Создание агента

Задачи:

- Ознакомиться с компонентом NavMeshAgent
- Написать собственного ИИ

Описание работы:

Для полноценного игрового процесса у нас не хватает врага, добавим его. Пользоваться будем встроенным компонентом NavMeshAgent, в котором уже реализован алгоритм поиска пути, а также есть удобные свойства и методы. Если не пользоваться NavMeshAgent, можно столкнуться с большим количеством проблем, которые связаны не только с реализацией алгоритма поиска пути, но и с построением навигационной сетки. Наша главная задача – познакомиться с функционалом, предоставляемым движком Unity. Этим и займемся.

7.1 Изучение компонента NavMeshAgent

Сразу же перейдем к практике. Увеличим поверхность в несколько раз, удалим подъем и создадим капсулу Enemy.

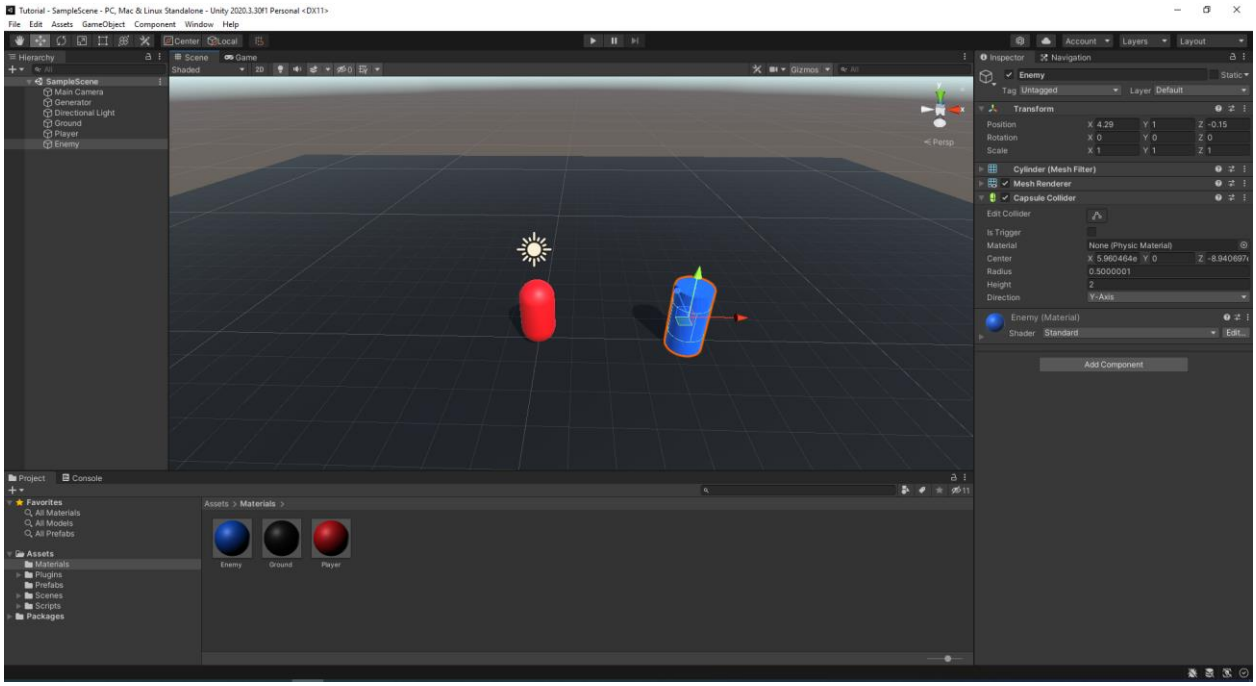

Рисунок 7.1 – Создание врага

Перейдем к настройке NavMeshAgent. Для полноценной работы с этим компонентом, скачаем дополнительные компоненты, не входящие во встроенный в Unity NavMeshAgent. Для скачивания необходимо перейти по ссылке: https://github.com/Unity-Technologies/NavMeshComponents, скачать архив и переместить его в проект. Нужно переместить только папку NavMeshComponents / **Assets**. Переименуем ее в NavMeshAgentAssets.

