

# **Система Навигации И Поиска Пути**

## **Руководство пользователя**

## Содержание

Введение	3
Что такое СНИПП?	3
Навигация в СНИПП	3
Работа с редактором СНИПП	4
Начало работы	4
Типы путей. Создание, удаление, визуализация.	6
Создание узлов пути	7
Работа со связями узлов	10
Атипичные соединения	11
Работа с кривыми	12
Дополнительные настройки	13
Работа с файлами навигации и динамическая загрузка данных	14
Что такое бипоинты и как их использовать?	17
Функции библиотеки поиска путей для работы с ИИ	19
Классы библиотеки поиска путей для работы с ИИ	35
Класс Node	35
Класс RouteData	36
Класс MotionCurve	37
Класс CurveData	38
Класс Path	39
Основные компоненты СНИПП	40
Библиотека методов и классов СНИПП - Pathfinding	40
Компонент Navigator	40
Компонент NPCMovement	45
Контактная информация	47

## **Введение**

### **Что такое СНИПП?**

Система Навигации и Поиска Путей (СНИПП) или Navigation And Pathfinding System(NAPS) позволит вам быстро осуществить навигацию для любого типа ИИ на вашем уровне. СНИПП сочетает в себе простоту использования, так называемых, вейпоинтов и возможность описания больших территорий, которую дает navigation mesh. При этом вы получаете отличную производительность, что очень важно, при использовании навигации на уровне большим количеством экземпляров ИИ. С помощью функций библиотеки СНИПП вы можете построить любой алгоритм следования по путям, от простого следования от точки к точке, до сложной системы перемещения, основанной на видимости определенных зон, с учетом оптимального пути, срезания углов, возобновление маршрута в случае потери видимости точки и др. Также важной особенностью СНИПП является алгоритм предрасчета большей части данных путей. Это значит, что вам не нужно производить сложный расчет при прокладывании маршрута из одной точки карты в другую, что приходится делать в случае использования алгоритма A\*. Базовый расчет проводится единожды, в случае изменения структуры сетки путей, а значит вы можете использовать огромное количество юнитов, которые одновременно прокладывают для себя пути, с минимальными потерями производительности. При этом все данные навигации хранятся в файлах, которые вы можете динамически загружать/выгружать для использования на разных участках карты или же разных типов путей. Это удобно при работе с очень большими территориями, когда загрузка всех данных навигации одновременно, слишком накладна. Для более подробной информации рассмотрите приложенные примеры ИИ перемещения, а также описание функций библиотеки поиска путей СНИПП(Pathfinding.cs).

### **Навигация в СНИПП**

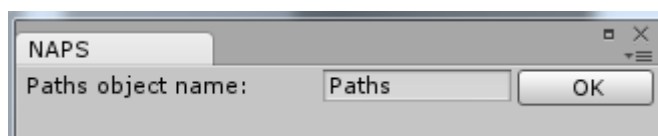
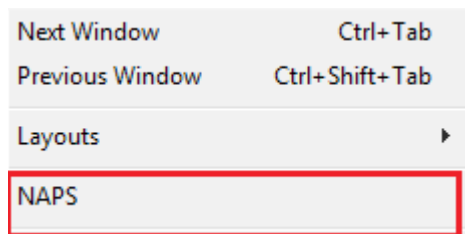
Навигация в СНИПП осуществляется благодаря сетке путей, состоящей из узлов. Между любыми узлами этой сетки можно проложить кратчайший путь. Для того чтобы разделить перемещения различных типов ИИ, общую сеть путей можно разбивать на подсети, тогда определенному типу ИИ можно задать допустимые подсети которые он может использовать для движения, а не указанные подсети не будут использоваться данным типом ИИ. Так как же создаются эти подсети? Мы уже говорили о том, что сеть состоит из узлов. Каждый узел имеет тип, тип представляет собой всего лишь набор символов, то есть имеет строковое представление и любое название, которое вы придумаете, может

послужить типом узла. Объединение узлов одного типа и образует подсеть. Каждый узел имеет также уникальный порядковый номер внутри своей подсети путей. Допустим в вашей игре есть два типа существ, это люди и драконы. Пускай люди имеют возможность ходить только по наземным маршрутам(лестницам, комнатам, дорогам и т. д.). Драконы также могут использовать некоторые человеческие маршруты, когда не находятся в воздухе, но драконы должны уметь летать еще и по своим специальным «воздушным» путям, в отличии от людей. Таким образом целесообразно создать два типа подсетей путей , один - только для людей, второй — только для драконов. Назовем подсеть путей, которую будут использовать люди - «human», а подсеть, которую будут использовать драконы - «dragon». Таким образом, разместив на земле узлы типа «human», мы создадим подсеть наземных путей, а подсеть «dragon» создадим, разместив в воздухе узлы типа «dragon». И так, мы получили две отдельные подсети, которые могут использоваться по отдельности. Но ведь нам нужно, чтобы драконы могли использовать не только подсеть «dragon», а еще и подсеть «human». Для этого нужно создать переходы между этими подсетями, соединив некоторые их узлы, более подробно об этом в разделе «Атипичные соединения». Преимущества подсетей еще и в том, что каждая подсеть путей рассчитывается предварительно, а не в реальном времени. Это значит, что ИИ, который может использовать сразу несколько подсетей, как дракон, о котором говорилось выше, не нуждается в сложном расчете своего маршрута, так как использует уже рассчитанные данные. При написании ИИ перемещения вы сможете определить какой тип ИИ может использовать тот или иной тип подсети(смотрите примеры ИИ).

## Работа с редактором СНИПП

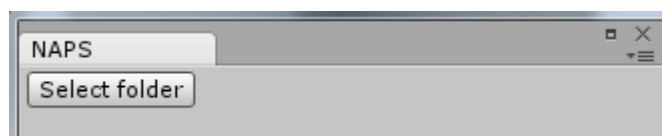
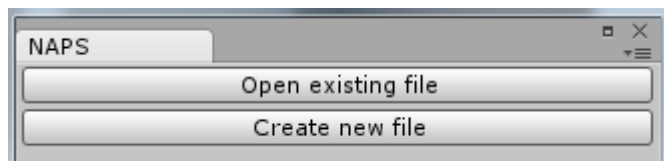
### Начало работы

Для того чтобы открыть окно СНИПП, выполните в главном меню редактора Window\NAPS и увидите окно СНИПП.

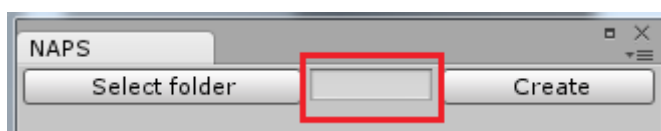


В поле «Path object name» по умолчанию записано «Paths» - это имя объекта, который будет хранить объекты всех узлов сетки путей в сцене. Можете изменить его, если это имя уже используется для какого ни будь

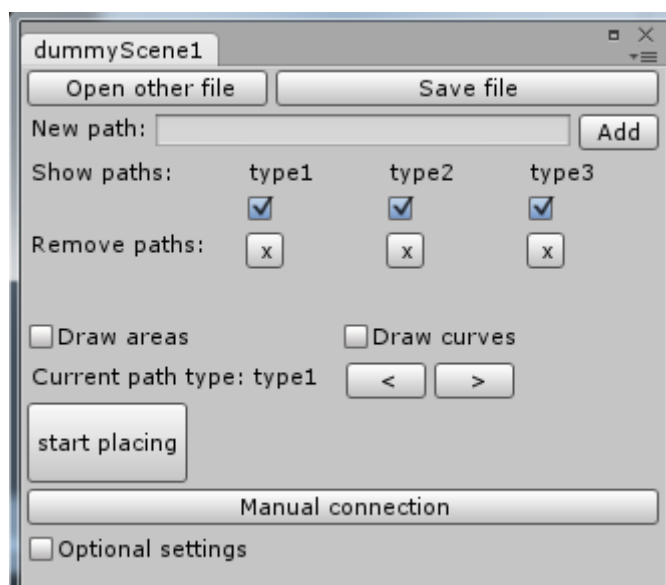
объекта в сцене, так как это имя должно быть уникальным. После нажатия кнопки «ОК» откроется файловый диалог.



Если у вас уже есть файл навигации СНИПП вы можете открыть его, нажав на кнопку «Open existing file». В противном случае нажмите кнопку «Create new file», для того чтобы создать новый файл путей. При этом вы увидите кнопку выбора папки «Select folder», по нажатию которой откроется диалог выбора папки для сохранения файла. После того, как папка для сохранения файла выбрана, появится диалог для ввода имени файла.

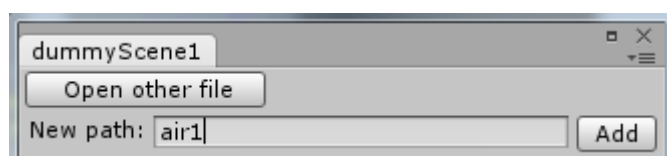


Введите имя файла в поле и нажмите кнопку «Create». При создании нового файла СНИПП фактически создает три файла, два из которых с одинаковыми именами. Один из них хранит непосредственно данные навигации и имеет расширение \*.nvf, этот файл является основным при работе с навигацией. Второй файл хранит данные о цвете отображения подсетей в редакторе, которые находятся в файле с таким же именем, но с разрешением \*.nvf. Этот файл имеет расширение \*.pst и является вспомогательным, то есть файл навигации может работать без этого файла, но информация о цветах отображения подсетей путей в редакторе СНИПП при этом сбросятся на стандартные. Третий файл — это файл пользовательских настроек, он сохраняет все дополнительные параметры редактора СНИПП, кроме цветов отображения подсетей путей. Этот файл носит название «Settings» и имеет расширение \*.wst, по умолчанию располагается в корневой директории СНИПП. После открытия имеющегося файла или создания нового вы увидите главное меню СНИПП.



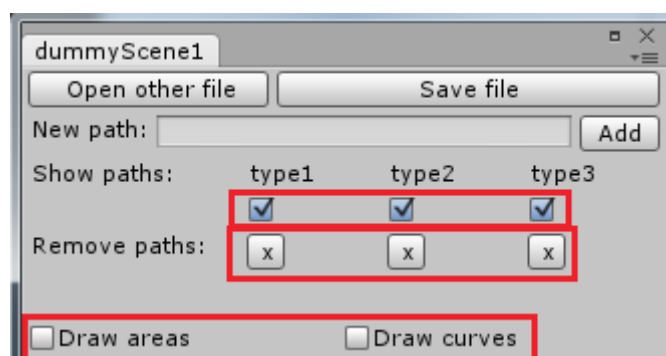
### Типы путей. Создание, удаление, визуализация.

Для того чтобы создать новую подсеть путей, введите ее название (тип) в поле «New path» в верхнем левом углу окна СНИПП и нажмите кнопку «Add». Так вы создадите новую пустую подсеть путей, в которую сможете добавить узлы.



В качестве названия типа подсети можно использовать любую комбинацию цифр и символов, но для собственного удобства, рекомендую давать им информативные названия, например «human», что показывает, что этот тип пути преимущественно используют люди, или подобные существа.

