COMP4002/G54GAM Lab Exercise 05 – AI

05/03/19

This lab exercise involves creating a third-person game with a simple AI (artificial intelligence) based enemy character that patrols and follows the player around the level.

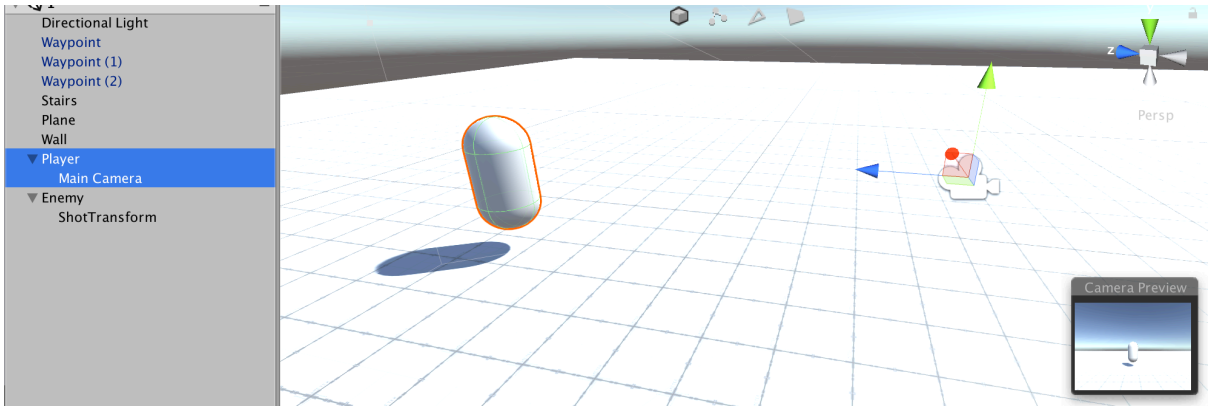
The game should have the following features:

- A third-person character with a basic avatar
- An enemy character with a simple AI behaviour that
    - Patrols between waypoints
    - Moves close to the player when they are seen
        - Fires on the player
    - Attempts to find the player again when lost, before returning to their original patrolling behaviour

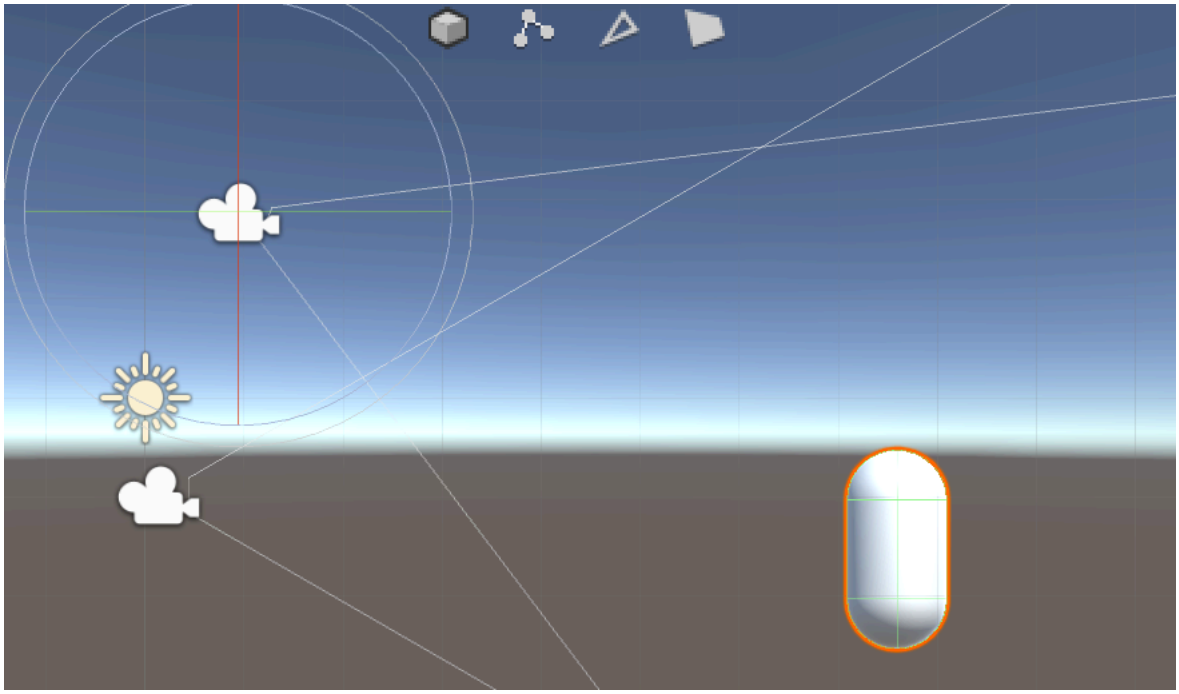
This exercise will introduce the use of navigation, or path finding, by AI characters to find their way around the level, and also *Finite State Machines* – a common software design pattern for controlling the behaviour of the AI character.