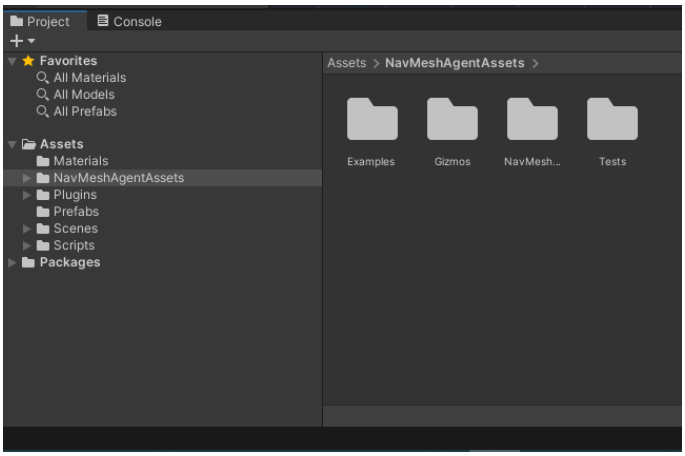

Рисунок 7.2 – Папка NavMeshAgentAssets

Теперь приступим непосредственно к работе. Откроем вкладку, связанную с NavMeshAgent: Window / AI / Navigation.

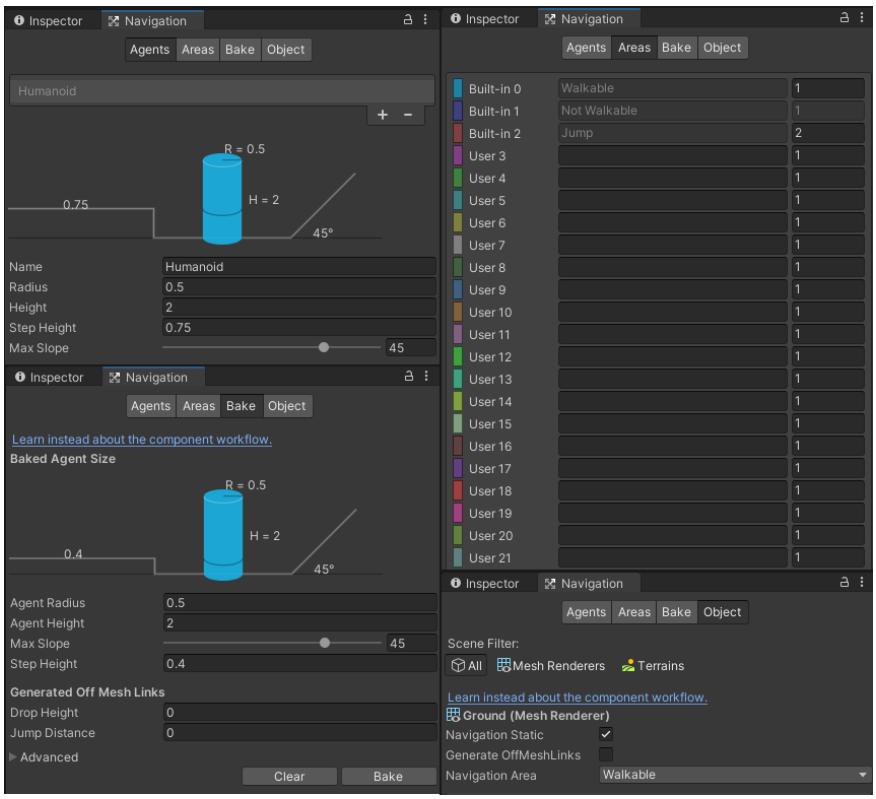

Рисунок 7.3 – Окно «Навигация»

Именно здесь настраивается навигационная сетка, позволяющая ИИ спокойно передвигаться по местности. Сетка создается относительно параметров цилиндра: радиус, высота, максимальный угол подъема, максимальная высота ступени. Процесс создания навигационной сетки называют запеканием (bake). Для правильного запекания необходимо включить Navigation Static у игровых объектов, на котором требуется запечь сетку. Для этого можно, не закрывая окна navigation, перейти в Object и, нажав на нужный игровой объект на сцене, поставить ему галочку. То же самое можно сделать через инспектор.

