Ant AI. Guide & Docs

Created by Anton Karlov (ant-karlov.ru)

Contents

Overview

Ant AI is a library for creating artificial intelligence based on GOAP (Goal-Oriented Action Planning). The peculiarity of this algorithm is that when developing artificial intelligence, it is not necessary to create a strict branching tree for making decisions. All possible actions are set separately from each other and the action plan is built dynamically based on the current state and goals.

An advantage of this algorithm is that the description of the state of the game world or object for the planner is given by a list of logical variables where there can be only two values: true or false. For example: “the character has weapon: true”, “the character has ammo: false” - on the basis of these conditions, the planner can decide that the character should look for ammo.

You can read more about GOAP theory here: Goal-Oriented Action Planning (GOAP)

Key features of **Ant AI**:

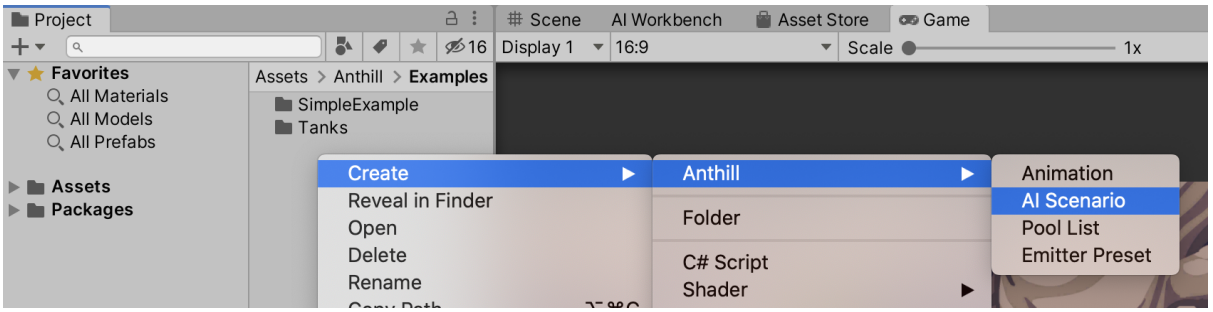
- Possibility to create AI scenarios where is setup conditions, actions and goals;
- Possibility to use the same behavior scenario for multiple game entities simultaneously;
- Convenient editor for editing and debugging behavior scenarios;
- Simple standart AI planner Implementation;
- Custom AI planner where you can transfer a list of conditions and get only a list of actions at the output;
- Works with any type of view: 3D, 2D, top-down, isometria e.t.c.;
- Works with any platforms;
- Beatiful examples.

AI Scenario

Scenario is a *ScriptableObject* that contains all the necessary data for the planner to work. You can assign one instance of the scenario to different units that must follow the same strategy. For example, if there are delivery units in your game, then all delivery entities can use one scenario.

Scenario creation

In the **“Project”** window, open the context menu and select **“Create”> “Anthill”> “AI Scenario”**. After that, a *ScriptableObject* will be created that will contain all the necessary data for the planner to work.

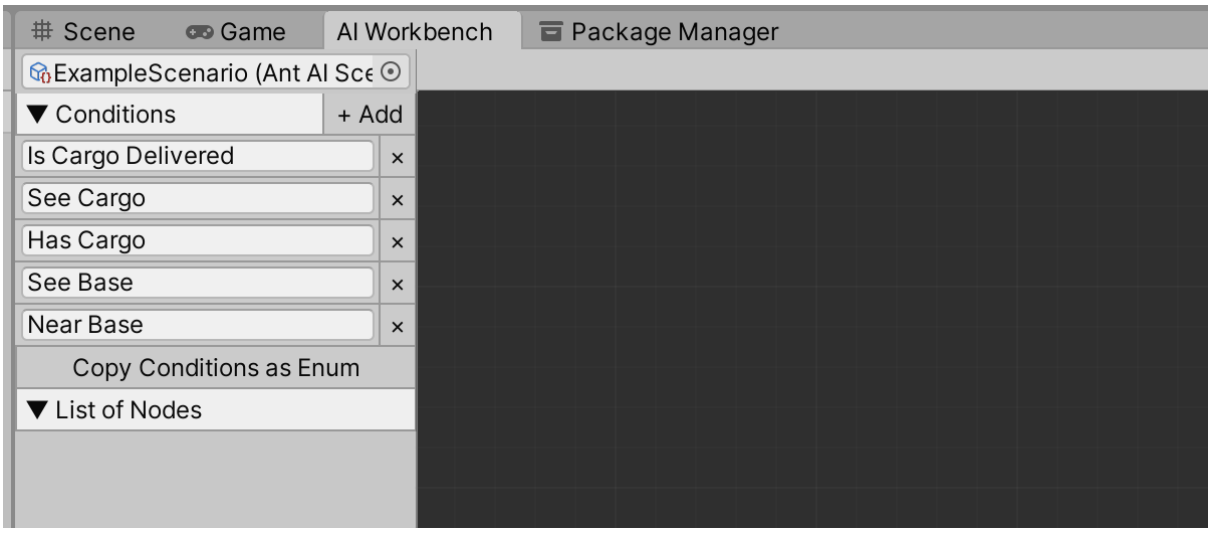


Select the created *ScriptableObject* and in the **“Inspector”** window click the **“Open AI Workbench”**. Arrange **“AI Workbench”** window in a way convenient for you.

Conditions

Conditions is a list of logical variables that can take only two values: *“true”* or *“false”*. Conditions are the basis for creating a scenario and are used to describe the **“World State”**, as well as to describe the input and output data for actions.

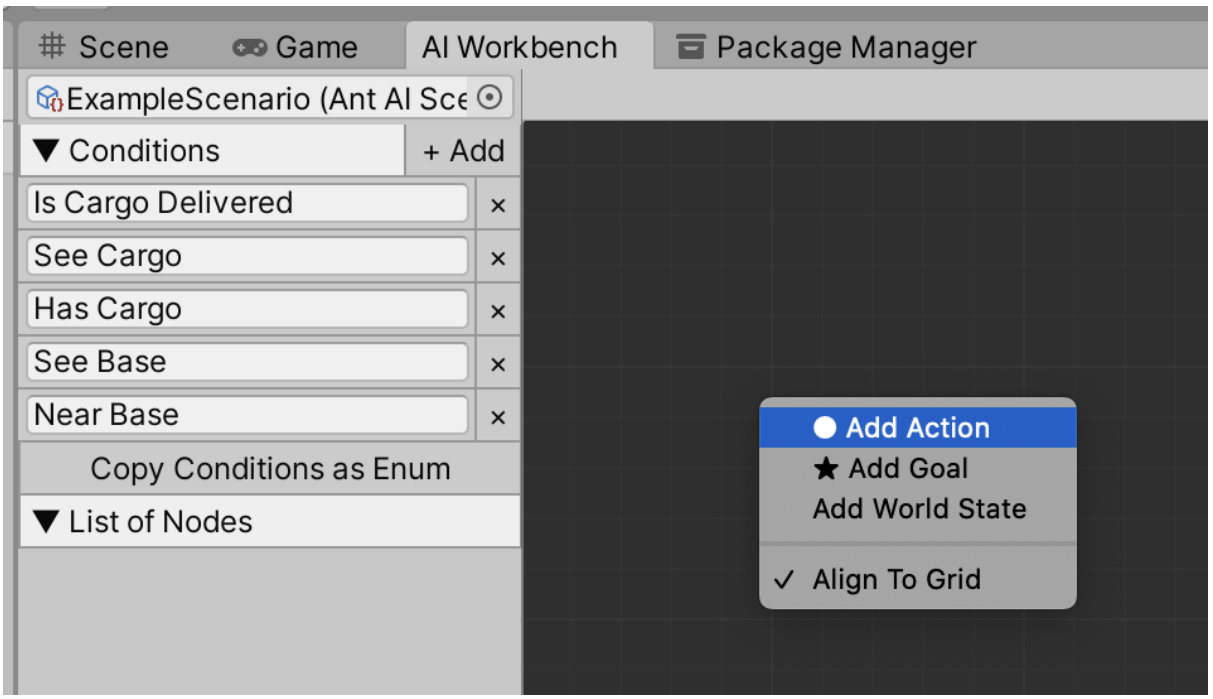
For example, add the following conditions: **“Is Cargo Delivered”**, **“See Cargo”**, **“Has Cargo”**, **“See Base”**, **“Near Base”** — this list of conditions will allow us to create a simple scenario for a delivery unit.