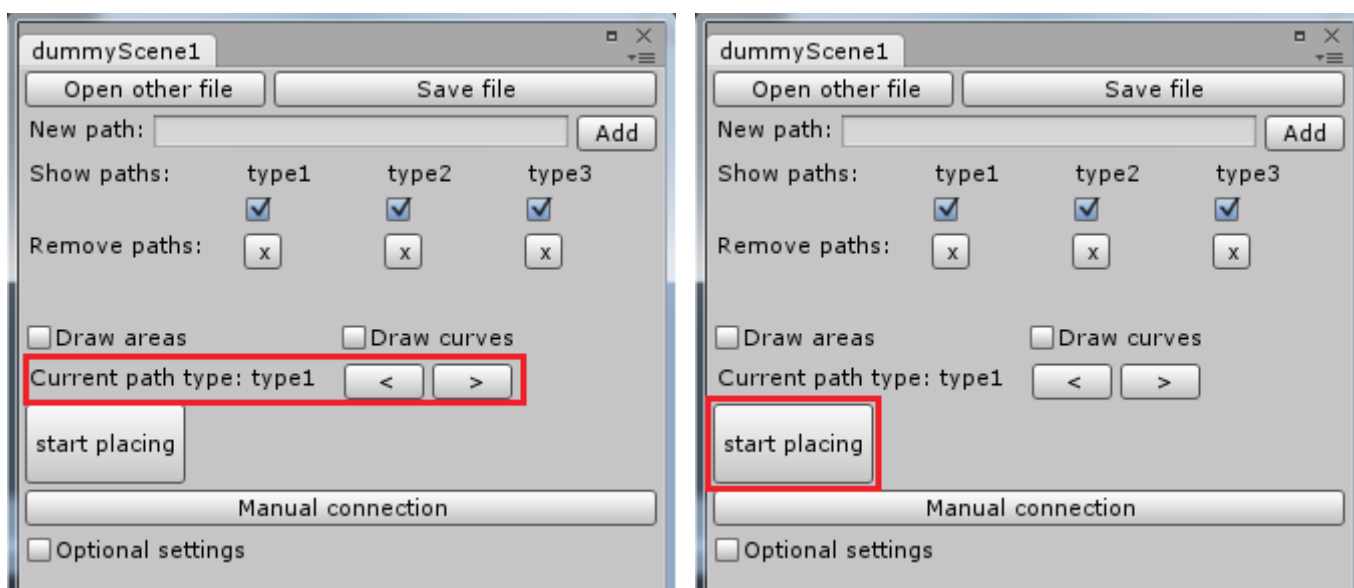
После того, как будет создана одна или более подсетей путей, появится опция визуализации путей. Здесь вы можете отключить отображение всех или только некоторых подсетей снимая/устанавливая флажок под названием подсети.



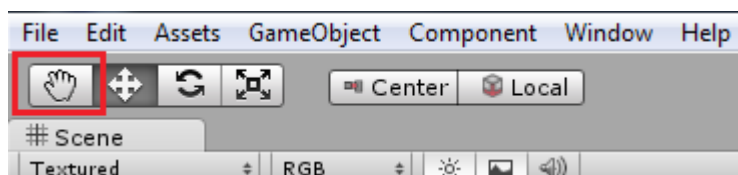
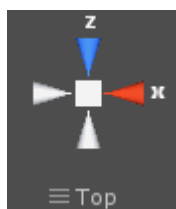
Если какая либо подсеть путей больше не нужна, то вы можете удалить ее нажав кнопку с изображением крестика под названием подсети. Вы также можете включить/отключить визуализацию областей перемещения, а также кривых, устанавливая/снимая флажки «Draw areas» и «Draw curves» соответственно. Визуализация большого количества связей узлов может сильно понижать количество кадров в секунду в редакторе, поэтому для комфортной работы удобно отключать визуализацию ненужных, в данный момент, подсетей.

## Создание узлов пути

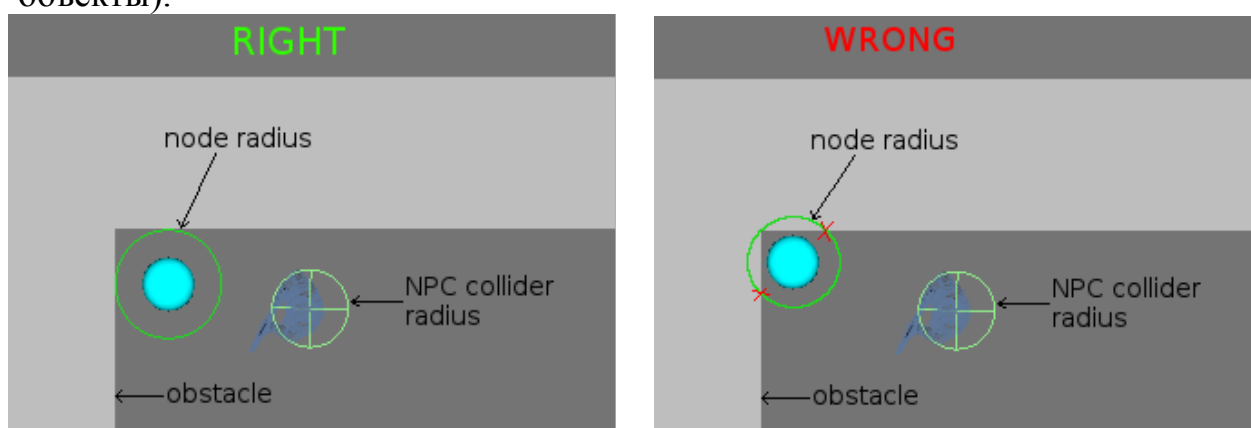
После того, как вы добавили одну или более подсетей, можно приступить к созданию узлов внутри одной из них. Для этого нужно выбрать тип подсети, в которой хотите создать узел, это можно сделать нажимая кнопки стрелок в поле «Current path type». Текущий тип подсети пути при этом будет отображаться слева от кнопок(или справа от названия поля).



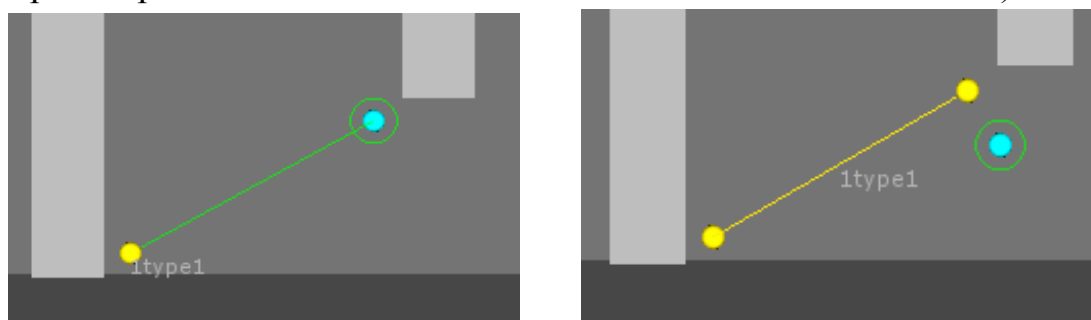
После того, как нужная подсеть была выбрана, нажмите кнопку «Start placing». Для удобства разверните окно сцены на весь экран(клавиша «пробел» по умолчанию), для того, чтобы размещать точки используется направление взгляда камеры сцены. Если планируете прокладывать путь на открытой местности, а не внутри комнаты или другого помещения, то можете включить вид сверху (в сцене в правом верхнем углу нажать на стрелочку указывающую на верхнюю часть квадрата) и включить ортогональный вид(нажать на сам квадратик).



После чего можете перемещать камеру с помощью инструмента «Hand» . Когда взгляд камеры будет направлен на объект с назначенным коллайдером, вы увидите в центре экрана изображение кружка на поверхности объекта. Радиус кружка показывает радиус точки, этот радиус должен быть равен, или быть немного больше радиуса коллайдера персонажа который сможет в дальнейшем использовать эту точку. Настроить радиус можно в дополнительных параметрах окна СНИПП. Этот кружок нужен лишь для правильного расположения точек, так что следите за тем, чтобы он никогда не был перекрыт другими объектами, сквозь которые персонаж не сможет пройти(обычно это статические объекты).



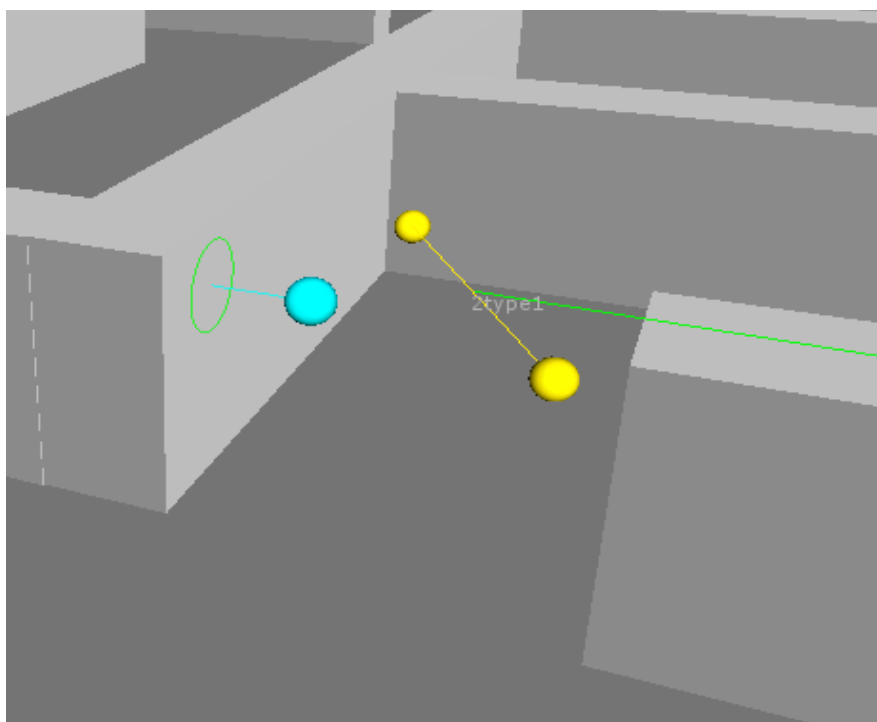
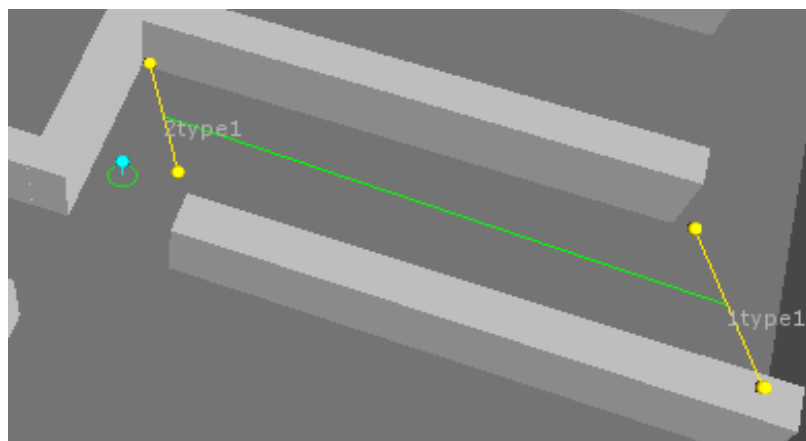
Вы можете размещать как одиночные точки, так и бипоинты. Для того, чтобы разместить одиночную точку в нужном месте нажмите цифру «1» затем «2» .Вы увидите созданную точку. Чтобы разместить бипоинт нажмите «1» в первой точке, затем переместите камеру в место расположения второй точки и снова нажмите «1», и вы увидите созданный бипоинт, между точками которого, проходит линия, цвет которой вы можете настроить в дополнительных параметрах(подробнее о бипоинтах смотрите в разделе «Что такое бипоинты и как их использовать?»).



