Lab 4 (Unity): Behaviour Trees with Unity Behavior Graph + Blackboard

Focus: Build a guard NPC using a **Behaviour Tree/Graph** and a **Blackboard** in **Unity 6.x** using the modern **Unity Behavior** package—while still writing meaningful C# where it matters (sensing + "glue" logic).
Note: We're using Unity 6.3 / Unity Behavior in this course. If you're on a slightly different Unity 6 version, the core ideas and most steps are the same.

---

1. Learning goals

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

- Build a Behaviour Tree/Graph that chooses between **Patrol** → **Chase** → **Search** → **Patrol**.
- Use a **Blackboard** to store shared AI state (e.g., `Target`, `HasLineOfSight`, `LastKnownPosition`).
- Explain (and demonstrate) **Running** with real "multi-frame" actions like moving and waiting.
- Write and integrate **custom C# nodes** in Unity Behavior (Action nodes), while relying on engine tooling for the rest.

---

2. Prerequisites

You do **not** need to have completed Lab 1/2/3.

Packages you'll likely use:

- **Behavior** (Unity Behavior Graph)
- **AI Navigation** (NavMeshSurface, baking, NavMeshAgent)

---

3. Scenario: “The guard” revisited (now with a Behaviour Tree)

You will create a small prototype:

- A **Player** (capsule) moves with WASD.
- A **Guard**:
    1. **Patrols** between waypoints
    2. If it **sees** the player → **Chases**
    3. If it **loses** the player → goes to **LastKnownPosition** and **Searches**
    4. Then returns to **Patrolling**

This is the same “story” as Lab 1’s FSM guard, except you’ll express it using a **tree/graph + blackboard** instead of a switch on an enum.

---

4. Step 0 — Project setup (new or reuse)

Choose one:

Option A: New project

1. Create a new **3D Core** (or URP) Unity project.
2. Name it something like: `AI_Lab4_BehaviorTrees_YourName`
3. Create a scene: `Lab4_BehaviorTreeScene`

Option B: Reuse an earlier project

- Add a new scene for Lab 4 and keep it separate.

---

5. Step 1 — Install packages (Behavior + Navigation)

1. Open **Window** → **Package Manager**
2. Install:
    - **Behavior** (Unity Behavior Graph)
      Use a pre-defined node | Behavior | 1.0.13
      Behavior graph node types | Behavior | 1.0.13
    - **AI Navigation** (NavMeshSurface, etc.)

---

6. Step 2 — Build a tiny test level + NavMesh

1. Create a **Plane** (Ground).
2. Add 3–6 **Cube** obstacles (walls).
3. Ensure obstacles have colliders (default cubes do).

NavMesh

1. Add a **NavMeshSurface** component (either on Ground or a parent object).
2. Click **Bake**.
3. You should see a navigable area (depending on your visualization settings).

---

7. Step 3 — Create Player + Guard (basic movement)

Player

1. Create a **Capsule** named `Player`
2. Add the script **`SimplePlayerController.cs`** (Appendix A.1), or use a previously written controller script of yours.
3. (Optional) Tag it as `Player`

Guard

1. Create a **Capsule** named `Guard`
2. Add a **NavMeshAgent** component
3. Give it a distinct material color

---

8. Step 4 — Create patrol waypoints

1. Create 3–5 empty GameObjects: `WP_1`, `WP_2`, …
2. Place them around the scene.

---

9. Step 5 — Create the Behavior Graph asset + Blackboard variables

1. In Project window: **Create** → **Behavior** → **Behavior Graph** (name it `BG_Guard`)
2. Select your `Guard` object and add **BehaviorGraphAgent** component
3. Assign `BG_Guard` to the agent

Now open `BG_Guard` and create these **Blackboard** variables:

Required:

- `Target` : **GameObject**
- `HasLineOfSight` : **bool**
- `LastKnownPosition` : **Vector3**

Recommended:

- `SearchDuration` : **float** (e.g., 2.0–5.0 seconds)
- `ForgetTargetAfter` : **float** (e.g., 6.0–10.0 seconds)

- `TimeSinceLastSeen` : **float** (starts at a large value)

In Unity Behavior, the agent manages the graph lifecycle and stores data through blackboard variables. Class BehaviorGraphAgent

---

10. Step 6 — Add sensors to the Guard (C# "perception" component)

Add **`GuardSensors.cs`** (Appendix A.2) to the `Guard`.

In the inspector, configure:

- view distance
- view angle
- obstacle/occlusion layers
- (optional) a `Transform` for "eyes" (or use the guard transform)

This keeps *perception math* in normal Unity C#, while the BT decides *what to do*.

---

11. Step 7 — Create 2 custom Behavior actions (C# nodes)

You will create **two custom Action nodes** in the Behavior Graph editor (so Unity generates the correct boilerplate), then paste in the code from the appendix.