Actions

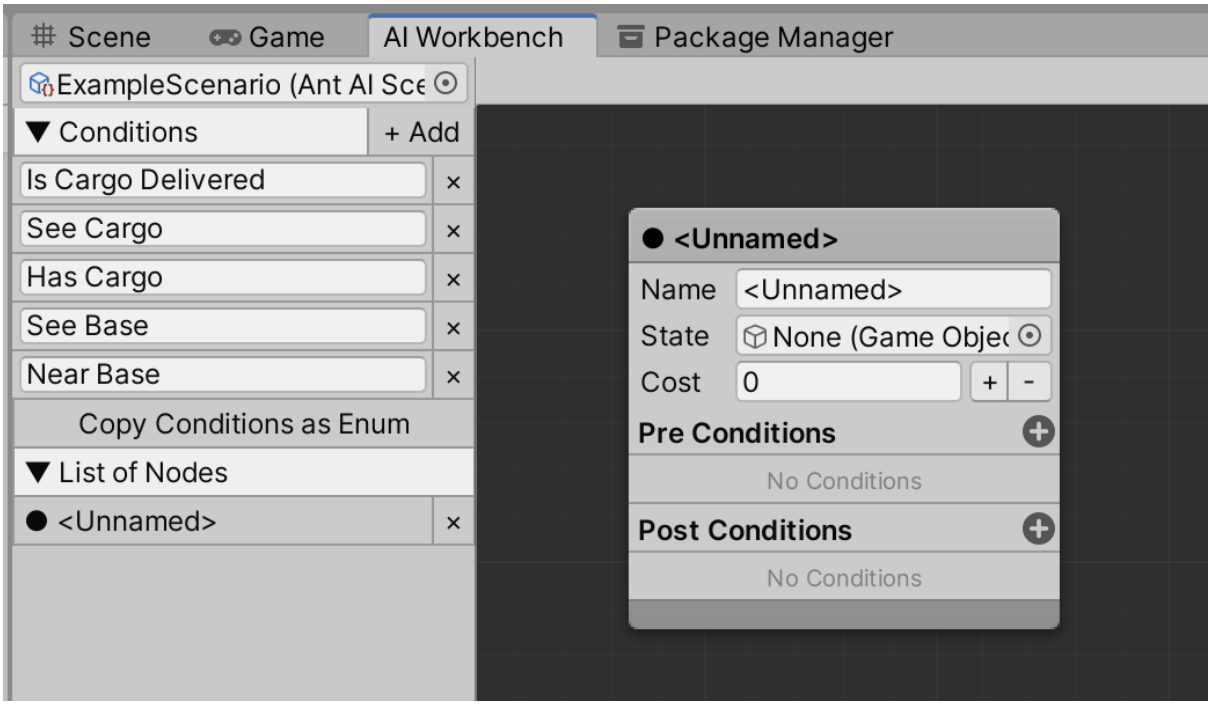
Actions are what we can perform, for example: **“Load the cargo”**, **“Deliver the cargo”**, **“Unload the cargo”**.

To create a new action, open the context menu by clicking on the background of the editor and select **“Add Action”**.



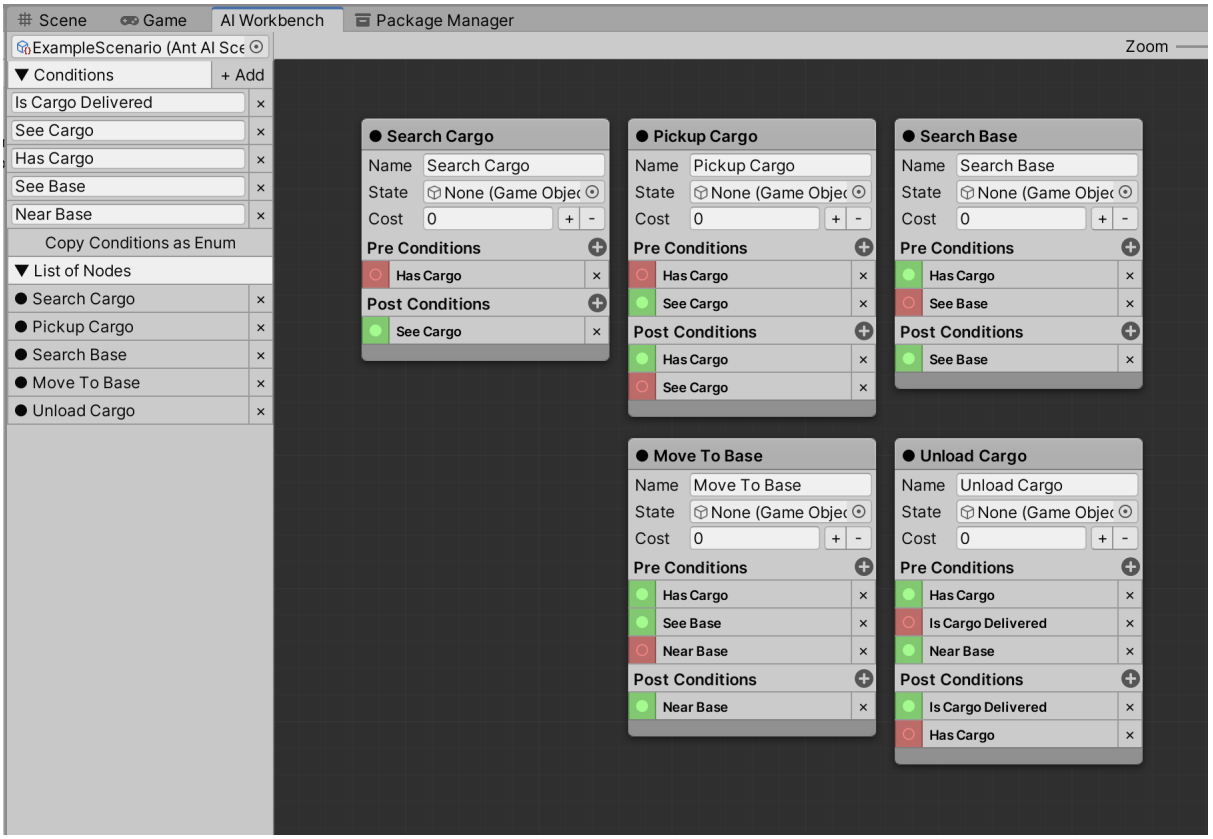
After the action is added, a new card will appear in the editor window. You can freely rearrange cards in the editor window.

To delete a card, select it and open the context menu for it, select the **“Delete”** item.



Actions should be given a name, a state script should be attached (optional), the cost of use should be indicated, as well as a list of conditions **(Pre Conditions)** that must be at peace so that the planner can use this action and a list of conditions **(Post Conditions)** that will be changed after perform this action.

Create some actions for our example as in the screenshot below.



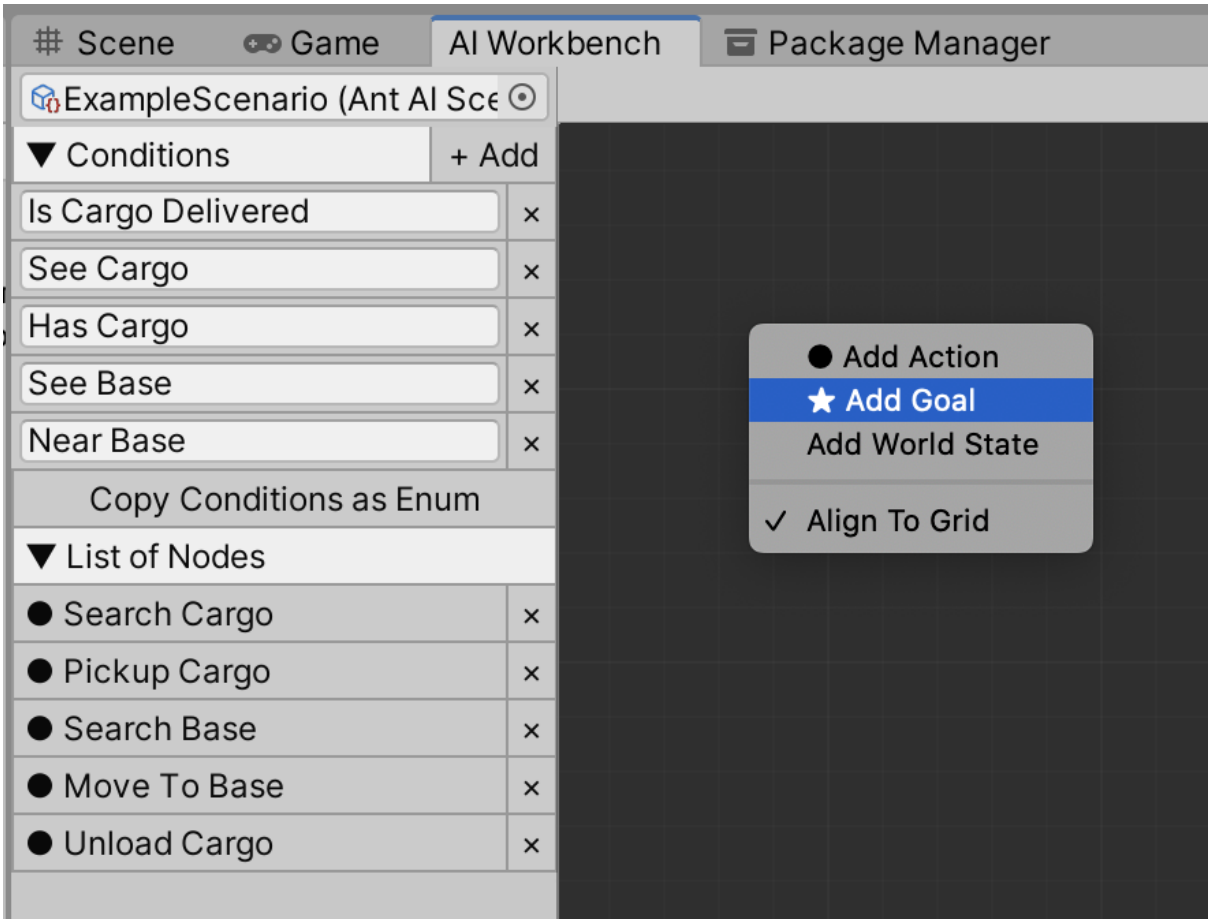
If you carefully consider the created cards, you can find the logic: if the unit has no cargo, then it will begin to look for it. If a unit sees the cargo, then it will approach it and load, after which it will look for the base if it is not visible, and when it sees, it will approach and unload the cargo.

But the planner will not be able to build an action plan if he does not know about his goal.