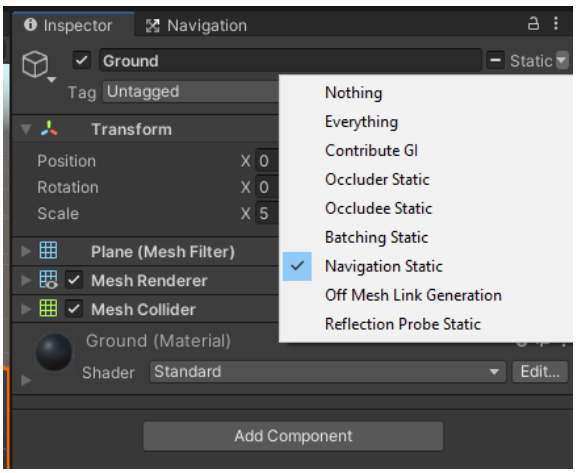

Рисунок 7.4 – Присваивание Navigation Static объекту

Для запекания сетки сетку нужно нажать Bake, чтобы удалить Clear. Если Ground была помечена Navigation Static, то на всей плоскости покажется синяя сетка на сцене. Сетка показывает, куда может перемещаться ИИ.

Во вкладке Areas можно отметить разные слои с указанием их стоимости (веса) специально для ИИ, так как NavMeshAgent пользуется алгоритмом поиска пути A*.

Так же, во вкладке Agents можно создать шаблонные размеры ИИ, для дальнейшего запекания ландшафта для каждого «агента».

Создание навигационной сетки – это только первая часть настройки ИИ. Следующим шагом необходимо настроить самого агента. Навесим на Enemy компонент NavMeshAgent. Компонент уже имеет механику передвижения, так что в самом компоненте можно настроить разные параметры передвижения. Здесь можно выбрать тип агента, относительно которого будет произведено запекание. А также, что очень важно, можно настроить Area Mask – области, по которым будет перемещаться ИИ. То есть, ИИ можно ограничить в передвижении не только его размером и параметрами агента, но и выбранными областями.

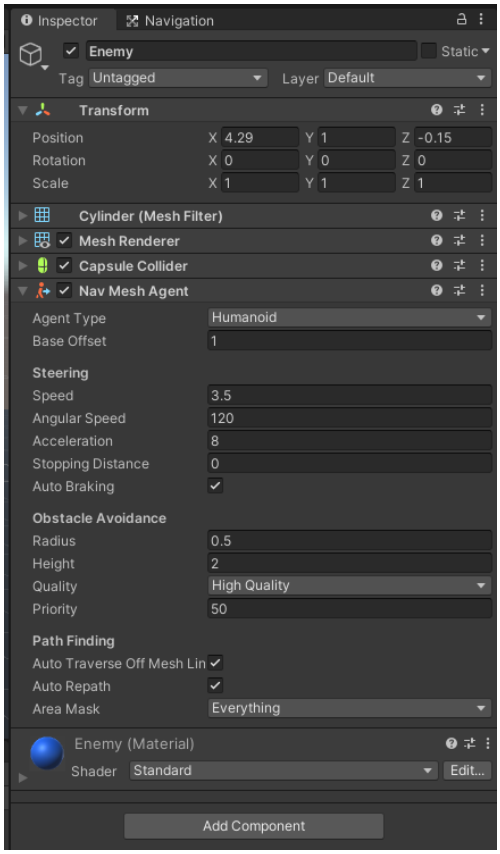

Рисунок 7.5 – Компонент Nav Mesh Agent

При запекании через окно «Навигация» появляется одна проблема. Она заключается в том, что запекание производится относительно не конкретного типа агента, а относительно установленных значений параметров во вкладке Bake. Для решения данной проблемы воспользуемся другим компонентом, который был скачен из официального репозитория разработчиков NavMeshAgent.