11.1 Create "Update Perception" node

1. In the graph editor: right-click → **Create new** → **Action**
   Workflow note: When you create Create new Action in the graph editor, Unity creates a C# script file in your project. Find this file in your Project window, open it, and replace its contents with your code/appendix code.
2. Name: `Update Perception`
3. Category: `Action/Sensing`
4. Description/story can be simple (you can refine it later)
5. After Unity creates the script, open it and replace its contents with **Appendix A.3**.

This action should:

- update `HasLineOfSight`
- update `Target` (if sensed)
- update `LastKnownPosition`
- update `TimeSinceLastSeen`

11.2 Create “Clear Target” node

Repeat the process and replace with **Appendix A.4**.

---

12. Step 8 — Build the Behaviour Tree/Graph logic

You’re going to assemble something like this:

```
On Start (Repeat)
└── Run In Parallel
    ├── [Left] Repeat → Update Perception
    └── [Right] Try In Order (Selector)
        ├── Sequence: Chase
        │   ├── Condition: HasLineOfSight == true
        │   └── Navigate To Target
        ├── Sequence: Search
        │   ├── Condition: Target != null AND HasLineOfSight == false
        │   ├── Navigate To Location (LastKnownPosition)
        │   ├── Wait (SearchDuration)
        │   └── Clear Target
        └── Sequence: Patrol
            └── Patrol (waypoints)
```

Note: The order of sequences in “Try In Order” matters. Chase is checked first (has LOS?), then Search (had a target but lost LOS?), then Patrol (default). This is the “priority” that makes Selectors powerful.

How to build it in Unity Behavior:

1. Add **On Start** and enable **Repeat** (so it loops continuously).How behavior graph works | Behavior | 1.0.13

2. Add a **Run In Parallel** node (Flow category) Behavior graphs | Behavior | 1.0.13

    - Left branch: a **Repeat** node with `Update Perception` inside (so sensing keeps updating)

    - Right branch: a **Try In Order** or **Selector-like** structure for your main decision

3. For **movement**, use built-in nodes:

    - **Navigate To Target** (uses NavMeshAgent)

    - **Navigate To Location / Target Position**

    - **Patrol** (with your waypoints)

> **Important concept:** movement and waiting nodes return **Running** for multiple frames while the action is still in progress. In Unity Behavior, nodes can have statuses like `Running`, `Success`, `Failure`, etc. Enum Node.Status | Behavior | 1.0.13

Waypoints hookup

The built-in Patrol node needs to know which waypoints to visit. There are a few ways to set this up:

Option A: Direct reference in the Patrol node (simplest)

1. Select the Patrol node in your graph
2. In the Inspector, look for a field like `Waypoints` or `Locations`
3. Set the array size (e.g., 4) and drag your waypoint GameObjects (WP_1, WP_2, etc.) directly into the slots

This works but means the waypoints are "baked into" the graph asset itself.

Option B: Blackboard variable (more flexible)

1. Create a blackboard variable: `PatrolWaypoints` of type `List<GameObject>` (or `List<Transform>`)
2. In the Patrol node, connect this blackboard variable to the waypoints field
3. On your Guard's `BehaviorGraphAgent` component, you can override this variable per-instance in the Inspector

This lets different guards use different patrol routes while sharing the same graph.

Option C: Manual patrol logic (if the Patrol node fights you)

If the built-in Patrol node is confusing or doesn't behave as expected, you can build patrol manually:

Sequence: Patrol

├── Navigate To Location (CurrentWaypoint)

└── Action: Advance Waypoint Index

This requires a small custom action that increments an index and updates a `CurrentWaypoint` blackboard variable. It's more work, but teaches you how the Patrol node works internally.

For this lab, **Option A is fine**. Options B and C are useful to know for your project.

---

13. Step 9 — Test + Debug ("Running" and reactivity)

Press Play and verify:

1. Guard patrols.
2. When player enters FOV/LOS, guard switches to chase.
3. When player breaks LOS, guard goes to last known position, waits, clears target, patrols again.

Debugging tips

- Use the Behavior Graph visual debugger to see which branch is active.
- If something "never switches," check:
    - Is `Update Perception` running continuously?
    - Are your blackboard variables correctly connected?
    - Do you have NavMesh baked?

Checkpoint questions

- Where do you observe `Running` in this graph? (Which node(s) stay active for multiple frames?)
- What would be different if every node always returned `Success` immediately?

---

14. Step 10 — Optional extensions (pick 1–3)

1. **Interrupt chase properly (Abort/Restart)**
    - Use a **Restart** or **Abort** flow node so that when `HasLineOfSight` changes, the active branch gets interrupted cleanly.
2. **Add an "Attack"**
    - If distance to player < attack range → log "Attack!" or trigger an animation.
3. **Integrate earlier labs**
    - Replace `Navigate To…` with:
        - your **grid A*** path follower from Lab 2 (global plan), or
        - your **steering** from Lab 3 (local motion)
    - This becomes your "AI stack": **BT decides**, **Pathfinding plans**, **Steering moves**.
4. **Multiple guards**
    - Add 2–5 guards. Do they clump? How could steering/separation help?

---

15. Hand-in instructions (Omniway)

Submit one of:

- A **link to your Git repository**, or
- A **.zip** (include your name in the filename)

Include:

1. Your project (or at least Assets + ProjectSettings if you zip).
2. Optional: A short **video/GIF** showing:
    - patrol → chase → search → patrol
3. A short reflection file: `Lab4_Reflection_YourName.txt` (or `.md`)

---

16. Common Mistakes

- Forgetting to enable **Repeat** on the root node (tree runs once and stops)
- Not connecting blackboard variables to node fields (values never update)
- Placing Update Perception inside the Selector instead of in parallel (perception only updates when that branch is active)
- Not baking your NavMesh properly

---

17. Reflection questions (few sentences each)

1. Compare **FSM vs Behaviour Tree** for this guard. What became easier? What became harder?

2. The **Navigate To Target** node return Running while the guard is moving. What would happen if it returned Success immediately instead? Describe the visual behaviour you'd see.

3. What did you put in the **blackboard**, and why does it belong there (vs inside one node)?

4. If you were to scale this guard into a "real enemy," what would you add next: more sensors, more actions, more structure (subgraphs), or better movement?

Appendix A — Provided C# code

Put these scripts in `Assets/Scripts/Lab4/` (or similar).
For the custom **Behavior Action** scripts: create them via the Behavior Graph editor first, then replace the generated script contents with the code below. Create a custom node | Behavior | 1.0.13

Appendix A.1 — `SimplePlayerController.cs`

```
using UnityEngine;

namespace GameAI.Lab4
{
    /// <summary>
    /// Simple WASD/Arrow movement for a capsule player.
    /// Not physics-based (keeps the lab focused on BTs).
    /// </summary>
    public class SimplePlayerController : MonoBehaviour
    {
        [SerializeField] private float moveSpeed = 6f;
        [SerializeField] private bool faceMoveDirection = true;

        private void Update()
        {
            float x = Input.GetAxisRaw("Horizontal");
            float z = Input.GetAxisRaw("Vertical");

            Vector3 input = new Vector3(x, 0f, z);
            if (input.sqrMagnitude < 0.0001f) return;

            Vector3 dir = input.normalized;
            transform.position += dir * moveSpeed * Time.deltaTime;

            if (faceMoveDirection)
            {
                transform.forward = dir;
            }
        }
    }
}
```

Appendix A.2 — `GuardSensors.cs`

```
using UnityEngine;

namespace GameAI.Lab4
{
    /// <summary>
    /// Perception helper: detects a Player target in range/FOV and
checks line-of-sight via raycast.
    /// The BT uses this through a custom action node.
    /// </summary>
    public class GuardSensors : MonoBehaviour
    {
        [Header("Target")]
        [SerializeField] private string targetTag = "Player";

        [Header("View")]
        [SerializeField] private float viewDistance = 10f;
        [Range(1f, 180f)]
        [SerializeField] private float viewAngleDegrees = 90f;

        [Header("Line of Sight")]
        [SerializeField] private Transform eyes;
        [SerializeField] private LayerMask occlusionMask = ~0; //
everything by default

        private Transform cachedTarget;

        public float ViewDistance => viewDistance;
        public float ViewAngleDegrees => viewAngleDegrees;

        private Transform EyesTransform => eyes != null ? eyes :
transform;

        private void Awake()
        {
            // Cache once; good enough for a lab. (You can improve
later.)
```

```
            GameObject go =
GameObject.FindGameObjectWithTag(targetTag);
            cachedTarget = go != null ? go.transform : null;
        }

        public bool TrySenseTarget(out GameObject target, out
Vector3 lastKnownPosition, out bool hasLineOfSight)
        {
            target = null;
            lastKnownPosition = default;
            hasLineOfSight = false;

            if (cachedTarget == null) return false;

            Vector3 eyePos = EyesTransform.position;
            Vector3 toTarget = cachedTarget.position - eyePos;

            float dist = toTarget.magnitude;
            if (dist > viewDistance) return false;

            Vector3 toTargetDir = toTarget / Mathf.Max(dist,
0.0001f);

            float halfAngle = viewAngleDegrees * 0.5f;
            float angle = Vector3.Angle(EyesTransform.forward,
toTargetDir);
            if (angle > halfAngle) return false;

            // Raycast to check occlusion
            if (Physics.Raycast(eyePos, toTargetDir, out RaycastHit
hit, dist, occlusionMask))
            {
                // If we hit something BEFORE reaching the target,
it's blocked.
                if (hit.transform != cachedTarget)
                {
                    return false;
                }
            }

            target = cachedTarget.gameObject;
```

```
            lastKnownPosition = cachedTarget.position;
            hasLineOfSight = true;
            return true;
        }
    }
}
```

Appendix A.3 — Custom Behavior Action — `UpdatePerception.cs`

```
using System;
using Unity.Behavior;
using Unity.Properties;
using UnityEngine;

namespace GameAI.Lab4
{
    /// <summary>
    /// Custom Unity Behavior Action node:
    /// - Reads GuardSensors
    /// - Writes to blackboard: Target, HasLineOfSight,
LastKnownPosition, TimeSinceLastSeen
    ///
    /// NOTE: Best workflow:
    /// 1) Create this node via the Behavior Graph editor (Create
new -> Action),
    /// 2) Then replace the generated script content with this file.
    /// </summary>
    [Serializable, GeneratePropertyBag]
    [NodeDescription(
        name: "Update Perception",
        description: "Updates Target/LOS/LastKnownPosition from
GuardSensors.",
        story: "Update perception and write to the blackboard.",
        category: "Action/Sensing",
        id: "b0c8b9b7-8f64-4c0c-a9a0-0d9e7c2e2fb8"
    )]
    public class UpdatePerception : Action
    {
```

```
        [SerializeReference] public BlackboardVariable<GameObject>
Target;
        [SerializeReference] public BlackboardVariable<bool>
HasLineOfSight;
        [SerializeReference] public BlackboardVariable<Vector3>
LastKnownPosition;
        [SerializeReference] public BlackboardVariable<float>
TimeSinceLastSeen;

        protected override Node.Status OnStart()
        {
            // Ensure we have sane defaults.
            if (TimeSinceLastSeen != null && TimeSinceLastSeen.Value
< 0f)
                TimeSinceLastSeen.Value = 9999f;

            return Node.Status.Success;
        }

        protected override Node.Status OnUpdate()
        {
            var sensors = GameObject != null ?
GameObject.GetComponent<GuardSensors>() : null;
            if (sensors == null)
            {
                // No sensors attached -> treat as "can't see
anything"
                if (HasLineOfSight != null) HasLineOfSight.Value =
false;
                if (TimeSinceLastSeen != null)
TimeSinceLastSeen.Value += Time.deltaTime;
                return Node.Status.Success;
            }

            bool sensed = sensors.TrySenseTarget(
                out GameObject sensedTarget,
                out Vector3 sensedPos,
                out bool hasLOS
            );

            if (sensed && hasLOS)
```

```
            {
                if (Target != null) Target.Value = sensedTarget;
                if (HasLineOfSight != null) HasLineOfSight.Value =
true;
                if (LastKnownPosition != null)
LastKnownPosition.Value = sensedPos;
                if (TimeSinceLastSeen != null)
TimeSinceLastSeen.Value = 0f;
            }
            else
            {
                // Keep Target as-is (we "remember" who we were
chasing),
                // but mark that we don't currently have LOS.
                if (HasLineOfSight != null) HasLineOfSight.Value =
false;
                if (TimeSinceLastSeen != null)
TimeSinceLastSeen.Value += Time.deltaTime;
            }

            // This node is a fast "service-like" update; it
finishes immediately each tick.
            return Node.Status.Success;
        }
    }
}
```

Appendix A.4 — Custom Behavior Action — `ClearTarget.cs`

```
using System;
using Unity.Behavior;
using Unity.Properties;
using UnityEngine;

namespace GameAI.Lab4
{
    [Serializable, GeneratePropertyBag]
    [NodeDescription(
        name: "Clear Target",
```

```
        description: "Clears Target and resets LOS memory.",
        story: "Forget the target and reset perception flags.",
        category: "Action/Sensing",
        id: "d1b6d9f2-7c9b-4e7b-a8e5-4c4ff1c2a4dd"
    )]
    public class ClearTarget : Action
    {
        [SerializeReference] public BlackboardVariable<GameObject>
Target;
        [SerializeReference] public BlackboardVariable<bool>
HasLineOfSight;
        [SerializeReference] public BlackboardVariable<float>
TimeSinceLastSeen;

        protected override Node.Status OnUpdate()
        {
            if (Target != null) Target.Value = null;
            if (HasLineOfSight != null) HasLineOfSight.Value =
false;
            if (TimeSinceLastSeen != null) TimeSinceLastSeen.Value =
9999f;

            return Node.Status.Success;
        }
    }
}
```