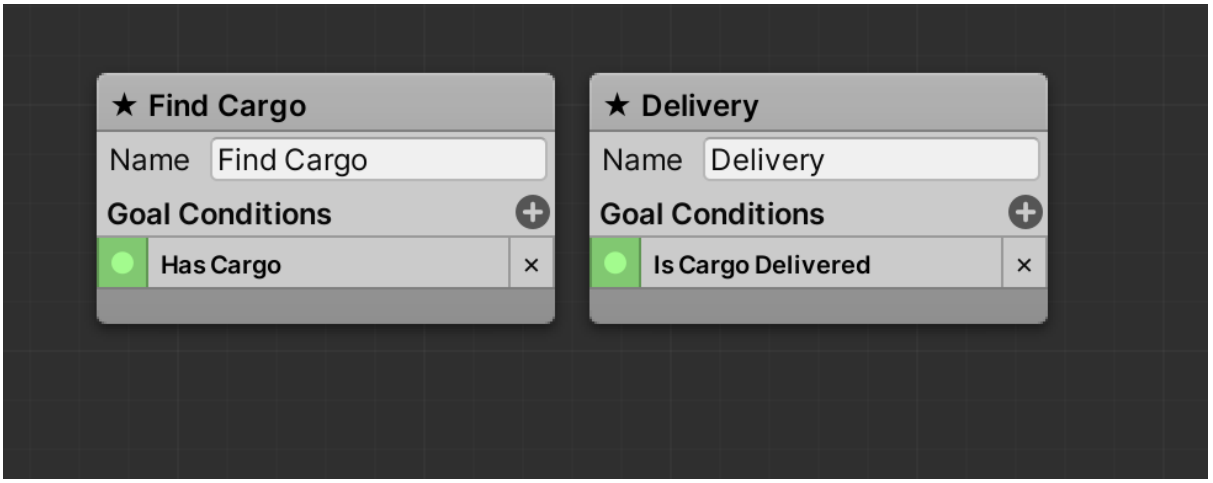
Goals

The **Goal** is a special card that lets to know the planner what it should to achieve. If there are no goals in our scenario list, then the planner will not be able to build an action plan.

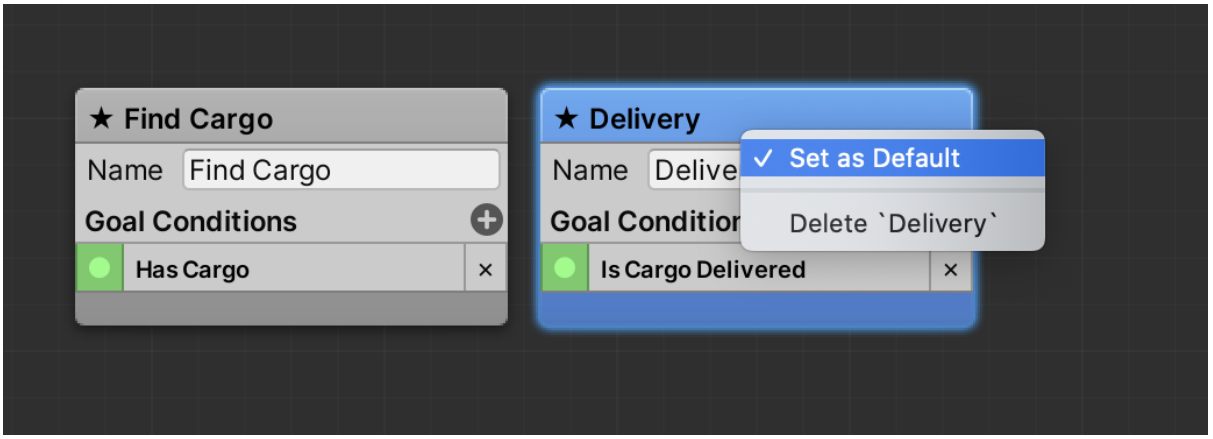
To create a goal, call the context menu in the editor window and select **“Add Goal”**.



Create two goals for our scenario as in the screenshot below. Make the goal **“Find Cargo”** as the default — this means that this goal will be active by default. If necessary, you can switch goals from the code, if you need to change the strategy of the planner.



To set the **“Find Cargo”** goal as the default goal, select the card you need and in the context menu for it, select the **“Set as Default”** menu item. The default goal will be displayed in blue.



In the goal card **“Find Cargo”** we indicated that **“Has Cargo”** should be equal to **“True”**, so the planner will try to draw up an action plan in order to achieve this condition.

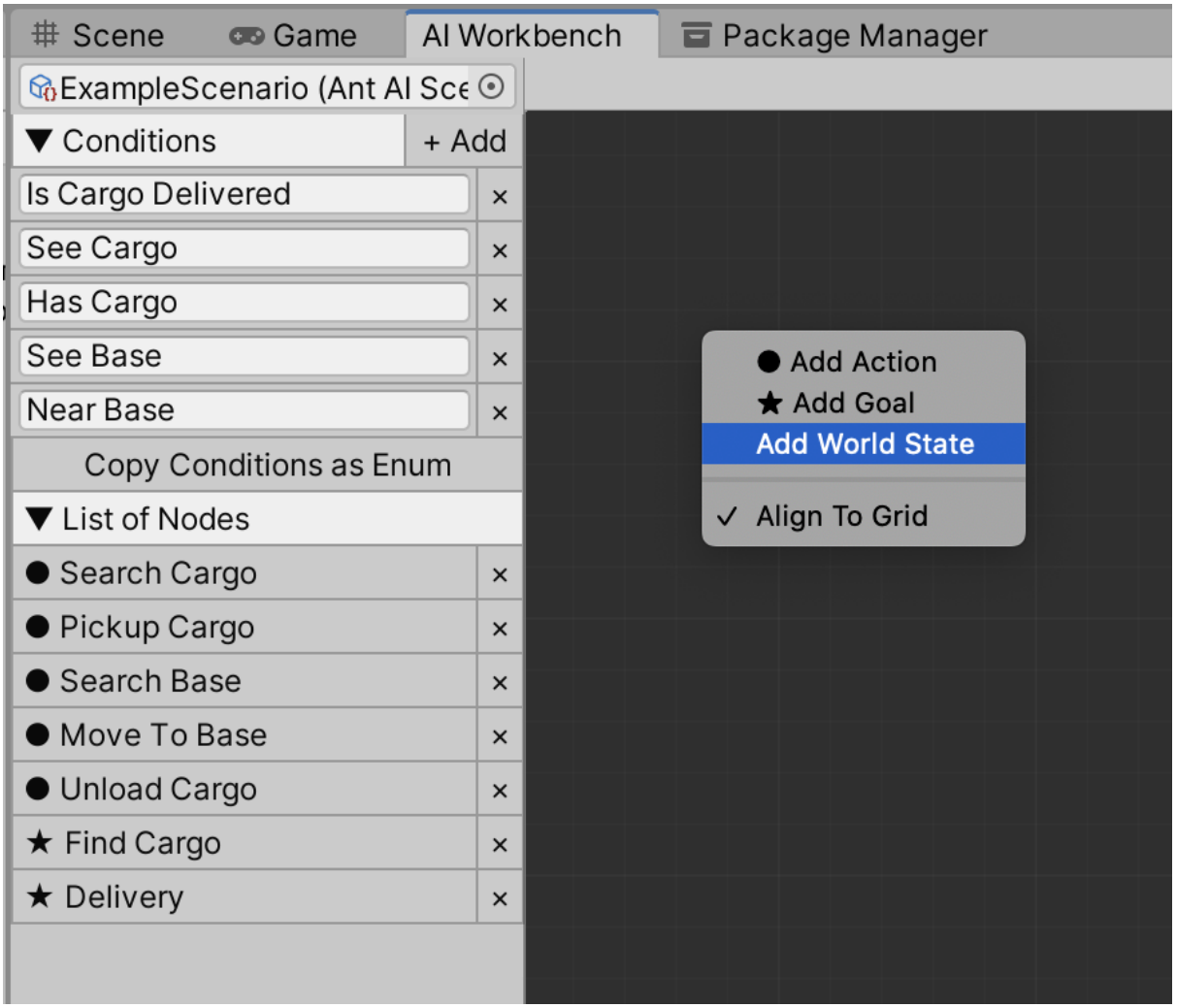
World State

The **World State** is a list of conditions describing the current state of a game object. The state of the world allows the planner to understand the current situation and, based on this, the planner builds an action plan that changes the state of the world to the desired.

