos);
        if (d < dist)
        {
```

```
                vertex = v;
                dist = d;
            }
        }
        return vertex;
    }
```

11. Реализуем функцию GetNeighbors:

```
public override Vertex[] GetNeighbours(Vertex v)
{
    List<Edge> edges = v.neighbours;
    Vertex[] ns = new Vertex[edges.Count];
    int i;
    for (i = 0; i < edges.Count; i++)
    {
        ns[i] = edges[i].vertex;
    }
    return ns;
}
```

12. И наконец, определим функцию GetEdges:

```
public override Edge[] GetEdges(Vertex v)
{
    return vertices[v.id].neighbours.ToArray();
}
```

Как это работает...

Когда агент или игрок входит в область, принадлежащую вершине, он посылает сообщение родительскому классу графа и указывает на вершину в словаре объектов, упрощая дискретизацию. Когда игрок покидает область, выполняется обратное действие. Когда игрок отображается в несколько вершин, функция возвращает индекс ближайшей из них.

Кроме того, использование словаря упрощает процесс преобразования идентификаторов экземпляров объектов в индексы массива вершин.

Дополнительная информация...

Имейте в виду, что при использовании реализованного метода размещение вершин и установка связей между ними (ребра) должны выполняться вручную. Если вам понадобится более удобный (или автоматизированный) механизм, попробуйте реализовать его сами.

В следующем рецепте мы рассмотрим такой автоматизированный способ получения соседних вершин путем отбрасывания лучей, который может послужить отправной точкой для реализации.

Полезные ссылки

- Рецепт *«Представление игрового мира с помощью точек видимости»*.

Представление игрового мира с помощью точек видимости

Это еще один часто применяемый метод представления игрового мира в виде точек, расположенных в области навигации, размещаемых вручную или автоматически, с помощью сценариев. Здесь используются точки, размещаемые вручную и подключаемые автоматически с помощью сценария.

Подготовка

Так же, как в предыдущем рецепте, сначала определим, что нам потребуется:

- имеющийся класс `Edge` следует добавить в тот же файл, перед классом `Graph`;
- определить функцию `GetEdges` в классе `Graph`;
- реализовать класс `Vertex`.

Обратите внимание, что к объектам вершин в сцене необходимо присоединить компонент коллайдера и добавить тег `Vertex`. В качестве коллайдера рекомендуется использовать примитив сферы `Sphere`.

Как это реализовать...

Создадим класс графового представления и собственный класс `Vertex`:

1. Определим класс `VertexVisibility`, наследующий класс `Vertex`:

```
using UnityEngine;
using System.Collections.Generic;

public class VertexVisibility : Vertex
{
    void Awake()
    {
```

```
        neighbours = new List<Edge>();
    }
}
```

2. Определим функцию FindNeighbours для автоматизации процесса соединения вершин между собой:

```
public void FindNeighbours(List<Vertex> vertices)
{
    Collider c = gameObject.GetComponent<Collider>();
    c.enabled = false;
    Vector3 direction = Vector3.zero;
    Vector3 origin = transform.position;
    Vector3 target = Vector3.zero;
    RaycastHit[] hits;
    Ray ray;
    float distance = 0f;
    // дальнейшая реализация описывается ниже
}
```

3. Обойдем все объекты, отбрасывая лучи для проверки прямой видимости, попутно пополняя список соседей:

```
for (int i = 0; i < vertices.Count; i++)
{
    if (vertices[i] == this)
        continue;
    target = vertices[i].transform.position;
    direction = target - origin;
    distance = direction.magnitude;
    ray = new Ray(origin, direction);
    hits = Physics.RaycastAll(ray, distance);
    if (hits.Length == 1)
    {
        if (hits[0].collider.gameObject.tag.Equals("Vertex"))
        {
            Edge e = new Edge();
            e.cost = distance;
            GameObject go = hits[0].collider.gameObject;
            Vertex v = go.GetComponent<Vertex>();
            if (v != vertices[i])
                continue;
            e.vertex = v;
            neighbours.Add(e);
        }
    }
}
c.enabled = true;
```

4. Определим класс GraphVisibility:

```
using UnityEngine;
using System.Collections.Generic;

public class GraphVisibility : Graph
{
    // дальнейшая реализация описывается ниже
}
```

5. Определим функцию Load для создания связей между вершинами:

```
public override void Load()
{
    Vertex[] verts = GameObject.FindObjectsOfType<Vertex>();
    vertices = new List<Vertex>(verts);
    for (int i = 0; i < vertices.Count; i++)
    {
        VertexVisibility vv = vertices[i] as VertexVisibility;
        vv.id = i;
        vv.FindNeighbours(vertices);
    }
}
```

6. Определим функцию GetNearesteVertex:

```
public override Vertex GetNearestVertex(Vector3 position)
{
    Vertex vertex = null;
    float dist = Mathf.Infinity;
    float distNear = dist;
    Vector3 posVertex = Vector3.zero;
    for (int i = 0; i < vertices.Count; i++)
    {
        posVertex = vertices[i].transform.position;
        dist = Vector3.Distance(position, posVertex);
        if (dist < distNear)
        {
            distNear = dist;
            vertex = vertices[i];
        }
    }
    return vertex;
}
```

7. Определим функцию GetNeighbours:

```
public override Vertex[] GetNeighbours(Vertex v)
{
    List<Edge> edges = v.neighbours;
    Vertex[] ns = new Vertex[edges.Count];
    int i;
    for (i = 0; i < edges.Count; i++)
    {
        ns[i] = edges[i].vertex;
    }
    return ns;
}
```

8. И наконец, переопределим функцию GetEdges:

```
public override Edge[] GetEdges(Vertex v)
{
    return vertices[v.id].neighbours.ToArray();
}
```

Как это работает...

Родительский класс GraphVisibility выполняет обход всех вершин в сцене и применяет к каждой функцию FindNeighbours, чтобы автоматически создать граф, но исходные видимые точки должны размещаться вручную. Кроме того, расстояние между двумя точками определяет стоимость соответствующего ребра.

Дополнительная информация...

Ребрами соединяются только точки, находящиеся в прямой видимости друг с другом. Это решение подходит также для создания интеллектуальных графов, учитывающих лестницы и скалы, нужно лишь поместить функцию Load в доступный из редактора класс, чтобы ее можно было вызвать в режиме редактирования, с возможностью дальнейшего изменения или удаления соответствующих ребер, для достижения нужного результата.

Вернитесь к разделу *«Подготовка»* из предыдущего рецепта, если почувствовали, что упустили что-то важное.

За дополнительной информацией о пользовательских редакторах, редакторе сценариев и порядке выполнения кода в режиме редактирования обращайтесь к документации Unity:

- http://docs.unity3d.com/ScriptReference/Editor.html;

- http://docs.unity3d.com/ScriptReference/ExecuteInEditMode.html;
- http://docs.unity3d.com/Manual/PlatformDependentCompilation.html.

Полезные ссылки

- Рецепт *«Представление игрового мира с помощью областей Дирихле»*.

Представление игрового мира с помощью навигационного меша

В сложных ситуациях, например когда применяется несколько видов графов, а ручное размещение вершин слишком хлопотно, поскольку требует массы времени для покрытия больших областей, необходимо использовать навигационный меш.

Поэтому сейчас мы посмотрим, как из меша модели получить навигационный меш, используя в качестве вершин центры треугольников, и затем обработать его, как описывалось в предыдущем рецепте.

Подготовка

Этот рецепт требует определенного опыта создания сценариев для управления редактором и понимания реализации точек видимости в графовом представлении. Также стоит отметить, что сценарий автоматически создает экземпляр игрового объекта `CustomNavMesh` в сцене, и ему необходим шаблонный объект, подобно прочим графовым представлениям.

Наконец, нам потребуется определить класс `CustomNavMesh`, наследующий класс `GraphVisibility`:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class CustomNavMesh : GraphVisibility
{
    public override void Start()
    {
        instIdToId = new Dictionary<int, int>();
    }
}
```

Как это реализовать...

Создадим окно редактора, чтобы упростить процесс автоматизации и обойтись без утяжеления функции Start графа, что чревато замедлением загрузки сцены.

1. Создадим класс CustomNavMeshWindow и поместим его в каталог Editor:

```
using UnityEngine; using UnityEditor;
using System.Collections;
using System.Collections.Generic;

public class CustomNavMeshWindow : EditorWindow
{
    // дальнейшая реализация описывается ниже
}
```

2. Добавим атрибуты в окно редактора:

```
static bool isEnabled = false;
static GameObject graphObj;
static CustomNavMesh graph;
static CustomNavMeshWindow window;
static GameObject graphVertex;
```

3. Реализуем функцию для инициализации и отображения окна:

```
[MenuItem("UAIPC/Ch02/CustomNavMeshWindow")]
static void Init()
{
    window = EditorWindow.GetWindow<CustomNavMeshWindow>();
    window.title = "CustomNavMeshWindow";
    SceneView.onSceneGUIDelegate += OnScene;
    graphObj = GameObject.Find("CustomNavMesh");
    if (graphObj == null)
    {
        graphObj = new GameObject("CustomNavMesh");
        graphObj.AddComponent<CustomNavMesh>();
        graph = graphObj.GetComponent<CustomNavMesh>();
    }
    else
    {
        graph = graphObj.GetComponent<CustomNavMesh>();
        if (graph == null)
            graphObj.AddComponent<CustomNavMesh>();
        graph = graphObj.GetComponent<CustomNavMesh>();
    }
}
```

4. Определим функцию OnDestroy:

```
void OnDestroy()
{
    SceneView.onSceneGUIDelegate -= OnScene;
}
```

5. Реализуем функцию OnGUI для прорисовки интерьера окна:

```
void OnGUI()
{
    isEnabled = EditorGUILayout.Toggle("Enable Mesh Picking",
                                       isEnabled);
    if (GUILayout.Button("Build Edges"))
    {
        if (graph != null)
            graph.LoadGraph();
    }
}
```

6. Реализуем первую половину функции OnScene для обработки щелчка левой кнопкой мыши на окне сцены:

```
private static void OnScene(SceneView sceneView)
{
    if (!isEnabled)
        return;
    if (Event.current.type == EventType.MouseDown)
    {
        graphVertex = graph.vertexPrefab;
        if (graphVertex == null)
        {
            Debug.LogError("No Vertex Prefab assigned");
            return;
        }
        Event e = Event.current;
        Ray ray = HandleUtility.GUIPointToWorldRay(e. mousePosition);
        RaycastHit hit;
        GameObject newV;
        // дальнейшая реализация описывается ниже
    }
}
```

7. Реализуем вторую половину функции обработки щелчка на меше:

```
if (Physics.Raycast(ray, out hit))
{
```

```
        GameObject obj = hit.collider.gameObject;
        Mesh mesh = obj.GetComponent<MeshFilter>().sharedMesh;
        Vector3 pos;
        int i;
        for (i = 0; i < mesh.triangles.Length; i += 3)
        {
            int i0 = mesh.triangles[i];
            int i1 = mesh.triangles[i + 1];
            int i2 = mesh.triangles[i + 2];
            pos = mesh.vertices[i0];
            pos += mesh.vertices[i1];
            pos += mesh.vertices[i2];
            pos /= 3;
            newV = (GameObject)Instantiate(graphVertex, pos,
                                           Quaternion.identity);
            newV.transform.Translate(obj.transform.position);
            newV.transform.parent = graphObj.transform;
            graphObj.transform.parent = obj.transform;
        }
    }
```

Как это работает...

Код, представленный выше, создает нестандартное окно редактора и настраивает обработчик событий `OnScene` в окне сцены. Кроме того, он создает узлы графа, выполняя обход массива вершин меша и определяя центр каждого треугольника. В заключение вызовом функции `LoadGraph` графа определяются соседи.

Поиск выхода из лабиринта с помощью алгоритма DFS

Алгоритм поиска в глубину **(Depth-First Search, DFS)** описывает поиск маршрута и широко используется на устройствах с небольшим объемом памяти. С небольшими изменениями, касающимися списков посещавшихся и обнаруженных узлов, этот алгоритм также часто используется для автоматического создания лабиринтов, при этом суть алгоритма от этого не меняется.

Подготовка

Это алгоритм высокого уровня, опирающийся на наличие общих функций во всех реализациях графов, поэтому сам он реализован в классе `Graph`.

Как это реализовать...

Несмотря на то что рецепт содержит только определение функции, обратите внимание на комментарии в коде, чтобы лучше разобраться в его структуре:

1. Определим функцию GetPathDFS:

```
public List<Vertex> GetPathDFS(GameObject srcObj, GameObject dstObj)
{
    // дальнейшая реализация описывается ниже
}
```

2. Проверим входные объекты на значение null:

```
if (srcObj == null || dstObj == null)
    return new List<Vertex>();
```

3. Объявим и инициализируем необходимые переменные:

```
Vertex src = GetNearestVertex(srcObj.transform.position);
Vertex dst = GetNearestVertex(dstObj.transform.position);
Vertex[] neighbours;
Vertex v;
int[] previous = new int[vertices.Count];
for (int i = 0; i < previous.Length; i++)
    previous[i] = -1;
previous[src.id] = src.id;
Stack<Vertex> s = new Stack<Vertex>();
s.Push(src);
```

4. Реализуем алгоритм DFS поиска пути:

```
while (s.Count != 0)
{
    v = s.Pop();
    if (ReferenceEquals(v, dst))
    {
        return BuildPath(src.id, v.id, ref previous);
    }

    neighbours = GetNeighbours(v);
    foreach (Vertex n in neighbours)
    {
        if (previous[n.id] != -1)
            continue;
        previous[n.id] = v.id;
        s.Push(n);
    }
}
```

Как это работает...

Алгоритм основан на итерационной версии DFS. Он также базируется на обходе в порядке, определяемом графом, и принципе LIFO (первый пришел, первый ушел), который реализуется с помощью стека для хранения посещавшихся и вновь обнаруженных узлов.

Что дальше...

Здесь вызывается функция `BuildPath`, которая еще не реализована. Важно отметить, что она применяется практически во всех алгоритмах поиска, рассматриваемых в этой главе, и поэтому не является частью рецепта.

Код метода `BuildPath` приводится ниже:

```
private List<Vertex> BuildPath(int srcId, int dstId, ref int[] prevList)
{
    List<Vertex> path = new List<Vertex>();
    int prev = dstId;
    do
    {
        path.Add(vertices[prev]);
        prev = prevList[prev];
    } while (prev != srcId);
    return path;
}
```

Поиск кратчайшего пути в сети с помощью алгоритма BFS

Алгоритм поиска в ширину (**Breadth-First Search, BFS**) – еще один базовый механизм обхода графа и предназначен для получения кратчайшего пути за наименьшее количество шагов. Он требует достаточно много памяти и обычно используется на мощных консолях и компьютерах.

Подготовка

Это алгоритм высокого уровня, опирающийся на наличие общих функций во всех реализациях графов, поэтому сам он реализован в классе `Graph`.

Как это реализовать...

Несмотря на то что рецепт содержит только определение функции, обратите внимание на комментарии в коде, чтобы лучше разобраться в его структуре:

1. Объявим функцию GetPathBFS:

```
public List<Vertex> GetPathBFS(GameObject srcObj, GameObject dstObj)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    // дальнейшая реализация описывается ниже
}
```

2. Объявим и инициализируем необходимые переменные:

```
Vertex[] neighbours;
Queue<Vertex> q = new Queue<Vertex>();
Vertex src = GetNearestVertex(srcObj.transform.position);
Vertex dst = GetNearestVertex(dstObj.transform.position);
Vertex v;
int[] previous = new int[vertices.Count];
for (int i = 0; i < previous.Length; i++)
    previous[i] = -1;
previous[src.id] = src.id;
q.Enqueue(src);
```

3. Реализуем алгоритм BFS поиска пути:

```
while (q.Count != 0)
{
    v = q.Dequeue();
    if (ReferenceEquals(v, dst))
    {
        return BuildPath(src.id, v.id, ref previous);
    }

    neighbours = GetNeighbours(v);
    foreach (Vertex n in neighbours)
    {
        if (previous[n.id] != -1) continue;
        previous[n.id] = v.id;
        q.Enqueue(n);
    }
}
return new List<Vertex>();
```

Как это работает...

Алгоритм BFS похож на алгоритм DFS, поскольку так же основан на определенном порядке обхода графа, но вместо стека, как в алгоритме DFS, алгоритм BFS использует очередь для хранения посещавшихся и вновь обнаруженных узлов.

Что дальше...

Метод BuildPath здесь не реализован – его реализация приводится в рецепте поиска в глубину.

Полезные ссылки

- Рецепт *«Поиск выхода из лабиринта с помощью алгоритма DFS»*.

Поиск кратчайшего пути с помощью алгоритма Дейкстры

Алгоритм Дейкстры первоначально был разработан для решения задачи поиска кратчайшего пути в графе. Поэтому алгоритм позволяет с минимальными затратами определить маршрут к любой цели из заданной точки. Рассмотрим его использование, опираясь на два разных подхода.

Подготовка

Прежде всего нужно импортировать в проект класс бинарной кучи с сайта **Game Programming Wiki (GPWiki)**, потому что фреймворки .Net и Mono не имеют встроенных средств для работы с бинарной кучей или приоритетными очередями.

Загрузить исходный файл класса и больше узнать о двоичной куче можно на сайте GP Wiki: http://content.gpwiki.org/index.php/C_sharp:BinaryHeapOfT.

Как это реализовать...

Сначала реализуем алгоритм Дейкстры с тем же количеством параметров, что и другие алгоритмы, а затем изменим реализацию для выполнения алгоритмом первоначальной цели.

1. Определим функцию GetPathDijkstra вместе с внутренними переменными:

```
public List<Vertex> GetPathDijkstra(GameObject srcObj, GameObject dstObj)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    Vertex src = GetNearestVertex(srcObj.transform.position);
    Vertex dst = GetNearestVertex(dstObj.transform.position);
    GPWiki.BinaryHeap<Edge> frontier = new GPWiki.BinaryHeap<Edge>();
    Edge[] edges;
```

```
    Edge node, child;
    int size = vertices.Count;
    float[] distValue = new float[size];
    int[] previous = new int[size];
    // дальнейшая реализация описывается ниже
}
```

2. Добавим исходный узел в кучу (работает подобно приоритетной очереди) и назначим всем узлам, кроме исходного, расстояние, равное бесконечности:

```
node = new Edge(src, 0);
frontier.Add(node);
distValue[src.id] = 0;
previous[src.id] = src.id;
for (int i = 0; i < size; i++)
{
    if (i == src.id)
        continue;
    distValue[i] = Mathf.Infinity;
    previous[i] = -1;
}
```

3. Определим цикл обхода очереди:

```
while (frontier.Count != 0)
{
    node = frontier.Remove();
    int nodeId = node.vertex.id;
    // дальнейшая реализация описывается ниже
}
return new List<Vertex>();
```

4. Код обработки прибытия в место назначения:

```
if (ReferenceEquals(node.vertex, dst))
{
    return BuildPath(src.id, node.vertex.id, ref previous);
}
```

5. В противном случае выполняются обработка посещавшихся узлов и добавление их соседей в очередь. Затем полученный путь возвращается (если имеется):

```
edges = GetEdges(node.vertex);
foreach (Edge e in edges)
{
    int eId = e.vertex.id;
```

```
        if (previous[eId] != -1)
            continue;
        float cost = distValue[nodeId] + e.cost;
        if (cost < distValue[e.vertex.id])
        {
            distValue[eId] = cost;
            previous[eId] = nodeId;
            frontier.Remove(e);
            child = new Edge(e.vertex, cost);
            frontier.Add(child);
        }
    }
```

Как это работает...

Алгоритм Дейкстры работает подобно алгоритму BFS, но оценивает неотрицательные стоимости ребер для построения наилучшего маршрута из исходной вершины в любую другую. Поэтому здесь используется массив для хранения предыдущих вершин.

Дополнительная информация...

А теперь рассмотрим, как изменить текущий алгоритм Дейкстры, чтобы задействовать приемы предварительной обработки и сократить время поиска. Этот процесс можно разбить на три крупных этапа: изменение основного алгоритма, создание функции предварительной обработки (которую удобно использовать, например, в режиме редактора) и, наконец, определение функции поиска.

1. Изменим сигнатуру основной функции:

   ```
   public int[] Dijkstra(GameObject srcObj)
   ```

2. Изменим возвращаемое значение:

   ```
   return previous;
   ```

3. Удалим строки, добавленные на шаге 4 в разделе *«Как это реализовать»*.
4. Дополнительно удалим следующую строку в начале:

   ```
   Vertex dst = GetNearestVertex(dstObj.transform.position);
   ```

5. Добавим новое свойство в класс `Graph`:

   ```
   List<int[]> routes = new List<int[]>();
   ```

6. Определим функцию предварительной обработки `DijkstraProcessing`:

```
public void DijkstraProcessing()
{
    int size = GetSize();
    for (int i = 0; i < size; i++)
    {
        GameObject go = vertices[i].gameObject;
        routes.add(Dijkstra(go));
    }
}
```

7. Реализуем новую функцию `GetPathDijkstra` поиска:

```
public List<Vertex> GetPathDijkstra(GameObject srcObj, GameObject dstObj)
{
    List<Vertex> path = new List<Vertex>();
    Vertex src = GetNearestVertex(srcObj);
    Vertex dst = GetNearestVertex(dstObj);
    return BuildPath(src.id, dst.id, ref routes[dst.id]);
}
```

Метод `BuildPath` здесь не реализован – его реализация приводится в рецепте поиска в глубину.

Полезные ссылки

- Рецепт *«Поиск выхода из лабиринта с помощью алгоритма DFS»*.

Поиск оптимального пути с помощью алгоритма A*

Благодаря простоте и эффективности, а также широким возможностям оптимизации алгоритм A* чаще других используется для поиска пути. Не случайно существует несколько алгоритмов, основанных на нем. Поскольку алгоритм A* имеет общие корни с алгоритмом Дейкстры, в их реализациях просматривается сходство.

Подготовка

Так же как алгоритм Дейкстры, этот рецепт использует бинарную кучу с сайта GPWiki. Кроме того, важно понимать, что такое делегаты (обработчики) и для чего они предназначены. И наконец, в этом рецепте мы входим в мир осознанного поиска, а это означает, что вы должны понимать, что такое эвристика и для чего она нужна.

Проще говоря, этот рецепт основан на эвристической функции вычисления приблизительной стоимости перехода между двумя верши-

нами, которая позволяет сравнивать альтернативы и выбирать те, что характеризуются минимальной стоимостью.

Внесем небольшие изменения в класс Graph:

1. Определим ссылку на делегата:

```
public delegate float Heuristic(Vertex a, Vertex b);
```

2. Реализуем функцию вычисления эвклидова расстояния для использования в качестве эвристики по умолчанию:

```
public float EuclidDist(Vertex a, Vertex b)
{
    Vector3 posA = a.transform.position;
    Vector3 posB = b.transform.position;
    return Vector3.Distance(posA, posB);
}
```

3. Реализуем функцию манхэттенского расстояния для использования в качестве другой эвристики. Это позволит сравнить результаты с использованием различных эвристик:

```
public float ManhattanDist(Vertex a, Vertex b)
{
    Vector3 posA = a.transform.position;
    Vector3 posB = b.transform.position;
    return Mathf.Abs(posA.x - posB.x) + Mathf.Abs(posA.y - posB.y);
}
```

Как это реализовать...

Несмотря на то что рецепт содержит только определение функции, обратите внимание на комментарии в коде, чтобы лучше разобраться в его структуре:

1. Определим функцию GetPathDijkstra с внутренними переменными:

```
public List<Vertex> GetPathAstar(GameObject srcObj,
                                 GameObject dstObj, Heuristic h = null)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    if (ReferenceEquals(h, null))
        h = EuclidDist;

    Vertex src = GetNearestVertex(srcObj.transform.position);
    Vertex dst = GetNearestVertex(dstObj.transform.position);
    GPWiki.BinaryHeap<Edge> frontier = new GPWiki.BinaryHeap<Edge>();
```

```
    Edge[] edges; Edge node, child;
    int size = vertices.Count;
    float[] distValue = new float[size];
    int[] previous = new int[size];
    // дальнейшая реализация описывается ниже
}
```

2. Добавим исходный узел в кучу (работает подобно приоритетной очереди) и назначим всем узлам, кроме исходного, расстояние, равное бесконечности:

```
node = new Edge(src, 0);
frontier.Add(node);
distValue[src.id] = 0;
previous[src.id] = src.id;
for (int i = 0; i < size; i++)
{
    if (i == src.id)
        continue;
    distValue[i] = Mathf.Infinity;
    previous[i] = -1;
}
```

3. Определим цикл для обхода графа:

```
while (frontier.Count != 0)
{
    // дальнейшая реализация описывается ниже
}
return new List<Vertex>();
```

4. Реализуем условие возвращения пути:

```
node = frontier.Remove();
int nodeId = node.vertex.id;
if (ReferenceEquals(node.vertex, dst))
{
    return BuildPath(src.id, node.vertex.id, ref previous);
}
```

5. Получим соседние вершины (в некоторых книгах они называются дочерними):

```
edges = GetEdges(node.vertex);
```

6. Обойдем соседей и вычислим стоимость:

```
foreach (Edge e in edges)
{
```

```
        int eId = e.vertex.id;
        if (previous[eId] != -1)
            continue;
        float cost = distValue[nodeId] + e.cost;
        // ключевой момент
        cost += h(node.vertex, e.vertex);
        // дальнейшая реализация описывается ниже
    }
```

7. Расширим список исследуемых узлов (границ) и изменим стоимости, если это необходимо:

```
if (cost < distValue[e.vertex.id])
{
    distValue[eId] = cost;
    previous[eId] = nodeId;
    frontier.Remove(e);
    child = new Edge(e.vertex, cost);
    frontier.Add(child);
}
```

Как это работает...

Алгоритм A* работает аналогично алгоритму Дейкстры. Однако вместо узлов с действительно низкой стоимостью он выбирает наиболее перспективные, основываясь на заданной эвристической функции. В нашем случае в качестве эвристики по умолчанию используется эвклидово расстояние между вершинами и дополнительно имеется возможность использовать манхэттенское расстояние.

Дополнительная информация...

Поэкспериментируйте с различными эвристическими функциями, наиболее подходящими для игры и контекста. Пример ниже демонстрирует, как это сделать:

1. Определим эвристическую функцию в классе `Graph`:

```
public float Heuristic(Vertex a, Vertex b)
{
float estimation = 0f;
// здесь помещается логика
return estimation;
}
```

Здесь важно то, чтобы разрабатываемая эвристика являлась *приемлемой* и *последовательной*. Теоретические обоснования по этой теме

можно найти в книге Рассела (Russel) и Норвига (Norvig) «*Artificial Intelligence: A Modern Approach*»[1].

Метод `BuildPath` здесь не реализован – его реализация приводится в рецепте поиска в глубину.

Полезные ссылки

- Рецепт «*Поиск кратчайшего пути с помощью алгоритма Дейкстры*».
- Рецепт «*Поиск выхода из лабиринта с помощью алгоритма DFS*».

Более подробную информацию о делегатах можно найти в официальной электронной документации на странице https://unity3d.com/learn/tutorials/modules/intermediate/scripting/delegates.

Улучшенный алгоритм A* с меньшим использованием памяти – алгоритм IDA*

Алгоритм IDA* – один из вариантов алгоритма поиска в глубину с итеративным углублением. Он требует меньше памяти, чем алгоритм A*, поскольку позволяет не хранить структуры данных посещавшихся и вновь обнаруженных узлов.

Подготовка

Для реализации этого рецепта важно понимать, как работает рекурсия.

Как это реализовать...

Это длинный рецепт, поэтому разобьем его на два этапа: создание главной и внутренней рекурсивных функций. Обращайте внимание на комментарии, чтобы лучше разобраться в его структуре:

1. Начнем с определения главной функции `GetPathIDAstar`:

```
public List<Vertex> GetPathIDAstar(GameObject srcObj,
                                   GameObject dstObj, Heuristic h = null)
{
    if (srcObj == null || dstObj == null)
        return new List<Vertex>();
    if (ReferenceEquals(h, null))
        h = EuclidDist;
    // дальнейшая реализация описывается ниже
}
```

[1] *Рассел С., Норвиг П.* Искусственный интеллект: современный подход. 2-е изд. М.: Вильямс, 2015. ISBN: 978-5-8459-1968-7. – *Прим. ред.*

2. Объявим и инициализируем переменные, используемые алгоритмом:

```
List<Vertex> path = new List<Vertex>();
Vertex src = GetNearestVertex(srcObj.transform.position);
Vertex dst = GetNearestVertex(dstObj.transform.position);
Vertex goal = null;
bool[] visited = new bool[vertices.Count];
for (int i = 0; i < visited.Length; i++)
    visited[i] = false;
visited[src.id] = true;
```

3. Реализуем цикл алгоритма:

```
float bound = h(src, dst);
while (bound < Mathf.Infinity)
{
    bound = RecursiveIDAstar(src, dst, bound, h, ref goal,
                             ref visited);
}
if (ReferenceEquals(goal, null))
    return path;
return BuildPath(goal);
```

4. Теперь определим внутреннюю рекурсивную функцию:

```
private float RecursiveIDAstar(
        Vertex v,
        Vertex dst,
        float bound,
        Heuristic h,
        ref Vertex goal,
        ref bool[] visited)
{
    // дальнейшая реализация описывается ниже
}
```

5. Подготовим все для начала рекурсии:

```
// базовый случай
if (ReferenceEquals(v, dst))
    return Mathf.Infinity;
Edge[] edges = GetEdges(v);
if (edges.Length == 0)
    return Mathf.Infinity;
```

6. Применим рекурсию к каждому из соседей:

```
// рекурсивный случай
float fn = Mathf.Infinity;
foreach (Edge e in edges)
{
    int eId = e.vertex.id;
    if (visited[eId])
        continue;
    visited[eId] = true;
    e.vertex.prev = v;
    float f = h(v, dst);
    float b;
    if (f <= bound)
    {
        b = RecursiveIDAstar(e.vertex, dst, bound, h, ref goal,
                             ref visited);
        fn = Mathf.Min(f, b);
    }
    else
        fn = Mathf.Min(fn, f);
}
```

7. Вернем значение, полученное в ходе рекурсии:

```
return fn;
```

Как это работает...

Как видите, алгоритм очень похож на рекурсивную версию поиска в глубину (DFS), но использует принцип принятия решений на основе эвристики из алгоритма A*. Основная функция отвечает за запуск рекурсии и построение пути. Рекурсивная функция отвечает за обход графа и поиск узла назначения.