Одиночные точки имеет смысл размещать в местах , где вам не нужна вариативность пути, например - узкий переулок. Для этого достаточно на входе и на выходе переулка поместить по одной одиночной точке, потому что внутри узкого прохода двигаться боту все равно придется по прямой



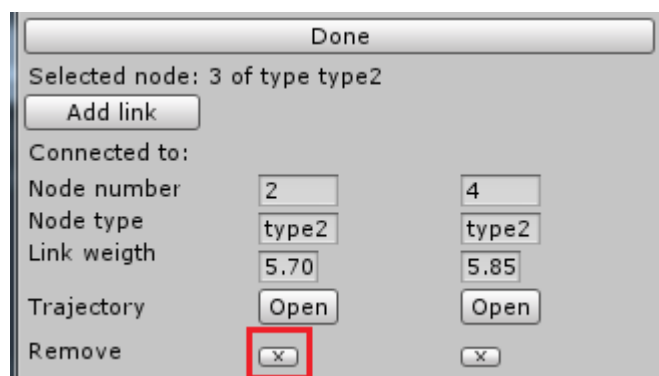
линии. Если же наоборот, нужно расположить точки на широкой открытой улице, так, чтобы бот имел возможность использовать максимум площади для перемещения, то нужно использовать именно бипоинты. Для размещения узлов внутри закрытых помещений удобнее всего использовать режим камеры сцены «Fly». Для этого включите перспективное отображение для камеры сцены(нажмите на квадратик в верхнем правом углу вида сцены). После этого зажмите ПКМ и с помощью кнопок W,A,S,D можете «летать» камерой как в FPS игре, при этом, вместо прицела вы будете видеть позицию размещаемой точки.



Любой созданный узел автоматически соединяется с окружающими узлами такого же типа. Вы можете корректировать положение узлов с помощью инструмента «Move». Для этого просто нажмите ЛКМ на требуемом узле в сцене и переместите его как обычный объект.

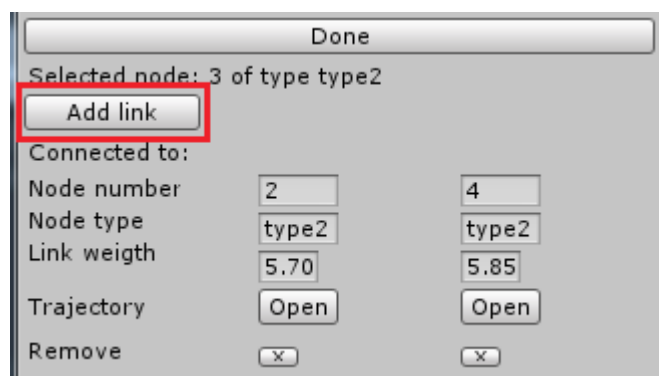
## Работа со связями узлов

Когда вы создаете новые узлы в определенной подсети путей, то они автоматически соединяются со всеми видимыми узлами этой подсети. Связи между узлами могут быть двухсторонними или односторонними. Двухсторонняя связь между двумя узлами — это связь, при которой из одного узла можно попасть в другой узел и наоборот. Односторонняя связь между двумя узлами — это связь, при которой можно перейти от одного узла в другой, но не наоборот. В некоторых случаях это полезно, если вы хотите, к примеру, создать путь или участок пути только с односторонним движением. Для каждого созданного узла автоматически устанавливаются двухсторонние связи со всеми видимыми узлами его подсети. Для того, чтобы убрать лишние связи или добавить новые, выделите кликом мыши любую точку нужного узла и в окне СНИПП нажмите кнопку «Manual connection». Вы увидите диалог для работы с соединениями. Если вам нужно удалить определенные связи данного узла с другими узлами, то обратите внимание на список присутствующих соединений с другими узлами под заголовком «Connected to». Затем найдите удаляемый узел в списке и нажмите кнопку «Remove» под его данными.

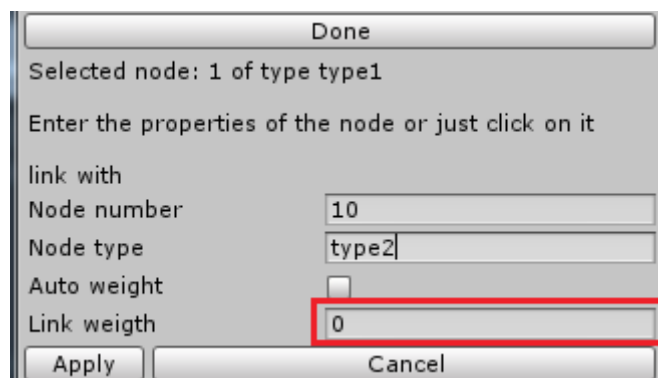
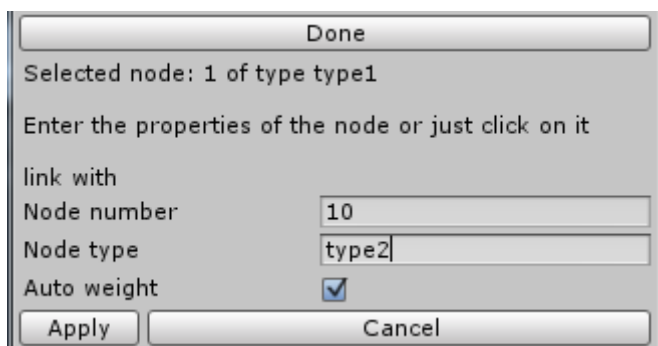


Но помните, что при этом вы удаляете связь данного узла со вторым узлом, но не наоборот. То есть пройти напрямую из данного узла во второй узел будет уже невозможно, но можно пройти из второго узла в данный (остается односторонняя связь). Для того, чтобы полностью удалить любые связи между этими узлами, нужно также удалить из списка связей второго узла данный узел.

Для того чтобы добавить новую связь данного узла с другим узлом нажмите кнопку «Add link».



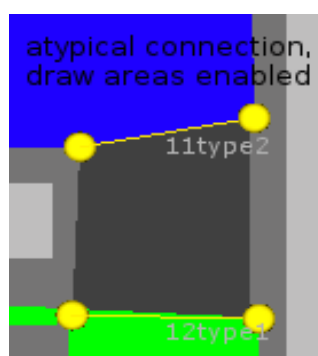
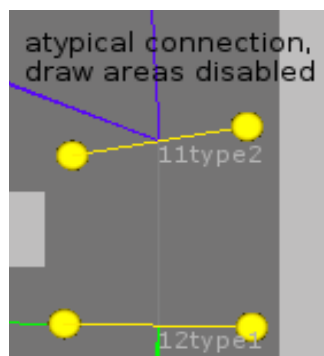
В появившемся меню введите номер и тип нужного узла и нажмите кнопку «Apply».



Если флажок «Auto weight» установлен, то вес соединения установится автоматически(в качестве значения принимается расстояние между узлами), в противном случае вы можете ввести любое требуемое значение. Для того, чтобы отменить добавление новой связи нажмите кнопку «Cancel».

### Атипичные соединения

Для того, чтобы NPC могли использовать несколько типов подсетей путей, нужно создать атипичные соединения между узлами этих подсетей. Эти соединения служат своеобразными «мостиками» между определенными типами подсетей, они показывают из каких узлов одной подсети можно перейти в какие либо узлы другой подсети. Для большего понимания, можно, опять же, привести пример с людьми и драконами. Допустим есть подсеть под названием «human» и люди могут использовать все узлы этой подсети. Есть также подсеть «dragon» - воздушный путь, который могут использовать драконы. Но драконы должны уметь еще и спускаться на землю и двигаться по некоторым наземным маршрутам. Таким образом получается, что драконы, в отличие от людей, должны уметь использовать два типа подсетей. И здесь нам помогают атипичные соединения. Атипичные соединения — это обычное одностороннее или двухстороннее соединение между точками разных подсетей.

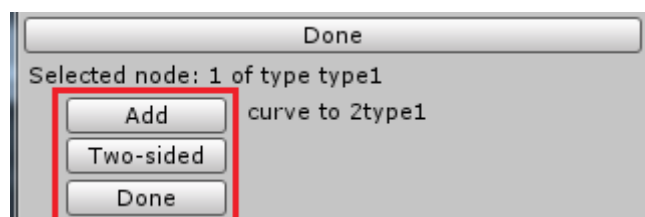
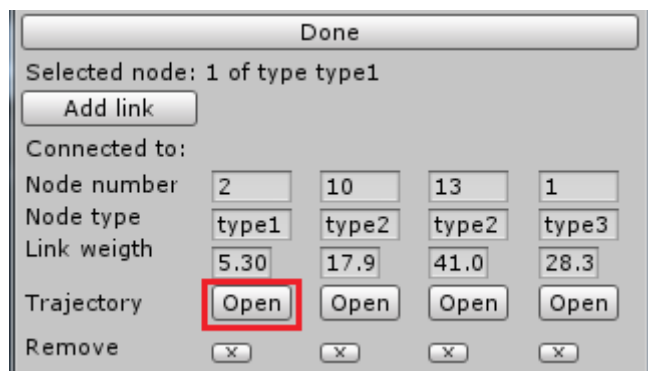


Так что с их помощью мы можем объединить несколько разных подсетей в одну. В этом случае дракон будет знать из какого узла своей подсети («dragon») можно перейти в наземную подсеть («human»).

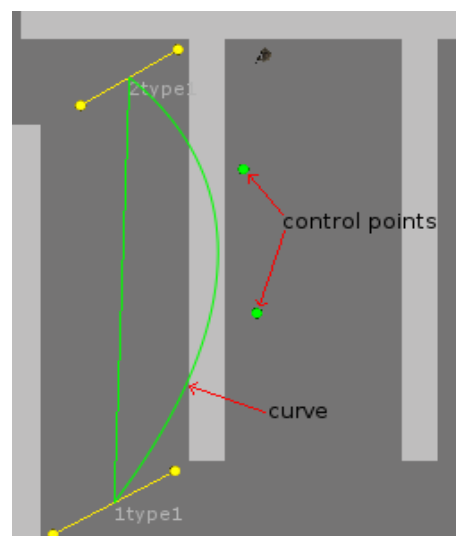
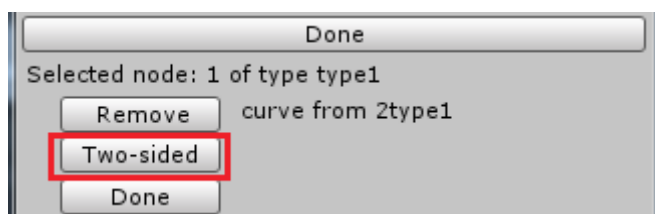
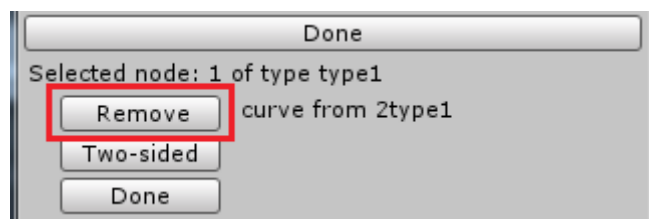
## Работа с кривыми

В СНИПП вы можете добавлять кривые Безье к существующим соединениям между узлами, это может пригодиться, к примеру, если нужно создать кинематографические сцены в вашей игре. Для этого между узлами задаются кривые, по которым может двигаться любой объект, в данном случае — камера. Затем можно записать последовательность узлов, соединенных кривыми, как маршрут для камеры, создав кинематографичный пролет.

Для того, чтобы добавить кривую между парой узлов, нужно, чтобы между этими узлами имелась связь. Если соединение между узлами существует, то к нему можно добавить кривую. Для этого нужно выделить узел, который хотите соединить кривой с другим узлом, затем перейти в меню «Manual connection», для работы с соединениями. В списке соединений найдите следующий узел, с которым хотите соединить кривой данный узел. Если соединения со вторым узлом не существует, то вы можете добавить его через меню «Add link», как при работе с обычными соединениями. Когда соединение существует, вы можете видеть в параметрах соединения поле «Trajectory» и кнопку «Open» напротив него.