Создадим пустой игровой объект и навесим на него компонент NavMeshSurface.

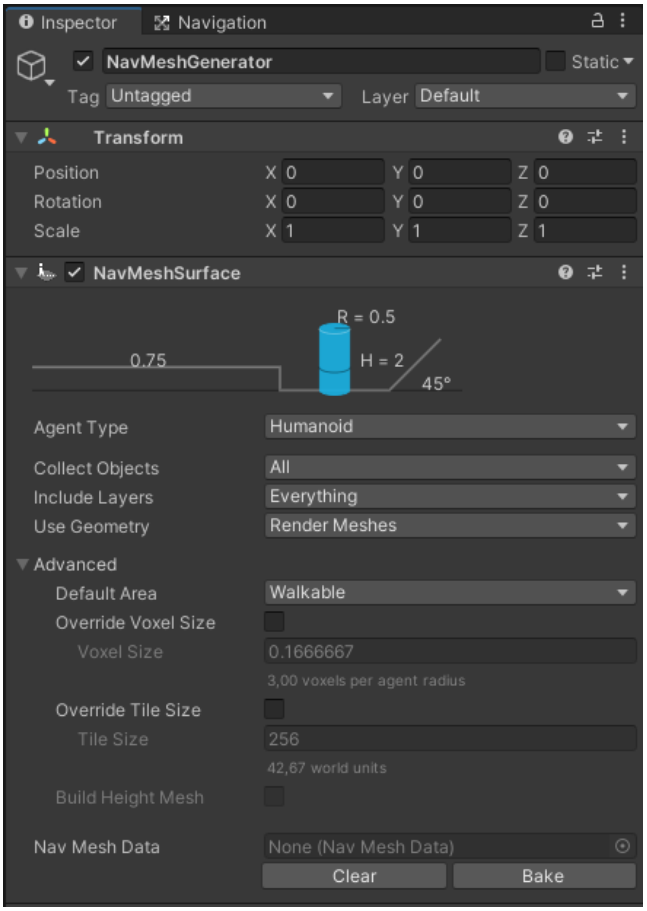

Рисунок 7.6 – Компонент NavMeshSurface

В этом компоненте присутствует возможность настройки запекания для каждого типа агентов. Данные навигационной сетки стандартно сохраняются в Scene / SampleScene. Для запекания через данный компонент не необходимо ставить Navigation Static каждому игровому объекту. Запекание происходит относительно выбранных слоев Include Layers. Выберем только Ground и только созданный слой Obstacle.

7.2 Запекание навигационной сетки

Теперь перейдем к скриптингу. Запекание обычной поверхности неинтересно, а самостоятельно создавать ландшафт обременительно, так что воспользуемся готовой генерацией от расстояния. Как только все игровые объекты будут сгенерированы, запечем весь ландшафт в скрипте DistanceGeneration. Для удобства, навесим скрипт DistanceGeneration на NavMeshGenerator, чтобы удобнее получать ссылку на компонент. (префабу цилиндра нужно поставить новый слой Obstacle).

```
private NavMeshSurface _navMeshSurface;
...
Event function
private void Start()
{
    ...
    _navMeshSurface.BuildNavMesh();
}
```

Рисунок 7.7 – Запекание сетки в классе DistanceGeneration

Если все было сделано правильно, то нажав Play, должны сгенерироваться цилиндры, а после должна была запечься сетка относительно этих цилиндров.