Что дальше...

На этот раз необходимо реализовать другую функцию `BuildPath`, отличающуюся от применяемой в предыдущих рецептах поиска пути. Поэтому реализуем этот метод, который пока не определен:

```
private List<Vertex> BuildPath(Vertex v)
{
    List<Vertex> path = new List<Vertex>();
    while (!ReferenceEquals(v, null))
```

```
    {
        path.Add(v);
        v = v.prev;
    }
    return path;
}
```

Планирование навигации на несколько кадров вперед: поиск с квантованием времени

Расчет пути в больших графах может занять много времени и даже приостановить игру на пару секунд. Это, как минимум, может разрушить целостность восприятия. К счастью, имеется достаточно методов избежать этого.

Этот рецепт основывается на использовании сопрограмм для предотвращения задержек в игре и позволяет выполнять поиск в фоновом режиме. Для его реализации необходимо понимать особенности работы сопрограмм.

Подготовка

Рассмотрим реализацию поиска пути с использованием сопрограмм на примере модификации алгоритма A*, описанного выше, преобразовав сигнатуру его функции.

Как это реализовать...

Несмотря на то что рецепт содержит только определение функции, обратите внимание на комментарии в коде, чтобы лучше разобраться в его структуре:

1. Изменим класс `Graph` и добавим в него несколько свойств. Одно – для хранения пути, и другое – для признака завершения сопрограммы:

```
public List<Vertex> path;
public bool isFinished;
```

2. Определим метод:

```
public IEnumerator GetPathInFrames(GameObject srcObj,
                                   GameObject dstObj, Heuristic h = null)
{
```

```
    // дальнейшая реализация описывается ниже
}
```

3. Добавим в начало инициализацию следующих свойств:

```
isFinished = false;
path = new List<Vertex>();
if (srcObj == null || dstObj == null)
{
    path = new List<Vertex>();
    isFinished = true;
    yield break;
}
```

4. Изменим цикл обхода графа:

```
while (frontier.Count != 0)
{
    // отличается от алгоритма A*
    yield return null;
    ////////////////////////////
    node = frontier.Remove();
```

5. Кроме того, включим еще проверки извлекаемого пути:

```
if (ReferenceEquals(node.vertex, dst))
{
    // отличается от алгоритма A*
    path = BuildPath(src.id, node.vertex.id, ref previous);
    break;
    ////////////////////////////
}
```

6. Наконец, сбросим нужные значения и вернем управление в конец функции после завершения основного цикла:

```
isFinished = true;
yield break;
```

Как это работает...

Оператор `yeld return null` внутри основного цикла действует как флаг для передачи управления функции более высокого уровня. Так с помощью внутренней поддержки многозадачности в Unity в каждом новом кадре выполняется новый цикл.

Полезные ссылки

- Рецепт *«Поиск оптимального пути с помощью алгоритма A*»*.

Более подробную информацию о сопрограммах и примеры их использования можно найти в официальной электронной документации по адресам:

- http://docs.unity3d.com/Manual/Coroutines.html;
- https://unity3d.com/learn/tutorials/modules/intermediate/scripting/coroutines.

Сглаживание маршрута

При использовании графов с равномерно распределенными вершинами, таких как сети, перемещения агентов в игре обычно выглядят *механистическими*. В некоторых видах игр этого можно избежать, применив сглаживание пути, как описывается ниже.

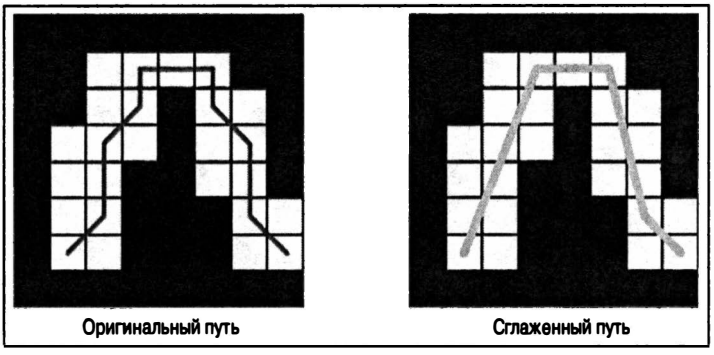


Рис. 2.3

Подготовка

Определим в редакторе Unity новый тег `Wall` и присвоим его всем объектам в сцене, которые играют роль стен и прочих препятствий.

Как это реализовать...

Это простая, но очень мощная функция:

1. Определим функцию сглаживания `Smooth`:

```
public List<Vertex> Smooth(List<Vertex> path)
{
    // дальнейшая реализация описывается ниже
}
```

2. Проверим, стоит ли просчитывать новый путь:

```
List<Vertex> newPath = new List<Vertex>();
if (path.Count == 0)
    return newPath;
if (path.Count < 3)
    return path;
```

3. Обойдем список и построим новый путь:

```
newPath.Add(path[0]);
int i, j;
for (i = 0; i < path.Count - 1;)
{
    for (j = i + 1; j < path.Count; j++)
    {
        // дальнейшая реализация описывается ниже
    }
    i = j - 1;
    newPath.Add(path[i]);
}
return newPath;
```

4. Объявим и инициализируем переменные, используемые функцией для отбрасывания лучей:

```
Vector3 origin = path[i].transform.position;
Vector3 destination = path[j].transform.position;
Vector3 direction = destination - origin;
float distance = direction.magnitude;
bool isWall = false;
direction.Normalize();
```

5. Отбросим луч от текущего начального узла к следующему:

```
Ray ray = new Ray(origin, direction);
RaycastHit[] hits;
hits = Physics.RaycastAll(ray, distance);
```

6. Если луч пересекает стену, прервем цикл:

```
foreach (RaycastHit hit in hits)
{
    string tag = hit.collider.gameObject.tag;
    if (tag.Equals("Wall"))
    {
        isWall = true;
```

```
            break;
        }
    }
        if (isWall)
    break;
```

Как это работает...

При построении нового пути в качестве отправной точки принимается начальный узел, из него отбрасывается луч к следующему узлу пути, и так повторяется, пока луч не пересечет стену. Когда это произойдет, выбирается предыдущий узел в качестве следующего узла для нового пути и отправной точки для обхода выбранного ранее узла, пока не будут проверены все узлы. Таким способом определяется более интуитивный путь.

Глава 3

Принятие решений

В этой главе рассматриваются следующие рецепты:

- выбор с помощью дерева принятия решений;
- работа конечного автомата;
- усовершенствование конечного автомата: иерархические конечные автоматы;
- комбинирование конечных автоматов и деревьев принятия решений;
- реализация деревьев моделей поведения;
- работа с нечеткой логикой;
- представление состояний с помощью числовых значений: система Маркова;
- принятие решений в моделях целенаправленного поведения.

Введение

Принятие решений или изменение течения игры на основании ее состояний может стать весьма хаотичным, если полагаться лишь на простые управляющие структуры. Поэтому рассмотрим различные методы принятия решений, достаточно гибкие для адаптации к различным типам игр и достаточно надежные, чтобы сделать возможным построение модульных систем принятия решений.

Методы, представленные в этой главе, в основном базируются на деревьях, автоматах и матрицах. Кроме того, некоторые из них требуют хорошего понимания рекурсии, наследования и полиморфизма, поэтому имеет смысл ознакомиться с этими темами.

Выбор с помощью дерева принятия решений

Деревья принятия решений считаются одним из простейших механизмов благодаря наглядности и простоте реализации. Как следствие сегодня это один из наиболее часто используемых методов. Он широко используется в других областях управления персонажами, таких как анимация.

Подготовка

Этот рецепт требует хорошего знания рекурсии и наследования, поскольку в его реализации используются виртуальные функции.

Как это реализовать...

Реализация этого рецепта требует большой внимательности из-за большого количества обрабатываемых файлов. Здесь мы создадим родительский класс DecisionTreeNode, который наследуют все остальные. Далее рассмотрим реализацию нескольких узлов для стандартных решений:

1. Во-первых, создадим родительский класс DecisionTreeNode:

```
using UnityEngine;
using System.Collections;
public class DecisionTreeNode : MonoBehaviour
{
    public virtual DecisionTreeNode MakeDecision()
    {
        return null;
    }
}
```

2. Создадим псевдоабстрактный класс Decision, наследующий родительский класс DecisionTreeNode:

```
using UnityEngine;
using System.Collections;
public class Decision : DecisionTreeNode
{
    public Action nodeTrue;
    public Action nodeFalse;

    public virtual Action GetBranch()
    {
```

```
            return null;
        }
    }
```

3. Определим псевдоабстрактный класс Action:

```
using UnityEngine;
using System.Collections;
public class Action : DecisionTreeNode
{
    public bool activated = false;

    public override DecisionTreeNode MakeDecision()
    {
        return this;
    }
}
```

4. Реализуем виртуальную функцию LateUpdate:

```
public virtual void LateUpdate()
{
    if (!activated)
        return;
    // поместите сюда реализацию конкретных моделей поведения
}
```

5. Определим законченный класс DecisionTree:

```
using UnityEngine;
using System.Collections;
public class DecisionTree : DecisionTreeNode
{
    public DecisionTreeNode root;
    private Action actionNew;
    private Action actionOld;
}
```

6. Переопределим функцию MakeDecision:

```
public override DecisionTreeNode MakeDecision()
{
    return root.MakeDecision();
}
```

7. Наконец, реализуем функцию Update:

```
void Update()
{
    actionNew.activated = false;
```

```
        actionOld = actionNew;
        actionNew = root.MakeDecision() as Action;
        if (actionNew == null)
            actionNew = actionOld;
        actionNew.activated = true;
    }
```

Как это работает...

Выбор пути в узлах дерева решений выполняется вызовом рекурсивной функции MakeDecision. Обратите внимание, что элементы верхнего уровня дерева (рис. 3.1) являются решениями, а нижнего уровня – действиями. При создании дерева следует проявить особую осторожность, чтобы в нем не образовались циклы.

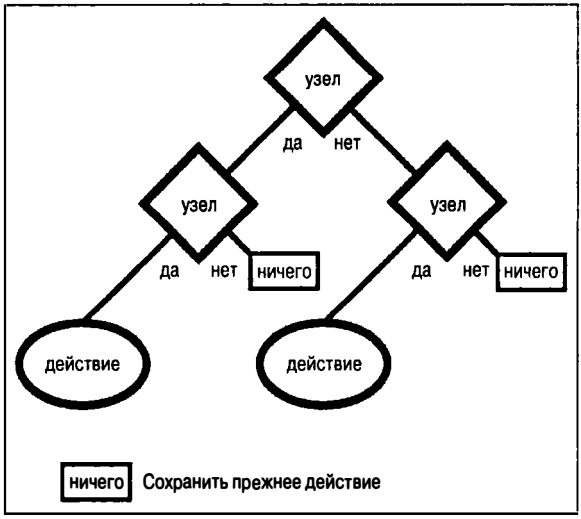


Рис. 3.1 ❖ Дерево решений

Дополнительная информация...

Созданные ранее псевдоабстрактные классы послужат основой пользовательских решений и действий. Например, решение начать атаку на игрока или убежать от него.

Пример пользовательского решения:

```
using UnityEngine;
using System.Collections;
```

```
public class DecisionBool : Decision
{
    public bool valueDecision;
    public bool valueTest;

    public override Action GetBranch()
    {
        if (valueTest == valueDecision)
            return nodeTrue;
        return nodeFalse;
    }
}
```

Работа конечного автомата

Другой интересный и простой прием основан на использовании **конечных автоматов (FSM)**. Для его реализации требуется взглянуть на проблему под другим углом, не как в предыдущем рецепте. Конечные автоматы хороши, когда необходимо ориентироваться на конкретные события, а модель поведения должна оставаться прежней, пока не изменятся условия.

Подготовка

Такой механизм главным образом базируется на моделях поведения автоматов и закладывает фундамент для следующего рецепта, который станет улучшенной версией текущего.

Как это реализовать...

Этот рецепт включает последовательную реализацию трех классов, смысл использования которых станет ясен на заключительном этапе:

1. Класс `Condition`:

```
public class Condition
{
    public virtual bool Test()
    {
        return false;
    }
}
```

2. Класс `Transition`:

```
public class Transition
{
```

```
    public Condition condition;
    public State target;
}
```

3. И класс State:

```
using UnityEngine;
using System.Collections.Generic;

public class State : MonoBehaviour
{
    public List<Transition> transitions;
}
```

4. Реализуем функцию Awake:

```
public virtual void Awake()
{
    transitions = new List<Transition>();
    // здесь помещается настройка переходов
```

5. Определим функцию инициализации:

```
public virtual void OnEnable()
{
    // здесь выполняется инициализация состояния
}
```

6. Определим функцию завершения:

```
public virtual void OnDisable()
{
    // здесь выполняется финализация состояния
}
```

7. Определим функцию реализации модели поведения, соответствующей состоянию:

```
public virtual void Update()
{
    // Будет реализована ниже
    // здесь определяется модель поведения
}
```

8. Реализуем функцию, определяющую состояние для следующего шага:

```
public void LateUpdate()
{
    foreach (Transition t in transitions)
```

```
        {
            if (t.condition.Test())
            {
                t.target.enabled = true;
                this.enabled = false;
                return;
            }
        }
    }
```

Как это работает...

Каждое состояние – это сценарий MonoBehaviour, который выполняет или не выполняет переход к следующему состоянию (рис. 3.2). Воспользуемся функцией LateUpdate, чтобы не менять привычного подхода к разработке моделей поведения, и реализуем в ней проверку необходимости перехода в другое состояние. Важно отключать каждое состояние в объекте игры, кроме начального.

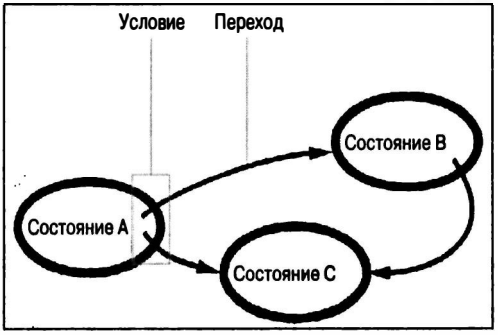


Рис. 3.2 ❖ Переходы между состояниями

Дополнительная информация...

Для иллюстрации разработки дочерних классов, наследующих класс Condition, рассмотрим несколько примеров: один проверяет вхождение значения в диапазон, а второй проверяет соблюдение двух условий:

Ниже приводится определение первого класса – ConditionFloat:

```
using UnityEngine;
using System.Collections;

public class ConditionFloat : Condition
```

```
{
    public float valueMin;
    public float valueMax;
    public float valueTest;
    public override bool Test()
    {
        if (valueMax >= valueTest && valueTest >= valueMin)
            return true;
        return false;
    }
}
```

И второго класса ConditionAnd:

```
using UnityEngine;
using System.Collections;

public class ConditionAnd : Condition
{
    public Condition conditionA;
    public Condition conditionB;
    public override bool Test()
    {
        if (conditionA.Test() && conditionB.Test())
            return true;
        return false;
    }
}
```

Усовершенствование конечного автомата: иерархические конечные автоматы

Конечные автоматы можно усовершенствовать, введя поддержку слоев или иерархий. Сам принцип остается тем же, но состояния получают возможность включать собственные конечные автоматы, что делает их более гибкими и масштабируемыми.

Подготовка

В основу этого рецепта положен предыдущий рецепт, поэтому для его понимания важно знать, как работает рецепт конечных автоматов.

Как это реализовать...

Создадим состояние, способное содержать внутренние состояния, для разработки многоуровневых иерархических автоматов:

1. Определим класс StateHighLevel, наследующий класс State:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class StateHighLevel : State
{
}
```

2. Добавим свойства для управления внутренними состояниями:

```
public List<State> states;
public State stateInitial;
protected State stateCurrent;
```

3. Переопределим функцию инициализации:

```
public override void OnEnable()
{
    if (stateCurrent == null)
        stateCurrent = stateInitial;
    stateCurrent.enabled = true;
}
```

4. Переопределим функцию финализации:

```
public override void OnDisable()
{
    base.OnDisable();
    stateCurrent.enabled = false;
    foreach (State s in states)
    {
        s.enabled = false;
    }
}
```

Как это работает...

Класс состояния верхнего уровня позволяет активировать внутренние конечные автоматы, рекурсивно изменяя его внутренние состояния. Принцип работы остается неизменным благодаря списку состояний и подходу родительского класса к выполнению переходов.

Полезные ссылки

Естественно, ссылка на рецепт «*Работа конечного автомата*».

Комбинирование конечных автоматов и деревьев принятия решений

Предыдущие рецепты отличаются удивительной простотой, поэтому мы без труда сможем объединить их и получить мощную систему принятия решений, обладающую достоинствами обоих подходов, что позволит успешно применять ее в самых разных обстоятельствах.

Подготовка

Рассмотрим, как модифицировать и расширить дочерние классы для создания конечного автомата, способного выполнять сложные переходы на основе дерева принятия решений.

Как это реализовать...

Этот рецепт основан на создании пары дочерних классов, наследующих класс, описанный выше:

1. Определим новый класс действий со ссылкой на состояние:

```
using UnityEngine;
using System.Collections;

public class ActionState : DecisionTreeNode
{
    public State state;

    public override DecisionTreeNode MakeDecision()
    {
        return this;
    }
}
```

2. Реализуем класс переходов, хранящий дерево решений:

```
using UnityEngine;
using System.Collections;

public class TransitionDecision : Transition
{
    public DecisionTreeNode root;

    public State GetState()
    {
        ActionState action;
        action = root.MakeDecision() as ActionState;
        return action.state;
    }
}
```

3. Изменим функцию LateUpdate в классе State для поддержки переходов обоих видов:

```
public void LateUpdate()
{
    foreach (Transition t in transitions)
    {
        if (t.condition.Test())
        {
                State target;
            if (t.GetType().Equals(typeof(TransitionDecision)))

                TransitionDecision td = t as TransitionDecision;
                target = td.GetState();
            }
            else
                target = t.target;
            target.enabled = true;
            this.enabled = false;
            return;
        }
    }
}
```

Как это работает...

Изменения в классе State позволяют ему обрабатывать новый вид переходов. Новые дочерние классы созданы специально, чтобы обхитрить обе системы и получить в результате нужный узел действия, который сам ничего не делает, но возвращает новое состояние, которое активируется после выбора с помощью дерева принятия решений.

Полезные ссылки

Ссылки на следующие рецепты:

- *«Выбор с помощью дерева принятия решений»*;
- *«Работа конечного автомата»*.

Реализация деревьев моделей поведения

Деревья моделей поведения можно рассматривать как синтез целого ряда других методов искусственного интеллекта, например конечных автоматов, планирования и деревьев принятия решений. На самом деле они имеют некоторое сходство с конечными автоматами, только вместо состояний в них используются действия, заключенные в древовидную структуру.

Подготовка

Этот рецепт требует знания сопрограмм.

Как это реализовать...

По аналогии с реализацией дерева принятия решений определим три псевдоабстрактных класса:

1. Определим базовый класс Task:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Task : MonoBehaviour
{
    public List<Task> children;
    protected bool result = false;
    protected bool isFinished = false;
}
```

2. Реализуем функцию финализации:

```
public virtual void SetResult(bool r)
{
    result = r;
    isFinished = true;
}
```

3. Реализуем функцию для создания моделей поведения:

```
public virtual IEnumerator Run()
{
    SetResult(true);
    yield break;
}
```

4. Реализуем общую функцию для запуска поведения:

```
public virtual IEnumerator RunTask()
{
    yield return StartCoroutine(Run());
}
```

5. Определим класс ConditionBT:

```
using UnityEngine;
using System.Collections;

public class ConditionBT : Task
```

```
{
    public override IEnumerator Run()
    {
        isFinished = false;
        bool r = false;
        // реализуйте здесь свою модель поведения,
        // возвращающую в r значение true или false
        //---------
        SetResult(r);
        yield break;
    }
}
```

6. Определим базовый класс действий:

```
using UnityEngine;
using System.Collections;

public class ActionBT : Task
{
    public override IEnumerator Run()
    {
        isFinished = false;
        // реализуйте здесь свою модель поведения
        //---------
        return base.Run();
    }
}
```

7. Определим класс Selector:

```
using UnityEngine;
using System.Collections;

public class Selector : Task
{
    public override void SetResult(bool r)
    {
        if (r == true)
            isFinished = true;
    }

    public override IEnumerator RunTask()
    {
        foreach (Task t in children)
            yield return StartCoroutine(t.RunTask());
    }
}
```

8. Определим также класс Sequence:

```
using UnityEngine;
using System.Collections;

public class Sequence : Task
{
    public override void SetResult(bool r)
    {
        if (r == true)
            isFinished = true;
    }

    public override IEnumerator RunTask()
    {
        foreach (Task t in children)
            yield return StartCoroutine(t.RunTask());
    }
}
```

Как это работает...

Деревья моделей поведения действуют подобно деревьям принятия решений. Однако здесь конечные узлы называются задачами, и некоторые узлы не являются условиями. Запустить набор задач можно одним из двух способов: с помощью селектора или последовательности. Селекторы запускают набор задач и возвращают значение `true`, если одна из задач вернет `true`. Их можно рассматривать как узлы ИЛИ. Последовательности выполняют набор задач и возвращают значение `true`, если все задачи вернули `true`. Их можно рассматривать как узлы И.

Полезные ссылки

Познакомиться с теорией по этой теме можно в книге Яна Миллингтона (Ian Millington) «*Artificial Intelligence for Games*».

Работа с нечеткой логикой

Иногда приходится иметь дело с серыми областями, где решения принимаются не на основе альтернативных значений. В таких случаях на выручку приходит нечеткая логика – набор математических методов, помогающих справиться с подобными задачами.

Представьте, что вы разрабатываете автопилот, которому доступны два действия: управление рулевым колесом и скоростью движения, каждое из которых определяется рядом значений. Автопилот должен

выбрать, куда повернуть и какую при этом поддерживать скорость. Это и есть один из видов серых областей, представлять и обслуживать которые призвана нечеткая логика.

Подготовка

Для этого рецепта потребуется определить ряд состояний, пронумерованных последовательными целыми числами. Поскольку представление зависит от конкретной игры, займемся обработкой исходных данных таких состояний, а также их *размыванием* (*fuzzification*), чтобы получить генератор нечетких решений общего назначения. Генератор будет возвращать набор нечетких значений, отражающих степень принадлежности каждого состояния.

Как это реализовать...

Создадим два базовых класса и генератор нечетких решений:

1. Родительский класс `MembershipFunction`:

```
using UnityEngine;
using System.Collections;

public class MembershipFunction : MonoBehaviour
{
    public int stateId;
    public virtual float GetDOM(object input)
    {
        return 0f;
    }
}
```

2. Класс `FuzzyRule`:

```
using System.Collections;
using System.Collections.Generic;

public class FuzzyRule
{
    public List<int> stateIds;
    public int conclusionStateId;
}
```

3. И класс `FuzzyDecisionMaker`:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

```
public class FuzzyDecisionMaker : MonoBehaviour
{
}
```

4. Определим сигнатуру функции принятия решений и ее свойства:

```
public Dictionary<int,float> MakeDecision(object[] inputs,
    MembershipFunction[][] mfList, FuzzyRule[] rules)
{
    Dictionary<int, float> inputDOM = new Dictionary<int, float>();
    Dictionary<int, float> outputDOM = new Dictionary<int, float>();
    MembershipFunction memberFunc;
    // дальнейшая реализация описывается ниже
}
```

5. Реализуем цикл для обхода исходных данных и заполнения начальной степени принадлежности (Degree of Membership, DOM) каждого состояния:

```
foreach (object input in inputs)
{
    int r, c;
    for (r = 0; r < mfList.Length; r++)
    {
        for (c = 0; c < mfList[r].Length; c++)
        {
            // дальнейшая реализация описывается ниже
        }
    }
}
// дальнейшая реализация описывается ниже
```

6. Определим тело самого внутреннего цикла, вызывающего функцию спецификации принадлежности:

```
memberFunc = mfList[r][c];
int mfId = memberFunc.stateId;
float dom = memberFunc.GetDOM(input);
if (!inputDOM.ContainsKey(mfId))
{
    inputDOM.Add(mfId, dom);
    outputDOM.Add(mfId, 0f);
}
else
    inputDOM[mfId] = dom;
```

7. Обойдем правила для настройки исходящих степеней принадлежности:

```
foreach (FuzzyRule rule in rules)
{
    int outputId = rule.conclusionStateId;
    float best = outputDOM[outputId];
    float min = 1f;
    foreach (int state in rule.stateIds)
    {
        float dom = inputDOM[state];
        if (dom < best)
            continue;
        if (dom < min)
            min = dom;
    }
    outputDOM[outputId] = min;
}
```

8. И наконец, вернем набор степеней принадлежности:

```
return outputDOM;
```

Как это работает...

Здесь для обработки входных данных в виде данных объектного типа используется механизм упаковки/распаковки (boxing/unboxing). Процесс *размывания* (*fuzzification*) осуществляется с помощью пользовательских функций спецификации принадлежности, унаследованных от базового класса, созданного вначале. Затем берется минимальная степень принадлежности входного состояния для каждого правила, и вычисляется исходящая степень принадлежности для каждого исходящего состояния с учетом максимального исходящего состояния всех применимых правил.

Дополнительная информация...

Можно попробовать создать пример функции, помогающей определить – находится ли противник в ярости, зная, что уровень его здоровья (варьируется в диапазоне от 0 до 100) меньше или равен 30:

```
using UnityEngine;
using System;
using System.Collections;

public class MFEnraged : MembershipFunction
{
```

```
    public override float GetDOM(object input)
    {
        if ((int)input <= 30)
            return 1f;
        return 0f;
    }
}
```

Обратите внимание, что общим требованием является наличие полного набора правил, по одному для каждой комбинации состояний из каждого набора входных данных. Это ведет к потере масштабируемости, но хорошо подходит для небольшого количества входных переменных и небольшого количества состояний на одну переменную.

Полезные ссылки

Познакомиться с теорией *размывания* и проблемами недостаточной масштабируемости можно в книге Яна Миллингтона (Ian Millington) «*Artificial Intelligence for Games*».

Представление состояний с помощью числовых значений: система Маркова

После знакомства с нечеткой логикой добавим несколько способов расширения функциональности конечных автоматов. Однако нечеткая логика не работает непосредственно со значениями – они должны быть *размыты* для применения в этой области. Цепи Маркова – это математическая система, позволяющая разработать систему принятия решений, которую можно рассматривать как автомат нечетких состояний.

Подготовка

Этот рецепт использует классы матриц и векторов из Unity для иллюстрации теоретического подхода и подготовки рабочего примера, который можно улучшить с применением пользовательских классов матриц и векторов, поддерживающих обязательные методы, такие как умножение векторов и матриц.

Как это реализовать...

