

# 4

## Finding Your Way

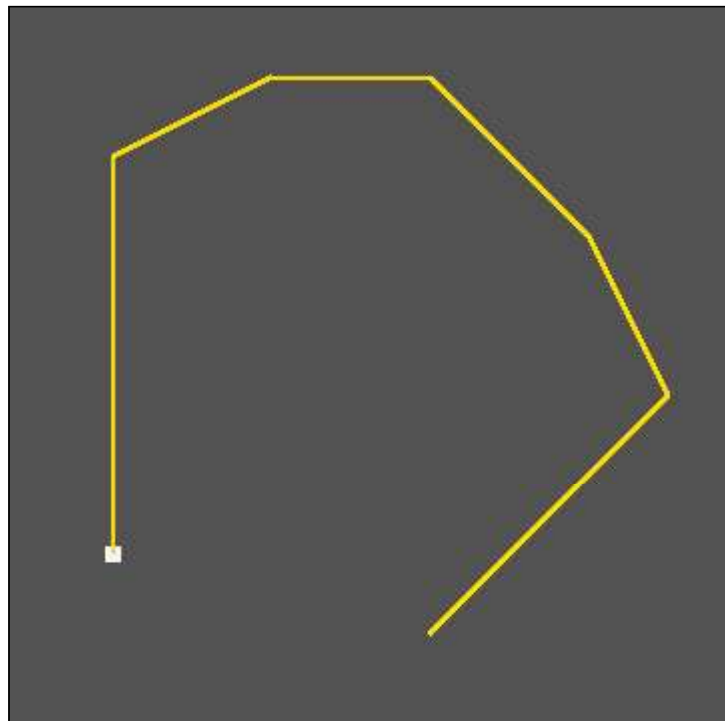
Obstacle avoidance is a simple behavior for the AI entities to reach a target point. It's important to note that the specific behavior implemented in this chapter is meant to be used for behaviors such as crowd simulation, where the main objective of each agent entity is just to avoid the other agents and reach the target. There's no consideration on what would be the most efficient and shortest path. We'll learn about the A\* Pathfinding algorithm in the next section.

In this chapter, we will cover the following topics:

- Path following and steering
- A custom A\* Pathfinding implementation
- Unity's built-in NavMesh

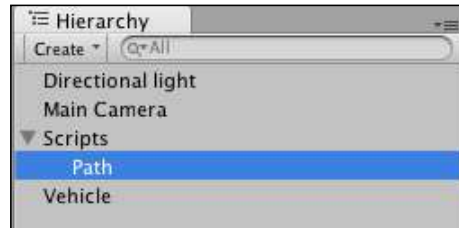
## Following a path

Paths are usually created by connecting waypoints together. So, we'll set up a simple path, as shown in the following screenshot, and then make our cube entity follow along the path smoothly. Now, there are many ways to build such a path. The one we are going to implement here could arguably be the simplest one. We'll write a script called `Path.cs` and store all the waypoint positions in a `Vector3` array. Then, from the editor, we'll enter those positions manually. It's bit of a tedious process right now. One option is to use the position of an empty game object as waypoints. Or, if you want, you can create your own editor plugins to automate these kind of tasks, but that is outside the scope of this book. For now, it should be fine to just enter the waypoint information manually, since the number of waypoints that we are creating here are not that substantial.