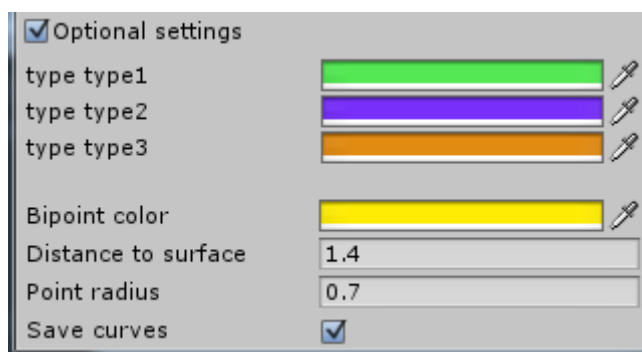
Нажав кнопку «Open» вы откроете диалог работы с кривыми. Для добавления новой кривой к соединению нажмите кнопку «Add». Для того, чтобы кривая отобразилась в сцене, установите флажок «Draw curves» в главном меню СНИПП. При добавлении новой кривой в центральных точках бипойнта отобразятся небольшие сферы, аналогичные сферам, которые обозначают точки узлов - это контрольные точки кривой, перемещая которые, вы можете задать любой вид данной кривой.



Если же вам нужно сделать данную кривую доступной из второго узла (двухсторонняя связь), то вы можете сделать это, нажав кнопку «Two-sided» или же вручную добавить кривую ко второму узлу. Если кривая уже добавлена к текущему соединению, то в диалоге работы с кривыми вместо кнопки «Add» будет отображаться кнопка «Remove», по нажатию которой кривая, из данного соединения, будет удалена. Для выхода из диалога работы с кривыми нажмите кнопку «Done».

## Дополнительные настройки

В меню дополнительных настроек вы можете изменить параметры визуализации сетки путей в сцене, а также некоторые другие параметры. Для того, чтобы открыть меню дополнительных настроек, установите флажок «Optional settings» в главном меню СНИПП.



В верхней части меню находятся поля с названием «type», напротив которых находятся названия подсетей путей, а также поля настройки цвета. Эти параметры отвечают за цвета отображения соответствующих подсетей в сцене. Вы можете работать с ними, как с обычными полями настройки цвета. Эти настройки сохраняются автоматически при

изменении структуры сетки путей и хранятся в файле с расширением \*.pst. Под настройками цвета подсетей находится поле «Vipoint color», этот параметр отвечает за цвет отображения непосредственно самих узлов сетки путей. Параметр «Distance to surface» указывает расстояние, на котором по умолчанию находится размещаемый узел от поверхности, на которой он был размещен. Этот параметр нужен лишь для удобства размещения узлов на фиксированной дистанции от поверхности, при необходимости вы сможете изменить позицию узла в редакторе, используя инструмент «Move». Параметр «Point radius» отвечает за визуализацию диаметра размещаемого узла, непосредственно при размещении, то есть после нажатия кнопки «Start placing». Для того, чтобы правильно разместить узел, нужно установить параметр «Point radius» равным радиусу коллайдера персонажа, который сможет перемещаться через этот узел + небольшое приращение, для страховки. Например, радиус коллайдера персонажа равен 0.5 м, тогда параметр «Point radius» следует установить равным, примерно, 0.6м. После чего нужно размещать узел так, чтобы отображаемый в центре экрана кружок, не перекрывался непроходимыми препятствиями.

Флажок «Save curves» отвечает за сохранение данных кривых в текущем редактируемом файле. Если флажок установлен, то все кривые будут сохранены вместе с остальными данными навигации в файл. В случае, если вам не нужно применять кривые в данном файле навигации, то снимите этот флажок, так как это сократит размер файла, если его размер настолько велик, что это имеет значение. В противном случае, то есть, если вы не используете кривые в текущем файле навигации, но флажок «Save curves» был установлен, в файле навигации будет зарезервировано место для хранения кривых, что незначительно увеличит размер файла. Все параметры, кроме параметров цвета отображения подсетей, сохраняются в файле Settings.wst в корневой директории СНИПП.

## **Работа с файлами навигации и динамическая загрузка данных**

Данные навигации СНИПП сохраняются в файл с расширением \*.nvf. При сохранении данных, редактор СНИПП также создает файл с расширением \*.est, который хранит лишь настройки отображения подсетей путей, которые хранятся в файле навигации. В случае потери или повреждения файла \*.est можно обойтись без него, так как для работы с навигацией важен лишь файл с расширением \*.nvf. В СНИПП вы можете

динамически загружать данные навигации, для этого в библиотеке Pathfinding.cs имеется функция загрузки данных — LoadNavigationData.

*static function LoadNavigationData(string filePath, string fileName) : Void*

Для того, чтобы загрузить данные просто вызовите эту функцию, указав путь к файлу, как filePath и имя файла, как fileName, расширение файла при этом вводить не нужно.

Пример:

```
Pathfinding.LoadNavigationData( "D:\NAPS\Paths", "newMap");
```

Еще одной полезной функцией, при работе с данными навигации, является функция DataLoaded библиотеки Pathfinding.cs. Эта функция возвращает результат булева типа, true – в случае, если данные навигации загружены; false – в противном случае. Вы можете создать много файлов навигации для вашего игрового уровня, если он слишком огромен, а затем загружать файлы с нужными участками данных. Ниже приведен код скрипта PathfindingManager.cs, который отвечает за загрузку данных навигации.

Код:

```
using UnityEngine;
using System.Collections;

public class PathfindingManager : MonoBehaviour {
    //содержит имя файла
    public string fileName = "dummyScene1";
    //содержит путь к файлу
    public string filePath = "D:/Unity
Projects/NAPS/Assets/NAPS/Paths";
    /*определяет можно ли автоматически загружать данные навигации
при загрузке сцены
*/
    public bool autoload = false;
    //флаг для работы с интерфейсом
    bool newFilePathDialog = true;

    void Awake(){
        //если автозагрузка запрещена,
        if(!autoload)
        // прекратить выполнение
        return;
        //иначе, если данные навигации еще не загружены,
        if(!Pathfinding.DataLoaded()){
```

```

//загрузить данные навигации
    Pathfinding.LoadNavigationData(filePath,fileName);
}

}

void Update(){
//если автозагрузка запрещена,
    if(!autoload)
// прекратить выполнение
        return;
//иначе, если данные навигации еще не загружены,
    if(!Pathfinding.DataLoaded()){
//загрузить данные навигации
        Pathfinding.LoadNavigationData(filePath,fileName);
    }
}

/*код пользовательского интерфейса, позволяет загрузить
необходимый файл, указав путь к нему и его имя, уже после
загрузки сцены
*/
void OnGUI(){
    GUILayout.BeginVertical ();
    if(newFilePathDialog){
        GUILayout.BeginHorizontal ();
        GUILayout.Label("file path:");
        filePath =
        GUILayout.TextField(filePath,GUILayout.MaxWidth(250));
        GUILayout.EndHorizontal ();
        GUILayout.BeginHorizontal ();
        GUILayout.Label("file name:",GUILayout.MaxWidth(65));
        fileName =
        GUILayout.TextField(fileName,GUILayout.MaxWidth(100));
        GUILayout.EndHorizontal ();
        if(GUILayout.Button ("Load",GUILayout.MaxWidth(40))){
            Pathfinding.LoadNavigationData(filePath,fileName);
            newFilePathDialog = false;
        }
        }else{
            if(GUILayout.Button ("Load other file"))
                newFilePathDialog = true;
        }
    GUILayout.EndVertical ();
}

```



```
}  
}
```

Для того чтобы воспользоваться этим скриптом, необходимо, чтобы он был прикреплен к активному объекту в сцене, после чего, с помощью небольшого меню, можно будет загрузить необходимый файл.

Если вам нужно, чтобы навигационный файл автоматически загружался с помощью PathfindingManager в скомпилированном проекте, тогда следует сделать следующее:

- придумайте любое имя для папки в которой вы хотите хранить ваши навигационные файлы в скомпилированном проекте и пропишите это название в поле fileFolder компонента PathfindingManager(например, название папки - «NAPS\_Data»);

- пропишите название файла , который будет автоматически загружен, в поле fileName компонента PathfindingManager (например, «de\_dust») и установите флажок autoLoad;

- скомпилируйте проект;

- откройте папку скомпилированного проекта и создайте папку с указанным именем(«NAPS\_Data») в \*\_Data папке вашего проекта(пример: «D:\CompiledProject\CompiledProject\_Data\NAPS\_Data»);

- скопируйте в эту папку требуемый \*.nvf файл (пример: «D:\CompiledProject\CompiledProject\_Data\NAPS\_Data\de\_dust.nvf»);

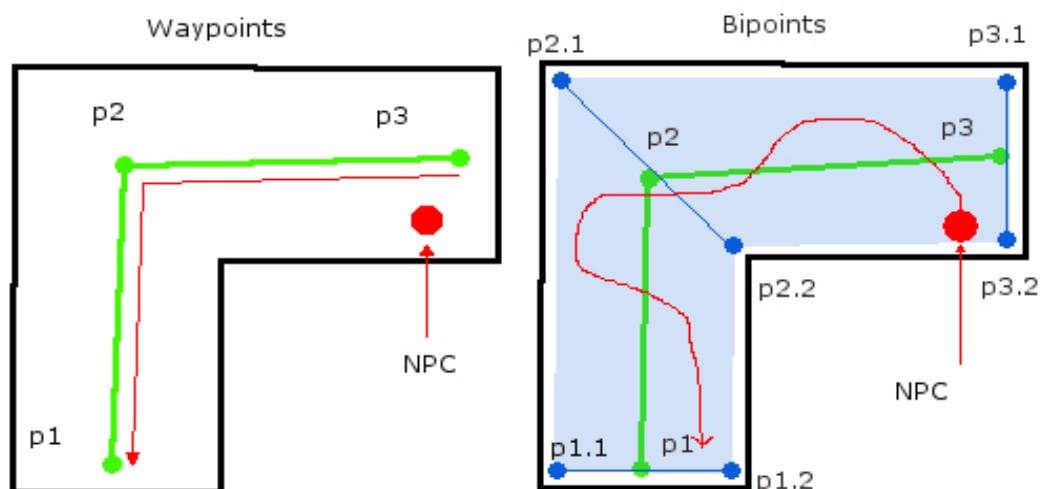
Теперь при запуске приложения файл de\_dust.nvf будет загружен автоматически.

### **Что такое бипоинты и как их использовать?**

Когда я разрабатывал систему навигации для своей игры, изначально максимально простым и эффективным было выбрано решение основанное на вейпоинтах. Но классические вейпоинты обладают также несколькими серьезными недостатками, главным из которых, для нас, было то, что, фактически, они описывают лишь путь в виде линии между двумя точками. В нашем случае нужна была вариативность возможных путей, чтобы игровые персонажи могли каждый раз строить уникальные пути пробегая по одним и тем же улицам, помещениям и т. д. Таким образом эту проблему в случае классических вейпоинтов можно решить добавлением новых точек, что означает увеличение количества расчетов и

в некоторых случаях пришлось бы добавить слишком большое количество точек для того чтобы обеспечить уникальные пути ботам. Так мне пришла идея модифицирования классических одиночных соединительных узлов - вейпоинтов в узлы, состоящие из 2-х точек, я назвал их бипоинтами. Смысл бипоинтов в том, что между парой точек описывается не линия, а целая площадь, которую может использовать бот для перемещения. При этом алгоритм расчета пути не усложняется и рассчитывает пару бипоинтов, как пару точек в пространстве, а количество хранимых данных практически остается таким же.

