Lab 7 - Path Following and Steering

COMP396-Game Programming 2

Purpose: Implement Path Following and Steering Behaviours

Contents

1 Introduction

Let's implement:

- Following a Path
  - Demo Scene
  - Path Script
  - Path Following Agents
  - Playtesting
- Avoiding Ostacles
  - Adding a Custom Layer
  - Obstacle Avoidance
  - Playtesting

2 Basic Path Following behavior

Definition: A **path** is a sequence of points in the game, connecting a point **A** to a point **B**.

2.1 Demo Scene

Set up a demo scene, named **Lab7_{YourInitials}_PathFollowing**, with the following game objects:

Game Object	Type	Parent	P(x,y,z)	R(x,y,z)	S(x,y,z)	Color
Floor	Plane	Root	(0,0,0)	(0,0,0)	(20,1,20)	Beige
Path	Empty	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
Waypoint1-4	Empty	Path	Random	(0,0,0)	(1,1,1)	-
Vehicle	Cube	Root	Waypoint1	Random	Random	White

Make sure we put markers in the waypoints so we can see them in scene view.

- *Take Snapshot*

Let's write a **Path.cs** script that takes a **list** of game objects as waypoints and creates a **path** out of them.

2.2 Path script

```
using UnityEngine;
public class Path : MonoBehaviour {
 public bool isDebug = true;
 public bool isLoop = true;
 public Transform[] waypoints;
 public int Length {
  get {
   return waypoints.Length;
  }
 }
 public Vector3 GetPoint(int index) {
  return waypoints[index].position;
 }
 void OnDrawGizmos() {
  if (!isDebug)
   return;
  for (int i = 1; i < waypoints.Length; i++) {
    Debug.DrawLine(waypoints[i-1].position, waypoints[i].position, Color.red);
  }
  if (isLoop)
  {
    Debug.DrawLine(waypoints[this.Length - 1].position, waypoints[0].position,
Color.red);
```

```
  }
 }
}
```

> Notice the members :
>
> - **Length** property - returns the number of waypoints.
> - **GetPoint** method - returns the position of a particular waypoint at a specified index in the array.
> - **OnDrawGizmos** method - draws lines between waypoints, making the path visible in the editor environment. The drawing here won't be rendered in the game view unless the gizmos flag, located in the top right corner, is turned on.
> - **isDebug** - check it to draw the path (lines between waypoints).
> - **isLoop** - check it to draw a closed path (loop)

- *Take Snapshot* - Select the Path object. - Fill the **Waypoints** array in the Inspector with the actual waypoint markers.

Let's create a character that can follow this path. We do that in the following section.

2.3 Path-following agents

Our character is represented the cube named **Vehicle**. - Let's start by creating a **VehicleFollowing.cs** script. - First, we specify all the script properties.

```
using UnityEngine;
public class VehicleFollowing : MonoBehaviour {
  public Path path;
  public float speed = 10.0f;
  [Range(1.0f, 1000.0f)]
  public float steeringInertia = 100.0f;
  public bool isLooping = true;
  public float waypointRadius = 1.0f;
  //Actual speed of the vehicle
  private float curSpeed;
  private int curPathIndex = 0;
  private int pathLength;
  private Vector3 targetPoint;
  Vector3 velocity;
  void Start () {
    pathLength = path.Length;
    velocity = transform.forward;
  }
  void Update() {
    //Unify the speed
    curSpeed = speed * Time.deltaTime;
    targetPoint = path.GetPoint(curPathIndex);
    //If reach the radius of the waypoint then move to next point in the path
    if (Vector3.Distance(transform.position, targetPoint) < waypointRadius) {
      //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);
  }
  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;
    //Normalize the desired Velocity
    desiredVelocity.Normalize();
    //
    if (bFinalPoint && dist < waypointRadius)
      desiredVelocity *= curSpeed * (dist / waypointRadius);
    else
      desiredVelocity *= curSpeed;
    //Calculate the force Vector
    Vector3 steeringForce = desiredVelocity - velocity;
    return steeringForce / steeringInertia;
  }
}
```

Notice that:

- We start with **public** members
  - Path **path**
  - float **speed**, **steeringInertia**, **waypointRadius**
  - bool **isLooping**
- Then we have some **private** members
  - float curSpeed
  - int curPathIndex, pathLength
  - Vector3 targetPoint, velocity
- We continue with definitions of **methods**
  - void Start() - initialize the pathLength and velocity (direction only)
  - void Update() -
  - Vector3 Steer(Vector3 target, bool finalPoint=true)

In the **Update** method, we check whether the entity has reached a particular waypoint by calculating if the **distance** between its current position and the **target** waypoint is smaller than the waypoint's **radius**. If it is, we increase the index, setting in this way the target position to the **next waypoint** in the waypoints array. If it was the last waypoint, we check the **isLooping** flag. If it is active, we set the destination to the **starting** waypoint; otherwise, we stop.

