6
Sensors and Activities

In the previous chapters on pathfinding and behavior trees, we had AI characters moving through our AI environments and changing states, but they didn't really react to anything. They knew about the navigation mesh and different points in the scene, but there was no way for them to sense different objects in the game and react to them. This chapter changes that; we will look at how to tag objects in the game so that our characters can sense and react to them.

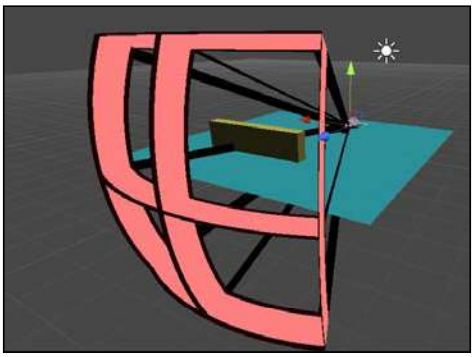
In this chapter, you will learn about:

- Sensors and tagging game objects so that they can be sensed
- AI characters that use sensors in RAIN
- Advanced configuration of sensors in RAIN
- Having AI characters react to different objects and perform different activities once they are sensed

An overview of sensing

A part of having good game AI is having the AI characters react to other parts of the game in a realistic way. For example, let's say you have an AI character in a scene searching for something, such as the player to attack them or items to collect (as in the demo in this chapter). We could have a simple proximity check, for example, if the enemy is 10 units from the player, it starts attacking. However, what if the enemy wasn't looking in the direction of the player and wouldn't be able to see or hear the player in real life? Having the enemy attack then is very unrealistic. We need to be able to set up more realistic and configurable sensors for our AI.

To set up senses for our characters, we will use RAIN's senses system. You might assume that we will use standard methods to query a scene in Unity, such as performing picking through Unity's ray casting methods. This works for simple cases, but RAIN has several advanced features to configure sensors for more realism. The senses RAIN supports are seeing and hearing. They are defined as volumes attached to an object, and the AI might be able to sense objects only inside the volume. Not everything in the volume can be sensed because there might be additional restrictions such as not being able to see through walls. A visualization illustrates this volume in the editor view to make configuring them easier. The following figure is based on the visualization of a sense in a RAIN AI:



The early versions of RAIN included additional senses, such as smell, with the idea that more senses meant more realism. However, adding more senses was confusing for users and was used only in rare cases, so they were cut from the current versions. If you need a sense such as smell for something like the ant demo we saw in *Chapter 5*, *Crowd Control*, try modifying how you use vision or hearing, such as using a visual for smell and have it on a layer not visible to players in game.

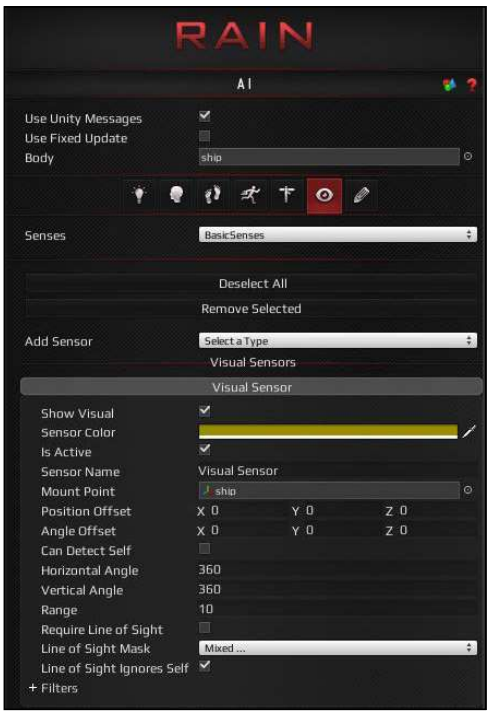
While setting up characters to sense game objects in their environment, you might think that the AI system would automatically analyze everything in the scene (game objects and geometry) to determine what is sensed. This will work for small levels but as we've seen before, we run into the problem of scaling if we have a very large scene with many objects. Larger scenes will mostly have background items that our AI doesn't care about, and we will need a more complex system to analyze all the objects to be efficient. Typically, AI systems work using a simplified version of the level, for example, how pathfinding uses navigation meshes to find a path instead of using the geometry from the level directly because it is much more efficient. Similarly, our senses don't work on everything; for an object to be sensed, it needs to be tagged.

In RAIN, the AI characters we create have an `AIRig` object, but for items we want to detect in the scene, we add a RAIN **Entity** component to them. The **RAIN** menu in Unity has a **Create Entity** option that is used to add an **Entity** component. The tags that you can set on the entities are called **aspects**, and the two types of aspects correspond to our two sensor types: visual aspects and audio aspects. So, a typical workflow to make your AI characters sense the environment is to put **Entity** components on game objects to detect, add aspects to those entities with the different tags a sensor can detect, and create sensors on your AI characters.
We will look at a demo of this, but first let's discuss sensors in detail.

Advanced visual sensor settings

We've heard stories of people setting up their sensors—especially visual ones—and starting the game, but nothing happens or it seems to work incorrectly. Configuring the senses' advanced settings can help avoid issues such as these and make development easier.

To see visual sensor settings, add a RAIN AI to a game object and click on the eye icon, select **Visual Sensor** from the **Add Sensor** dropdown, and then click on the gear icon in the upper-right corner and select **Show Advanced Settings**. The following screenshot shows the **Visual Sensor** section in RAIN:



Here are some of the properties of the sensor:

- **Show Visual / Sensor Color**: These are used to show how the sensor will look in the Unity editor, not in the game.
- **Is Active**: This flag determines whether the sensor is currently trying to sense aspects in the scene or whether it is disabled.
- **Sensor Name**: This shows the name of the sensor. This is useful when using the sensor in behavior trees, which we will see in this chapter's demo.
- **Mount Point**: This is the game object the sensor is attached to.
- **Horizontal Angle / Vertical Angle / Range**: These three define the volume of the sense; nothing outside of it will be picked up. The visualization of the sense matches these dimensions. You will want to customize these settings for different characters in your game. Unexpected behavior can occur from setting these up incorrectly.
- **Require Line of Sight**: This flag requires a line from the character to the aspect without intersecting other objects for the aspect to be seen. Without this flag, a character could appear to have X-Ray vision.
- **Can Detect Self / Line of Sight Ignores Self**: These flag if the sensor should ignore the AI character. This is important as it prevents a common problem. For example, we can have several soldier characters with a soldier aspect and then add a soldier from a different team that attacks the other soldiers. However, the attacking soldier when sensing might pick up its own aspect and try to start attacking itself, and this is definitely not what we want.
- **Line of Sight Mask**: To further help control what can be seen, layer masks can be used. These work the same as Unity's ray casting masks.

Advanced audio sensor settings

The properties for the audio sensor is similar to that of the visual sensor, except it doesn't have any line of sight properties and the volume of the sense is a radius and doesn't have vertical or horizontal angle limits. The important properties are:

- **Range**: This specifies how far the sensor can detect

- **Volume Threshold**: When listening for aspects, this is the lowest volume that the sensor can hear

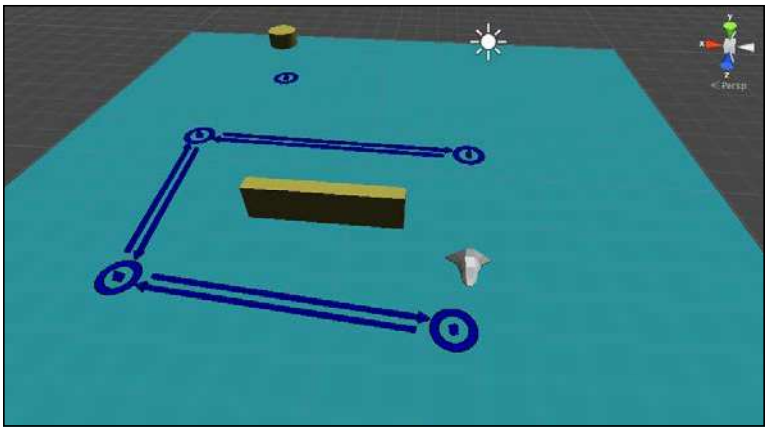
Now that we understand all of our sensor options, let's start the demo.

Using senses with RAIN

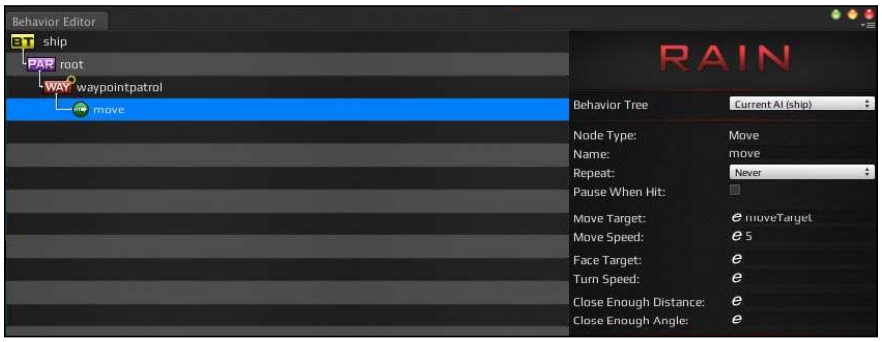
For this demo, we will use RAIN 2.14 and have a ship that patrols a path, looks for pieces of gold, and picks them up. To start, we'll use a setup similar to that of the demo in *Chapter 3*, *Behavior Trees*. You can start from there or recreate it; we just need a ship, a wall, a path, with the ground being a little larger, and the objects spread out a little.

When changing the base geometry of your game levels, you need to regenerate the navigation mesh. This is done by selecting the **Navigation Mesh** object in your scene and clicking on the **Generate NavMesh** button.

Here is our basic setup. The following image shows the starting point of our sensor demo:



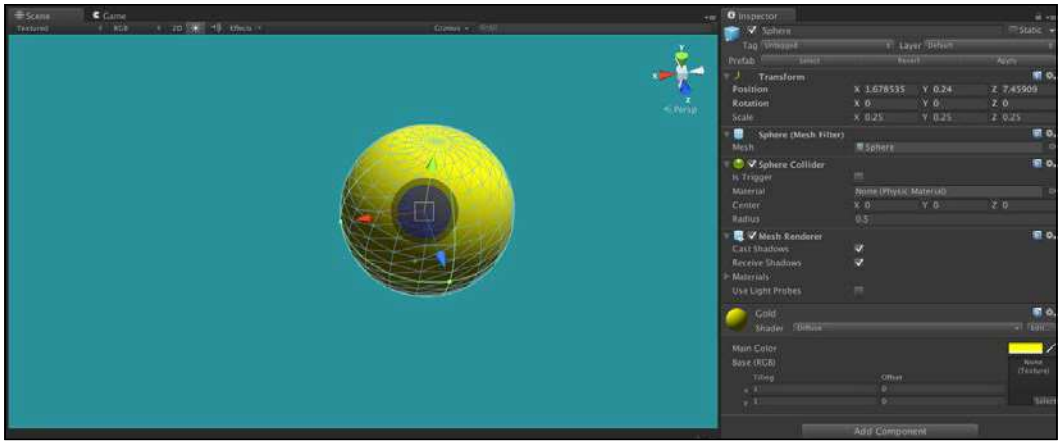
We also just need the behavior tree for the ship to only patrol the path. Set up this behavior like we did in *Chapter 2*, *Patrolling*, or if you are using the behavior tree demo, delete the timer node functionality. The new behavior tree should look like the following screenshot:



This will be the starting point of the behavior tree for our sensor demo. If you start the demo now, the ship will just keep circling the wall.

Setting up aspects in RAIN

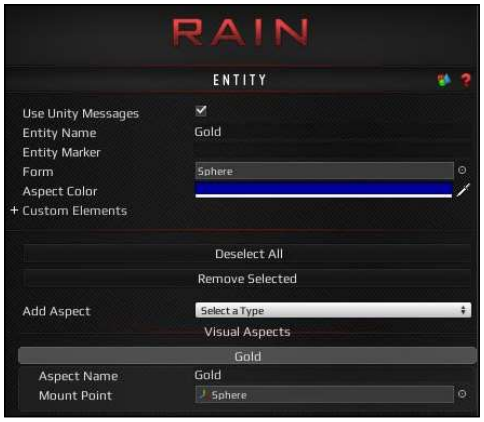
For our sensor demo, we will have the ship look for gold, which will be represented by a simple game object. Create a **Sphere** object in Unity by navigating to **Game Object** | **Create Other** | **Sphere**. Make it a little smaller by giving it **Scale** of **0.25** for **X**, **Y**, and **Z**, and change the material to a golden color. We'll be duplicating the object later so if you want duplicating to be easier, make it a prefab. This is our starting point, as illustrated in the following screenshot:



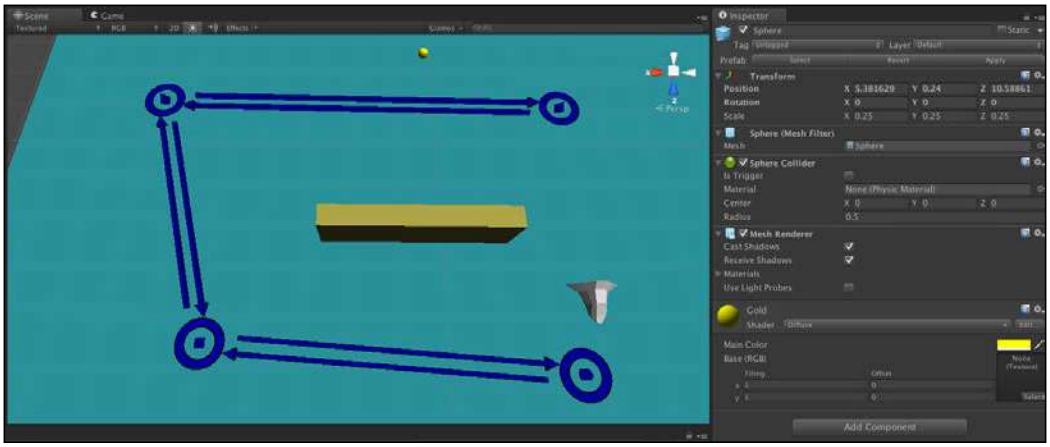
The starting point of our object (Sphere)

To have an aspect, the game object needs an **Entity** component. With **Gold** selected, go to **RAIN | Create Entity**. There are a few settings to customize, but for now just change the **Entity Name** field to **Gold**. The other important setting is **Form**, which is the game object attached to it; we can leave it to **Sphere**.

Click on the **Add Aspect** dropdown and select **Visual Aspect**. Set the aspect name to **Gold** as well. The setting should look like the following screenshot:



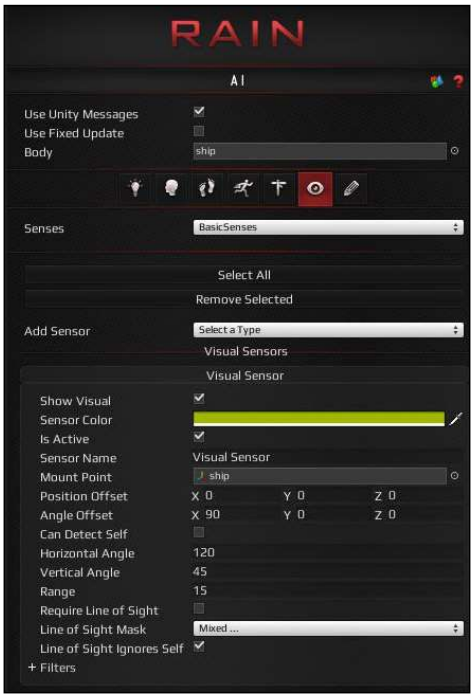
We now have an entity with a visual aspect. Create a **Prefab** tab for this **Gold** object and then add it to the opposite side of the wall as the ship. The scene should look like the following screenshot:



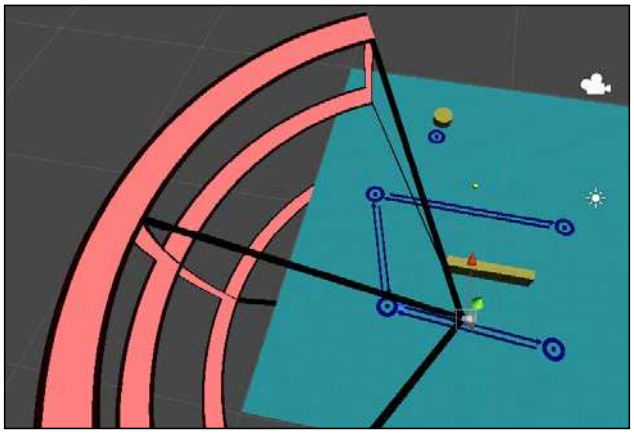
A sensor demo with gold

Setting up a visual sensor in RAIN

We have the gold aspect; next we need a visual sensor. Select **Ship AI**, click on the eye icon for the sensors tab, and from the **Add Sensor** dropdown, select **Visual Sensor**. Go to the **Advanced Settings** (selecting the gear icon) icon and adjust the horizontal and vertical angles as well as the range until the sensor can see a bit in front of the ship. Typically, you will make these very large so that the character can see most of the level. For this demo, the sensor values are **120** for **Horizontal Angle**, **45** for **Vertical Angle**, and **15** for **Range**. Also, check the **Require Line of Sight** option so that the ship can't see gold through the wall. The setup should look like the following screenshot:



If you run the demo now, you will see the ship moving with the sensor (in **Editor View**). The ship with a visual sensor should look like the following image:



This completes setting up the sensor for our ship.

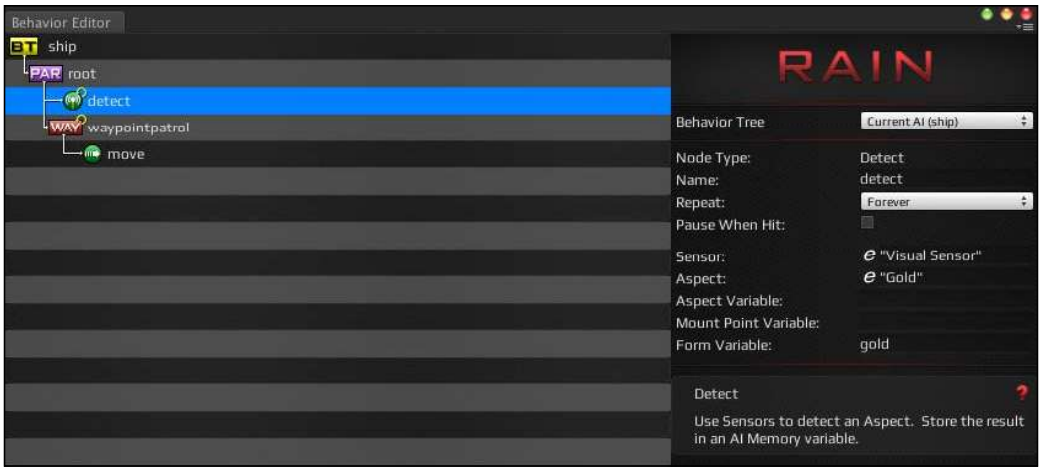
Changing activities based on sensing

We now have the ship sensing the gold as it passes by, but it still doesn't react to it. To do this, we will update the behavior tree for the ship.

The first thing we want is a detect node as the ship is moving so it can know if it sees **Gold**. Open the behavior tree for the ship and create a detect node. As the detect node will be running continuously, change its **Repeat** type to **Forever** and right-click on the root node and change its type to **Parallel**. For the detection part of the detect node to work, set the **Aspect** field to **"Gold"** and set the sensor it will be using to **"Visual Sensor**". Finally, we need to set the form of the aspect, the game object attached. Set **Form Variable** to **gold**.

The whole quotes thing in RAIN can be confusing: why some fields need quotes and others don't. This is planned to be improved in future versions of RAIN, but for now for Expressions (fields with the little e symbol) a value with quotes means the name of an object and without means the value of a variable. So in our case, **"Visual Sensor"** and **"Gold"** were both in quotes as they were referring to objects by name, but **gold** is an actual variable we store data in, so it doesn't have quotes.

Your setup should look like the following screenshot:

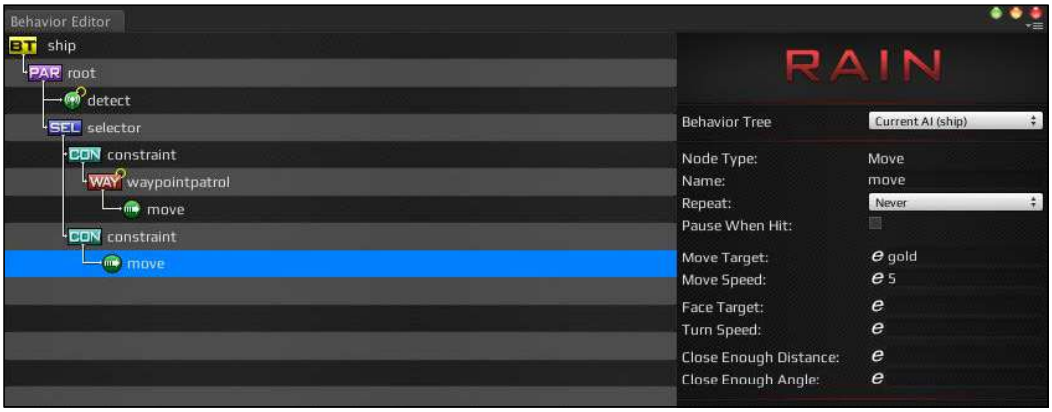


In the preceding screenshot, you can see the behavior tree with the **detect** node.

Now if you run the game, the gold will be detected, but the ship still doesn't move to it yet. To do this, we will use a selector node similar to the original behavior tree demo. Place a **selector** node under root and create a **constraint** node as a child with a **Constraint** value of **gold == null**. Then, move the original patrol node to be a child of the constraint. The setup should look the following screenshot:

The preceding screenshot shows a detection behavior tree with a **constraint** node.

Now if you run the demo, when the ship sees the gold, the gold value will not be null and it will stop moving. However, instead of stopping, we want it to move over to the gold; so, add another constraint node with the **Expressiongold != null** value and a **move** node below it that has a **Move Target** value of **gold**. Here is how the behavior tree with the **detect** node settings will look:



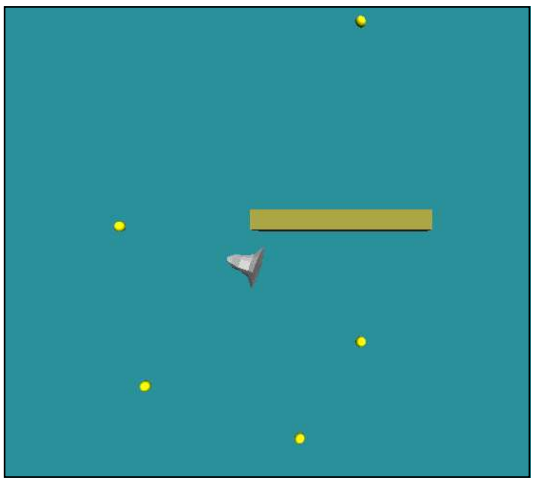
If you run the demo now, the ship will move to the gold when it sees it. However, let's change this so that the ship goes back to patrolling after the pickup. Make sure that both root and selector nodes are set to **Forever** for their **Repeat** type. Then, create a new custom action node (like in *Chapter 3*, *Behavior Trees*) and put it under the move node for the gold. Create a new class for the custom action and call it `PickUpGold`. Set its code to this:

```
using UnityEngine;
using RAIN.Core;
using RAIN.Action;

[RAINAction]
public class PickUpGold : RAINAction
{
    public PickUpGold()
    {
        actionName = "PickUpGold";
    }

    public override void Start(AI ai)
    {
```

```
            base.Start(ai);

            GameObject gold =
            ai.WorkingMemory.GetItem<GameObject>("gold");

            ai.WorkingMemory.SetItem<GameObject>("gold", null);

            Object.Destroy(gold);
        }

        public override ActionResult Execute(AI ai)
        {
            return ActionResult.SUCCESS;
        }

        public override void Stop(AI ai)
        {
            base.Stop(ai);
        }
    }
```

The important code here is in the `Start` method. We got the `gold` game object that was sensed from the memory and then erased it from memory by setting the `gold` value to `null`. Then, we destroyed the `gold` object, so it won't be sensed anymore. If you run the code now, the ship will follow the path, pick up gold when it sees it, and then go back to the path.

Next, try adding several more gold prefabs to the scene and run the demo, as shown in the following image:



Now in the demo, the ship will go and collect all the different gold pieces it sees and then return to the path.

RAIN sensor filters

If you tried running the demo with multiple gold pieces, you must have seen a small problem. The ship always goes to the first piece of gold it sees, but that might not be the closest. If it sees a distant piece from the corner of its eye, it will go straight to it even if there are ones closer to it. A quick fix for this is to add a filter to the RAIN sensors. Filters are ways to manipulate the list of sensed objects, and RAIN might have more in the future but for now, it just has one: **NearestXFilter**. Select **Visual Sensor** in the ship and set the **Size** field to **1** and select **NearestXFilter** under the **Filters** section. The following screenshot will show the settings of **NearestXFilter** on the sensor:

The **NearestXFilter** filter will send a given number of closest objects to the sensor. In our case, we just leave it to one. If you run the demo now, the ship will always pick up the gold that it can see and that is the closest to it first. This completes our ship demo.

Summary

In this chapter, we looked at how to set up sensors for our AI characters so that they can see the environment. We also saw how to tag objects with aspects so that they are visible to our AI. We also saw how to change a character's activities based on sensing, and we discussed different settings for sensors and how to tweak them. Sensors and aspects can make your game's AI more realistic, but they need to be carefully adjusted to give good results.

In the next chapter, we will look at taking our work with navigation and mind development to make our characters react more to their environments. Specifically, we will see how all of the AI we have used so far can make our AI characters adapt to different game events and create more complex AI.