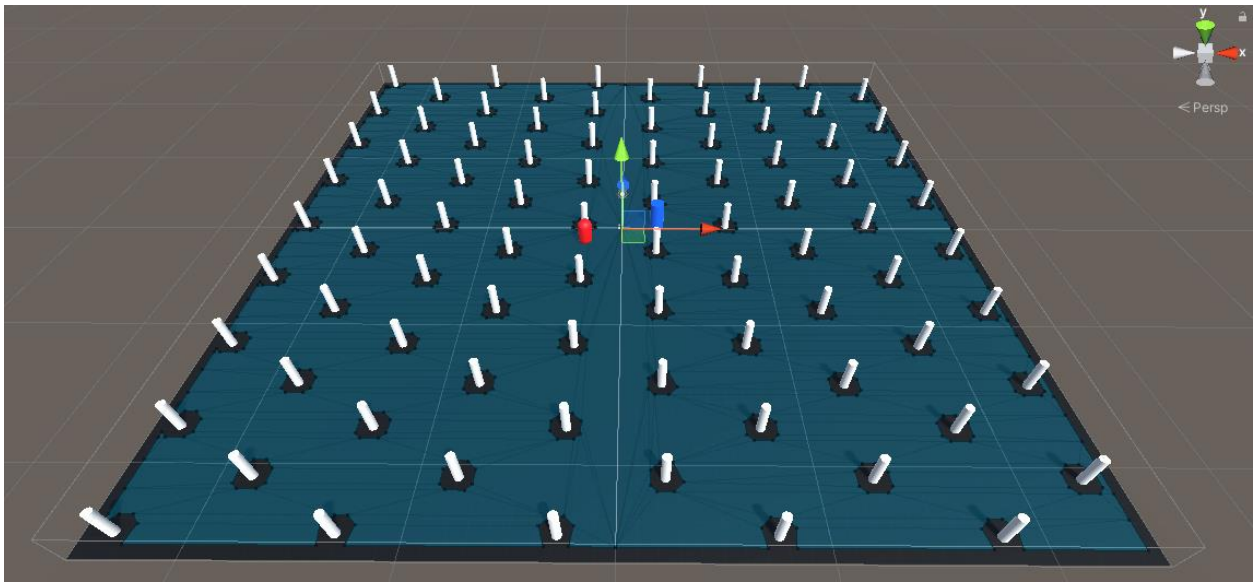

Рисунок 7.8 – Полученная сетка

7.3 Создание ИИ (бота)

Для проверки работоспособности сетки, напишем реализацию передвижения Enemy через нажатия на ЛКМ.

```
using UnityEngine;
using UnityEngine.AI;

1 asset usage  2 usages
public class EnemyController : MonoBehaviour
{
    private NavMeshAgent _navMeshAgent;

    Event function
    private void Awake()
    {
        _navMeshAgent = GetComponent<NavMeshAgent>();
    }

    Event function
    private void Update()
    {
        if (Input.GetMouseButtonDown(0))
            SetDestinationToMousePosition();
    }

    Frequently called  1 usage
    private void SetDestinationToMousePosition()
    {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

        if (Physics.Raycast(ray, out RaycastHit hit))
            _navMeshAgent.SetDestination(hit.point);
    }
}
```

Рисунок 7.9 – Класс EnemyController

Смысл реализации следующий: мы пускаем луч из основной (main) камеры туда, где находится курсор. Дальше просто идет проверка рейкастом, сталкивается ли этот луч с коллайдерами. Если да, то используется внутренний метод перемещения NavMeshAgent SetDestination, которому передаем в параметр точку столкновения. В Update же идет постоянная проверка на то, нажали ли мы кнопку ЛКМ (кнопка ПКМ это 1, колесико 2).

Теперь при запуске проекта и при нажатии на поверхность, Enemy будет передвигаться к точке нажатия, обходя все преграды автоматически.

Конечно, это не ИИ, здесь перемещение зависит от нас, так что перейдем к непосредственной разработке ИИ. Будем реализовывать самую простейшую версию ИИ, умеющую только патрулировать и преследовать. Начнем.

```
public class EnemyController : MonoBehaviour
{
    [SerializeField] private Transform _playerTransform;  Player (Transform)
    [SerializeField] private LayerMask _playerMask;  Serializable
    [SerializeField] private float _sightRange;  "10"
    [SerializeField] private float _walkRange;  "10"

    private NavMeshAgent _enemyAgent;

    private Vector3 _walkPoint;
    private bool _walkPointSet;

    Event function
    private void Awake()
    {
        _enemyAgent = GetComponent<NavMeshAgent>();
    }

    Event function
    private void Update()
    {
        if (IsPlayerInSight()) MoveToTarget(_playerTransform.position);
        else Patrol();
    }
```

Рисунок 7.10 – Начало класса EnemyController

Так как название полей и методов составлялись правильно, то данную часть кода очень легко понять. В Update идет проверка того, видит ли ИИ игрока, если да, то ИИ идет к игроку, если нет, то патрулирует.