Рассмотрим пример:



На рисунках видим пример навигации для одного и того же помещения. Черной линией показаны границы помещения. На рисунке «Waypoints» видим путь проложенный внутри помещения, который состоит из 3-х точек p1, p2, p3. Траектория любого перемещения NPC по данному пути, несмотря на большое доступное пространство, будет сводиться к ломаной линии, которую представляют соединенные точки пути, как, например, траектория движения NPC из точки p3 в точку p1 (выделена красным цветом). Конечно, вполне логично будет применить сглаживание, например при повороте к следующей точке, но траектория фактически останется той же. Само собой, для реализации более менее реалистичного поведения бота, нужна большая свобода перемещения на местности.

Теперь рассмотрим пример навигации на той же карте, но с использованием бипоинтов (картинка «Bipoints»). В этом примере путь рассчитывается также по 3-м точкам, в матрице библиотеки поиска путей СНИПП (Pathfinding.cs) при этом также будет храниться информация только о 3-х точках, но в сцене каждая из точек p1, p2, p3 представлена 2-мя

точками( $p1 - p1.1;p1.2, p2 - p2.1;p2.2, p3 - p3.1;p3.2$ ), единственную информацию об этих точках которую нужно хранить - это их позиции в пространстве. И так, бипоинт - это узел пути, который описывает не точку в пространстве, а линию, но при расчете учитывается только центральная точка данной линии. При этом соединив пару бипоинтов, мы получаем площадь описанную 4-мя точками, а не 2-мя. Построив путь с помощью бипоинтов, вы получаете область перемещения ограниченную фигурой, которую описали бипоинты(синяя область). Таким образом NPC может проложить любую траекторию движения в пределах этой области. Приведенный пример траектории, показанный красной линией, в данном случае, подошел бы для существа типа зомби, или какого ни будь блуждающего животного и т. д. Но вы также можете построить максимально оптимизированный путь с учетом среза угла, насколько это возможно, для умных NPC, используя крайние точки доступной области , а также многие другие функции. Все функции для построения любых типов перемещения имеются в библиотеке поиска путей СНИПП(Pathfinding.cs). Информацию об этих функциях, а также их использовании смотрите в разделе «Функции библиотеки поиска путей для работы с ИИ» и приложенных примерах ИИ перемещения.

## Функции библиотеки поиска путей для работы с ИИ

LoadNavigationData – загружает определенный навигационный файл. В качестве параметров принимает путь к файлу и его имя, без учета расширения. Путь к файлу и его название должны иметь тип string. Количество перегрузок — 1.

*public static void LoadNavigationData(string filePath, string fileName)*

Пример(перегрузка №1):

```
public string fileFolder = "NAPS/NavigationData";  
public string fileName = "de_dust";  
  
void Awake(){  
    string filePath = Application.dataPath+"/"+fileFolder;  
    Pathfinding.LoadNavigationData(filePath,fileName);  
}
```

DataLoaded – проверяет были ли загружены навигационные данные, если да — возвращает true, в противном случае — false. Количество перегрузок — 1.

*public static bool DataLoaded()*

Пример(перегрузка №1):

```
void Awake(){  
    if(!Pathfinding.DataLoaded())  
        Debug.Log("Navigation data was not loaded!");  
}
```

FindPathOfType – принимает в качестве параметра название типа пути, типа string, возвращает индекс требуемого типа пути в массиве Pathfinding.Paths. Количество перегрузок — 1.

*public static int FindPathOfType(string type)*

Пример (перегрузка №1):

```
int pathIndex = FindPathOfType("air");  
Debug.Log("Size path of type air =" + Paths[pathIndex].size);
```

GetWaypointInSpace — возвращает точку в пространстве определенного одиночного узла или точку внутри бипойнта. Если узел одиночный, то параметр v игнорируется и результатом будет позиция узла в пространстве. Если узел является бипойнтом, то с помощью значения v можно получить любую точку внутри бипойнта, используя значения от 0 до 1, при этом v = 0 – первая точка бипойнта, v=1 – вторая точка бипойнта, v=0.5 – центр бипойнта. Количество перегрузок — 3.

*public static Vector3 GetWaypointInSpace(float v, int index, string pathType)*  
*public static Vector3 GetWaypointInSpace(float v, int index, int pathIndex)*  
*public static Vector3 GetWaypointInSpace(float v, Node node)*

Пример(перегрузка №1):

```
Node node = new Node(1, "air");
```

```
void Start(){
```

```

Vector3 bipointCenter =
GetWaypointInSpace(0.5f,node.number,node.type);
Debug.Log("Center of bipoint:"+bipointCenter);
}

```

Пример(перегрузка №2):

```

Node node = new Node(1, «air»);
int pathIndex;

```

```

void Start(){
pathIndex = FindPathOfType(node.type);
Vector3 bipointCenter =
GetWaypointInSpace(0.5f,node.number,pathIndex);
Debug.Log("Center of bipoint:"+bipointCenter);
}

```

Пример(перегрузка №3):

```

Node node = new Node(1, «air»);

```

```

void Start(){
Vector3 bipointCenter = GetWaypointInSpace(0.5f,node);
Debug.Log("Center of bipoint:"+bipointCenter);
}

```

GetPointTransform – возвращает родительский трансформ определенного узла. Количество перегрузок — 1.

```

public static Transform GetPointTransform(int nodeNumber, string type)

```

Пример(перегрузка №1):

```

Node node = new Node(1, «air»);

```

```

void Start(){
Transform nodeTransform =
GetPointTransform(node.number,node.type);
if(nodeTransform.childCount>0)
Debug.Log("Node is bipoint!");
else
Debug.Log("Node is single point!");
}

```

NodeInArray – проверяет находится ли искомый узел в массиве узлов, если да, то возвращает true, иначе — false. Количество перегрузок — 1.

```
public static bool NodeInArray(Node node, Node[] nodes)
```

Пример(перегрузка №1):

```
Node node = new Node(1, «air»);
```

```
Node[] array = new Node[2];
```

```
void Start(){  
array[0] = new Node(Random.Range(0,10), «air»);  
array[1] = node;  
if(NodeInArray(node,array))  
Debug.Log(“This node in array!”);  
}
```

LinkExist – проверяет существует ли связь между двумя узлами, если да, то возвращает true, иначе — false. Количество перегрузок — 1.

```
public static bool LinkExist(Node first, Node second)
```

Пример(перегрузка №1):

```
Node firstNode = new Node(1, «air»);
```

```
Node secondNode = new Node(2, «land»);
```

```
void Start(){  
if(LinkExist(firstNode,secondNode))  
Debug.Log(“I can land on the  
node :”+secondNode.number+secondNode.type);  
}
```

FindClosestPointTo – находит ближайший к текущей позиции видимый узел любого доступного для данного ИИ типа пути. В качестве параметров принимает массив доступных для данного ИИ типов путей и позицию бота/NPC в пространстве. Количество перегрузок — 2.

```
public static Node FindClosestPointTo(string[] types, Vector3 v3)  
public static Node FindClosestPointTo(string[] types, Vector3 v3, float radius,  
int layerMask)
```

Пример(перегрузка №1):

```
string[] availablePaths = new string[2];
Node closestNode;

void Start(){
    availablePaths[0] = "land";
    availablePaths[1] = "air";
    closestNode =
    FindClosestPointTo(availablePaths,transform.position);
    Debug.Log("Closest node to this
    position :"+closestNode.number+closestNode.type);
}
```

Пример(перегрузка №2):

```
string[] availablePaths = new string[2];
Node closestNode;
int ignorePlayerLayer;
float NPCradius;

void Start(){
    ignorePlayerLayer = LayerMask.NameToLayer("Player");
    ignorePlayerLayer = 1<<ignorePlayerLayer;
    ignorePlayerLayer = ~ ignorePlayerLayer;
    CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
    NPCradius = thisCollider.radius;
    availablePaths[0] = "land";
    availablePaths[1] = "air";
    closestNode =
    FindClosestPointTo(availablePaths,transform.position,NPCradius,
    ignorePlayerLayer);
    Debug.Log("Closest node to this
    position :"+closestNode.number+closestNode.type);
}
```

GetRouteForPoint – возвращает маршрут до определенной точки в пространстве.Количество перегрузок -1.

```
public static RouteData GetRouteForPoint(RouteData newRouteData,string[]
types,Vector3 destPoint,Vector3 plrPos,float plrRadius,int layerMsk)
```

Пример(перегрузка №1):

```
string[] availablePaths ;
RouteData routeData = new RouteData();
```

```

Vector3 destinationPoint;
int ignorePlayerLayer;
float NPCradius;

void Start(){
movement = GetComponent<NPCMovement>();
ignorePlayerLayer = LayerMask.NameToLayer("Player");
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
availablePaths = new string[Pathfinding.Paths.Length];
int i;
for(i=0;i<availablePaths.Length;i++){
availablePaths[i] = Pathfinding.Paths[i].type;
}
destinationPoint = GetRandomNavigationPoint(availablePaths);
if(destinationPoint !=Vector3.zero){
routeData =
GetRouteForPoint(routeData,availablePaths,destinationPoint,tran
sform.position,NPCradius,ignorePlayerLayer);
}

}

```

ResumeRoute – возобновляет маршрут в случае потери видимости текущего узла. Количество перегрузок -1.

```

public static RouteData ResumeRoute(RouteData newRouteData,string[]
types,Vector3 plrPos,float plrRadius,int layerMsk)

```

GetRandomNavigationPoint – возвращает случайную точку в пространстве, сгенерированную внутри из одно из доступных типов путей. Количество перегрузок -1.

```

public static Vector3 GetRandomNavigationPoint(string[] types)

```

Пример(перегрузка №1):

```

string[] availablePaths ;
Vector3 randomPoint;

void Start(){
availablePaths = new string[Pathfinding.Paths.Length];

```



```

int i;
for(i=0;i<availablePaths.Length;i++){
availablePaths[i] = Pathfinding.Paths[i].type;
}
randomPoint = GetRandomNavigationPoint(availablePaths);
Debug.Log("random navigation point:"+randomPoint);
}

```

GetRandomPathsNode – возвращает случайный узел одного из доступных типов путей. В качестве параметра принимает массив доступных типов путей. Количество перегрузок — 1.

```

public static Node GetRandomPathsNode(string[] types)

```

Пример(перегрузка №1):

```

string[] availablePaths ;
Node randomNode;

void Start(){
availablePaths = new string[Pathfinding.Paths.Length];
int i;
for(i=0;i<availablePaths.Length;i++){
availablePaths[i] = Pathfinding.Paths[i].type;
}
randomNode = GetRandomPathsNode(availablePaths);
Debug.Log("New random
node :"+closestNode.number+closestNode.type);
}

```

NodeIsEmpty – проверяет существует ли определенный узел, если нет , то возвращает true, иначе — false. Количество перегрузок — 1.

```

public static bool NodeIsEmpty(Node node)

```

Пример(перегрузка №1):

```

Node node ;

void Start(){
node = new Node(-1, "");
//return true,because node is empty
//Debug will print : "Node exist : false");
Debug.Log("Node exist:"+!NodeIsEmpty(node));
}

```

NodeIsBipoint – проверяет является ли узел бипоинтом, если да , то возвращает true, иначе — false. Количество перегрузок — 1.