Third Person Character

The third-person character controller is similar in movement to the first-person character implemented in labs 03 and 04, however rather than looking through the "eyes" of the character it will have a third-person perspective from the viewpoint of a camera behind the character. This will require the character to now have a visible avatar – for this exercise this will just be a simple capsule shape, however could obviously be an avatar with animations.



This variant of the third-person character can again be moved forwards and backwards, left and right using the appropriate keys, and will rotate left and right using the mouse. The difference here is that the up and down movement of the mouse should again rotate the camera, but about the X-axis of the character, not the camera. This gives the player control over the camera in the sense that they can look from a higher or lower vantage point, but the camera is always made to point at the player character.



```
float y = 0.0f;

// Update is called once per frame
void Update () {

    float rotation = Input.GetAxis("Mouse X");
    transform.Rotate(0, 5.0f * rotation, 0);

    float updown = Input.GetAxis("Mouse Y");

    // clamp allowed rotation to 30
    if (y + updown > 50 || y + updown < 0)
    {
        updown = 0;
    }
```

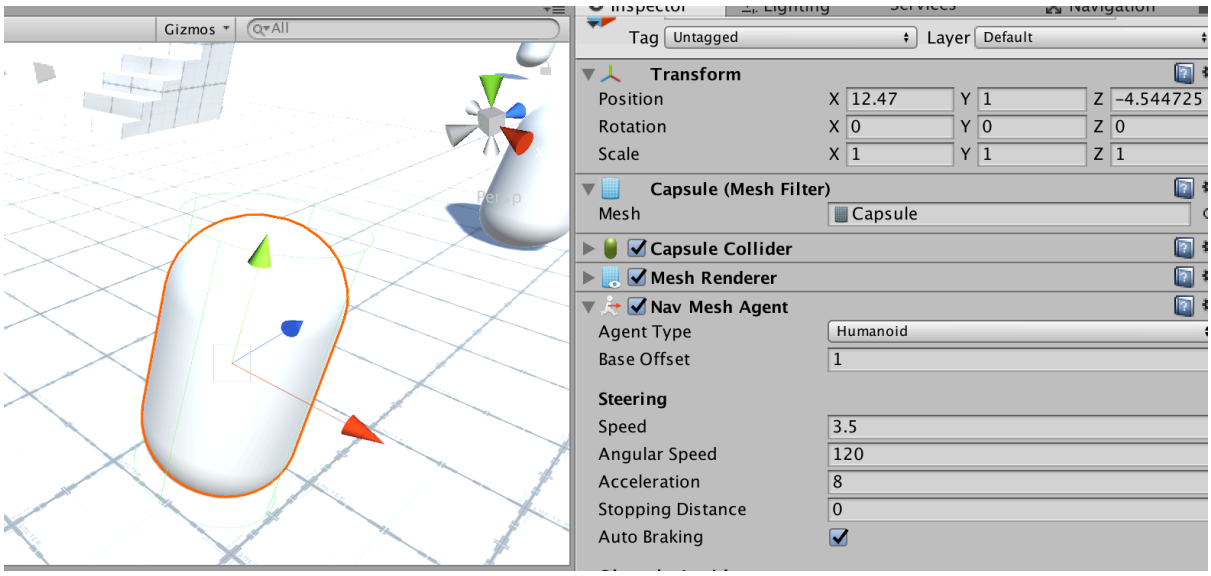
```
y += updown;

Camera.main.transform.RotateAround(transform.position,
    transform.right,
    updown);
Camera.main.transform.LookAt(transform);
```

The character, rather than an empty game object, can now have a visible shape. The camera should still be a child of the object but should be set back from the parent character by an appropriate distance, depending on how large you wish the character to appear in the camera view. Otherwise, the character remains the same as in lab 03.