Пойдем дальше.

```
private void MoveToTarget(Vector3 target)
{
    _enemyAgent.SetDestination(target);
}

Frequently called  1 usage
private bool IsPlayerInSight()
{
    Vector3 direction = _playerTransform.position - transform.position;

    Debug.DrawLine(start: transform.position, end: transform.position + direction.normalized * _sightRange,
        Color.magenta);
    Debug.DrawLine(start: transform.position + Vector3.up, end: transform.position + direction.normalized * direction.magnitude,
        Color.blue);

    bool player = Physics.Raycast(origin: transform.position, direction, _sightRange, (int)_playerMask);
    bool obstacle = Physics.Raycast(origin: transform.position, direction, direction.magnitude, ~_playerMask);

    return !(!player | obstacle); // implication inversion
}
```

Рисунок 7.11 – Методы MoveToTarget и IsPlayerInSight

С методом MoveToTarget все также понятно, а вот проверку IsPlayerInSight желательно разобрать. В начале метода находим вектор направления от ИИ к игроку, для того, чтобы понимать в какую сторону бросать Raycast. Здесь должны учитываться два фактора: находиться ли игрок в радиусе видимости, и стоит ли перед игроком какое-либо препятствие. Если перед игроком есть какое-то препятствие, то понятно, что ИИ не должен его видеть, если игрока нет в радиусе видимости и нет препятствия, то тоже понятно, что ИИ не должен никак реагировать и т.д. Объединив все условия в единую таблицу, получается таблица истинности.

A	B	A ? B
0	0	0
0	1	0
1	0	1
1	1	0

Рисунок 7.12 – Таблица истинности

Где A – это игрок, а B – это препятствие. Остается теперь понять, каким именно логическим оператором можно связать данные переменные.

Мы это уже сделали и определили, что данная таблица истинности соответствует инверсии прямой импликации.

a	b	$\neg(a \to b), a > b$
0	0	0
0	1	0
1	0	1
1	1	0

Рисунок 7.13 – Таблица истинности инверсии прямой импликации

Так как в C#, Unity, а также во многих других языках программирования часто не предусмотрен отдельный оператор для импликации, то в нашем случае используем такую формулировку как: `!A | B`. В нашем случае нам нужна инверсия импликации, так что: `!(!A | B)`. Для решения данной проблемы есть и более простое решение, без каких-либо сложных операторов, но это остается на вас. Продолжим.

В данном методе появились 2 неизвестные вещи: ~playerMask и Debug.DrawLine.

Первое необходимо, чтобы инвертировать битовую маску игрока. Это означает, что будут учитываться столкновения со всеми игровыми объектами, кроме игрока (оператор ~ инвертирует битовую маску).

Второе – это уже известный класс Debug, необходимый для отладки в режиме исполнения. Метод DrawLine рисует прямую между двумя точками.

Оба DrawLine были использованы для демонстрации работы Raycast для игрока и для препятствий. Raycast для игрока ищет игрока только в пределах видимости, то есть, бросает raycast в сторону direction с длиной _sightRange. Raycast для препятствий ищет препятствие между ИИ и игроком, то есть, бросает raycast в сторону direction с длиной direction.magnitude (длина вектора направления).

Есть одно замечание – методу Raycast можно передавать ненормализованный вектор, так как при необходимости, он сам его нормализует, но в Debug.DrawLine для корректной отрисовки прямой мы нормализовали вектор direction с помощью normalized (2-ой DrawLine был чуть-чуть поднят, чтобы прямые не сливались друг с другом).