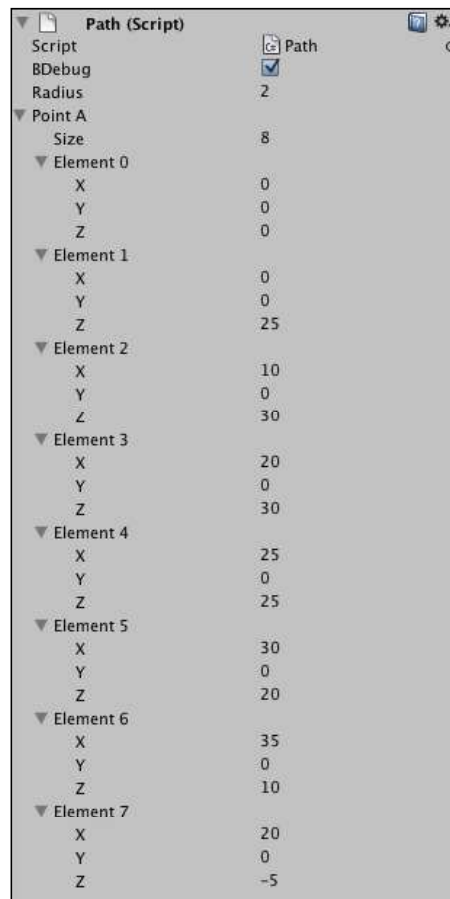
An object path

First, we create an empty game entity and add our path script component, as shown in the following screenshot:



The organized Hierarchy

Then, we populate our **Point A** variable with all the points we want to be included in our path:



Properties of our path script

The preceding list shows the waypoints needed to create the path that was described earlier. The other two properties are `debug mode` and `radius`. If the `debug mode` property is checked, the path formed by the positions entered will be drawn as gizmos in the editor window. The `radius` property is a range value for the path-following entities to use so that they can know when they've reached a particular waypoint if they are in this radius range. Since to reach an exact position can be pretty difficult, this range radius value provides an effective way for the path-following agents to navigate through the path.

## The path script

So, let's take a look at the path script itself. It will be responsible for managing the path for our objects. Look at the following code in the `Path.cs` file:

```
using UnityEngine;
using System.Collections;

public class Path : MonoBehaviour {
    public bool bDebug = true;
    public float Radius = 2.0f;
    public Vector3[] pointA;

    public float Length {
        get {
            return pointA.Length;
        }
    }

    public Vector3 GetPoint(int index) {
        return pointA[index];
    }

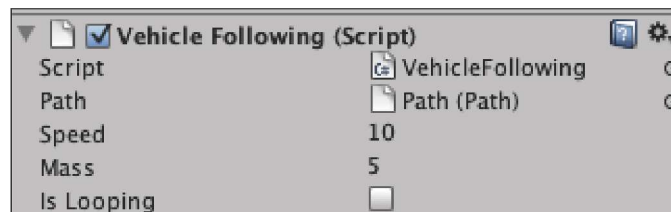
    void OnDrawGizmos() {
        if (!bDebug) return;

        for (int i = 0; i < pointA.Length; i++) {
            if (i + 1 < pointA.Length) {
                Debug.DrawLine(pointA[i], pointA[i + 1],
                    Color.red);
            }
        }
    }
}
```

As you can see, this is a very simple script. It has a `Length` property that returns the length and size of the waypoint array if requested. The `GetPoint` method returns the `Vector3` position of a particular waypoint at a specified index in the array. Then, we have the `OnDrawGizmos` method that is called by Unity frame to draw components in the editor environment. The drawing here won't be rendered in the game view unless gizmos, located in the top-right corner of the game view, are turned on.

## Using the path follower

Next, we have our vehicle entity, which is just a simple cube object in this example. We can replace the cube later with whatever 3D models we want. After we create the script, we add the `VehicleFollowing` script component, as shown in the following screenshot:



The properties of our `VehicleFollowing` script

The script takes a couple of parameters. First is the reference to the path object it needs to follow. Then, the `Speed` and `Mass` properties, which are needed to calculate its acceleration properly. The `IsLooping` flag is a flag that makes this entity follow the path continuously if it's checked. Let's take a look at the following code in the `VehicleFollowing.cs` file:

```
using UnityEngine;
using System.Collections;

public class VehicleFollowing : MonoBehaviour {
    public Path path;
    public float speed = 20.0f;
    public float mass = 5.0f;
    public bool isLooping = true;

    //Actual speed of the vehicle
    private float curSpeed;

    private int curPathIndex;
    private float pathLength;
    private Vector3 targetPoint;

    Vector3 velocity;
```

