4
Coordination and Tactics

In this chapter, we will learn techniques for coordination and devising tactics:

- Handling formations
- Extending A* for coordination: A*mbush
- Creating good waypoints
- Analyzing waypoints by height
- Analyzing waypoints by cover and visibility
- Exemplifying waypoints for decision making
- Introducing influence maps
- Improving influence with map flooding
- Improving influence with convolution filters
- Building a fighting circle

Introduction

As we will see, this is not a chapter focused on a sole subject, but rather a chapter that has its own original recipes and also learns from previous recipes in order to create new or improved techniques.

In this chapter, we will learn different recipes for coordinating different agents into one organism, such as formations and techniques that allow us to make tactical decisions based on graphs (such as waypoints) and influence maps. These techniques use different elements from the previous chapters and recipes, especially from the graph construction and path finding algorithms found in *Chapter 2*, *Navigation*.

Handling formations

This is a key algorithm for creating flocks or a group of military agents. It is designed to be flexible enough to give you the chance to create your own formations.

The end result from this recipe will be a set of target positions and rotations for each agent in the formation. Then, it is up to you to create the necessary algorithms to move the agent to the previous targets.

We can use the movement algorithms learnt in *Chapter 1*, *Movement*, in order to target those positions.

Getting ready

We will need to create three base classes that are the data types to be used by the high-level classes and algorithms. The `Location` class is very similar to the `Steering` class and is used to define a target position and rotation given the formation's anchor point and rotation. The `SlogAssignment` class is a data type to match a list's indices and agents. Finally, the `Character` class component holds the target `Location` class.

The following is the code for the `Location` class:

```
using UnityEngine;
using System.Collections;

public class Location
{
    public Vector3 position;
    public Quaternion rotation;

    public Location ()
    {
        position = Vector3.zero;
        rotation = Quaternion.identity;
    }

    public Location(Vector3 position, Quaternion rotation)
```

```
        {
            this.position = position;
            this.rotation = rotation;
        }
    }
```

The following is the code for the `SlotAssignment` class:

```
using UnityEngine;
using System.Collections;

public class SlotAssignment
{
    public int slotIndex;
    public GameObject character;

    public SlotAssignment()
    {
        slotIndex = -1;
        character = null;
    }
}
```

The following is the code for the `Character` class:

```
using UnityEngine;
using System.Collections;

public class Character : MonoBehaviour
{
    public Location location;

    public void SetTarget (Location location)
    {
        this.location = location;
    }
}
```

How to do it...

We will implement two classes—`FormationPattern` and `FormationManager`:

1. Create the `FormationPattern` pseudo-abstract class:

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

```
using System.Collections.Generic;

public class FormationPattern: MonoBehaviour
{
    public int numOfSlots;
    public GameObject leader;
}
```

2. Implement the `Start` function:

```
void Start()
{
    if (leader == null)
        leader = transform.gameObject;
}
```

3. Define the function for getting the position for a given slot:

```
public virtual Vector3 GetSlotLocation(int slotIndex)
{
    return Vector3.zero;
}
```

4. Define the function for retrieving whether a given number of slots is supported by the formation:

```
public bool SupportsSlots(int slotCount)
{
    return slotCount <= numOfSlots;
}
```

5. Implement the function for setting an offset in the locations, if necessary:

```
public virtual Location GetDriftOffset(List<SlotAssignment>
slotAssignments)
{
    Location location = new Location();
    location.position = leader.transform.position;
    location.rotation = leader.transform.rotation;
    return location;
}
```

6. Create the appropriate class for managing the formation:

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

public class FormationManager : MonoBehaviour
{
```

```
    public FormationPattern pattern;
    private List<SlotAssignment> slotAssignments;
    private Location driftOffset;
}
```

7. Implement the `Awake` function:

```
void Awake()
{
    slotAssignments = new List<SlotAssignment>();
}
```

8. Define the function for updating the slot assignments given the list's order:

```
public void UpdateSlotAssignments()
{
    for (int i = 0; i < slotAssignments.Count; i++)
    {
        slotAssignments[i].slotIndex = i;
    }
    driftOffset = pattern.GetDriftOffset(slotAssignments);
}
```

9. Implement the function for adding a character in the formation:

```
public bool AddCharacter(GameObject character)
{
    int occupiedSlots = slotAssignments.Count;
    if (!pattern.SupportsSlots(occupiedSlots + 1))
        return false;
    SlotAssignment sa = new SlotAssignment();
    sa.character = character;
    slotAssignments.Add(sa);
    UpdateSlotAssignments();
    return true;
}
```

10. Implement the function for removing a character in the formation:

```
public void RemoveCharacter(GameObject agent)
{
    int index = slotAssignments.FindIndex(x => x.character.
Equals(agent));
    slotAssignments.RemoveAt(index);
    UpdateSlotAssignments();
}
```

11. Implement the function for updating the slots:

```
public void UpdateSlots()
{
    GameObject leader = pattern.leader;
    Vector3 anchor = leader.transform.position;
    Vector3 slotPos;
    Quaternion rotation;
    rotation = leader.transform.rotation;
    foreach (SlotAssignment sa in slotAssignments)
    {
        // next step
    }
}
```

12. Finally, implement the `foreach` loop:

```
Vector3 relPos;
slotPos = pattern.GetSlotLocation(sa.slotIndex);
relPos = anchor;
relPos += leader.transform.TransformDirection(slotPos);
Location charDrift = new Location(relPos, rotation);
Character character = sa.character.GetComponent<Character>();
character.SetTarget(charDrift);
```

How it works...

The `FormationPattern` class contains the relative positions to a given slot. For example, a child `CircleFormation` class will implement the `GetSlotLocation` class, given the number of slots and its locations over 360 degrees. It is intended to be a basic class, so it is up to the manager to add a layer for permissions and rearrangement. That way, the designer can focus on simple formation scripting, deriving from the base class.

The `FormationManager` class, as stated earlier, handles the high-level layer and arranges the locations in line with the formation's needs and permissions. The calculations are based on the leader's position and rotation, and they apply the necessary transformations given the pattern's principles.

There is more...

It is worth mentioning that the `FormationManager` and `FormationPattern` classes are intended to be components of the same object. When the leader field in the manager is set to null, the leader is the object itself. That way, we could have a different leader object in order to have a clean inspector window and class modularity.

See also

- Refer to *Chapter 1*, *Movement*, the *Arriving and leaving* recipe
- For further information on drift offset and how to play with this value, please refer to Ian Millington's book, *Artificial Intelligence for Games*

Extending A* for coordination: A*mbush

After learning how to implement A* for path finding, we will now use its power and flexibility to develop some kind of coordinated behavior in order to ambush the player. This algorithm is especially useful when we want a non-expensive solution for the aforementioned problem, and one that is also easy to implement.

This recipe sets the path for every agent to be taken into account when it comes to ambushing a given vertex or point in the graph.

Getting ready

We need a special component for the agents called `Lurker`. This class will hold the paths that are to be used later in the navigation process.

The following is the code for `Lurker`:

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

public class Lurker : MonoBehaviour
{
    [HideInInspector]
    public List<int> pathIds;
    [HideInInspector]
    public List<GameObject> pathObjs;

    void Awake()
    {
        if (pathIds == null)
            pathIds = new List<int>();
        if (pathObjs == null)
            pathObjs = new List<GameObject>();
    }
}
```

How to do it...

We will create the main function for setting the ambush path for all the agents and then the function for setting each agent's path.

1. Define the main function for the ambush:

```
public void SetPathAmbush(GameObject dstObj, List<Lurker> lurkers)
{
    Vertex dst = GetNearestVertex(dstObj.transform.position);
    foreach (Lurker l in lurkers)
    {
        Vertex src = GetNearestVertex(l.transform.position);
        l.path = AStarMbush(src, dst, l, lurkers);
    }
}
```

2. Declare the function for finding each path:

```
public List<Vertex> AStarMbush(
        Vertex src,
        Vertex dst,
        Lurker agent,
        List<Lurker> lurkers,
        Heuristic h = null)
{    // next steps
}
```

3. Declare the necessary members for handling the extra cost of computations:

```
int graphSize = vertices.Count;
float[] extra = new float[graphSize];
float[] costs = new float[graphSize];
int i;
```

4. Initialize the regular cost and the extra cost variables:

```
for (i = 0; i < graphSize; i++)
{
    extra[i] = 1f;
    costs[i] = Mathf.Infinity;
}
```

5. Add extra cost to each vertex that is contained in another agent's path:

```
foreach (Lurker l in lurkers)
{
    foreach (Vertex v in l.path)
    {
```

```
            extra[v.id] += 1f;
        }
    }
```

6. Declare and initialize the variables for computing A*:

```
Edge[] successors;
int[] previous = new int[graphSize];
for (i = 0; i < graphSize; i++)
    previous[i] = -1;
previous[src.id] = src.id;
float cost = 0;
Edge node = new Edge(src, 0);
GPWiki.BinaryHeap<Edge> frontier = new GPWiki.BinaryHeap<Edge>();
```

7. Start implementing the A* main loop:

```
frontier.Add(node);
while (frontier.Count != 0)
{
    if (frontier.Count == 0)
        return new List<GameObject>();
    // next steps
}
return new List<Vertex>();
```

8. Validate that the goal has already been reached; otherwise it's not worth computing the costs, and it would be better to continue with the usual A* algorithm:

```
node = frontier.Remove();
if (ReferenceEquals(node.vertex, dst))
    return BuildPath(src.id, node.vertex.id, ref previous);
int nodeId = node.vertex.id;
if (node.cost > costs[nodeId])
    continue;
```

9. Traverse the neighbors and check whether they have been visited:

```
successors = GetEdges(node.vertex);
foreach (Edge e in successors)
{
    int eId = e.vertex.id;
    if (previous[eId] != -1)
        continue;
    // next step
}
```

10. If they haven't been visited, add them to the frontier:

```
cost = e.cost;
cost += costs[dst.id];
cost += h(e.vertex, dst);
if (cost < costs[e.vertex.id])
{
    Edge child;
    child = new Edge(e.vertex, cost);
    costs[eId] = cost;
    previous[eId] = nodeId;
    frontier.Remove(e);
    frontier.Add(child);
}
```

How it works...

The A*mbush algorithm analyses the path of every agent and increases the cost of that node. That way, when an agent computes its path using A*, it is better to choose a different route than the one chosen by other agents, thus, creating the perception of an ambush among the target positions.

There is more...