> An alternative solution (strategy) is to program it so that our object turns around and goes back the way it came. Take this as a **challenge** (see the **Challenges** section in the end).

Now, we calculate the **acceleration** and **rotation** of the entity using the Steer method. In this method, we rotate and update the entity's position according to the speed and direction of the velocity vector:

The **Steer** method takes two parameters: the **target** position and a boolean, which tells us whether this is the **final waypoint** in the path. At first, we calculate the **remaining distance** from the current position to the target position. Then we subtract the current position vector from the target position vector to get a vector pointing toward the target position. We are not interested in the vector's size, just in its **direction**, so we normalize it.

Now, suppose we are moving to the final waypoint, and its distance from us is less than the waypoint **radius**. In that case, we want to slow down gradually until the velocity becomes zero precisely at the waypoint position so that the character correctly stops in place (**arrival**). Otherwise, we update the target velocity with the desired maximum speed value. Then, in the same way as before, we can calculate the new **steering** vector by subtracting the current velocity vector from this target velocity vector. Finally, by dividing this vector by the steering **inertia** value of our entity, we get a smooth steering (note that the minimal value for the steering inertia is **1**, corresponding to instantaneous steering).

- Attach **VehicleFollowing.cs** script to the **Vehicle** game object.
- Playtest
- Notice that the vehicle (the cube) follows the path.
- Notice also the path in the editor view.
- Play around with the **speed**, **steering inertia**, **waypointRadius** values; see how they affect the system's overall behavior.
- *Take Snapshot*

3 Avoiding obstacles

Let's implement **obstacle avoidance**. We will:

- Create a demo scene with **obstacles**.
- Create a **script** for the main character (the Vehicle) to **avoid obstacles** while trying to reach the **target** point.
    - The algorithm presented here uses the raycasting method, which is very straightforward. However, this means it can only avoid obstacles that are blocking its path directly in front of it

3.1 Demo Scene

- Set up a demo scene named **Lab7_{YourInitials}_ObstacleAvoidance**, with the following game objects:

Game Object	Type	Parent	P(x,y,z)	R(x,y,z)	S(x,y,z)	Color
Floor	Plane	Root	(0,0,0)	(0,0,0)	(20,1,20)	Beige
Obstacles	Empty	Root	(0,0,0)	(0,0,0)	(1,1,1)	-
Obstacle 1-9	Cube	Obstacles	Random	Random	(1,1,1)	Brown
Vehicle	Cube	Root	Random	Random	Random	White

- *Take Snapshot*

> The **Vehicle** object does **not** perform pathfinding, that is, the active search for a path to the destination. Instead, it only **avoids** obstacles locally as it follows the path. Roughly speaking, it is the difference between you planning a path from your home to the mall, and avoiding the possible people and obstacles you may find along the path. As such, if we set too many walls up, the Vehicle might have a hard time finding the target: for instance, if the Agent ends up facing a dead-end in a **U-shaped** object, it may not be able to get out. Try a few different **wall setups** and see how your agent performs.

3.2 Adding a custom layer

We now add a custom layer to the Obstacles object:

1. Go to **Edit → Project Settings**
2. Go to the **Tags and Layer** section.
3. Assign the name **Obstacles** to **User Layer 8**.
4. We then go back to **Vehicle** game object and set its **Layers** property to **Obstacles**
5. Say **OK** when asked to propagate the **Layer** setting to the children (**Obstacle** game objects).
6. *Take Snapshot*

> When using **raycasting** to detect **obstacles**, we check for those entities, but only on this **layer**. This way, the physics system can ignore objects hit by a ray that are not an obstacle, such as bushes or vegetation

> ⓘ **INFO** In games, we use layers to let cameras render only a part of the scene or have lights illuminate only a subset of the objects. However, layers can also be

used by raycasting to ignore colliders selectively or to create collisions. You can learn more about this at https://docs.unity3d.com/Manual/Layers.html.