Если IsPlayerInSight будет возвращать false, то ИИ будет просто патрулировать, а значит будет вызываться метод Patrol.

```
private void Patrol()
{
    Vector3 directionToPoint = _walkPoint - transform.position;
    if (directionToPoint.magnitude < 1f)
        _walkPointSet = false;

    if (_walkPointSet) MoveToTarget(_walkPoint);
    else SearchPoint();
}
```

Рисунок 7.14 – Метод Patrol

Смысл метода патрулирования заключается в том, что ищется случайная точка в некоторой области. Если она найдена, то _walkPointSet становиться true и в следствии вызывается MoveToTarget к этой точке. Если же_walkPointSet = false, то идет поиск точки. Первое же условие предоставляет информацию о том, когда ИИ подойдет к подобранной точке. Проверка осуществляется при помощи расстояния, когда оно будет меньше одного юнита, то _walkPointSet станет false. Перейдем к методу SearchPoint.

```
private void SearchPoint()
{
    float randomOffsetX = Random.Range(-_walkRange, _walkRange);
    float randomOffsetZ = Random.Range(-_walkRange, _walkRange);
    _walkPoint = new Vector3(x: transform.position.x + randomOffsetX, transform.position.y,
        z: transform.position.z + randomOffsetZ);

    NavMeshPath path = new NavMeshPath();
    _enemyAgent.CalculatePath(_walkPoint, path);
    if (path.status == NavMeshPathStatus.PathComplete)
    {
        _walkPointSet = true;
        Debug.DrawLine(start: _walkPoint, end: _walkPoint + Vector3.up, Color.green, duration: 3f);
    }
    else
        Debug.DrawLine(start: _walkPoint, end: _walkPoint + Vector3.up, Color.red, duration: 3f);

}
```

Рисунок 7.15 – Метод SearchPoint

Первые 3 строчки должны уже быть понятны, так что их пропустим. Дальше интереснее. Создается объект NavMeshPath path для того, чтобы в дальнейшем сохранить информацию пути в этой переменной, при помощи метода CalculatePath. Это делается, чтобы случайно подобранная точка находилась в том месте, в которое сможет попасть ИИ.

Метод CalculatePath заранее вычисляет и сохраняет информацию о пути. Этой информацией мы и пользуемся, проверяя статус пути. Всего статусов 3: Complete, Partial и Invalid. Нам необходимо, чтобы путь имел статус Complete, то есть путь заканчивался в пункте назначения. Если статус Complete, то _walkPointSet становиться true, и отрисовывается прямая в точке назначения зеленый цветом, если нет, то отрисовывается прямая, но другим цветом.

Последний метод – это метод OnDrawGizmos. Он является внутренним методом класса MonoBehaviour, также, как Start, Update и остальные. Его задача заключается в том, чтобы визуализировать Gizmos в режиме просмотра сцены.

```
private void OnDrawGizmos()
{
    Gizmos.color = Color.yellow;
    Gizmos.DrawWireSphere(transform.position, _sightRange);
    Gizmos.color = Color.cyan;
    Gizmos.DrawWireCube(center: transform.position,
        size: new Vector3(x: _walkRange * 2, transform.position.y, z: _walkRange * 2));
}
```

Рисунок 7.16 – Метод OnDrawGizmos

Gizmo – это класс для визуальной отладки или для помощи при настройки в режиме просмотра сцены. В данном случае, мы создаем Gizmo сферы и куба. Первое показывает радиус, в котором ИИ может увидеть игрока, второй – радиус, в котором будут подбираться точки для патрулирования.

Конечно, все дебаги, отрисовки и визуализации не необходимы, без них код станет намного лаконичней. Но с ними легче проводить настройку и поиск багов. Так что рекомендуется не избегать данного удобного инструмента.

Так как мы отрисовывали каждое действие ИИ, то при запуске проекта, можно наблюдать, какие ИИ подбирает точки, были ли они подходящими (зеленая прямая), или нет (красная), в каком радиусе, и где относительно ИИ находится игрок.