There is an easy-to-implement improvement over the algorithm, which leads to the P-A*mbush variation. Simply ordering the lurkers' list from the closest to the farthest might provide a better result at almost no extra cost in computation. This is due to the fact that the ordering operation is handled just once, and could be easily implemented via a priority queue, and then retrieves it as a list to the main A*mbush algorithm with no extra changes.

Creating good waypoints

There are times when the number of waypoints must be reduced at a certain point during the game or just for memory constraints. In this recipe, we will learn a technique called condensation that helps us deal with this problem, forcing the waypoints to compete with each other given their assigned value.

Getting ready

In this recipe, we will deal with static member functions. It is important that we understand the use and value of static functions.

How to do it...

We will create the `Waypoint` class and add the functions for condensing the set of waypoints.

1. Create the `Waypoint` class, deriving not only from `MonoBehaviour`, but also from the `IComparer` interface:

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

public class Waypoint : MonoBehaviour, IComparer
{
    public float value;
    public List<Waypoint> neighbours;
}
```

2. Implement the `Compare` function from the aforementioned interface:

```
public int Compare(object a, object b)
{
    Waypoint wa = (Waypoint)a;
    Waypoint wb = (Waypoint)b;
    if (wa.value == wb.value)
        return 0;
    if (wa.value < wb.value)
        return -1;
    return 1;
}
```

3. Implement the static function to compute whether an agent can move between two waypoints:

```
public static bool CanMove(Waypoint a, Waypoint b)
{
    // implement your own behaviour for
    // deciding whether an agent can move
    // easily between two waypoints
    return true;
}
```

4. Start declaring the function for condensing the waypoints:

```
public static void CondenseWaypoints(List<Waypoint> waypoints,
float distanceWeight)
{
    // next steps
}
```

5. Initialize some variables and sort the waypoints in descending order:

```
distanceWeight *= distanceWeight;
waypoints.Sort();
waypoints.Reverse();
List<Waypoint> neighbours;
```

6. Start the loop for computing each waypoint:

```
foreach (Waypoint current in waypoints)
{
    // next steps
}
```

7. Retrieve the waypoint neighbors, sort them, and start the loop to make them compete with each other:

```
neighbours = new List<Waypoint>(current.neighbours);
neighbours.Sort();
foreach (Waypoint target in neighbours)
{
    if (target.value > current.value)
        break;
    if (!CanMove(current, target))
        continue;
    // next steps
}
```

8. Compute the target's position:

```
Vector3 deltaPos = current.transform.position;
deltaPos -= target.transform.position;
deltaPos = Vector3.Cross(deltaPos, deltaPos);
deltaPos *= distanceWeight;
```

9. Compute the target's overall value and decide whether to keep it or throw it:

```
float deltaVal = current.value - target.value;
deltaVal *= deltaVal;
if (deltaVal < distanceWeight)
{
    neighbours.Remove(target);
    waypoints.Remove(target);
}
```

How it works...

The waypoints are ordered according to their relevance (such as height to be used as a sniping or advantage location) and then their neighbors are checked to see which ones are going to be dismissed from the condensation. Naturally, the less valuable waypoints are kept to the end of the computation cycle. In the next recipe, we will learn how to analyze waypoints.

See also

Refer to the the following recipes:

- *Analyzing waypoints by height*
- *Analyzing waypoints by cover and visibility*

Analyzing waypoints by height

This recipe lets us evaluate a waypoint according to its position. Strategically speaking, lower positions are at a disadvantage. In this case, we will use a flexible algorithm to get the quality of a waypoint, given the heights of its surroundings.

Getting ready

This recipe is simple enough, so there is no extra content to be aware of. The algorithm is flexible enough to receive a list of positions, which is given by the waypoint's neighbors or just the complete graph of waypoints. The surroundings heuristic is kept outside for our perusal and it gives the game's specific design.

How to do it...

We will implement a function to evaluate a location given its height and its surrounding points:

1. Declare the function for evaluating the quality:

```
public static float GetHeightQuality (Vector3 location, Vector3[]
surroundings)
{
    // next steps
}
```

2. Initialize the variables for handling the computation:

```
float maxQuality = 1f;
float minQuality = -1f;
float minHeight = Mathf.Infinity;
float maxHeight = Mathf.NegativeInfinity;
float height = location.y;
```

3. Traverse the surroundings in order to find the maximum and minimum height:

```
foreach (Vector3 s in surroundings)
{
    if (s.y > maxHeight)
        maxHeight = s.y;
    if (s.y < minHeight)
        minHeight = s.y;
}
```

4. Compute the quality in the given range:

```
float quality = (height-minHeight) / (maxHeight - minHeight);
quality *= (maxQuality - minQuality);
quality += minQuality;
return quality;
```

How it works...

We traverse the list of surroundings to find the maximum and minimum width and then compute the location value in the range of -1, 1. We could change this range to meet our game's design, or invert the importance of the height in the formula.

Analyzing waypoints by cover and visibility

When dealing with military games, especially FPS, we need to define a waypoint value, by its capacity, to be a good cover point with the maximum visibility for shooting or reaching other enemies visually. This recipe helps us compute a waypoint's value given these parameters.

Getting ready

We need to create a function for checking whether a position is in the same room as others:

```
public bool IsInSameRoom(Vector3 from, Vector3 location, string
tagWall = "Wall")
{
    RaycastHit[] hits;
    Vector3 direction = location - from;
    float rayLength = direction.magnitude;
    direction.Normalize();
    Ray ray = new Ray(from, direction);
    hits = Physics.RaycastAll(ray, rayLength);
    foreach (RaycastHit h in hits)
    {
        string tagObj = h.collider.gameObject.tag;
```

```
            if (tagObj.Equals(tagWall))
                return false;
        }
        return true;
    }
```

How to do it...

We will create the function that computes the quality of the waypoint:

1. Define the function with its parameters:

```
public static float GetCoverQuality(
        Vector3 location,
        int iterations,
        Vector3 characterSize,
        float radius,
        float randomRadius,
        float deltaAngle)
{
    // next steps
}
```

2. Initialize the variable for handling the degrees of rotation, possible hits received, and valid visibility:

```
float theta = 0f;
int hits = 0;
int valid = 0;
```

3. Start the main loop for the iterations to be computed on this waypoint and return the computed value:

```
for (int i = 0; i < iterations; i++)
{
    // next steps
}
return (float)(hits / valid);
```

4. Create a random position near the waypoint's origin to see if the waypoint is easily reachable:

```
Vector3 from = location;
float randomBinomial = Random.Range(-1f, 1f);
from.x += radius * Mathf.Cos(theta) + randomBinomial *
randomRadius;
from.y += Random.value * 2f * randomRadius;
from.z += radius * Mathf.Sin(theta) + randomBinomial *
randomRadius;
```

5. Check whether the random position is in the same room:

```
if (!IsInSameRoom(from, location))
    continue;
valid++;
```

6. If the random position in the same room, then:

```
Vector3 to = location;
to.x += Random.Range(-1f, 1f) * characterSize.x;
to.y += Random.value * characterSize.y;
to.z += Random.Range(-1f, 1f) * characterSize.z;
```

7. Cast a ray to the visibility value to check whether:

```
Vector3 direction = to - location;
float distance = direction.magnitude;
direction.Normalize();
Ray ray = new Ray(location, direction);
if (Physics.Raycast(ray, distance))
    hits++;
theta = Mathf.Deg2Rad * deltaAngle;
```

How it works...

We create a number of iterations and then start putting random numbers around the waypoint to verify that it is reachable and hittable. After that, we compute a coefficient to determine its quality.

Exemplifying waypoints for decision making

Just like we learned with decision-making techniques, sometimes it is not flexible enough to just evaluate a waypoints' value, but rather a more complex condition. In this case, the solution is to apply techniques learned previously and couple them into the waypoint for attacking that problem.

The key idea is to add a condition to the node so that it can be evaluated, for example, using a decision tree and developing more complex heuristics for computing a waypoint's value.

Getting ready

It is important to revisit the recipe that handled state machines in the previous chapter before diving into the following recipe.

How to do it...

We will make a little adjustment:

1. Add `public Condition` to the `Waypoint` class:

   ```
   public Condition condition;
   ```

2. Now, you'll be able to easily integrate it into decision-making techniques using derived condition classes such as `ConditionFloat`.

How it works...

The pseudo-abstract `Condition` class, which we learned about previously, has a member function called `Test`, which evaluates whether or not that condition is met.

See also

- *Chapter 3*, *Decision Making*

Influence maps

Another way to use graphs is to represent how much reach or influence an agent, or in this case a unit, has over an area of the world. In this context, influence is represented as the total area of a map an agent, or a group of agents of the same party, covers.

This is a key element for creating good AI decision mechanisms based on the military presence in real-time simulation games, or games where it is important to know how much of the world is taken by a group of agents, each representing a given faction.

Getting ready

This is a recipe that requires the experience of graph building, so it is based on the general `Graph` class. However, we will need to derive it from a specific graph definition, or define our own methods to handle vertices and the neighbors retrieval logic, as learned in *Chapter 2*, *Navigation*.

We will learn how to implement the specific algorithms for this recipe, based on the `Graph` class general functions and the `Vertex` class.

Finally, we will need a base Unity component for our agent and `Faction enum`.

The following is the code for the `Faction enum` and `Unit` classes. They can be written in the same file, called `Unit.cs`:

```
using UnityEngine;
using System.Collections;

public enum Faction
{
    // example values
    BLUE, RED
}

public class Unit : MonoBehaviour
{
    public Faction faction;
    public int radius = 1;
    public float influence = 1f;

    public virtual float GetDropOff(int locationDistance)
    {
        return influence;
    }
}
```

How to do it...

We will build the `VertexInfluence` and `InfluenceMap` classes, used for handle vertices and the graph, respectively:

1. Create the `VertexInfluence` class, deriving from `Vertex`:

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

public class VertexInfluence : Vertex
{
    public Faction faction;
    public float value = 0f;
}
```

2. Implement the function for setting up values and notifying success:

```
public bool SetValue(Faction f, float v)
{
    bool isUpdated = false;
    if (v > value)
    {
        value = v;
        faction = f;
        isUpdated = true;
    }
    return isUpdated;
}
```

3. Create the `InfluenceMap` class deriving from `Graph` (or a more specific graph implementation):

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

public class InfluenceMap : Graph
{
    public List<Unit> unitList;
    // works as vertices in regular graph
    GameObject[] locations;
}
```

4. Define the `Awake` function for initialization:

```
void Awake()
{
    if (unitList == null)
        unitList = new List<Unit>();
}
```

5. Implement the function for adding a unit on the map:

```
public void AddUnit(Unit u)
{
    if (unitList.Contains(u))
        return;
    unitList.Add(u);
}
```

6. Implement the function for removing a unit from the map:

```
public void RemoveUnit(Unit u)
{
    unitList.Remove(u);
}
```

7. Start building the function for computing the influence:

```
public void ComputeInfluenceSimple()
{
    int vId;
    GameObject vObj;
    VertexInfluence v;
    float dropOff;
    List<Vertex> pending = new List<Vertex>();
    List<Vertex> visited = new List<Vertex>();
    List<Vertex> frontier;
    Vertex[] neighbours;

    // next steps
}
```

8. Continue by creating a loop for iterating over the list of units:

```
foreach(Unit u in unitList)
{
    Vector3 uPos = u.transform.position;
    Vertex vert = GetNearestVertex(uPos);
    pending.Add(vert);

    // next step
}
```

9. Finally, apply a BFS-based code for spreading influence given the radius reach:

```
// BFS for assigning influence
for (int i = 1; i <= u.radius; i++)
{
    frontier = new List<Vertex>();
    foreach (Vertex p in pending)
    {
        if (visited.Contains(p))
            continue;
        visited.Add(p);
        v = p as VertexInfluence;
        dropOff = u.GetDropOff(i);
        v.SetValue(u.faction, dropOff);
```

```
            neighbours = GetNeighbours(vert);
            frontier.AddRange(neighbours);
        }
        pending = new List<Vertex>(frontier);
    }
```

How it works...

The influence-map graph works exactly as a general graph as well as the influence-based vertex because there's just a couple of extra parameters for mapping the influence across the graph. The most relevant part relies on the computation of the influence, and it is based on the BFS algorithm.

For each unit on the map, we spread its influence given the radius. When the computed influence (drop off) is greater than the vertex original faction, the vertex faction is changed.

There is more...

The drop-off function should be tuned according to your specific game needs. We can define a smarter function with the following example code, using the distance parameter:

```
public virtual float GetDropOff(int locationDistance)
{
    float d = influence / radius * locationDistance;
    return influence - d;
}
```

It is important to note that the distance parameter is an integer indicating the distance measured in vertices.

Finally, we could avoid using factions and instead use a reference to the unit itself. That way, we could map the influence based on individual units, but we think it makes the most sense to think in terms of factions or teams.

See also

- *Chapter 2*, *Navigation*, the *Representing the world with grids* and *Finding the shortest path in a grid with BFS* recipes

Improving influence with map flooding

The previous influence computation is good when dealing with a simple influence that is based on individual units helping a faction. However, this could lead to holes in the map instead of covering a whole section. One technique to resolve that problem is flooding, based on the Dijkstra algorithm.

Getting ready

In this case, we will blend the faction capability for tagging a vertex, with the unit's logic for having a drop-off function, into a class called `Guild`. This is a component to include in the game object; one for each desired guild:

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

public class Guild : MonoBehaviour
{
    public string guildName;
    public int maxStrength;
    public GameObject baseObject;
    [HideInInspector]
    public int strenghth

    public virtual void Awake()
    {
        strength = maxStrength;
    }
}
```

It also needs a drop-off function. However, this time we wanted to create an example using Euclidean distance:

```
public virtual float GetDropOff(float distance)
{
    float d = Mathf.Pow(1 + distance, 2f);
    return strength / d;
}
```

Finally, we will need a `GuildRecord` data type for the Dijkstra algorithm representation of a node:

1. Create the `GuildRecord` struct, deriving from the `IComparable` interface:

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

public struct GuildRecord : IComparable<GuildRecord>
{
    public Vertex location;
    public float strength;
    public Guild guild;
}
```

2. Implement the `Equal` functions:

```
public override bool Equals(object obj)
{
    GuildRecord other = (GuildRecord)obj;
    return location == other.location;
}

public bool Equals(GuildRecord o)
{
    return location == o.location;
}
```

3. Implement the required `IComparable` functions:

```
public override int GetHashCode()
{
    return base.GetHashCode();
}

public int CompareTo(GuildRecord other)
{
    if (location == other.location)
        return 0;
    // the substraction is inverse for
    // having a descending binary heap
    return (int)(other.strength - strength);
}
```

How to do it...

Now, we just need to modify some files and add functionalities:

1. Include the `guild` member in the `VertexInfluence` class:

```
public Guild guild;
```

2. Include new members in the `InfluenceMap` class:

```
public float dropOffThreshold;
private  Guild[] guildList;
```

3. Also, in `InfluenceMap`, add the following line in the `Awake` function:

```
guildList = gameObject.GetComponents<Guild>();
```

4. Create the map-flooding function:

```
public List<GuildRecord> ComputeMapFlooding()
{
}
```

5. Declare the main necessary variables:

```
GPWiki.BinaryHeap<GuildRecord> open;
open = new GPWiki.BinaryHeap<GuildRecord>();
List<GuildRecord> closed;
closed = new List<GuildRecord>();
```

6. Add the initial nodes for each guild in the priority queue:

```
foreach (Guild g in guildList)
{
    GuildRecord gr = new GuildRecord();
    gr.location = GetNearestVertex(g.baseObject);
    gr.guild = g;
    gr.strength = g.GetDropOff(0f);
    open.Add(gr);
}
```

7. Create the main Dijkstra iteration and return the assignments:

```
while (open.Count != 0)
{
    // next steps here
}
return closed;
```

8. Take the first node in the queue and get its neighbors:

```
GuildRecord current;
current = open.Remove();
GameObject currObj;
currObj = GetVertexObj(current.location);
Vector3 currPos;
currPos = currObj.transform.position;
List<int> neighbours;
neighbours = GetNeighbors(current.location);
```

9. Create the loop for computing each neighbor, and put the current node in the closed list:

```
foreach (int n in neighbours)
{
    // next steps here
}
closed.Add(current);
```

10. Compute the drop off from the current vertex, and check whether it is worth trying to change the guild assigned:

```
GameObject nObj = GetVertexObj(n);
Vector3 nPos = nObj.transform.position;
float dist = Vector3.Distance(currPos, nPos);
float strength = current.guild.GetDropOff(dist);
if (strength < dropOffThreshold)
    continue;
```

11. Create an auxiliary `GuildRecord` node with the current vertex's data:

```
GuildRecord neighGR = new GuildRecord();
neighGR.location = n;
neighGR.strength = strength;
VertexInfluence vi;
vi = nObj.GetComponent<VertexInfluence>();
neighGR.guild = vi.guild;
```

12. Check the closed list and validate the time when a new assignment must be avoided:

```
if (closed.Contains(neighGR))
{
    int location = neighGR.location;
    int index = closed.FindIndex(x => x.location == location);
    GuildRecord gr = closed[index];
    if (gr.guild.name != current.guild.name
            && gr.strength < strength)
        continue;
}
```

13. Check the priority queue for the same reasons:

```
else if (open.Contains(neighGR))
{
    bool mustContinue = false;
    foreach (GuildRecord gr in open)
    {
        if (gr.Equals(neighGR))
        {
            mustContinue = true;
            break;
        }
    }
    if (mustContinue)
        continue;
}
```

14. Create a new `GuildRecord` assignment and add it to the priority queue when everything else fails:

```
else
{
    neighGR = new GuildRecord();
    neighGR.location = n;
}
neighGR.guild = current.guild;
neighGR.strength = strength;
```

15. Add it to the priority queue if necessary:

```
open.Add(neighGR);
```

How it works...

The algorithm returns the guild's assignment for every vertex. It traverses the whole graph starting from the guilds' bases and computes.

The algorithm traverses the whole graph starting from the guilds' positions. Given our previous inverse subtraction, the priority queue always starts from the strongest node and computes the assignment until it reaches a value below `dropOffThreshold`. It also checks for ways to avoid a new assignment if the conditions are not met: if the vertex value is greater than the current strength, or if the guild assignment is the same.

See also

- The *Introducing influence maps* recipe
- *Chapter 2*, *Navigation*, the *Finding the shortest path with Dijkstra* recipe

Improving influence with convolution filters

Convolution filters are usually applied via image processing software, but we can use the same principles to change a grid's influence given a unit's value and its surroundings. In this recipe, we will explore a couple of algorithms to modify a grid using matrix filters.

Getting ready

It is important to have grasped the concept of influence maps before implementing this recipe, so that you can understand the context in which it is applied.

How to do it...

We will implement the `Convolve` function:

1. Declare the `Convolve` function:

```
public static void Convolve(
        float[,] matrix,
        ref float[,] source,
        ref float[,] destination)
{
    // next steps
}
```

2. Initialize the variables for handling the computations and traversal of arrays:

```
int matrixLength = matrix.GetLength(0);
int size = (int)(matrixLength - 1) / 2;
int height = source.GetLength(0);
int width = source.GetLength(1);
int I, j, k, m;
```

3. Create the first loop for iterating over the destination and source grids:

```
for (i = 0; i < width-- size; i++)
{
    for (j = 0; j < height-- size; j++)
    {
        // next steps
    }
}
```

4. Implement the second loop for iterating over the filter matrix:

```
destination[i, j] = 0f;
for (k = 0; k < matrixLength; k++)
{
    for (m = 0; m < matrixLength; m++)
    {
        int row = i + k-- size;
        int col = j + m-- size;
        float aux = source[row, col] * matrix[k,m];
        destination[i, j] += aux;
    }
}
```

How it works...

We create a new grid to be swapped with the original source grid after the application of the matrix filter on each position. Then, we iterate over each position that is to be created as a destination grid and compute its result, taking the original grid's value and applying the matrix filter to it.

It is important to note that the matrix filter must be an odd-square array for the algorithm to work as expected.

There is more...

The following `ConvolveDriver` function helps us iterate using the `Convolve` function implemented previously:

1. Declare the `ConvolveDriver` function:

```
public static void ConvolveDriver(
        float[,] matrix,
        ref float[,] source,
        ref float[,] destination,
        int iterations)
{
    // next steps
}
```

2. Create the auxiliary variables for holding the grids:

```
float[,] map1;
float[,] map2;
int i;
```

3. Swap the maps, regardless of whether the iterations are odd or even:

```
if (iterations % 2 == 0)
{
    map1 = source;
    map2 = destination;
}
else
{
    destination = source;
    map1 = destination;
    map2 = source;
}
```

4. Apply the previous function during the iterations and swap:

```
for (i = 0; i < iterations; i++)
{
    Convolve(matrix, ref source, ref destination);
    float[,] aux = map1;
    map1 = map2;
    map2 = aux;
}
```

See also

- The *Introducing influence maps* recipe

Building a fighting circle

This recipe is based on the Kung-Fu Circle algorithm devised for the game, *Kingdoms of Amalur: Reckoning*. Its purpose is to offer an intelligent way for enemies to approach a given player and set attacks on it. It is very similar to the formation recipe, but it uses a stage manager that handles approach and attack permissions based on enemy weights and attack weights. It is also implemented so that the manager has the capability to handle a list of fighting circles; this is especially aimed at multiplayer games.

Getting ready

Before implementing the fighting circle algorithm, it is important to create some components that accompany the technique. First, the `Attack` class is a pseudo-abstract class for creating general-purpose attacks for each enemy, and it works as a template for our custom attacks in our game. Second, we need the `Enemy` class, which is the holder of the enemy's logic and requests. As we will see, the `Enemy` class holds a list of the different attack components found in the game object.

The code for the `Attack` class is as follows:

```
using UnityEngine;
using System.Collections;

public class Attack : MonoBehaviour
{
    public int weight;

    public virtual IEnumerator Execute()
    {
        // your attack behaviour here
        yield break;
    }
}
```

The steps to build the `Enemy` component are as follows:

1. Create the `Enemy` class:

```
using UnityEngine;
using System.Collections;

public class Enemy : MonoBehaviour
{
    public StageManager stageManager;
    public int slotWeight;
    [HideInInspector]
    public int circleId = -1;
    [HideInInspector]
    public bool isAssigned;
    [HideInInspector]
    public bool isAttacking;
    [HideInInspector]
    public Attack[] attackList;
}
```

2. Implement the `Start` function:

```
void Start()
{
    attackList = gameObject.GetComponents<Attack>();
}
```

3. Implement the function for assigning a target fighting circle:

```
public void SetCircle(GameObject circleObj = null)
{
    int id = -1;
    if (circleObj == null)
    {
        Vector3 position = transform.position;
        id = stageManager.GetClosestCircle(position);
    }
    else
    {
        FightingCircle fc;
        fc = circleObj.GetComponent<FightingCircle>();
        if (fc != null)
            id = fc.gameObject.GetInstanceID();
    }
    circleId = id;
}
```

4. Define the function for requesting a slot to the manager:

```
public bool RequestSlot()
{
    isAssigned = stageManager.GrantSlot(circleId, this);
    return isAssigned;
}
```

5. Define the function for releasing the slot from the manager:

```
public void ReleaseSlot()
{
    stageManager.ReleaseSlot(circleId, this);
    isAssigned = false;
    circleId = -1;
}
```

6. Implement the function for requesting an attack from the list (the order is the same from the Inspector):

```
public bool RequestAttack(int id)
{
    return stageManager.GrantAttack(circleId, attackList[id]);
}
```

7. Define the virtual function for the attack behavior:

```
public virtual IEnumerator Attack()
{
    // TODO
    // your attack behaviour here
    yield break;
}
```

How to do it...

Now, we implement the `FightingCircle` and `StageManager` classes

1. Create the `FightingCircle` class along with its member variables:

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

public class FightingCircle : MonoBehaviour
{
    public int slotCapacity;
    public int attackCapacity;
    public float attackRadius;
    public GameObject player;
    [HideInInspector]
    public int slotsAvailable;
    [HideInInspector]
    public int attackAvailable;
    [HideInInspector]
    public List<GameObject> enemyList;
    [HideInInspector]
    public Dictionary<int, Vector3> posDict;
}
```

2. Implement the `Awake` function for initialization:

```
void Awake()
{
    slotsAvailable = slotCapacity;
    attackAvailable = attackCapacity;
    enemyList = new List<GameObject>();
    posDict = new Dictionary<int, Vector3>();
    if (player == null)
        player = gameObject;
}
```

3. Define the `Update` function so that the slots' positions get updated:

```
void Update()
{
    if (enemyList.Count == 0)
        return;
    Vector3 anchor = player.transform.position;
    int i;
    for (i = 0; i < enemyList.Count; i++)
    {
        Vector3 position = anchor;
        Vector3 slotPos = GetSlotLocation(i);
        int enemyId = enemyList[i].GetInstanceID();
        position += player.transform.TransformDirection(slotPos);
        posDict[enemyId] = position;
    }
}
```

4. Implement the function for adding enemies to the circle:

```
public bool AddEnemy(GameObject enemyObj)
{
    Enemy enemy = enemyObj.GetComponent<Enemy>();
    int enemyId = enemyObj.GetInstanceID();
    if (slotsAvailable < enemy.slotWeight)
        return false;
    enemyList.Add(enemyObj);
    posDict.Add(enemyId, Vector3.zero);
    slotsAvailable -= enemy.slotWeight;
    return true;
}
```

5. Implement the function for removing enemies from the circle:

```
public bool RemoveEnemy(GameObject enemyObj)
{
    bool isRemoved = enemyList.Remove(enemyObj);
    if (isRemoved)
    {
        int enemyId = enemyObj.GetInstanceID();
        posDict.Remove(enemyId);
        Enemy enemy = enemyObj.GetComponent<Enemy>();
        slotsAvailable += enemy.slotWeight;
    }
    return isRemoved;
}
```