3.3 Obstacle avoidance script

Let's write the script **VehicleAvoidance.cs** that makes the cube entity avoid the walls.

```
using UnityEngine;
public class VehicleAvoidance : MonoBehaviour {
  public float vehicleRadius = 1.2f;
  public float speed = 10.0f;
  public float force = 50.0f;
  public float minimumDistToAvoid = 10.0f;
  public float targetReachedRadius = 3.0f;
  //Actual speed of the vehicle
  private float curSpeed;
  private Vector3 targetPoint;
  // Use this for initialization
  void Start() {
    targetPoint = Vector3.zero;
  }
  void OnGUI() {
    GUILayout.Label("Click anywhere to move the vehicle to the clicked point");
  }
  void Update() {
    //Vehicle move by mouse click
    var ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    if (Input.GetMouseButtonDown(0) && Physics.Raycast(ray, out var 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);

    //Don't move the vehicle when the target point is reached
    if (Vector3.Distance(targetPoint, transform.position) < targetReachedRadius)
      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;
    transform.position = new Vector3(transform.position.x, 0, transform.position.z);
  }
  public void AvoidObstacles(ref Vector3 dir) {
    //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.SphereCast(transform.position, vehicleRadius, transform.forward, out
var hit,
      minimumDistToAvoid, layerMask)) {
      //Get the normal of the hit point to calculate the new direction
      Vector3 hitNormal = hit.normal;
      //Don't want to move in Y-Space
      hitNormal.y = 0.0f;
      //Get the new directional vector by adding force to vehicle's current forward
vector
      dir = transform.forward + hitNormal * force;
    }
  }
}
```

We:

- Initialize our entity script with the default properties.
- Draw **GUI text** in our **OnGUI** method.
- In the **Update** method we:
  - retrieve the **position** of the mouse-click (via raycast from camera forward).
  - use this position to determine the desired **target position** of our character.
  - calculate the **direction** vector by subtracting the **current position** vector from the **target position** vector ($\overrightarrow{dir} = \overrightarrow{T_p} - \overrightarrow{C_p}$).
  - call the **AvoidObstacles** method passing this direction to it
- In the **AvoidObstacles** method we use another very useful Unity physics utility: a **SphereCast**.

> A **SphereCast** is similar to the **Raycast** but, instead of detecting a collider by firing a dimensionless **ray**, it fires a chunky **sphere**. In practice, a SphereCast gives **width** to the Raycast ray. We need this, because our character is not dimensionless. We want to be sure that the entire body of the character can avoid the collision.

> Also, the SphereCast interacts selectively with the **Obstacles** layer we specified at User Layer 8 in the Unity3D Tag Manager. The SphereCast method accepts a **layer mask** parameter to determine which layers to **ignore** and **consider** during raycasting.

3.3.1 Layer Masks

In the Layer Manager, there are a total of 32 layers. So, it's enough to use just one int (4 bytes = 4*8=32 bits), therefore using one bit per layer. A Mask bit works like a bitwise **and**: 0 | x => 0, 1 | 0 => 0, 1 | 1 => 1 (the pipe '|' is the bitwise **and** operator). The value **0** is the mask (output is **0** no matter what the actual value is), whereas the value **1** is the **see-**

through value (output of mask | value is the value itself). So to mask layers just put **0** in their corresponding bits.

Therefore, Unity3D 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, Unity3D uses the first eight layers as built-in layers. So, when you use a Raycast or a SphereCast without using a layer mask parameter, it detects every object in those eight layers. We can represent this interaction mask with a bitmask, as follows:

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

In this demo, we set the Obstacles layer as layer 8 (9th index). Because we only want to detect obstacles in this layer, we want to set up the 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, **1**, at the **9th** index, which means we can just move that bit eight places to the left. So, we use the **left shift** operator to move the bit **eight** 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, as follows:

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

> For more information on layer masks see also http://answers.unity3d.com/questions/8715/how-do-i-use-layermasks.html.

Once we have the layer mask, we call the **Physics.SphereCast** method from the current entity's position and in the forward direction. We use a sphere of radius **vehicleRadius** (make sure that is big enough to contain the cubic vehicle in its entirety) and a detection distance defined by the **minimumDistToAvoid** variable. In fact, we want to detect only the objects that are close enough to affect our movement. Then, we take the normal vector of the **hit** ray, multiply it with the **force** vector, and add it to the current direction of the entity to get the new resultant direction vector, which we return from this method. Then, in the **Update** method, we use this new **direction** to **rotate** the AI entity and update the position according to the **speed** value:

- Attach this new script to the **Vehicle** object.
- Playtest.
- Notice that the vehicle navigates across the plane around the obstacles without any trouble.
- *Take Snapshot*.
- Experiment with the Inspector parameters to tweak the vehicle behavior.

- *Take Snapshot* s.

4 Summary

In this chapter, we:

- set up two scenes
- studied how to build agents with
  - **path-following** behavior, and
  - **obstacle avoidance** behavior.
- Worked with the following Unity3D features:
  - **layers** and **layer masks**
  - **Raycasts**, and
  - **SphereCasts**
- To effectuate the movement needed we used **Steering** (application of Newton's Second Law)

In the next lab, we'll study how to implement a pathfinding algorithm, called A*, to determine the optimal path before moving, while still avoiding static obstacles.

Challenge(s): In path following section was decided this challenge:

- An alternative solution (strategy) is to program it so that our object turns around and goes back the way it came. Take this as a challenge.