1. Создадим родительский класс для обработки переходов:

```
using UnityEngine;
using System.Collections;
```

```
public class MarkovTransition : MonoBehaviour
{
    public Matrix4x4 matrix;
    public MonoBehaviour action;
}
```

2. Реализуем метод IsTriggered:

```
public virtual bool IsTriggered()
{
    // здесь находится реализация
    return false;
}
```

3. Определим автомат состояний Маркова вместе с его свойствами:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class MarkovStateMachine : MonoBehaviour
{
    public Vector4 state;
    public Matrix4x4 defaultMatrix;
    public float timeReset;
    public float timeCurrent;
    public List<MarkovTransition> transitions;
    private MonoBehaviour action;
}
```

4. Определим функцию Start для инициализации:

```
void Start()
{
    timeCurrent = timeReset;
}
```

5. Реализуем функцию Update:

```
void Update()
{
    if (action != null)
        action.enabled = false;

    MarkovTransition triggeredTransition;
    triggeredTransition = null;
    // дальнейшая реализация описывается ниже
}
```

6. Найдем требуемый переход:

```
foreach (MarkovTransition mt in transitions)
{
    if (mt.IsTriggered())
    {
        triggeredTransition = mt;
        break;
    }
}
```

7. Если поиск увенчался успехом, расчитаем матрицу состояния игры:

```
if (triggeredTransition != null)
{
    timeCurrent = timeReset;
    Matrix4x4 matrix = triggeredTransition.matrix;
    state = matrix * state;
    action = triggeredTransition.action;
}
```

8. В противном случае обновим таймер обратного отсчета и рассчитаем матрицу состояния игры по умолчанию, если это необходимо:

```
else
{
    timeCurrent -= Time.deltaTime;
    if (timeCurrent <= 0f)
    {
        state = defaultMatrix * state;
        timeCurrent = timeReset;
        action = null;
    }
}
```

Как это работает...

Мы определили состояние игры, исходя из числового значения свойства, представленного 4-мерным вектором, каждый элемент которого соответствует одному состоянию. Значения состояния игры изменяются при любом переходе, в соответствии с присоединенной матрицей. Когда срабатывает механизм перехода, состояние игры изменяется, при этом мы используем таймер обратного отсчета для выполнения перехода по умолчанию и соответствующего изменения

игры. Это может пригодиться, когда нужно сбросить состояние игры после истечения определенного периода времени или просто регулярно применять преобразования.

Полезные ссылки

Познакомиться с теорией применения процессов Маркова в играх с искусственным интеллектом можно в книге Яна Миллингтона (Ian Millington) «*Artificial Intelligence for Games*».

Принятие решений в моделях целенаправленного поведения

Модели целенаправленного поведения позволяют дать агентам не только интеллект, но и определенную свободу выбора, основанную на цели и заданном наборе правил.

Представьте, что мы разрабатываем модель поведения воина, который должен не только добраться до вражеского флага (основная цель), но и позаботиться о сохранении своей жизни и боеприпасов (внутренние цели для достижения главной цели). Для реализации этой модели поведения можно использовать универсальный алгоритм обработки целей, такой, чтобы агент мог проявить нечто, похожее на свободу выбора.

Подготовка

Далее мы рассмотрим создание механизма выбора (селектора) целенаправленных действий, осуществляющего выбор с учетом главной цели, избегающего посторонних действий, ведущих к неудаче, и учитывающего длительность действий. Так же как в предыдущем рецепте, здесь потребуется моделировать цели с помощью числовых значений.

Как это реализовать...

Кроме селектора, определим также базовые классы для действий и целей:

1. Определим базовый класс действий:

```
using UnityEngine;
using System.Collections;

public class ActionGOB : MonoBehaviour
{
```

```
    public virtual float GetGoalChange(GoalGOB goal)
    {
        return 0f;
    }

    public virtual float GetDuration()
    {
        return 0f;
    }
}
```

2. Определим родительский класс целей GoalGOB с методами:

```
using UnityEngine;
using System.Collections;

public class GoalGOB
{
    public string name;
    public float value;
    public float change;
}
```

3. Определим соответствующие функции вычисления оценки неудовлетворительности и изменений с течением времени:

```
public virtual float GetDiscontentment(float newValue)
{
    return newValue * newValue;
}

public virtual float GetChange()
{
    return 0f;
}
```

4. Определим класс ActionChooser:

```
using UnityEngine;
using System.Collections;

public class ActionChooser : MonoBehaviour
{
}
```

5. Реализуем функцию обработки ненужных действий:

```
public float CalculateDiscontentment(ActionGOB action, GoalGOB[] goals)
{
    float discontentment = 0;
```

```
    foreach (GoalGOB goal in goals)
    {
        float newValue = goal.value + action.GetGoalChange(goal);
        newValue += action.GetDuration() * goal.GetChange();
        discontentment += goal.GetDiscontentment(newValue);
    }
    return discontentment;
}
```

6. Реализуем функцию выбора действия:

```
public ActionGOB Choose(ActionGOB[] actions, GoalGOB[] goals)
{
    ActionGOB bestAction;
    bestAction = actions[0];
    float bestValue = CalculateDiscontentment(actions[0], goals);
    float value;
    // дальнейшая реализация описывается ниже
}
```

7. Выбор лучшего действия зависит от оценки наименьшего ущерба:

```
foreach (ActionGOB action in actions)
{
    value = CalculateDiscontentment(action, goals);
    if (value < bestValue)
    {
        bestValue = value;
        bestAction = action;
    }
}
```

8. Вернем наилучшее действие:

```
return bestAction;
```

Как это работает...

Функции неудовлетворительности помогают избежать ненужных действий в зависимости от степени отклонения от цели и основываются на понятиях действия и времени, необходимого для его выполнения. Функция выбора определяет самое оптимальное действие с точки зрения минимального ущерба (неудовлетворительность).

Глава 4

Координирование и тактика

В этой главе рассматриваются методы координации и выработки тактики:

- обработка групповых формирований;
- расширение алгоритма A* для координации: алгоритм A*mbush;
- выбор удобных точек позиций;
- анализ точек позиций по их высоте;
- анализ точек позиций по обзорности и незаметности;
- оценка точек позиций для принятия решения;
- карты влияния;
- улучшение карт влияния путем заполнения;
- улучшение карт влияния с помощью фильтров свертки;
- построение боевых кругов.

Введение

Эта глава не ограничивается одним направлением, в ней содержатся не только оригинальные рецепты, но и описывается применение знаний, полученных при реализации предыдущих рецептов, для создания совершенно новых или усовершенствованных приемов.

В этой главе рассматриваются различные рецепты координации действий агентов, превращающие их в единый организм, и методы принятия тактических решений, основанные на графах (например, точки позиций) и картах влияния. В этих методах используются различные элементы из предыдущих глав и рецептов, особенно это касается алгоритмов построения графов и алгоритмов выбора пути из *главы 2 «Навигация»*.

Обработка формирований

Здесь рассматривается ключевой алгоритм создания групп военных агентов. Он обладает достаточной гибкостью, чтобы позволить создавать собственные групповые формирования.

Результатом этого рецепта является набор целевых позиций и поворотов для каждого агента в формировании. Вам останется только разработать алгоритмы перемещения агентов к полученным целям.

Для этого можно использовать алгоритмы, описанные *в главе 1 «Интеллектуальные модели поведения: перемещение»*.

Подготовка

Здесь мы определим три базовых класса – три типа данных для использования классами верхнего уровня и алгоритмами. Класс `Location` очень похож на класс `Steering` и используется для определения целевой позиции и поворота, с учетом опорной точки и поворота формирования. Класс `SlotAssignment` – это тип данных, определяющий список соответствий индексов и агентов. И наконец, класс компонента `Character` хранит целевой класс `Location`.

Ниже приводится код класса `Location`:

```
using UnityEngine;
using System.Collections;

public class Location
{
    public Vector3 position;
    public Quaternion rotation;
    public Location ()
    {
        position = Vector3.zero;
        rotation = Quaternion.identity;
    }

    public Location(Vector3 position, Quaternion rotation)
    {
        this.position = position;
        this.rotation = rotation;
    }
}
```

класса SlotAssignment:

```
using UnityEngine;
using System.Collections;

public class SlotAssignment
{
    public int slotIndex;
    public GameObject character;

    public SlotAssignment()
    {
        slotIndex = -1;
        character = null;
    }
}
```

и класса Character:

```
using UnityEngine;
using System.Collections;

public class Character : MonoBehaviour
{
    public Location location;

    public void SetTarget (Location location)
    {
        this.location = location;
    }
}
```

Как это реализовать...

Определим два класса, FormationPattern и FormationManager:

1. Псевдоабстрактный класс FormationPattern:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class FormationPattern: MonoBehaviour
{
    public int numOfSlots;
    public GameObject leader;
}
```

2. Определим функцию Start:

```
void Start()
{
```

```
    if (leader == null)
        leader = transform.gameObject;
}
```

3. Определим функцию, возвращающую позицию заданного слота:

```
public virtual Vector3 GetSlotLocation(int slotIndex)
{
    return Vector3.zero;
}
```

4. Определим функцию, возвращающую признак поддержки формированием заданного количества слотов:

```
public bool SupportsSlots(int slotCount)
{
    return slotCount <= numOfSlots;
}
```

5. Определим функцию установки смещений в местах, где это необходимо:

```
public virtual Location GetDriftOffset(List<SlotAssignment>
slotAssignments)
{
    Location location = new Location();
    location.position = leader.transform.position;
    location.rotation = leader.transform.rotation;
    return location;
}
```

6. Определим класс управления формированием:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class FormationManager : MonoBehaviour
{
    public FormationPattern pattern;
    private List<SlotAssignment> slotAssignments;
    private Location driftOffset;
}
```

7. Реализуем функцию Awake:

```
void Awake()
{
```

```
        slotAssignments = new List<SlotAssignment>();
    }
```

8. Определим функцию обновления индексов слотов в соответствии с порядком их следования в списке:

```
    public void UpdateSlotAssignments()
    {
        for (int i = 0; i < slotAssignments.Count; i++)
        {
            slotAssignments[i].slotIndex = i;
        }
        driftOffset = pattern.GetDriftOffset(slotAssignments);
    }
```

9. Реализуем функцию добавления персонажа в формирование:

```
    public bool AddCharacter(GameObject character)
    {
        int occupiedSlots = slotAssignments.Count;
        if (!pattern.SupportsSlots(occupiedSlots + 1))
            return false;
        SlotAssignment sa = new SlotAssignment();
        sa.character = character;
        slotAssignments.Add(sa);
        UpdateSlotAssignments();
        return true;
    }
```

10. Реализуем функцию удаления персонажа из формирования:

```
    public void RemoveCharacter(GameObject agent)
    {
        int index = slotAssignments.FindIndex(x => x.character.
    Equals(agent));
        slotAssignments.RemoveAt(index);
        UpdateSlotAssignments();
    }
```

11. Реализуем функцию обновления слотов:

```
    public void UpdateSlots()
    {
        GameObject leader = pattern.leader;
        Vector3 anchor = leader.transform.position;
        Vector3 slotPos;
        Quaternion rotation;
```

```
        rotation = leader.transform.rotation;
        foreach (SlotAssignment sa in slotAssignments)
        {
            // дальнейшая реализация описывается ниже
        }
    }
```

12. И наконец, определим тело цикла foreach:

```
Vector3 relPos;
slotPos = pattern.GetSlotLocation(sa.slotIndex);
relPos = anchor;
relPos += leader.transform.TransformDirection(slotPos);
Location charDrift = new Location(relPos, rotation);
Character character = sa.character.GetComponent<Character>();
character.SetTarget(charDrift);
```

Как это работает...

Класс FormationPattern хранит относительные позиции слотов. Например, дочерний класс CircleFormation будет иметь свою функцию GetSlotLocation, возвращающую относительные позиции слотов, размещенных на окружности. Он служит базовым классом, к которому диспетчер FormationManager добавляет слой, управляющий перегруппировкой. Благодаря такой организации разработчик сможет без труда создавать новые виды формирований, наследующие базовый класс.

Класс FormationManager, как уже упоминалось, реализует верхний слой и управляет размещением формирования. Все вычисления в нем основаны на позиции и повороте лидера, к которым применяются необходимые преобразования.

Что дальше...

Стоит отметить, что классы FormationManager и FormationPattern должны быть компонентами одного и того же объекта. Если поле, определяющее лидера, имеет значение null, лидером становится сам объект. То есть лидером можно сделать другой объект, чтобы избавиться от лишних деталей в окне инспектора и обеспечить более ясное разделение классов.

Полезные ссылки

- Рецепт *«Достижение цели и уход от погони»* в главе 1 *«Интеллектуальные модели поведения: перемещение»*.

- Более подробную информацию о дрейфовом смещении и экспериментах с этим значением можно найти в книге Яна Миллингтона (Ian Millington) «*Artificial Intelligence for Games*».

Расширение алгортма A* для координации: алгоритм A*mbush

После знакомства с применением алгоритма A* для поиска пути воспользуемся его мощью и гибкостью для разработки модели скоординированного поведения, а именно устройства засады на игрока. Этот алгоритм особенно полезен, когда требуется недорогое решение упомянутой выше задачи, к тому же он прост в реализации.

Этот рецепт определяет маршрут для каждого агента, его перемещение при устройстве засады в заданной вершине или точке графа.

Подготовка

Создадим специальный компонент для агентов Lurker. Этот класс хранит маршруты, которые в дальнейшем используются в процессе навигации.

Ниже приводится код компонента Lurker:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Lurker : MonoBehaviour
{
    [HideInInspector]
    public List<int> pathIds;
    [HideInInspector]
    public List<GameObject> pathObjs;

    void Awake()
    {
        if (pathIds == null)
            pathIds = new List<int>();
        if (pathObjs == null)
            pathObjs = new List<GameObject>();
    }
}
```

Как это реализовать...

Создадим сначала главную функцию, определяющую перемещения всех агентов при устройстве засады, а затем функцию настройки маршрута для каждого агента.

1. Определим главную функцию для засады:

```
public void SetPathAmbush(GameObject dstObj, List<Lurker> lurkers)
{
    Vertex dst = GetNearestVertex(dstObj.transform.position);
    foreach (Lurker l in lurkers)
    {
        Vertex src = GetNearestVertex(l.transform.position);
        l.path = AStarMbush(src, dst, l, lurkers);
    }
}
```

2. Объявим функцию для нахождения каждого отдельного пути:

```
public List<Vertex> AStarMbush(
        Vertex src,
        Vertex dst,
        Lurker agent,
        List<Lurker> lurkers,
        Heuristic h = null)
{    // дальнейшая реализация описывается ниже
}
```

3. Объявим необходимые переменные для вычисления дополнительных стоимостей:

```
int graphSize = vertices.Count;
float[] extra = new float[graphSize];
float[] costs = new float[graphSize];
int i;
```

4. Инициализируем переменные обычных и дополнительных стоимостей:

```
for (i = 0; i < graphSize; i++)
{
    extra[i] = 1f;
    costs[i] = Mathf.Infinity;
}
```

5. Добавим дополнительные стоимости для всех вершин, которые включены в маршрут другого агента:

```
foreach (Lurker l in lurkers)
{
    foreach (Vertex v in l.path)
    {
        extra[v.id] += 1f;
    }
}
```

6. Объявим и инициализируем переменные для расчетов по алгоритму A*:

```
Edge[] successors;
int[] previous = new int[graphSize];
for (i = 0; i < graphSize; i++)
    previous[i] = -1;
previous[src.id] = src.id;
float cost = 0;
Edge node = new Edge(src, 0);
GPWiki.BinaryHeap<Edge> frontier = new GPWiki.BinaryHeap<Edge>();
```

7. Начнем реализацию основного цикла алгоритма A*:

```
frontier.Add(node);
while (frontier.Count != 0)
{
    if (frontier.Count == 0)
        return new List<GameObject>();
    // дальнейшая реализация описывается ниже
}
return new List<Vertex>();
```

8. Проверим достижение цели. Если цель не достигнута, прервать расчет стоимостей и продолжить выполнение обычного алгоритма A*:

```
node = frontier.Remove();
if (ReferenceEquals(node.vertex, dst))
    return BuildPath(src.id, node.vertex.id, ref previous);
int nodeId = node.vertex.id;
if (node.cost > costs[nodeId])
    continue;
```

9. Обойдем соседние вершины и проверим, посещались ли они прежде:

```
successors = GetEdges(node.vertex);
foreach (Edge e in successors)
{
    int eId = e.vertex.id;
    if (previous[eId] != -1)
        continue;
    // дальнейшая реализация описывается ниже
}
```

10. Если они еще не посещались, добавим их в исследуемую область:

```
cost = e.cost;
cost += costs[dst.id];
cost += h(e.vertex, dst);
if (cost < costs[e.vertex.id])
{
    Edge child;
    child = new Edge(e.vertex, cost);
    costs[eId] = cost;
    previous[eId] = nodeId;
    frontier.Remove(e);
    frontier.Add(child);
}
```

Как это работает...

Алгоритм A*mbush анализирует маршруты всех агентов и увеличивает стоимость их узлов. Благодаря этому, когда агент будет расчитывать свой маршрут с помощью A*, он выберет маршрут, не выбиравшийся ранее другими агентами, таким способом будет создана осмысленная сеть целевых позиций при устройстве засады.

Что дальше...

Существует простая в реализации улучшенная версия алгоритма – вариант P-A*mbush. Простое упорядочение участников засады, от ближних к дальним, способно значительно улучшить результат, практически не добавляя при этом вычислительных затрат. Это связано с тем, что операция упорядочения выполняется только один раз и легко реализуется с помощью приоритетной очереди, которая затем используется как обычный список в алгоритме A*mbush без необходимости внесения больших изменений.

Выбор удобных точек позиций

Бывают ситуации, когда количество точек позиций должно быть уменьшено в определенный момент игры или из-за ограниченного объема памяти. В этом рецепте применяется механизм, называемый уплотнением, который помогает решить эту задачу, оценивая позиции по присвоенным им значениям.

Подготовка

В этом рецепте используются статические методы, поэтому важно, чтобы вы понимали значение и особенности использования статических функций.

Как это реализовать...

Определим класс Waypoint и добавим в него функцию уплотнения набора позиций.

1. Определим класс Waypoint, наследующий класс MonoBehaviour и дополнительно реализующий интерфейс IComparer:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Waypoint : MonoBehaviour, IComparer
{
    public float value;
    public List<Waypoint> neighbours;
}
```

2. Реализуем функцию Compare упомянутого интерфейса:

```
public int Compare(object a, object b)
{
    Waypoint wa = (Waypoint)a;
    Waypoint wb = (Waypoint)b;
    if (wa.value == wb.value)
        return 0;
    if (wa.value < wb.value)
        return -1;
    return 1;
}
```

3. Реализуем статическую функцию расчета возможности перемещения агента между двумя позициями:

```
public static bool CanMove(Waypoint a, Waypoint b)
{
    // реализуйте свою модель поведения для
    // определения возможности перемещения агента
    // между двумя позициями
    return true;
}
```

4. Объявим функцию уплотнения:

```
public static void CondenseWaypoints(List<Waypoint> waypoints, float
distanceWeight)
{
    // дальнейшая реализация описывается ниже
}
```

5. Инициализируем несколько переменных и отсортируем позиции в порядке убывания:

```
distanceWeight *= distanceWeight;
waypoints.Sort();
waypoints.Reverse();
List<Waypoint> neighbours;
```

6. Определим цикл расчета для каждой позиции:

```
foreach (Waypoint current in waypoints)
{
    // дальнейшая реализация описывается ниже
}
```

7. Получим соседние позиции, отсортируем их и запустим цикл оценки:

```
neighbours = new List<Waypoint>(current.neighbours);
neighbours.Sort();
foreach (Waypoint target in neighbours)
{
    if (target.value > current.value)
        break;
    if (!CanMove(current, target))
        continue;
    // дальнейшая реализация описывается ниже
}
```

8. Вычислим целевые позиции:

```
Vector3 deltaPos = current.transform.position;
deltaPos -= target.transform.position;
```

```
deltaPos = Vector3.Cross(deltaPos, deltaPos);
deltaPos *= distanceWeight;
```

9. Вычислим общую стоимость позиции и решим: сохранить ее или отбросить:

```
float deltaVal = current.value - target.value;
deltaVal *= deltaVal;
if (deltaVal < distanceWeight)
{
    neighbours.Remove(target);
    waypoints.Remove(target);
}
```

Как это работает...

Точки позиций упорядочиваются согласно их важности (например, по высоте, при использовании для размещения снайперов или другим преимуществам местоположения), а затем выполняется проверка соседних позиций, чтобы исключить часть из них. Естественно, менее ценные позиции проверяются в конце цикла. В следующем рецепте речь пойдет об анализе позиций.

Полезные ссылки

- Рецепт *«Анализ точек позиций, основанный на их высоте»*.
- Рецепт *«Анализ точек позиций, основанный на обзоре и видимости»*.

Анализ точек позиций по их высоте

Этот рецепт позволяет оценить позицию в соответствии с ее положением. Со стратегической точки зрения позиции с небольшой высотой являются невыгодными. Здесь используется гибкий алгоритм определения качества позиции, основывающийся на высотах окружающих ее позиций.

Подготовка

Этот рецепт достаточно прост, и его реализация не требует особенных познаний. Алгоритм достаточно гибок и может обрабатывать список позиций, соседних с данной, или полный граф. Эвристический подход к хранению окрестностей здесь не приводится, поскольку он специфичен для каждой игры.

Как это реализовать...

Реализуем функцию оценки местоположения по его высоте и высоте окружающих позиций:

1. Объявим функцию оценки качества:

```
public static float GetHeightQuality (Vector3 location,
                                      Vector3[] surroundings)
{
    // дальнейшая реализация описывается ниже
}
```

2. Инициализируем переменные, необходимые для расчетов:

```
float maxQuality = 1f;
float minQuality = -1f;
float minHeight = Mathf.Infinity;
float maxHeight = Mathf.NegativeInfinity;
float height = location.y;
```

3. Обойдем окрестности, чтобы определить максимальную и минимальную высоты:

```
foreach (Vector3 s in surroundings)
{
    if (s.y > maxHeight)
        maxHeight = s.y;
    if (s.y < minHeight)
        minHeight = s.y;
}
```

4. Вычислим оценку качества в заданном диапазоне:

```
float quality = (height-minHeight) / (maxHeight - minHeight);
quality *= (maxQuality - minQuality);
quality += minQuality;
return quality;
```

Как это работает...

Мы выполняем обход списка окрестных пунктов, чтобы определить максимальную и минимальную высоты. Затем рассчитываем оценки позиций в диапазоне от –1 до 1. Этот диапазон можно изменить для нужд конкретной игры или инвертировать значимость высоты в формуле.

Анализ точек позиций по обзорности и незаметности

При разработке военных игр, особенно шутеров от первого лица, часто необходимо определить ценность позиции по широте обзора для стрельбы или незаметности для противника. Этот рецепт помогает вычислить оценку позиции, основываясь на этих критериях.

Подготовка

Создадим функцию, проверяющую – находится ли позиция в той же комнате, что и другие:

```
public bool IsInSameRoom(Vector3 from, Vector3 location,
                         string tagWall = "Wall")
{
    RaycastHit[] hits;
    Vector3 direction = location - from;
    float rayLength = direction.magnitude;
    direction.Normalize();
    Ray ray = new Ray(from, direction);
    hits = Physics.RaycastAll(ray, rayLength);
    foreach (RaycastHit h in hits)
    {
        string tagObj = h.collider.gameObject.tag;
        if (tagObj.Equals(tagWall))
            return false;
    }
    return true;
}
```

Как это реализовать...

Создадим функцию оценки позиции:

1. Определим функцию с ее параметрами:

```
public static float GetCoverQuality(
        Vector3 location,
        int iterations,
        Vector3 characterSize,
        float radius,
        float randomRadius,
        float deltaAngle)
{
    // дальнейшая реализация описывается ниже
}
```

2. Инициализируем переменную для обработки поворота в градусах, количества пересечений и фактической видимости:

```
float theta = 0f;
int hits = 0;
int valid = 0;
```

3. Начнем с главного цикла расчета оценки позиции с возвратом вычисленного значения:

```
for (int i = 0; i < iterations; i++)
{
    // дальнейшая реализация описывается ниже
}
return (float)(hits / valid);
```

4. Создадим случайные позиции вблизи первоначальной, чтобы определить ее незаметность:

```
Vector3 from = location;
float randomBinomial = Random.Range(-1f, 1f);
from.x += radius * Mathf.Cos(theta) + randomBinomial * randomRadius;
from.y += Random.value * 2f * randomRadius;
from.z += radius * Mathf.Sin(theta) + randomBinomial * randomRadius;
```

5. Если случайная позиция оказалась в другой комнате, отбросить ее:

```
if (!IsInSameRoom(from, location))
    continue;
valid++;
```

6. Если случайная позиция оказалась в той же комнате, тогда:

```
Vector3 to = location;
to.x += Random.Range(-1f, 1f) * characterSize.x;
to.y += Random.value * characterSize.y;
to.z += Random.Range(-1f, 1f) * characterSize.z;
```

7. Отбросим луч для проверки незаметности:

```
Vector3 direction = to - location;
float distance = direction.magnitude;
direction.Normalize();
Ray ray = new Ray(location, direction);
if (Physics.Raycast(ray, distance))
    hits++;
theta = Mathf.Deg2Rad * deltaAngle;
```

Как это работает...

Мы создали цикл, в котором выбираем случайные позиции вокруг заданной и проверяем ее достижимость. После этого рассчитывается коэффициент, определяющий качество позиции.

Оценка точек позиций для принятия решения

Описанные выше методы принятия решений часто недостаточно гибки для простого вычисления ценности позиций, так как предназначены для проверки более сложных условий. В этом случае решить задачу можно с применением рассмотренных ранее методов оценки позиций.

Главная идея заключается в добавлении к узлу условия, чтобы его можно было оценить, например, используя дерево решений или более сложные эвристики вычисления значимости позиций.

Подготовка

Прежде чем приступать к этому рецепту, прочитайте еще раз рецепт *«Работа конечного автомата»* из предыдущей главы.

Как это реализовать...

Внесем некоторые изменения:

1. Добавим в класс `Waypoint` общедоступное свойство условия с типом данных `Condition`:

```
public Condition condition;
```

2. Теперь его легко можно интегрировать с методами принятия решений, использующими дочерние классы условий, такие как `ConditionFloat`.

Как это работает...

Псевдоабстрактный класс `Condition`, описанный ранее, содержит метод `Test`, проверяющий условие.

Полезные ссылки

- *Глава 3 «Принятие решений»*.

Карты влияния

Еще один способ использования графов – оценка, насколько далеко простирается влияние агента или, в данном случае, боевого модуля на определенную область игрового мира. В этом контексте влияние представляется в виде области, окружающей агента или их группу агентов из одного и того же подразделения.

Такое представление является ключевым элементом создания механизмов принятия решений с использованием искусственного интеллекта в играх, связанных с эмуляцией военных действий, или когда важно знать, какая часть игрового мира контролируется заданной группой агентов.

Подготовка

Этот рецепт требует построения графа, поэтому он основан на общем классе графа `Graph`. Однако для обработки вершин или поиска соседей наобходимо создать свой дочерний класс графа или определить свои методы, как было описано в *главе 2 «Навигация»*.

Далее мы рассмотрим порядок реализации конкретного алгоритма для этого рецепта, опираясь на общие функции классов `Graph` и `Vertex`.

Нам также потребуются базовый компонент `Unity` для реализации агента и перечисление `Faction`.

Ниже приводятся определения перечисления `Faction` и класса `Unit`. Их можно поместить в общий файл `Unit.cs`:

```
using UnityEngine;
using System.Collections;

public enum Faction
{
    // примеры значений
    BLUE, RED
}

public class Unit : MonoBehaviour
{
    public Faction faction;
    public int radius = 1;
    public float influence = 1f;
```

```
    public virtual float GetDropOff(int locationDistance)
    {
        return influence;
    }
}
```

Как это реализовать...

Определим классы VertexInfluence и InfluenceMap для управления вершинами и графом соответственно:

1. Создадим класс VertexInfluence, наследующий класс Vertex:

```
using UnityEngine;
using System.Collections.Generic;

public class VertexInfluence : Vertex
{
    public Faction faction;
    public float value = 0f;
}
```

2. Реализуем функцию установки значений с уведомлением об успехе:

```
public bool SetValue(Faction f, float v)
{
    bool isUpdated = false;
    if (v > value)
    {
        value = v;
        faction = f;
        isUpdated = true;
    }
    return isUpdated;
}
```

3. Определим класс InfluenceMap, наследующий класс Graph (или более специализированного его потомка):

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class InfluenceMap : Graph
{
    public List<Unit> unitList;
    // аналог вершин в обычном графе
```

```
    GameObject[] locations;
}
```

4. Определим функцию Awake для инициализации:

```
void Awake()
{
    if (unitList == null)
        unitList = new List<Unit>();
}
```

5. Реализуем функцию добавления боевого модуля на карту:

```
public void AddUnit(Unit u)
{
    if (unitList.Contains(u))
        return;
    unitList.Add(u);
}
```

6. Реализуем функцию удаления боевого модуля с карты:

```
public void RemoveUnit(Unit u)
{
    unitList.Remove(u);
}
```

7. Начнем определение функции оценки влияния:

```
public void ComputeInfluenceSimple()
{
    int vId;
    GameObject vObj;
    VertexInfluence v;
    float dropOff;
    List<Vertex> pending = new List<Vertex>();
    List<Vertex> visited = new List<Vertex>();
    List<Vertex> frontier;
    Vertex[] neighbours;

    // дальнейшая реализация описывается ниже
}
```

8. Продолжим, добавив цикл перебора списка модулей:

```
foreach(Unit u in unitList)
{
    Vector3 uPos = u.transform.position;
    Vertex vert = GetNearestVertex(uPos);
    pending.Add(vert);
```

```
        // дальнейшая реализация описывается ниже
    }
```

9. И наконец, применим алгоритм BFS для распространения влияния с учетом радиуса дальнобойности:

```
// применение алгоритма BFS для распространения влияния
for (int i = 1; i <= u.radius; i++)
{
    frontier = new List<Vertex>();
    foreach (Vertex p in pending)
    {
        if (visited.Contains(p))
            continue;
        visited.Add(p);
        v = p as VertexInfluence;
        dropOff = u.GetDropOff(i);
        v.SetValue(u.faction, dropOff);
        neighbours = GetNeighbours(vert);
        frontier.AddRange(neighbours);
    }
    pending = new List<Vertex>(frontier);
}
```

Как это работает...

Граф карты влияния работает подобно обычному графу, но дополнительно хранит информацию о влиянии вершин, для чего необходимо лишь несколько дополнительных параметров. Наиболее значимой частью является расчет влияния, основанный на алгоритме BFS.

Области влияния для каждого боевого модуля простираются на карте с учетом радиуса дальнобойности. Если вычисленное влияние (растекание) распространяется за пределы оригинального набора вершин, изменяется набор вершин.

Что дальше...

Функция растекания должна быть настроена под потребности конкретной игры. Можно определить функцию, как показано ниже, использующую параметр расстояния:

```
public virtual float GetDropOff(int locationDistance)
{
    float d = influence / radius * locationDistance;
    return influence - d;
}
```

Важно отметить, что параметр расстояния является целым числом и определяет расстояние, измеряемое в вершинах.

Наконец, можно избежать использования групп, применяя вместо них ссылку на модуль. Таким способом можно отображать влияние отдельных модулей, рассматривая их в качестве групп или команд.

Полезные ссылки

- Рецепт *«Представление игрового мира с помощью сетей»* и *«Поиск кратчайшего пути в сети с помощью алгоритма BFS»* в *главе 2* «Навигация».

Улучшение карт влияния путем заполнения

Представленный выше механизм оценки влияния отлично подходит для простых случаев, основанных на влияниях отдельных единиц, представляющих группы. Однако его применение может привести к появлению пробелов на карте, когда их в действительности не должно быть. Решить эту проблему можно методом заполнения, основанным на алгоритме Дейкстры.

Подготовка

В этом случае следует объединить возможность маркировки вершин с логикой функции растекания для модулей в класс гильдии `Guild`. Этот компонент следует присоединить к игровому объекту, одному в каждой гильдии:

```
using UnityEngine; using System;
using System.Collections;

public class Guild : MonoBehaviour
{
    public string guildName;
    public int maxStrength;
    public GameObject baseObject;
    [HideInInspector]
    public int strenghth

    public virtual void Awake()
    {
        strength = maxStrength;
    }
}
```

В него также нужно добавить функцию растекания. Однако на этот раз в примере будет использоваться эвклидово расстояние:

```
public virtual float GetDropOff(float distance)
{
    float d = Mathf.Pow(1 + distance, 2f);
    return strength / d;
}
```

Наконец, нам потребуется тип данных GuildRecord для представления узла в алгоритме Дейкстры:

1. Создадим структуру GuildRecord, реализующую интерфейс IComparable:

```
using UnityEngine;
using System.Collections; using System;

public struct GuildRecord : IComparable<GuildRecord>
{
    public Vertex location;
    public float strength;
    public Guild guild;
}
```

2. Реализуем функцию Equal:

```
public override bool Equals(object obj)
{
    GuildRecord other = (GuildRecord)obj;
    return location == other.location;
}

public bool Equals(GuildRecord o)
{
    return location == o.location;
}
```

3. Реализуем обязательные функции интерфейса IComparable:

```
public override int GetHashCode()
{
    return base.GetHashCode();
}

public int CompareTo(GuildRecord other)
{
    if (location == other.location)
        return 0;
```

```
    // вычитание является обратным действием для
    // двоичной кучи, упорядоченной по убыванию
    return (int)(other.strength - strength);
}
```

Как это реализовать...