6. Implement the function for swapping enemy positions in the circle:

```
public void SwapEnemies(GameObject enemyObjA, GameObject
enemyObjB)
{
    int indexA = enemyList.IndexOf(enemyObjA);
    int indexB = enemyList.IndexOf(enemyObjB);
    if (indexA != -1 && indexB != -1)
    {
        enemyList[indexB] = enemyObjA;
        enemyList[indexA] = enemyObjB;
    }
}
```

7. Define the function for getting an enemy's spatial position according to the circle:

```
public Vector3? GetPositions(GameObject enemyObj)
{
    int enemyId = enemyObj.GetInstanceID();
    if (!posDict.ContainsKey(enemyId))
        return null;
    return posDict[enemyId];
}
```

8. Implement the function for computing the spatial location of a slot:

```
private Vector3 GetSlotLocation(int slot)
{
    Vector3 location = new Vector3();
    float degrees = 360f / enemyList.Count;
    degrees *= (float)slot;
    location.x = Mathf.Cos(Mathf.Deg2Rad * degrees);
    location.x *= attackRadius;
    location.z = Mathf.Cos(Mathf.Deg2Rad * degrees);
    location.z *= attackRadius;
    return location;
}
```

9. Implement the function for virtually adding attacks to the circle:

```
public bool AddAttack(int weight)
{
    if (attackAvailable - weight < 0)
        return false;
    attackAvailable -= weight;
    return true;
}
```

10. Define the function for virtually releasing the attacks from the circle:

```
public void ResetAttack()
{
    attackAvailable = attackCapacity;
}
```

11. Now, create the `StageManager` class:

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

public class StageManager : MonoBehaviour
{
    public List<FightingCircle> circleList;
    private Dictionary<int, FightingCircle> circleDic;
    private Dictionary<int, List<Attack>> attackRqsts;
}
```

12. Implement the `Awake` function for initialization:

```
void Awake()
{
    circleList = new List<FightingCircle>();
    circleDic = new Dictionary<int, FightingCircle>();
    attackRqsts = new Dictionary<int, List<Attack>>();
    foreach(FightingCircle fc in circleList)
    {
        AddCircle(fc);
    }
}
```

13. Create the function for adding circles to the manager:

```
public void AddCircle(FightingCircle circle)
{
    if (!circleList.Contains(circle))
        return;
    circleList.Add(circle);
    int objId = circle.gameObject.GetInstanceID();
    circleDic.Add(objId, circle);
    attackRqsts.Add(objId, new List<Attack>());
}
```

14. Also, create the function for removing circles from the manager:

```
public void RemoveCircle(FightingCircle circle)
{
    bool isRemoved = circleList.Remove(circle);
    if (!isRemoved)
        return;
    int objId = circle.gameObject.GetInstanceID();
    circleDic.Remove(objId);
    attackRqsts[objId].Clear();
    attackRqsts.Remove(objId);
}
```

15. Define the function for getting the closest circle, if given a position:

```
public int GetClosestCircle(Vector3 position)
{
    FightingCircle circle = null;
    float minDist = Mathf.Infinity;
    foreach(FightingCircle c in circleList)
    {
        Vector3 circlePos = c.transform.position;
        float dist = Vector3.Distance(position, circlePos);
        if (dist < minDist)
        {
            minDist = dist;
            circle = c;
        }
    }
    return circle.gameObject.GetInstanceID();
}
```

16. Define the function for granting an enemy a slot in a given circle:

```
public bool GrantSlot(int circleId, Enemy enemy)
{
    return circleDic[circleId].AddEnemy(enemy.gameObject);
}
```

17. Implement the function for releasing an enemy from a given circle ID:

```
public void ReleaseSlot(int circleId, Enemy enemy)
{
    circleDic[circleId].RemoveEnemy(enemy.gameObject);
}
```

18. Define the function for granting attack permissions and adding them to the manager:

```
public bool GrantAttack(int circleId, Attack attack)
{
    bool answer = circleDic[circleId].AddAttack(attack.weight);
    attackRqsts[circleId].Add(attack);
    return answer;
}
```

19. Step:

```
public IEnumerator ExecuteAtacks()
{
    foreach (int circle in attackRqsts.Keys)
    {
        List<Attack> attacks = attackRqsts[circle];
        foreach (Attack a in attacks)
            yield return a.Execute();
    }
    foreach (FightingCircle fc in circleList)
        fc.ResetAttack();
}
```

How it works...

The Attack and Enemy classes control the behaviors when needed, so the Enemy class can be called from another component in the game object. The `FightingCircle` class is very similar to `FormationPattern`, in that it computes the target positions for a given enemy. It just does it in a slightly different way. Finally, the `StageManager` grants all the necessary permissions for assigning and releasing enemy and attack slots for each circle.

There is more...

It is worth noting that the fighting circle can be added as a component of a game object that works as the target player itself, or a different empty object that holds a reference to the player's game object.

Also, you could move the functions for granting and executing attacks to the fighting circle. We wanted to keep them in the manager so that attack executions are centralized, and the circles just handle target positions, just like formations.

See also

- Refer to the *Handling formations* recipe
- For further information on the Kung-Fu Circle algorithm, please refer to the book, *Game AI Pro*, by Steve Rabin