*public static bool NodeIsBipoint(Node node)*

Пример(перегрузка №1):

```
Node node ;
string[] availablePaths = {"path1","path2"};

void Start(){
node = GetRandomPathsNode(availablePaths);
if(NodeIsBipoint(node))
Debug.Log("node "+node.number+node.type+" is bipoint");
}
```

NodesEqual – проверяет равны ли узлы, то есть если номера и типы узлов совпадают, то возвращает true, в противном случае — false. Количество перегрузок — 1.

*public static bool NodesEqual(Node a, Node b)*

Пример(перегрузка №1):

```
Node aNode = new Node (5, "human");
Node bNode = new Node (6, "human");
bool result;
```

```
void Start(){
result = NodesEqual(aNode,bNode);
//will print "false"
Debug.Log(result);
aNode = bNode;
//will print "true"
Debug.Log(result);
}
```

GetCurve – возвращает кривую между двумя узлами, в качестве параметров принимает первый и второй узел типа Node. Количество перегрузок — 1.

*public static CurveData GetCurve(Node first, Node second)*

Пример(перегрузка №1):

```
Node firstNode = new Node (2, "air");
Node secondNode = new Node (6, "land");
CurveData curveData;

void Start(){
if(CurveExistBetween(firstNode,secondNode))
curveData = GetCurve(firstNode,secondNode);
}
```

CurveExistBetween – проверяет существует ли кривая между двумя узлами, если да, то возвращает true, иначе — false. Количество перегрузок -1.

*public static bool CurveExistBetween(Node first, Node second)*

Пример(перегрузка №1):

```
Node firstNode = new Node (2, "air");
Node secondNode = new Node (6, "land");
CurveData curveData;

void Start(){
if(CurveExistBetween(firstNode,secondNode))
curveData = GetCurve(firstNode,secondNode);
}
```

GetCurveLength – возвращает длину определенной кривой. Для получения длины кривой используется разбиение ее на отрезки, чем больше количество отрезков, тем больше времени тратиться на вычисление длины. Количество перегрузок — 1.

*public static float GetCurveLength(int segments, CurveData curve)*

Пример(перегрузка №1):

```
Node firstNode = new Node (2, "air");
Node secondNode = new Node (6, "land");
CurveData curveData;
float curveLength = 0f;

void Start(){
if(CurveExistBetween(firstNode,secondNode)){
```

```

curveData = GetCurve(firstNode,secondNode);
    if(curveData !=null)
curveLength = GetCurveLength(100,curveData);
Debug.Log("Curve length = "+curveLength);
}
}

```

GetBipointLength – возвращает расстояние между точками бипointа. Количество перегрузок — 1.

```

public static float GetBipointLength(Node node)

```

Пример(перегрузка №1):

```

Node node = new Node (1, "path1");

float bipointLength;

void Start(){
if(NodeIsBipoint(node)){
    bipointLength = GetBipointLength(node);
    Debug.Log("node "+node.number+node.type+" is
bipoint,length:"+bipointLength);
}
}

```

MoveByCurve – используется для движения по определенной кривой. Работает с классом MotionCurve. Для более подробной информации смотрите описание класса MotionCurve. Количество перегрузок -1.

```

public static MotionCurve MoveByCurve(MotionCurve motionCurve, Vector3
curPos, float increment)

```

Для более подробной информации смотрите пример движения по кривой — FollowCurve.

GetPointOnBezier – возвращает точку на кривой в зависимости от передаваемого значения(0 — исходная точка, 1- конечная точка кривой, 0-1 — промежуточные точки кривой). Количество перегрузок — 1.

```

public static Vector3 GetPointOnBezier(float t, CurveData newCurve)

```

Пример(перегрузка №1):

```
Node first = new Node (0, "land");
Node second = new Node(1, "land");
CurveData curveData;
Vector3 middleCurvePoint;

void Start(){
if(CurveExistBetween(first,second)){
curveData = GetCurve(first,second);
if(curveData != null){
//Get middle point of curve
middleCurvePoint = GetPointOnBezier(0.5f,curveData);
}
}
}
```

DistanceToNode - возвращает минимальное расстояние от точки в пространстве до определенного узла. Количество перегрузок — 1.

```
public static float DistanceToNode(Node thisNode, Vector3 curPos)
```

Пример(перегрузка №1):

```
Node node = new Node (0, "land");
float distance = 0f;

void Update(){
distance = DistanceToNode(node, transform.position);
Debug.Log("distance to node:"+distance);
}
```

VisibilityOfNode – определяет видимость определенного узла из конкретной точки . Возвращаемые значения: - 1 — если узел не виден; 0 — если узел виден и является одиночной точкой ; 1 — если узел является бипоинтом и видна только одна из его точек; 2 — если узел является бипоинтом и видны обе его точки. Количество перегрузок -3.

```
public static int VisibilityOfNode(Node node, Vector3 curPos)
public static int VisibilityOfNode(int nodeNumber, int nodePathIndex, Vector3
curPos,int layerMsk)
public static int VisibilityOfNode(Node node, Vector3 curPos,float
playerRadius,int layerMsk)
```

Пример(перегрузка №1):

```
int pointVisibility;  
Node node = new Node(1, "land");  
Vector3 nextPoint = Vector3.zero;  
  
void Update(){  
pointVisibility = VisibilityOfNode(node,transform.position);  
if(pointVisibility<0){  
Debug.Log("Node is not visible!");  
}else if(pointVisibility<1){  
nextPoint = GetWaypointInSpace(0.5f,node);  
}else{  
nextPoint = GetWaypointInSpace(Random.value,node);  
}  
MoveTo(nextPoint);  
}
```

ObjectIsVisible – проверяет видимость точки или объекта, возвращает true если точка (или объект) видима, в противном случае — false. Количество перегрузок -4.

```
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos, float radius)  
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos, float radius,  
int layerMsk)  
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos,Vector3  
lookDir,float FOV)  
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos,Vector3  
lookDir,float FOV,int layerMsk)
```

Пример(перегрузка №1):

```
float NPCradius;  
public Transform object;  
  
void Start(){  
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();  
NPCradius = thisCollider.radius;  
if(ObjectIsVisible(object.position,transform.position,NPCradius  
)  
Debug.Log("object "+object.name+" is visible");  
}
```

Пример(перегрузка №2):

```

float NPCradius;
public Transform object;
int ignorePlayerLayer;

void Start(){
ignorePlayerLayer = gameObject.layer;
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
if(ObjectIsVisible(object.position,transform.position,NPCradius
,ignorePlayerLayer)
Debug.Log("object "+object.name+" is visible");
}

```

Пример(перегрузка №4):

```

public Transform object;
int ignorePlayerLayer;
float fieldOfView = 54f;

void Start(){
ignorePlayerLayer = gameObject.layer;
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
if(ObjectIsVisible(object.position,transform.position,transform
.forward,fieldOfView,ignorePlayerLayer)
Debug.Log("object "+object.name+" is in my field of view!");
}

```

GetRoute – возвращает глобальный маршрут между двумя узлами(возвращает последовательность переходных точек между отдельными подсетями путей), в виде массива узлов типа Node. Количество перегрузок -1.

```

public static Node[] GetRoute(string[] types,Node startPoint,Node
endPoint):Node[]

```

Пример(перегрузка №1):

```

Node[] globalRoute ;
Node startNode = new Node(1, «type1»);
Node endNode = new Node(27, «type3»);

```

```
string[] allowedPaths = {"type1", "type2", "type3", "type4"};

void Start(){
globalRoute = GetRoute(allowedPaths, startNode, endNode);
}
```

Более подробную информацию об использовании данной функции смотрите в приложенных примерах ИИ.

GetPointInArea – возвращает случайную точку в пространстве, внутри области, описанной двумя узлами. В зависимости от параметра `range`, можно получить случайную точку во всей области, или же отсечь часть области, в которой не будет генерироваться точка. Обычно это используется для генерации случайной точки, внутри области, для NPC, который движется по этой области. Таким образом генерируемая точка всегда будет находиться впереди NPC, что обеспечивает ему движение к определенному узлу данной области. В этом случае с помощью параметра `range` можно расширять или сужать область генерирования точки впереди NPC. Количество перегрузок — 1.

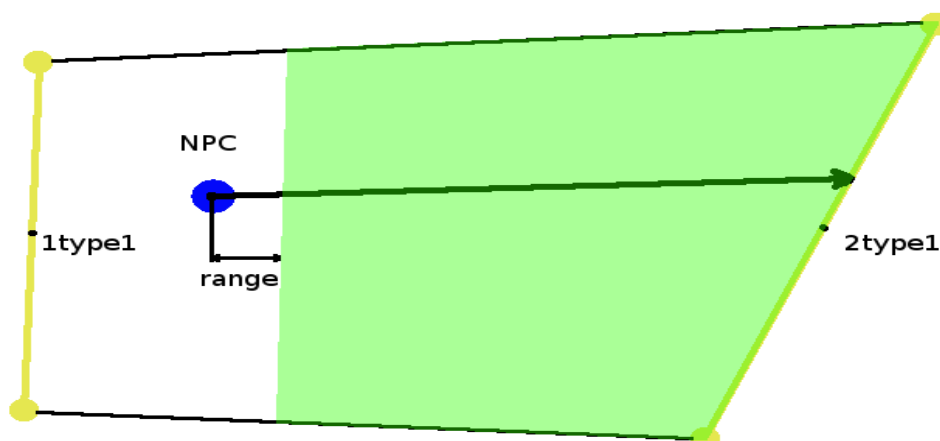
```
public static Vector3 GetPointInArea (Node first, Node second, Vector3
curPos,float range)
```

Пример(перегрузка№1):

```
Node startNode = new Node(0, "type1");
Node endNode = new Node(1, "type1");
Vector3 targetPoint;
```

```
void Update(){
targetPoint =
GetPointInArea(startNode,endNode,transform.position,1f);
//MoveTo(targetPoint);
}
```

В данном примере предполагается, что NPC движется от узла 1 `type1` к узлу 2 `type1`, `transform.position` – позиция NPC в области описанной этими узлами. Область для генерирования точки (показана зеленым цветом), впереди NPC, немного сократили (приблизительно на 1 м) с помощью параметра `range`.





В случае, когда нужно сгенерировать случайную точку внутри всей области, описанной двумя узлами, параметру `range` следует задать отрицательное значение, например -1.

`GetOptimizedTrajectoryPoint` – возвращает значение точки внутри узла, в виде дробного значения. Эта точка является оптимальной для движения к следующему узлу через текущий узел, оптимальность выбора точки зависит от передаваемых параметров. Для того чтобы преобразовать полученное значение в точку в пространстве, следует воспользоваться функцией `GetWaypointInSpace`. Количество перегрузок — 1.

```
public static float GetOptimizedTrajectoryPoint(Node curPoint, Node  
nextPoint, float lastValue, float pathOptimization, Vector3 curPos, float  
defaultAmplitude, float minDistance, float maxAmplitude):float
```

Пример(перегрузка №1):

```
Node curPoint = new Node(1, "type1");  
Node nextPoint = new Node(2, "type1");  
float pathOptimization = 2.3f;  
float targetPointValue = -1f;  
float criticalDistance = 7f;  
Vector3 targetPoint;  
  
void Update(){  
targetPointValue = GetOptimizedTrajectoryPoint(curPoint,  
nextPoint, targetPointValue, pathOptimization,  
transform.position, Random.Range(0.5f,2f), 7f, 0.2f);  
targetPoint = GetWaypointInSpace(targetPointValue,  
curPoint.number, curPoint.type);  
//MoveTo(targetPoint);  
}
```

На рисунке ниже показано, как работает данный код. В данном примере NPC движется к узлу `nextPoint` через узел `curPoint` (текущий узел). Узлы `curPoint` и `nextPoint` являются бипoints, то есть состоят из 2-х точек в пространстве, которые отмечены номерами 1 и 2 у каждого узла. Для того, чтобы выбрать оптимальную точку внутри узла `curPoint` для движения в узел `nextPoint`, функция определяет через какую точку бипoints `curPoint` следует двигаться, чтобы максимально сократить длину пути. Как видно на рисунке, путь через точку 1 (показан зеленым цветом) будет более

The diagram illustrates the path optimization algorithm. An NPC (blue dot) is shown moving towards a target point (green dot) on a yellow path. The path is divided into segments by points labeled 1 and 2. A green line connects the NPC to the optimal point (point 1). A red line connects the NPC to the next point (point 2). A black line connects the NPC to the target point. A gray rectangle indicates the critical distance for path optimization.