Patrolling Enemy Character and NavMesh

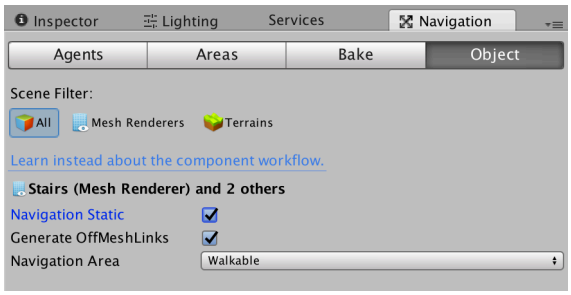
The main component of this game will be an enemy with a rudimentary AI behaviour. The hazards in lab 02 moved up and down driven by an animation curve, however here we will implement a character that can intelligently determine the best route to travel between a set of waypoints, allowing us to set up a patrolling route as its default behaviour.

- Create a game object to form the basis of the enemy character. This could simply be a capsule shape again. Unlike the player character it does not need a *CharacterController* component, instead add a *Nav Mesh Agent* component – this will be responsible for navigating the enemy, and moving the capsule object.

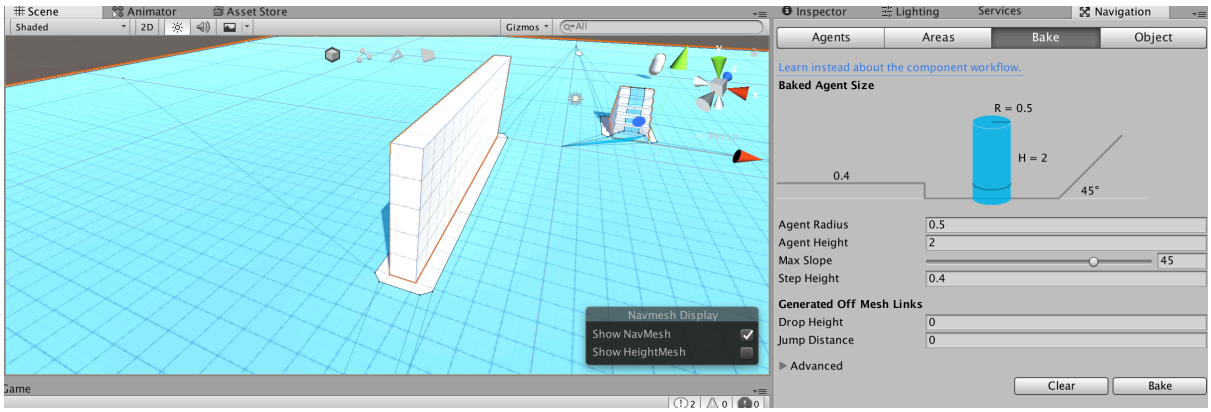


The NavMeshAgent can route from any point to any other point on the level provided they are connected via a *Nav Mesh* – this is a pre-computed routing map derived from the various objects in the level. Open the *Navigation* window (Window->Navigation).

- First, ensure that *Navigation Static* is enabled in the *Object* tab.



- Next, select objects in the scene that should be treated as terrain that can be navigated. In the screenshot, there is a simple plane and a wall. Clicking *Bake* in the *Bake* tab will pre-compute the routing information based on these objects, and if *Show NavMesh* is selected the areas that the enemy will be able to navigate will be shown in blue.



- Create an *EnemyController* script for the enemy character. With the NavMeshAgent component and the baked NavMesh in place, we can simply tell the character to move to the desired location (to start with just an empty game object), and it will do so based on the movement properties (speed etc) specified in the NavMeshAgent.

