Ant AI. Guide & Docs

Created by Anton Karlov (ant-karlov.ru)

Содержание

Обзор

Ant AI — это библиотека для создания искуственного интеллекта на базе GOAP (Goal-Oriented Action Planning). Особенность данного алгоритма заключается в том, что при разработке искуственного интеллекта не нужно создавать строгое дерево разветвлений для принятия решений. Все возможные действия задаются отдельно друг от друга и план действий выстраивается динамически, исходя из текущего состояния и поставленных целей.

Приемуществом данного подхода является и то, что описание состояния игрового мира или объекта для планировщика задается списком логических переменных, где может быть только два значения: правда или ложь. Например: “персонаж имеет оружие: правда”, “персонаж имеет патроны: ложь” — исходя из этих условий планировщик может принять решение, что персонажу следует искать патроны.

Более подробно о теории GOAP вы можете почитать на домашней страничке, где собрана вся необходимая информация: Goal-Oriented Action Planning (GOAP)

Ключевые особенности данной библиотеки:

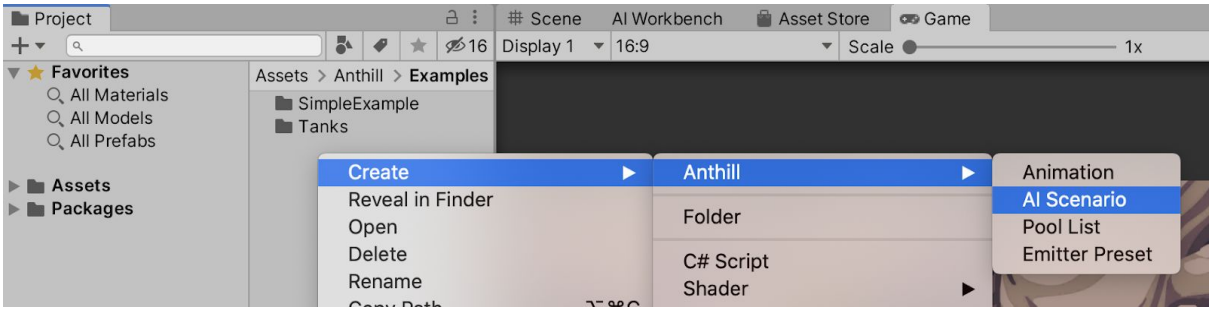
- Возможность создавать сценарии: наборы условий, действий и целей;
- Возможность использовать один и тот же сценарий поведения для нескольких игровых сущностей одновременно;
- Удобный редактор для редактирования и отладки сценариев поведения;
- Стандартная реализация планировщика;
- Кастомная реализация планировщика, где вы можете передать перечень условий и на выходе получить только список действий.

Сценарий

Сценарий — это *ScriptableObject,* который содержит все необходимые данные для работы планировщика. Вы можете назначить один экземпляр сценария разным юнитам, которые должны следовать одной стратегии. Например, если в вашей игре есть юниты-курьеры, то все сущности курьеров могут использовать один сценарий, разработанный для юнитов, занимающихся доставкой.

Создание сценария

В окне **“Project”** откройте контекстное меню и выберите **“Create” > “Anthill” > “AI Scenario”**. После чего будет создан *ScriptableObject,* который будет содержать все необходимые данные для работы планировщика.

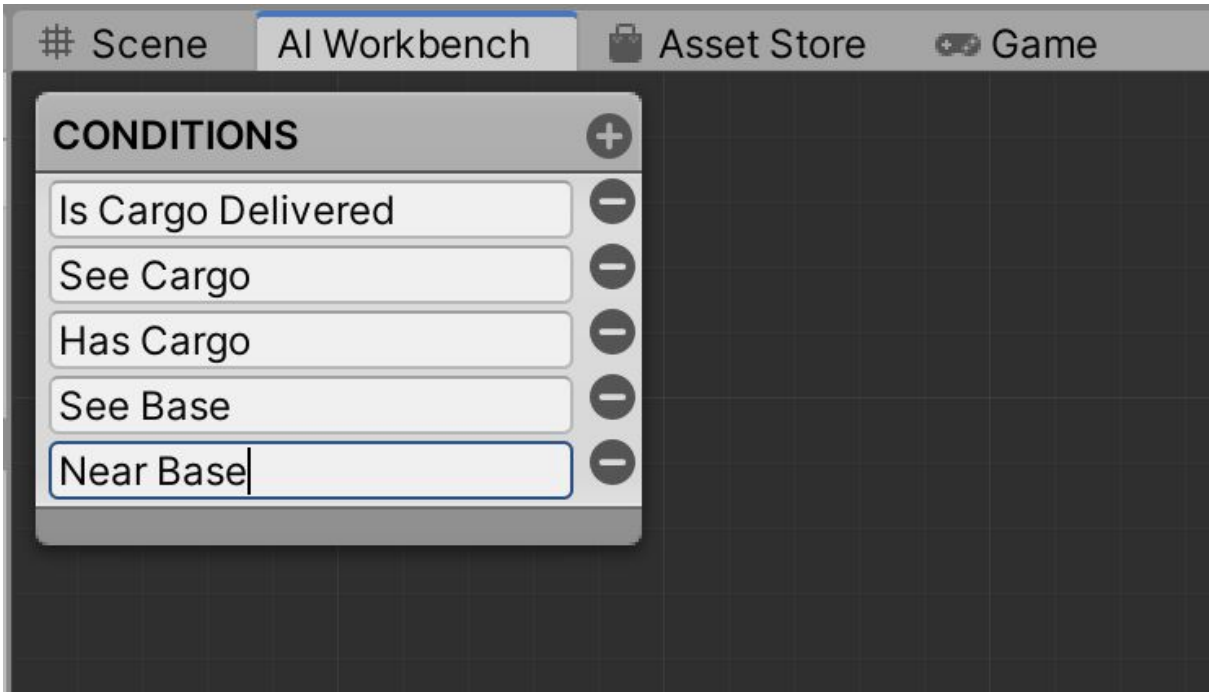


Выделите созданный *ScriptableObject* и в окне **“Inspector”** нажмите кнопку **“Open AI Workbench”** — перед вами откроется основное окно редактора. Расположите его удобным для себя образом.

Условия

Условия — это список логических переменных, которые могут принимать только два значения: **“true”** или **“false”**. Условия являются основой для создания сценария и используются для описания **“состояния мира”**, а так же для описания входных и выходных данных для действий.

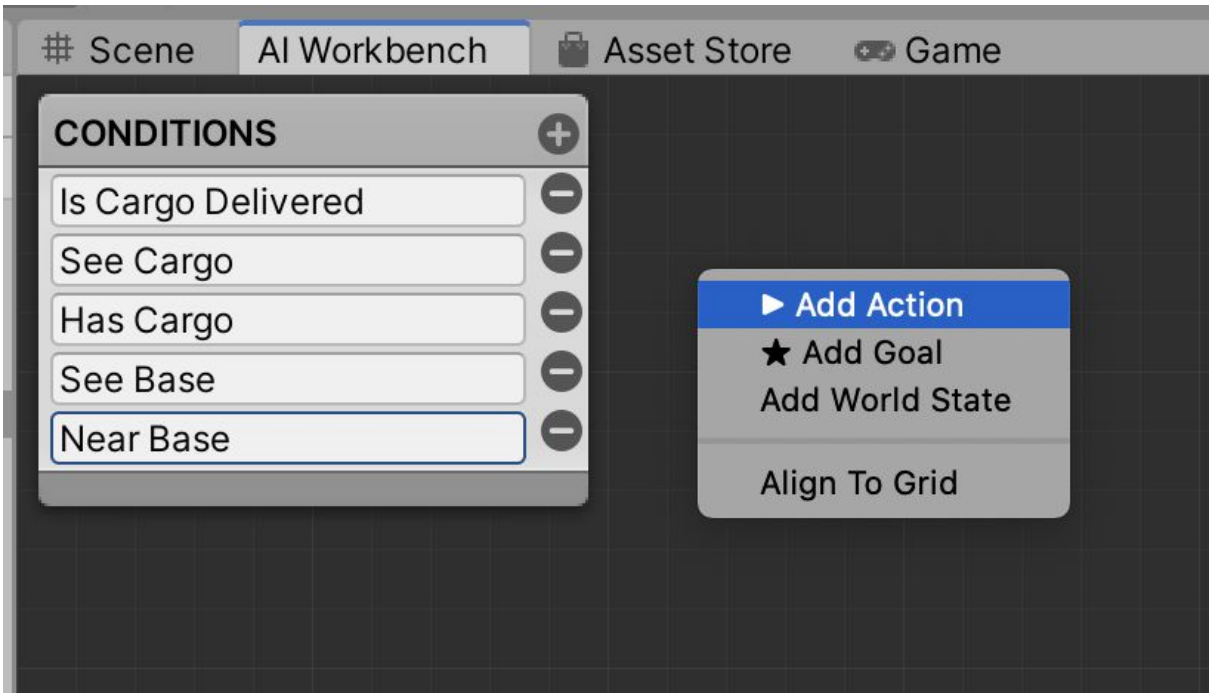
Для примера, добавьте следующие условия: **“Is Cargo Delivered”**, **“See Cargo”**, **“Has Cargo”**, **“See Base”**, **“Near Base”** — этот список условий позволит создать нам простой сценарий для юнита-курьера.



