Lab 2: Grid-Based A* Pathfinding

At this point, you've:

- Implemented a **Finite State Machine** for a patrolling guard and used **Unity NavMesh** to move it around.
- Seen in Lecture 2 how game levels can be modelled as **graphs**, and how **A*** searches those graphs to find paths.

Now you'll build your **own** pathfinding system on a grid, instead of relying on Unity's built-in NavMesh.

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

Hand in your solution by submitting a repo link or relevant code on Omniway (same as Lab 1).

Learning Goals

After this lab, you should be able to:

- Represent a **grid** as a graph in code (nodes, edges, costs).
- Implement the **A*** pathfinding algorithm on that grid.
- Make an agent **walk along** the resulting path in a Unity scene.
- Describe the role of **g(n)**, **h(n)** and **f(n)** in A*.
- Reason about **heuristics** (e.g. Manhattan distance) and the effect of different **movement costs**.

Before you start

You should:

- Have looked at the Lecture 2 prep material and warm-up questions (graphs, Manhattan distance, A* intuition).
- Be comfortable with:
    - C# basics (arrays/lists, loops, simple classes).
    - Creating scenes, placing cubes/planes in Unity.
- Have Lab 1 fresh in your mind (even better if you still have your project).

If you want to use **Unreal** or **Godot**:

- The **ideas** are the same (grid, nodes, A*, path as a list of waypoints).
- You'll need to adapt the **API calls** (input, raycasts, movement). Short hints are included in some steps.

Scenario: A grid and a pathfinding agent

You'll build:

- A **GridManager** that:
    - Creates a 2D grid of tiles.
    - Keeps a `Node[,]` array representing the graph.
    - Lets you toggle tiles between **walkable** and **wall**.
- A **Pathfinder** that:
    - Runs A* on this grid.
    - Visualizes the final **path** between a start and goal.
- An **AgentMover** that:
    - Requests a path.
    - Moves an agent along it.

Optionally, you can:

- Let the **player** move around, and pathfind **towards** them.
- Combine this with your **Lab 1 guard** later (stretch goal).

Step 0 – Project setup

You have two options:

Option A – New project

- Create a new 3D (URP or Core) project:
    - Name it: `AI_Lab2_AStar_YourName`.
- Create a scene `AStarLabScene`.
- Optional: set up version control (Git or Unity Plastic).

Option B – Reuse Lab 1 project

- Open your **Lab 1** Unity project.
- Create a **new scene** for this lab:
    - `Scenes/AStarLabScene` (for example).
- Keep your existing **Player** controller if you want, or use a SimplePlayerController from the course repo if provided.

In both cases, make sure you have a **Camera** that looks at the grid area from above or at a slight angle.

Step 1 – Build a simple grid level

Goal: Have a visual **grid of tiles** and a corresponding **Node grid** in code.

1.1 Create a ground and camera

1. Add a **Plane** as ground; scale it so it's large enough (e.g. 10×10 or 20×20).
2. Position the **Camera** so it can see the entire grid area.

1.2 Create a Tile prefab

1. Create a **Cube**, rename to `Tile`.
2. Scale to something like `(1, 0.2, 1)` so it looks like a flat tile.
3. Create materials in `Materials/`:
   - `Mat_WalkableTile`
   - `Mat_Wall`
   - `Mat_PathTile`
   - `Mat_StartTile`
   - `Mat_GoalTile`
4. Create a **Material** for walkable tiles (e.g. `Mat_WalkableTile`) and assign it.
5. Drag `Tile` from Hierarchy to `Prefabs/Tile.prefab`.
6. Delete the Tile instance from the scene (we'll instantiate from script).

1.3 GridManager & Node

A minimal `Node` class in Scripts/Grid/:

```
public class Node
{
    public int x;
    public int y;
    public bool walkable;
    public GameObject tile;

    public float gCost;
    public float hCost;
    public Node parent;

    public float fCost => gCost + hCost;

    public Node(int x, int y, bool walkable, GameObject tile)
    {
        this.x = x;
        this.y = y;
        this.walkable = walkable;
        this.tile = tile;

        gCost = float.PositiveInfinity;
        hCost = 0f;
        parent = null;
    }
}
```

Your `GridManager` should:

- Instantiate `width × height` tiles.
- Create a `Node` for each tile: `nodes[x, y]`.
- Have methods like:
    - `GetNode(int x, int y)`
    - `GetNeighbours(Node node, bool allowDiagonals = false)`
    - `SetWalkable(Node node, bool walkable)`

Create Scripts/Grid/GridManager.cs:

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

namespace GameAI.Day02_AStar.Grid
{
    public class GridManager : MonoBehaviour
    {
        [Header("Grid Settings")]
        [SerializeField] private int width = 10;
        [SerializeField] private int height = 10;
        [SerializeField] private float cellSize = 1f;

        [Header("Prefabs & Materials")]
        [SerializeField] private GameObject tilePrefab;
        [SerializeField] private Material walkableMaterial;
        [SerializeField] private Material wallMaterial;
        private Node[,] nodes;
        private Dictionary<GameObject, Node> tileToNode = new();

        // Input action for click
        private InputAction clickAction;

        public int Width => width;
        public int Height => height;
        public float CellSize => cellSize;
```

```
        private void Awake()
        {
            GenerateGrid();
        }

        private void OnEnable()
        {
            clickAction = new InputAction(
                name: "Click",
                type: InputActionType.Button,
                binding: "<Mouse>/leftButton"
            );
            clickAction.performed += OnClickPerformed;
            clickAction.Enable();
        }

        private void OnDisable()
        {
            if (clickAction != null)
            {
                clickAction.performed -= OnClickPerformed;
                clickAction.Disable();
            }
        }

        private void GenerateGrid()
        {
            nodes = new Node[width, height];

            for (int x = 0; x < width; x++)
            {
                for (int y = 0; y < height; y++)
                {
                    Vector3 worldPos = new Vector3(x * cellSize, 0f,
y * cellSize);
                    GameObject tileGO = Instantiate(tilePrefab,
worldPos, Quaternion.identity, transform);
                    tileGO.name = $"Tile_{x}_{y}";

                    Node node = new Node(x, y, true, tileGO);
                    nodes[x, y] = node;
```

```
                tileToNode[tileGO] = node;

                SetTileMaterial(node, walkableMaterial);
            }
        }
    }

    private void OnClickPerformed(InputAction.CallbackContext
ctx)
    {
        HandleMouseClick();
    }

    private void HandleMouseClick()
    {
        Camera cam = Camera.main;
        if (cam == null) return;

        Ray ray =
cam.ScreenPointToRay(Mouse.current.position.ReadValue());
        if (Physics.Raycast(ray, out RaycastHit hit))
        {
            GameObject clicked = hit.collider.gameObject;
            if (tileToNode.TryGetValue(clicked, out Node node))
            {
                bool newWalkable = !node.walkable;
                SetWalkable(node, newWalkable);
            }
        }
    }

    public Node GetNode(int x, int y)
    {
        if (x < 0 || x >= width || y < 0 || y >= height)
            return null;
        return nodes[x, y];
    }
```

```
        public Node GetNodeFromWorldPosition(Vector3 worldPos)
        {
            int x = Mathf.RoundToInt(worldPos.x / cellSize);
            int y = Mathf.RoundToInt(worldPos.z / cellSize);
            return GetNode(x, y);
        }

        public IEnumerable<Node> GetNeighbours(Node node, bool
allowDiagonals = false)
        {
            int x = node.x;
            int y = node.y;
            // 4-neighbour
            yield return GetNode(x + 1, y);
            yield return GetNode(x - 1, y);
            yield return GetNode(x, y + 1);
            yield return GetNode(x, y - 1);

            if (allowDiagonals)
            {
                yield return GetNode(x + 1, y + 1);
                yield return GetNode(x - 1, y + 1);
                yield return GetNode(x + 1, y - 1);
                yield return GetNode(x - 1, y - 1);
            }
        }

        public void SetWalkable(Node node, bool walkable)
        {
            node.walkable = walkable;
            SetTileMaterial(node, walkable ? walkableMaterial :
wallMaterial);
        }

        private void SetTileMaterial(Node node, Material mat)
        {
            var renderer = node.tile.GetComponent<MeshRenderer>();
            if (renderer != null && mat != null)
            {
                renderer.material = mat;
```

```
            }
        }
    }
}
```

Hook it up in Unity

- Select GridManager GameObject.
- Drag Tile.prefab into tilePrefab.
- Drag materials:
    - Mat_WalkableTile → walkableMaterial
    - Mat_Wall → wallMaterial
- Set width and height to e.g. 10, 10.
- Press Play: you should see a 10×10 grid of tiles.
- Click tiles: they toggle between walkable and wall.

Checkpoint questions:

1. In your own words, what is a **node** and what is an **edge** in this grid?
2. How does your grid coordinate `(x, y)` map to **world position** `(x * cellSize, z * cellSize)`?
3. What happens if you try to access `nodes[x, y]` with coordinates outside the array bounds? How are you preventing that?

Unreal hint:
Use a `UObject` or `struct` for `FNode` and keep a `TArray<TArray<FNode>>` (or `TArray<FNode>` with manual width/height indexing). Use `SpawnActor` for tiles, or a `UInstancedStaticMeshComponent` for efficiency.

Step 2 – Make tiles toggle walkability (walls)

Goal: Click tiles to toggle **walkable vs wall** and update their materials.

The GridManager code above already shows one way to do this; in this step, make sure you understand what happens and, if you like, re-implement or modify it yourself

2.1 Materials

Create at least:

- `Mat_WalkableTile` – default.
- `Mat_Wall` – for non-walkable.

Your `GridManager` should have:

```
public Material walkableMaterial;
public Material wallMaterial;
```

2.2 Mouse click → toggle

Implement a method that:

- Raycasts from the camera through the mouse position.
- Finds the clicked tile GameObject.
- Looks up its `Node`.
- Toggles `node.walkable`.
- Updates the material accordingly.

If you're using the **new Input System**, you'll use `Mouse.current.position` and an `InputAction` bound to `<Mouse>/leftButton`. If you're using the old `Input` API, `Input.GetMouseButtonDown(0)` is fine.

Checkpoint questions:

1. What Unity function are you using to convert a screen position to a 3D ray?
2. Why is it useful to **visualize** walkable vs non-walkable tiles clearly?

3. In your own words, what does it mean in graph terms when you turn a tile into a wall?

Unreal hint:
Use a `LineTraceByChannel` from `PlayerController`'s `GetHitResultUnderCursor`. When you hit a tile actor, toggle a `bWalkable` flag and change its material.

Step 3 – Implement A* on the grid

Goal: Write a **FindPath** function that returns a list of Nodes from start to goal, using **A***.

3.1 Decide how to pick start & goal

For the lab, you can:

- Hard-code start at `(0,0)` and goal at `(width-1, height-1)`, **or**
- Use special "Start" and "Goal" markers in the scene and convert their positions to nodes, **or**
- Let the player click to set start and goal (optional extra).

Pick one approach and stick with it for now.

3.2 A* core

Create a `Pathfinder` script that references your `GridManager`. Below is **pseudocode**, not a full C# implementation. You will implement the C# version yourself based on this and the lecture demo.:

```
public List<Node> FindPath(Node startNode, Node goalNode)
{
    // 1. Reset node costs
    // 2. Initialize openSet and closedSet
    // 3. Set gCost and hCost for startNode
    // 4. Loop until openSet is empty:
    //    - pick node with lowest fCost
    //    - if this is goalNode, reconstruct and return path
    //    - otherwise, move it to closedSet
    //    - for each neighbour:
    //          - skip if null, not walkable, or in closedSet
    //          - compute tentativeG = current.gCost + stepCost
    //          - if tentativeG < neighbour.gCost:
    //                - update neighbour.parent, gCost, hCost
    //                - ensure neighbour is in openSet
}
```

Use **Manhattan distance** as the heuristic (excuse the ugly formatting):

```
h(n) = |x_n - x_goal| + |y_n - y_goal|
```

Implement a helper:

```
float HeuristicCost(Node a, Node b)
{
    int dx = Mathf.Abs(a.x - b.x);
    int dy = Mathf.Abs(a.y - b.y);
    return dx + dy;
}
```

For the **step cost** between neighbours on a uniform grid, use `1f`.

Solution suggestion for the pathfinding and agent code will be provided within a day after this lab is handed out.

3.3 Visualize the path

Once `FindPath` works:

- Get a path `List<Node>` from start to goal.
- Set the material of each node on the path to `Mat_PathTile`.
- Re-apply `Mat_WalkableTile` / `Mat_Wall` to non-path tiles before each run.

Checkpoint questions:

1. In your code, where do you actually compute **g(n)**, **h(n)** and **f(n)**?
2. Why do we use `tentativeG < neighbour.gCost` to decide whether we found a better path to a neighbour?
3. What would happen if your **heuristic** severely **overestimates** the real remaining distance?

Theory questions:

1. If you set `h(n) = 0` for all nodes, what classic algorithm does A* become?
2. If you ignored `g(n)` completely and only sorted by `h(n)`, what behaviour would you expect to see?

Godot hint:
Use a 2D array or `Dictionary<Vector2I, Node>` for nodes. A* can be implemented manually in GDScript or C#. Godot also has an `AStar2D` helper class, but try the manual version first to learn the algorithm.

Step 4 – Let an agent walk along the path

Goal: Use your path to move an agent **smoothly** from start to goal.

4.1 Create an Agent

1. Add a **Capsule** or **Cube** called `Agent`.
2. Add a simple script `AgentMover` (or similar).

4.2 Path-following logic

Your `AgentMover` might:

- Ask `Pathfinder` for a path from its **current tile** to some **goal tile**.
- Store the path as a `List<Node>` or `List<Vector3>` world positions.
- Use a coroutine or `Update()` to:
    - Move towards the **next waypoint** in the path.
    - When close enough, move on to the **next one**, until the goal is reached.

Pseudo-structure:

```
public class AgentMover : MonoBehaviour
{
    public Pathfinder pathfinder;
    public float moveSpeed = 3f;

    private List<Node> currentPath;
    private int currentIndex = 0;

    public void FollowPath(List<Node> path)
    {
        currentPath = path;
        currentIndex = 0;
    }

    private void Update()
    {
        if (currentPath == null || currentPath.Count == 0) return;

        Node targetNode = currentPath[currentIndex];
        Vector3 targetPos = NodeToWorldPosition(targetNode);

        // Move towards targetPos...
        // When close enough:
        //    currentIndex++;
        //    if currentIndex >= currentPath.Count, we're done.
    }
```

```
    private Vector3 NodeToWorldPosition(Node node)
    {
        return new Vector3(node.x * cellSize, transform.position.y,
node.y * cellSize);
    }
}
```

Triggering movement:

- For now, you can call `FollowPath` once at `Start()` or when you press a key (e.g. R).
- Later, you can connect this to **player position** (chase the player) or **clicks** (go to clicked tile).

Checkpoint questions:

1. In your own words, what is the difference between:
    - **Computing a path**, and
    - **Moving along that path**?

2. What happens in your movement code if there is **no path** (FindPath returns null)?
    - Do you handle that case gracefully?

Unreal hint:
Instead of manually updating `transform.position`, you might use `AIController::MoveToLocation` with your own list of waypoints. For learning, it's still valuable to implement a simple "move towards next point" loop in Blueprints or C++.

Step 5 – Connect to your player (optional but recommended)

Goal: Make your agent pathfind **towards a dynamic target**, ideally the player.

Ideas:

- Use your **Lab 1 Player** controller in this scene:
    - Convert the player's world position to a **goal node**.
    - Periodically recompute a path from agent to player (e.g. every 0.5–1 second).
    - Let the agent follow the updated path.

- Alternatively, let the player **click on a tile**, and the agent pathfinds to the clicked tile.

Checkpoint questions:

1. If your goal (player) can move, how often do you think you should recompute the path? What trade-offs appear?

2. What happens if the player runs into a location that is currently unreachable because of walls? How could you detect and handle that?

Step 6 – Reflection and theory

When your basic system works, take a moment to think and scribble a few short answers (not for grading, but we may discuss them):

1. How is your **grid + A*** system conceptually similar to Unity's **NavMesh + NavMeshAgent** from Lab 1? How is it different?

2. In which situations would you prefer:

    - A **NavMesh** with built-in pathfinding?

    - A custom **grid-based A*** like this?

3. Imagine you had a **large RTS map** with hundreds of units.

    - What performance issues could appear with naive A* usage?

    - What strategies might you use to reduce pathfinding cost?

Step 7 – Stretch goals (optional)

Pick one or more if you have time and curiosity.

7.1 Diagonal movement

- Allow **8-neighbour** movement (diagonals).

- Give diagonals a slightly higher cost (e.g. `1.4f` vs `1f`).

- Adjust your heuristic so that it matches this movement type (e.g. octile distance).

Questions:

- How does this change the shape of paths?

- How do you ensure your heuristic is still **admissible** (never overestimates)?

7.2 Terrain costs

- Introduce different terrain types:
    - Road (cost 1),
    - Mud (cost 3),
    - Water (cost 5), etc.
- Store a `movementCost` on `Node` or derive it from tile material.
- Use `tentativeG = current.gCost + movementCost` instead of `+1`.

Questions:

- Does the path now **avoid** expensive tiles when possible?
- How would you encode “danger” (e.g. enemy vision tiles) as a cost?

7.3 Multiple agents

- Spawn several agents on the grid.
- Each agent computes its own path to the same goal.
- Observe:
    - Do they overlap and “clip” through each other?
    - How might you add **simple avoidance**?

7.4 Engine port (Unreal / Godot)

If you’re feeling adventurous:

- Port the **core logic** (Node, grid, A*, path following) to:
    - Unreal (BluePrints or C++), or
    - Godot (GDScript or C#).
- You don’t need identical visuals – focus on:

    - Grid representation,
    - A* search,
    - Moving a pawn/character along the path.

Write a short note (few bullets) on what felt different between engines.

What to hand in

On Omniway, submit:

1. Link to your **Git repo** (or a zipped project if you really can't use Git).
2. A short **readme** or **text file** with:
    - What works (grid, A*, agent movement, any extras).
    - Any **known issues** or **limitations**.
    - 3–5 bullet points reflecting on what you learned or found tricky.

By the end of Lab 2, you should be able to:

- Run your scene and demonstrate:
    - A grid of tiles with some walls.
    - A computed path from start to goal (visualized).
    - An agent walking along that path.
- Explain:
    - What **g(n)**, **h(n)** and **f(n)** mean in your own words.
    - Where in your code the **search** happens vs where the **movement** happens.