First, we initialize the properties and set up the direction of our velocity vector with the entity's forward vector in the Start method, as shown in the following code:

```
void Start () {
    pathLength = path.Length;
    curPathIndex = 0;

    //get the current velocity of the vehicle
    velocity = transform.forward;
}
```

There are only two methods that are important in this script, the Update and Steer methods. Let's take a look at the following code:

```
void Update () {
    //Unify the speed
    curSpeed = speed * Time.deltaTime;

    targetPoint = path.GetPoint(curPathIndex);

    //If reach the radius within the path then move to next
    //point in the path
    if (Vector3.Distance(transform.position, targetPoint) <
        path.Radius) {
        //Don't move the vehicle if path is finished
        if (curPathIndex < pathLength - 1) curPathIndex++;
        else if (isLooping) curPathIndex = 0;
        else return;
    }

    //Move the vehicle until the end point is reached in
    //the path
    if (curPathIndex >= pathLength ) return;

    //Calculate the next Velocity towards the path
    if (curPathIndex >= pathLength-1&& !isLooping)
        velocity += Steer(targetPoint, true);
    else velocity += Steer(targetPoint);

    //Move the vehicle according to the velocity
    transform.position += velocity;
    //Rotate the vehicle towards the desired Velocity
    transform.rotation = Quaternion.LookRotation(velocity);
}
```

In the `Update` method, we check whether our entity has reached a particular waypoint by calculating the distance between its current position and the path's radius range. If it's in the range, we just increase the index to look it up from the waypoints array. If it's the last waypoint, we check if the `isLooping` flag is set. If it is set, we set the target to the starting waypoint; otherwise, we just stop at that point. Though, if we wanted, we could make it so that our object turned around and went back the way it came. In the next part, we will calculate the acceleration from the `Steer` method. Then, we rotate our entity and update the position according to the speed and direction of the velocity:

```
//Steering algorithm to steer the vector towards the target
public Vector3 Steer(Vector3 target,
    bool bFinalPoint = false) {
    //Calculate the directional vector from the current
    //position towards the target point
    Vector3 desiredVelocity = (target - transform.position);
    float dist = desiredVelocity.magnitude;

    //Normalise the desired Velocity
    desiredVelocity.Normalize();

    //Calculate the velocity according to the speed
    if (bFinalPoint && dist < 10.0f) desiredVelocity *=
        (curSpeed * (dist / 10.0f));
    else desiredVelocity *= curSpeed;

    //Calculate the force Vector
    Vector3 steeringForce = desiredVelocity - velocity;
    Vector3 acceleration = steeringForce / mass;

    return acceleration;
}
```

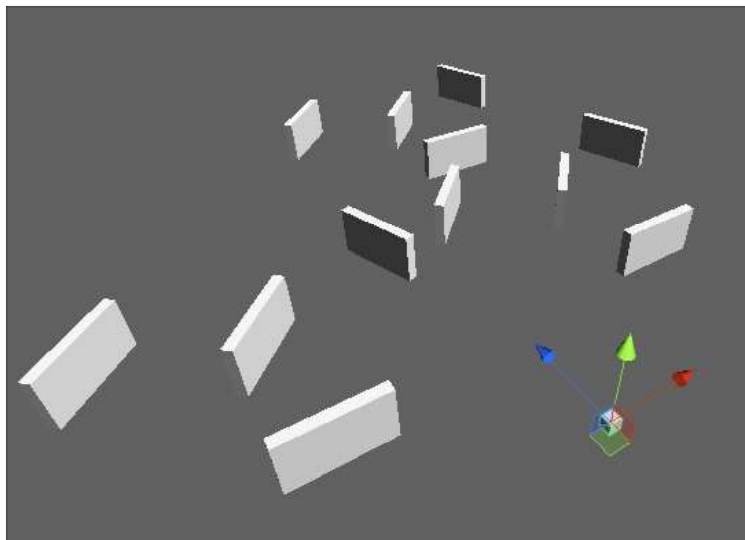
The `Steer` method takes the parameter `target`, which is a `Vector3` position representing the final waypoint in the path. The first thing we do is to calculate the remaining distance from the current position to the target position. The target position vector minus the current position vector gives a vector toward the target position vector. The magnitude of this vector is the remaining distance. We then normalize this vector just to preserve the `direction` property. Now, if this is the final waypoint, and the distance is less than 10 of a number we just decided to use, we slow down the velocity gradually according to the remaining distance to our point until the velocity finally becomes zero, otherwise, we just update the target velocity with the specified speed value. By subtracting the current velocity vector from this target velocity vector, we can calculate the new steering vector. Then, by dividing this vector with the mass value of our entity, we get the acceleration.

