Lab 1: Patrolling Guard with FSM

At this point in your education, you know how to do some C# programming in Unity. That's more than enough! Some of you may feel more confident than others, and that's to be expected.

Labs are not graded, but mandatory.

Hand in your solution by submitting a repo link or relevant code on Omniway.

Learning Goals

After this lab, you should be able to:

- Explain what a **Finite State Machine (FSM)** is in the context of game AI.
- Design a simple FSM for an enemy character (e.g. PATROL -> CHASE -> RETURN).
- Implement that FSM in Unity using **C#** and an **enum**.
- Use **Unity's NavMesh** system to move an agent around a level.

First, do some research

It's usually a good idea to perform an exploratory phase before you get into coding. Don't stay there for too long. Take focused notes, look up different sources, pose yourself questions, try to answer them, and you'll be more ready for the (experimental) coding phase.

1. Finite State Machines in games
    - Look up an article or tutorial on "Finite State Machines (FSM) for game AI"
    - Focus on: *states, transitions, conditions*
    - Suggestions:
        i. https://en.wikipedia.org/wiki/Finite-state_machine
        ii. https://gameprogrammingpatterns.com/state.html
        iii. https://www.youtube.com/watch?v=-ZP2Xm-mY4E
2. Unity NavMesh basics
    - Look up "Unity Manual - NavMeshAgent" and "NavMesh baking"
    - Focus on: *NavMeshSurface/NavMesh, NavMeshAgent, SetDestination*

Answer some questions

To understand something, you need to know some answers. To have some answers, you need to pose yourself some questions. These won't be handed in, but we will discuss them.

1. In your own words, what is a **state** in game AI? Give two examples.
2. What triggers a **transition** between states? Give two examples.
3. Why do you think game AI often uses simple techniques (FSMs) instead of "real AI" like deep learning?

Scenario: The patrolling guard

You will create a tiny prototype:

- The **Player** is a capsule you can move around on a flat level.
- A **Guard** (another capsule) walks between predefined waypoints.
- When the Player gets close enough, the Guard **starts chasing**.
- If the Player escapes, the Guard **returns to patrol.**

Everything is made from primitive objects (planes, cubes, capsules, empties). No fancy models or animations, unless you want to!

Lab tasks on the next page ->

Lab Tasks

Step 0 - Create your own project

1. Create new **3D URP (or 3D Core)** project.
2. Name it something like: *AI_Lab1_PatrollingGuard_YourName*.
3. Save the default scene as MainScene.

Optional: Create a local Git repo for this project. Ask for help if you're new to Git.

Step 1 - Build a tiny test level

1. Add a **Plane** as the ground (if not already existing in your scene). Scale it up (e.g. 10x10)
2. Add a few **Cubes** as walls or obstacles
3. Add a **Player:**
    a. Create a Capsule, name it *Player*.
    b. Add a simple script ***PlayerController*** that moves with WASD/arrow keys.

Hint: If you don't remember how to do this, search your earlier course material/work or Unity documentation.

Checkpoint questions:

- What coordinate axes respond to "forward" and "up" in Unity?
- What is the difference between ***`Update()`*** and ***`FixedUpdate()`*** in a movement script?

Step 2 - Set up NavMesh and Guard

1. Add a **Guard:**
    a. Create another Capsule, rename to *Guard.*
    b. Give it a different color (Material) so you can tell it apart.
2. Enable **Navigation:**
    a. Add a **NavMeshSurface** (if using the component-based system) to your ground object, or use the AI Navigation window.
    b. Mark the ground as **walkable**.
    c. Click **Bake** to generate a NavMesh.
3. Add a **NavMeshAgent** component to the Guard.

Checkpoint questions:

- What happens if the Guard has a NavMeshAgent but there is no baked NavMesh?
- In your own words, what is the difference between a **path** and the **movement** along that path?

Step 3 - Guard patrol behaviour

Goal: the Guard constantly walks between a set of waypoints.

1. Create 3-4 empty GameObjects as **waypoints** (e.g. *Waypoint1, Waypoint2, …)*. Place them around the level.
2. Create a script **GuardPatrol** and attach it to the Guard.

Suggested structure:

```
public class GuardPatrol : MonoBehaviour
{
    public Transform[] waypoints;
    public float waypointTolerance = 0.5f;

    int _currentIndex = 0;
    NavMeshAgent _agent;

    void Awake()
    {
        _agent = GetComponent<NavMeshAgent>();
    }

    void Start()
    {
        if (waypoints.Length > 0)
        {
            _agent.SetDestination(waypoints[_currentIndex].position);
        }
    }

    void Update()
    {
        if (waypoints.Length == 0) return;

        // Check if we've reached the current waypoint
        if (!_agent.pathPending && _agent.remainingDistance <= waypointTolerance)
        {
            // Go to next waypoint (loop)
            _currentIndex = (_currentIndex + 1) % waypoints.Length;
            _agent.SetDestination(waypoints[_currentIndex].position);
        }
    }
}
```

3. Assign the waypoint objects to the waypoints array in the Inspector.

Theory questions (short answers):

- Why do we check ***!_agent.pathfinding*** before reading ***remainingDistance***?
- What happens if we forget to use ***% waypoints.Length*** when incrementing the index?

Step 4 - Introduce a Finite State Machine

Now you'll extend the guard's behavior with **states**.

We'll use three states:

- **Patrolling** - walking between waypoints.
- **Chasing** - chasing the player.
- **ReturningToPatrol** - going back to the nearest waypoint after losing the player.

1. Add an **enum** inside your GuardPatrol script:

```
public enum GuardState
{
    Patrolling,
    Chasing,
    ReturningToPatrol
}
```

2. Add a field to store the current state:

```
public GuardState currentState = GuardState.Patrolling;
public Transform player;
public float chaseRange = 5f;
public float loseRange = 7f;
```

3. In *Update()*, refactor your code into a **switch on** *currentState*:

```
void Update()
{
    switch (currentState)
    {
        case GuardState.Patrolling:
            UpdatePatrol();
            break;
        case GuardState.Chasing:
            UpdateChase();
            break;
        case GuardState.ReturningToPatrol:
            UpdateReturning();
            break;
    }
}
```

4. Implement the the helper functions:
    a. `UpdatePatrol()` - same logic as before (follow waypoints), plus:
        i. If distance to *player* < *chaseRange*, switch to *Chasing*.
    b. `UpdateChase()` - set destination to player.position, every frame or at intervals.
        i. If distance to *player* > *loseRange*, remember a suitable waypoint and switch to *ReturninToPatrol*.
    c. `UpdateReturning()` - set destination to that waypoint.
        i. When you reach it, switch back to *Patrolling*.

Tip: Start simple. You can first do only two states: *Patrolling* and *Chasing*. Then add ReturningToPatrol when the basics work. Rule of thumb: Make it work, then make it right.

Theory questions:

- Draw a small state diagram for your Guard, with arrows labelled by conditions.
- What would the code look like if you did not have an enum and tried to manage states with only booleans like `isPatrolling`, `isChasing`? Why might that be harded to maintain?

Step 5 - Stretch goals (optional)

If you want to push your system further:

- Add a **field of view:** Guard only sees the player if they are in front (use angle/dot product)
- Use `Physics.Raycast` to check **line of sight**, so that walls/obstacles block vision.
- Add a simple **Search** state: when the player escapes, Guard goes to last known position and looks around for a few seconds.

Show of your scene, give it personal flavor!

By the the end of this lab, you should be able to

- Run the scene and demonstrate:
    - Player moves around
    - Guard patrols between waypoints
    - Guard switches to chase behaviour when the player is close
    - Guard returns to patrolling when the player escapes

- Briefly exlain your FSM states and transitions in.
- Point to the part of your code where the state change happens.

You are encouraged to make the system/scene more interesting, or use the system in another project.

Labs are mandatory but ungraded. They prepare you for the graded project.