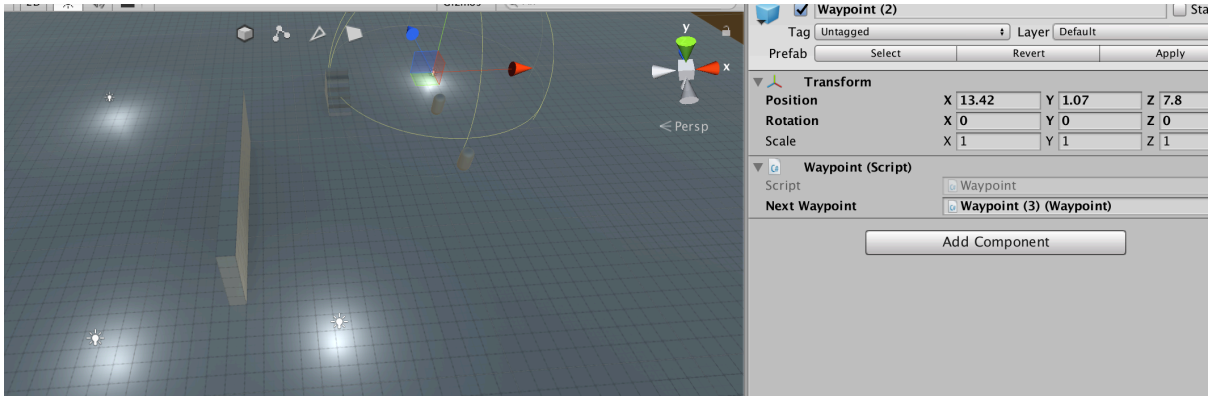
```
using UnityEngine.AI;

...

    NavMeshAgent agent;
    public Transform destination;


    // Use this for initialization
    void Start ()
    {
        agent = GetComponent<NavMeshAgent>();
        agent.destination = destination.position;
    }
```

Our AI character to start with will patrol around the level by navigating between a series of waypoints in sequence. The first step is to annotate the level – by introducing the locations that the AI must move to, and then make it move from one to another.

Create an empty game object and add a script called *Waypoint* to it. The script does not need to do anything but should contain a single public variable allowing us to reference where to go next.

```
public class Waypoint : MonoBehaviour {

    public Waypoint nextWaypoint;
}
```

This now allows us to construct a simple singly linked list of Waypoints, where each Waypoint contains a reference to the next. Make the object a *prefab*, and place three or four in the level and set the nextWaypoint variables so that the first waypoint points to the second, the second to the third etc., and the last back to the first, forming a circuit (or some permutation thereof). A light has been added to each waypoint in the screenshot to make them visible.



- Update the enemy controller script to navigate between these waypoints. The enemy should have a public variable to allow their first waypoint to be set, and then from then on, on reaching this waypoint, should update the target location to be the next waypoint referenced.

```
public Waypoint waypoint;
NavMeshAgent agent;

// Use this for initialization
void Start ()
{
    agent = GetComponent<NavMeshAgent>();
    agent.destination = waypoint.transform.position;
}

// Update is called once per frame
void Update () {

    if (!agent.pathPending && agent.remainingDistance < 0.5f)
    {
        Waypoint nextWaypoint = waypoint.nextWaypoint;
        waypoint = nextWaypoint;
        agent.destination = waypoint.transform.position;
    }
}
```

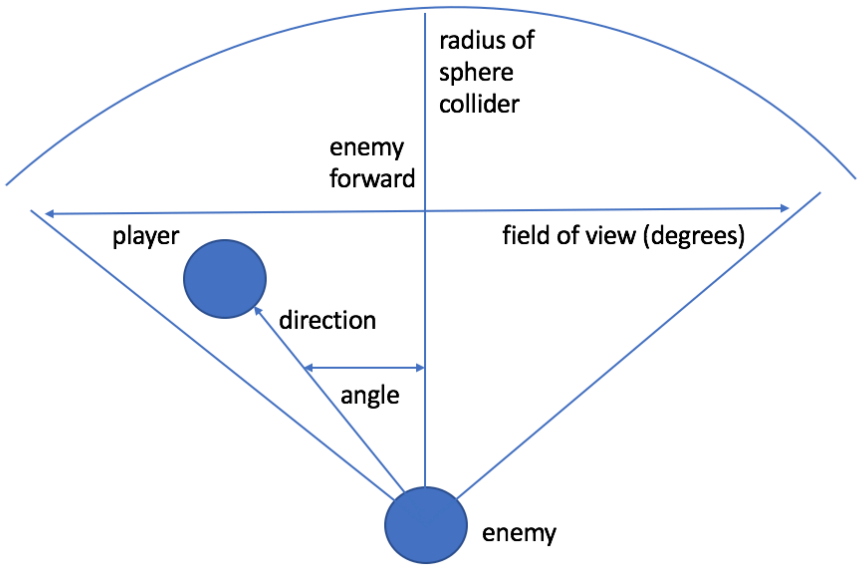
The benefit of using the NavMesh is that the waypoints do not need to be of line of sight of one another – the agent will correctly route around corners and over stairs as appropriate to reach the next destination.