А теперь остается лишь изменить некоторые файлы и добавить в них функции:

1. Добавим свойство guild в класс VertexInfluence:

```
public Guild guild;
```

2. Включим новые свойства в класс InfluenceMap:

```
public float dropOffThreshold;
private Guild[] guildList;
```

3. Также добавим следующую строку в функцию Awake класса InfluenceMap:

```
guildList = gameObject.GetComponents<Guild>();
```

4. Определим функцию заполнения карты:

```
public List<GuildRecord> ComputeMapFlooding()
{
}
```

5. Объявим основные переменные:

```
GPWiki.BinaryHeap<GuildRecord> open;
open = new GPWiki.BinaryHeap<GuildRecord>();
List<GuildRecord> closed;
closed = new List<GuildRecord>();
```

6. Добавим начальные узлы из каждой гильдии в приоритетную очередь:

```
foreach (Guild g in guildList)
{
    GuildRecord gr = new GuildRecord();
    gr.location = GetNearestVertex(g.baseObject);
    gr.guild = g;
    gr.strength = g.GetDropOff(0f);
    open.Add(gr);
}
```

7. Создадим основной цикл алгоритма Дейкстра и вернем результаты:

```
while (open.Count != 0)
{
    // дальнейшая реализация описывается ниже
}
return closed;
```

8. Выберем первый узел в очереди и получим его соседей:

```
GuildRecord current;
current = open.Remove();
GameObject currObj;
currObj = GetVertexObj(current.location);
Vector3 currPos;
currPos = currObj.transform.position;
List<int> neighbours;
neighbours = GetNeighbors(current.location);
```

9. Создадим цикл, выполняющий вычисления для каждого из соседей, и поместим текущий узел в список закрытых:

```
foreach (int n in neighbours)
{
    // дальнейшая реализация описывается ниже
}
closed.Add(current);
```

10. Вычислим заполнение для текущей вершины и проверим, можно ли сказать, что узел принадлежит гильдии:

```
GameObject nObj = GetVertexObj(n);
Vector3 nPos = nObj.transform.position;
float dist = Vector3.Distance(currPos, nPos);
float strength = current.guild.GetDropOff(dist);
if (strength < dropOffThreshold)
    continue;
```

11. Создадим вспомогательный узел GuildRecord с данными из текущей вершины:

```
GuildRecord neighGR = new GuildRecord();
neighGR.location = n;
neighGR.strength = strength;
VertexInfluence vi;
vi = nObj.GetComponent<VertexInfluence>();
neighGR.guild = vi.guild;
```

12. Проверим список закрытых узлов и условие, когда следует избегать новых назначений:

```
if (closed.Contains(neighGR))
{
    int location = neighGR.location;
    int index = closed.FindIndex(x => x.location == location);
    GuildRecord gr = closed[index];
    if (gr.guild.name != current.guild.name
            && gr.strength < strength)
        continue;
}
```

13. Проверим также приоритетную очередь:

```
else if (open.Contains(neighGR))
{
    bool mustContinue = false; foreach (GuildRecord gr in open)
    {
        if (gr.Equals(neighGR))
        {
            mustContinue = true; break;
        }
    }
    if (mustContinue)
        continue;
}
```

14. Создадим новую запись GuildRecord и добавим ее в приоритетную очередь, если все предыдущие проверки провалились:

```
else
{
    neighGR = new GuildRecord();
    neighGR.location = n;
}
neighGR.guild = current.guild;
neighGR.strength = strength;
```

15. Добавим запись в приоритетную очередь:

```
open.Add(neighGR);
```

Как это работает...

Для каждой вершины алгоритм возвращает ее принадлежность к гильдии. Он обходит весь граф, начиная с вершин, принадлежащих гильдиям, и выполняет расчеты.

Алгоритм просматривает весь граф, начиная с позиций, принадлежащих гильдиям. Учитывая инвертирующее вычитание, выполнен-

ное выше, приоритетная очередь всегда начинается с сильных узлов, и определение принадлежности продолжается, пока не будут достигнуты значения меньше `dropOffThreshold`. Также выполняется проверка условия отказа от изменения принадлежности: изменить принадлежность вершины можно, только если оценка влияния гильдии больше текущей и выполняется присваивание другой гильдии.

Полезные ссылки

- Рецепт *«Карты влияния»*.
- Рецепт *«Поиск кратчайшего пути с помощью алгоритма Дейкстры»* из *главы 2 «Навигация»*.

Улучшение карт влияния с помощью фильтров свертки

Фильтры свертки обычно применяются для обработки изображений, но заложенные в них принципы вполне можно использовать для корректировки карт влияния, задавая ценность модулей и их окружения. В этом рецепте рассматривается пара алгоритмов изменения сети с помощью матричных фильтров.

Подготовка

Перед знакомством с реализацией этого рецепта прочитайте рецепт «Карты влияния», чтобы понимать контекст, в котором применяется данный рецепт.

Как это реализовать...

Реализуем функцию свертки `Convolve`:

1. Объявим функцию `Convolve`:

```
public static void Convolve(
        float[,] matrix,
        ref float[,] source,
        ref float[,] destination)
{
    // дальнейшая реализация описывается ниже
}
```

2. Подготовим переменные к проведению расчетов и обходу массивов:

```
int matrixLength = matrix.GetLength(0);
int size = (int)(matrixLength - 1) / 2;
int height = source.GetLength(0);
int width = source.GetLength(1);
int i, j, k, m;
```

3. Создадим первый цикл для обхода конечной и исходной сеток:

```
for (i = 0; i < width-- size; i++)
{
    for (j = 0; j < height-- size; j++)
    {
        // дальнейшая реализация описывается ниже
    }
}
```

4. Реализуем второй цикл для обхода матрицы фильтра:

```
destination[i, j] = 0f;
for (k = 0; k < matrixLength; k++)
{
    for (m = 0; m < matrixLength; m++)
    {
        int row = i + k - size;
        int col = j + m - size;
        float aux = source[row, col] * matrix[k,m];
        destination[i, j] += aux;
    }
}
```

Как это работает...

Создается новая сеть для замены исходной после применения матричного фильтра к каждой вершине. Затем выполняется обход всех вершин, которые должны быть помещены в конечную сеть, производятся вычисления, исходными данными для которых являются значения вершины из исходной сети, и к результатам применяется матричный фильтр.

Важно отметить, что для нормальной работы алгоритма матричный фильтр должен быть массивом с размерностью, равной квадрату нечетного числа.

Что дальше...

Ниже приводится функция `ConvolveDriver`, которая помогает выполнить обход с использованием функции `Convolve`, реализованной ранее:

1. Объявим функцию ConvolveDriver:

```
public static void ConvolveDriver(
        float[,] matrix,
        ref float[,] source,
        ref float[,] destination,
        int iterations)
{
    // дальнейшая реализация описывается ниже
}
```

2. Создадим вспомогательные переменные для хранения сетей:

```
float[,] map1;
float[,] map2;
int i;
```

3. Заменим карты в зависимости от четного или нечетного количества итераций:

```
if (iterations % 2 == 0)
{
    map1 = source;
    map2 = destination;
}
else
{
    destination = source;
    map1 = destination;
    map2 = source;
}
```

4. Применим предыдущую функцию в ходе итераций и замены:

```
for (i = 0; i < iterations; i++)
{
    Convolve(matrix, ref source, ref destination);
    float[,] aux = map1;
    map1 = map2;
    map2 = aux;
}
```

Полезные ссылки

- Рецепт *«Карты влияния»*.

Построение боевых кругов

Этот рецепт основан на алгоритме «Круг кунг-фу», разработанном для игры *Kingdoms of Amalur: Reckoning*. Его цель: дать противникам интеллектуальный способ подойти к заданному игроку и атаковать его. Он очень похож на рецепт формирований, но используется на этапе управления, где на основе оценки сил игрока и противников принимается решение о подходе и дается разрешение на атаку. Его реализация позволяет управлять списком боевых кругов, что особенно важно в многопользовательских играх.

Подготовка

Перед реализацией алгоритма боевых кругов необходимо создать несколько вспомогательных компонентов. Во-первых, это псевдоабстрактный класс Attack, являющийся обобщенным представлением атак для каждого противника, который используется как шаблон для создания атак в игре. Во-вторых, это класс противника Enemy, который хранит логику действий противников. Как будет показано ниже, класс Enemy хранит список различных компонентов атаки, содержащихся в игровых объектах.

Ниже приводится определение класса Attack:

```
using UnityEngine;
using System.Collections;

public class Attack : MonoBehaviour
{
    public int weight;

    public virtual IEnumerator Execute()
    {
        // поместите сюда свои модели поведения для атак
        yield break;
    }
}
```

Далее описываются этапы создания компонента противника Enemy:

1. Определим класс Enemy:

```
using UnityEngine;
using System.Collections;

public class Enemy : MonoBehaviour
{
```

```
    public StageManager stageManager;
    public int slotWeight;
    [HideInInspector]
    public int circleId = -1;
    [HideInInspector]
    public bool isAssigned;
    [HideInInspector]
    public bool isAttacking;
    [HideInInspector]
    public Attack[] attackList;
}
```

2. Определим функцию Start:

```
void Start()
{
    attackList = gameObject.GetComponents<Attack>();
}
```

3. Реализуем функцию назначения цели для боевого круга:

```
public void SetCircle(GameObject circleObj = null)
{
    int id = -1;
    if (circleObj == null)
    {
        Vector3 position = transform.position;
        id = stageManager.GetClosestCircle(position);
    }
    else
    {
        FightingCircle fc;
        fc = circleObj.GetComponent<FightingCircle>();
        if (fc != null)
            id = fc.gameObject.GetInstanceID();
    }
    circleId = id;
}
```

4. Определим функцию для запроса слота в диспетчере:

```
public bool RequestSlot()
{
    isAssigned = stageManager.GrantSlot(circleId, this);
    return isAssigned;
}
```

5. Определим функцию для освобождения слота в диспетчере:

```
public void ReleaseSlot()
{
    stageManager.ReleaseSlot(circleId, this);
    isAssigned = false;
    circleId = -1;
}
```

6. Реализуем функцию для запроса атаки из списка (порядок тот же, что и в инспекторе):

```
public bool RequestAttack(int id)
{
    return stageManager.GrantAttack(circleId, attackList[id]);
}
```

7. Определим виртуальную функцию для модели поведения атаки:

```
public virtual IEnumerator Attack()
{
    // Будет реализована ниже
    // поместите сюда свои модели поведения для атак
    yield break;
}
```

Как это реализовать...

Теперь реализуем классы `FightingCircle` и `StageManager`.

1. Создадим класс `FightingCircle` со свойствами:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class FightingCircle : MonoBehaviour
{
    public int slotCapacity;
    public int attackCapacity;
    public float attackRadius;
    public GameObject player;
    [HideInInspector]
    public int slotsAvailable;
    [HideInInspector]
    public int attackAvailable;
    [HideInInspector]
    public List<GameObject> enemyList;
```

```
    [HideInInspector]
    public Dictionary<int, Vector3> posDict;
}
```

2. Реализуем функцию Awake для инициализации:

```
void Awake()
{
    slotsAvailable = slotCapacity;
    attackAvailable = attackCapacity;
    enemyList = new List<GameObject>();
    posDict = new Dictionary<int, Vector3>();
    if (player == null)
        player = gameObject;
}
```

3. Определим функцию Update, обновляющую позиции слотов:

```
void Update()
{
    if (enemyList.Count == 0)
        return;
    Vector3 anchor = player.transform.position;
    int i;
    for (i = 0; i < enemyList.Count; i++)
    {
        Vector3 position = anchor;
        Vector3 slotPos = GetSlotLocation(i);
        int enemyId = enemyList[i].GetInstanceID();
        position += player.transform.TransformDirection(slotPos);
        posDict[enemyId] = position;
    }
}
```

4. Реализуем функцию добавления противников в круг:

```
public bool AddEnemy(GameObject enemyObj)
{
    Enemy enemy = enemyObj.GetComponent<Enemy>();
    int enemyId = enemyObj.GetInstanceID();
    if (slotsAvailable < enemy.slotWeight)
        return false;
    enemyList.Add(enemyObj);
    posDict.Add(enemyId, Vector3.zero);
    slotsAvailable -= enemy.slotWeight;
    return true;
}
```

5. Реализуем функцию удаления противников из круга:

```
public bool RemoveEnemy(GameObject enemyObj)
{
    bool isRemoved = enemyList.Remove(enemyObj);
    if (isRemoved)
    {
        int enemyId = enemyObj.GetInstanceID();
        posDict.Remove(enemyId);
        Enemy enemy = enemyObj.GetComponent<Enemy>();
        slotsAvailable += enemy.slotWeight;
    }
    return isRemoved;
}
```

6. Реализуем функцию для смены позиций противников в круге:

```
public void SwapEnemies(GameObject enemyObjA, GameObject enemyObjB)
{
    int indexA = enemyList.IndexOf(enemyObjA);
    int indexB = enemyList.IndexOf(enemyObjB);
    if (indexA != -1 && indexB != -1)
    {
        enemyList[indexB] = enemyObjA;
        enemyList[indexA] = enemyObjB;
    }
}
```

7. Определим функцию для получения пространственного положения противника в круге:

```
public Vector3 GetPositions(GameObject enemyObj)
{
    int enemyId = enemyObj.GetInstanceID();
    if (!posDict.ContainsKey(enemyId))
        return null;
    return posDict[enemyId];
}
```

8. Реализуем функцию вычисления пространственного положения слота:

```
private Vector3 GetSlotLocation(int slot)
{
    Vector3 location = new Vector3();
    float degrees = 360f / enemyList.Count;
    degrees *= (float)slot;
```

```
        location.x = Mathf.Cos(Mathf.Deg2Rad * degrees);
        location.x *= attackRadius;
        location.z = Mathf.Cos(Mathf.Deg2Rad * degrees);
        location.z *= attackRadius;
        return location;
    }
```

9. Реализуем функцию фактического добавления атак в круг:

```
public bool AddAttack(int weight)
{
    if (attackAvailable - weight < 0)
        return false;
    attackAvailable -= weight;
    return true;
}
```

10. Определим функцию фактического удаления атак из круга:

```
public void ResetAttack()
{
    attackAvailable = attackCapacity;
}
```

11. Теперь определим класс StageManager:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class StageManager : MonoBehaviour
{
    public List<FightingCircle> circleList;
    private Dictionary<int, FightingCircle> circleDic;
    private Dictionary<int, List<Attack>> attackRqsts;
}
```

12. Реализуем функцию Awake для инициализации:

```
void Awake()
{
    circleList = new List<FightingCircle>();
    circleDic = new Dictionary<int, FightingCircle>();
    attackRqsts = new Dictionary<int, List<Attack>>();
    foreach(FightingCircle fc in circleList)
    {
        AddCircle(fc);
    }
}
```

13. Создадим функцию для добавления кругов в диспетчере:

```
public void AddCircle(FightingCircle circle)
{
    if (!circleList.Contains(circle))
        return;
    circleList.Add(circle);
    int objId = circle.gameObject.GetInstanceID();
    circleDic.Add(objId, circle);
    attackRqsts.Add(objId, new List<Attack>());
}
```

14. Кроме того, создадим функцию удаления кругов из диспетчера:

```
public void RemoveCircle(FightingCircle circle)
{
    bool isRemoved = circleList.Remove(circle);
    if (!isRemoved)
        return;
    int objId = circle.gameObject.GetInstanceID();
    circleDic.Remove(objId);
    attackRqsts[objId].Clear();
    attackRqsts.Remove(objId);
}
```

15. Определим функцию для получения ближайшего круга по заданной позиции:

```
public int GetClosestCircle(Vector3 position)
{
    FightingCircle circle = null;
    float minDist = Mathf.Infinity;
    foreach(FightingCircle c in circleList)
    {
        Vector3 circlePos = c.transform.position;
        float dist = Vector3.Distance(position, circlePos);
        if (dist < minDist)
        {
            minDist = dist;
            circle = c;
        }
    }
    return circle.gameObject.GetInstanceID();
}
```

16. Определим функцию выделения слота для противника в заданном круге:

```
public bool GrantSlot(int circleId, Enemy enemy)
{
    return circleDic[circleId].AddEnemy(enemy.gameObject);
}
```

17. Реализуем функцию удаления противника из заданного круга:

```
public void ReleaseSlot(int circleId, Enemy enemy)
{
    circleDic[circleId].RemoveEnemy(enemy.gameObject);
}
```

18. Определим функцию, разрешающую атаку и добавляющую ее в список диспетчера:

```
public bool GrantAttack(int circleId, Attack attack)
{
    bool answer = circleDic[circleId].AddAttack(attack.weight);
    attackRqsts[circleId].Add(attack);
    return answer;
}
```

19. Реализуем функцию выполнения атак:

```
public IEnumerator ExecuteAtacks()
{
    foreach (int circle in attackRqsts.Keys)
    {
        List<Attack> attacks = attackRqsts[circle];
        foreach (Attack a in attacks)
            yield return a.Execute();
    }
    foreach (FightingCircle fc in circleList)
        fc.ResetAttack();
}
```

Как это работает...

Классы `Attack` и `Enemy` управляют поведением объектов, когда это необходимо, что позволяет вызывать класс `Enemy` из другого компонента, присоединенного к игровому объекту. Класс `FightingCircle` похож на класс `FormationPattern`, в том смысле что вычисляет целевую позицию для заданного противника, но делает это немного иначе. Наконец, класс `StageManager` дает все необходимые разрешения на выделение и освобождение слотов противников и атак во всех кругах.

Что дальше...

Реализацию боевых кругов можно добавить в виде компонента в игровой объект, который является целевым игроком или другим пустым объектом, содержащим ссылку на объект игрока.

Кроме того, функции, разрешающие атаку и выполнение атаки, можно поместить в боевой круг. Но предпочтительнее размещать их в диспетчере, чтобы обеспечить централизованное управление атаками, а за кругами оставить обработку позиций целей, подобно тому, как это было сделано для формирований.

Полезные ссылки

- Рецепт «*Обработка формирований*».

Более подробную информацию об алгоритме «Круг кунг-фу» можно найти в книге Стива Рабина (Steve Rabin) «*Game AI Pro*».

Глава 5

Органы чувств агентов

В этой главе рассматриваются алгоритмы имитации органов чувств агентов:

- имитация зрения с применением коллайдера;
- имитация слуха с применением коллайдера;
- имитация обоняния с применением коллайдера;
- имитации зрения с применением графа;
- имитация слуха с применением графа;
- имитация обоняния с применением графа;
- реализация органов чувств в стелс-игре.

Введение

В этой главе рассматриваются различные способы имитации органов чувств агентов. Для этого используются уже знакомые инструменты моделирования: коллайдеры и графы.

В первом случае используются отбрасывание лучей, коллайдеры и функции `MonoBehaviour`, связанные с этим компонентом, такие как функция `OnCollisionEnter`, используемая для выделения ближних объектов в трехмерном пространстве. Затем мы посмотрим, как сымитировать органы чувств с применением теории графов, чтобы получить представление об игровом мире.

В заключение будет описана реализация органов чувств агента, использующая смешанный подход, заключающийся в применении рассмотренных ранее алгоритмов на новом уровне.

Имитации зрения с применением коллайдера

Это, пожалуй, самый простой способ имитации зрения. Здесь коллайдер Unity, простой или меш-коллайдер, используется как инструмент, помогающий определить, находится ли объект в прямой видимости агента.

Подготовка

Очень важно присоединить компонент коллайдера к игровому объекту, который использует сценарий из этого рецепта, а также всех других рецептов в этой главе, основанных на использовании коллайдеров. В данном случае рекомендуется использовать пирамидальный коллайдер, который хорошо подходит для имитации конуса обзора. Чем меньше полигонов, тем быстрее коллайдер действует в процессе игры.

Как это реализовать...

Создадим компонент, способный «увидеть» находящихся поблизости противников:

1. Создадим компонент `Visor` и объявим его свойства. Также добавим приведенные ниже теги в конфигурацию Unity:

```
using UnityEngine;
using System.Collections;

public class Visor : MonoBehaviour
{
    public string tagWall = "Wall";
    public string tagTarget = "Enemy";
    public GameObject agent;
}
```

2. Реализуем функцию инициализации игрового объекта, когда компонент уже присоединен к нему:

```
void Start()
{
    if (agent == null)
        agent = gameObject;
}
```

3. Объявим функцию проверки столкновений в каждом кадре и отложим ее реализацию на более поздние этапы:

```
public void OnTriggerStay(Collider coll)
{
    // дальнейшая реализация описывается ниже
}
```

4. Отбросим столкновения, не связанные с целью:

```
string tag = coll.gameObject.tag;
if (!tag.Equals(tagTarget))
    return;
```

5. Получим позицию игрового объекта и определим направление на него:

```
GameObject target = coll.gameObject;
Vector3 agentPos = agent.transform.position;
Vector3 targetPos = target.transform.position;
Vector3 direction = targetPos - agentPos;
```

6. Вычислим расстояние и создадим новый направленный луч:

```
float length = direction.magnitude;
direction.Normalize();
Ray ray = new Ray(agentPos, direction);
```

7. Отбросим вновь созданный луч и определим все пересечения:

```
RaycastHit[] hits;
hits = Physics.RaycastAll(ray, length);
```

8. Проверим присутствие стен между компонентом `Visor` и целью. Если их нет, можно переходить к вызову нужных функций, реализующих модель поведения:

```
int i;
for (i = 0; i < hits.Length; i++)
{
    GameObject hitObj;
    hitObj = hits[i].collider.gameObject;
    tag = hitObj.tag;
    if (tag.Equals(tagWall))
        return;
}
// поместите сюда код
// модели поведения
```

Как это работает...

Компонент коллайдера в каждом кадре проверяет столкновения с любыми игровыми объектами в сцене. Для сокращения накладных расходов обрабатываются только нужные столкновения.

Если после проверки целевой объект оказывается в области видимости, представленной коллайдером, производится отбрасывание луча, чтобы исключить объекты, которые закрыты стенами, как показано на рис. 5.1.

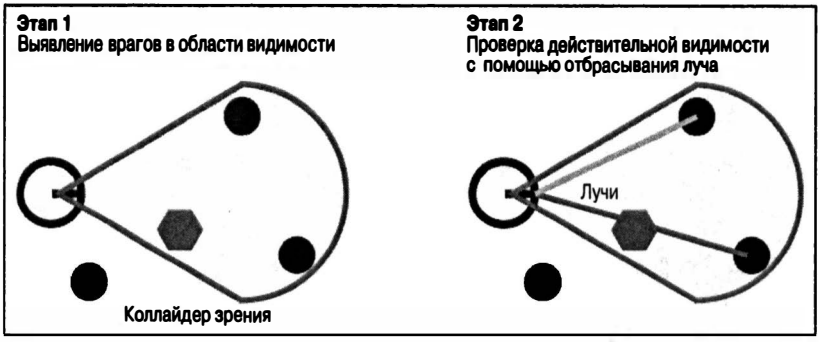


Рис. 5.1 ❖ Отбрасывание луча, чтобы исключить объекты, закрытые стенами

Имитация слуха с применением коллайдера

Имитация слуха в этом рецепте реализована с применением двух сущностей: излучателя и приемника звуков. Он основан на принципах, предложенных Миллингтоном (Millington), и использует возможности коллайдеров Unity по обнаружению излучателей, расположенных поблизости.

Подготовка

Подобно другим рецептам, основанным на коллайдерах, этот также требует подключения коллайдера к каждому объекту, обладающему слухом, и присоединения компонентов `Rigidbody` к излучателям или к приемникам.

Как это реализовать...

Создадим класс SoundReceiver для агентов и класс SoundEmitter для таких звуков, как, например, сигнал тревоги:

1. Определим класс приемника звуков:

```
using UnityEngine;
using System.Collections;

public class SoundReceiver : MonoBehaviour
{
    public float soundThreshold;
}
```

2. Определим функцию, реализующую реакцию на звук:

```
public virtual void Receive(float intensity, Vector3 position)
{
    // поместите сюда код модели поведения
}
```

3. А теперь определим класс излучателя звука:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SoundEmitter : MonoBehaviour
{
    public float soundIntensity;
    public float soundAttenuation;
    public GameObject emitterObject;
    private Dictionary<int, SoundReceiver> receiverDic;
}
```

4. Инициализируем список близлежащих объектов приемников и излучателей, в случае если компонент подключен непосредственно:

```
void Start()
{
    receiverDic = new Dictionary<int, SoundReceiver>();
    if (emitterObject == null)
        emitterObject = gameObject;
}
```

5. Реализуем функцию для добавления новых приемников в список при входе в область действия излучателя:

```
public void OnTriggerEnter(Collider coll)
{
    SoundReceiver receiver;
    receiver = coll.gameObject.GetComponent<SoundReceiver>();
    if (receiver == null)
        return;
    int objId = coll.gameObject.GetInstanceID();
    receiverDic.Add(objId, receiver);
}
```

6. Также реализуем функцию удаления приемников из списка, когда они выходят за пределы досягаемости:

```
public void OnTriggerExit(Collider coll)
{
    SoundReceiver receiver;
    receiver = coll.gameObject.GetComponent<SoundReceiver>();
    if (receiver == null)
        return;
    int objId = coll.gameObject.GetInstanceID();
    receiverDic.Remove(objId);
}
```

7. Определим функцию испускания звуковых волн для близлежащих агентов:

```
public void Emit()
{
    GameObject srObj;
    Vector3 srPos;
    float intensity;
    float distance;
    Vector3 emitterPos = emitterObject.transform.position;
    // дальнейшая реализация описывается ниже
}
```

8. Вычислим затухание звука для каждого из приемников:

```
foreach (SoundReceiver sr in receiverDic.Values)
{
    srObj = sr.gameObject;
    srPos = srObj.transform.position;
    distance = Vector3.Distance(srPos, emitterPos);
    intensity = soundIntensity;
    intensity -= soundAttenuation * distance;
    if (intensity < sr.soundThreshold)
```

```
            continue;
        sr.Receive(intensity, emitterPos);
    }
```

Как это работает...

Триггеры коллайдеров помогают составить список агентов в области действия излучателя. Функция излучения звука учитывает расстояние от агента до излучателя, снижая громкость звука в соответствии с законом его затухания (рис. 5.2).

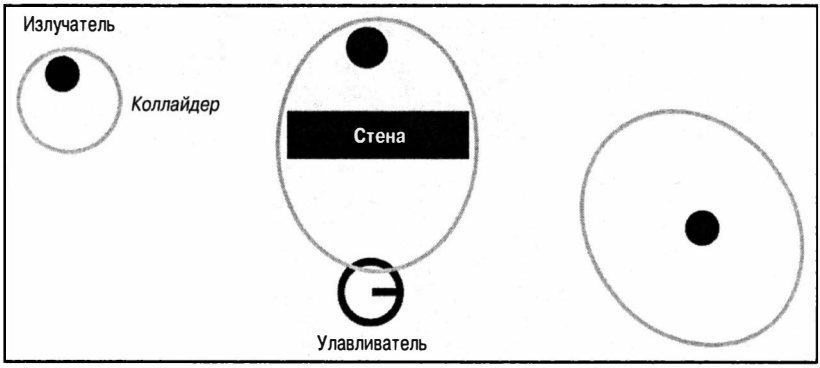


Рис. 5.2 ❖ Агент-приемник и излучатели звука

Что дальше...

Алгоритм можно улучшить, добавив учет влияния стен различного типа на интенсивность звука. Для этого достаточно добавить отбрасывание луча и использовать полученные значения для расчета затухания звука:

1. Создадим словарь для хранения типов стен в виде строк (тегов) и соответствующих им параметров ослабления:

```
public Dictionary<string, float> wallTypes;
```

2. Уменьшим громкость звука следующим образом:

```
intensity -= GetWallAttenuation(emitterPos, srPos);
```

3. Определим функцию, вызванную на предыдущем шаге:

```
public float GetWallAttenuation(Vector3 emitterPos, Vector3 receiverPos)
{
    // дальнейшая реализация описывается ниже
}
```

4. Вычислим значения, необходимые для отбрасывания луча:

```
float attenuation = 0f;
Vector3 direction = receiverPos - emitterPos;
float distance = direction.magnitude;
direction.Normalize();
```

5. Отбросим вновь созданный луч и определим все пересечения:

```
Ray ray = new Ray(emitterPos, direction);
RaycastHit[] hits = Physics.RaycastAll(ray, distance);
```

6. Для каждого типа стен, определяемого с помощью тегов, учтем величину ослабления (хранится в словаре):

```
int i;
for (i = 0; i < hits.Length; i++)
{
    GameObject obj;
    string tag;
    obj = hits[i].collider.gameObject;
    tag = obj.tag;
    if (wallTypes.ContainsKey(tag))
        attenuation += wallTypes[tag];
}
return attenuation;
```

Имитация обоняния с применением коллайдера

При переносе чувства обоняния из реального мира в виртуальный возникают большие сложности. Существует несколько методов такого переноса, но большинство из них опирается на использование коллайдеров или логики графов.

Обоняние можно смоделировать путем вычисления столкновений между агентом и переносящими запах частицами, рассеянными по всему уровню игры.

Подготовка

Подобно другим рецептам, основанным на коллайдерах, этот также требует присоединения коллайдера к каждому объекту, обладающему обонянием, и компонентов твердого тела к излучателям или к приемникам.

Как это реализовать...

Разработаем сценарии для представления частиц, переносящих запахи, и агентов, способных их улавливать:

1. Создадим сценарий для частиц и определим свойства для расчета периода их существования:

```
using UnityEngine;
using System.Collections;

public class OdourParticle : MonoBehaviour
{
    public float timespan;
    private float timer;
}
```

2. Реализуем функцию `Start` для проверки на допустимость:

```
void Start()
{
    if (timespan < 0f)
        timespan = 0f;
    timer = timespan;
}
```

3. Реализуем таймер и организуем уничтожение частиц после завершения цикла их существования:

```
void Update()
{
    timer -= Time.deltaTime;
    if (timer < 0f)
        Destroy(gameObject);
}
```

4. Определим класс агента, обладающего обонянием:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Smeller : MonoBehaviour
{
    private Vector3 target;
    private Dictionary<int, GameObject> particles;
}
```

5. Инициализируем словарь для хранения частиц, переносящих запахи:

```
void Start()
{
    particles = new Dictionary<int, GameObject>();
}
```

6. Добавим в словарь объекты, к которым присоединен компонент частиц с запахом:

```
public void OnTriggerEnter(Collider coll)
{
    GameObject obj = coll.gameObject;
    OdourParticle op;
    op = obj.GetComponent<OdourParticle>();
    if (op == null)
        return;
    int objId = obj.GetInstanceID();
    particles.Add(objId, obj);
    UpdateTarget();
}
```

7. Удалим частицы с запахом из локального словаря при их уничтожении или выходе за пределы диапазона восприятия агента:

```
public void OnTriggerExit(Collider coll)
{
    GameObject obj = coll.gameObject;
    int objId = obj.GetInstanceID();
    bool isRemoved;
    isRemoved = particles.Remove(objId);
    if (!isRemoved)
        return;
    UpdateTarget();
}
```

8. Создадим функцию для вычисления центра источника запаха на основании текущих элементов словаря:

```
private void UpdateTarget()
{
    Vector3 centroid = Vector3.zero;
    foreach (GameObject p in particles.Values)
    {
        Vector3 pos = p.transform.position;
        centroid += pos;
    }
    target = centroid;
}
```

9. Реализуем функцию, возвращающую центр источника запаха, если таковой имеется:

```
public Vector3 GetTargetPosition()
{
    if (particles.Keys.Count == 0)
        return null;
    return target;
}
```

Как это работает...