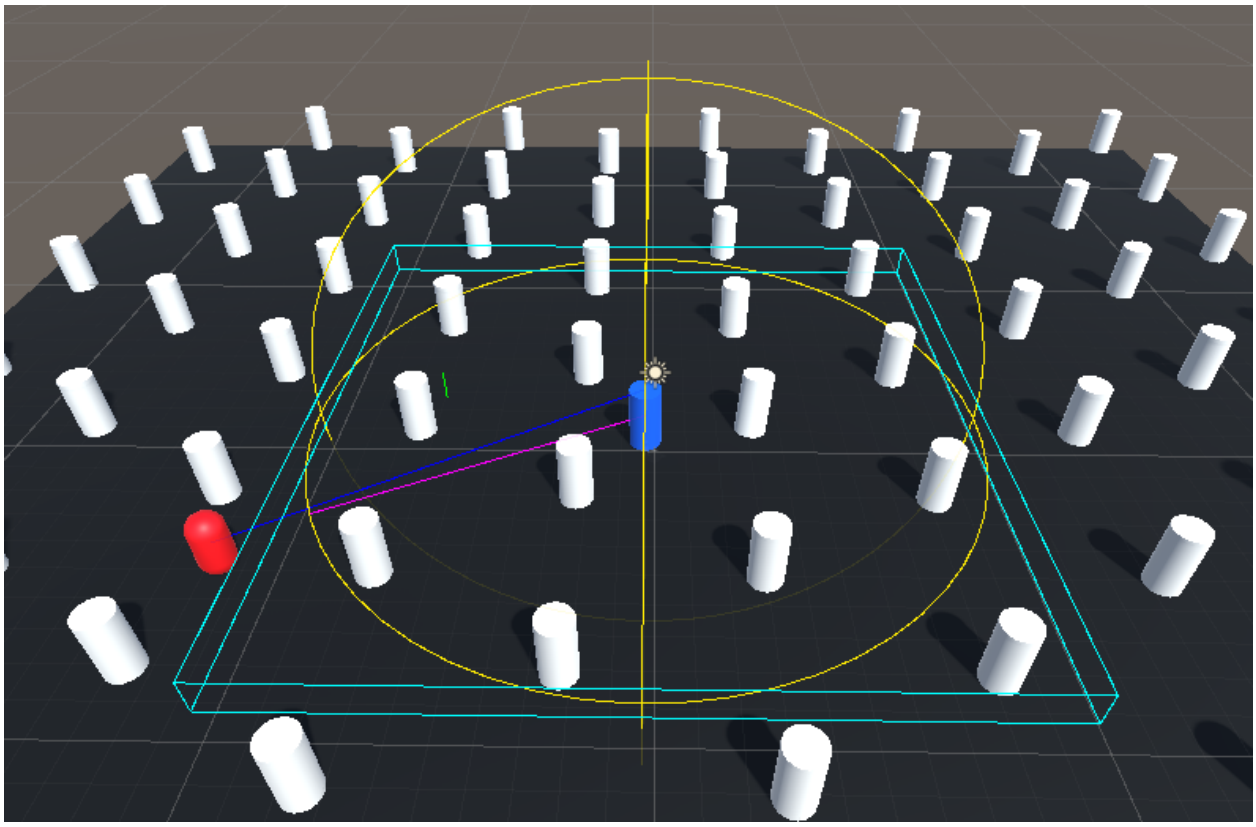

Рисунок 7.17 – Отрисовка всех действий ИИ

Данная реализация ИИ неидеальна и требует доработок. К примеру, так как проверка на препятствие идет с помощью прямой, то даже если игрок будет стоять за тонким цилиндром, ИИ его не обнаружит. Это можно исправить, если бросать от врага несколько лучей от границ его видимости, или же просто бросать не луч, а цилиндр. Но дальнейшие усложнения и доработки, на данный момент, не необходимы. Для того, чтобы понимать и программировать что-то сложное, нужно понимать и уметь делать что-то простое.

Самостоятельное задание:

Добавить ИИ скрытное поведение. Нужно добавить радиус, в котором ИИ не будет преследовать игрока, но будет его видеть. Прячась за объектами, он должен постепенно подходить к игроку, пока не достигнет ближнего радиуса, в котором ИИ начнет прямо преследовать игрока.