3
Implementing Sensors

In this chapter, we'll learn to implement AI behaviors using the concept of a sensory system similar to what living entities have. As we discussed earlier, a character AI system needs to have awareness of its environment such as where the obstacles are, where the enemy it's looking for is, whether the enemy is visible in the player's sight, and others. The quality of AI of our NPCs completely depends on the information it can get from the environment. Based on that information, the AI characters will decide which logic to execute. If there's not enough information for the AI, our AI characters can show strange behaviors, such as choosing the wrong places to take cover, idling, looping strange actions, and not knowing what decision to make. Search for AI glitches on YouTube, and you'll find some funny behaviors of AI characters even in AAA games.

We can detect all the environment parameters and check against our predetermined values if we want. But using a proper design pattern will help us maintain code and thus, will be easy to extend. This chapter will introduce a design pattern that we can use to implement sensory systems. We will be covering:

- What sensory systems are
- What some of the different sensory systems that exist are
- How to set up a sample tank with sensing

Basic sensory systems

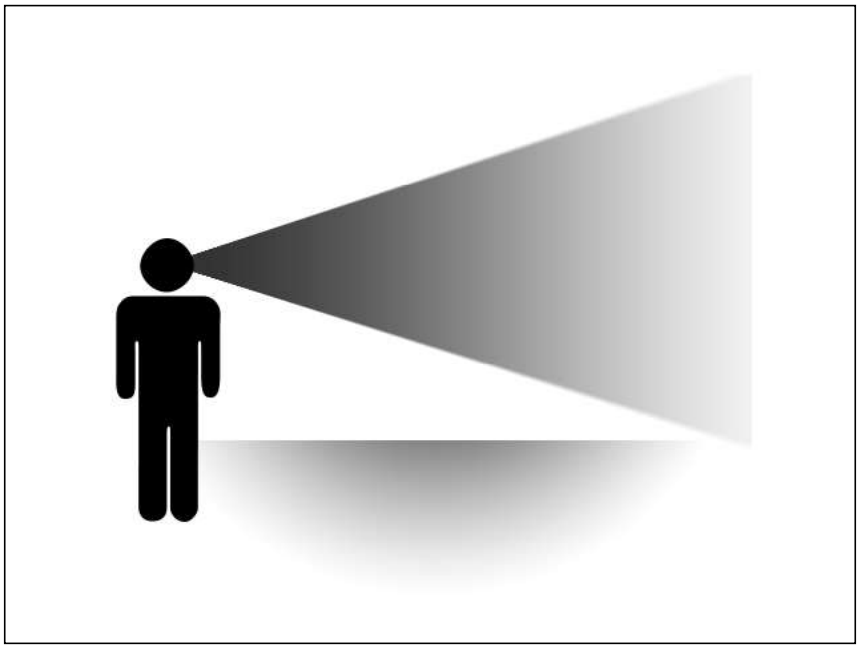
The AI sensory systems emulate senses such as perspectives, sounds, and even scents to track and identify objects. In game AI sensory systems, the agents will have to examine the environment and check for such senses periodically, based on their particular interest.

The concept of a basic sensory system is that there will be two components: `Aspect` and `Sense`. Our AI characters will have senses, such as perception, smell, and touch. These senses will look out for specific aspects such as enemy and bandit. For example, you could have a patrol guard AI with a perception sense that's looking for other game objects with an enemy aspect, or it could be a zombie entity with a smell sense looking for other entities with an aspect defined as brain.

For our demo, this is basically what we are going to implement: a base interface called `Sense` that will be implemented by other custom senses. In this chapter, we'll implement perspective and touch senses. Perspective is what animals use to see the world around them. If our AI character sees an enemy, we want to be notified so that we can take some action. Likewise, with touch, when an enemy gets too close, we want to be able to sense that; almost as if our AI character can hear that the enemy is nearby. Then we'll write a minimal `Aspect` class that our senses will be looking for.

Cone of sight

In the example provided in *Chapter 2*, *Finite State Machines and You*, we set up our agent to detect the player tank using line of sight, which is literally a line in the form of a raycast. A raycast is a feature in Unity that allows you to determine which objects are intersected by a line cast from a point toward a given direction. While this is a fairly efficient to handle visual detection in a simple way, it doesn't accurately model the way vision works for most entities. An alternative to using line of sight is using a cone-shaped field of vision. As the following figure illustrates, the field of vision is literally modeled using a cone shape. This can be in 2D or 3D, as appropriate for your type of game.



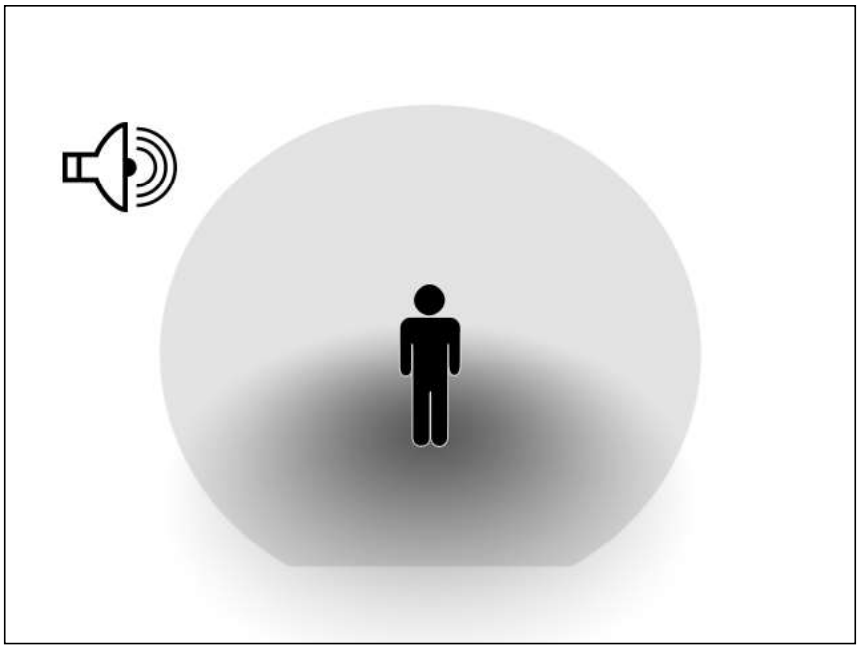
The preceding figure illustrates the concept of a cone of sight. In this case, beginning with the source, that is, the agent's eyes, the cone grows, but becomes less accurate with the distance, as represented by the fading color of the cone.

The actual implementation of the cone can vary from a basic overlap test to a more complex realistic model, mimicking eyesight. In the simple implementation, it is only necessary to test whether an object overlaps with the cone of sight, ignoring distance or periphery. The complex implementation mimics eyesight more closely; as the cone widens away from the source, the field of vision grows, but the chance of getting to see things toward the edges of the cone diminishes compared to those near the center of the source.

Hearing, feeling, and smelling using spheres

One very simple, yet effective way of modeling sounds, touch, and smell is via the use of spheres. For sounds, for example, we imagine the center as being the source, and the loudness dissipating the farther from the center the listener is. Inversely, the listener can be modeled instead of, or in addition to, the source of the sound. The listener's hearing is represented with a sphere, and the sounds closest to the listener are more likely to be "heard". We can modify the size and position of the sphere relative to our agent to accommodate feeling and smelling.

The following figure visualizes our sphere and how our agent fits into the setup:



As with sight, the probability of an agent registering the sensory event can be modified, based on the distance from the sensor or as a simple overlap event, where the sensory event is always detected as long as the source overlaps the sphere.

Expanding AI through omniscience

Truth be told, omniscience is really a way to make your AI cheat. While your agent doesn't necessarily know everything, it simply means that they can know anything. In some ways, this can seem like the antithesis to realism, but often the simple solution is the best solution. Allowing our agent access to seemingly hidden information about their surroundings or other entities in the game world can be a powerful tool to give it an extra layer of complexity.

