TUTORIAL DOCUMENTATION

Behavior Trees and AI Decision-Making in Unity
A Structured Educational Guide

SECTION 1: THEORETICAL FOUNDATIONS

1.1 What Are Behavior Trees?

A Behavior Tree (BT) is a hierarchical AI decision-making model used in game development to control NPC actions. Behavior Trees break down complex decisions into small, reusable nodes and evaluate those nodes every frame to determine what the AI should do.

BTs are commonly used because they allow modular behavior design, scale easily, and make debugging clearer compared to large if-else structures or finite state machines.

1.2 Node States

Every node in a Behavior Tree returns one of three states:

SUCCESS – The node's action completed successfully.
FAILURE – The node could not complete its action.
RUNNING – The node is still executing.

These states allow predictable and consistent decision-making across complex AI systems.

1.3 Types of Nodes

Action Nodes (Leaf Nodes):
These perform actions such as checking distance, moving, or selecting a waypoint.

Composite Nodes (Branch Nodes):
There are two major composite types:

Selector (OR logic):
Evaluates children in order. If any child returns SUCCESS or RUNNING, the Selector stops evaluating and returns that result.
Used for fallback logic such as: Try to Chase. If that fails, Patrol.

Sequence (AND logic):
Evaluates children in order. If any child returns FAILURE, the Sequence stops and returns FAILURE. If all children succeed, the Sequence returns SUCCESS.
Used for multi-step behaviors.

Decorator Nodes:
Modify a single child's behavior (invert result, repeat, limit). Not used in your patrol/chase system but useful in advanced systems.

1.4 Why Use Behavior Trees?

Behavior Trees provide:

• Modularity
• Scalability
• Clear visual structure
• Easy debugging
• Real-time reactive decision-making

They are widely used in modern 3D games because they allow designers and programmers to add new behaviors without rewriting existing logic.

SECTION 2: SYSTEM ARCHITECTURE

The implemented AI uses a Selector node with two child behaviors:

1. Chase Behavior – Active when the player is within a detection radius.
2. Patrol Behavior – Default movement between waypoints.

Selector logic:
Try Chase first. If Chase fails, run Patrol.

2.1 Behavior Tree Diagram

ROOT
→ Selector
→ Chase Behavior
→ Patrol Behavior

2.2 Behavior Flow Summary

Chase Behavior is attempted every frame.
If the player is close enough, Chase runs.
If not, Chase fails and Patrol becomes active.
This evaluation loop repeats every frame for responsive AI.

SECTION 3: STEP-BY-STEP IMPLEMENTATION GUIDE (CONCEPTUAL)

This section outlines the conceptual and structural steps of building the Behavior Tree AI system. All implementation details are described without code.

3.1 Unity Scene Setup

1. Create a ground plane named GROUND.
2. Add two Capsule objects: PLAYER and ENEMY.
3. Place them far enough apart to observe transitions between patrol and chase.

3.2 Navigation Setup

NavMesh Surface Setup:

1. Select GROUND.
2. Add a NavMeshSurface component.
3. Bake the NavMesh so the enemy can navigate.

Enemy Navigation Setup:

1. Add a NavMeshAgent component to the ENEMY.
2. Configure speed, acceleration, stopping distance as needed.

3.3 Waypoint Setup

1. Create empty object named WAYPOINTS.
2. Add children: waypoint1, waypoint2, waypoint3.
3. Position them to form a patrol route.

The enemy will cycle through these points.

3.4 Behavior Tree Folder Organization

Scripts should follow this structure:

BehaviourTree
• Core (node definitions)
• Composites (Selector, Sequence)
• Actions (Patrol behavior, Chase behavior)

EnemyAI script uses these Behavior Tree classes to evaluate actions each frame.

3.5 Patrol Behavior Description

Patrol behavior consists of:

1. Moving to the next waypoint.
2. Detecting arrival.
3. Selecting the next waypoint.
4. Looping indefinitely.

Patrol is a continuous RUNNING behavior.

3.6 Chase Behavior Description

Chase behavior consists of:

1. Measuring distance to the player.
2. If within chase range, pursue player.
3. If outside chase range, return FAILURE.

This FAILURE result triggers the Selector to choose Patrol instead.

3.7 Selector Logic

Selector evaluation:

Try Chase Behavior first.
If Chase returns FAILURE, switch to Patrol Behavior.
If Chase returns RUNNING or SUCCESS, continue chasing.

This ensures appropriate fallback behavior.

SECTION 4: DIAGRAMS

4.1 Behavior Tree Diagram

ROOT
→ Selector
→ Chase Behavior
→ Patrol Behavior

4.2 Patrol Loop Diagram

Waypoint 1 → Waypoint 2 → Waypoint 3 → back to Waypoint 1 → repeat.

4.3 Chase Condition Diagram

Check Player Distance
• Within chaseRange → Chase
• Not within range → Fail → Patrol

SECTION 5: EXERCISES

These exercises help students understand Behavior Tree extensions and decision-making flow.

Exercise 1: Add Idle Behavior
After reaching a waypoint, enemy should pause for a few seconds before moving on.
Students learn timing logic and state transitions.

Exercise 2: Add Attack Behavior
If enemy is extremely close to the player:
• Stop moving
• Perform attack action (animation or debug text)

This requires combining conditions using a Sequence.

Exercise 3: Add Search Behavior
If the player escapes chase range:
• Go to last known player position
• Turn and look around
• Return to patrol

Students learn fallback logic and state memory.

Exercise 4: Add Hearing Detection
Player makes noise events.
Enemy reacts to noise even when player is out of visual range.

Students learn multi-sensor AI systems.

Exercise 5: Add Line-of-Sight Check
Enemy only chases the player if there is no obstacle between them.

Students incorporate vision cones and raycasting logic concepts.

SECTION 6: SUMMARY

This tutorial covered:

- Behavior Tree theory
- Node types and states
- Selector-based logic
- Patrol and chase design
- Behavior reevaluation every frame
- System architecture overview
- Diagrams illustrating BT flow
- Exercises to extend learning

Behavior Trees are a foundational AI technique for creating modular, scalable, and maintainable NPC behavior in Unity and other modern game engines.