По аналогии с имитацией слуха на основе коллайдера, здесь триггерные коллайдеры используются для регистрации частиц, переносящих запахи, воспринимаемые агентом (реализуется с помощью словаря). При добавлении или удалении частиц вычисляется позиция центра источника запаха. Однако функция реализована так, что положение внутреннего центра при отсутствии зарегистрированных частиц не обновляется (рис. 5.3).

Рис. 5.3 ❖ Вычисление позиции центра необходимо для определения направления на источник запаха

Что дальше...

Логика испускания частиц реализуется согласно потребностям конкретной игры, и, как правило, для этого создаются экземпляры шаблонных объектов частиц с запахом. Кроме того, рекомендуется прикрепить к агентам компоненты `Rigidbody`. Массовое создание частиц с запахом снижает производительность игры.

Имитации зрения с применением графа

С этого раздела начинается описание рецептов, в которых для имитаций органов чувств используется логика работы с графами. И снова начнем с имитации зрения.

Подготовка

Обязательно разберите главу, где описываются приемы поиска пути, чтобы понять принцип действия рецептов, основанных на графах.

Как это реализовать...

Создадим для реализации новый файл:

1. Определим класс для имитации зрения:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class VisorGraph : MonoBehaviour
{
    public int visionReach;
    public GameObject visorObj;
    public Graph visionGraph;
}
```

2. Проверим допустимость объекта имитации зрения:

```
void Start()
{
    if (visorObj == null)
        visorObj = gameObject;
}
```

3. Определим и начнем реализацию функции определения видимости заданного набора узлов:

```
public bool IsVisible(int[] visibilityNodes)
{
    int vision = visionReach;
    int src = visionGraph.GetNearestVertex(visorObj);
    HashSet<int> visibleNodes = new HashSet<int>();
    Queue<int> queue = new Queue<int>();
    queue.Enqueue(src);
}
```

4. Реализуем алгоритм поиска сначала в ширину:

```
while (queue.Count != 0)
{
    if (vision == 0)
        break;
    int v = queue.Dequeue();
    List<int> neighbours = visionGraph.GetNeighbors(v);
    foreach (int n in neighbours)
    {
        if (visibleNodes.Contains(n))
            continue;
        queue.Enqueue(v);
        visibleNodes.Add(v);
    }
}
```

5. Сравним набор видимых узлов с набором, полученным системой зрения:

```
foreach (int vn in visibleNodes)
{
    if (visibleNodes.Contains(vn))
        return true;
}
```

6. Вернем false, если отсутствует соответствие между двумя наборами узлов:

```
return false;
```

Как это работает...

Рецепт отыскивает узлы в пределах видимости, используя алгоритм поиска в ширину, и сравнивает этот набор с набором узлов, расположенных неподалеку от агентов.

Вычисление входного массива не приводится, поскольку это выходит за рамки рецепта и зависит от точного определения, например положения каждого агента или объекта, для которого необходимо проверить видимость.

Имитация слуха с применением графа

Имитация слуха реализуется подобно зрению, но из-за свойств звука обнаруживает не только его источники в прямой видимости. Кроме

того, для имитации слуха все еще необходимо использовать приемник. Но, в отличие от предыдущей реализации, в этом рецепте уже не агент будет непосредственным приемником, а сам звук будет распространяться по графу и передаваться узлам.

Подготовка

Обязательно разберите главу, где описываются приемы поиска пути, чтобы понять принцип действия рецептов, основанных на графах.

Как это реализовать...

1. Определим класс излучателя:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class EmitterGraph : MonoBehaviour
{
    // дальнейшая реализация описывается ниже
}
```

2. Объявим свойства:

```
public int soundIntensity;
public Graph soundGraph;
public GameObject emitterObj;
```

3. Реализуем проверку допустимости ссылки на объект излучателя:

```
public void Start()
{
    if (emitterObj == null)
        emitterObj = gameObject;
}
```

4. Объявим функцию излучения звука:

```
public int[] Emit()
{
    // дальнейшая реализация описывается ниже
}
```

5. Объявим и инициализируем необходимые переменные:

```
List<int> nodeIds = new List<int>();
Queue<int> queue = new Queue<int>();
```

```
List<int> neighbours;
int intensity = soundIntensity;
int src = soundGraph.GetNearestVertex(emitterObj);
```

6. Добавим исходный узел в список доступных узлов и в очередь:

```
nodeIds.Add(src);
queue.Enqueue(src);
```

7. Цикла обхода узлов для поиска в ширину:

```
while (queue.Count != 0)
{
    // дальнейшая реализация описывается ниже
}
return nodeIds.ToArray();
```

8. Завершим цикл, если громкость звука упала до нуля:

```
if (intensity == 0)
    break;
```

9. Извлечем узел из очереди и получим его соседей:

```
int v = queue.Dequeue();
neighbours = soundGraph.GetNeighbors(v);
```

10. Проверим соседей и добавим их в очередь при необходимости:

```
foreach (int n in neighbours)
{
    if (nodeIds.Contains(n))
        continue;
    queue.Enqueue(n);
    nodeIds.Add(n);
}
```

11. Уменьшим громкость звука:

```
intensity--;
```

Как это работает...

Рецепт возвращает список узлов в зоне досягаемости звука (с учетом затухания), который составляется с помощью алгоритма поиска в ширину. Алгоритм останавливается, когда заканчиваются непосещавшиеся узлы или когда громкость звука при обходе графа падает до нуля.

Что дальше...

После знакомства с реализациями слуха с применением коллайдера и графа можно попробовать реализовать новый гибридный эвристический алгоритм, принимающий на входе расстояние. Если узел оказывается вне зоны досягаемости звука, его соседей не нужно добавлять в очередь.

Полезные ссылки

Рецепты из *главы 2 «Навигация»*:

- *«Поиск кратчайшего пути в сети с помощью алгоритма BFS»*;
- *«Поиск оптимального пути с помощью алгоритма A*»* (принимающий эвристические функции в качестве аргумента).

Имитация обоняния с применением графа

В этом рецепте используется смешанный подход к отметке вершин частицами, переносящими запах, которые сталкиваются с ними.

Подготовка

К вершине должен быть присоединен обширный коллайдер, захватывающий частицы с запахом, находящиеся неподалеку.

Как это реализовать...

1. Добавим следующее свойство в сценарий частиц, переносящих запах, для хранения идентификатора их родителя:

```
public int parent;
```

2. Определим новый класс с поддержкой обоняния, наследующий оригинальный класс вершины:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class VertexOdour : Vertex
{
    private Dictionary<int, OdourParticle> odourDic;
}
```

3. Инициализируем словарь запахов:

```
public void Start()
{
    odourDic = new Dictionary<int, OdourParticle>();
}
```

4. Определим функцию добавления запаха в словарь, принадлежащий вершине:

```
public void OnCollisionEnter(Collision coll)
{
    OdourOdourParticle op;
    op = coll.gameObject.GetComponent<OdourParticle>();
    if (op == null)
        return;
    int id = op.parent;
    odourDic.Add(id, op);
}
```

5. Определим функцию удаления запаха из словаря:

```
public void OnCollisionExit(Collision coll)
{
    OdourParticle op;
    op = coll.gameObject.GetComponent<OdourParticle>();
    if (op == null)
        return;
    int id = op.parent;
    odourDic.Remove(id);
}
```

6. Реализуем проверку присутствия запахов:

```
public bool HasOdour()
{
    if (odourDic.Values.Count == 0)
        return false;
    return true;
}
```

7. Реализуем проверку присутствия заданного запаха в вершине:

```
public bool OdourExists(int id)
{
    return odourDic.ContainsKey(id);
}
```

Как это работает...

Частицы с запахом, сталкиваясь с вершинами, регистрируют свой запах в их словарях. После этого агенты могут проверить, присутствует ли заданный запах в ближайших вершинах.

Полезные ссылки

- Рецепты из *главы 2 «Навигация»*, конструирующие графы и использующие алгоритм поиска в ширину (BFS).

Реализация органов чувств в стелс-игре

Теперь, после знакомства с реализацией алгоритмов имитации органов чувств, пришло время рассмотреть особенности их использования в методах более высокого уровня.

Данный рецепт основывается на игре *Mark of the Ninja*, созданной Бруком Майлзом (Brook Miles) и его командой из Klei Entertainment. Механизм игры построен на идее источников раздражителей органов чувств – зрения и слуха – и обработке их диспетчером чувств.

Подготовка

Поскольку все крутится вокруг источников, нам понадобятся два перечисления, определяющих виды источников (видимые или слышимые) и приоритеты, а также тип (то есть класс) источника.

Вот как выглядит перечисление видов источников:

```
public enum InterestSense
{
    SOUND,
    SIGHT
};
```

А это перечисление приоритетов:

```
public enum InterestPriority
{
    LOWEST = 0,
    BROKEN = 1,
    MISSING = 2,
    SUSPECT = 4,
    SMOKE = 4,
    BOX = 5,
    DISTRACTIONFLARE = 10,
```

```
    TERROR = 20
};
```

Далее приводится тип данных, реализующий источник (раздражитель):

```
using UnityEngine;
using System.Collections;

public struct Interest
{
    public InterestSense sense;
    public InterestPriority priority;
    public Vector3 position;
}
```

Перед тем как приступить к определению классов, необходимых для реализации этой идеи, следует отметить, что функции на уровне органов чувств будут оставлены пустыми, чтобы сохранить рецепт гибким и открытым для конкретной реализации, например с использованием рецептов, описанных выше.

Как это реализовать...

Это длинный рецепт – в нем будут реализованы два объемных класса, поэтому внимательно ознакомьтесь со следующими этапами:

1. Начнем с создания класса, определяющего агентов и их свойства:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AgentAwared : MonoBehaviour
{
    protected Interest interest;
    protected bool isUpdated = false;
}
```

2. Определим функцию для проверки соответствия заданного источника (раздражителя):

```
public bool IsRelevant(Interest i)
{
    int oldValue = (int)interest.priority;
    int newValue = (int)i.priority;
    if (newValue <= oldValue)
```

```
            return false;
        return true;
    }
```

3. Реализуем функцию обработки появления нового источника (раздражителя):

```
public void Notice(Interest i)
{
    StopCoroutine(Investigate());
    interest = i;
    StartCoroutine(Investigate());
}
```

4. Определим пользовательскую функцию проверки источников. Это должна быть ваша собственная реализация, учитывающая наличие соответствующих органов чувств у агента:

```
public virtual IEnumerator Investigate()
{
    // определите свою реализацию здесь
    yield break;
}
```

5. Определим пользовательскую функцию для лидера. Она реализует действия агента, ответственного за раздачу приказов. Это должна быть ваша собственная реализация:

```
public virtual IEnumerator Lead()
{
    // определите свою реализацию здесь
    yield break;
}
```

6. Определим класс источников (раздражителей):

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class InterestSource : MonoBehaviour
{
    public InterestSense sense;
    public float radius;
    public InterestPriority priority;
    public bool isActive;
}
```

7. Реализуем свойство для получения информации о нем:

```
public Interest interest
{
    get
    {
        Interest i;
        i.position = transform.position;
        i.priority = priority;
        i.sense = sense;
        return i;
    }
}
```

8. Определим функцию для проверки возможности влияния источника на органы зрения агента. Ее можно включить в класс агента, но для этого придется изменить часть следующих этапов. Это одна из функций уровня органов чувств:

```
protected bool IsAffectedSight(AgentAwared agent)
{
    // определите свою реализацию здесь
    return false;
}
```

9. Реализуем следующую функцию уровня органов чувств для проверки влияния источника на органы слуха агента. К ней относится все то же самое, что было сказано на предыдущем этапе:

```
protected bool IsAffectedSound(AgentAwared agent)
{
    // определите свою реализацию здесь
    return false;
}
```

10. Определим функцию для получения списка агентов, на которые влияет источник. Она объявлена как виртуальная, чтобы можно было определить ее позднее или просто изменить порядок ее работы:

```
public virtual List<AgentAwared> GetAffected(AgentAwared[] agentList)
{
    List<AgentAwared> affected;
    affected = new List<AgentAwared>();
    Vector3 interPos = transform.position;
    Vector3 agentPos;
```

```
    float distance;
    // дальнейшая реализация описывается ниже
}
```

11. Объявим основной цикл обхода списка агентов и вернем список попадающих под влияние:

```
foreach (AgentAwared agent in agentList)
{
    // дальнейшая реализация описывается ниже
}
return affected;
```

12. Пропустить агента, находящегося вне радиуса действия источника:

```
agentPos = agent.transform.position;
distance = Vector3.Distance(interPos, agentPos);
if (distance > radius)
    continue;
```

13. Проверить возможность влияния источника (раздражителя) на органы чувств агента:

```
bool isAffected = false; switch (sense)
{
    case InterestSense.SIGHT:
        isAffected = IsAffectedSight(agent);
        break;
    case InterestSense.SOUND:
        isAffected = IsAffectedSound(agent);
        break;
}
```

14. Если агент имеет соответствующие органы чувств, добавить его в список:

```
if (!isAffected)
    continue;
affected.Add(agent);
```

15. Определим класс диспетчера источников (раздражителей):

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SensoryManager : MonoBehaviour
```

```
{
    public List<AgentAwared> agents;
    public List<InterestSource> sources;
}
```

16. Реализуем функцию Awake:

```
public void Awake()
{
    agents = new List<AgentAwared>();
    sources = new List<InterestSource>();
}
```

17. Объявим функцию получения списка разведчиков для заданной группы агентов:

```
public List<AgentAwared> GetScouts(AgentAwared[] agents,
                                   int leader = -1)
{
    // дальнейшая реализация описывается ниже
}
```

18. Проверить общее количество агентов:

```
if (agents.Length == 0)
    return new List<AgentAwared>(0);
if (agents.Length == 1)
    return new List<AgentAwared>(agents);
```

19. Удалить лидера по его индексу:

```
List<AgentAwared> agentList;
agentList = new List<AgentAwared>(agents);
if (leader > -1)
    agentList.RemoveAt(leader);
```

20. Рассчитать количество извлекаемых разведчиков:

```
List<AgentAwared> scouts;
scouts = new List<AgentAwared>();
float numAgents = (float)agents.Length;
int numScouts = (int)Mathf.Log(numAgents, 2f);
```

21. Получить список разведчиков, выбирая их случайным образом из списка агентов:

```
while (numScouts != 0)
{
    int numA = agentList.Count;
```

```
    int r = Random.Range(0, numA);
    AgentAwared a = agentList[r];
    scouts.Add(a);
    agentList.RemoveAt(r);
    numScouts--;
}
```

22. Вернуть список разведчиков:

```
return scouts;
```

23. Определим функцию для проверки списка интересующих источников:

```
public void UpdateLoop()
{
    List<AgentAwared> affected;
    AgentAwared leader;
    List<AgentAwared> scouts;
    foreach (InterestSource source in sources)
    {
        // дальнейшая реализация описывается ниже
    }
}
```

24. Пропустить неактивный источник:

```
if (!source.isActive)
    continue;
source.isActive = false;
```

25. Пропустить источник, не влияющий ни на одного агента:

```
affected = source.GetAffected(agents.ToArray());
if (affected.Count == 0)
    continue;
```

26. Выбрать случайного лидера и получить список разведчиков:

```
int l = Random.Range(0, affected.Count);
leader = affected[l];
scouts = GetScouts(affected.ToArray(), l);
```

27. Сообщить лидеру о его назначении, если необходимо:

```
if (leader.Equals(scouts[0]))
    StartCoroutine(leader.Lead());
```

28. Наконец, передать разведчикам сообщение о замеченном источнике (раздражителе), если он соответствует им:

```
foreach (AgentAwared a in scouts)
{
    Interest i = source.interest;
    if (a.IsRelevant(i))
        a.Notice(i);
}
```

Как это работает...

Существует список источников (раздражителей), которые могут воздействовать на органы чувств агентов. Этот список хранится в диспетчере, который глобально обновляет каждый источник, уделяя внимание только активным.

Источник получает список агентов игрового мира и в два этапа извлекает из него только агентов, способных воспринимать его. На первом этапе отбрасываются агенты, находящиеся за пределами радиуса действия источника, а затем остаются только агенты, обладающие соответствующими органами чувств.

Наконец, диспетчер обрабатывает оставшихся агентов, выбирая среди них лидера и разведчиков.

Что дальше...

Стоит отметить, что класс `SensoryManager` действует как концентратор для хранения и упорядочения списка агентов и списка источников (раздражителей), поэтому его экземпляр должен присутствовать в единственном числе. Появление дубликатов может привести к нежелательным сложностям и воздействиям.

Внимание агентов к источникам автоматически переключается диспетчером источников (раздражителей) с учетом приоритетов. Тем не менее его можно переключить принудительно, вызвав общедоступную функцию `Notice`.

Дальнейшие возможности совершенствования рецепта зависят от особенностей конкретной игры. Списки разведчиков могут пересекаться друг с другом, поэтому вы должны предусмотреть обработку в игре этой ситуации. Тем не менее уже разработанная система использует значения приоритетов для принятия решений.

Полезные ссылки

Дополнительную информацию об идеях, лежащих в основе этого рецепта, можно найти в книге Стива Рабина (Steve Rabin) «*Game AI Pro*».

Глава 6

Настольные игры с искусственным интеллектом

В этой главе рассматривается семейство алгоритмов для настольных игр с искусственным интеллектом:

- класс игрового дерева;
- введение в алгоритм Minimax;
- алгоритм Negamax;
- алгоритм AB Negamax;
- алгоритм Negascouting;
- реализация соперника для игры в крестики-нолики;
- реализация соперника для игры в шашки.

Введение

В этой главе рассматривается семейство алгоритмов искусственного интеллекта, применяемых при разработке настольных игр. Они основываются на идее игрового дерева (графа), охватывающего оценку состояния и принятие решения о переходе в соседние (по отношению к нему) состояния. Эти алгоритмы предназначены для создания настольных игр с двумя соперниками. Но с помощью некоторых усилий часть из них можно расширить для применения в играх с большим количеством игроков.

Класс игрового дерева

Состояние игры можно представить многими различными способами, но здесь будут рассмотрены расширяемые классы, подходящие для использования алгоритмов искусственного интеллекта высокого уровня.

Подготовка

Потребуется четкое понимание объектно-ориентированного программирования, особенно наследования и полиморфизма. Это вызвано тем, что здесь будут созданы универсальные функции, применимые к решению задач ряда настольных игр, а затем реализованы конкретные подклассы, наследующие и конкретизирующие эти функции.

Как это реализовать...

Создадим два класса для представления игрового дерева, как описывается ниже:

1. Определим абстрактный класс хода `Move`:

```
using UnityEngine;
using System.Collections;

public abstract class Move
{

}
```

2. Определим псевдоабстрактный класс игрового поля `Board`:

```
using UnityEngine;
using System.Collections;

public class Board
{
    protected int player;
    // дальнейшая реализация описывается ниже
}
```

3. Определим конструктор по умолчанию:

```
public Board()
{
    player = 1;
}
```

4. Реализуем виртуальную функцию для получения следующих вероятных ходов:

```
public virtual Move[] GetMoves()
{
    return new Move[0];
}
```

5. Реализуем виртуальную функцию хода в настольной игре:

```
public virtual Board MakeMove(Move m)
{
    return new Board();
}
```

6. Определим виртуальную функцию для проверки окончания игры:

```
public virtual bool IsGameOver()
{
    return true;
}
```

7. Реализуем виртуальную функцию для получения текущего игрока:

```
public virtual int GetCurrentPlayer()
{
    return player;
}
```

8. Реализуем виртуальную функцию для проверки значения игры для данного игрока:

```
public virtual float Evaluate(int player)
{
    return Mathf.NegativeInfinity;
}
```

9. Также реализуем виртуальную функцию для проверки значения игры для текущего игрока:

```
public virtual float Evaluate()
{
    return Mathf.NegativeInfinity;
}
```

Как это работает...

Здесь заложен фундамент для алгоритмов, что описываются ниже. Класс `Board` выступает в качестве узла, представляющего текущее состояние игры, а класс `Move` служит ребром. Функция `GetMoves` из-

влекает ребера, ведущие к соседним состояниям игры относительно текущего.

Полезные ссылки

Более подробные сведения о теоретических идеях, лежащих в основе методов, описанных в этой главе, можно найти в книге Рассела (Russel) и Норвига (Norvig) «*Artificial Intelligence: a Modern Approach*»[1] и в книге Яна Миллингтона (Ian Millington) «*Artificial Intelligence for Games*» (настольные игры).

Введение в алгоритм Minimax

Алгоритм Minimax основывается на идее минимизации потерь, возможных в худшем случае (максимальные потери). Алгоритм Minimax применяется не только в разработке и теории игр, но также в статистике, теории принятия решений и философии.

Этот метод первоначально сформулирован в виде теории игры для двух игроков с нулевой суммой, когда выигрыш одного игрока проводит к появлению потерь у его противника. Однако в данном случае он обладает достаточной гибкостью для работы и с большим количеством игроков.

Подготовка

Для знакомства с алгоритмом важно понимать разницу между динамическими и статическими методами, а также уметь использовать рекурсию. Динамические методы привязаны к экземпляру класса, в то время как статические методы – к самому классу. Статический метод можно вызывать без создания экземпляра объекта. Он отлично подходит для алгоритмов общего назначения, подобных тем, что будут использоваться в этом рецепте.

В случае использования рекурсии не всегда имеется четкое понимание (в отличие от итераций), что она представляет итеративный процесс, требующий реализации базового (также называемого условием завершения) и рекурсивного случаев (продолжающего итерации).

Как это реализовать...

Определим базовый класс для обработки всех основных алгоритмов и реализуем функцию `Minimax`:

---

[1] *Рассел С., Норвиг П.* Искусственный интеллект: современный подход. 2-е изд. М.: Вильямс, 2015. ISBN: 978-5-8459-1968-7. – *Прим. ред.*

1. Определим класс BoardAI:

```
using UnityEngine;
using System.Collections;

public class BoardAI
{

}
```

2. Объявим функцию Minimax:

```
public static float Minimax(
        Board board,
        int player,
        int maxDepth,
        int currentDepth,
        ref Move bestMove)
{
    // дальнейшая реализация описывается ниже
}
```

3. Определим базовый случай:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate(player);
```

4. Зададим начальные значения в зависимости от игрока:

```
bestMove = null;
float bestScore = Mathf.Infinity;
if (board.GetCurrentPlayer() == player)
    bestScore = Mathf.NegativeInfinity;
```

5. Выполним перебор всех возможных ходов и вернем лучший:

```
foreach (Move m in board.GetMoves())
{
    // дальнейшая реализация описывается ниже
}
return bestScore;
```

6. Создадим новое состояние игры с учетом текущего хода:

```
Board b = board.MakeMove(m);
float currentScore;
Move currentMove = null;
```

7. Начнем рекурсию:

```
currentScore = Minimax(b, player, maxDepth, currentDepth + 1,
                       ref currentMove);
```

8. Проверим счет текущего игрока:

```
if (board.GetCurrentPlayer() == player)
{
    if (currentScore > bestScore)
    {
        bestScore = currentScore;
        bestMove = currentMove;
    }
}
```

9. Проверим счет соперника:

```
else
{
    if (currentScore < bestScore)
    {
        bestScore = currentScore;
        bestMove = currentMove;
    }
}
```

Как это работает...

Алгоритм работает как ограниченный поиск в глубину. На каждом шаге выбирается ход, приводящий к достижению максимального счета, и предполагается, что соперник выберет вариант, ведущий к уменьшению счета игрока до минимального значения, пока не будет достигнут конечный узел (лист дерева).

Перебор ходов осуществляется с использованием рекурсии и эвристики выбора или предугадывания варианта, реализованного в виде функции `Evaluate`.

Полезные ссылки

- Рецепт *«Класс игрового дерева»* выше.

Алгоритм Negamax

Когда в игре с нулевой суммой участвуют лишь два игрока, алгоритм Minimax можно усовершенствовать, воспользовавшись тем обстоятельством, что потеря для одного из игроков является выгодой для другого. То есть усовершенствованный алгоритм приведет к тем же результатам, что и алгоритм Minimax, но при этом отпадает необходимость отслеживать ходы.

Подготовка

Для знакомства с алгоритмом важно понимать разницу между динамическими и статическими методами, а также уметь использовать рекурсию. Динамические методы привязаны к экземпляру класса, в то время как статические методы – к самому классу. Статический метод можно вызывать без создания экземпляра объекта. Он отлично подходит для алгоритмов общего назначения, подобных тем, что будут использоваться в этом рецепте.

В случае использования рекурсии не всегда имеется четкое понимание (в отличие от итераций), что она представляет итеративный процесс, требующий реализации базового (также называемого условием завершения) и рекурсивного случаев (продолжающего итерации).

Как это реализовать...

Добавим в класс `BoardAI` новую функцию:

1. Определим функцию `Negamax`:

```
public static float Negamax(
        Board board,
        int maxDepth,
        int currentDepth,
        ref Move bestMove)
{
    // дальнейшая реализация описывается ниже
}
```

2. Проверим базовый случай:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate();
```

3. Зададим начальные значения:

```
bestMove = null;
float bestScore = Mathf.NegativeInfinity;
```

4. Выполним перебор всех возможных ходов и вернем лучший:

```
foreach (Move m in board.GetMoves())
{
    // дальнейшая реализация описывается ниже
}
return bestScore;
```

5. Создадим новое состояние игры с учетом текущего хода:

```
Board b = board.MakeMove(m);
float recursedScore;
Move currentMove = null;
```

6. Начнем рекурсию:

```
recursedScore = Negamax(b, maxDepth, currentDepth + 1,
                        ref currentMove);
```

7. Установим текущий счет, обновим лучший счет и выполним ход, если необходимо:

```
float currentScore = -recursedScore;
if (currentScore > bestScore)
{
    bestScore = currentScore;
    bestMove = m;
}
```

Как это работает...

Базовый алгоритм действует подобно предыдущему, но обладает некоторыми преимуществами. На каждом шаге рекурсии счет сравнивается со счетом предыдущих ходов с обратным знаком. Вместо выбора наилучшего варианта алгоритм изменяет знак счета, устраняя необходимость отслеживать результат хода.

Дополнительная информация...

Поскольку алгоритм Negamax меняет игроков местами на каждом шаге, функция оценки не содержит параметров.

Полезные ссылки

- Рецепт «*Класс игрового дерева*».
- Рецепт «*Алгоритм Minimax*».

Алгоритм AB Negamax

Алгоритм Negamax также можно усовершенствовать. Несмотря на всю его эффективность, он имеет один недостаток, просматривая больше узлов (например, позиции настольной игры), чем это необходимо. Для преодоления этой проблемы алгоритм Negamax используется в сочетании со стратегией поиска, которую для краткости называют «альфа-бета» (alpha-beta).

Подготовка

Для знакомства с алгоритмом важно понимать разницу между динамическими и статическими методами, а также уметь использовать рекурсию. Динамические методы привязаны к экземпляру класса, в то время как статические методы – к самому классу. Статический метод можно вызывать без создания экземпляра объекта. Он отлично подходит для алгоритмов общего назначения, подобных тем, что будут использоваться в этом рецепте.

В случае использования рекурсии не всегда имеется четкое понимание (в отличие от итераций), что она представляет итеративный процесс, требующий реализации базового (также называемого условием завершения) и рекурсивного случаев (продолжающего итерации).

Как это реализовать...