If you run the scene, you should see your cube object following the path. You can also see the path that is drawn in the editor view. Play around with the speed and mass value of the follower and radius values of the path and see how they affect the overall behavior of the system.

## Avoiding obstacles

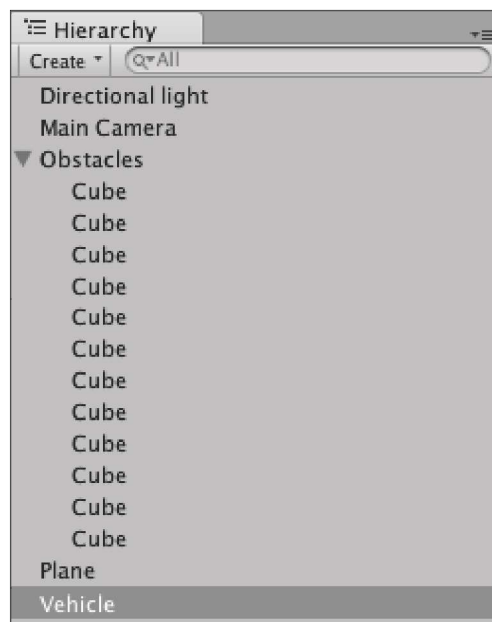
In this section, we'll set up a scene, as shown in the following screenshot, and make our AI entity avoid the obstacles while trying to reach the target point. The algorithm presented here using the raycasting method is very simple, so it can only avoid the obstacles blocking the path in front of it. The following screenshot will show us our scene:





A sample scene setup

To create this, we make a few cube entities and group them under an empty game object called `Obstacles`. We also create another cube object called `Agent` and give it our obstacle avoidance script. We then create a ground plane object to assist in finding a target position.



The organized Hierarchy

It is worth noting that this `Agent` object is not a pathfinder. As such, if we set too many walls up, our `Agent` might have a hard time finding the target. Try a few wall setups and see how our `Agent` performs.

## Adding a custom layer

We will now add a custom layer to our object. To add a new layer, we navigate to **Edit | Project Settings | Tags**. Assign the name `Obstacles` to **User Layer 8**. Now, we go back to our cube entity and set its `layer` property to `Obstacles`.



Creating a new layer

This is our new layer, which is added to Unity. Later, when we do the raycasting to detect obstacles, we'll only check for these entities using this particular layer. This way, we can ignore some objects that are not obstacles that are being hit by a ray, such as bushes or vegetation.



Assigning our new layer

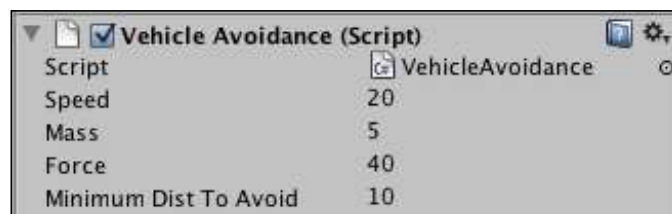
For larger projects, our game objects probably already have a layer assigned to them. So, instead of changing the object's layer to `Obstacles`, we would instead make a list using bitmaps of layers for our cube entity to use when detecting obstacles. We will talk more about bitmaps in the next section.



Layers are most commonly used by cameras to render a part of the scene, and by lights to illuminate only some parts of the scene. But, they can also be used by raycasting to selectively ignore colliders or create collisions. You can learn more about this at <http://docs.unity3d.com/Documentation/Components/Layers.html>.

## Implementing the avoidance logic

Now it is time to make the script that will help our cube entity avoid these walls.



The properties of our VehicleAvoidance (script)

As usual, we first initialize our entity script with the default properties and draw a GUI text in our `OnGUI` method. Let's take a look at the following code in the `VehicleAvoidance.cs` file:

```
using UnityEngine;
using System.Collections;

public class VehicleAvoidance : MonoBehaviour {
    public float speed = 20.0f;
    public float mass = 5.0f;
    public float force = 50.0f;
    public float minimumDistToAvoid = 20.0f;

    //Actual speed of the vehicle
    private float curSpeed;
    private Vector3 targetPoint;

    // Use this for initialization
    void Start () {
        mass = 5.0f;
        targetPoint = Vector3.zero;
    }

    void OnGUI() {
        GUILayout.Label("Click anywhere to move the vehicle.");
    }
}
```

Then in our `Update` method, we update the agent entity's position and rotation, based on the direction vector returned by the `AvoidObstacles` method:

```
//Update is called once per frame
void Update () {
    //Vehicle move by mouse click
    RaycastHit hit;
    var ray = Camera.main.ScreenPointToRay
        (Input.mousePosition);

    if (Input.GetMouseButtonDown(0) &&
        Physics.Raycast(ray, out hit, 100.0f)) {
        targetPoint = hit.point;
    }

    //Directional vector to the target position
    Vector3 dir = (targetPoint - transform.position);
    dir.Normalize();

    //Apply obstacle avoidance
    AvoidObstacles(ref dir);

    //...
}
```

The first thing we do in our `Update` method is retrieve the mouse click position so that we can move our AI entity. We do this by shooting a ray from the camera in the direction it's looking. Then, we take the point where the ray hit the ground plane as our target position. Once we get the target position vector, we can calculate the direction vector by subtracting the current position vector from the target position vector. Then, we call the `AvoidObstacles` method and pass in this direction vector:

```
//Calculate the new directional vector to avoid
//the obstacle
public void AvoidObstacles(ref Vector3 dir) {
    RaycastHit hit;

    //Only detect layer 8 (Obstacles)
    int layerMask = 1<<8;

    //Check that the vehicle hit with the obstacles within
    //it's minimum distance to avoid
    if (Physics.Raycast(transform.position,
        transform.forward, out hit,
```

---

```

        minimumDistToAvoid, layerMask)) {
//Get the normal of the hit point to calculate the
//new direction
Vector3 hitNormal = hit.normal;
hitNormal.y = 0.0f; //Don't want to move in Y-Space

//Get the new directional vector by adding force to
//vehicle's current forward vector
dir = transform.forward + hitNormal * force;
    }
}
}

```

The `AvoidObstacles` method is also quite simple. The only trick to note here is that raycasting interacts selectively with the `Obstacles` layer that we specified at **User Layer 8** in our `Unity TagManager`. The `Raycast` method accepts a layer mask parameter to determine which layers to ignore and which to consider during raycasting. Now, if you look at how many layers you can specify in `TagManager`, you'll find a total of 32 layers. Therefore, Unity uses a 32-bit integer number to represent this layer mask parameter. For example, the following would represent a zero in 32 bits:

```
0000 0000 0000 0000 0000 0000 0000 0000
```

By default, Unity uses the first eight layers as built-in layers. So, when you raycast without using a layer mask parameter, it'll raycast against all those eight layers, which could be represented like the following in a bitmask:

```
0000 0000 0000 0000 0000 0000 1111 1111
```

Our `Obstacles` layer was set at layer 8 (9th index), and we only want to raycast against this layer. So, we'd like to set up our bitmask in the following way:


```
0000 0000 0000 0000 0000 0001 0000 0000
```

The easiest way to set up this bitmask is by using the bit shift operators. We only need to place the 'on' bit or 1 at the 9th index, which means we can just move that bit 8 places to the left. So, we use the left shift operator to move the bit 8 places to the left, as shown in the following code:

```
int layerMask = 1<<8;
```

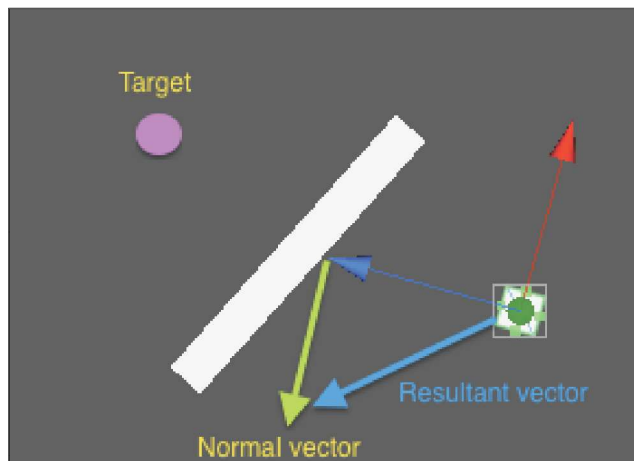
If we wanted to use multiple layer masks, say layer 8 and layer 9, an easy way would be to use the bitwise `OR` operator like this:

```
int layerMask = (1<<8) | (1<<9);
```

 You can also find a good discussion on using layermasks on Unity3D online. The question and answer site can be found at <http://answers.unity3d.com/questions/8715/how-do-i-use-layermasks.html>.

Once we have the layer mask, we call the `Physics.Raycast` method from the current entity's position and in the forward direction. For the length of the ray, we use our `minimumDistToAvoid` variable so that we'll only avoid those obstacles that are being hit by the ray within this distance.

Then, we take the normal vector of the hit ray, multiply it with the force vector, and add it to the current direction of our entity to get the new resultant direction vector, which we return from this method.



The cube entity avoids a wall

Then in our `Update` method, we use this new direction after avoiding obstacles to rotate the AI entity and update the position according to the speed value:

```
void Update () {  
  
    //...  
  
    //Don't move the vehicle when the target point  
    //is reached  
    if (Vector3.Distance(targetPoint,  
        transform.position) < 3.0f) return;  
  
    //Assign the speed with delta time
```

```
        curSpeed = speed * Time.deltaTime;

        //Rotate the vehicle to its target
        //directional vector
        var rot = Quaternion.LookRotation(dir);
        transform.rotation = Quaternion.Slerp
            (transform.rotation, rot, 5.0f *
            Time.deltaTime);

        //Move the vehicle towards
        transform.position += transform.forward *
            curSpeed;
    }
```

## A\* Pathfinding

Next up, we'll be implementing the A\* algorithm in a Unity environment using C#. The A\* Pathfinding algorithm is widely used in games and interactive applications even though there are other algorithms, such as Dijkstra's algorithm, because of its simplicity and effectiveness. We've briefly covered this algorithm previously in *Chapter 1, The Basics of AI in Games*, but let's review the algorithm again from an implementation perspective.

## Revisiting the A\* algorithm

Let's review the A\* algorithm again before we proceed to implement it in the next section. First, we'll need to represent the map in a traversable data structure. While many structures are possible, for this example, we will use a 2D grid array. We'll implement the `GridManager` class later to handle this map information. Our `GridManager` class will keep a list of the `Node` objects that are basically tiles in a 2D grid. So, we need to implement that `Node` class to handle things such as node type (whether it's a traversable node or an obstacle), cost to pass through and cost to reach the goal `Node`, and so on.