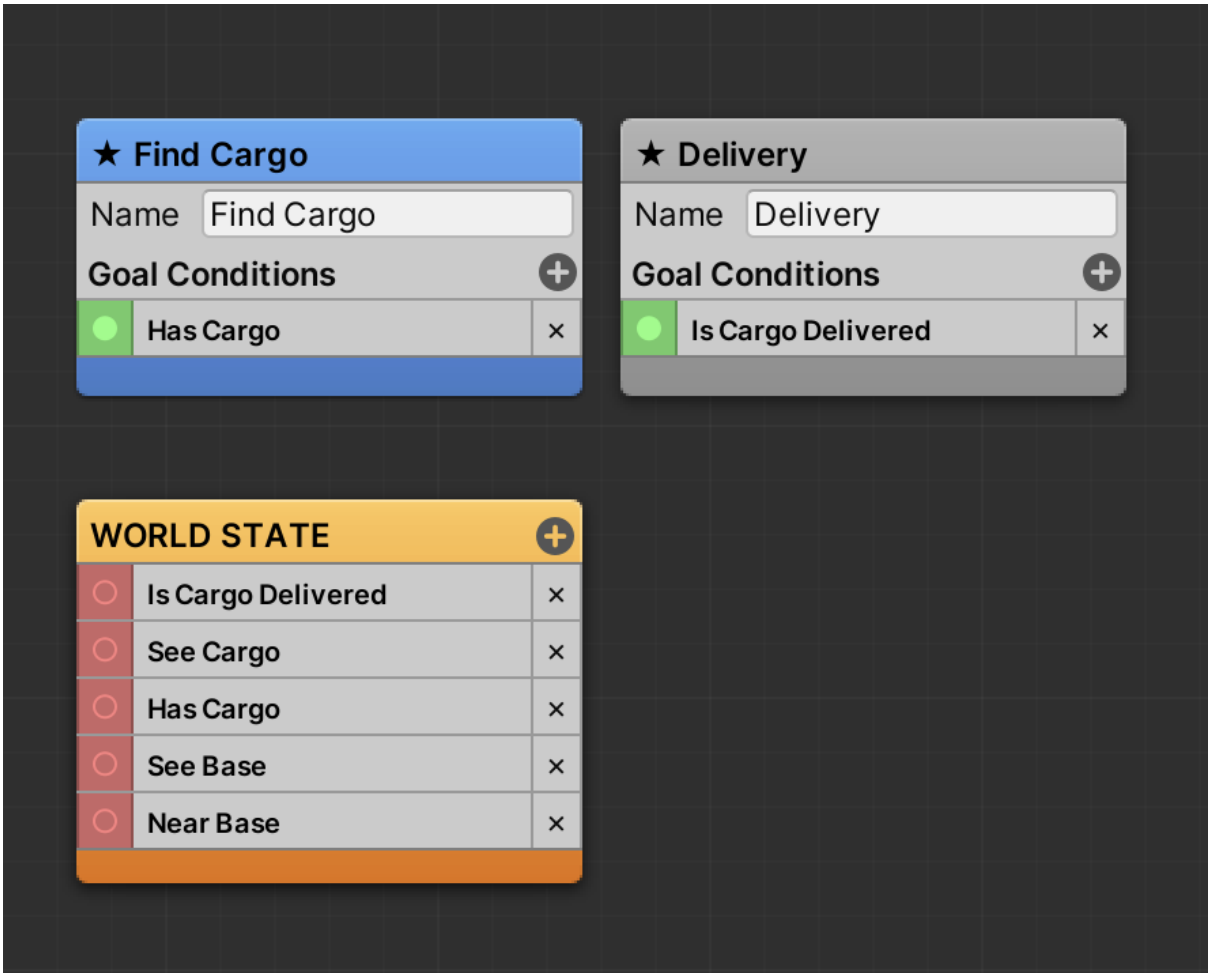
The state of the world is not a complete description of the game world, it is usually a state of the world for a specific game unit. Each unit and its scenario may have a different set of conditions describing only their state of the world.

The state of the world is built on the basis of the unit's sensors: vision, touch, hearing, inventory contents and e.t.c. How the state of the world for units will be built is up to you.

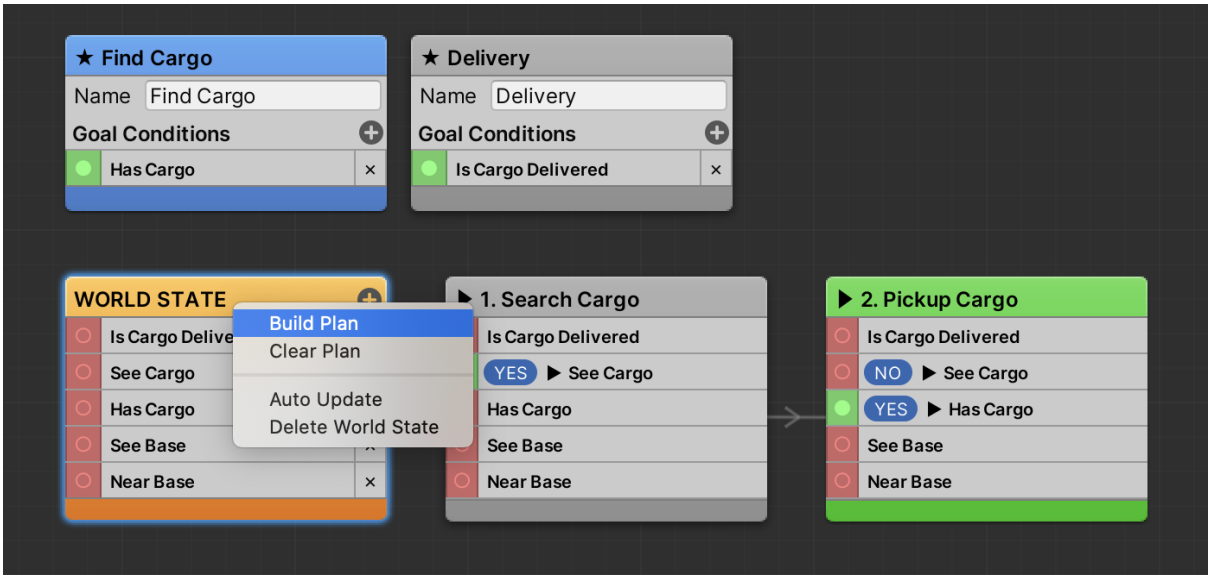
To create a test state of the world for testing the scenario, call the context menu and select the menu item **"Add World State"**.



Add all or several conditions to the world state card. Our test condition should look like the image below.



To build a plan from the current state, select the **“World State”** card and select the **“Build Plan”** menu item in the context menu. Or you can enable option Auto Update to rebuild plan automatically when conditions is changed.



We set the current goal of **“Find Cargo”** - the task of the planner to set the condition **“Has Cargo”** to **“true”**. The planner built a plan of two actions: **“Search Cargo”** and **“Pickup Cargo”** — since the sequential execution of these actions will change the specified condition to the value we need.

The conditions that changed during each action are marked in bold and have the additional tag **“YES”** or **“NO”** — depending on how the value has changed.

The green color of the last action means that it was this action that led to success and the goal was achieved. In cases where it is impossible to achieve the goal, the last card may be red, which means that the planner cannot find solutions to achieve the goal. But even a failed plan can be useful, because it can contain actions that the state of the world can change and further updates to the plan will lead to success.

You can experiment with the state of the world by turning ON or turning OFF different conditions and rebuilding the plan to see how the scenario will behave in different situations.

[Important] The state of the world created inside the scenario is used exclusively for testing and does not affect the work of the planner using this scenario inside the game.

Implementation

This library provides two types of implementation: standard and custom.

Standard implementation includes a set of components that connects the game object with a planner, scenario, as well as implement the work of actions.

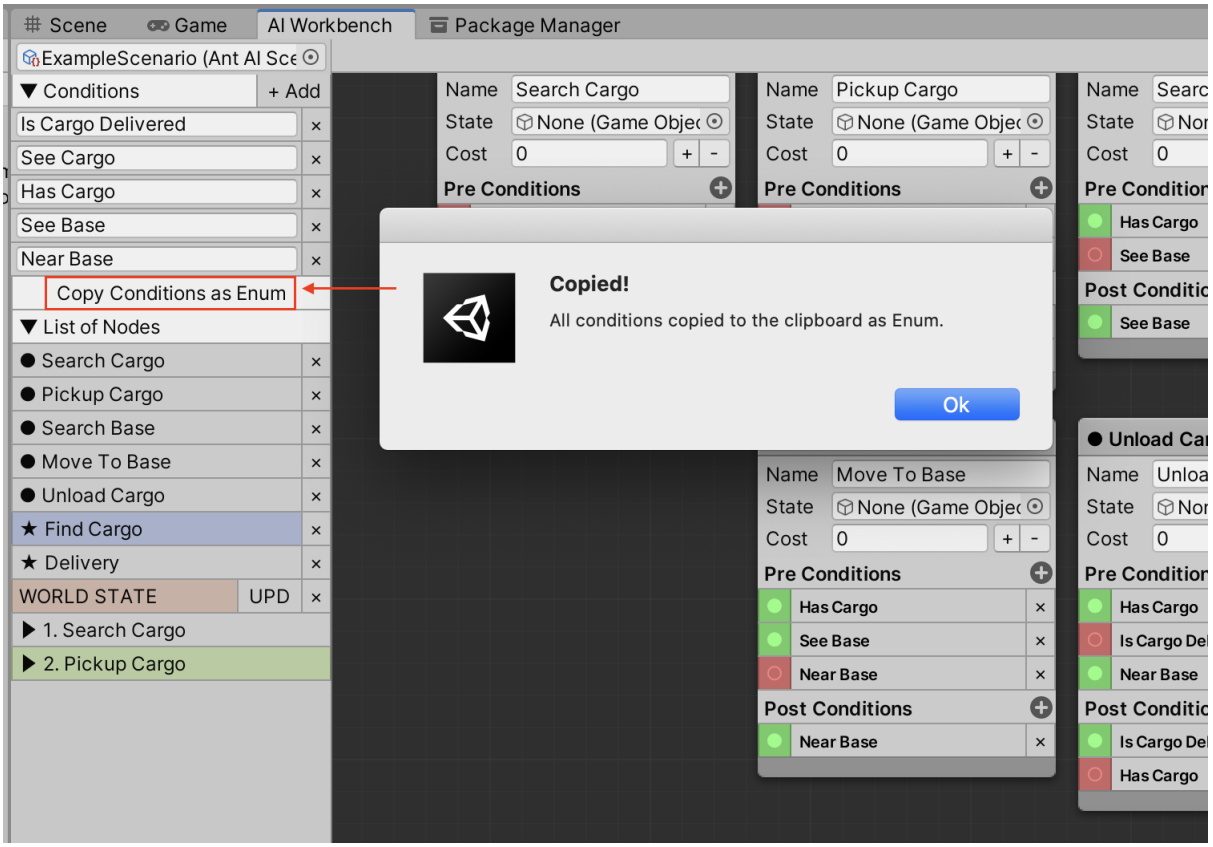
If for some reason you are not satisfied with the standard implementation of the planner and / or you know how you can implement the work of states / actions for your project better — you can use custom implementation where you work with the "raw" data of the planner.

Standart Implementation

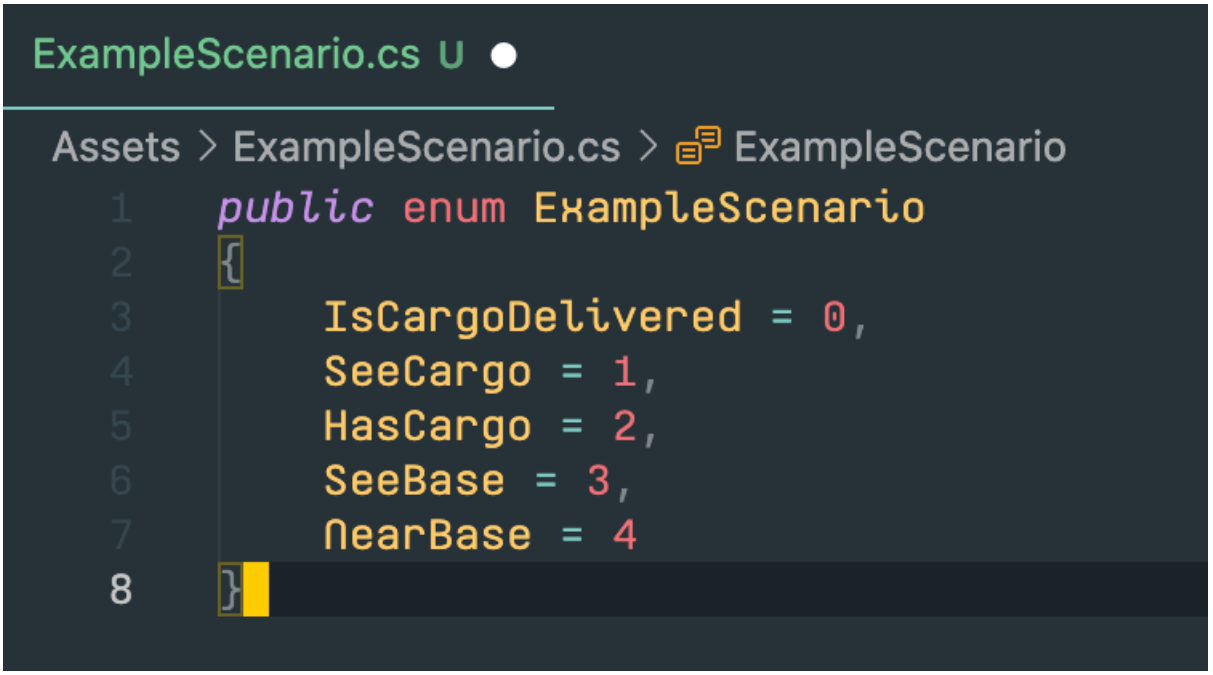
Standard implementation allows you to bind scripts to actions (states) of an object through an **”AI Workbench”**. In this section, we will look at the simplest example of how this can be done.

(The source code for this example can be found in the folder:
Anthill/Examples/DefaultUse)

First of all, press button “Copy Conditions as Enum” in the AI Workbench.



This is copies all Conditions into clipboard as Enum. Create new *.cs file and paste there copied code from Clipboard:

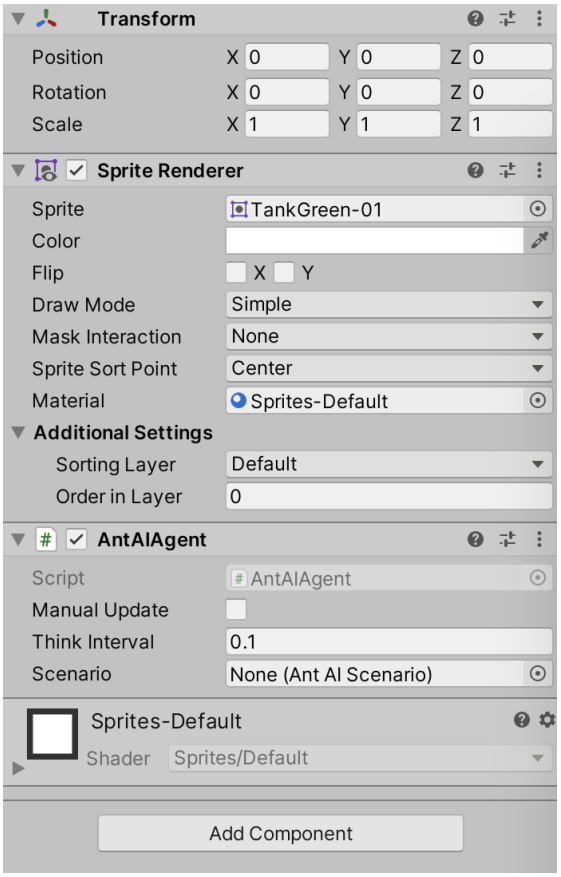


This is allows you to abandon string names for conditions in the code, which eliminates the possibility of making mistakes and it will be easier for you to update and add conditions if

you have deleted or changed them, since after updating everything will be highlighted with errors and you can easily correct the changes in the code after the update of the enum.

1. Create a new scene and add to it any object that will be a game unit.

2. Attach the components to the unit object: **SpriteRenderer** and **AntAIAgent** as in the image below.



AntAIAgent is a standard component of the **Ant AI** library that implements the relationship between the game object, scenario, and planner.

3. In the **Scenario** field of the **AntAIAgent** component, set the scenario (ScriptableObject) that we created in the previous sections of this document.

The **Manual Update** property allows to prohibit automatic updating of the plan, in which case you will need to manually call the `Execute(deltaTime, timeScale)` method for **AntAIAgent** to update the plan. This feature will be useful, for example, if your game is turn-based and there is no need to rebuild the plan for each game tick.

The **Think Interval** property allows you to set the interval between plan updates in seconds. The process of building a plan is not very complicated, but before building a plan, **AntAIAgent** calls the unit's sensors (ISensor) to update the state of the world. If the

implementation of the sensors is complex, then it is worth choosing the optimal delay value so as not to rebuild the plan too often.

4. Now we will create the script **UnitControl.cs** - this script will be responsible for the inventory of the unit. Add this script to the unit object.

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

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

5. Next, we need to create a script **UnitSense.cs** that implements all the sensors of the unit. This script should implement the `ISense` interface so that **AntAIAgent** can automatically find this sensory organ and call the `CollectConditions()` method for it every time it needs it. This script should also be attached to the unit object.

```
// 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(ExampleScenario.IsCargoDelivered, false);
            aWorldState.Set(ExampleScenario.SeeCargo, IsSeeCargo());
            aWorldState.Set(ExampleScenario.HasCargo, _control.HasCargo);
            aWorldState.Set(ExampleScenario.SeeBase, IsSeeBase());
            aWorldState.Set(ExampleScenario.NearBase, 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. Now create a script that implements the action **“Search Cargo”**, name the script **SearchCargoState.cs** and add the following code to it:

```
// 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();
        }
    }
}
```

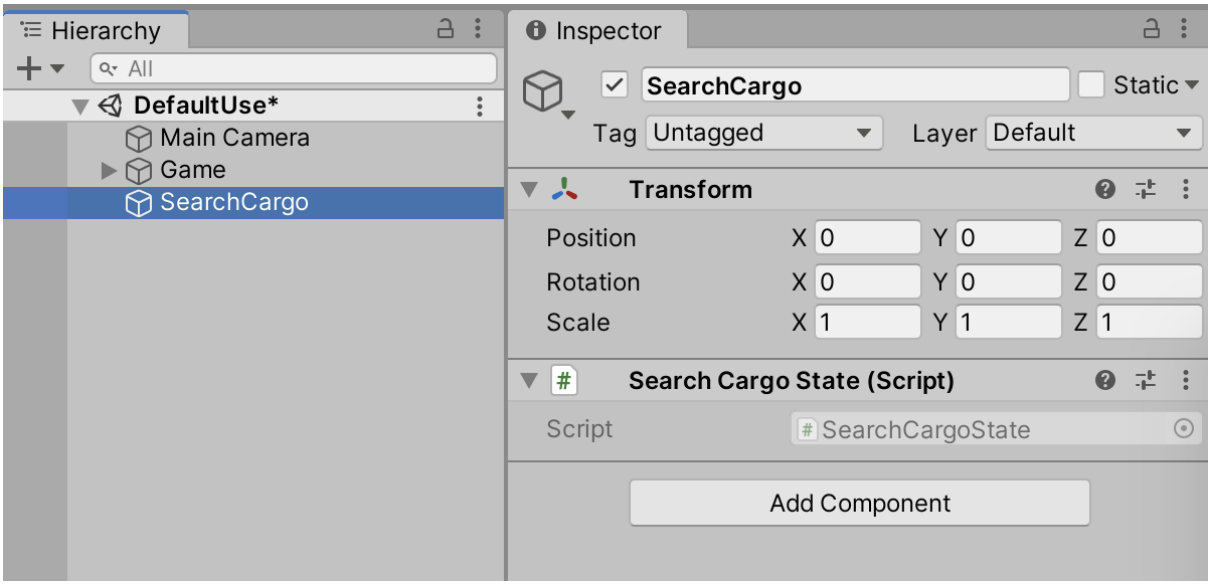
Please note that this script is inherited from **AntAIState** - this is an abstract class that implements the necessary functionality for handling states. Each state script must be inherited from this class.

AntAIState has the following useful methods that you can override to implement the behavior you need:

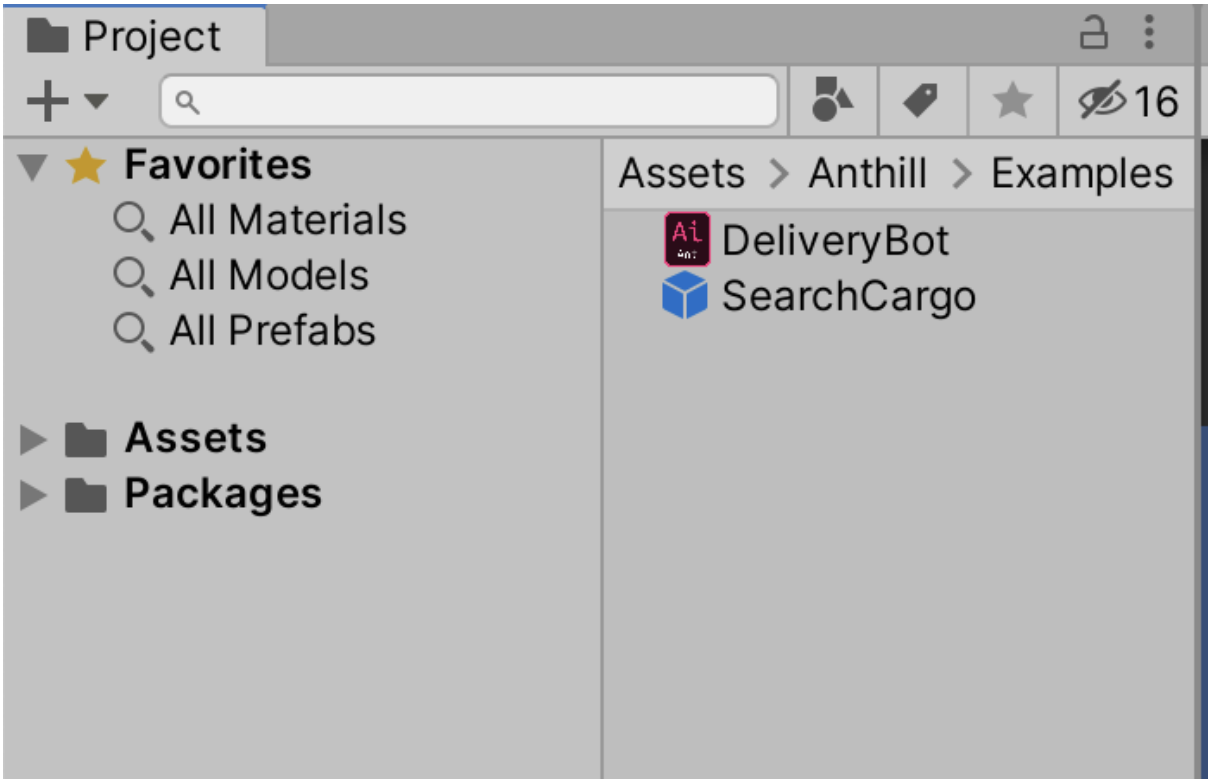
`void Create(GameObject aGameObject)` — called once when the game object is initialized and the current state is created. As an argument, a **GameObject** is passed to which **AntAIAgent** is attached, which gives you the opportunity to get all the necessary components of the game object for further work with them.

`void Enter()` — It is called once before this action is activated.
`void Execute(float aDeltaTime, float aTimeScale)` — every game tick is called while this action is active.
`void Exit()` — It is called once before this action is deactivated.

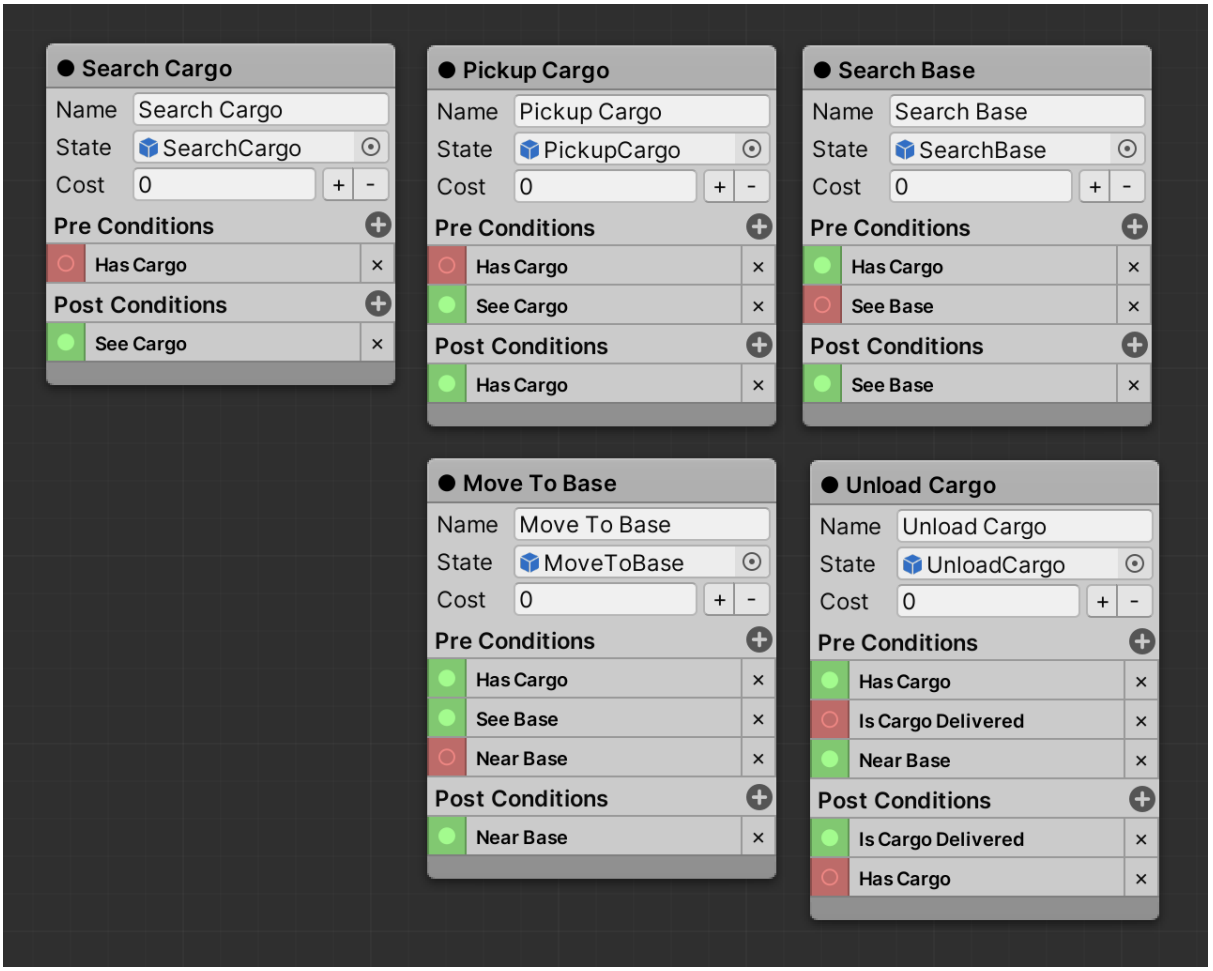
7. To add the created action script to our scenario, you need to bind it to an empty object on the stage. Create a new object on the **"Scene"** in the **"Hierarchy"** window and add the **SearchCargoScript.cs** script to it as in the image below.



8. Save the created object as a prefab by dragging it into the **"Project"** window in the same folder where the script is located, as in the image below.

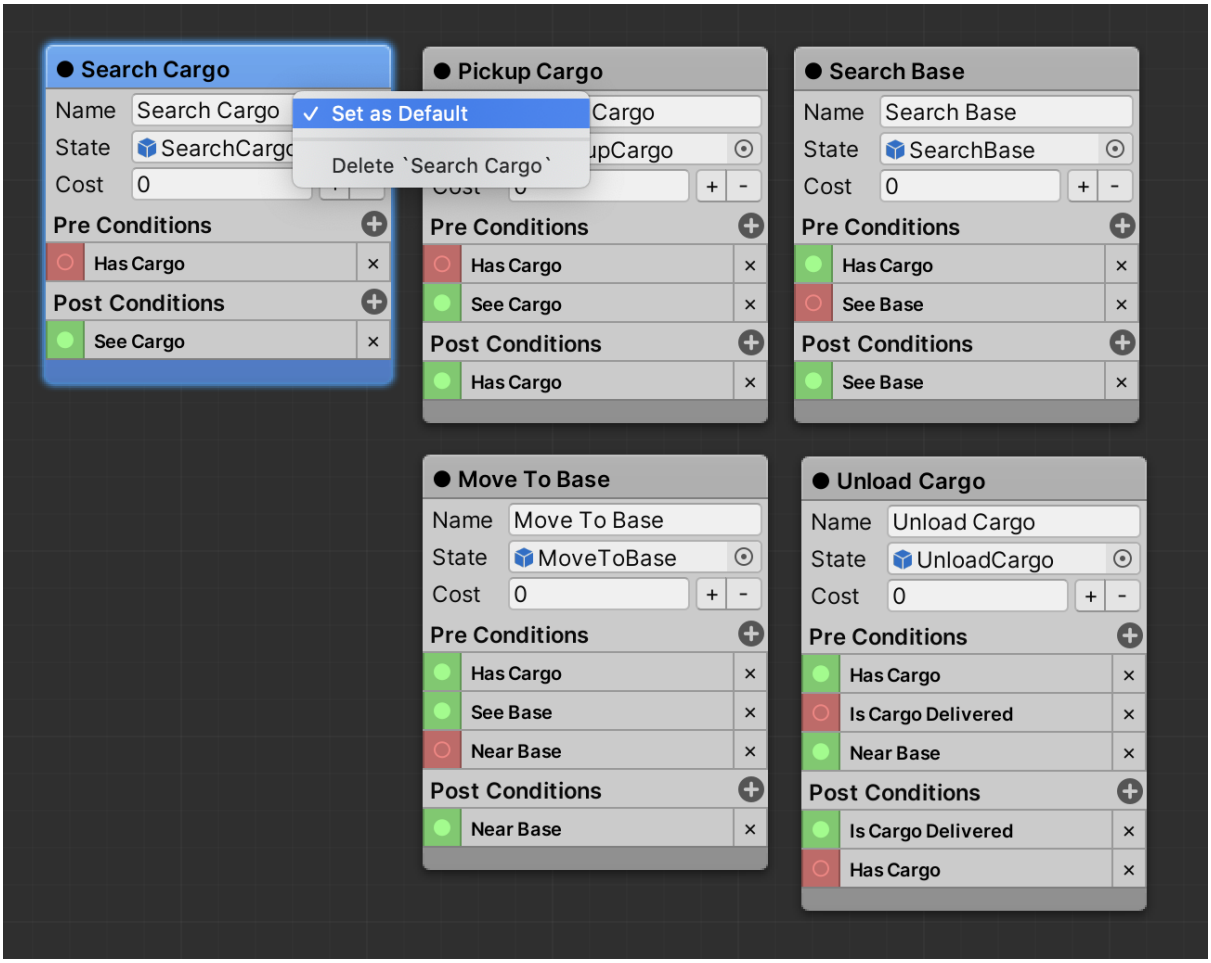


9. Open the **“DeliveryBot”** scenario in the **”AI Workbench”** and drag the newly created action into the **State** field for all action cards as in the picture below:

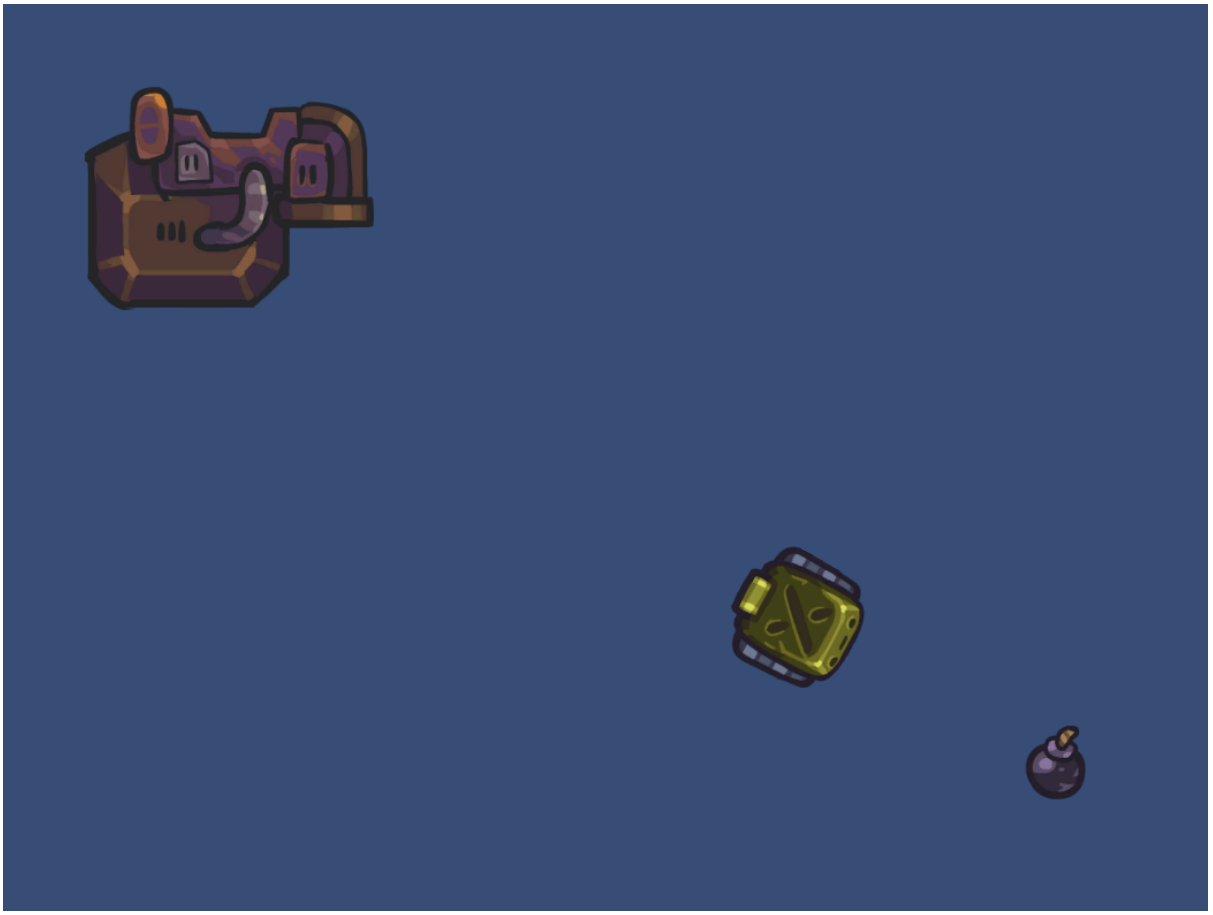


It looks a little silly. But we want to get the result as quickly as possible! **AntAIAgent** will not work until it is convinced that all actions configured in the scenario have a state script, so this is a good way to fool it ;)

10. Select the **“Search Cargo”** card and set it as the default action. Thus, we will let **AntAIAgent** know that in any incomprehensible situation, it should use this action. In your projects you can create a separate card for this, which can be called as **“Idle”** or **“Stuck”**.



11. Start the game and see how the unit headed for the cargo, which is shown in the picture below as a bomb. Everything works as it should, though after reaching the load, the bot will get stuck near it - this is because we do not have the **“Pickup Cargo”** action.



12. Now create the script **PickupCargoState.cs** and add the following code to it:

```
// 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();
    }
}
```

This action will be performed as soon as we activated it. Just like last time, create an empty object on the stage, attach the **PickupCargoState.cs** script to it and save the prefab in the **"Project"** window.

13. Create the **SearchBaseState.cs** script and add the following code to it:

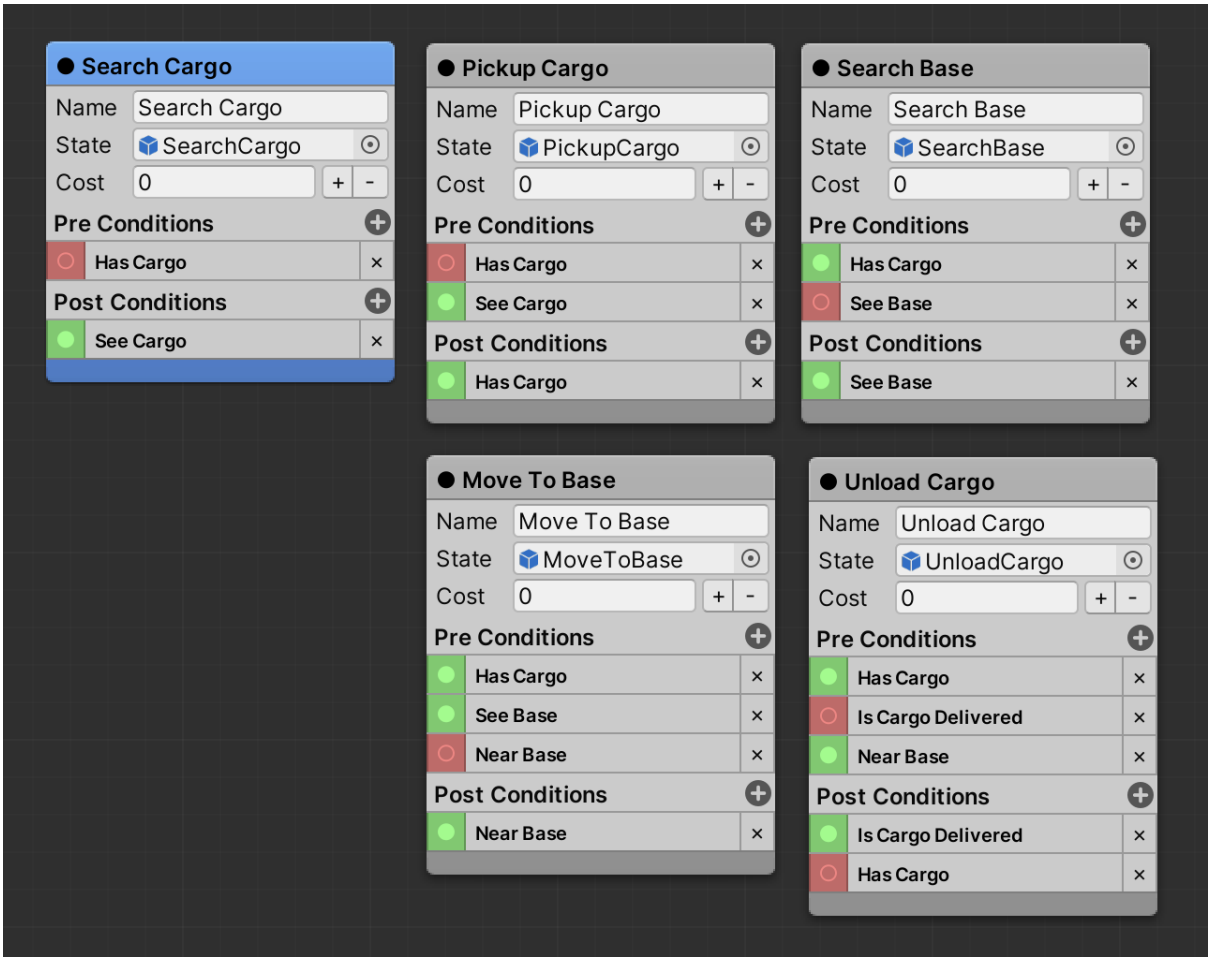
```
// SearchBaseState.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;

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();
        }
    }
}
```

Please note that this script practically repeats the code of the script **SeachCargoState.cs** - this suggests that it may be worth putting control and movement of the unit into the **UnitControl.cs** class, and in states only give commands where the unit will move and wait for the command to be executed to finish act.

Just like last time, create a new object in the **"SearchBase"** scene, attach the **SearchBaseState.cs** script to it and drag it into the **"Project"** window to save it as a file.

14. Link the newly created actions to the actions in the scenario by dragging the prefabs from the **"Project"** into the **State** fields for the action cards as in the screenshot below.



Then you can run the example and make sure that when the unit picked up the cargo, it will go to a random point near the base if the base is out of its field of vision. But when he finds a base and gets to it, then everything breaks down. This is because we have a **"SearchCargo"** state attached to the **"Move To Base"** action - we'll fix this.

15. Create a new script **MoveToBaseState.cs** and add the following code to it:

```
// 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();
        }
    }
}
```

Again, create an empty object on the stage and bind the **MoveToBase.cs** script to it and make prefab by dropping it to the **"Project"** window, so that we can then attach it to the script.

16. Create another **UnloadCargoState.cs** script and add the following code to it:

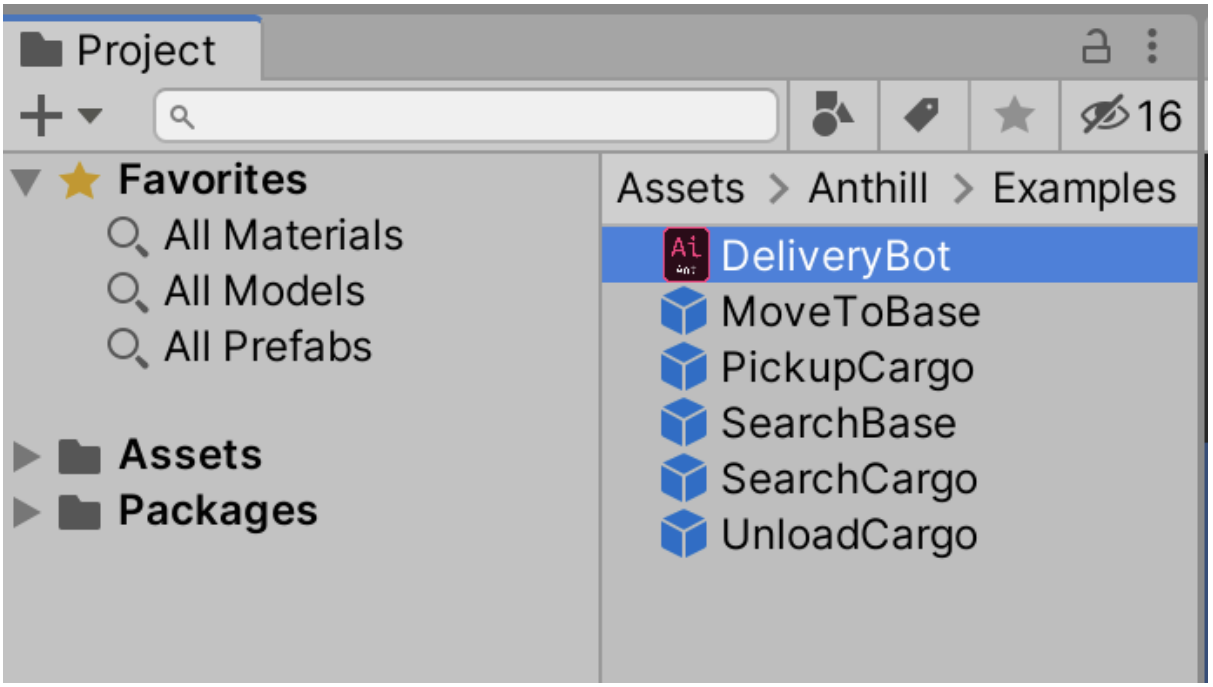
```
// UnloadCargoState.cs --
using UnityEngine;
using Anthill.AI;
using Anthill.Utils;
```

```
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();
    }
}
```

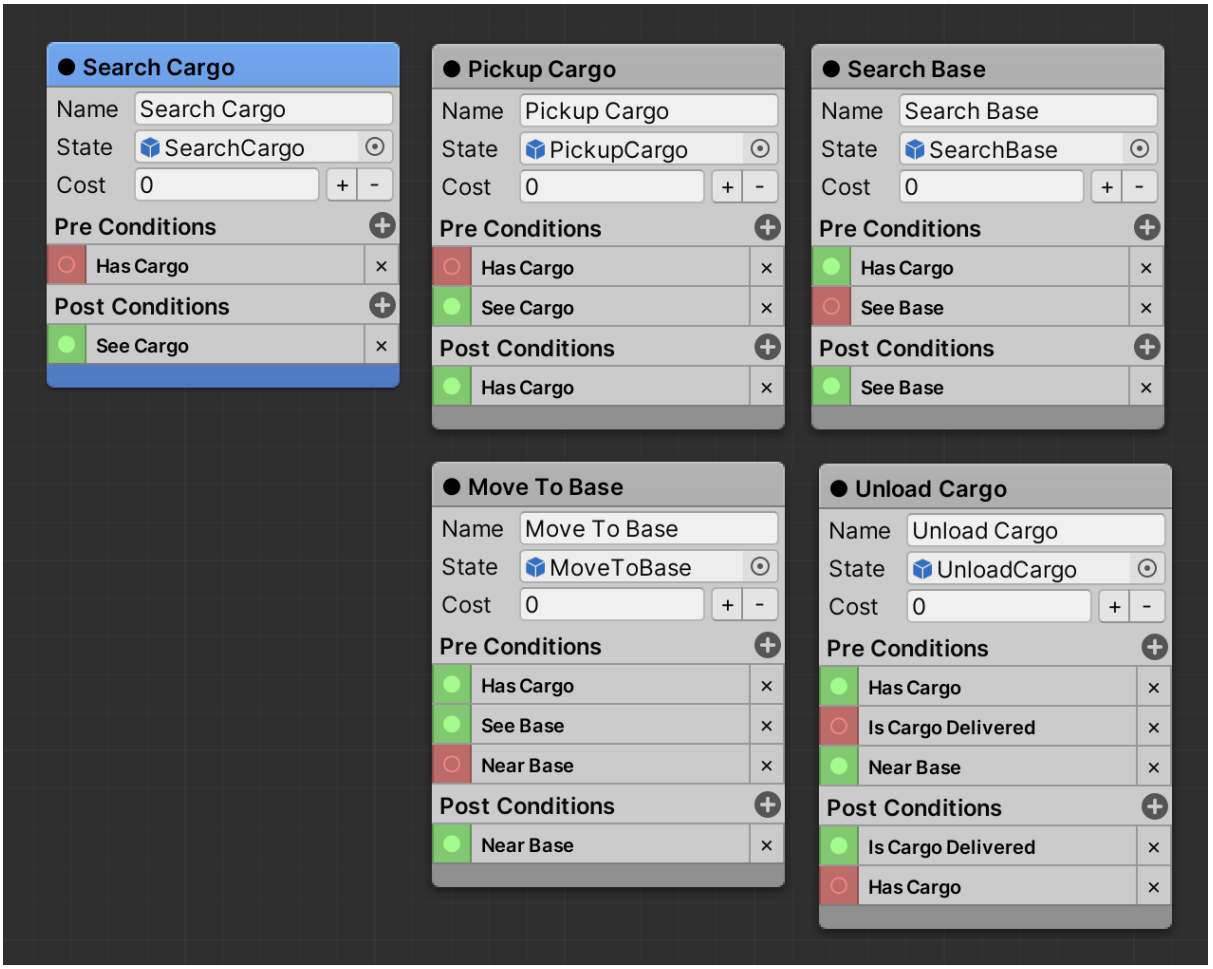
In this script, when creating an action, we will always save the reference to the cargo object on the stage in order to recreate it again when the previous cargo is delivered. This is not the right decision, but enough for this example. When the unit unloads its cargo, a new cargo will appear on the scene and this will happen indefinitely.

We create a new game object on the stage, attach the **UnloadCargoState.cs** script to it, drop it in the **"Project"** window and attach it to our script.

17. As a result, we should get a few actions:



Which should be attached to the scenario as follows:



Now you can run an example and enjoy the work of a delivery unit who will endlessly deliver cargo to the base.

This is all you need to know in order to start implementing **Ant AI** in your game!

Custom Implementation

To implement your implementation for the planner and build an action plan for your units, you need only a few things:

- Create an instance of the planner;
- Load scenario into planner (*SciptableObject*);
- Set the current state of the world;
- Set or change a goal (optional).
- Build a plan and use it to organize the further behavior of your unit.

The code for the simplest custom implementation is shown in the screenshot below:

```
// 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(ExampleScenario.IsCargoDelivered, false);
        worldState.Set(ExampleScenario.SeeCargo, false);
        worldState.Set(ExampleScenario.HasCargo, false);
        worldState.Set(ExampleScenario.SeeBase, false);
        worldState.Set(ExampleScenario.NearBase, 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(ExampleScenario.HasCargo, 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]}");
        }
    }
}
```

The source code for this example can be found here: **"Anthill/Examples/SimpleExample/"**

Feedback & Support

If you have any difficulties, you found bugs, you need advice or you have suggestions - feel free to contact with me:

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