Добавим в класс BoardAI новую функцию:

1. Определим функцию ABNegamax:

```
public static float ABNegamax(
        Board board,
        int player,
        int maxDepth,
        int currentDepth,
        ref Move bestMove,
        float alpha,
        float beta)
{
    // дальнейшая реализация описывается ниже
}
```

2. Проверим базовый случай:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate(player);
```

3. Зададим начальные значения:

```
bestMove = null;
float bestScore = Mathf.NegativeInfinity;
```

4. Выполним перебор всех возможных ходов и вернем лучший:

```
foreach (Move m in board.GetMoves())
{
    // дальнейшая реализация описывается ниже
}
return bestScore;
```

5. Создадим новое состояние игры с учетом текущего хода:

```
Board b = board.MakeMove(m);
```

6. Зададим значения для вызова рекурсии:

```
float recursedScore;
Move currentMove = null;
int cd = currentDepth + 1;
float max = Mathf.Max(alpha, bestScore);
```

7. Начнем рекурсию:

```
recursedScore = ABNegamax(b, player, maxDepth, cd,
                          ref currentMove, -beta, max);
```

8. Установим текущий счет, обновим лучший счет и выполним ход, если необходимо. При необходимости остановим итерации:

```
float currentScore = -recursedScore;
if (currentScore > bestScore)
{
    bestScore = currentScore;
    bestMove = m;

    if (bestScore >= beta)
        return bestScore;
}
```

Как это работает...

Основная идея алгоритма была описана выше, поэтому сосредоточимся на стратегии поиска. Существуют два значения: альфа и бета. Альфа-значение является самым низким счетом, которого игрок сможет достичь, поэтому из рассмотрения исключаются любые действия противника, направленные на его уменьшение. Аналогично бета-значение является верхним пределом счета, и независимо от того, насколько заманчивым выглядит его достижение, алгоритм предполагает, что противник не позволит получить его.

Поскольку соперники делают ходы поочередно (минимум и максимум), на каждом шаге требуется проверить только одно значение.

Полезные ссылки

- Рецепт *«Класс игрового дерева»*.
- Рецепт *«Алгоритм Minimax»*.
- Рецепт *«Алгоритм Negamax»*.

Алгоритм Negascouting

Подключение стратегий поиска расчищает место для новых задач. Алгоритм Negascouting появился в результате сужения диапазона поиска с применением улучшенной отсекающей эвристики. Он основан на идее **окна поиска**, определяющей интервал между значениями альфа и бета. То есть сужение окна поиска увеличивает вероятность исключения ветвей.

Подготовка

Для знакомства с алгоритмом важно понимать разницу между динамическими и статическими методами, а также уметь использовать рекурсию. Динамические методы привязаны к экземпляру класса, в то время как статические методы – к самому классу. Статический метод можно вызывать без создания экземпляра объекта. Он отлично подходит для алгоритмов общего назначения, подобных тем, что будут использоваться в этом рецепте.

В случае использования рекурсии не всегда имеется четкое понимание (в отличие от итераций), что она представляет итеративный процесс, требующий реализации базового (также называемого условием завершения) и рекурсивного случаев (продолжающего итерации).

Как это реализовать...

Добавим в класс `BoardAI` новую функцию:

1. Определим функцию `ABNegascout`:

```
public static float ABNegascout (
        Board board,
        int player,
        int maxDepth,
        int currentDepth,
        ref Move bestMove,
        float alpha,
        float beta)
{
    // дальнейшая реализация описывается ниже
}
```

2. Проверим базовый случай:

```
if (board.IsGameOver() || currentDepth == maxDepth)
    return board.Evaluate(player);
```

3. Зададим начальные значения:

```
bestMove = null;
float bestScore = Mathf.NegativeInfinity;
float adaptiveBeta = beta;
```

4. Выполним перебор всех возможных ходов и вернем лучший:

```
foreach (Move m in board.GetMoves())
{
    // дальнейшая реализация описывается ниже
}
return bestScore;
```

5. Создадим новое состояние игры с учетом текущего хода:

```
Board b = board.MakeMove(m);
```

6. Зададим значения для рекурсии:

```
Move currentMove = null;
float recursedScore;
int depth = currentDepth + 1;
float max = Mathf.Max(alpha, bestScore);
```

7. Вызовем рекурсию:

```
recursedScore = ABNegamax(b, player, maxDepth, depth,
                          ref currentMove, -adaptiveBeta, max);
```

8. Установим текущий счет и проверим его допустимость:

```
float currentScore = -recursedScore;
if (currentScore > bestScore)
{
    // дальнейшая реализация описывается ниже
}
```

9. Проверим возможность исключения:

```
if (adaptiveBeta == beta || currentDepth >= maxDepth - 2)
{
    bestScore = currentScore;
    bestMove = currentMove;
}
```

10. В противном случае поищем вокруг:

```
else
{
    float negativeBest;
```

```
        negativeBest = ABNegascout(b, player, maxDepth, depth,
                                   ref bestMove, -beta, -currentScore);
        bestScore = -negativeBest;
    }
```

11. При необходимости завершим цикл. В противном случае обновим адаптивное значение:

```
if (bestScore >= beta)
    return bestScore;

adaptiveBeta = Mathf.Max(alpha, bestScore) + 1f;
```

Как это работает...

Этот алгоритм использует оценку первого хода для каждого узла. Следующие ходы оцениваются в следующей итерации, с использованием окна, суженного после первого хода. Если итерация завершилась неудачей, она повторяется для полной ширины окна. В результате происходит отсечение большой части ветвей и удается избежать неудач.

Полезные ссылки

- Рецепт *«Алгоритм AB Negamax»*.

Реализация соперника для игры в крестики-нолики

Для иллюстрации применения предыдущих рецептов рассмотрим один из способов реализации соперника в популярной игре в крестики-нолики. Это не только поможет расширить базовые классы, но и даст возможность освоить создание соперников в других настольных играх.

Подготовка

Определим конкретный класс хода для настольный игры в крестики-нолики, наследующий родительский класс, который был создан в начале этой главы:

```
using UnityEngine;
using System.Collections;

public class MoveTicTac : Move
{
    public int x;
```

```
    public int y;
    public int player;

    public MoveTicTac(int x, int y, int player)
    {
        this.x = x;
        this.y = y;
        this.player = player;
    }
}
```

Как это реализовать...

Определим новый класс, наследующий класс Board, который переопределет методы родителя и добавляет новые.

1. Определим класс BoardTicTac, наследующий класс Board, и добавим в него свойства для хранения значений настольной игры:

```
using UnityEngine; using System;
using System.Collections;
using System.Collections.Generic;

public class BoardTicTac : Board
{
    protected int[,] board;
    protected const int ROWS = 3;
    protected const int COLS = 3;
}
```

2. Определим конструктор по умолчанию:

```
public BoardTicTac(int player = 1)
{
    this.player = player;
    board = new int[ROWS, COLS];
    board[1,1] = 1;
}
```

3. Определим функцию выбора следующего игрока согласно очередности:

```
private int GetNextPlayer(int p)
{
    if (p == 1)
        return 2;
    return 1;
}
```

4. Определим функцию оценки позиции с точки зрения заданного игрока:

```
private float EvaluatePosition(int x, int y, int p)
{
    if (board[y, x] == 0)
        return 1f;
    else if (board[y, x] == p)
        return 2f;
    return -1f;
}
```

5. Определим функцию оценки соседних позиций с точки зрения заданного игрока:

```
private float EvaluateNeighbours(int x, int y, int p)
{
    float eval = 0f;
    int i, j;
    for (i = y - 1; i < y + 2; y++)
    {
        if (i < 0 || i >= ROWS)
            continue;
        for (j = x - 1; j < x + 2; j++)
        {
            if (j < 0 || j >= COLS)
                continue;
            if (i == j)
                continue;
            eval += EvaluatePosition(j, i, p);
        }
    }
    return eval;
}
```

6. Реализуем конструктор для создания новых состояний с их стоимостями:

```
public BoardTicTac(int[,] board, int player)
{
    this.board = board;
    this.player = player;
}
```

7. Переопределим метод получения доступных ходов в текущем состоянии:

```
public override Move[] GetMoves()
{
    List<Move> moves = new List<Move>();
    int i;
    int j;
    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            if (board[i, j] != 0)
                continue;
            MoveTicTac m = new MoveTicTac(j, i, player);
            moves.Add(m);
        }
    }
    return moves.ToArray();
}
```

8. Переопределим функцию получения нового состояния для заданного хода:

```
public override Board MakeMove(Move m)
{
    MoveTicTac move = (MoveTicTac)m;
    int nextPlayer = GetNextPlayer(move.player);
    int[,] copy = new int[ROWS, COLS];
    Array.Copy(board, 0, copy, 0, board.Length);
    copy[move.y, move.x] = move.player;
    BoardTicTac b = new BoardTicTac(copy, nextPlayer);
    return b;
}
```

9. Определим функцию оценки текущего состояния с точки зрения заданного игрока:

```
public override float Evaluate(int player)
{
    float eval = 0f; int i, j;
    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            eval += EvaluatePosition(j, i, player);
            eval += EvaluateNeighbours(j, i, player);
```

```
            }
        }
        return eval;
    }
```

10. Реализуем функцию оценки текущего состояния с точки зрения текущего игрока:

```
public override float Evaluate()
{
    float eval = 0f; int i, j;
    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            eval += EvaluatePosition(j, i, player);
            eval += EvaluateNeighbours(j, i, player);
        }
    }
    return eval;
}
```

Как это работает...

Мы определили новый класс хода для конкретной настольной игры, который хорошо сочетается с базовыми алгоритмами, поскольку они используют его только на высоком уровне, как структуру данных. Соль рецепта заключается в переопределении виртуальных функций класса `Board` для решения проблем моделирования. Для хранения ходов игроков используется двумерный целочисленный массив (0 представляет незанятую клетку), а для определения стоимости заданного состояния относительно соседних применяются эвристические методы.

Что дальше...

Эвристические функции оценки счета (состояния) являются приемлемыми, но не оптимальными. Поэтому исследуем эту проблему и переработаем реализации вышеупомянутых функций для получения более совершенного соперника.

Полезные ссылки

- Рецепт *«Класс игрового дерева»*.

Реализация соперника для игры в шашки

Этот продвинутый пример иллюстрирует расширение приведенных выше рецептов. Здесь рассматривается модель настольной игры в шашки, реализующая функции для нашего фреймворка настольных игр с применением искусственного интеллекта.

В данном примере используются шахматная доска (8 × 8) и соответствующее ей количество шашек (12). Однако его легко можно настроить для других значений, если потребуется другой размер доски.

Подготовка

Прежде всего нам нужно определить новый класс хода `MoveDraughts` для данного конкретного случая:

```
using UnityEngine;
using System.Collections;

public class MoveDraughts : Move
{
    public PieceDraughts piece;
    public int x;
    public int y;
    public bool success;
    public int removeX;
    public int removeY;
}
```

Эта структура хранит шашку, перемещаемую в новую позицию с координатами `x` и `y`, если при перемещении шашка побьет шашку соперника, соответствующая шашка будет удалена.

Как это реализовать...

Мы реализуем два основных класса для моделирования шашек и доски соответственно. Это будет длительный процесс, поэтому внимательно ознакомьтесь с каждым его этапом:

1. Создадим новый файл с именем `PieceDraughts.cs` и добавим в него следующие операторы:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
```

2. Добавим перечисление PieceColor:

```
public enum PieceColor
{
    WHITE,
    BLACK
};
```

3. Добавим перечисление PieceType:

```
public enum PieceType
{
    MAN,
    KING
};
```

4. Начнем конструирование класса PieceDraughts:

```
public class PieceDraughts : MonoBehaviour
{
    public int x;
    public int y;
    public PieceColor color;
    public PieceType type;
    // дальнейшая реализация описывается ниже
}
```

5. Определим функцию настройки шашек:

```
public void Setup(int x, int y, PieceColor color, PieceType type =
PieceType.MAN)
{
    this.x = x;
    this.y = y;
    this.color = color;
    this.type = type;
}
```

6. Определим функцию перемещения шашек по доске:

```
public void Move (MoveDraughts move, ref PieceDraughts [,] board)
{
    board[move.y, move.x] = this;
    board[y, x] = null;
    x = move.x;
    y = move.y;
    // дальнейшая реализация описывается ниже
}
```

7. Если при выполнении хода шашка бьет шашку соперника, удалить соответствующую шашку:

```
if (move.success)
{
    Destroy(board[move.removeY, move.removeX]);
    board[move.removeY, move.removeX] = null;
}
```

8. Остановить процесс, если шашка прошла в дамки:

```
if (type == PieceType.KING)
    return;
```

9. Изменить тип шашки, если это обычная шашка и она дошла до противоположного края доски:

```
int rows = board.GetLength(0);
if (color == PieceColor.WHITE && y == rows)
    type = PieceType.KING;
if (color == PieceColor.BLACK && y == 0)
    type = PieceType.KING;
```

10. Определим функцию для проверки выполнения хода в пределах доски:

```
private bool IsMoveInBounds(int x, int y,
    ref PieceDraughts[,] board)
{
    int rows = board.GetLength(0);
    int cols = board.GetLength(1);
    if (x < 0 || x >= cols || y < 0 || y >= rows)
        return false;
    return true;
}
```

11. Определим общую функцию для получения возможных ходов:

```
public Move[] GetMoves(ref PieceDraughts[,] board)
{
    List<Move> moves = new List<Move>();
    if (type == PieceType.KING)
        moves = GetMovesKing(ref board);
    else
        moves = GetMovesMan(ref board);
    return moves.ToArray();
}
```

12. Начнем реализацию функции получения ходов для обычной шашки:

```
private List<Move> GetMovesMan(ref PieceDraughts[,] board)
{
    // дальнейшая реализация описывается ниже
}
```

13. Добавим переменную для хранения двух возможных ходов:

```
List<Move> moves = new List<Move>(2);
```

14. Определим переменную для хранения двух возможных горизонтальных вариантов:

```
int[] moveX = new int[] { -1, 1 };
```

15. Определим переменную для хранения вертикального направления, зависящего от цвета шашки:

```
int moveY = 1;
if (color == PieceColor.BLACK)
    moveY = -1;
```

16. Реализуем цикл обхода двух возможных вариантов и вернем имеющиеся ходы. Тело цикла будет реализовано ниже:

```
foreach (int mX in moveX)
{
    // дальнейшая реализация описывается ниже
}
return moves;
```

17. Объявим две новые переменные для расчета следующей позиции:

```
int nextX = x + mX;
int nextY = y + moveY;
```

18. Проверим выход за пределы доски:

```
if (!IsMoveInBounds(nextX, y, ref board))
    continue;
```

19. Перейдем к следующему варианту, если ход блокируется шашкой того же цвета:

```
PieceDraughts p = board[moveY, nextX];
if (p != null && p.color == color)
    continue;
```

20. Создадим новый ход для добавления в список, поскольку он допустим:

```
MoveDraughts m = new MoveDraughts();
m.piece = this;
```

21. Создадим обычный ход, если позиция доступна:

```
if (p == null)
{
    m.x = nextX;
    m.y = nextY;
}
```

22. В противном случае проверим, может ли шашка взять шашку соперника, и соответствующим образом изменим ход:

```
else
{
    int hopX = nextX + mX;
    int hopY = nextY + moveY;
    if (!IsMoveInBounds(hopX, hopY, ref board))
        continue;
    if (board[hopY, hopX] != null)
        continue;
    m.y = hopX;
    m.x = hopY;
    m.success = true;
    m.removeX = nextX;
    m.removeY = nextY;
}
```

23. Добавим ход в список:

```
moves.Add(m);
```

24. Начнем реализацию функции получения доступных ходов для дамки:

```
private List<Move> GetMovesKing(ref PieceDraughts[,] board)
{
    // дальнейшая реализация описывается ниже
}
```

25. Объявим переменную для хранения возможных ходов:

```
List<Move> moves = new List<Move>();
```

26. Создадим переменные для поиска в четырех направлениях:

```
int[] moveX = new int[] { -1, 1 };
int[] moveY = new int[] { -1, 1 };
```

27. Начнем реализацию цикла проверки всех возможных ходов. Тело внутреннего цикла будет реализовано на следующем этапе:

```
foreach (int mY in moveY)
{
    foreach (int mX in moveX)
    {
        // дальнейшая реализация описывается ниже
    }
}
return moves;
```

28. Создадим переменные для проверки ходов и достижения границ доски:

```
int nowX = x + mX;
int nowY = y + mY;
```

29. Добавим цикл проверки хода в этом направлении, пока не будет достигнута граница доски:

```
while (IsMoveInBounds(nowX, nowY, ref board))
{
    // дальнейшая реализация описывается ниже
}
```

30. Получить ссылку на шашку по позиции:

```
PieceDraughts p = board[nowY, nowX];
```

31. Если эта шашка того же цвета, дальше идти нельзя:

```
if (p != null && p.color == color)
    break;
```

32. Определим переменную для создания новых доступных ходов:

```
MoveDraughts m = new MoveDraughts();
m.piece = this;
```

33. Создадим обычный ход, если позиция доступна:

```
if (p == null)
{
    m.x = nowX;
    m.y = nowY;
}
```

34. В противном случае проверим, можно ли взять шашку соперника, и соответствующим образом изменим ход:

```
else
{
    int hopX = nowX + mX;
    int hopY = nowY + mY;
    if (!IsMoveInBounds(hopX, hopY, ref board))
        break;
    m.success = true;
    m.x = hopX;
    m.y = hopY;
    m.removeX = nowX;
    m.removeY = nowY;
}
```

35. Добавим ход и сделаем еще шаг в текущем направлении:

```
moves.Add(m);
nowX += mX;
nowY += mY;
```

36. Создадим новый класс `BoardDraughts` в новом файле:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class BoardDraughts : Board
{
    public int size = 8;
    public int numPieces = 12;
    public GameObject prefab;
    protected PieceDraughts[,] board;
}
```

37. Реализуем функцию `Awake`:

```
void Awake()
{
    board = new PieceDraughts[size, size];
}
```

38. Начнем реализацию функции `Start`. Обратите внимание, что она может отличаться для разных представлений пространства игры:

```
void Start()
{
    // Инициализация и настройка доски
    // ваша реализация может отличаться

    // дальнейшая реализация описывается ниже
}
```

39. Выведем сообщение об ошибке, если к шаблонному объекту не присоединен сценарий PieceDraught:

```
PieceDraughts pd = prefab.GetComponent<PieceDraughts>();
if (pd == null)
{
    Debug.LogError("No PieceDraught component detected");
    return;
}
```

40. Добавим переменные циклов:

```
int i;
int j;
```

41. Реализуем цикл для расстановки белых шашек:

```
int piecesLeft = numPieces;
for (i = 0; i < size; i++)
{
    if (piecesLeft == 0)
        break;
    int init = 0;
    if (i % 2 != 0)
        init = 1;
    for (j = init; j < size; j+=2)
    {
        if (piecesLeft == 0)
            break;
        PlacePiece(j, i);
        piecesLeft--;
    }
}
```

42. Реализуем цикл расстановки черных шашек:

```
piecesLeft = numPieces;
for (i = size - 1; i >= 0; i--)
{
    if (piecesLeft == 0)
        break;
    int init = 0;
    if (i % 2 != 0)
        init = 1;
    for (j = init; j < size; j+=2)
    {
        if (piecesLeft == 0)
            break;
```

```
            PlacePiece(j, i);
            piecesLeft--;
        }
    }
```

43. Реализуем функцию установки конкретной шашки. Может отличаться и зависит от способа визуализации:

```
private void PlacePiece(int x, int y)
{
    // преобразования, связанные
    // с размещением в пространстве
    Vector3 pos = new Vector3();
    pos.x = (float)x;
    pos.y = -(float)y;
    GameObject go = GameObject.Instantiate(prefab);
    go.transform.position = pos;
    PieceDraughts p = go.GetComponent<PieceDraughts>();
    p.Setup(x, y, color);
    board[y, x] = p;
}
```

44. Реализуем функцию Evaluate без параметров:

```
public override float Evaluate()
{
    PieceColor color = PieceColor.WHITE;
    if (player == 1)
        color = PieceColor.BLACK;
    return Evaluate(color);
}
```

45. Реализуем функцию Evaluate с параметром:

```
public override float Evaluate(int player)
{
    PieceColor color = PieceColor.WHITE;
    if (player == 1)
        color = PieceColor.BLACK;
    return Evaluate(color);
}
```

46. Начнем реализацию общей функции оценки:

```
private float Evaluate(PieceColor color)
{
    // дальнейшая реализация описывается ниже
}
```

47. Определим переменные для хранения оценки и присвоенных баллов:

```
float eval = 1f;
float pointSimple = 1f;
float pointSuccess = 5f;
```

48. Создадим переменные для хранения границ доски:

```
int rows = board.GetLength(0);
int cols = board.GetLength(1);
```

49. Определим переменные циклов:

```
int i;
int j;
```

50. Выполним обход доски в поисках ходов:

```
for (i = 0; i < rows; i++)
{
    for (j = 0; j < cols; j++)
    {
        PieceDraughts p = board[i, j];
        if (p == null)
            continue;
        if (p.color != color)
            continue;
        Move[] moves = p.GetMoves(ref board);
        foreach (Move mv in moves)
        {
            MoveDraughts m = (MoveDraughts)mv;
            if (m.success)
                eval += pointSuccess;
            else
                eval += pointSimple;
        }
    }
}
```

51. Вернем оценку:

```
return eval;
```

52. Начнем функцию выбора доступных на доске ходов:

```
public override Move[] GetMoves()
{
    // дальнейшая реализация описывается ниже
}
```

53. Определим переменные для хранения ходов, границ доски и счетчиков:

```
List<Move> moves = new List<Move>();
int rows = board.GetLength(0);
int cols = board.GetLength(1);
int i;
int j;
```

54. Получим ходы для всех шашек на доске:

```
for (i = 0; i < rows; i++)
{
    for (j = 0; i < cols; j++)
    {
        PieceDraughts p = board[i, j];
        if (p == null)
            continue;
        moves.AddRange(p.GetMoves(ref board));
    }
}
```

55. Вернем найденные ходы:

```
return moves.ToArray();
```

Как это работает...

Эта настольная игра моделируется примерно так же, как предыдущая, но сам процесс игры, определяемый ее правилами, гораздо сложнее. Перемещения связаны с ходами шашек, что создает каскадный эффект, который следует обрабатывать с особой тщательностью. Ходы делятся на два типа в зависимости от цвета шашки и ее вида.

Но, как видите, правила на верхнем уровне остаются неизменными. Требуются лишь терпение и понимание, чтобы разработать хорошие функции оценки и процедуры получения доступных на доске ходов.

Что дальше...

Функцию `Evaluate` нельзя считать совершенной. Мы реализовали эвристический алгоритм, основанный исключительно на количестве ходов и взятых шашек соперника, который можно усовершенствовать, чтобы исключить ходы, после которых шашка игрока может быть взята следующим ходом соперника.

Кроме того, следует внести собственные изменения в функцию `PlacePiece` класса `BoardDraughts`. Мы реализовали прямой метод, который, скорее всего, не пригоден для пространственной настройки игры.

Глава 7

Механизмы обучения

В этой главе рассматриваются вопросы, имеющие отношения к автоматизации обучения:

- предугадывание действий с помощью алгоритма прогнозирования N-Gram;
- усовершенствованный иерархический алгоритм N-Gram;
- использование классификаторов Байеса;
- использование деревьев принятия решений;
- использование закрепления рефлекса;
- обучение с помощью искусственных нейронных сетей.

Введение

Эта глава посвящена проблеме машинного обучения. Это очень обширная и важная область, в которой даже на изучение тривиальных вопросов приходится затрачивать большое количество времени, поскольку ее механизмы требуют доработки и проведения экспериментов.

Однако рецепты, представленные в этой главе, станут отличной отправной точкой для освоения и применения механизмов машинного обучения в играх. Их можно использовать различными способами, но наилучшие, как правило, требуют наиболее сложной настройки.

Наконец, в дополнение к знакомству с рецептами рекомендуется прочесть специализированные книги по этому вопросу, чтобы вооружиться теоретическими знаниями, освещение которых выходит за рамки этой главы.

Предугадывание действий с помощью алгоритма прогнозирования N-Gram

Предугадывание действий – отличный способ перейти от случайного выбора к выбору, основанному на прошлых действиях. Один из ва-

риантов обучения заключается в использовании вероятностей для предсказания действий игрока, чем и занимается алгоритм прогнозирования N-Gram.

Для предугадывания следующего выбора алгоритм прогнозирования N-Gram хранит сведения о вероятностях решений (которыми обычно являются ходы), учитывая все комбинации вариантов предыдущих *n* ходов.

Подготовка

При реализации этого рецепта используются обобщенные типы. Поэтому желательно иметь хотя бы общее представление о том, как они работают, поскольку очень важно использовать их правильно.

Прежде всего нужно реализовать тип данных для хранения действий и их вероятностей, что будет сделано с помощью класса `KeyDataRecord`.

Создадим файл `KeyDataRecord.cs` и добавим в него следующий код:

```
using System.Collections;
using System.Collections.Generic;

public class KeyDataRecord<T>
{
    public Dictionary<T, int> counts;
    public int total;

    public KeyDataRecord()
    {
        counts = new Dictionary<T, int>();
    }
}
```

Как это реализовать...

Реализацию алгоритма прогнозирования N-Gram можно разделить на пять крупных этапов:

1. Определим шаблонный класс в файле с таким же именем:

```
using System.Collections;
using System.Collections.Generic;
using System.Text;

public class NGramPredictor<T>
{
    private int nValue;
    private Dictionary<string, KeyDataRecord<T>> data;
}
```

2. Реализуем конструктор для инициализации свойств:

```
public NGramPredictor(int windowSize)
{
    nValue = windowSize + 1;
    data = new Dictionary<string, KeyDataRecord<T>>();
}
```

3. Реализуем статическую функцию для преобразования набора действий в строковый ключ:

```
public static string ArrToStrKey(ref T[] actions)
{
    StringBuilder builder = new StringBuilder();
    foreach (T a in actions)
    {
        builder.Append(a.ToString());
    }
    return builder.ToString();
}
```

4. Определим функцию регистрации набора последовательностей:

```
public void RegisterSequence(T[] actions)
{
    string key = ArrToStrKey(ref actions);
    T val = actions[nValue - 1];
    if (!data.ContainsKey(key))
        data[key] = new KeyDataRecord<T>();
    KeyDataRecord<T> kdr = data[key];
    if (kdr.counts.ContainsKey(val))
        kdr.counts[val] = 0;
    kdr.counts[val]++;
    kdr.total++;
}
```

5. И наконец, реализуем функцию прогнозирования действий:

```
public T GetMostLikely(T[] actions)
{
    string key = ArrToStrKey(ref actions);
    KeyDataRecord<T> kdr = data[key];
    int highestVal = 0;
    T bestAction = default(T);
    foreach (KeyValuePair<T,int> kvp in kdr.counts)
    {
        if (kvp.Value > highestVal)
```

```
            {
                bestAction = kvp.Key;
                highestVal = kvp.Value;
            }
        }
        return bestAction;
    }
```

Как это работает...

Алгоритм прогнозирования регистрирует набор действий в соответствии с размером окна (количество действий, регистрируемых для подготовки прогноза) и присваивает им результирующие значения. Например, при размере окна, равном 3, первые три действия сохраняются как ключ для предсказания четвертого действия, следующего за ними.

Функция предсказания вычисляет вероятность этого действия с учетом набора предыдущих действий. Чем больше зарегистрированных действий, тем точнее прогноз (с некоторыми ограничениями).

Что дальше...

Имейте в виду, что для объекта типа `T` необходимо переопределить функции `ToString` и `Equals`, чтобы обеспечить их корректную работу в качестве индекса внутренних словарей.

Усовершенствованный иерархический алгоритм N-Gram

Алгоритм N-Gram можно усовершенствовать, добавив подготовку нескольких прогнозов с диапазоном от 1 до *n* действий и выбирая наиболее предпочтительное после сравнения предположений.

Подготовка

Прежде чем преступить к реализации иерархического алгоритма прогнозирования N-Gram, необходимо произвести некоторые корректировки. Добавим следующий метод в класс `NGramPredictor`:

```
public int GetActionsNum(ref T[] actions)
{
    string key = ArrToStrKey(ref actions);
    if (!data.ContainsKey(key))
        return 0;
    return data[key].total;
}
```

Как это реализовать...

Подобно предыдущей реализации алгоритма N-Gram, реализация иерархической версии алгоритма требует лишь нескольких этапов:

1. Определим новый класс:

```
using System;
using System.Collections;
using System.Text;

public class HierarchicalNGramP<T>
{
    public int threshold;
    public NGramPredictor<T>[] predictors;
    private int nValue;
}
```

2. Реализуем конструктор для инициализации свойств:

```
public HierarchicalNGramP(int windowSize)
{
    nValue = windowSize + 1;
    predictors = new NGramPredictor<T>[nValue];
    int i;
    for (i = 0; i < nValue; i++)
        predictors[i] = new NGramPredictor<T>(i + 1);
}
```

3. Определим функцию регистрации последовательности, подобную той, что была реализована для предыдущего алгоритма:

```
public void RegisterSequence(T[] actions)
{
    int i;
    for (i = 0; i < nValue; i++)
    {
        T[] subactions = new T[i+1];
        Array.Copy(actions, nValue - i - 1, subactions, 0, i+1);
        predictors[i].RegisterSequence(subactions);
    }
}
```

4. И наконец, реализуем функцию прогнозирования:

```
public T GetMostLikely(T[] actions)
{
    int i;
    T bestAction = default(T);
    for (i = 0; i < nValue; i++)
```

```
        {
            NGramPredictor<T> p;
            p = predictors[nValue - i - 1];
            T[] subactions = new T[i + 1];
            Array.Copy(actions, nValue - i - 1, subactions, 0, i + 1);
            int numActions = p.GetActionsNum(ref actions);
            if (numActions > threshold)
                bestAction = p.GetMostLikely(actions);
        }
        return bestAction;
    }
```

Как это работает...

Иерархический алгоритм N-Gram работает почти так же, как его предшественник, с той лишь разницей, что хранит набор алгоритмов прогнозирования и при расчете использует функции дочерних алгоритмов. Зарегистрированные последовательности – наиболее вероятные будущие действия – разделяются на наборы и подпитывают дочерние алгоритмы.

Использование классификаторов Байеса

Обучение на примерах тяжело дается даже людям. Так, далеко не всегда получается уловить связь между двумя наборами значений. Одним из решений этой проблемы является систематизация одного набора значений для дальнейшей его обработки с помощью классификационных алгоритмов.

Классификаторы Байеса – это алгоритмы прогнозирования, основанные на присвоении меток проблемным экземплярам, с последующим применением к ним вероятностей и теоремы Байеса со строгой независимостью предположений между переменными для анализа. Одним из ключевых преимуществ классификаторов Байеса является их масштабируемость.

Подготовка

Построить общий классификатор совсем непросто, поэтому при его создании будем предполагать, что входные данные могут быть помечены как положительные или отрицательные. Итак, первым делом определим типы меток для обработки классификатором в виде перечисления `NBCLabel`:

```
public enum NBCLabel
{
    POSITIVE,
    NEGATIVE
}
```

Как это реализовать...

Реализацию классификатора можно разбить на пять крупных этапов:

1. Определим класс и его свойства:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class NaiveBayesClassifier : MonoBehaviour
{
    public int numAttributes;
    public int numExamplesPositive;
    public int numExamplesNegative;

    public List<bool> attrCountPositive;
    public List<bool> attrCountNegative;
}
```

2. Определим метод `Awake` для инициализации:

```
void Awake()
{
    attrCountPositive = new List<bool>();
    attrCountNegative = new List<bool>();
}
```

3. Реализуем функцию обновления классификатора:

```
public void UpdateClassifier(bool[] attributes, NBCLabel label)
{
    if (label == NBCLabel.POSITIVE)
    {
        numExamplesPositive++;
        attrCountPositive.AddRange(attributes);
    }
    else
    {
        numExamplesNegative++;
        attrCountNegative.AddRange(attributes);
    }
}
```

4. Определим функцию для упрощенного вычисления вероятности:

```
public float NaiveProbabilities(
        ref bool[] attributes,
        bool[] counts,
        float m,
        float n)
{
    float prior = m / (m + n);
    float p = 1f;
    int i = 0;
    for (i = 0; i < numAttributes; i++)
    {
        p /= m;
        if (attributes[i] == true)
            p *= counts[i].GetHashCode();
        else
            p *= m - counts[i].GetHashCode();
    }
    return prior * p;
}
```

5. И наконец, реализуем функцию прогнозирования:

```
public bool Predict(bool[] attributes)
{
    float nep = numExamplesPositive;
    float nen = numExamplesNegative;
    float x = NaiveProbabilities(ref attributes,
        attrCountPositive.ToArray(), nep, nen);
    float y = NaiveProbabilities(ref attributes,
        attrCountNegative.ToArray(), nen, nep);
    if (x >= y)
        return true;
    return false;
}
```

Как это работает...