*defaultAmplitude* – этот параметр определяет амплитуду вариации реальной точки в пределах параметра *pathOptimization*, если дистанция до реальной точки больше значения параметра *criticalDistance*;

*criticalDistance* – параметр, который определяет на каком расстоянии, от реальной точки, амплитуда варьирования реальной точки уменьшается;

*maxAmplitude* – параметр, который является амплитудой варьирования реальной точки внутри бипоинта, при достижении параметра *criticalDistance*.

В данном примере NPC имеет параметр оптимизации пути *pathOptimization*, который равен 2.3, это значит, что после вычисления оптимальной точки для движения через бипоинт, реальная точка будет колебаться в пределах 2.3м от оптимальной точки. Амплитуда вариации относительно последней реальной точки на расстоянии более 7 метров от этой точки, будет составлять от 0.5м до 2м, а после достижения критической дистанции(7м и меньше) амплитуда составит 0.2 м от последней реальной точки в пределах параметра оптимизации.

Таким образом, используя данную функцию можно варьировать движение NPC, сделать его более оптимизированным или же наоборот — сделать траекторию движения непредсказуемой.

## Классы библиотеки поиска путей для работы с ИИ

### Класс Node

Для работы с навигацией на вашей карте вам постоянно придется обращаться к определенным узлам подсетей путей, искать маршруты между ними и многое другое. Для этих целей служит класс *Node(Pathfinding.Node)*.

*Node* – наиболее используемый класс при работе с навигацией, он используется для представления узлов.

Переменные:

`public int number` – используется для хранения номера узла;

`public string type` – используется для хранения типа узла(пути);

Пример использования:

Допустим, что нам нужно получить положение узла с номером 2 и типом «road1», в пространстве, если узел является бипоинтом, то получить его среднюю точку. Сделать это можно следующим образом:

*/\*объявляем новый экземпляр узла, где -1 – начальный номер, а «» - строковое представление типа узла, тип узла – это переменная типа string, вы можете придумать любое название типа при создании путей в редакторе. Цифра -1 и пустые*

*кавычки, вместо названия типа показывают, что этот узел пока что пуст.*

```
*/  
Pathfinding.Node node = new Node(-1, "");  
  
void Start(){  
/* теперь нужно задать пустому узлу требуемые значения, это  
можно сделать так:  
*/  
node.number = 1; /*так как нумерация ведется от 0, то реальное  
значение узла в коде будет равно отображаемому в редакторе  
значению -1, то есть 2-1 = 1.  
*/  
node.type = "road1"; //задаем этому узлу тип  
/*объявляем переменную типа Vector3 и с помощью функции  
GetWaypointInSpace получаем требуемую позицию внутри данного  
узла;  
*/  
Vector3 nodePosition =  
Pathfinding.GetWaypointInSpace(0.5f,node.number,node.type);  
}
```

## Класс RouteData

Для работы с маршрутами используется класс RouteData. Если вы хотите, чтобы ваш ИИ имел возможность использовать пути для движения, то вам нужно будет создать для него экземпляр класса RouteData. Навигация в СНИПП представляет собой сетку, состоящую из узлов. Соединения между различными узлами представляют собой пути. Пути могут быть однотипными или же атипичными. Однотипный путь — это путь который состоит из узлов одного типа. Например, такой путь будет однотипным : 0 “human” - 3 “human” - 6 “human”. Атипичный путь — это путь проходящий через узлы с разными типами. Пример атипичного пути: 0 “human” - 2 “dragon” - 6 “human”. Внутри общей сетки путей, узлы одного типа образуют свою подсеть. Такая подсеть будет предварительно просчитана и если ИИ нужно двигаться только в пределах этой подсети, то ему не придется просчитывать путь для себя вообще, так как он просто будет получать доступ к уже рассчитанной подсети. В случае, когда ИИ должен уметь перемещаться сразу через несколько подсетей, образованных узлами разных типов, то ему придется просчитать оптимальный путь между различными подсетями путей, но используя уже предрасчитанные данные каждой подсети. Такой подход позволяет во много раз сократить сложность расчета маршрута. И так, класс RouteData по сути является

хранилищем найденного маршрута, с той лишь разницей, что он учитывает предрасчитанные данные сетки путей, а также некоторые вспомогательные данные. Благодаря этому каждый экземпляр ИИ хранит не весь маршрут, то есть каждую точку пути, через которые он будет двигаться, а только переходные точки между предрасчитанными участками сетки путей, а промежуточные точки в этих участках получаются динамически из рассчитанных подсетей. В большинстве случаев вам не придется помнить все эти тонкости, достаточно изучить примеры и понять, как использовать данный класс.

Переменные:

`public Node[] route` – хранит переходные точки маршрута, проще говоря — глобальный маршрут;  
`public Node curPoint` – хранит текущий узел в маршруте, к которому нужно двигаться;  
`public Node nextPoint` – хранит следующий после текущего узел в маршруте;  
`public Node destinationPoint` – хранит конечный узел данного маршрута;  
`public int nextPointIndex` – хранит индекс переходной точки для типа подсети путей, по которой в данный момент движется NPC. Эта переменная используется только для правильного расчета пути, обычно нету никакой необходимости обращаться к ней.

Для большего понимания смотрите пример использования класса `RouteData` на примере приложенного ИИ следования по путям `PathFollowingAI.cs`.

## Класс `MotionCurve`

В случае, когда нужно организовать движение какого либо объекта по кривой, нужно использовать класс `MotionCurve`. Этот класс используется для хранения данных о движении по текущей кривой.

Переменные:

`public CurveData curve` – текущая кривая, по которой происходит движение;  
`public float curveLength` – длина текущей кривой;  
`public float passedDist` – пройденное расстояние по кривой;

`public Vector3 lastPoint` – хранит последнюю позицию объекта, движущегося по данной кривой(нужно для правильного вычисления пройденного расстояния по кривой);

`public Vector3 newPoint` – хранит точку в пространстве к которой нужно двигаться в данный момент, чтобы следовать кривой

Практически все переменные указанные здесь используются функциями перемещения по кривой и не представляют интереса, если только вы не хотите углубиться в сам процесс перемещения по кривой. Для того, чтобы просто организовать движение по кривой, не вдаваясь в подробности, достаточно использовать переменную `newPoint`, которая всегда содержит точку, к которой нужно двигаться, чтобы повторять кривую. Для того, чтобы понять как именно использовать класс `MotionCurve` рассмотрите пример движения по кривой `FollowCurve.cs`.

### Класс `CurveData`

Данный класс используется для хранения и записи в файл данных кривых Безье. Так как в СНИПП используются кривые Безье 3-го порядка, то они представляются 4-мя точками в пространстве, где первая и последняя точки — это соответственно начальная и конечная точки кривой, а две промежуточные точки — это контрольные точки, которые задают форму кривой.

Переменные:

`public Vector3S[] tangents` – массив, представляющий точки кривой(`tangents[0]` – начальная точка кривой;`tangents[1]` – первая контрольная точка ;`tangents[2]` – вторая контрольная точка; `tangents[3]` – конечная точка кривой).

Так как класс `CurveData` используется не только для временного хранения данных, а еще и для записи в файл, то вместо привычного `Vector3` используется собственный класс — `Vector3S`. Этот класс имеет всего 2 функции, которые помогают либо сохранить данные в экземпляр класса, либо загрузить их из него:

`public Vector3S(UnityEngine.Vector3 v3)` – используется для сохранения переменной типа `Vector3` в экземпляре класса `Vector3S` при его объявлении;

`public UnityEngine.Vector3 GetVector3()` - возвращает ранее сохраненный вектор.

Пример: Зададим кривую Безье 3-го порядка с помощью класса `CurveData` и вспомогательного класса `Vector3S`.

```
//создаем экземпляр класса CurveData для хранения новой кривой  
CurveData newCurve = new CurveData();
```

```

void Start(){
//задаем массиву точек размер = 4
newCurve.tangents = new Pathfinding.Vector3S[4];
//задаем первую точку кривой
newCurve.tangents[0] = new Vector3S(new Vector3(0,0,0));
//задаем вторую точку кривой
newCurve.tangents[1] = new Vector3S(new Vector3(5,4,2));
//задаем третью точку кривой
newCurve.tangents[2] = new Vector3S(new Vector3(7,6,8));
//задаем последнюю точку кривой
newCurve.tangents[3] = new Vector3S(new Vector3(10,0,10));

//теперь получим сохраненные точки обратно, в виде Vector3
Debug.Log("first point of
curve:"+newCurve.tangents[0].GetVector3());
Debug.Log("control point
№1:"+newCurve.tangents[1].GetVector3());
Debug.Log("control point
№2:"+newCurve.tangents[2].GetVector3());
Debug.Log("end point of
curve:"+newCurve.tangents[3].GetVector3());
}

```

В большинстве случаев вам не придется погружаться в тонкости использования переменных данного класса, потому что за вас это сделают функции для работы с кривыми.

## Класс Path

Данный класс используется для хранения данных подсети путей. Большую часть переменных этого класса вы не будете использовать напрямую, так как при работе с навигацией в этом нет необходимости.

Переменные:

```

public string type – тип данной подсети путей;
public float[,] connectionLength – матрица соединений узлов данной подсети;
public float[,] pathLength – матрица длин путей между всеми узлами данной подсети;
public int[,] nextPoint – матрица маршрутов между всеми узлами данной подсети;
public int size — количество узлов в данной подсети;

```

`public Waypoint[] Waypoints` -массив экземпляров класса `Waypoint`, который содержит трансформы всех узлов данной подсети, а также все атипичные соединения каждого узла подсети.

При использовании функции библиотеки `Pathfinding.cs` практически нет необходимости обращаться к переменным данного класса. В большинстве случаев пригодиться могут лишь переменные `size` и `type`.

## Основные компоненты СНИПП

### Библиотека методов и классов СНИПП - `Pathfinding`

Библиотека `Pathfinding.cs` является основой СНИПП, она хранит загруженные навигационные данные, а также необходимые методы и классы для работы с ними. В ней также содержатся методы и классы, упрощающие создание игрового искусственного интеллекта. Обращение к этим методам и классам может осуществляться через класс `Pathfinding` (например: `Pathfinding.Node`, `Pathfinding.RouteData`, `Pathfinding.NodeIsBipoint()`).

### Компонент `Navigator`

В СНИПП версии 1.3 вся базовая работа с навигацией перенесена в компонент `Navigator`. Это сделано для повышения универсализации и упрощения создания ИИ. Теперь все что нужно сделать для перемещения персонажа - это назначить для него компоненты `NPCMovement`, `Navigator.cs` и передать конечную точку в пространстве в компонент `Navigator`. Всю работу по обновлению маршрута и другие вычисления `Navigator` выполняет самостоятельно, после чего передает требуемую точку для движения в `NPCMovement`, непосредственно для перемещения персонажа. `Navigator` имеет набор параметров, которые помогают регулировать перемещение персонажа.