AI Perception and Sight

To make a more challenging enemy for the player we will give it *sight* – i.e. if the player character is visible to the enemy – it should figure out where the player is, and if the player subsequently goes out of sight it should remember where it last saw them. As you might imagine sight working as a sense, the enemy should be able to see a certain distance, with a certain *field of view* (i.e. it won't have eyes in the back of its head), and it should not be able to see through walls (to allow the player to hide from it). However, you could vary or experiment with any of these to implement a different kind of perception.

Implementing perception for the enemy rather counterintuitively starts with the script knowing exactly where the player is, and then deciding whether we should be able to "see" them:

- Is the player within sight range of the enemy?
    - Calculate the angle between the player and the direction in which the enemy is currently facing
        - Is this less than the enemy's field of view?
            - Is there anything in between the enemy and the player?



- Add a *SphereCollider* component to the enemy and make it a *trigger*. This should be quite large (for example 10 units to begin with) as it will define the distance that the enemy can see.

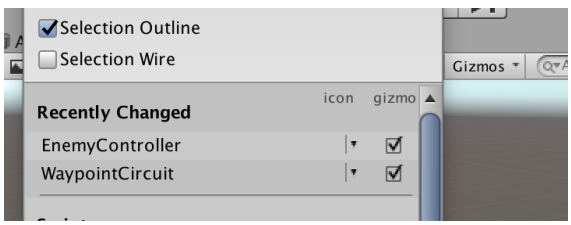
- Modify your enemy controller script to handle the *OnTriggerStay* event – this will fire for each object overlapping the enemy sphere collider, i.e. everything within our seeing range. You will need to create variables for the field of view (110 degrees is a good starting point), a flag to say that the target has been seen, the position at

which it was seen, and a public variable so that the player can be set as the target object. When the enemy is started, you will also need to store a reference to the collider component so that its radius can be used to set the length of the raycast.

https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnTriggerStay.html

```
private void OnTriggerStay(Collider other)
{
    // is it the player?
    if(other.gameObject == target)
    {
        // angle between us and the player
        Vector3 direction = other.transform.position - transform.position;
        float angle = Vector3.Angle(direction, transform.forward);

         // reset whether we've seen the player
        seenTarget = false;

        RaycastHit hit;

        // is it less than our field of view
        if (angle < sightFov * 0.5f)
        {
            // if the raycast hits the player we know
            // there is nothing in the way
            // adding transform.up raises up from the floor by 1 unit
            if (Physics.Raycast(transform.position + transform.up,
                direction.normalized,
                out hit,
                collider.radius))
            {
                if (hit.collider.gameObject == target)
                {
                    // flag that we've seen the player
                    // remember their position
                    seenTarget = true;
                    lastSeenPosition = target.transform.position;
                }
```

Debugging and understanding how and whether this is working can be difficult if the enemy is moving. Here a *Gizmo* overlay, drawn by the *OnDrawGizmos* method of a script, can help to visualise what is going on by drawing primitive lines and wireframe spheres to represent the sight lines and limits of the enemy sight perception. Gizmos can be enabled and disabled for objects in the *gizmo* menu both in the scene and the game view:



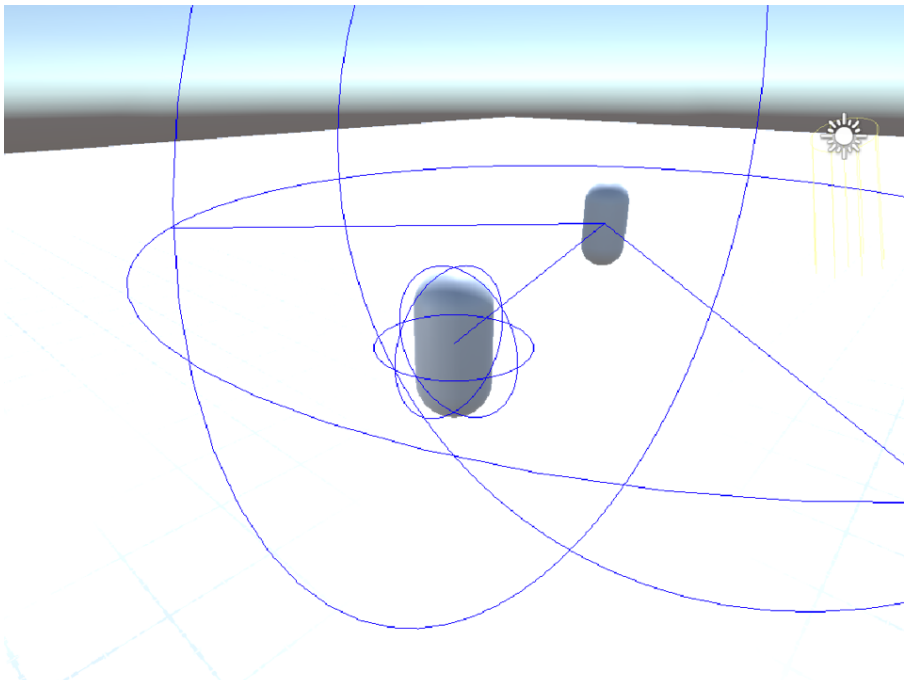
You will need to complete / adapt this to suit your own code.

```
void OnDrawGizmos()
{
    Gizmos.color = Color.blue;
```

```
if (collider!=null)
{
    Gizmos.DrawWireSphere(transform.position, collider.radius);

    if (seenTarget)
        Gizmos.DrawLine(transform.position, lastSeenPosition);

    if (lastSeenPosition != Vector3.zero)
        // draw a small sphere

    // calculate left fov vector
    Vector3 rightPeripheral;
    rightPeripheral = (Quaternion.AngleAxis(sightFov * 0.5f, Vector3.up)
        * transform.forward * collider.radius)

    // draw lines for the left and right edges of the field of view
```

The Quaternion *AngleAxis* function returns a Quaternion that represents a rotation of half of the field of view angle around the "up" (Y) axis. Multiplying this Quaternion object by transform.forward produces a vector corresponding to the edge of the enemy's vision, and multiplying it by the radius of the collider puts it at the correct distance. You can then draw a line from the enemy to this point to show the edge of their field of view.

https://docs.unity3d.com/ScriptReference/Quaternion.html



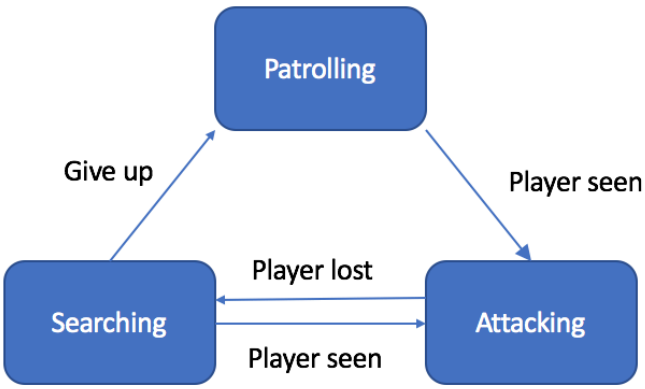
Putting it together: Finite State Machine

These various capabilities of the enemy character (seeing, moving, patrolling) can now be put together to enable more complex behaviours. Rather than continuing to extend the updates and triggers of the enemy controller, we will introduce a common software design pattern used for implementing simple game AI, the *finite state machine*:

A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one

state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.

In this case, the desired behaviour of the enemy as set out at the beginning of the document is broken down into a set of manageable chunks, or “states”. The enemy can be in only one state at a time, but it can transition from one state to another when the right conditions arise (i.e. the enemy sees the player, or loses sight of them):



The implementation of this is derived from a C++ implementation given in Programming Game AI by Example, by Mat Buckland. It involves implementing a simple state machine class, and several state classes that implement a standard state interface.