Функция `UpdateClassifier` получает набор входных значений и сохраняет их. Она вызывается первой. Функция `NaiveProbabilities` отвечает за вычисление вероятностей, необходимых для функции прогнозирования. И наконец, функция `Predict` вызывается второй и получает результаты классификации.

Использование деревьев принятия решений

Ранее уже не раз упоминалось о мощности и гибкости деревьев в роли компонента принятия решений для игр. Как оказывается, их можно строить динамически, с помощью системы управления обучением. В этом и состоит причина возврата к ним в этой главе.

Существует несколько алгоритмов построения дерева принятия решений, предназначенных для различных целей, таких как прогнозирование или классификация. Здесь будет рассмотрено дерево принятия решений с обучением на примере реализации алгоритма ID3.

Подготовка

Несмотря на то что деревья принятия решений уже создавались в предыдущей главе, и они основывались на тех же принципах, что и осуществляемые здесь, в данной главе для их реализации будут использоваться другие типы данных, учитывающие потребности алгоритма обучения.

Нам понадобятся два типа данных: один – для узлов решений и один – для хранения обучающих примеров.

Тип данных для узлов решений `DecisionNode`:

```
using System.Collections.Generic;

public class DecisionNode
{
    public string testValue;
    public Dictionary<float, DecisionNode> children;

    public DecisionNode(string testValue = "")
    {
        this.testValue = testValue;
        children = new Dictionary<float, DecisionNode>();
    }
}
```

И тип данных `Example`:

```
using UnityEngine;
using System.Collections.Generic;

public enum ID3Action
{
    STOP, WALK, RUN
}
```

```
public class ID3Example : MonoBehaviour
{
    public ID3Action action;
    public Dictionary<string, float> values;

    public float GetValue(string attribute)
    {
        return values[attribute];
    }
}
```

Как это реализовать...

Определим класс ID3 с функциями для конструирования дерева принятия решений.

1. Создадим класс ID3:

```
using UnityEngine;
using System.Collections.Generic;
public class ID3 : MonoBehaviour
{
    // дальнейшая реализация описывается ниже
}
```

2. Начнем реализацию функции, ответственной за разделение атрибутов на наборы:

```
public Dictionary<float, List<ID3Example>> SplitByAttribute(
        ID3Example[] examples,
        string attribute)
{
    Dictionary<float, List<ID3Example>> sets;
    sets = new Dictionary<float, List<ID3Example>>();
    // дальнейшая реализация описывается ниже
}
```

3. Обойдем все полученные примеры и оценим их значение для присваивания набору:

```
foreach (ID3Example e in examples)
{
    float key = e.GetValue(attribute);
    if (!sets.ContainsKey(key))
        sets.Add(key, new List<ID3Example>());
    sets[key].Add(e);
}
return sets;
```

4. Определим функцию вычисления энтропии набора примеров:

```
public float GetEntropy(ID3Example[] examples)
{
    if (examples.Length == 0) return 0f;
    int numExamples = examples.Length;
    Dictionary<ID3Action, int> actionTallies;
    actionTallies = new Dictionary<ID3Action, int>();
    // дальнейшая реализация описывается ниже
}
```

5. Обойдем все примеры и вычислим квоты их действий:

```
foreach (ID3Example e in examples)
{
    if (!actionTallies.ContainsKey(e.action))
        actionTallies.Add(e.action, 0);
    actionTallies[e.action]++;
}
```

6. Вычислим энтропию:

```
int actionCount = actionTallies.Keys.Count;
if (actionCount == 0) return 0f;
float entropy = 0f;
float proportion = 0f;
foreach (int tally in actionTallies.Values)
{
    proportion = tally / (float)numExamples;
    entropy -= proportion * Mathf.Log(proportion, 2);
}
return entropy;
```

7. Реализуем функцию вычисления энтропии для всех наборов примеров. Она очень похожа на предыдущую; в действительности она использует ее:

```
public float GetEntropy(
        Dictionary<float, List<ID3Example>> sets,
        int numExamples)
{
    float entropy = 0f;
    foreach (List<ID3Example> s in sets.Values)
    {
        float proportion;
        proportion = s.Count / (float)numExamples;
```

```
            entropy -= proportion * GetEntropy(s.ToArray());
        }
        return entropy;
    }
```

8. Определим функцию построения дерева принятия решений:

```
    public void MakeTree(
            ID3Example[] examples,
            List<string> attributes,
            DecisionNode node)
    {
        float initEntropy = GetEntropy(examples);
        if (initEntropy <= 0) return;
        // дальнейшая реализация описывается ниже
    }
```

9. Объявим и инициализируем свойства, необходимые для выполнения этого задания:

```
int numExamples = examples.Length;
float bestInfoGain = 0f;
string bestSplitAttribute = "";
float infoGain = 0f;
float overallEntropy = 0f;
Dictionary<float, List<ID3Example>> bestSets;
bestSets = new Dictionary<float, List<ID3Example>>();
Dictionary<float, List<ID3Example>> sets;
```

10. Обойдем все атрибуты, чтобы получить предпочтительный набор, основанный на приросте информации:

```
foreach (string a in attributes)
{
    sets = SplitByAttribute(examples, a);
    overallEntropy = GetEntropy(sets, numExamples);
    infoGain = initEntropy - overallEntropy;
    if (infoGain > bestInfoGain)
    {
        bestInfoGain = infoGain;
        bestSplitAttribute = a;
        bestSets = sets;
    }
}
```

11. Выберем корневой узел, опираясь на предпочтительное разделение атрибутов, и изменим прочие атрибуты для создания остальной части дерева:

```
node.testValue = bestSplitAttribute;
List<string> newAttributes = new List<string>(attributes);
newAttributes.Remove(bestSplitAttribute);
```

12. Обойдем все прочие атрибуты вызовом рекурсивной функции:

```
foreach (List<ID3Example> set in bestSets.Values)
{
    float val = set[0].GetValue(bestSplitAttribute);
    DecisionNode child = new DecisionNode();
    node.children.Add(val, child);
    MakeTree(set.ToArray(), newAttributes, child);
}
```

Как это работает...

Класс является модульным в смысле функциональности. Он не хранит никакой информации, но способен рассчитать и получить все необходимое для функции, конструирующей дерево принятия решений. Метод `SplitByAttribute` принимает примеры и делит их на наборы, необходимые для вычисления их энтропии. Метод `GetEntropy` имеет две перегруженные версии и рассчитывает список примеров и все наборы примеров с помощью формул, определенных в алгоритме ID3. Наконец, метод `MakeTree` рекурсивно строит дерево решений, опираясь на наиболее значимые атрибуты.

Полезные ссылки

- Рецепт *«Выбор с помощью дерева принятия решений»* в *главе 3 «Принятие решений»*.

Использование закрепления рефлекса

Представьте, что требуется создать противника, способного выбирать различные действия во время прохождения игроком уровня (причем модель поведения противника изменяется с течением игры), или игру с различными видами домашних животных, которые должны обладать определенной свободой воли.

Для решения задач такого вида можно использовать целый ряд методов, нацеленных на моделирование обучения, базирующееся на опыте. Одним из них является алгоритм Q-обучения, который и будет реализован в этом рецепте.

Подготовка

Прежде чем углубиться в реализацию основного алгоритма, необходимо создать несколько структур данных: структуру состояния игры, структуру действия игры и класс экземпляра проблемы. Их можно поместить в один файл.

Ниже приводится пример структуры для определения состояния игры:

```
public struct GameState
{
    // Добавьте сюда свое определение состояния игры
}
```

Далее следует пример структуры для определения действий игры:

```
public struct GameAction
{
    // Добавьте сюда свое определение действия
}
```

И наконец, построим тип данных для определения экземпляра проблемы:

1. Создадим файл и класс:

```
public class ReinforcementProblem
{
}
```

2. Определим виртуальную функцию для получения случайного состояния. В зависимости от вида игры может потребоваться получить случайное состояние, учитывающее текущее состояние игры:

```
public virtual GameState GetRandomState()
{
    // Добавьте сюда определение своей модели поведения
    return new GameState();
}
```

3. Определим виртуальную функцию для извлечения всех доступных действий для данного состояния игры:

```
public virtual GameAction[] GetAvailableActions(GameState s)
{
    // Определите здесь свою модель поведения
    return new GameAction[0];
}
```

4. Определим виртуальную функцию выполнения действий с последующим возвратом полученного состояния и выгоды:

```
public virtual GameState TakeAction(
        GameState s,
        GameAction a,
        ref float reward)
{
    // Определите здесь свою модель поведения
    reward = 0f;
    return new GameState();
}
```

Как это реализовать...

Далее мы реализуем два класса. Первый хранит в словаре значения, предназначенные для обучения, а второй – алгоритм Q-обучения:

1. Определим класс `QValueStore`:

```
using UnityEngine;
using System.Collections.Generic;

public class QValueStore : MonoBehaviour
{
    private Dictionary<GameState, Dictionary<GameAction, float>> store;
}
```

2. Реализуем конструктор:

```
public QValueStore()
{
    store = new Dictionary<GameState, Dictionary<GameAction, float>>();
}
```

3. Определим функцию получения значения действия для состояния игры. Здесь следует учесть, что не все действия доступны для конкретного состояния:

```
public virtual float GetQValue(GameState s, GameAction a)
{
    // Добавьте сюда свою модель поведения
    return 0f;
}
```

4. Реализуем функцию получения предпочтительного действия для определенного состояния:

```
public virtual GameAction GetBestAction(GameState s)
{
```

```
    // Добавьте сюда свою модель поведения
    return new GameAction();
}
```

5. Реализуем функцию сохранения Q-значения:

```
public void StoreQValue(
    GameState s, GameAction a, float val)
{
    if (!store.ContainsKey(s))
    {
        Dictionary<GameAction, float> d;
        d = new Dictionary<GameAction, float>();
        store.Add(s, d);
    }
    if (!store[s].ContainsKey(a))
    {
        store[s].Add(a, 0f);
    }
    store[s][a] = val;
}
```

6. Перейдем к разработке класса `QLearning`, запускающего алгоритм:

```
using UnityEngine;
using System.Collections;

public class QLearning : MonoBehaviour
{
    public QValueStore store;
}
```

7. Определим функцию выбора случайного действия из заданного набора:

```
private GameAction GetRandomAction(GameAction[] actions)
{
    int n = actions.Length;
    return actions[Random.Range(0, n)];
}
```

8. Реализуем функцию обучения. Реализация будет производиться в несколько этапов. Начнем с определения. Обратите внимание, что это сопрограмма:

```
public IEnumerator Learn(
        ReinforcementProblem problem,
        int numIterations,
        float alpha,
```

```
        float gamma,
        float rho,
        float nu)
{
    // дальнейшая реализация описывается ниже
}
```

9. Проверим, инициализирован ли список хранилища:

```
if (store == null)
    yield break;
```

10. Получим случайное состояние:

```
GameState state = problem.GetRandomState();
for (int i = 0; i < numIterations; i++)
{
    // дальнейшая реализация описывается ниже
}
```

11. Вернем значение null для продолжения работы с текущим кадром:

```
yield return null;
```

12. Проверим допустимость длины маршрута:

```
if (Random.value < nu)
    state = problem.GetRandomState();
```

13. Получим список доступных действий для текущего состояния игры:

```
GameAction[] actions;
actions = problem.GetAvailableActions(state);
GameAction action;
```

14. Получим действие, зависящее от результата зондирования, проведенного случайным образом:

```
if (Random.value < rho)
    action = GetRandomAction(actions);
else
    action = store.GetBestAction(state);
```

15. Вычислим новое состояние путем применения выбранного действия к текущему состоянию и значение выгоды:

```
float reward = 0f;
GameState newState;
newState = problem.TakeAction(state, action, ref reward);
```

16. Получим значение q с учетом текущего состояния игры и применения действия к вновь рассчитанному состоянию:

```
float q = store.GetQValue(state, action);
GameAction bestAction = store.GetBestAction(newState);
float maxQ = store.GetQValue(newState, bestAction);
```

17. Применим формулу Q-обучения:

```
q = (1f - alpha) * q + alpha * (reward + gamma * maxQ);
```

18. Сохраним вычисленное значение q, использовав его родителей в качестве индексов:

```
store.StoreQValue(state, action, q);
state = newState;
```

Как это работает...

При использовании алгоритма Q-обучения игровой мир интерпретируется как конечный автомат. Обратите внимание на суть следующих параметров:

- alpha: показатель обучения;
- gamma: показатель уценки;
- rho: случайное зондирование;
- nu: длина маршрута.

Обучение с помощью искусственных нейронных сетей

Представьте, что вам требуется создать противника или игровую систему, основанную на имитации работы мозга. Именно так действуют нейронные сети. Они базируются на нейронах, поэтому введем термин **персептрон** для описания групп нейронов. Их входящие и исходящие состояния и составляют нейронную сеть.

В этом рецепте описывается построение нейронной системы, от персептронов до их соединения для образования сети.

Подготовка

Нам потребуется тип данных для обработки ввода InputPerceptron:

```
public class InputPerceptron
{
    public float input;
    public float weight;
}
```

Как это реализовать...

Реализуем два крупных класса. Первый из них представляет персептрон, а второй предназначен для управления нейронной сетью.

1. Реализуем класс `Perceptron`, наследующий класс `InputPerceptron`, который был определен ранее:

```
public class Perceptron : InputPerceptron
{
    public InputPerceptron[] inputList;
    public delegate float Threshold(float x);
    public Threshold threshold;
    public float state;
    public float error;
}
```

2. Реализуем конструктор и настроим количество входных данных:

```
public Perceptron(int inputSize)
{
    inputList = new InputPerceptron[inputSize];
}
```

3. Определим функцию обработки входных данных:

```
public void FeedForward()
{
    float sum = 0f;
    foreach (InputPerceptron i in inputList)
    {
        sum += i.input * i.weight;
    }
    state = threshold(sum);
}
```

4. Реализуем функцию для настройки весов:

```
public void AdjustWeights(float currentError)
{
    int i;
    for (i = 0; i < inputList.Length; i++)
    {
        float deltaWeight;
        deltaWeight = currentError * inputList[i].weight * state;
        inputList[i].weight = deltaWeight;
        error = currentError;
    }
}
```

5. Определим функцию преобразования весов на основе типа входных данных:

```
public float GetIncomingWeight()
{
    foreach (InputPerceptron i in inputList)
    {
        if (i.GetType() == typeof(Perceptron))
            return i.weight;
    }
    return 0f;
}
```

6. Определим класс для обработки набора персептронов, объединенных в сеть:

```
using UnityEngine;
using System.Collections;

public class MLPNetwork : MonoBehaviour
{
    public Perceptron[] inputPer;
    public Perceptron[] hiddenPer;
    public Perceptron[] outputPer;
}
```

7. Реализуем функцию передачи входных данных из одного конца нейронной сети в другой:

```
public void GenerateOutput(Perceptron[] inputs)
{
    int i;
    for (i = 0; i < inputs.Length; i++)
        inputPer[i].state = inputs[i].input;

    for (i = 0; i < hiddenPer.Length; i++)
        hiddenPer[i].FeedForward();

    for (i = 0; i < outputPer.Length; i++)
        outputPer[i].FeedForward();
}
```

8. Определим функцию, которая фактически эмулирует обучение:

```
public void BackProp(Perceptron[] outputs)
{
    // дальнейшая реализация описывается ниже
}
```

9. Обойдем исходящий слой для вычисления значений:

```
int i;
for (i = 0; i < outputPer.Length; i++)
{
    Perceptron p = outputPer[i];
    float state = p.state;
    float error = state * (1f - state);
    error *= outputs[i].state - state;
    p.AdjustWeights(error);
}
```

10. Обойдем внутренние слои Perceptron, исключая слой входных данных:

```
for (i = 0; i < hiddenPer.Length; i++)
{
    Perceptron p = outputPer[i];
    float state = p.state;
    float sum = 0f;
    for (i = 0; i < outputs.Length; i++)
    {
        float incomingW = outputs[i].GetIncomingWeight();
        sum += incomingW * outputs[i].error;
        float error = state * (1f - state) * sum;
        p.AdjustWeights(error);
    }
}
```

11. Реализуем функцию верхнего уровня для упрощения применения:

```
public void Learn(
        Perceptron[] inputs,
        Perceptron[] outputs)
{
    GenerateOutput(inputs);
    BackProp(outputs);
}
```

Как это работает...

Мы реализовали два вида персептронов, связанных между собой: первый вид обрабатывает внешние входные данные, а второй – внутренние. Вот почему *базовый* класс Perceptron наследует последний. Функция FeedForward обрабатывает входные данные и распределяет

их по сети. А для обратной передачи используется функция из числа тех, что отвечают за корректировки весов. Эта *корректировка весов* и эмулирует обучение.

Создание непредсказуемых частиц с помощью алгоритма поиска гармонии

Поскольку я музыкант, этот рецепт особенно близок моему сердцу. Представьте группу музыкантов, готовящихся сыграть мелодию. Они никогда не играли вместе, и, поскольку мелодия им не знакома, им требуется настроить свои инструменты и выработать общий стиль игры. Имитация такой адаптации осуществляется с помощью алгоритма поиска гармонии.

Подготовка

Нам необходимо определить целевую функцию-делегата и назначить ее перед вызовом.

Как это реализовать...

Реализуем алгоритм с помощью класса:

1. Определим класс `HarmonySearch`:

```
using UnityEngine;
using System.Collections;

public class HarmonySearch : MonoBehaviour
{

}
```

2. Определим общедоступные входные данные, которые требуется настроить:

```
[Range(1, 100)]
public int memorySize = 30;
public int pitchNum;
// оценка консолидации
[Range(0.1f, 0.99f)]
public float consRate = 0.9f;
// оценка коррекции
[Range(0.1f, 0.99f)]
public float adjsRate = 0.7f;
public float range = 0.05f;
```

```
public int numIterations;
[Range(0.1f, 1.0f)]
public float par = 0.3f;
```

3. Определим список границ. Границы имеют тип данных Vector2, поле x которого соответствует нижнему пределу, а y – верхнему. Количество границ должно быть равно числу тонов:

```
public Vector2[] bounds;
```

4. Определим закрытые свойства для алгоритма:

```
private float[,] memory;
private float[] solution;
private float fitness;
private float best;
```

5. Реализуем функцию инициализации:

```
private void Init()
{
    memory = new float[memorySize, pitchNum];
    solution = new float[memorySize];
    fitness = ObjectiveFunction(memory);
}
```

6. Начнем определение функции создания гармонии:

```
private float[] CreateHarmony()
{
    float[] vector = new float[pitchNum];
    int i;
    // дальнейшая реализация описывается ниже
}
```

7. Обойдем все тона (инструменты):

```
for (i = 0; i < pitchNum; i++)
{
    // дальнейшая реализация описывается ниже
}
```

8. Вычислим новое количество возможных гармоний, учитывая случайное значение:

```
if (Random.value < consRate)
{
    int r = Random.Range(0, memory.Length);
    float val = memory[r, i];
```

```
        if (Random.value < adjsRate)
            val = val + range * Random.Range(-1f, 1f);
        if (val < bounds[i].x)
            val = bounds[i].x;
        if (val > bounds[i].y)
            val = bounds[i].y;
        vector[i] = val;
    }
```

9. Определим значение на случай, если оно должно быть случайным:

```
    else
    {
        vector[i] = Random.Range(bounds[i].x, bounds[i].y);
    }
```

10. Получим новый вектор:

```
    return vector;
```

11. Определим функцию, которая проделает всю работу:

```
    public float[] Run()
    {
        // дальнейшая реализация описывается ниже
    }
```

12. Инициализируем значения:

```
    Init();
    int iterations = numIterations;
    float best = Mathf.Infinity;
```

13. Вызовем приведенные выше функции и произведем вычисления для получения списка наилучших тонов:

```
    while (iterations != 0)
    {
        iterations--;
        float[] harm = CreateHarmony();
        fitness = ObjectiveFunction(harm);
        best = Mathf.Min(fitness, best);
        memory = harm;
    }
```

14. Вернем список наилучших тонов:

```
    return
```

Как это работает...

Алгоритм инициализирует все значения с учетом общедоступных входных данных и внутренних свойств. Он многократно выполняет цикл для получения списка наилучших тонов среди множества границ и разных тонов, созданных с помощью определенной ранее целевой функции.

Глава 8

Прочее

В этой главе рассматриваются самые разные механизмы:

- улучшенная обработка случайных чисел;
- соперник для игры в воздушный хоккей;
- соперник для настольного футбола;
- программное создание лабиринтов;
- реализация автопилота для автомобиля;
- управление гонками с преградами в виде системы адаптивных ограничителей.

Введение

В этой последней главе рассматривается применение новых и старых методов и алгоритмов, описанных в предыдущих главах, для создания новых моделей поведения, которые не укладываются ни в одну из перечисленных выше категорий. Цель этой главы – помочь вам увлекательно провести время и по-новому взглянуть на сочетание разных методов для достижения различных целей.

Улучшенная обработка случайных чисел

Иногда требуется создать модель случайного поведения, но в то же время это поведение должно не слишком сильно отклоняться от эталонного, например это касается моделирования стрельбы в цель. При применении модели нормализованного случайного поведения пули будут отклоняться на одинаковые расстояния, зависящие от расстояния до мишени, вдоль осей x и y. Однако было бы желательно, чтобы большая часть пуль ложилась ближе к цели, поскольку это выглядит естественнее.

Большинство функций, генераторов случайных чисел,возвращает нормализованные значения в заданном диапазоне, и это соответствует желаемым результатам. Но такое поведение, как только что было упомянуто, не всегда хорошо подходит для некоторых игр. Поэтому попробуем реализовать свои генераторы случайных чисел с нормальным распределением вместо нормализованного для использования в играх.

Подготовка

Важно понимать различие между равномерным и нормальным распределениями. На рис. 8.1 приводится сравнительное графическое представление требуемой модели с нормальным и равномерным распределениями.

На левом рисунке демонстрируется равномерное распределение по всему кругу, которое обычно применяется в качестве случайного распределения. Тем не менее при разработке некоторых механизмов, например в случае прицельной стрельбы, более естественно будет выглядеть случайное распределение, показанное на рис. 8.1 справа.

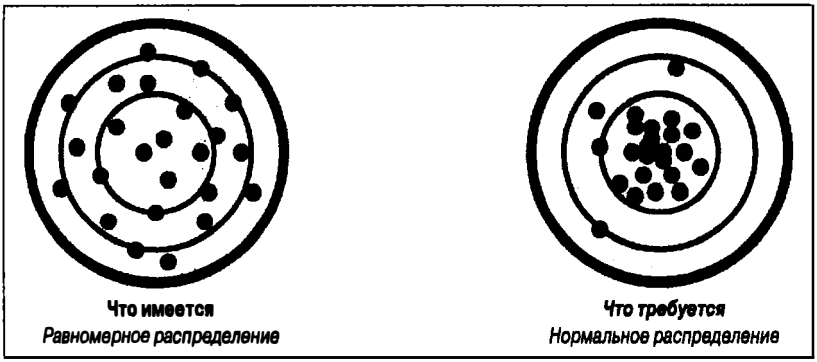


Рис. 8.1 ❖ Равномерное (слева) и нормальное (справа) распределения

Как это реализовать...

Определим простой класс.

1. Класс `RandomGaussian`:

```
using UnityEngine;

public class RandomGaussian
```

```
{
    // дальнейшая реализация описывается ниже
}
```

2. Определим метод RangeAdditive для инициализации свойств:

```
public static float RangeAdditive(params Vector2[] values)
{
    float sum = 0f;
    int i;
    float min, max;
    // дальнейшая реализация описывается ниже
}
```

3. Проверим количество параметров. Если оно равно нулю, создадим три новых значения:

```
if (values.Length == 0)
{
    values = new Vector2[3];
    for (i = 0; i < values.Length; i++)
        values[i] = new Vector2(0f, 1f);
}
```

4. Просуммируем:

```
for (i = 0; i < values.Length; i++)
{
    min = values[i].x;
    max = values[i].y;
    sum += Random.Range(min, max);
}
```

5. Вернем полученное случайное число:

```
return sum;
```

Дополнительная информация...

Всегда следует стремиться к эффективности. Именно по этой причине существует еще один способ для получения того же результата. Чтобы воспользоваться им, необходимо реализовать новый метод, основанный на решении, предлагаемом Рабином (Rabin) и другими авторами (подробности см. в разделе *«Полезные ссылки»*):

```
public static ulong seed = 61829450; public static float Range()
{
    double sum = 0;
    for (int i = 0; i < 3; i++)
```

```
    {
        ulong holdseed = seed;
        seed ^= seed << 13;
        seed ^= seed >> 17;
        seed ^= seed << 5;
        long r = (long)(holdseed * seed);
        sum += r * (1.0 / 0x7FFFFFFFFFFFFFFF);
    }
    return (float)sum;
}
```

Полезные ссылки

- Более подробную информацию о теоретических основах генератора случайных чисел Гаусса и других можно найти в статье с номером 3 в книге Стива Рабина (Steve Rabin) «*Game AI Pro*».

Соперник для игры в воздушный хоккей

Воздушный хоккей – одна из самых популярных игр, имевшихся в золотой век аркад, и все еще широко распространена сегодня. С появлением мобильных устройств с сенсорными экранами воздушный хоккей стал не только увлекательным способом проверки физических движков, но и игрой с интеллектуальным соперником, несмотря на ее явно низкую сложность.

Подготовка

Применяемый здесь механизм основывается на нескольких алгоритмах, рассмотренных в *главе 1 «Интеллектуальные модели поведения: перемещение»*, таких как `Seek`, `Arrive` и `Leave`, а также на отбрасывании луча в некоторых других рецептах, таких как сглаживание пути.

Для применения этого алгоритма необходим игровой объект лопатки, используемый агентом с присоединенными к нему компонентами `AgentBehaviour`, `Seek` и `Leave`. Также следует отметить тегами объекты, используемые как стены, то есть содержащие коллайдеры в виде короба, как показано на рис. 8.2.

Наконец, нужно создать перечисление, определяющее состояния соперника:

```
public enum AHRState
{
```

```
    ATTACK,
    DEFEND,
    IDLE
}
```

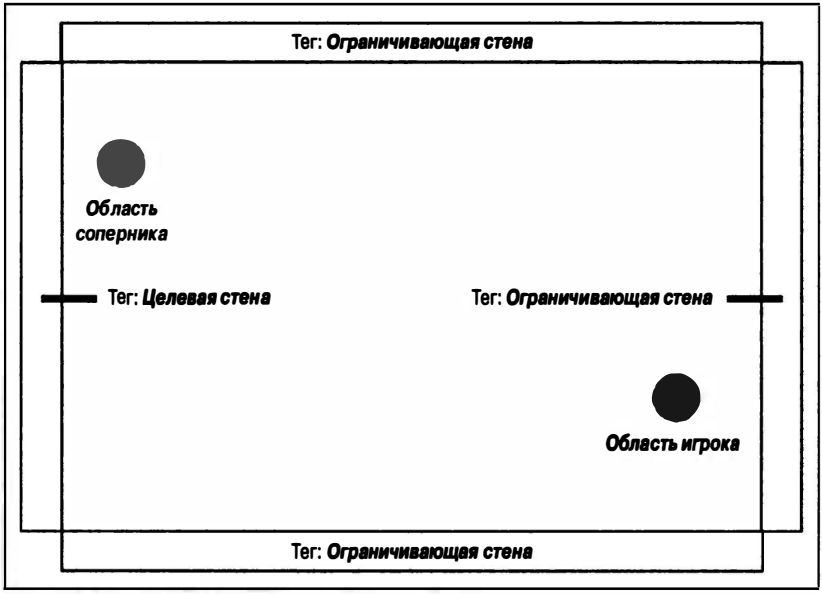


Рис. 8.2 ❖ Воздушный хоккей

Как это реализовать...

Класс получился объемным, поэтому будьте внимательны при выполнении описываемых далее шагов.