Параметры:

**`destinationPoint`** – конечная точка, которую требуется достичь;

**`distanceToDestinationPoint`** – расстояние по прямой до конечной точки (только для чтения);

**`destinationPointIsVisible`** – содержит `true`, если точка находится в прямой видимости, в противном случае — `false` (только для чтения).

**`distToCurPoint`** – расстояние до текущей точки в пространстве (только для чтения);



**Auto Ramble** – отвечает за генерацию случайных маршрутов для NPC. Если флаг установлен, то NPC будет генерировать для себя случайные конечные точки в пределах заданных навигационных опций, в противном случае генерация не происходит и NPC будет бездействовать, если не указывать требуемую конечную точку через `destinationPoint`.

**Move To Destination If Not Zero** – определяет, может ли NPC двигаться к конечной точке, если она не равна `Vector3.zero`. Если флаг установлен, то NPC начнет движение к конечной точке, в случае если она не равна `Vector3.zero`, а после достижения обнулит ее. Передать конечную точку в навигатор можно через переменную `Navigator.destinationPoint`.

**Use Visibility** - отвечает за проверку видимости узлов пути, для того, чтобы рациональнее выстраивать путь. Если флаг установлен, то NPC будет переходить к следующему узлу в маршруте, как только его увидит, в противном случае NPC будет достигать требуемую дистанцию до текущего узла и только после этого переходить к следующему.

**Use Destination Point Visibility** – определяет, может ли NPC считать достигнутой конечную точку в пространстве в случае, если он ее видит. Если флаг установлен, то может, в противном случае NPC должен будет достичь определенную дистанцию до этой точки, чтобы считать ее достигнутой.

**Use Node Distance** – определяет, каким образом измерять дистанцию до текущего узла. Если флаг установлен, то дистанция между NPC и текущим узлом измеряется как минимальное расстояние между позицией NPC и непосредственно текущим узлом пути, независимо от того к какой именно точке в пространстве внутри узла, NPC движется. В противном случае, если флаг не установлен, то расстояние измеряется между позицией NPC в пространстве и непосредственно точкой в пространстве внутри узла, к которой он движется.

*Примечание: для использования с большим количеством экземпляров NPC, последний метод работает быстрее.*

**Point Reach Radius** – дистанция, которую необходимо достичь NPC, для того, чтобы считать узел пути или конечную точку достигнутой, в случае если NPC не использует видимость для этой цели.

**Path Optimization** – расстояние в метрах, указывающее максимальное возможное отклонение от оптимальной точки внутри узла, при движении к следующей точке маршрута.

**Trajectory Update Interval** – интервал обновления точки внутри текущего узла, к которому движется NPC, указывается в секундах.

*Примечание: Если вам не нужна динамическая вариативность траектории движения NPC, то можете установить значение равным Mathf.Infinity.*

**Types** – массив доступных для использования для данного NPC типов путей. Имена должны иметь тип string.

**Modify target point coordinates** - определяет какие именно координаты для движения может получать данный NPC. Например, если не нужно учитывать высоту узла, то можно снять галочку с координаты «у». Таким образом NPC будет получать только 2D координаты, т.е. «x» и «z» от текущего узла.

Если вам нужно, чтобы NPC не генерировал случайные конечные точки для себя, а следовал к указанной точке в пространстве, то вам следует сделать следующее:

- убедиться, что для NPC назначен компонент NPCMovement;
- убедиться, что для NPC назначен компонент Navigator;
- снять флажок Auto Ramble компонента Navigator;
- установить флажок Move To Destination If Not Zero;
- передать конечную точку в виде Vector3 в Navigator.destinationPoint;

Для того, чтобы передавать конечную точку в Navigator удобно создать отдельный скрипт, который будет заниматься обновлением этой точки. Рассмотрим пример скрипта, который передает в Navigator позицию какого либо объекта, как конечную точку. Привяжите этот скрипт к экземпляру NPC, затем в его поле назначьте необходимый объект за которым будет двигаться персонаж.

Код скрипта:

```
using UnityEngine;
using System.Collections;

public class FollowObject : MonoBehaviour {
    //объект к которому нужно двигаться
    public Transform targetObject;
```

```

//расстояние между NPC и объектом при котором объект будет задавать
свою позицию как конечную точку
    public float distance = 1f;
    Navigator navigator;

    // Use this for initialization
    void Start () {
        navigator = GetComponent<Navigator>();
        if(navigator){
            //запрещаем генерацию случайного маршрута
            navigator.autoRamble = false;
//разрешаем движение к конечной точке, если она не равна
Vector3.zero
            navigator.moveToDestinationIfNotZero = true;
        }
    }

    // Update is called once per frame
    void Update () {
        if(navigator && targetObject){
if(Vector3.Distance(transform.position,targetObject.position)>
distance)
            navigator.destinationPoint = targetObject.position;
        }
    }
}

```

В данном примере конечная точка для следования указывается через переменную destinationPoint компонента Navigator, при этом установлен флаг moveToDestinationIfNotZero , что заставляет NPC двигаться к данной конечной точке. Но заставить NPC двигаться к требуемой точке в пространстве можно вызывая функцию MoveToDestinatination(Vector3 destPos) компонента Navigator из функции Update() любого активного скрипта, но для этого следует установить флаги autoRamble и moveToDestinationIfNotZero равными false.

Пример:

```

void Update(){
Navigator navigator = GetComponent<Navigator>();
if(navigator)
navigator.MoveToDestination(new Vector3(0,20,10));
}

```

Разница между этими методами лишь в том, что первый метод автоматически обнуляет достигнутую конечную точку, в то время как прямой вызов MoveToDestination лишь обновляет данные о видимости

точки и расстоянии до нее. Это может быть полезно, если вам не нужно автоматически обнулять конечную точку после ее достижения и провести какие либо манипуляции с ее координатами или что-то подобное.

***Примечание:** следует помнить, что передаваемая конечная точка всегда должна быть доступна(видима) для какого либо узла любого из разрешенных для NPC типов путей.*

В этой версии СНИПП используются проверки видимости объектов/узлов с учетом слоев, на которых эти объекты должны находиться. Для этой цели используются два слоя: первый — для объектов/NPC которые используют навигацию, объекты на этом слое не препятствуют видимости друг друга; второй слой — для других динамических объектов, которые не должны препятствовать видимости узлов, но должны препятствовать видимости объектов на первом слое. Например, «Player» - слой, на котором находятся все игроки. «Dynamic Object» - слой, на котором находятся другие динамические объекты. Предположим, у нас есть игровой уровень на котором находятся несколько зданий, каждое здание имеет как минимум одну дверь, которая обеспечивает вход в здание. Такая дверь управляется скриптом и может по команде скрипта либо открыться, либо закрыться. При этом нам нужно, чтобы NPC могли использовать двери для входа или выхода из здания. В таком случае следует поместить все двери зданий на слой «Dynamic Object», тогда дверь не будет препятствовать видимости узлов, внутри здания и NPC будет следовать к нужному узлу внутри здания так, как будто двери не существует, но когда подойдет на достаточное расстояние и определит препятствие как «дверь», то сможет применить команду для ее открытия. В тоже время NPC находящиеся на слое «Player» внутри помещения, не будут видимы для данного NPC до тех пор, пока он не откроет дверь. Размещение всех NPC на слое «Player» может быть полезным, когда нужно проверить видимость конкретного NPC игнорируя при этом других NPC на этом слое или для игнорирования столкновений между NPC.

Таким образом для использования компонента Navigator нужно выделить два слоя, для NPC и других динамических объектов соответственно и указать их номера в поля Player layer и Object layer компонента NPCMovement каждого NPC. Если вы не используете динамические объекты в вашей игре, то достаточно выделить один слой только для объектов использующих навигацию и в поле Object layer указать тот же номер, что и в поле Player layer.

***Примечание:** Следует помнить, что функции проверки видимости используют рейкастинг для каждого экземпляра NPC, поэтому при симуляции с большим количеством NPC производительность*

понижается. То же самое касается и функций проверки препятствий и определения заземленности NPC компонента NPCMovement. Поэтому для повышения производительности при симуляциях с большим количеством NPC можно отключать эти функции (`Navigator.useVisibility = false`, `Navigator.useDestinationPointVisibility = false`, `Navigator.useNodeDistance = false`, `NPCMovement.moveOnlyIfGrounded = false`, `NPCMovement.checkObstaclesInForward = false`).

## Компонент NPCMovement

Компонент NPCMovement отвечает за перемещение объекта непосредственно к указанной точке с учетом ускорения, вращения и т. д. Этот компонент может использоваться для перемещения любых объектов, но ориентирован в первую очередь на перемещение NPC.

Параметры компонента NPCMovement:

**targetPoint** – точка в пространстве, к которой нужно двигаться;

**lookAtPoint** – точка в пространстве, которая используется для вращения оси forward, данного объекта, в ее направлении.

**grounded** – показывает, находится ли объект на поверхности, в случае если флаг Move Only If Grounded не установлен, то всегда возвращает true;

**Max Speed** – максимальная скорость с которой может двигаться NPC/объект;

**Acceleration Time** - время в секундах , за которое NPC/объект может разогнаться до максимальной скорости;

**Deceleration Time** – время в секундах, за которое текущая скорость NPC/объекта может сократиться до 0.

**Movement Smooth** – коэффициент плавности изменения направления движения, при смене направления движения в сторону новой точки.

**Movement Smooth Distance** – расстояние от реальной точки, к которой в данный момент движется объект, на котором происходит интерполяция вектора движения. Этот параметр помогает предотвратить «проскакивание» реальной точки для движения, в случае движения

NPC/объекта на большой скорости и, благодаря запасу расстояния от реальной точки, корректно интерполирует вектор движения.

**Rotation Speed** – скорость вращения объекта в определенном направлении, которое зависит от параметров указанных ниже.

**Auto Rotation Speed** – может ли объект изменять скорость вращения вместе с изменением вектора движения, если флаг установлен — может, в противном случае — не может.

**Rotate Y Only** – определяет вращать ли объект только вокруг вертикальной оси.

**Rotate To Look Dir** – определяет можно ли вращать данный объект в направлении точки lookAtPoint. То есть, если вам нужно чтобы объект во время движения поворачивался осью forward в направлении определенной точки, то вам нужно передать требуемую точку в NPCMovement.lookAtPoint, затем установить флаг Rotate To Look Dir. В противном случае объект будет поворачиваться осью forward в сторону движения.

**Can Move** – если флаг установлен, то объект может перемещаться, в противном случае движение объекта запрещено.

**Can Rotate** – если флаг установлен, то объект может вращаться в нужном направлении, в противном случае NPCMovement не будет вращать объект.

**Use Slerp** – если флаг установлен, то вектор движения будет интерполироваться сферически, в противном случае — линейно.

**Move Only If Grounded** – если флаг установлен, то объект сможет двигаться к требуемой точке, только если он находится на какой либо поверхности, в противном случае объект не сможет двигаться.

**Check Obstacles In Forward** – если флаг установлен, то объект/NPC будет осуществлять проверку на наличие препятствия в направлении его движения и если препятствие находится прямо перед объектом, то перемещение объекта будет запрещено.

**Player Layer** – номер слоя, на котором должны находиться все NPC/объекты которые используют навигацию для перемещения.

**Object Layer** – слой на котором должны находиться динамические препятствия, которые игнорируются при проверке видимости узлов, но при этом они должны влиять на видимость других динамических объектов/NPC.

### **Контактная информация**

Если у вас возникли проблемы, которые не удастся решить с помощью данного руководства, у вас есть информация об ошибках в работе или у вас просто есть вопрос, связанный с СНИПП, то вы можете задать интересующий вас вопрос по адресу:

NavigationAndPathfindingSystem@gmail.com, автор — Владимир Маевский.