Действия

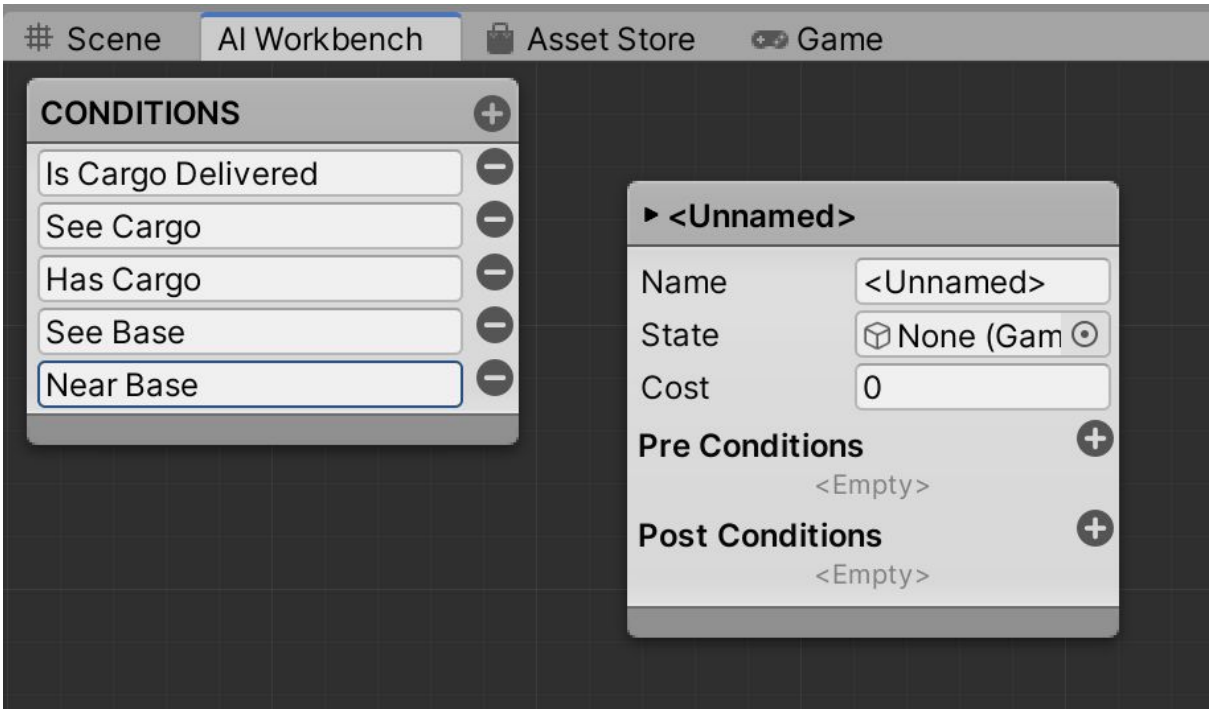
Действия — это то, что мы можем выполнить, например: “Загрузить груз”, “Доставить груз”, “Разгрузить груз”.

Чтобы создать новое действие откройте контекстное меню, кликнув по фону редактора, и выберите пункт **“Add Action”**.



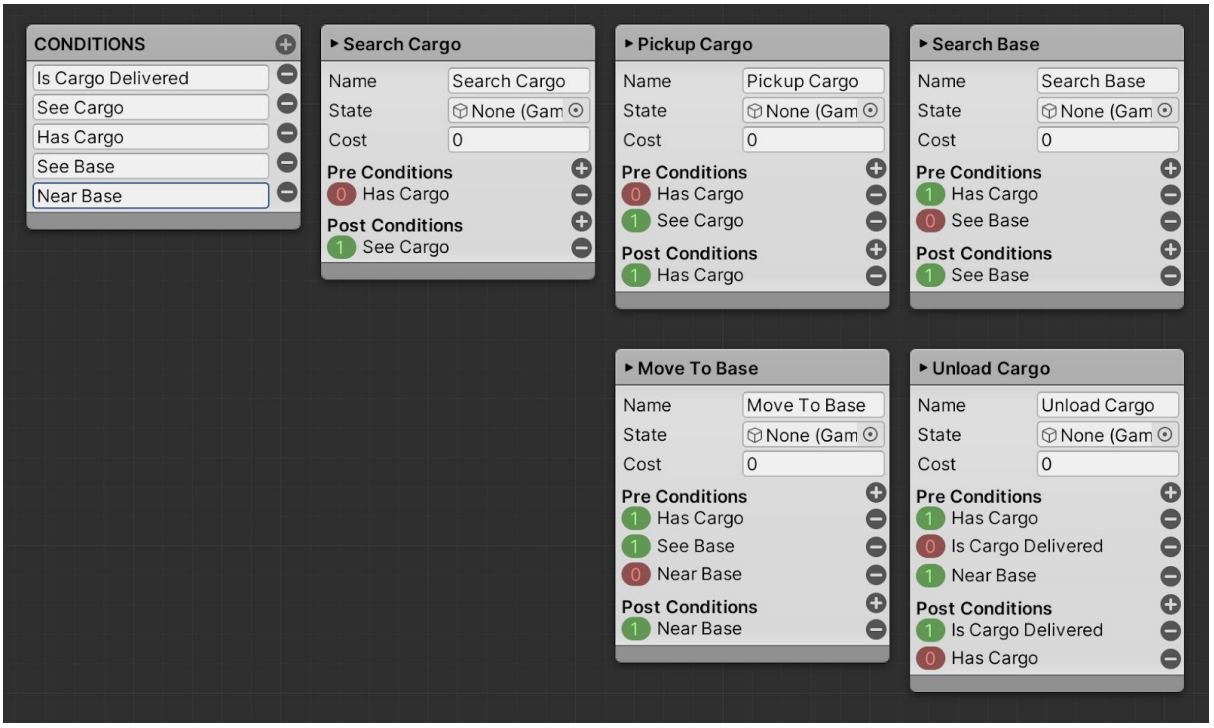
После того как действие будет добавлено, в окне редактора появится новая карточка. Вы можете свободно перестаскивать карточки в окне редактора.

Чтобы удалить карточку, выделите её и откройте для нее контекстное меню, выберите пункт **“Delete”**.



Действиям следует задать имя, привязать скрипт состояния (не обязательно), указать стоимость использования, а так же составить список условий **(Pre Conditions),** которые должны быть в состоянии мира, чтобы планировщик мог использовать данное действие и список условий **(Post Conditions),** которые будут изменены после выполнения данного действия.

Создайте несколько действий для нашего примера, как на скриншоте ниже.



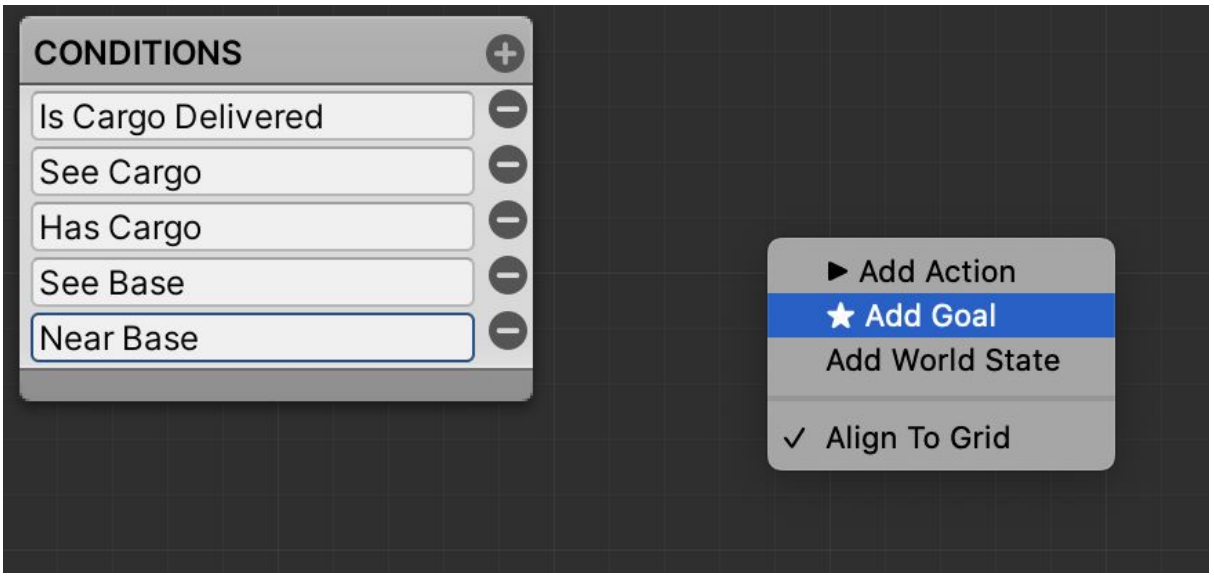
Если внимательно рассмотреть созданные карточки, можно обнаружить логику: если у юнита нет груза, то он начнет искать его. Если юнит увидит груз, то он приблизится к нему и загрузит, после чего будет искать базу, если её не видно, а когда увидит, приблизится и разгрузит груз.

Но планировщик не сможет построить план действий, если не будет знать о своей цели.

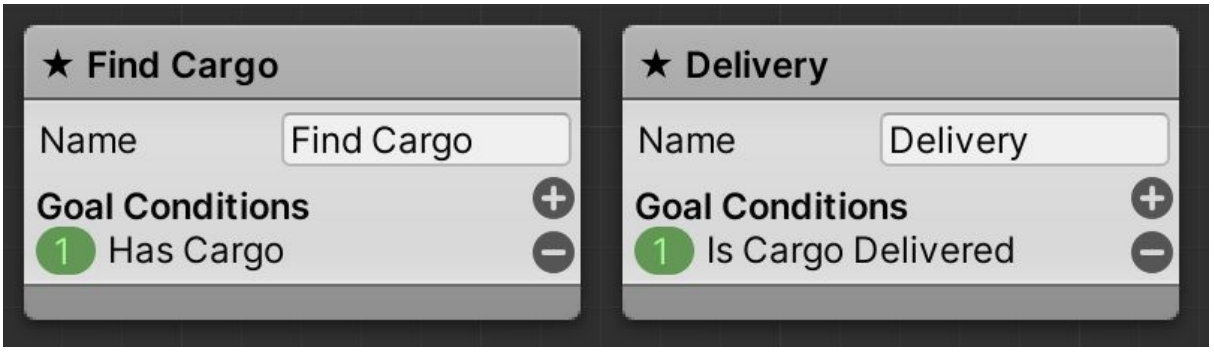
Цели

Цель — это особая карточка, которая дает знать планирвщику к чему он должен стремиться, когда выстраивает план. Если в ценарии не будет целей, то планировщик не сможет построить план действий.

Чтобы создать цель, вызовите контекстное меню в окне редактора и выберите **“Add Goal”**.



Создайте две цели на для нашего сценария, как на скриншоте ниже. Сделайте цель **“Find Cargo”** как цель по умолчанию — это значит, что эта цель будет активной по умолчанию. При необходимости вы сможете переключать цели из кода, если вам необходимо будет поменять стратегию планировщика.



Чтобы установить цель **“Find Cargo”,** как цель по умолчанию — выберите нужную карточку, и в контекстном меню для неё выберите пункт меню **“Set as Default”**. Цель по умолчанию будет отображаться синим цветом.

В карточке цели **“Find Cargo”** мы указали, что **“Has Cargo”** должен быть равен **“True”**, таким образом планировщик будет пытаться составить план действий так, чтобы достичь данного условия.

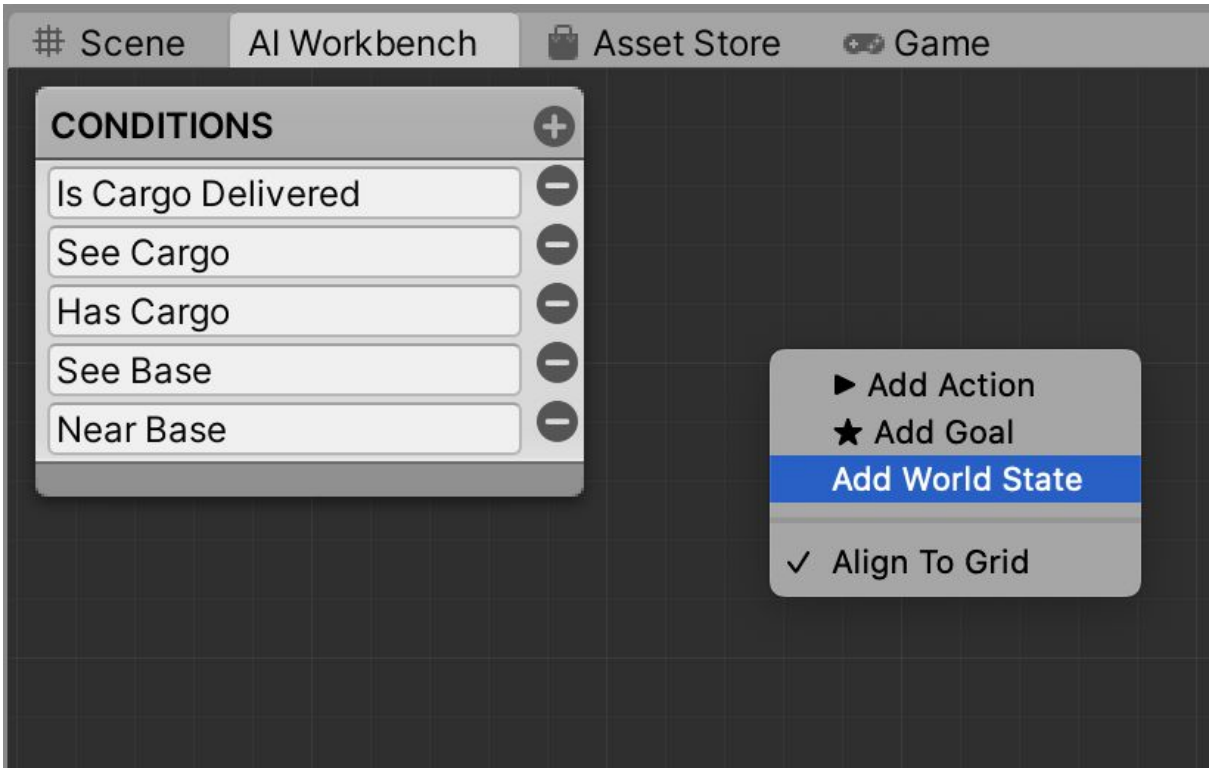
Состояние мира

Состояние мира — это список условий, описывающих текущее состояние игрового объекта. Состояние мира позволяет понять планировщику текущую обстановку, и исходя из этого планировщик строит план действий, которые меняют состояние мира до нужного.

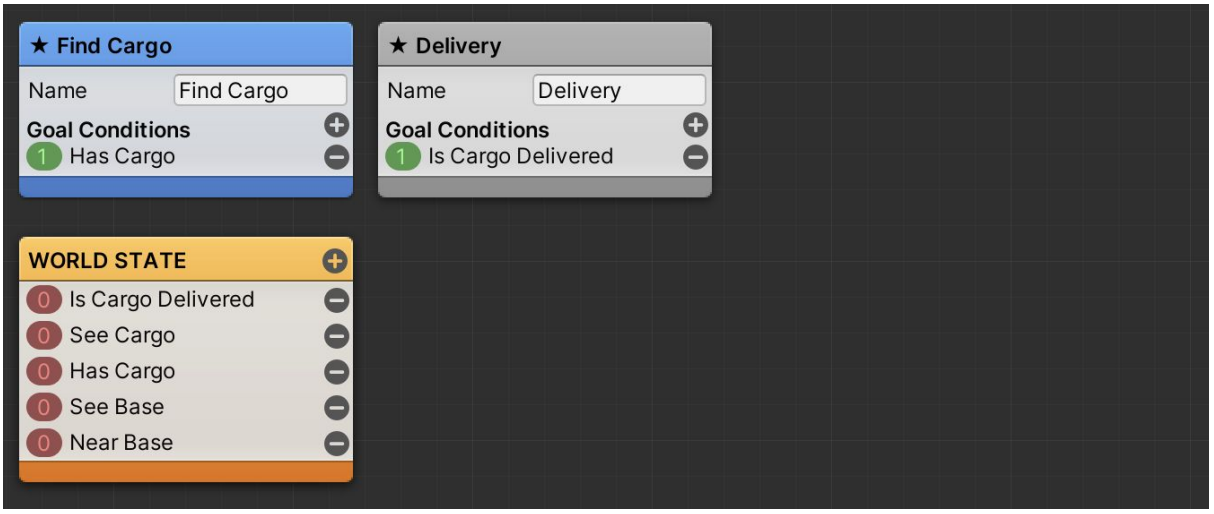
Состояние мира не является полным описанием игрового мира, обычно это состояние мира для конкретного игрового юнита. У каждого юнита и его сценария может быть разный набор условий, описывающий только их состояния мира.

Состояние мира строится на основе сенсоров юнита: зрение, касаний, слуха, содержимое инвентаря и тому подобное. Как будет выстроено состояние мира для юнитов, зависит только от вас.

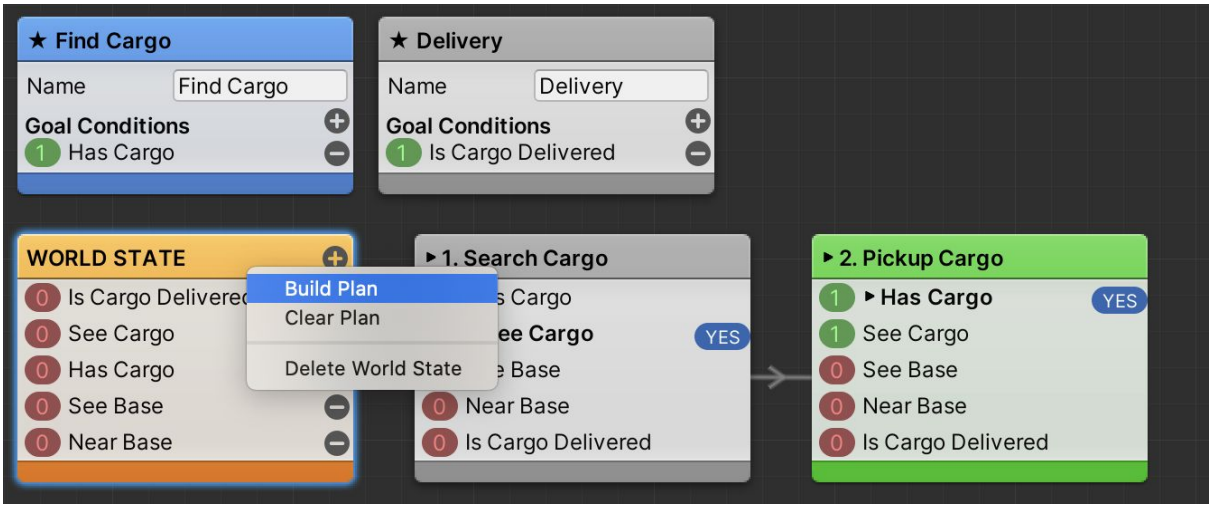
Чтобы создать тестовое состояние мира для тестирования сценария, вызовете контекстное меню и выберите пункт меню **“Add World State”**.



Добавьте в карточку состояния мира все или несколько условий. Наше тестовое состояние должно выглядеть, как на изображении ниже.



Чтобы построить план из текущего состояния, выберите карточку **“World State”** и в контекстном меню выберите пункт меню **“Build Plan”**.



Мы установили текущую цель **“Find Cargo”** — задача планировщика установить условие **“Has Cargo”** равным **“true”**. Планировщик построил план из двух действий: **“Search Cargo”** и **“Pickup Cargo”** — так как последовательное выполнение этих действий изменит заданное условие в нужное нам значение.

Условия, которые изменялись при выполнении каждого действия, отмеченны жирным шрифтом и дополнительным тегом **“YES”** или **“NO”** — в зависимости от того, как изменилось значение.

Зеленый цвет последнего действия означает, что именно это действие привело к успеху и цель была достигнута. В тех случаях когда невозможно достичь поставленной цели, последняя карточам может быть красной. Это значит, что планировщик не может найти решения для достижения поставленной цели. Но даже провальный план может быть полезным, так как он может содержать действия,

выполняя которые состояние мира может измениться и дальнешие обновления плана приведут к успеху.

Вы можете по экспериментировать с состоянием мира, выключая или включая разные условия и перестраивая план, чтобы посмотреть, как поведет себя сценарий в разных ситуациях.

[Важно] Состояние мира, созданное внутри сценария, используется исключительно для тестирования и никак не влияет на работу планировщика, использующего данный сценарий внутри игры.

Имплементация

Данная библиотека предусматривает два вида имплементации: *стандартную* и *кастомную*.

Стандартная имплементация подразумевает набор компонентов, которые связывают игровой объект с планировщиком, сценарием, а так же реализуют работу действий.

Если по какой-то причине вас не устраивает стандартная имплементация планировщика и/или вы знаете, как можно реализовать работу состояний/действий для своего проекта лучше — вы можете использовать кастомную имплементацию, где вы работаете с «сырыми» данными планировщика.

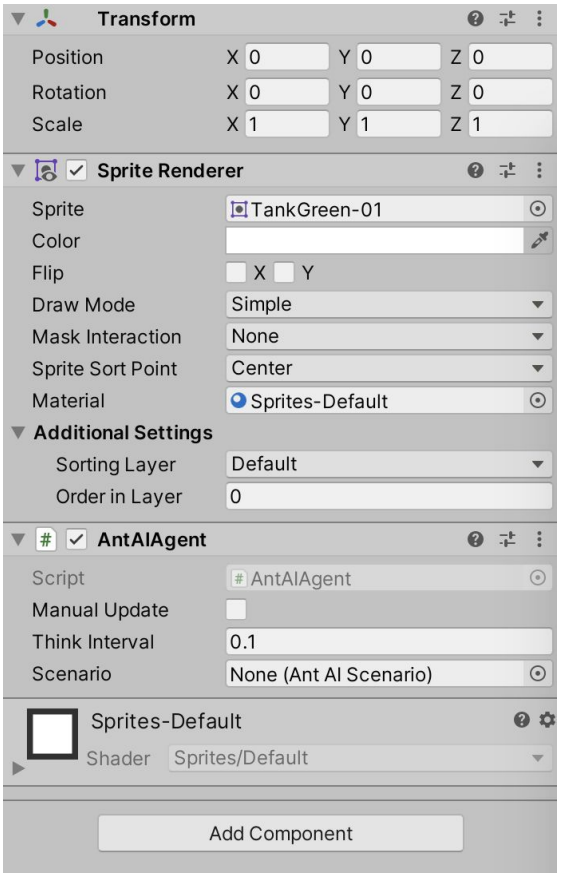
Стандартная

Стандартная имплементация позволяет привязывать скрипты для реализации действий (состояний) объекта через окно редактора. В данном разделе мы рассмотрим самый простой пример того, как это можно сделать.

(Исходный код данного примера вы можете найти в папке: **Anthill/Examples/DefaultUse**)

1. Создайте новую сцену и добавьте на нее любой объект, который будет представлять собой игрового юнита.

2. Прикрепите к объекту юнита компоненты: **SpriteRenderer** и **AntAIAgent,** как на изображении ниже.



AntAIAgent — это стандарный компонент библиотеки «Ant AI», реализующий связь между игровым объектом, сценарием и планировщиком.

3. В поле **Scenario** компонента **AntAIAgent** установите сценарий (*ScriptableObject*), который мы создали в предыдущих разделах данного документа.

Свойство **Manual Update** позволяет запретить автоматическое обновление плана, и в этом случае нужно будет вручную вызывать метод `Execute(deltaTime, timeScale)` для **AntAIAgent,** чтобы обновить план. Такая возможность будет полезна, например, если ваша игра пошаговая и нет необходимости перестраивать план каждый игровой тик.

Свойство **Think Interval** позволяет установить интервал между обновлениями плана в секундах. Сам процесс построения плана не очень сложный, но перед построением плана **AntAIAgent** опрашивает сенсоры юнита (ISensor), чтобы обновить состояние мира. Если реализация сенсоров сложная, то стоит подобрать оптимальное значение задержки, чтобы не перестраивать план слишком часто.

4. Теперь мы создадим скрипт **UnitControl.cs** — этот скрипт будет отвечать за инвентарь юнита. Добавьте этот скрипт к объекту юнита.

```
// UnitControl.cs --
using UnityEngine;

public class UnitControl : MonoBehaviour
{
    public bool HasCargo { get; set; }
}
```

5. Далее нам следует создать скрипт **UnitSense.cs,** который реализует все сенсоры юнита. Этот скрипт должен имплементировать интерфейс `ISense`, чтобы **AntAIAgent** автоматически мог найти данный орган чувств и вызывать для него метод `CollectConditions()`, каждый раз когда ему это потребуется. Этот скрипт так же следует прикрепить к объекту юнита.

```
// UnitSense.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;

public class UnitSense : MonoBehaviour, ISense
{
    private UnitControl _control;
    private Transform _t;
    private Transform _base;
    private Transform _cargo;

    private void Awake()
    {
        _control = GetComponent<UnitControl>();
        _t = GetComponent<Transform>();

        // Save reference for the bot base.
        var go = GameObject.Find("Base");
        Debug.Assert(go != null, "Base object not exists on the scene!");
        _base = go.GetComponent<Transform>();

        // Save reference for the cargo.
        go = GameObject.Find("Cargo");
        Debug.Assert(go != null, "Cargo object not exists on the scene!");
        _cargo = go.GetComponent<Transform>();
    }

    /// <summary>
    /// This method will be called automaticaly each time when
    /// AntAIAgent decide to update the plan. You should attach this script
    /// to the same object where is AntAIAgent.
    /// </summary>
    public void CollectConditions(AntAIAgent aAgent, AntAICondition aWorldState)
    {
        aWorldState.BeginUpdate(aAgent.planner);
```

```
        {
            aWorldState.Set("Is Cargo Delivered", false);
            aWorldState.Set("See Cargo", IsSeeCargo());
            aWorldState.Set("Has Cargo", _control.HasCargo);
            aWorldState.Set("See Base", IsSeeBase());
            aWorldState.Set("Near Base", IsNearBase());
        }
        aWorldState.EndUpdate();
    }

    private bool IsSeeCargo()
    {
        return (AntMath.Distance(_t.position, _cargo.position) < 1.0f);
    }

    private bool IsSeeBase()
    {
        return (AntMath.Distance(_t.position, _base.position) < 1.5f);
    }

    private bool IsNearBase()
    {
        return (AntMath.Distance(_t.position, _base.position) < 0.1f);
    }
}
```

6. Теперь создадим скрипт реализующий действие **“Search Cargo”**, назовем скрипт **SearchCargoState.cs** и добавим в него следующий код:

```
// SearchCargoState.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;

public class SearchCargoState : AntAIState
{
    private const float SPEED = 2.0f;

    private Transform _t;
    private Vector3 _targetPos;
    private float _targetAngle;

    public override void Create(GameObject aGameObject)
    {
        _t = aGameObject.GetComponent<Transform>();
    }

    public override void Enter()
    {
        // Search cargo on the map.
        var go = GameObject.Find("Cargo");
        if (go != null)
```

```
        {
            // Save position of the cargo for movement.
            _targetPos = go.transform.position;

            // Calc target angle.
            _targetAngle = AntMath.AngleDeg(_t.position, _targetPos);
            _t.rotation = Quaternion.Euler(_t.rotation.x, _t.rotation.y,
_targetAngle);
        }
        else
        {
            Debug.Log("Cargo not found!");
            Finish();
        }
    }

    public override void Execute(float aDeltaTime, float aTimeScale)
    {
        // Move to the cargo.
        var pos = _t.position;
        pos.x += SPEED * aDeltaTime * Mathf.Cos(_targetAngle);
        pos.y += SPEED * aDeltaTime * Mathf.Sin(_targetAngle);
        _t.position = pos;

        // Check distance to the cargo.
        if (AntMath.Distance(pos, _targetPos) <= 0.2f)
        {
            // We arrived!
            // Current action is finished.
            Finish();
        }
    }
}
```

Обратите внимание, что данный скрипт унаследован от **AntAIState** — это абстрактный класс, реализующий необходимый функционал для обработки состояний. Каждый скрипт состояния должен быть унаследован от этого класса.

AntAIState имеет полезные методы, которые вы можете перекрыть, чтобы реализовать нужное вам поведение:

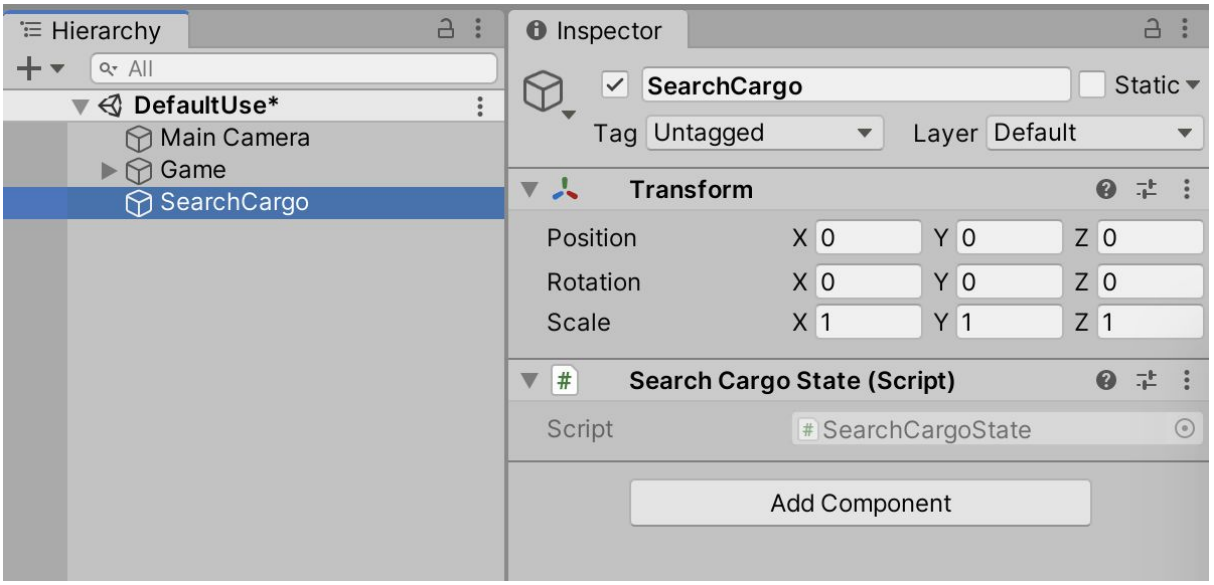
`void Create(GameObject aGameObject)` — вызывается один раз, когда будет инициализирован игровой объект и текущее состояние будет создано. В качестве аргумента передается *GameObject,* к которому привязан **AntAIAgent,** что дает вам возможность получить все необходимые компоненты игрового объекта для последующей работы с ними.

`void Enter()` — вызывается один раз перед тем, как данное действие будет активировано.

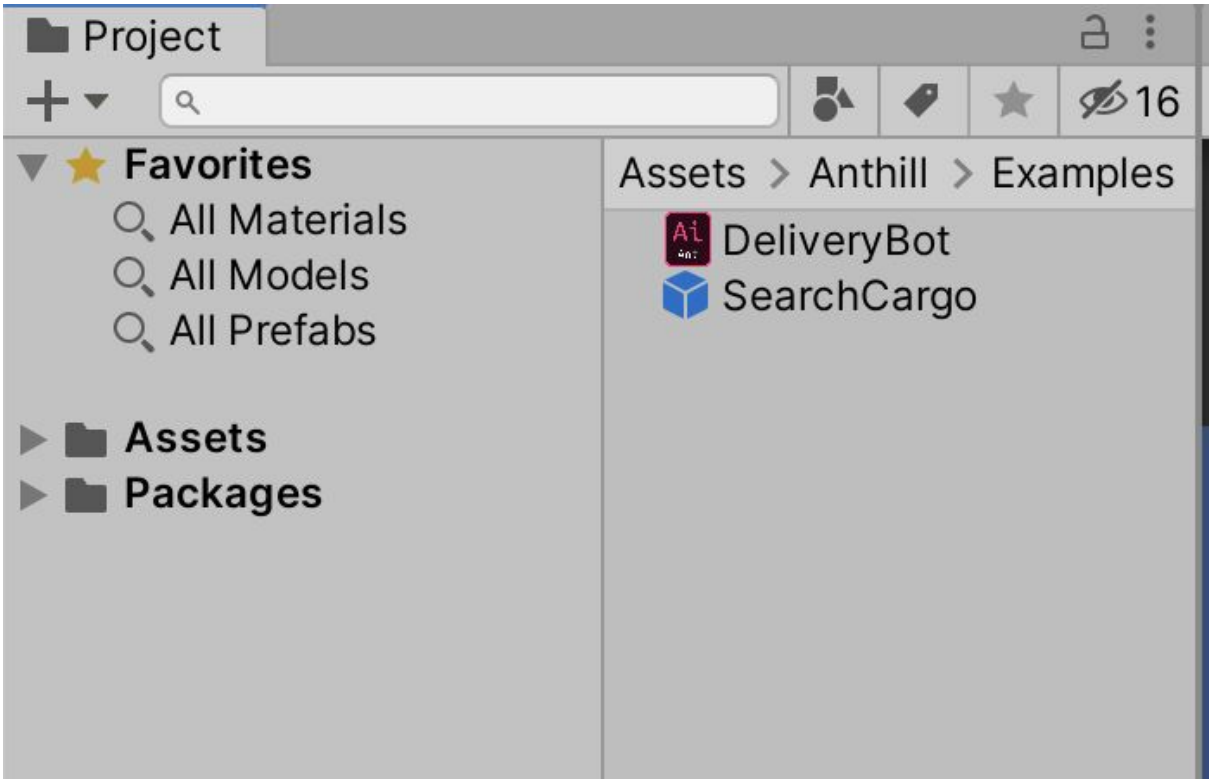
`void Execute(float aDeltaTime, float aTimeScale)` — вызывается каждый игровой тик, пока данное действие активно.

`void Exit()` — вызывается один раз перед тем, как данное действие будет деактивировано.

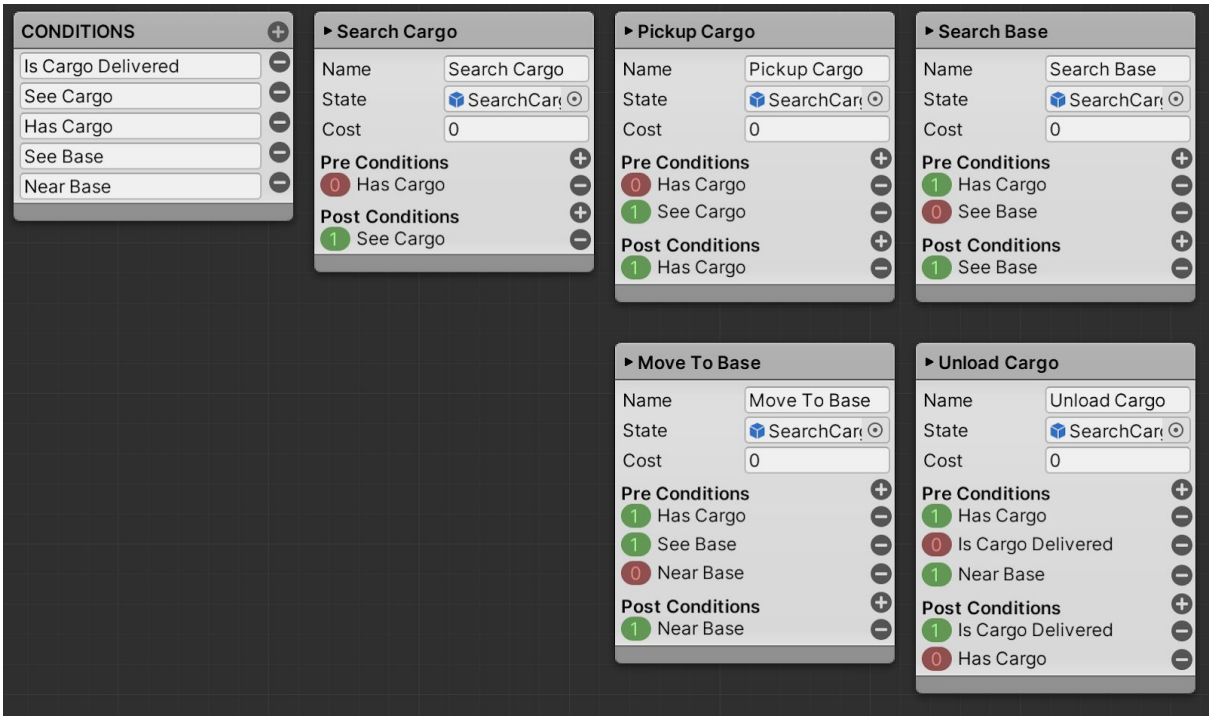
7. Чтобы добавить созданный скрипт действия к нашему сценарию необходимо его привязать к пустому объекту на сцене. Создайте новый объект на сцене в окне иерархии и добавьте к нему скрипт **SearchCargoScript.cs,** как на изображении ниже.



8. Сохраните созданный префаб, как шаблон, перетащив его в окно **“Project”** в ту же папку где располагается сценарий, как на изображении ниже.



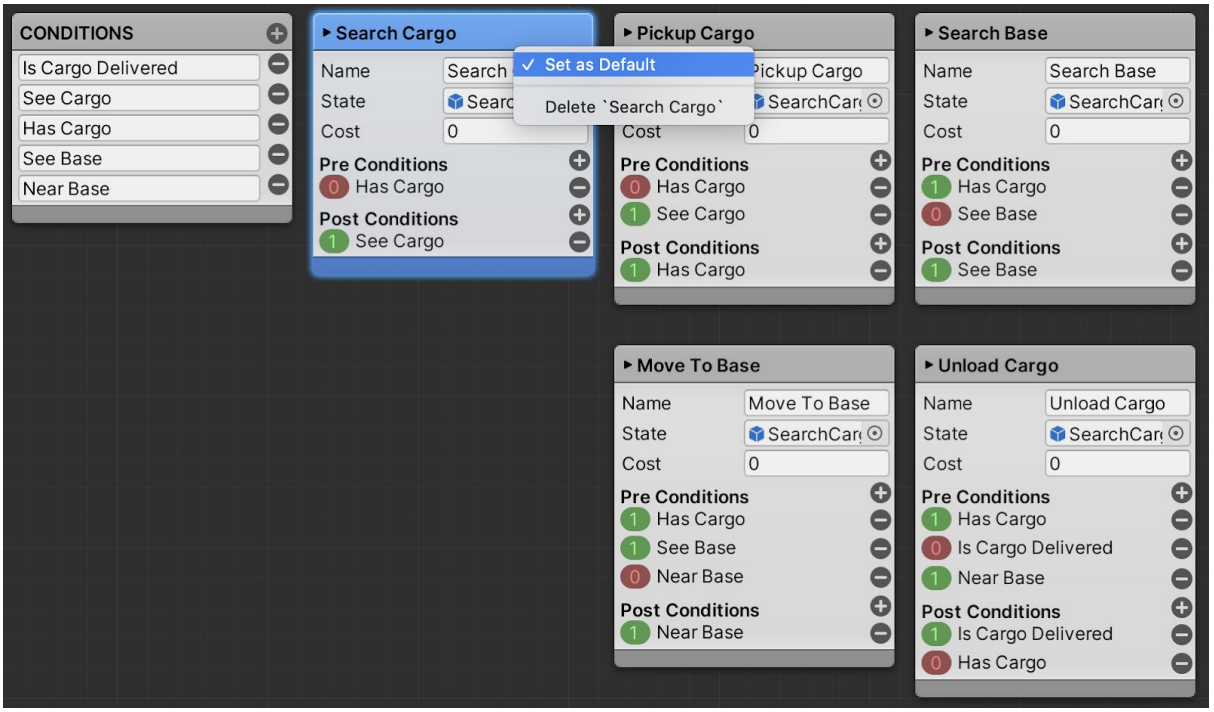
9. Откройте сценарий **DeliveryBot** в **AI Workbench** и перетащите только что созданное действие в поле *State* для всех карточек действий, как на картинке ниже:



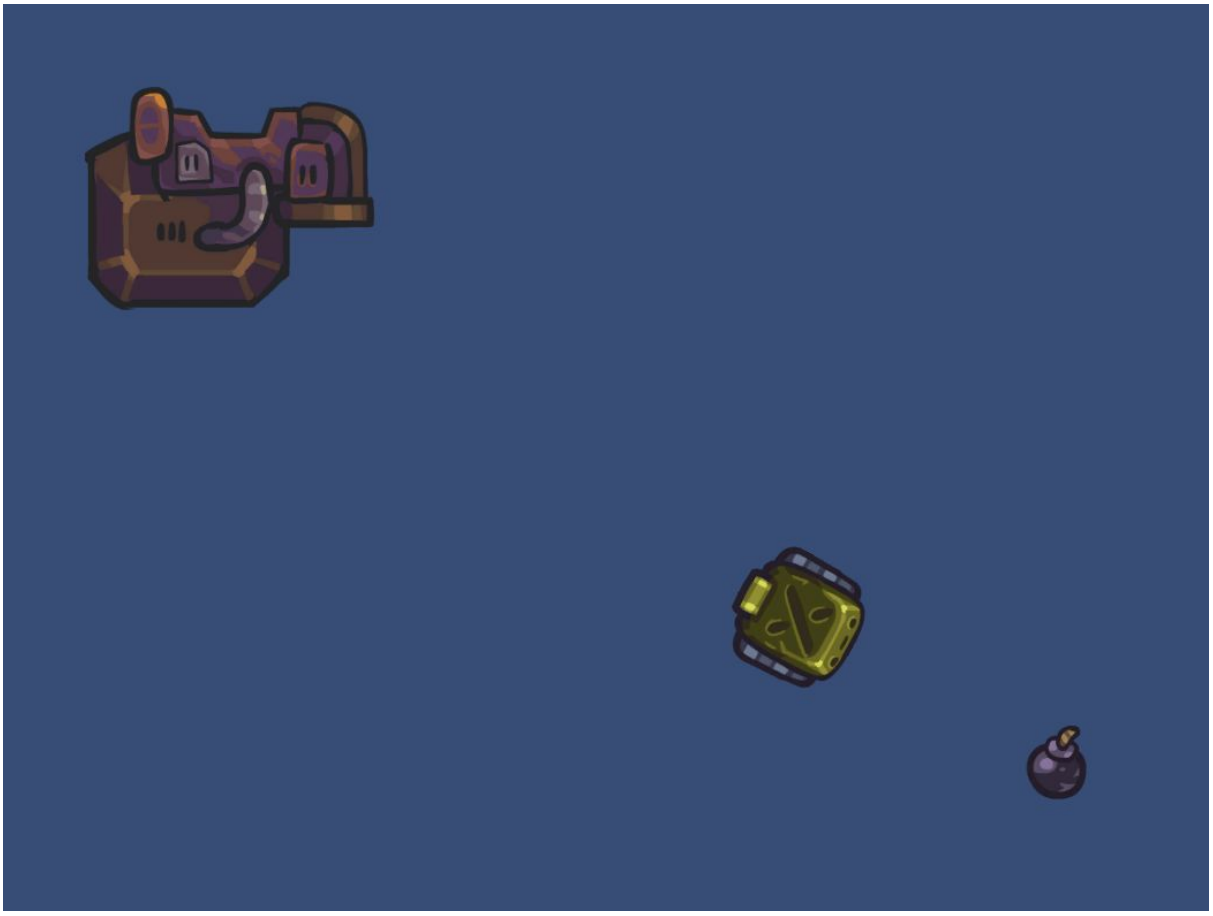
Это выглядит немного глупо. Но мы же хотим получить результат максимально быстро! **AntAIAgent** не будет работать пока не убедится, что все действия

настроенные в сценарии имеют скрипт состояния, поэтому это неплохой способ обмануть его ;)

10. Выберите карточку **“Search Cargo”** и установите её как действие по умолчанию. Таким образом мы дадим знать **AntAIAgent,** что в любой непонятной ситуации ему следует использовать это действие. В своих проектах вы можете создать отдельную карточку для этого, которую можно назвать как **“Idle”** или **“Stuck”**.



11. Запустите игру и посмотрите, как курьер направился к грузу, который изображен на картинке ниже, как бомба. Все работает как нужно, правда достигнув груза, бот застрянет возле него — это потому, что у нас нет действия **“Pickup Cargo”**.



12. Теперь создадим скрипт **PickupCargoState.cs** и добавим в него следующий код:

```
// PickupCargoState.cs --
using UnityEngine;
using Anthill.AI;

public class PickupCargoState : AntAIState
{
    private UnitControl _control;

    public override void Create(GameObject aGameObject)
    {
        _control = aGameObject.GetComponent<UnitControl>();
    }

    public override void Enter()
    {
        // Search cargo on the map and disable it.
        var go = GameObject.Find("Cargo");
        if (go != null)
        {
            go.SetActive(false);
        }
```

```
        _control.HasCargo = true;
        Finish();
    }
}
```

Это действие будет выполнено сразу как только мы его активировали. Так же как и в прошлый раз создайте пустой объект на сцене, прикрепите к нему скрипт **PickupCargoState.cs** и сохраните префаб в окне **Project**.

13. Создадим скрипт **SearchBaseState.cs** и добавим в него следующий код:

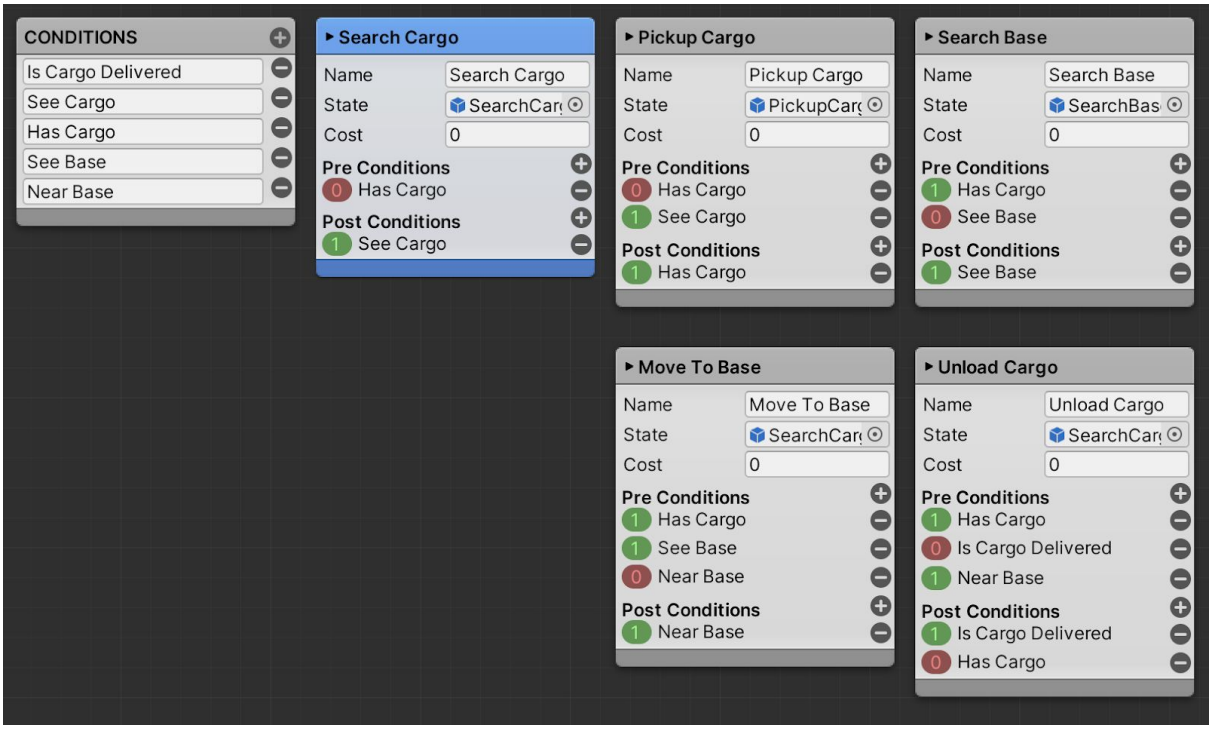
```
// SearchBaseState.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;

public class SearchBaseState : AntAIState
{
    private const float SPEED = 2.0f;

    private Transform _t;
    private Vector3 _targetPos;
    private float _targetAngle;

    public override void Create(GameObject aGameObject)
    {
        _t = aGameObject.GetComponent<Transform>();
    }

    public override void Enter()
        // Search base on the map.
        var go = GameObject.Find("Base");
        if (go != null)
        {
            // This action called if we don't see the base.
            // So try to find by selecting random position
            // around the base until we found it.
            var basePos = go.transform.position;
            _targetPos = new Vector3(
                basePos.x + AntMath.RandomRangeFloat(-2.0f, 2.0f),
                basePos.y + AntMath.RandomRangeFloat(-2.0f, 2.0f),
                0.0f
            );

            // Calc target angle.
            _targetAngle = AntMath.AngleDeg(_t.position, _targetPos);
            _t.rotation = Quaternion.Euler(_t.rotation.x, _t.rotation.y,
_targetAngle);
            _targetAngle *= Mathf.Deg2Rad;
        }
```

```
        else
        {
            Debug.Log("Base not found!");
            Finish();
        }
    }

    public override void Execute(float aDeltaTime, float aTimeScale)
    {
        // Move to the base.
        var pos = _t.position;
        pos.x += SPEED * aDeltaTime * Mathf.Cos(_targetAngle);
        pos.y += SPEED * aDeltaTime * Mathf.Sin(_targetAngle);
        _t.position = pos;

        // Check distance to the base.
        if (AntMath.Distance(pos, _targetPos) <= 0.2f)
        {
            // We arrived!
            // Current action is finished.
            Finish();
        }
    }
}
```

Обратите внимание, что этот скрипт практически повторяет код скрипта **SeachCargoState.cs** — это говорит о том, что возможно стоит вынести управление и движение юнита в класс **UnitControl.cs**, а в состояниях только давать команды куда юниту двигаться и ждать когда команда будет выполнена, чтобы закончить действие.

Так же как и в прошлый раз, создайте новый объект на сцене **SearchBase**, прикрепите к нему скрипт **SearchBaseState.cs** и перетащите в окно **“Project”,** чтобы сохранить его.

14. Привяжите только что созданные действия к действиям в сценарии, перетащив префабы из окна проекта в поля *State* для карточек действий, как на скриншоте ниже.



После чего вы можете запустить пример и убедиться в том, что когда юнит подобрал груз, он поедет в случайную точку возле базы, если база находится вне поля его зрения. Но когда он найдет базу и доедет до нее, то все ломается. Это потому, что к действию **“Move To Base”** у нас привязано состояние **SearchCargo** — исправим это.

15. Создаем новый скрипт **MoveToBaseState.cs** и добавляем в него следующий код:

```
// MoveToBaseState.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;

public class MoveToBaseState : AntAIState
{
    private const float SPEED = 2.0f;

    private Transform _t;
    private Vector3 _targetPos;
    private float _targetAngle;

    public override void Create(GameObject aGameObject)
    {
        _t = aGameObject.GetComponent<Transform>();
    }

    public override void Enter()
        // Search base on the map.
        var go = GameObject.Find("Base");
```

```
        if (go != null)
        {
            _targetPos = go.transform.position;

            // Calc target angle.
            _targetAngle = AntMath.AngleDeg(_t.position, _targetPos);
            _t.rotation = Quaternion.Euler(_t.rotation.x, _t.rotation.y,
_targetAngle);
            _targetAngle *= Mathf.Deg2Rad;
        }
        else
        {
            Debug.Log("Base not found!");
            Finish();
        }
    }

    public override void Execute(float aDeltaTime, float aTimeScale)
    {
        // Move to the base.
        var pos = _t.position;
        pos.x += SPEED * aDeltaTime * Mathf.Cos(_targetAngle);
        pos.y += SPEED * aDeltaTime * Mathf.Sin(_targetAngle);
        _t.position = pos;

        // Check distance to the base.
        if (AntMath.Distance(pos, _targetPos) <= 0.2f)
        {
            // We arrived!
            // Current action is finished.
            Finish();
        }
    }
}
```

Снова создаем пустой объект на сцене и привязываем к нему скрипт **MoveToBase.cs** и сохраняем в окне **Project**, чтобы потом привязать его к сценарию.

16. Создаем еще один скрипт **UnloadCargoState.cs** и добавляем в него следующий код:

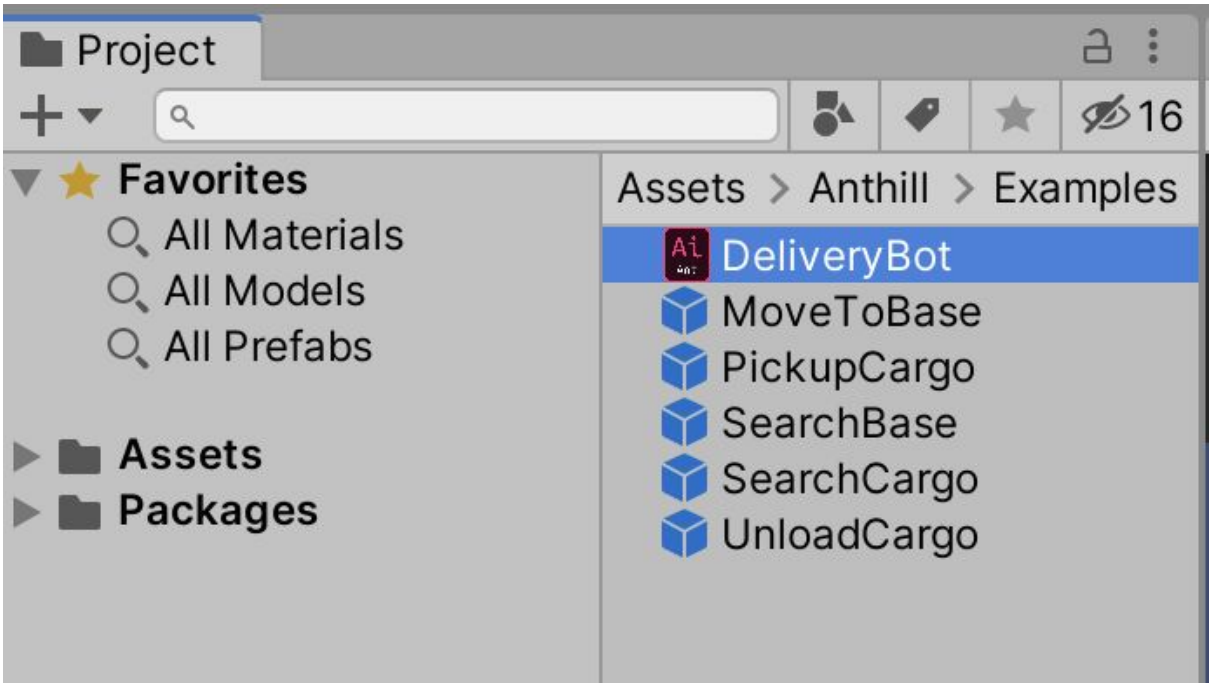
```
// UnloadCargoState.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;

public class UnloadCargoState : AntAIState
{
    private UnitControl _control;
    private GameObject _cargoRef;
    private Vector3 _initialPos;
```

```
    public override void Create(GameObject aGameObject)
    {
        // Save reference to the cargo to respawn it when
        // unit finish delivery.
        _cargoRef = GameObject.Find("Cargo");
        Debug.Asset(_cargoRef != null, "Cargo not found on the scene!");
        _initialPos = _cargoRef.transform.position;
        _control = aGameObject.GetComponent<UnitControl>();
    }

    public override void Enter()
    {
        // Spawn cargo again on the map in the random pos.
        _cargoRef.transform.position = new Vector3(
            _initialPos.x + AntMath.RandomRangeFloat(-2.0f, 2.0f),
            _initialPos.y + AntMath.RandomRangeFloat(-2.0f, 2.0f),
            0.0f
        );
        _cargoRef.SetActive(true);

        _control.HasCargo = false;
        Finish();
    }
}
```

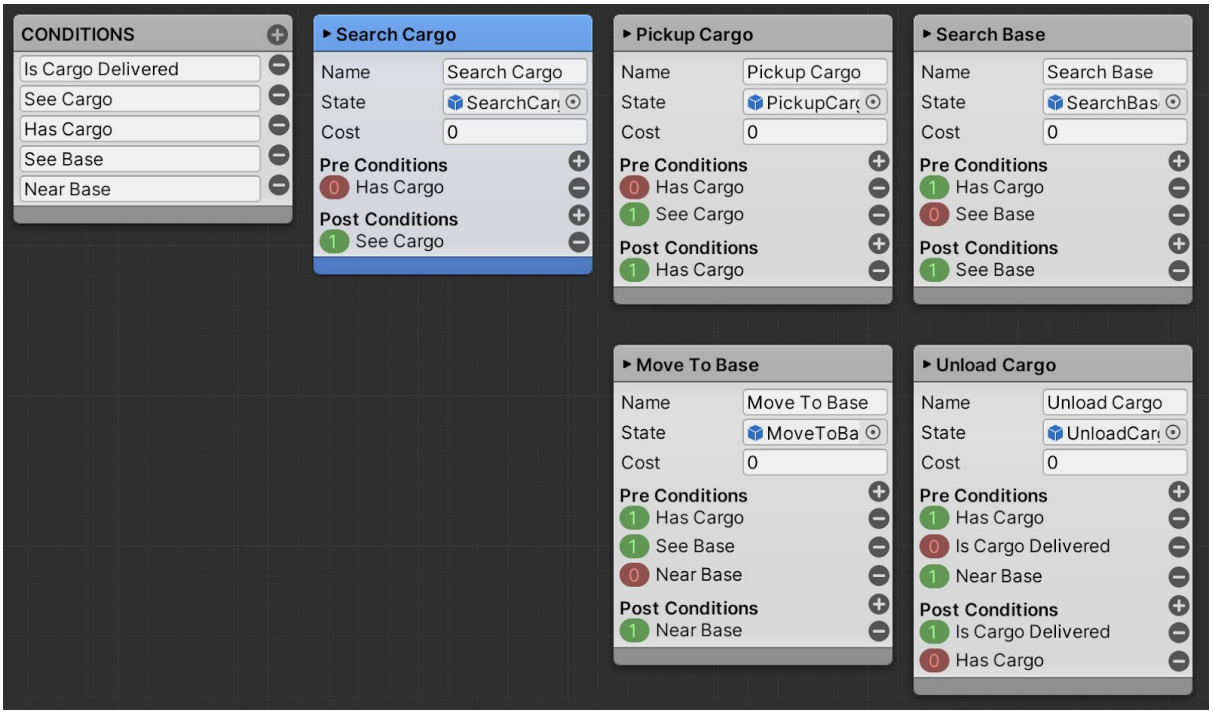
В этом скрипте при создании действия, мы навсегда сохраняем указатель на объект груза на сцене, чтобы воссоздать его вновь, когда предыдущий груз будет доставлен. Это неверный подход, но для данного примера этого достаточно. Когда юнит выгрузит свой груз, на сцене появится новый груз. И так будет происходить до бесконечности.

Создаем новый игровой объект на сцене, привязываем к нему скрипт **UnloadCargoState.cs**, сохраняем его в окне **Project** и привязываем к нашему сценарию.

17. Как итог, у нас должно получится несколько действий:



Которые должны быть привязаны к сценарию следующим образом:



Теперь вы можете запустить пример и наслаждаться работой курьера, который будет бесконечно доставлять груз на базу.

Это все, что вам необходимо знать, чтобы начать реализацию ИИ в вашем проекте!

Произвольная

Чтобы реализовать свою имплементацию для работы планировщика и построения плана действий для ваших юнитов, вам нужно всего несколько вещей:

- Создать экземпляр планировщика;
- Загрузить в него сценарий (SciptableObject);
- Установить текущее состояние мира;
- Установить или изменить цель (не обязательно).
- Построить план и использовать его для организации дальнейшего поведения вашего юнита.

Код самой простой кастомной имплементации представлен на скриншоте ниже:

```
// SimpleExample.cs --
using UnityEngine;
using Anthill.AI;

public class SimpleExample : MonoBehaviour
{
    public AntAIScenario scenario;

    private void Start()
    {
        // 1. Create the AI planner.
        // -------------------------
        var planner = new AntAIPlanner();

        // Load scenario for planner.
        Debug.Asset(scenario != null, "Scenario is missed!");
        planner.LoadScenario(scenario);

        // 2. Create world state.
        // ----------------------
        var worldState = new AntAICondition();
        worldState.BeginUpdate();
        worldState.Set("Is Cargo Delivered", false);
        worldState.Set("See Cargo", false);
        worldState.Set("Has Cargo", false);
        worldState.Set("See Base", false);
        worldState.Set("Near Base", false);
        worldState.EndUpdate();

        // 3. Build plan.
        // --------------
        var plan = new AntAIPlan();
        planner.MakePlan(ref plan, worldState, planner.GetDefaultGoal());

        // Output plan.
        Debug.Log("<b>Plan:</b>");
```

```
        for (int i = 0; i < plan.Count; i++)
        {
            Debug.Log($"{(i + 1)}. {plan[i]}");
        }

        // 4. Change world state and rebuild plan.
        // ---------------------------------------
        worldState.BeginUpdate(planner);
        worldState.Set("Has Cargo", true);
        worldState.EndUpdate();

        planner.MakePlan(ref plan, worldState, planner.FindGoal("Delivery"));

        // Output plan.
        Debug.Log("<b>Plan:</b>");
        for (int i = 0; i < plan.Count; i++)
        {
            Debug.Log($"{(i + 1)}. {plan[i]}");
        }
    }
}
```

Исходный код данного примера вы можете найти в комплекте с данным ассетом. Путь: **“Anthill/Examples/SimpleExample/”**

Поддержка

Если у вас возникли какие-либо сложности, вы нашли ошибки в работе редактора или алгоритма, вам нужен совет или у вас есть пожелания — не стесняйтесь! Просто свяжитесь со мной:

- E-mail — ant.anthill@gmail.com
- Facebook — https://www.facebook.com/groups/526566924630336/
- Discord — https://discord.gg/D9fASJ5