1. Определим класс соперника:

```
using UnityEngine;
using System.Collections;

public class AirHockeyRival : MonoBehaviour
{
    // дальнейшая реализация описывается ниже
}
```

2. Объявим общедоступные переменные и инициализируем их:

```
public GameObject puck;
public GameObject paddle;
```

```
public string goalWallTag = "GoalWall";
public string sideWallTag = "SideWall";
[Range(1, 10)]
public int maxHits;
```

3. Объявим внутренние переменные:

```
float puckWidth;
Renderer puckMesh;
Rigidbody puckBody;
AgentBehaviour agent;
Seek seek;
Leave leave;
AHRState state;
bool hasAttacked;
```

4. Реализуем метод Awake для создания внутренних классов с учетом общедоступных:

```
public void Awake()
{
    puckMesh = puck.GetComponent<Renderer>();
    puckBody = puck.GetComponent<Rigidbody>();
    agent = paddle.GetComponent<AgentBehaviour>();
    seek = paddle.GetComponent<Seek>();
    leave = paddle.GetComponent<Leave>();
    puckWidth = puckMesh.bounds.extents.z;
    state = AHRState.IDLE;
    hasAttacked = false;
    if (seek.target == null)
        seek.target = new GameObject();
    if (leave.target == null)
        leave.target = new GameObject();
}
```

5. Объявим метод Update. Его тело будет определено ниже:

```
public void Update()
{
    // дальнейшая реализация описывается ниже
}
```

6. Проверим текущее состояние и вызовем нужные функции:

```
switch (state)
{
    case AHRState.ATTACK:
        Attack();
```

```
            break; default:
        case AHRState.IDLE:
            agent.enabled = false;
            break;
        case AHRState.DEFEND:
            Defend();
            break;
    }
```

7. Вызовем функцию сброса активного состояния броска шайбы:

```
AttackReset();
```

8. Реализуем функцию установки состояния из внешних объектов:

```
public void SetState(AHRState newState)
{
    state = newState;
}
```

9. Реализуем функцию получения расстояния от лопатки до шайбы:

```
private float DistanceToPuck()
{
    Vector3 puckPos = puck.transform.position;
    Vector3 paddlePos = paddle.transform.position;
    return Vector3.Distance(puckPos, paddlePos);
}
```

10. Объявим метод броска. Он определен ниже:

```
private void Attack()
{
    if (hasAttacked)
        return;
    // дальнейшая реализация описывается ниже
}
```

11. Активируем компонент агента и рассчитаем расстояние до шайбы:

```
agent.enabled = true;
float dist = DistanceToPuck();
```

12. Проверим расстояние до шайбы. Если ее можно достать, сделаем это:

```
if (dist > leave.dangerRadius)
{
    Vector3 newPos = puck.transform.position;
    newPos.z = paddle.transform.position.z;
    seek.target.transform.position = newPos;
```

```
    seek.enabled = true;
    return;
}
```

13. Произведем бросок шайбы, если она в пределах досягаемости:

```
hasAttacked = true;
seek.enabled = false;
Vector3 paddlePos = paddle.transform.position;
Vector3 puckPos = puck.transform.position;
Vector3 runPos = paddlePos - puckPos;
runPos = runPos.normalized * 0.1f;
runPos += paddle.transform.position;
leave.target.transform.position = runPos;
leave.enabled = true;
```

14. Реализуем функцию сброса параметров броска шайбы:

```
private void AttackReset()
{
    float dist = DistanceToPuck();
    if (hasAttacked && dist < leave.dangerRadius)
        return;
    hasAttacked = false;
    leave.enabled = false;
}
```

15. Определим функцию защиты цели:

```
private void Defend()
{
    agent.enabled = true;
    seek.enabled = true;
    leave.enabled = false;
    Vector3 puckPos = puckBody.position;
    Vector3 puckVel = puckBody.velocity;
    Vector3 targetPos = Predict(puckPos, puckVel, 0);
    seek.target.transform.position = targetPos;
}
```

16. Реализуем функцию предугадывания будущей позиции шайбы:

```
private Vector3 Predict(Vector3 position, Vector3 velocity, int numHit)
{
    if (numHit == maxHits)
        return position;
    // дальнейшая реализация описывается ниже
}
```

17. Отбросим луч с учетом положения и направления полета шайбы:

```
RaycastHit[] hits = Physics.RaycastAll(position, velocity. normalized);
RaycastHit hit;
```

18. Проверим результат броска:

```
foreach (RaycastHit h in hits)
{
    string tag = h.collider.tag;
    // дальнейшая реализация описывается ниже
}
```

19. Проверим столкновение с бортиком-воротами. Базовый вариант:

```
if (tag.Equals(goalWallTag))
{
    position = h.point;
    position += (h.normal * puckWidth);
    return position;
}
```

20. Проверим столкновение с боковым бортиком. Рекурсивный вариант:

```
if (tag.Equals(sideWallTag))
{
    hit = h;
    position = hit.point + (hit.normal * puckWidth);
    Vector3 u = hit.normal;
    u *= Vector3.Dot(velocity, hit.normal);
    Vector3 w = velocity - u;
    velocity = w - u;
    break;
}
// конец цикла foreach
```

21. Добавим рекурсивный случай. Это делается из цикла foreach:

```
return Predict(position, velocity, numHit + 1);
```

Как это работает...

Агент рассчитывает следующий бросок шайбы с учетом ее текущей скорости до того момента, когда по результатам расчетов шайба столкнется с бортиком агента. Этот расчет позволяет агенту выставить ло-

патку навстречу шайбе. Кроме того, в момент, когда шайба находится вблизи лопатки и движется в ее направлении, выполняется переход в режим атаки. В противном случае режим изменяется на режим простоя или защиты в зависимости от расстояния.

Полезные ссылки

- Рецепты *«Преследование и уклонение»* и *«Достижение цели и уход от погони»* в *главе 1 «Интеллектуальные модели поведения: перемещение»*.

Соперник для настольного футбола

Другой популярной настольной игрой, которая нашла свой путь в цифровой мир, является настольный футбол. В этом рецепте создается соперник, имитирующий действия человека, и используются некоторые методы моделирования органов чувств человека с учетом их ограничений.

Подготовка

В этом рецепте используются знания, полученные в *главе 5 «Органы чувств агентов»*, для имитации органов зрения.

Сначала определим пару перечислений:

```
public enum TFRAxisCompare
{
    X, Y, Z
}

public enum TFRState
{
    ATTACK, DEFEND, OPEN
}
```

Как это реализовать...

Этот весьма обширный рецепт. Здесь мы определим пару классов: первый имитирует стержень настольного футбола, а второй – агента с искусственным интеллектом, который управляет стержнями:

1. Определим класс стержня, которым управляет искусственный интеллект:

```
using UnityEngine;
using System.Collections;

public class TFRBar : MonoBehaviour
{
    [HideInInspector]
    public int barId;
    public float barSpeed;
    public float attackDegrees = 30f;
    public float defendDegrees = 0f;
    public float openDegrees = 90f;
    public GameObject ball;
    private Coroutine crTransition;
    private bool isLocked;
    // дальнейшая реализация описывается ниже
}
```

2. Реализуем функцию Awake:

```
void Awake()
{
    crTransition = null;
    isLocked = false;
}
```

3. Определим функцию управления состоянием стержня:

```
public void SetState(TFRState state, float speed = 0f)
{
    // дальнейшая реализация описывается ниже
}
```

4. Проверим, заблокирован ли он (начато движение). Это является необязательным:

```
// необязательно
if (isLocked)
    return;
isLocked = true;
```

5. Проверим скорость:

```
if (speed == 0)
    speed = barSpeed;
float degrees = 0f;
```

6. Проверим состояние и примем решение:

```
switch(state)
{
```

```
        case TFRState.ATTACK:
            degrees = attackDegrees;
            break;
        default:
        case TFRState.DEFEND:
            degrees = defendDegrees;
            break;
        case TFRState.OPEN:
            degrees = openDegrees;
            break;
    }
```

7. Выполним переход:

```
if (crTransition != null)
    StopCoroutine(crTransition);
crTransition = StartCoroutine(Rotate(degrees, speed));
```

8. Определим функцию вращения стержня:

```
public IEnumerator Rotate(float target, float speed)
{
    // дальнейшая реализация описывается ниже
}
```

9. Тело функции, осуществляющей вращение:

```
while (transform.rotation.x != target)
{
    Quaternion rot = transform.rotation;
    if (Mathf.Approximately(rot.x, target))
    {
        rot.x = target;
        transform.rotation = rot;
    }
    float vel = target - rot.x;
    rot.x += speed * Time.deltaTime * vel;
    yield return null;
}
```

10. Вернем стержень в положение по умолчанию:

```
isLocked = false;
transform.rotation = Quaternion.identity;
```

11. Реализуем функцию перемещения стержня вбок:

```
public void Slide(float target, float speed)
{
```

```
    Vector3 targetPos = transform.position;
    targetPos.x = target;
    Vector3 trans = transform.position - targetPos;
    trans *= speed * Time.deltaTime;
    transform.Translate(trans, Space.World);
}
```

12. Определим класс основного механизма искусственного интеллекта:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class TFRival : MonoBehaviour
{

    public string tagPiece = "TFPiece";
    public string tagWall = "TFWall";
    public int numBarsToHandle = 2;
    public float handleSpeed;
    public float attackDistance;
    public TFRAxisCompare depthAxis = TFRAxisCompare.Z;
    public TFRAxisCompare widthAxis = TFRAxisCompare.X;
    public GameObject ball;
    public GameObject[] bars;
    List<GameObject>[] pieceList;
}
```

13. Инициализируем список участков в функции Awake:

```
void Awake()
{
    int numBars = bars.Length;
    pieceList = new List<GameObject>[numBars];
    for (int i = 0; i < numBars; i++)
    {
        pieceList[i] = new List<GameObject>();
    }
}
```

14. Начнем реализацию функции Update:

```
void Update()
{
    int[] currBars = GetNearestBars();
    Vector3 ballPos = ball.transform.position;
```

```
    Vector3 barsPos;
    int i;
    // дальнейшая реализация описывается ниже
}
```

15. Определим состояние каждой лопатки в зависимости от положения мяча:

```
for (i = 0; i < currBars.Length; i++)
{
    GameObject barObj = bars[currBars[i]];
    TFRBar bar = barObj.GetComponent<TFRBar>();
    barsPos = barObj.transform.position;
    float ballVisible = Vector3.Dot(barsPos, ballPos);
    float dist = Vector3.Distance(barsPos, ballPos);
    if (ballVisible > 0f && dist <= attackDistance)
        bar.SetState(TFRState.ATTACK, handleSpeed);
    else if (ballVisible > 0f)
        bar.SetState(TFRState.DEFEND);
    else
        bar.SetState(TFRState.OPEN);
}
```

16. Реализуем функцию `OnGUI`. Она выполняет прогнозирование с частотой 30 кадров в секунду:

```
public void OnGUI()
{
    Predict();
}
```

17. Определим функцию прогнозирования:

```
private void Predict()
{
    Rigidbody rb = ball.GetComponent<Rigidbody>();
    Vector3 position = rb.position;
    Vector3 velocity = rb.velocity.normalized;
    int[] barsToCheck = GetNearestBars();
    List<GameObject> barsChecked;
    GameObject piece;
    barsChecked = new List<GameObject>();
    int id = -1;
    // дальнейшая реализация описывается ниже
}
```

18. Определим основной цикл проверки траектории мяча:

```
do
{
    RaycastHit[] hits = Physics.RaycastAll(position,
                                           velocity. normalized);
    RaycastHit wallHit = null;
    foreach (RaycastHit h in hits)
    {
        // дальнейшая реализация описывается ниже
    }
} while (barsChecked.Count == numBarsToHandle);
```

19. Получим объект, участвующий в столкновении, и проверим, является ли он лопаткой и не проверялся ли ранее:

```
GameObject obj = h.collider.gameObject;
if (obj.CompareTag(tagWall))
    wallHit = h;
if (!IsBar(obj))
    continue;
if (barsChecked.Contains(obj))
    continue;
```

20. Проверим, является ли лопатка ближайшей к мячу:

```
bool isToCheck = false;
for (int i = 0; i < barsToCheck.Length; i++)
{
    id = barsToCheck[i];
    GameObject barObj = bars[id];
    if (obj == barObj)
    {
        isToCheck = true;
        break;
    }
}
if (!isToCheck)
    continue;
```

21. Получим точку столкновения с лопаткой и вычислим движение для блокировки мяча в ближайшем углублении:

```
Vector3 p = h.point;
piece = GetNearestPiece(h.point, id);
Vector3 piecePos = piece.transform.position;
```

```
float diff = Vector3.Distance(h.point, piecePos);
obj.GetComponent<TFRBar>().Slide(diff, handleSpeed);
barsChecked.Add(obj);
```

22. В противном случае выполним повторный расчет с учетом точки удара о стену:

```
C
```

23. Определим функцию расстановки углублений для соответствующей лопатки:

```
void SetPieces()
{
    // дальнейшая реализация описывается ниже
}
```

24. Создадим словарь для сравнения глубины углублений:

```
// Создание словаря, связывающего z-индекс и лопатку
Dictionary<float, int> zBarDict;
zBarDict = new Dictionary<float, int>();
int i;
```

25. Настройка словаря:

```
for (i = 0; i < bars.Length; i++)
{
    Vector3 p = bars[i].transform.position;
    float index = GetVectorAxis(p, this.depthAxis);
    zBarDict.Add(index, i);
}
```

26. Начнем сопоставление углублений с лопатками:

```
// Сопоставление участков с лопатками
GameObject[] objs = GameObject.FindGameObjectsWithTag(tagPiece);
Dictionary<float, List<GameObject>> dict;
dict = new Dictionary<float, List<GameObject>>();
```

27. Назначим углубления соответствующим элементам словаря:

```
foreach (GameObject p in objs)
{
    float zIndex = p.transform.position.z;
    if (!dict.ContainsKey(zIndex))
        dict.Add(zIndex, new List<GameObject>());
    dict[zIndex].Add(p);
}
```

28. Определим функцию получения индекса лопатки по заданной позиции:

```
int GetBarIndex(Vector3 position, TFRAxisCompare axis)
{
    // дальнейшая реализация описывается ниже
}
```

29. Проверим допустимость:

```
int index = 0;
if (bars.Length == 0)
    return index;
```

30. Объявим необходимые значения:

```
float pos = GetVectorAxis(position, axis);
float min = Mathf.Infinity;
float barPos; Vector3 p;
```

31. Обойдем список лопаток:

```
for (int i = 0; i < bars.Length; i++)
{
    p = bars[i].transform.position;
    barPos = GetVectorAxis(p, axis);
    float diff = Mathf.Abs(pos - barPos);
    if (diff < min)
    {
        min = diff;
        index = i;
    }
}
```

32. Вернем найденный индекс:

```
return index;
```

33. Реализуем функцию вычисления осей вектора:

```
float GetVectorAxis(Vector3 v, TFRAxisCompare a)
{
    if (a == TFRAxisCompare.X)
        return v.x;
    if (a == TFRAxisCompare.Y)
        return v.y;
    return v.z;
}
```

34. Определим функцию получения лопатки, ближайшей к мячу:

```
public int[] GetNearestBars()
{
    // дальнейшая реализация описывается ниже
}
```

35. Инициализируем необходимые переменные:

```
int numBars = Mathf.Clamp(numBarsToHandle, 0, bars.Length);
Dictionary<float, int> distBar;
distBar = new Dictionary<float, int>(bars.Length);
List<float> distances = new List<float>(bars.Length);
int i;
Vector3 ballPos = ball.transform.position;
Vector3 barPos;
```

36. Обойдем лопатки:

```
for (i = 0; i < bars.Length; i++)
{
    barPos = bars[i].transform.position;
    float d = Vector3.Distance(ballPos, barPos);
    distBar.Add(d, i);
    distances.Add(d);
}
```

37. Отсортируем расстояния:

```
distances.Sort();
```

38. Получим расстояния и воспользуемся словарем наоборот:

```
int[] barsNear = new int[numBars]; for (i = 0; i < numBars; i++)
{
    float d = distances[i];
    int id = distBar[d];
    barsNear[i] = id;
}
```

39. Вернем идентификаторы лопаток:

```
return barsNear;
```

40. Реализуем функцию, проверяющую, является ли заданный объект лопаткой:

```
private bool IsBar(GameObject gobj)
{
```

```
        foreach (GameObject b in bars)
        {
            if (b == gobj)
                return true;
        }
        return false;
    }
```

41. Начнем реализацию функции определения ближайшего к лопатке участка по заданной позиции:

```
private GameObject GetNearestPiece(Vector3 position, int barId)
{
    // дальнейшая реализация описывается ниже
}
```

42. Определим необходимые переменные:

```
float minDist = Mathf.Infinity;
float dist;
GameObject piece = null;
```

43. Обойдем список углублений и определим ближайшее:

```
foreach (GameObject p in pieceList[barId])
{
    dist = Vector3.Distance(position, p.transform.position);
    if (dist < minDist)
    {
        minDist = dist;
        piece = p;
    }
}
```

44. Вернем участок:

```
return piece;
```

Как это работает...

Реализация соперника для игры в настольный футбол основана на навыках, приобретенных при реализации соперника для игры в воздушный хоккей. К ним относятся: отбрасывание луча для определения траектории мяча и изменение положения лопатки, ближайшей к участку. Кроме того, перемещение лопатки зависит от того, находится ли соперник в режиме нападения или защиты, так как он может заблокировать мяч или пропустить его дальше.

Полезные ссылки

- Рецепт *«Имитация зрения с использованием коллайдера»* в *главе 5 «Органы чувств агентов»*.

Программное создание лабиринтов

Это совершенно новый рецепт, предназначенный для программного создания карт и уровней. Он обеспечивает полностью программное создание лабиринта. Кроме того, здесь рассматривается смешанный подход, позволяющий использовать дизайн уровня вместе с программной генерацией контента.

Подготовка

Для реализации этого рецепта необходимо разбираться в концепциях разбиения пространства на две части и понимать, как работает алгоритм поиска в ширину, описанный в *главе 2 «Навигация»*.

Как это реализовать...

Здесь мы определим два класса: один – для разделяемых узлов, а другой – для хранения всех узлов и представления лабиринта.

1. Определим класс BSPNode и его свойства:

```
using UnityEngine;

[System.Serializable]
public class BSPNode
{
    public Rect rect;
    public BSPNode nodeA;
    public BSPNode nodeB;
}
```

2. Реализуем конструктор класса:

```
public BSPNode(Rect rect)
{
    this.rect = rect;
    nodeA = null;
    nodeB = null;
}
```

3. Определим функцию разделения узлов на две подгруппы:

```
public void Split(float stopArea)
{
```

```
    // дальнейшая реализация описывается ниже
}
```

4. Проверим базовый случай:

```
if (rect.width * rect.height >= stopArea)
    return;
```

5. Инициализируем необходимые переменные:

```
bool vertSplit = Random.Range(0, 1) == 1;
float x, y, w, h;
Rect rectA, rectB;
```

6. Вычислим разделение по горизонтали:

```
if (!vertSplit)
{
    x = rect.x;
    y = rect.y;
    w = rect.width;
    h = rect.height / 2f;
    rectA = new Rect(x, y, w, h);
    y += h;
    rectB = new Rect(x, y, w, h);
}
```

7. Вычислим разделение по вертикали:

```
else
{
    x = rect.x;
    y = rect.y;
    w = rect.width / 2f;
    h = rect.height;
    rectA = new Rect(x, y, w, h);
    x += w;
    rectB = new Rect(x, y, w, h);
}
```

8. Определим класс для обработки подземелья и объявим его свойства:

```
using UnityEngine;
using System.Collections.Generic;

public class Dungeon : MonoBehaviour
{
    public Vector2 dungeonSize;
```

```
    public float roomAreaToStop;
    public float middleThreshold;
    public GameObject floorPrefab;
    private BSPNode root;
    private List<BSPNode> nodeList;
}
```

9. Реализуем функцию разделения:

```
public void Split()
{
    float x, y, w, h;
    x = dungeonSize.x / 2f * -1f;
    y = dungeonSize.y / 2f * -1f;
    w = dungeonSize.x;
    h = dungeonSize.y;
    Rect rootRect = new Rect(x, y, w, h);
    root = new BSPNode(rootRect);
}
```

10. Реализуем функцию рисования лабиринта из его узлов:

```
public void DrawNode(BSPNode n)
{
    GameObject go = Instantiate(floorPrefab) as GameObject;
    Vector3 position = new Vector3(n.rect.x, 0f, n.rect.y);
    Vector3 scale = new Vector3(n.rect.width, 1f, n.rect.height);
    go.transform.position = position;
    go.transform.localScale = scale;
}
```

Как это работает...

Лабиринт состоит из двух крупных структур данных. Одна относится к логике обработки с помощью алгоритма поиска в ширину, а другая – к визуальному отображению лабиринта. Идея, лежащая в основе этого представления, заключается в многократном делении пространства пополам, пока выполняется условие. Это называется двоичным разбиением пространства (Binary Space Partitioning).

Затем создаются помещения для полученных узлов, и, наконец, области соединяются в дерево, снизу вверх (от листьев к корню).

Дополнительная информация...

- Существует другой, более простой метод, но он требует привлечения команды художников или дизайнеров уровня, которые зара-

нее создают узлы и определяют их местоположение, а игра с применением алгоритма поиска в ширину соединяет их в случайном порядке.

- Участки можно вращать.
- Его можно улучшить с помощью описанных ранее генераторов случайных чисел и настройки размещения участков в списке.

Полезные ссылки

- Рецепт *«Поиск кратчайшего пути в сети с помощью алгоритма BFS»* из *главы 2 «Навигация»*.

Реализация автопилота для автомобиля

Разве может доставить удовольствие игра в автогонки без соперников? Их реализация является одной из сложнейших задач применения искусственного интеллекта в играх. Она обычно решается путем создания *хитрых* агентов, обходящих определенные ограничения, связанные с физической моделью поведения, в обязательном порядке налагаемые на игрока, поскольку эти ограничения могут приводить к неустойчивому или неточному поведению при вычислении средствами искусственного интеллекта. Здесь же будет использован органичный подход, использующий методы из предыдущей главы.

Подготовка

В этой главе рассматриваются способы создания автономного автомобиля, основанные на продвинутых методах из *главы 1 «Интеллектуальные модели поведения: перемещение»*, позволяющих следовать по маршруту, уклоняясь от стен. Поэтому для понимания реализации вы должны иметь полное представление, как они работают.

Как это реализовать...

1. Создадим пустой игровой объект **GameObject**.
2. Присоединим к нему компонент агента **Agent**.
3. Присоединим компонент следования по маршруту **FollowPath**.
4. Присоединим компонент уклонения от стен **WallAvoid**.
5. Создадим с помощью компонента **PathNode** трассу, состоящую из отдельных участков.
6. Отметим границы трассы как стены.
7. Убедимся, что трасса имеет завершенный вид.

Как это работает...

Используя системы из предыдущих глав, достаточно легко создать простую, но гибкую систему реализации интеллектуальных автомобилей.

Полезные ссылки

- Рецепты *«Следование по маршруту»* и *«Уклонение от встреч со стенами»* из *главы 1 «Интеллектуальные модели поведения: перемещение»*.

Управление гонками с адаптивными ограничениями

Обычно мы стремимся создать игровую среду, приспосабливающуюся к игроку, и гонки как нельзя лучше подходят ля этого, учитывая определенные расхождения в ее восприятии «хитрым» агентом.

В данном случае в качестве внутреннего механизма используется фреймворк, позволяющий подключать собственную эвристику для управления скоростью транспортного средства с учетом его статуса. Причем не важно, будет он применяться в аркадной гоночной игре или в игре-симуляторе, – фреймворк подойдет для обоих случаев.

Подготовка

Для реализации данного рецепта необходимы основные навыки, которые даются в *главе 1 «Интеллектуальные модели поведения: перемещение»*, чтобы иметь возможность разрабатывать стратегию расширения фреймворка под собственные нужды, то есть разбираться в особенностях работы класса агента и модели выбора направления движения. В общем смысле, речь пойдет о векторных операциях.

Как это реализовать...

Для реализации искусственного интеллекта на высоком и низком уровнях потребуются три класса:

1. Определим класс для простого агента соперника:

```
using UnityEngine;

public class RacingRival : MonoBehaviour
{
    public float distanceThreshold;
```

```
    public float maxSpeed;
    public Vector3 randomPos;
    protected Vector3 targetPosition;
    protected float currentSpeed;
    protected RacingCenter ghost;
}
```

2. Определим функцию Start:

```
void Start()
{
    ghost = FindObjectOfType<RacingCenter>();
}
```

3. Определим функцию Update для обработки целевой позиции движения:

```
public virtual void Update()
{
    targetPosition = transform.position + randomPos;
    AdjustSpeed(targetPosition);
}
```

4. Определим функцию регулировки скорости:

```
public virtual void AdjustSpeed(Vector3 targetPosition)
{
    // реализуйте здесь свою модель поведения
}
```

5. Определим класс гонщика-призрака, или непобедимого гонщика:

```
using UnityEngine;
public class RacingCenter : RacingRival
{
    public GameObject player;
}
```

6. Реализуем функцию поиска цели:

```
void Start()
{
    player = GameObject.FindGameObjectWithTag(;Player;);
}
```

7. Переопределим функцию Update, поскольку непобедимый автомобиль должен приспосабливаться к поведению игрока:

```
public override void Update()
{
    Vector3 playerPos = player.transform.position;
    float dist = Vector3.Distance(transform.position, playerPos);
    if (dist > distanceThreshold)
    {
        targetPosition = player.transform.position;
        base.Update();
    }
}
```

8. Реализуем его особую модель поведения:

```
public override void AdjustSpeed(Vector3 targetPosition)
{

    // При использовании в случае базовой модели
    // поведения также применяется регулировка скорости
    base.AdjustSpeed(targetPosition);
}
```

9. Определим класс искусственного интеллекта на верхнем уровне:

```
using UnityEngine;

public class Rubberband : MonoBehaviour
{
    RacingCenter ghost;
    RacingRival[] rivals;
}
```

10. Назначим каждому гонщику случайную позицию в системе адаптивных ограничений. В нашем случае используется система ограничений в форме окружности:

```
void Start()
{
    ghost = FindObjectOfType<RacingCenter>();
    rivals = FindObjectsOfType<RacingRival>();
    foreach (RacingRival r in rivals)
    {
        if (ReferenceEquals(r, ghost))
            continue;
        r.randomPos = Random.insideUnitSphere;
        r.randomPos.y = ghost.transform.position.y;
    }
}
```

Как это работает...

Искусственный интеллект верхнего уровня распределяет гонщиков по позициям в системе адаптивных ограничителей. У каждого гонщика имеется собственная модель регулировки скорости. Модель непобедимого гонщика отличается от прочих. Этот агент работает как центр масс системы адаптивных ограничений. Если игрок станет догонять его, он должен адаптироваться. Если нет, его модель поведения остается неизменной.

Предметный указатель

А

Автомобиль, реализация вождения, 264
Алгоритм A*
  использование
  для нахождения наиболее многообещающего пути, 91
  расширение алгортма A* для координирования, 132
  улучшение использования памяти, 95
Алгоритм AB Negamax, 196
Алгоритм A*mbush, 132
Алгоритм IDA*, 95
Алгоритм Negascouting, 199
Алгоритм Дейкстры, 88
Алгоритм поиска сначала в глубину, 84
Алгоритм поиска сначала в ширину
  нахождение кратчайшего пути, 86
Алгоритм прогнозирования N-Gram
  использование для предугадывания действий, 217
  усовершенствованный, 220

Б

Блуждание вокруг, 29
Боевые круги, 155

В

Влияние
  улучшение карт влияния с помощью заполнения, 147
  улучшение с помощью фильтров свертки, 152

Д

Деревья моделей поведения, реализация, 113
Деревья принятия решений и конечные автоматы, 112
  использование, 225
  осуществление выбора, 104
Достижение цели, 23

З

Закрепление рефлекса, использование, 229

И

Игровой мир
  представление с помощью областей Дирихле, 71
  представление с помощью сети, 61
  представление с помощью точек видимости, 77
Имитация зрения
  с помощью системы, основанной на коллайдере, 165
  с помощью системы, основанной на графе, 175
Имитация обоняния
  с помощью системы, основанной на графе, 179
  с помощью системы, основанной на коллайдере, 171
Имитация слуха
  с помощью системы, основанной на графе, 176

Использование искусственных нейронных сетей, 234

К

Карты, URL-адрес, 71
Карты влияния, 143
Класс дерева игры, работа, 190
Классификатор Байеса, 222
Конечные автоматы
  иерархические, 110
  работа, 107
  соединение с деревьями принятия решений, 112
Кратчайший путь, нахождение с помощью алгоритма Дейкстры, 88
Крестики-нолики, реализация соперника, 201

Л

Лабиринты, программное создание, 261

М

Маршрут
  сглаживание, 100
  следование, 31
Модели поведения
  смешивание по приоритету, 43
  смешивание с помощью весовых коэффициентов, 41
  создание шаблона, 17
  сочетание с помощью конвейера управления, 45

Н

Навигационный меш, использование для представления мира, 81
Навигация, планирование на несколько кадров, 98
Настольный футбол, разработка соперника, 251
Нечеткая логика, работа, 116

О

Области Дирихле, использование, 71
Ограничения из списка constraints, 48
Окно поиска, 199

П

Поиск гармоник, создание эмерджентных частиц, 238
Поиск с квантованием времени, 98
Преследование, 21
Принятие решений
  модели целенаправленного поведения, 123
  реализация, 103
Прогнозирование, предугадывание действий с помощью алгоритма прогнозирования N-Gram, 217

Р

Редакторы, URL-адрес, 80

С

Сайт Game Programming Wiki (GPWiki), 88
Сети
  использование для представления мира, 61
  нахождение кратчайшего пути, 86
Система Маркова, 120
Система, основанная на графе
  использование для имитации зрения, 175

использование для имитации обоняния, 179
использование для имитации слуха, 176
Система, основанная на коллайдере
использование для имитации зрения, 165
использование для имитации обоняния, 171
Система прыжков, создание, 53
Случайные числа, улучшенная обработка, 242
Снаряд
нацеливание, 52
прогнозирование места попадания, 50
стрельба, 49
Создание информированности в стелс-игре, 181
Создание соперника в воздушный хоккей, 245
Создание эмерджентных частиц с помощью алгоритма поиска гармонии, 238
Состояния, представление числовыми величинами, 120
Структура RaycastHit, URL-адрес, 41
Структура пула объектов, URL-адрес, 50

Т

Точки видимости, использование для представления мира, 77
Точки фиксации
анализ, основанный на высоте, 138
анализ, основанный на обзоре и видимости, 140
оценка для принятия решения, 142
создание, 136

У

Убегание, 21
Уклонение от встреч с агентами, 36
Уклонение от встреч со стенами, 39
Управление гонками с адаптивными ограничителями, 265
Уход от погони, 23

Ф

Фильтры свертки
для улучшения представления влияния, 152
Формирования
обработка, 127

Ц

Целенаправленное поведение, 123
Цели, 48

Ш

Шашки, реализация соперника, 206

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным
платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: www.alians-kniga.ru.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Хорхе Паласиос

Unity 5.x. Программирование искусственного интеллекта в играх

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Научный редактор *Киселев А. Н.*
Перевод *Рагимов Р. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 16. Тираж 200 экз.
Веб-сайт издательства: www.дмк.рф

Игровой движок Unity 5 включает в себя множество инструментов, помогающих разработчикам создавать потрясающие игры, снабженные мощным искусственным интеллектом. Эти инструменты вместе с прикладным программным интерфейсом Unity и встроенными средствами открывают безграничные возможности для создания собственных игровых миров и персонажей. Данная книга охватывает как общие, так специальные методы, позволяющие реализовать эти возможности.

Издание задумывалось как исчерпывающий справочник, помогающий расширить навыки программирования искусственного интеллекта в играх. Рассматриваются основные приемы работы с агентами, программирование перемещений и навигации в игровой среде, принятие решений и координации. Описание построено на практических примерах, в виде легко реализуемых «рецептов».

Из этой книги вы узнаете, как:

- с помощью таких алгоритмов, как A* и A*mbush, оснащать агентов возможностями поиска пути;
- создавать представления игрового мира для передвижения по нему агентов;
- формировать систему принятия решений для выполнения агентами различных действий;
- обеспечивать координацию действий разных агентов;
- имитировать работу органов чувств и применять эту имитацию в системе информирования;
- внедрять искусственный интеллект в настольные игры, например крестики-нолики и шашки.

Об авторе

Хорхе Паласиос (http://jorge.palacios.co) - профессиональный программист с более чем семилетним опытом. Занимается созданием игр, работая на различных должностях, от разработчика инструментария до ведущего программиста. Специализируется но программировании искусственного интеллекта и игровой логики. В настоящее время использует в своей работе Unity и HTML5. Также является преподавателем, лектором и оргонизоторов «геймджемов».

ISBN 978-5-97060-436-6

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

Интернет-магазин www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru