5
Agent Awareness

In this chapter, we will learn some algorithm recipes for simulating senses and agent awareness:

- The seeing function using a collider-based system
- The hearing function using a collider-based system
- The smelling function using a collider-based system
- The seeing function using a graph-based system
- The hearing function using a graph-based system
- The smelling function using a graph-based system
- Creating awareness in a stealth game

Introduction

In this chapter, we will learn different approaches on how to simulate sense stimuli on an agent. We will learn how to use tools that we are already familiar with to create these simulations: colliders, and graphs.

On the first approach, we will take advantage of ray casting, colliders, and the MonoBehaviour functions bound to this component, such as `OnCollisionEnter`, in order to leverage the need to acquire objects nearby in the three-dimensional world. Then, we will learn how to simulate the same stimuli using the graph theory and functions so that we can take advantage of this way of representing the world.

Finally, we'll learn how to implement agent awareness using a mixed approach that considers the previously learned sensory-level algorithms.

The seeing function using a collider-based system

This is probably the easiest way to simulate vision. We take a collider, be it a mesh or a Unity primitive, and use it as the tool for determining whether or not an object is inside the agent's vision range.

Getting ready

It's important to have a collider component attached to the same game object using the script on this recipe, as well as the other collider-based algorithms in this chapter. In this case, it's recommended that the collider is a pyramid-based one in order to simulate a vision cone. The fewer the polygons, the faster it will be in the game.

How to do it...

We will create a component that is able to see enemies nearby:

1. Create the `Visor` component declaring its member variables. It is important to add the following corresponding tags into Unity's configuration:

```
using UnityEngine;
using System.Collections;

public class Visor : MonoBehaviour
{
    public string tagWall = "Wall";
    public string tagTarget = "Enemy";
    public GameObject agent;
}
```

2. Implement the function for initializing the game object in case the component is already assigned to it:

```
void Start()
{
    if (agent == null)
        agent = gameObject;
}
```

3. Declare the function for checking collisions in every frame, and let's build it in the following steps:

```
public void OnTriggerStay(Collider coll)
{
    // next steps here
}
```

4. Discard the collision if it is not a target:

```
string tag = coll.gameObject.tag;
if (!tag.Equals(tagTarget))
    return;
```

5. Get the game object's position and compute its direction from the visor:

```
GameObject target = coll.gameObject;
Vector3 agentPos = agent.transform.position;
Vector3 targetPos = target.transform.position;
Vector3 direction = targetPos - agentPos;
```

6. Compute its length and create a new ray to be shot soon:

```
float length = direction.magnitude;
direction.Normalize();
Ray ray = new Ray(agentPos, direction);
```

7. Cast the created ray and retrieve all the hits:

```
RaycastHit[] hits;
hits = Physics.RaycastAll(ray, length);
```

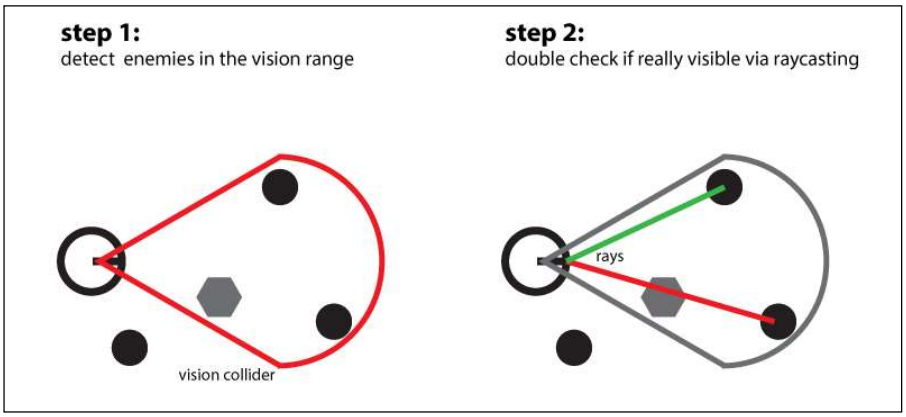
8. Check for any wall between the visor and target. If none, we can proceed to call our functions or develop our behaviors that are to be triggered:

```
int i;
for (i = 0; i < hits.Length; i++)
{
    GameObject hitObj;
    hitObj = hits[i].collider.gameObject;
    tag = hitObj.tag;
    if (tag.Equals(tagWall))
        return;
}
// TODO
// target is visible
// code your behaviour below
```

How it works...

The collider component checks every frame if it is colliding with any game object in the scene. We leverage the optimizations to Unity's scene graph and engine, and focus only on how to handle the valid collisions.

After checking, if a target object is inside the vision range represented by the collider, we cast a ray in order to check whether it is really visible or if there is a wall in between.



The hearing function using a collider-based system

In this recipe, we will emulate the sense of hearing by developing two entities: a sound emitter and a sound receiver. It is based on the principles proposed by Millington for simulating a hearing system, and it uses the power of Unity colliders to detect receivers near an emitter.

Getting ready

As with the other recipes based on colliders, we will need collider components attached to every object that is to be checked, and rigid body components attached to either emitters or receivers.

How to do it...

We will create the `SoundReceiver` class for our agents, and `SoundEmitter` for things such as alarms:

1. Create the class for the sound-receiver object:

```
using UnityEngine;
using System.Collections;

public class SoundReceiver : MonoBehaviour
{
    public float soundThreshold;
}
```

2. Define the function for our own behavior that is handling the reception of sound:

```
public virtual void Receive(float intensity, Vector3 position)
{
    // TODO
    // code your own behaviour here
}
```

3. Now, let's create the class for the sound-emitter object:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SoundEmitter : MonoBehaviour
{
    public float soundIntensity;
    public float soundAttenuation;
    public GameObject emitterObject;
    private Dictionary<int, SoundReceiver> receiverDic;
}
```

4. Initialize the list of nearby receivers and the emitter object, in case the component is attached directly:

```
void Start()
{
    receiverDic = new Dictionary<int, SoundReceiver>();
    if (emitterObject == null)
        emitterObject = gameObject;
}
```

5. Implement the function for adding new receivers to the list when they enter the emitter bounds:

```
public void OnTriggerEnter(Collider coll)
{
    SoundReceiver receiver;
    receiver = coll.gameObject.GetComponent<SoundReceiver>();
    if (receiver == null)
        return;
    int objId = coll.gameObject.GetInstanceID();
    receiverDic.Add(objId, receiver);
}
```

6. Also, implement the function for removing receivers from the list when they are out of reach:

```
public void OnTriggerExit(Collider coll)
{
    SoundReceiver receiver;
    receiver = coll.gameObject.GetComponent<SoundReceiver>();
    if (receiver == null)
        return;
    int objId = coll.gameObject.GetInstanceID();
    receiverDic.Remove(objId);
}
```

7. Define the function for emitting sound waves to nearby agents:

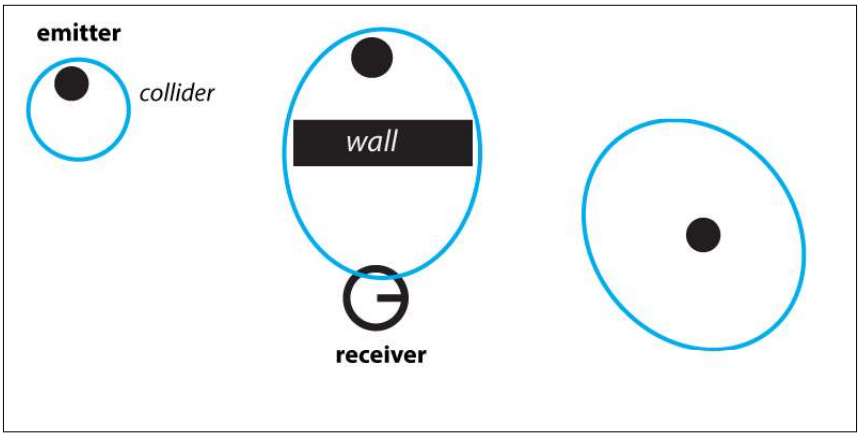
```
public void Emit()
{
    GameObject srObj;
    Vector3 srPos;
    float intensity;
    float distance;
    Vector3 emitterPos = emitterObject.transform.position;
    // next step here
}
```

8. Compute sound attenuation for every receiver:

```
foreach (SoundReceiver sr in receiverDic.Values)
{
    srObj = sr.gameObject;
    srPos = srObj.transform.position;
    distance = Vector3.Distance(srPos, emitterPos);
    intensity = soundIntensity;
    intensity -= soundAttenuation * distance;
    if (intensity < sr.soundThreshold)
        continue;
    sr.Receive(intensity, emitterPos);
}
```

How it works...

The collider triggers help register agents in the list of agents assigned to an emitter. The sound emission function then takes into account the agent's distance from the emitter in order to decrease its intensity using the concept of sound attenuation.



There is more...

We can develop a more flexible algorithm by defining different types of walls that affect sound intensity. It works by casting rays and adding up their values to the sound attenuation:

1. Create a dictionary for storing wall types as strings (using tags), and their corresponding attenuation:

```
public Dictionary<string, float> wallTypes;
```

2. Reduce sound intensity this way:

```
intensity -= GetWallAttenuation(emitterPos, srPos);
```

3. Define the function that was called in the previous step:

```
public float GetWallAttenuation(Vector3 emitterPos, Vector3
receiverPos)
{
    // next steps here
}
```

4. Compute the necessary values for ray casting:

```
float attenuation = 0f;
Vector3 direction = receiverPos - emitterPos;
float distance = direction.magnitude;
direction.Normalize();
```

5. Cast the ray and retrieve the hits:

```
Ray ray = new Ray(emitterPos, direction);
RaycastHit[] hits = Physics.RaycastAll(ray, distance);
```

6. For every wall type found via tags, add up its value (stored in the dictionary):

```
int i;
for (i = 0; i < hits.Length; i++)
{
    GameObject obj;
    string tag;
    obj = hits[i].collider.gameObject;
    tag = obj.tag;
    if (wallTypes.ContainsKey(tag))
        attenuation += wallTypes[tag];
}
return attenuation;
```

The smelling function using a collider-based system

Smelling is one of the trickiest senses to translate from the real to the virtual world. There are several techniques, but most of them are inclined to the use of colliders or graph logic.

Smelling can be simulated by computing a collision between an agent and odor particles scattered throughout the game level.

Getting ready

As with the other recipes based on colliders, we will need collider components attached to every object that is to be checked, and rigid body components attached to either emitters or receivers.

How to do it...

We will develop the scripts for representing odor particles and agents that are able to smell:

1. Create the particle's script and define its member variables for computing its lifespan:

```
using UnityEngine;
using System.Collections;

public class OdourParticle : MonoBehaviour
{
    public float timespan;
    private float timer;
}
```

2. Implement the `Start` function for proper validations:

```
void Start()
{
    if (timespan < 0f)
        timespan = 0f;
    timer = timespan;
}
```

3. Implement the timer and destroy the object after its lifecycle:

```
void Update()
{
    timer -= Time.deltaTime;
    if (timer < 0f)
        Destroy(gameObject);
}
```

4. Create the class for representing the sniffer agent:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class Smeller : MonoBehaviour
{
    private Vector3 target;
    private Dictionary<int, GameObject> particles;
}
```

5. Initialize the dictionary for storing odor particles:

```
void Start()
{
    particles = new Dictionary<int, GameObject>();
}
```

6. Add to the dictionary the colliding objects that have the odor-particle component attached:

```
public void OnTriggerEnter(Collider coll)
{
    GameObject obj = coll.gameObject;
    OdourParticle op;
    op = obj.GetComponent<OdourParticle>();
    if (op == null)
        return;
    int objId = obj.GetInstanceID();
    particles.Add(objId, obj);
    UpdateTarget();
}
```

7. Release the odor particles from the local dictionary when they are out of the agent's range or are destroyed:

```
public void OnTriggerExit(Collider coll)
{
    GameObject obj = coll.gameObject;
    int objId = obj.GetInstanceID();
    bool isRemoved;
    isRemoved = particles.Remove(objId);
    if (!isRemoved)
        return;
    UpdateTarget();
}
```

8. Create the function for computing the odor centroid according to the current elements in the dictionary:

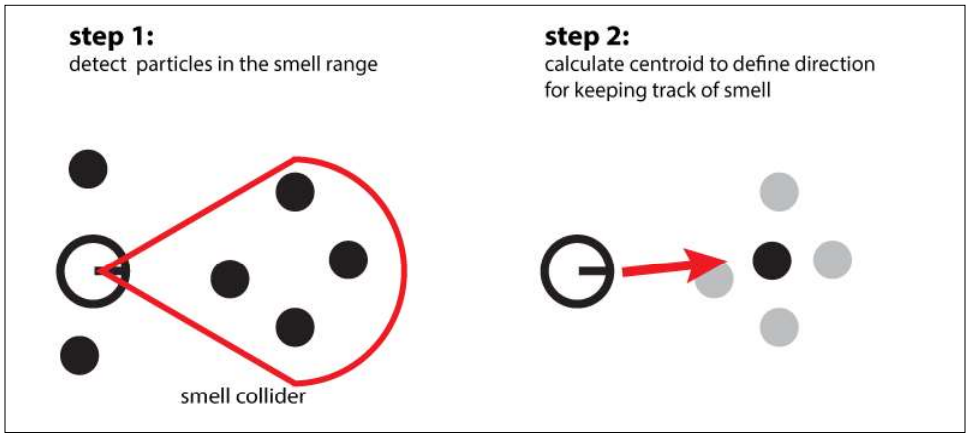
```
private void UpdateTarget()
{
    Vector3 centroid = Vector3.zero;
    foreach (GameObject p in particles.Values)
    {
        Vector3 pos = p.transform.position;
        centroid += pos;
    }
    target = centroid;
}
```

9. Implement the function for retrieving the odor centroids, if any:

```
public Vector3? GetTargetPosition()
{
    if (particles.Keys.Count == 0)
        return null;
    return target;
}
```

How it works...

Just like the hearing recipe based on colliders, we use the trigger colliders in order to register odor particles in an agent's perception (implemented using a dictionary). When a particle is included or removed, the odor centroid is computed. However, we implement a function for retrieving that centroid because when no odor particle is registered, the internal centroid position is not updated.



There is more...

Particle emission logic is left behind to be implemented according to our game's needs, and it's basically instantiating odor-particle prefabs. Also, it is recommended that you attach the rigid-body components to the agents. Odor particles are prone to be massively instantiated, reducing the game's performance.

The seeing function using a graph-based system

We will start the recipes oriented to use graph-based logic in order to simulate sense. Again, we start by developing the sense of vision.

Getting ready

It is important to have grasped the chapter regarding path finding in order to understand the inner workings of the graph-based recipes.

How to do it...

We will just implement a new file:

1. Create the class for handling vision:

```
using UnityEngine;
using System.Collections;
```

```
using System.Collections.Generic;

public class VisorGraph : MonoBehaviour
{
    public int visionReach;
    public GameObject visorObj;
    public Graph visionGraph;
}
```

2. Validate the visor object in case the com:

```
void Start()
{
    if (visorObj == null)
        visorObj = gameObject;
}
```

3. Define and start building the function for detecting the visibility of a given set of nodes:

```
public bool IsVisible(int[] visibilityNodes)
{
    int vision = visionReach;
    int src = visionGraph.GetNearestVertex(visorObj);
    HashSet<int> visibleNodes = new HashSet<int>();
    Queue<int> queue = new Queue<int>();
    queue.Enqueue(src);
}
```

4. Implement a Breadth-First Search algorithm:

```
while (queue.Count != 0)
{
    if (vision == 0)
        break;
    int v = queue.Dequeue();
    List<int> neighbours = visionGraph.GetNeighbors(v);
    foreach (int n in neighbours)
    {
        if (visibleNodes.Contains(n))
            continue;
        queue.Enqueue(v);
        visibleNodes.Add(v);
    }
}
```

5. Compare the set of visible nodes with the set of nodes reached by the vision system:

```
foreach (int vn in visibleNodes)
{
    if (visibleNodes.Contains(vn))
        return true;
}
```

6. Return false if there is no match between the two sets of nodes:

```
return false;
```

How it works...

The recipe uses the Breadth-First Search algorithm in order to discover nodes within its vision reach, and then compares this set of nodes to the set where agents reside.

The input array is computed outside, and it's out of the scope of this recipe because it relies on pinpointing, for example, the position of each agent or object that needs to be checked visibly.

The hearing function using a graph-based system

Hearing works similarly to vision but doesn't take into account the nodes direct visibility because of the properties of the sound. However, we still need a sound receiver in order to make it work. Instead of making an agent a direct sound receiver, in this recipe, the sound travels along the sound graph and is perceived by the graph nodes.

Getting ready

It is important to have grasped the chapter regarding path finding in order to understand the inner workings of the graph-based recipes.

How to do it...

1. Create the emitter class:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class EmitterGraph : MonoBehaviour
{
    // next steps
}
```

2. Declare the member variables:

```
public int soundIntensity;
public Graph soundGraph;
public GameObject emitterObj;
```

3. Implement the validation of the emitter object's reference:

```
public void Start()
{
    if (emitterObj == null)
        emitterObj = gameObject;
}
```

4. Declare the function for emitting sounds:

```
public int[] Emit()
{
    // next steps
}
```

5. Declare and assign the variables needed:

```
List<int> nodeIds = new List<int>();
Queue<int> queue = new Queue<int>();
List<int> neighbours;
int intensity = soundIntensity;
int src = soundGraph.GetNearestVertex(emitterObj);
```

6. Add the source node to the list of reached nodes and the queue:

```
nodeIds.Add(src);
queue.Enqueue(src);
```

7. Code the Breadth-First Search loop for reaching out to nodes:

```
while (queue.Count != 0)
{
    // next steps
}
return nodeIds.ToArray();
```

8. Finish the loop if the sound runs out of intensity:

```
if (intensity == 0)
    break;
```

9. Take a node from the queue and get its neighbors:

```
int v = queue.Dequeue();
neighbours = soundGraph.GetNeighbors(v);
```

10. Check the neighbors and add them to the queue if necessary:

```
foreach (int n in neighbours)
{
    if (nodeIds.Contains(n))
        continue;
    queue.Enqueue(n);
    nodeIds.Add(n);
}
```

11. Reduce the sound intensity:

```
intensity--;
```

How it works...

The recipe returns the list of affected nodes by the sound intensity using the Breadth-First Search algorithm. The algorithm stops when there are no more nodes to visit, or when the intensity of the sound is dimmed by the graph traversal.

There is more...

After learning how to implement hearing using both colliders and graph logic, you could develop a new hybrid algorithm that relies on a heuristic that takes distance as inputs. If a node goes beyond the sound's maximum distance, there's no need to add its neighbors to the queue.

See also

The following recipes of *Chapter 2*, *Navigation*:

- *Breadth-First Search algorithm*
- *A* algorithm (for taking a heuristic function as argument)*

The smelling function using a graph-based system

In this recipe, we take a mixed approach to tag vertices with a given odor particle that collides with it.

Getting ready

The vertices should have a broad collider attached so that they catch the odor particles nearby.

How to do it...

1. Add the following member variable to the odor-particle script to store its parent ID:

```
public int parent;
```

2. Create the new odor-enabled class, deriving from the original vertex:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class VertexOdour : Vertex
{
    private Dictionary<int, OdourParticle> odourDic;
}
```

3. Initialize the odor dictionary in the proper function:

```
public void Start()
{
    odourDic = new Dictionary<int, OdourParticle>();
}
```

4. Add the odor to the vertex's dictionary:

```
public void OnCollisionEnter(Collision coll)
{
    OdourOdourParticle op;
    op = coll.gameObject.GetComponent<OdourParticle>();
    if (op == null)
        return;
    int id = op.parent;
    odourDic.Add(id, op);
}
```

5. Remove the odor from the vertex's dictionary:

```
public void OnCollisionExit(Collision coll)
{
    OdourParticle op;
    op = coll.gameObject.GetComponent<OdourParticle>();
    if (op == null)
        return;
    int id = op.parent;
    odourDic.Remove(id);
}
```

6. Implement the function for checking if there is any odor tagged:

```
public bool HasOdour()
{
    if (odourDic.Values.Count == 0)
        return false;
    return true;
}
```

7. Implement the function for checking if a given type of odor is indexed in the vertex:

```
public bool OdourExists(int id)
{
    return odourDic.ContainsKey(id);
}
```

How it works...

The odor particles collide with the vertices, being indexed in their odor dictionary. From that point on, our agents can check whether a given odor is registered in a vertex nearby.

See also

- *Chapter 2*, *Navigation* recipe, *BFS and graph construction*

Creating awareness in a stealth game

Now that we know how to implement sensory-level algorithms, it's time to see how they could be taken into account in order to develop higher-level techniques for creating agent awareness.

This recipe is based on the work of Brook Miles and its team at Klei Entertainment for the game, *Mark of the Ninja*. The mechanism moves around the notion of having interest sources that can be seen or heard by the agents, and a sensory manager handling them.

Getting ready

As a lot of things move around the idea of interests, we'll need two data structures for defining an interest's sense and priority, and a data type for the interest itself.

This is the data structure for sense:

```
public enum InterestSense
{
    SOUND,
    SIGHT
};
```

This is the data structure for priority:

```
public enum InterestPriority
{
    LOWEST = 0,
    BROKEN = 1,
    MISSING = 2,
    SUSPECT = 4,
    SMOKE = 4,
    BOX = 5,
    DISTRACTIONFLARE = 10,
    TERROR = 20
};
```

The following is the interest data type:

```
using UnityEngine;
using System.Collections;

public struct Interest
{
    public InterestSense sense;
    public InterestPriority priority;
    public Vector3 position;
}
```

Before developing the necessary classes for implementing this idea, it's important to note that sensory-level functions are left blank in order to keep the recipe flexible and open to our custom implementations. These implementations could be developed using some of the previously learned recipes.

How to do it...

This is a long recipe where we'll implement two extensive classes. It is advised to carefully read the following steps:

1. Let's start by creating the class that defines our agents, and its member variables:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class AgentAwared : MonoBehaviour
{
    protected Interest interest;
    protected bool isUpdated = false;
}
```

2. Define the function for checking whether a given interest is relevant or not:

```
public bool IsRelevant(Interest i)
{
    int oldValue = (int)interest.priority;
    int newValue = (int)i.priority;
    if (newValue <= oldValue)
        return false;
    return true;
}
```

3. Implement the function for setting a new interest in the agent:

```
public void Notice(Interest i)
{
    StopCoroutine(Investigate());
    interest = i;
    StartCoroutine(Investigate());
}
```

4. Define the custom function for investigating. This will have our own implementation, and it will take into account the agent's interest:

```
public virtual IEnumerator Investigate()
{
    // TODO
    // develop your implementation
    yield break;
}
```

5. Define the custom function for leading. This will define what an agent does when it's in charge of giving orders, and will depend on our own implementation:

```
public virtual IEnumerator Lead()
{
    // TODO
    // develop your implementation
    yield break;
}
```

6. Create the class for defining interest sources:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class InterestSource : MonoBehaviour
{
    public InterestSense sense;
    public float radius;
    public InterestPriority priority;
    public bool isActive;
}
```

7. Implement a property for retrieving its interest:

```
public Interest interest
{
    get
    {
        Interest i;
        i.position = transform.position;
        i.priority = priority;
        i.sense = sense;
        return i;
    }
}
```

8. Define the function for checking whether or not the agent is affected by the interest source. This could be defined in the agent's class, but it requires some changes in some of the next steps. This is one of the sensory-level functions:

```
protected bool IsAffectedSight(AgentAwared agent)
{
    // TODO
    // your sight check implementation
    return false;
}
```

9. Implement the next sensory-level function for checking if an agent is affected by sound. It has the same architectural considerations as the previous step:

```
protected bool IsAffectedSound(AgentAwared agent)
{
    // TODO
    // your sound check implementation
    return false;
}
```

10. Define the function for getting the list of agents affected by the interest source. It is declared virtual, in case we need to specify further, or simply change the way it works:

```
public virtual List<AgentAwared> GetAffected(AgentAwared[]
agentList)
{
    List<AgentAwared> affected;
    affected = new List<AgentAwared>();
    Vector3 interPos = transform.position;
    Vector3 agentPos;
    float distance;
    // next steps
}
```

11. Start creating the main loop for traversing the list of agents and return the list of affected ones:

```
foreach (AgentAwared agent in agentList)
{
    // next steps
}
return affected;
```

12. Discriminate an agent if it is out of the source's action radius:

```
agentPos = agent.transform.position;
distance = Vector3.Distance(interPos, agentPos);
if (distance > radius)
    continue;
```

13. Check whether the agent is affected, given the source's type of sense:

```
bool isAffected = false;
switch (sense)
{
    case InterestSense.SIGHT:
        isAffected = IsAffectedSight(agent);
        break;
    case InterestSense.SOUND:
```

```
            isAffected = IsAffectedSound(agent);
            break;
    }
```

14. If the agent is affected, add it to the list:

```
if (!isAffected)
    continue;
affected.Add(agent);
```

15. Next, create the class for the sensory manager:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class SensoryManager : MonoBehaviour
{
    public List<AgentAwared> agents;
    public List<InterestSource> sources;
}
```

16. Implement its `Awake` function:

```
public void Awake()
{
    agents = new List<AgentAwared>();
    sources = new List<InterestSource>();
}
```

17. Declare the function for getting a set of scouts, given a group of agents:

```
public List<AgentAwared> GetScouts(AgentAwared[] agents, int
leader = -1)
{
    // next steps
}
```

18. Validate according to the number of agents:

```
if (agents.Length == 0)
    return new List<AgentAwared>(0);
if (agents.Length == 1)
    return new List<AgentAwared>(agents);
```

19. Remove the leader, if given its index:

```
List<AgentAwared> agentList;
agentList = new List<AgentAwared>(agents);
if (leader > -1)
    agentList.RemoveAt(leader);
```

20. Calculate the number of scouts to retrieve:

```
List<AgentAwared> scouts;
scouts = new List<AgentAwared>();
float numAgents = (float)agents.Length;
int numScouts = (int)Mathf.Log(numAgents, 2f);
```

21. Get random scouts from the list of agents:

```
while (numScouts != 0)
{
    int numA = agentList.Count;
    int r = Random.Range(0, numA);
    AgentAwared a = agentList[r];
    scouts.Add(a);
    agentList.RemoveAt(r);
    numScouts--;
}
```

22. Retrieve the scouts:

```
return scouts;
```

23. Define the function for checking the list of interest sources:

```
public void UpdateLoop()
{
    List<AgentAwared> affected;
    AgentAwared leader;
    List<AgentAwared> scouts;
    foreach (InterestSource source in sources)
    {
        // next steps
    }
}
```

24. Avoid inactive sources:

```
if (!source.isActive)
    continue;
source.isActive = false;
```

25. Avoid sources that don't affect any agent:

```
affected = source.GetAffected(agents.ToArray());
if (affected.Count == 0)
    continue;
```

26. Get a random leader and the set of scouts:

```
int l = Random.Range(0, affected.Count);
leader = affected[l];
scouts = GetScouts(affected.ToArray(), l);
```

27. Call the leader to its role if necessary:

```
if (leader.Equals(scouts[0]))
    StartCoroutine(leader.Lead());
```

28. Finally, inform the scouts about noticing the interest, in case it's relevant to them:

```
foreach (AgentAwared a in scouts)
{
    Interest i = source.interest;
    if (a.IsRelevant(i))
        a.Notice(i);
}
```

How it works...

There is a list of interest sources that could get the attention of a number of agents in the world. Those lists are kept in a manager that handles the global update for every source, taking into account only the active ones.

An interest source receives the list of agents in the world and retrieves only the affected agents after a two-step process. First, it sets aside all the agents that are outside its action radius and then only takes into account those agents that can be reached with a finer (and more expensive) sensory-level mechanism.

The manager handles the affected agents, sets up a leader and scouts, and finally

There is more...

It is worth mentioning that the `SensoryManager` class works as a hub to store and organize the list of agents and the list of interest sources, so it ought to be a singleton. Its duplication could bring undesired complexity or behavior.

An agent's interest is automatically changed by the sensory manager using the priority values. Still, it can be reset when needed, using the public function `Notice`.

There is room for improvement still, depending on our game. The scout lists can overlap with each other, and it's up to us and our game to handle this scenario the best way we can. However, the system we built takes advantage of the priority values to make decisions.

See also

For further information on the train of thought behind this recipe, please refer to Steve Rabin's book, *Game AI Pro*.