- Create a new script, called *StateMachine*. This specifies the interface *IState*, and a class *StateMachine* that keeps track of which state the enemy is in.

```
public interface IState
{
    void Enter();
    void Execute();
    void Exit();
}

public class StateMachine
{
    IState currentState;

    public void ChangeState(IState newState)
    {
        if (currentState != null)
            currentState.Exit();

        currentState = newState;
        currentState.Enter();
    }

    public void Update()
    {
        if (currentState != null) currentState.Execute();
    }
}
```

- We can now shift the logic for the enemy to patrol into its own state, that implements the IState interface. The state needs to keep a reference to the EnemyController via its constructor so it can access the necessary NavMeshAgent

and Waypoints. You might need to make some of the variables public in the owner class.

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

public class State_Patrol: IState
{
    EnemyController owner;
    NavMeshAgent agent;
    Waypoint waypoint;

    public State_Patrol(EnemyController owner) { this.owner = owner; }

    public void Enter()
    {
        Debug.Log("entering patrol state");

        waypoint = owner.waypoint;
        agent = owner.GetComponent<NavMeshAgent>();
        agent.destination = waypoint.transform.position;
        // start moving, in case we were previously stopped
        agent.isStopped = false;
    }

    public void Execute()
    {
        Debug.Log("updating patrol state");

        // same as before
        if (!agent.pathPending && agent.remainingDistance < 0.5f)
        {
            Waypoint nextWaypoint = waypoint.nextWaypoint;
            waypoint = nextWaypoint;
            agent.destination = waypoint.transform.position;
        }
    }

    public void Exit()
    {
        Debug.Log("exiting patrol state");
        // stop moving
        agent.isStopped = true;
    }
}
```

- Finally, add the state machine itself to the enemy controller. The initial ChangeState call calls the Enter() method of the patrolling state, and subsequent calls to the Update() function of the enemy drive the execution of the current state via the state machine.

```
    public StateMachine stateMachine = new StateMachine();

    // Use this for initialization
    void Start ()
    {
        ...

        stateMachine.ChangeState(new State_Patrol(this));
    }
```

```
    // Update is called once per frame
    void Update () {

        stateMachine.Update();
    }
```

- If, during the patrolling state, the enemy sees the player, then the state machine should transition into another state, one in which it attacks the player.

```
    public void Execute()
    {
        Debug.Log("updating patrol state");

        ...

        if(owner.seenTarget)
        {
            owner.stateMachine.ChangeState(new State_Attack(owner));
        }
    }
```

- Where State_Attack is another state for the state machine:

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

public class State_Attack : IState
{
    EnemyController owner;
    NavMeshAgent agent;

    public State_Attack(EnemyController owner) { this.owner = owner; }

    public void Enter()
    {
        Debug.Log("entering attack state");
        agent = owner.GetComponent<NavMeshAgent>();

        if (owner.seenTarget)
        {
            agent.destination = owner.lastSeenPosition;
            agent.isStopped = false;
        }
    }

    public void Execute()
    {
        Debug.Log("updating attack state");

        agent.destination = owner.lastSeenPosition;
        agent.isStopped = false;

        if (!agent.pathPending && agent.remainingDistance < 5.0f)
        {
            agent.isStopped = true;
        }

        if (owner.seenTarget != true)
        {
            Debug.Log("lost sight");
            // search for the player
```

```
            ...
        }

        // fire on the player
        ...
    }

    public void Exit()
    {
        Debug.Log("exiting attack state");
        agent.isStopped = true;
    }
}
```

Exercises

Firing

Attacking the player currently just involves moving close to the player. Extend the execution of this state by allowing the enemy to shoot at the player. Firing should be implemented as a function in the Enemy Controller but then called from the execution function, and could spawn a bullet object at a time limited rate as in the first-person shooter in lab 03.

Searching

Complete the implementation of the state diagram above by introducing a *Searching* state that the enemy transitions into from *Attacking* if it loses sight of the player. This state should involve the enemy moving to the last seen position of the player, and if it still can't see the player, transitioning into the *Patrolling* state. If it does regain sight of the player, it should transition back to the *Attacking* state.

Experiment with Stealth

A goal that requires the player to reach a certain location without being "seen" results in some surprisingly complex interactions. Construct a level that requires the player to avoid the AI, but if they are seen have the AI move very quickly towards them.