In games, we tend to model abstract concepts using concrete values. For example, we may represent a player's health with a numeric value ranging from 0 to 100. Giving our agent access to this type of information allows it to make realistic decisions, even though having access to that information is not realistic. You can also think of omniscience as your agent being able to "use the force" or sense events in your game world without having to "physically" experience them.

Getting creative with sensing

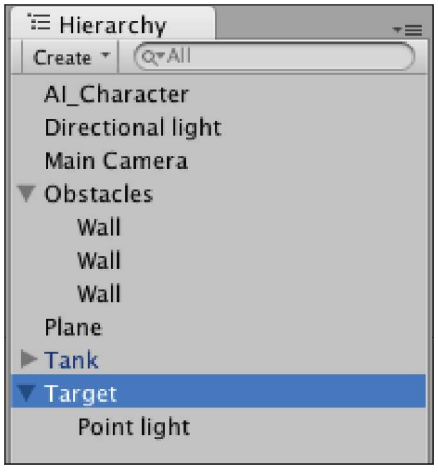
While these are among the most basic ways an agent can see, hear, and perceive their environment, they are by no means the only ways to implement these senses. If your game calls for other types of sensing, feel free to combine these patterns together. Want to use a cylinder or a sphere to represent a field of vision? Go for it. Want to use boxes to represent the sense of smell? Sniff away!

Setting up the scene

Now we have to get a little bit of setup out of the way to start implementing the topics we've discussed. We need to get our scene ready with environment objects, our agents, and some other items to help us see what the code is doing:

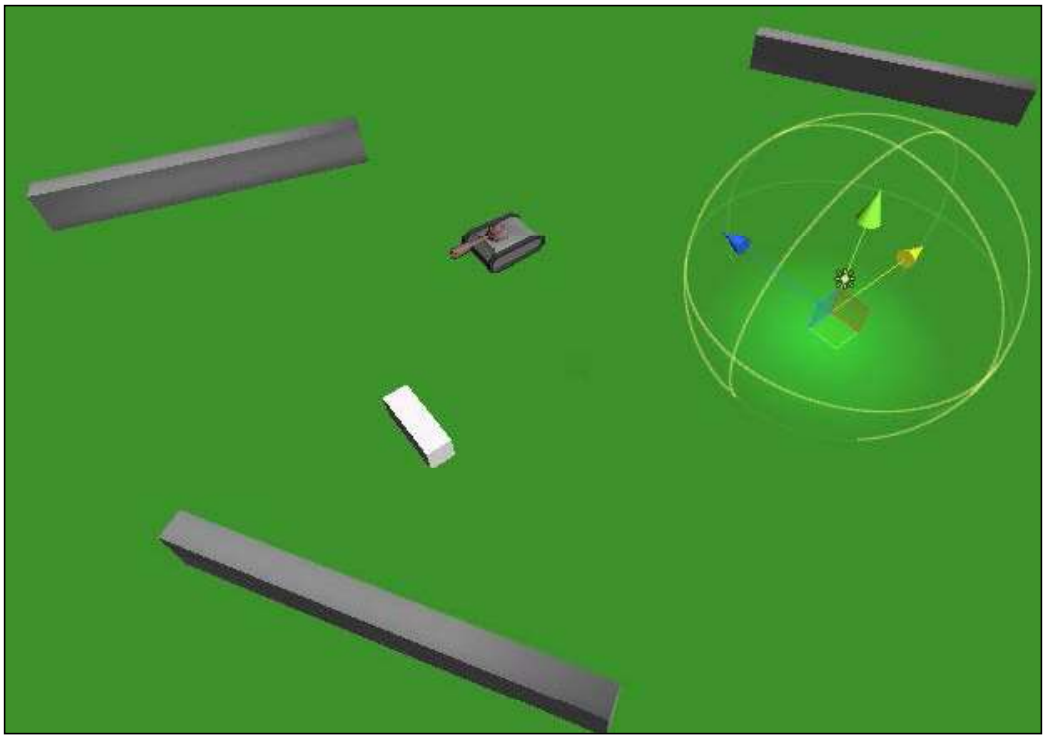
1. Let's create a few walls to block the line of sight from our AI character to the enemy. These will be short but wide cubes grouped under an empty game object called `Obstacles`.
2. Add a plane to be used as a floor.
3. Then, we add a directional light so that we can see what is going on in our scene.

We will be going over this next part in detail throughout the chapter, but basically, we will use a simple tank model for our player, and a simple cube for our AI character. We will also have a `Target` object to show us where the tank will move to in our scene. Our scene hierarchy will look similar to the following screenshot:



The hierarchy

Now we will position the tank, AI character, and walls randomly in our scene. Increase the size of the plane to something that looks good. Fortunately, in this demo, our objects float, so nothing will fall off the plane. Also, be sure to adjust the camera so that we can have a clear view of the following scene:



Where our tank and player will wander in

Now that we have the basics set up, we'll look at how to implement the tank, AI character, and aspects for our player character.

Setting up the player tank and aspect

Our `Target` object is a simple sphere object with the mesh render disabled. We have also created a point light and made it a child of our `Target` object. Make sure the light is centered, or it will not be very helpful for us.

Look at the following code in the `Target.cs` file:

```
using UnityEngine;
using System.Collections;

public class Target : MonoBehaviour {

  public Transform targetMarker;

  void Update () {
    int button = 0;
    //Get the point of the hit position when the mouse is being
// clicked.
    if (Input.GetMouseButtonDown(button)) {
      Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hitInfo;
      if (Physics.Raycast(ray.origin, ray.direction, out hitInfo)) {
        Vector3 targetPosition = hitInfo.point;
        targetMarker.position = targetPosition;
      }
    }
  }
}
```

Attach this script to our `Target` object, which is what we assign in the inspector to the `targetMarker` variable. The script detects the mouse click event and then, using the raycasting technique, detects the mouse click point on the plane in the 3D space. After that it updates the `Target` object to that position in our scene.

Implementing the player tank

Our player tank is the simple tank model we used in *Chapter 2*, *Finite State Machines and You*, with a non-kinematic rigid body component attached. The rigid body component is needed in order to generate trigger events whenever we do collision detection with any AI characters. The first thing we need to do is to assign the tag `Player` to our tank.

The tank is controlled by the `PlayerTank` script, which we will create in a moment. This script retrieves the target position on the map and updates its destination point and the direction accordingly.

The code in the `PlayerTank.cs` file is shown as follows:

```
using UnityEngine;
using System.Collections;

public class PlayerTank : MonoBehaviour {
  public Transform targetTransform;
  private float movementSpeed, rotSpeed;

  void Start () {
    movementSpeed = 10.0f;
    rotSpeed = 2.0f;
  }

  void Update () {
    //Stop once you reached near the target position
    if (Vector3.Distance(transform.position,
      targetTransform.position) < 5.0f)
      return;

    //Calculate direction vector from current position to target
//position
    Vector3 tarPos = targetTransform.position;
    tarPos.y = transform.position.y;
    Vector3 dirRot = tarPos - transform.position;

    //Build a Quaternion for this new rotation vector
    //using LookRotation method
    Quaternion tarRot = Quaternion.LookRotation(dirRot);

    //Move and rotate with interpolation
```

```
        transform.rotation= Quaternion.Slerp(transform.rotation,
            tarRot, rotSpeed * Time.deltaTime);

        transform.Translate(new Vector3(0, 0,
            movementSpeed * Time.deltaTime));
    }
}
```

Properties of our tank object

The preceding screenshot gives us a snapshot of our script in the inspector once applied to our tank.

This script retrieves the position of the `Target` object on the map and updates its destination point and the direction accordingly. After we assign this script to our tank, be sure to assign our `Target` object to the `targetTransform` variable.

Implementing the Aspect class

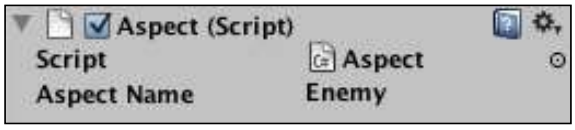
Next, let's take a look at the `Aspect.cs` class. Aspect is a very simple class with just one public property called `aspectName`. That's all of the variables we need in this chapter. Whenever our AI character senses something, we'll check against this with `aspectName` to see whether it's the aspect that the AI has been looking for.

The code in the `Aspect.cs` file is shown as follows:

```
using UnityEngine;
using System.Collections;

public class Aspect : MonoBehaviour {
  public enum aspect {
    Player,
    Enemy
  }
  public aspect aspectName;
}
```

Attach this aspect script to our player tank and set the `aspectName` property as `Enemy`, as shown in the following image:



Setting which aspect to look out for

Creating an AI character

Our AI character will be roaming around the scene in a random direction. It'll have two senses:

- The perspective sense will check whether the enemy aspect is within a set visible range and distance
- Touch sense will detect if the enemy aspect has collided with the box collider, soon to be surrounding our AI character

As we have seen previously, our player tank will have the `Enemy` aspect. So, these senses will be triggered when they detect the player tank.

The code in the `Wander.cs` file can be shown as follows:

```
using UnityEngine;
using System.Collections;

public class Wander : MonoBehaviour {
  private Vector3 tarPos;

  private float movementSpeed = 5.0f;
  private float rotSpeed = 2.0f;
  private float minX, maxX, minZ, maxZ;

  // Use this for initialization
  void Start () {
    minX = -45.0f;
    maxX = 45.0f;

    minZ = -45.0f;
    maxZ = 45.0f;

    //Get Wander Position
    GetNextPosition();
  }

  // Update is called once per frame
  void Update () {
    // Check if we're near the destination position
    if (Vector3.Distance(tarPos, transform.position) <= 5.0f)
      GetNextPosition(); //generate new random position

    // Set up quaternion for rotation toward destination
    Quaternion tarRot = Quaternion.LookRotation(tarPos -
        transform.position);

    // Update rotation and translation
    transform.rotation = Quaternion.Slerp(transform.rotation, tarRot,
        rotSpeed * Time.deltaTime);

    transform.Translate(new Vector3(0, 0,
```

```
            movementSpeed * Time.deltaTime));
    }

    void GetNextPosition() {
      tarPos = new Vector3(Random.Range(minX, maxX), 0.5f,
          Random.Range(minZ, maxZ));
    }
}
```

The `Wander` script generates a new random position in a specified range whenever the AI character reaches its current destination point. The `Update` method will then rotate our enemy and move it toward this new destination. Attach this script to our AI character so that it can move around in the scene.

Using the Sense class

The `Sense` class is the interface of our sensory system that the other custom senses can implement. It defines two virtual methods, `Initialize` and `UpdateSense`, which will be implemented in custom senses, and are executed from the `Start` and `Update` methods, respectively.

The code in the `Sense.cs` file can be shown as follows:

```
using UnityEngine;
using System.Collections;

public class Sense : MonoBehaviour {
  public bool bDebug = true;
  public Aspect.aspect aspectName = Aspect.aspect.Enemy;
  public float detectionRate = 1.0f;

  protected float elapsedTime = 0.0f;

  protected virtual void Initialize() { }
  protected virtual void UpdateSense() { }

  // Use this for initialization
  void Start () {
    elapsedTime = 0.0f;
    Initialize();
  }

  // Update is called once per frame
```

```
    void Update () {
      UpdateSense();
    }
}
```

The basic properties include its detection rate to execute the sensing operation as well as the name of the aspect it should look for. This script will not be attached to any of our objects.

Giving a little perspective

The perspective sense will detect whether a specific aspect is within its field of view and visible distance. If it sees anything, it will take the specified action.

The code in the `Perspective.cs` file can be shown as follows:

```
using UnityEngine;
using System.Collections;

public class Perspective : Sense {
  public int FieldOfView = 45;
  public int ViewDistance = 100;

  private Transform playerTrans;
  private Vector3 rayDirection;

  protected override void Initialize() {

    //Find player position
    playerTrans =

  GameObject.FindGameObjectWithTag("Player").transform;
  }

  // Update is called once per frame
  protected override void UpdateSense() {
    elapsedTime += Time.deltaTime;

    // Detect perspective sense if within the detection rate
    if (elapsedTime >= detectionRate) DetectAspect();
  }

  //Detect perspective field of view for the AI Character
  void DetectAspect() {
```

```
        RaycastHit hit;

        //Direction from current position to player position
        rayDirection = playerTrans.position -
            transform.position;

        //Check the angle between the AI character's forward
        //vector and the direction vector between player and AI
        if ((Vector3.Angle(rayDirection, transform.forward)) <
    FieldOfView) {
          // Detect if player is within the field of view
          if (Physics.Raycast(transform.position, rayDirection,
              out hit, ViewDistance)) {
            Aspect aspect =
            hit.collider.GetComponent<Aspect>();

            if (aspect != null) {
              //Check the aspect
              if (aspect.aspectName == aspectName) {
                print("Enemy Detected");
              }
            }
          }
        }
      }
```

We need to implement the `Initialize` and `UpdateSense` methods that will be called from the `Start` and `Update` methods of the parent `Sense` class, respectively. Then, in the `DetectAspect` method, we first check the angle between the player and the AI's current direction. If it's in the field of view range, we shoot a ray in the direction where the player tank is located. The ray length is the value of visible distance property. The `Raycast` method will return when it first hits another object. Then, we'll check against the aspect component and the aspect name. This way, even if the player is in the visible range, the AI character will not be able to see if it's hidden behind the wall.

The `OnDrawGizmos` method draws lines based on the perspective field of view angle and viewing distance so that we can see the AI character's line of sight in the editor window during play testing. Attach this script to our AI character and be sure that the aspect name is set to `Enemy`.

This method can be illustrated as follows:

```
    void OnDrawGizmos() {
      if (playerTrans == null) return;

      Debug.DrawLine(transform.position, playerTrans.position, Color.
red);

      Vector3 frontRayPoint = transform.position +
          (transform.forward * ViewDistance);

      //Approximate perspective visualization
      Vector3 leftRayPoint = frontRayPoint;
      leftRayPoint.x += FieldOfView * 0.5f;

      Vector3 rightRayPoint = frontRayPoint;
      rightRayPoint.x -= FieldOfView * 0.5f;

      Debug.DrawLine(transform.position, frontRayPoint, Color.green);

      Debug.DrawLine(transform.position, leftRayPoint, Color.green);

      Debug.DrawLine(transform.position, rightRayPoint, Color.green);
    }
}
```

Touching is believing

Another sense we're going to implement is `Touch.cs`, which is triggered when the player entity is within a certain area near the AI entity. Our AI character has a box collider component and its `IsTrigger` flag is on.

We need to implement the `OnTriggerEnter` event that will be fired whenever the collider component is collided with another collider component. Since our tank entity also has a collider and rigid body components, collision events will be raised as soon as the colliders of the AI character and player tank are collided.

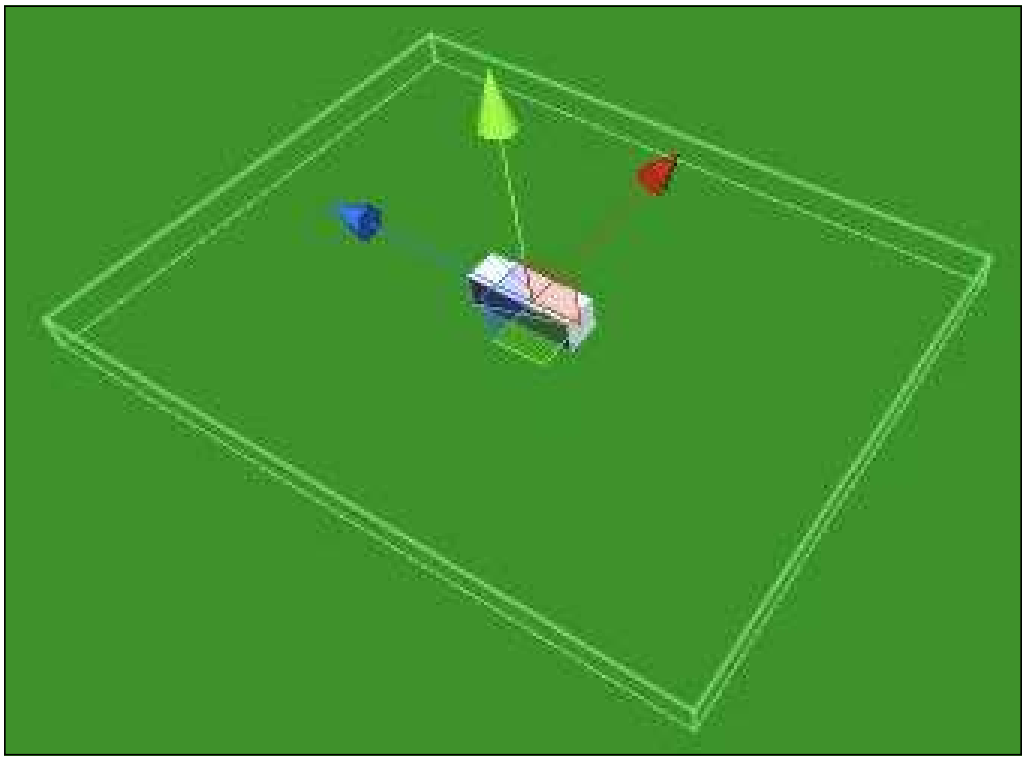
The code in the `Touch.cs` file can be shown as follows:

```
using UnityEngine;
using System.Collections;

public class Touch : Sense {
  void OnTriggerEnter(Collider other) {
    Aspect aspect = other.GetComponent<Aspect>();
```

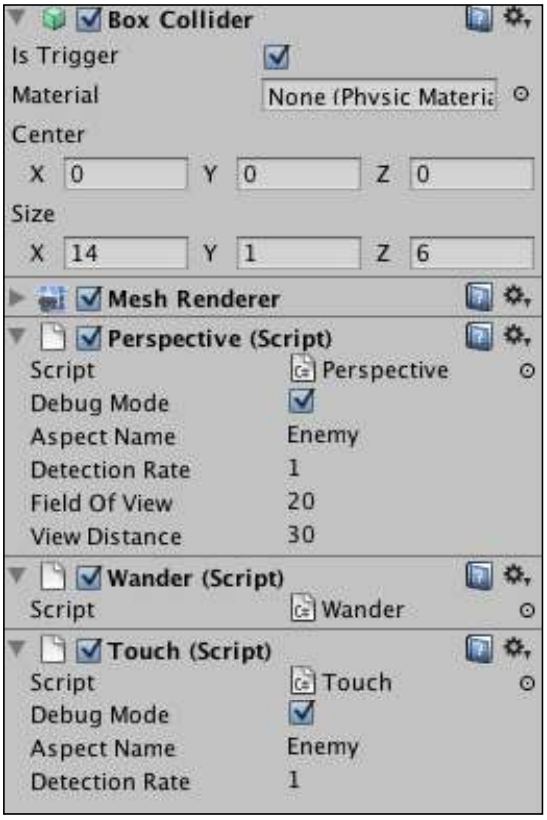
```
        if (aspect != null) {
          //Check the aspect
          if (aspect.aspectName == aspectName) {
            print("Enemy Touch Detected");
          }
        }
      }
    }
```

We implement the `OnTriggerEnter` event to be fired whenever the collider component is collided with another collider component. Since our tank entity also has a collider and the rigid body components, collision events will be raised as soon as the colliders of the AI character and the player tank are collided. Our trigger can be seen in the following screenshot:



The collider around our player

The preceding screenshot shows the box collider of our enemy AI that we'll use to implement the touch sense. In the following screenshot, we see how our AI character is set up:

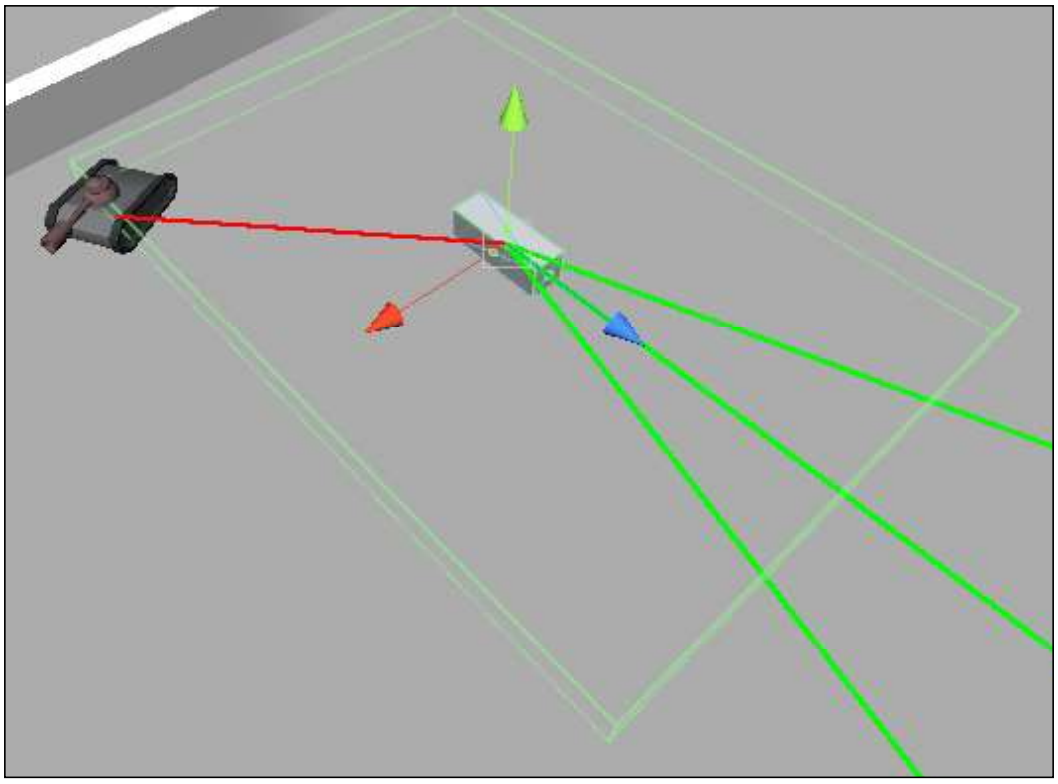


The properties of our player

Inside the `OnTriggerEnter` method, we access the aspect component of the other collided entity and check whether the name of the aspect is the aspect this AI character is looking for. And, for demo purposes, we just print out that the enemy aspect has been detected by touch sense. We can also implement other behaviors in real projects; maybe the player will turn over to an enemy and start chasing, attacking, and so on.

Testing the results

Play the game in Unity3D and move the player tank near the wandering AI character by clicking on the ground. You should see the **Enemy touch detected** message in the console log window whenever our AI character gets close to our player tank.



Our player and tank in action

The preceding screenshot shows an AI agent with touch and perspective senses looking for an enemy aspect. Move the player tank in front of the AI character, and you'll get the **Enemy detected** message. If you go to the editor view while running the game, you should see the debug drawings rendered. This is because of the `OnDrawGizmos` method implemented in the perspective `Sense` class.

Summary

This chapter introduced the concept of using sensors in implementing game AI and implemented two senses, perspective and touch, for our AI character. The sensory system is just part of the decision-making system of the whole AI system. We can use the sensory system in combination with a behavior system to execute certain behaviors for certain senses. For example, we can use an FSM to change to Chase and Attack states from the Patrol state once we have detected that there's an enemy within the line of sight. We'll also cover how to apply behavior tree systems in *Chapter 6*, *Behavior Trees*.

In the next chapter, we'll look at how to implement flocking behaviors in Unity3D as well as the Craig Reynold's flocking algorithm.