Pedagogical Report

AI Perception and Pursuit System Tutorial

Created by: Nachiket Sahare

December 2024

1. Teaching Philosophy

Target Audience

This tutorial targets beginner to intermediate Unreal Engine users who have basic blueprint knowledge but limited experience with game AI systems. The ideal student:

• Understands fundamental Unreal Engine concepts (actors, components, blueprints)

• Can read and create simple blueprint logic (variables, branches, functions)

• Wants to learn industry-standard AI techniques

• Is building an action, stealth, or adventure game requiring enemy AI

This audience was selected because AI programming presents a significant learning curve for beginners, yet the fundamental concepts are accessible when broken down systematically. By targeting intermediate users, the tutorial can skip basic engine operations while focusing deeply on AI-specific systems.

Learning Objectives

Students will be able to:

1. Knowledge: Explain how Behavior Trees, Blackboards, and AI Perception work together to create intelligent enemy behavior

2. Comprehension: Describe the advantages of Behavior Trees over Finite State Machines for game AI

3. Application: Implement a complete chase and pursuit system using PawnSensing, Behavior Trees, and distance-based logic

4. Analysis: Debug common AI issues by identifying where in the perception-decision-action pipeline problems occur

5. Synthesis: Extend the base system with new behaviors (patrol, search, multiple enemy types)

6. Evaluation: Assess AI behavior quality and adjust parameters for desired gameplay feel

Pedagogical Approach and Rationale

The tutorial follows a scaffolded learning approach:

Theory First: Students begin with concise theoretical foundations explaining why these systems exist and how they work conceptually. This provides mental models before implementation.

Incremental Building: The implementation builds complexity gradually. Students first create the AI Controller, then Blackboard, then Behavior Tree structure, then perception, then services. Each component is tested before adding the next.

Visual Learning: Every implementation step includes screenshots showing exactly what students should see. This reduces ambiguity and helps visual learners.

Immediate Feedback: Students test after each major component, providing immediate feedback on whether they implemented correctly before moving forward.

Error Prevention: The tutorial emphasizes common mistakes (case-sensitive key names, Observer Aborts settings) before students encounter them, reducing frustration.

Guided Discovery: Practice exercises encourage experimentation with parameters, allowing students to discover how systems respond to changes.

2. Concept Deep Dive

Technical Explanation

Behavior Trees as Graph Traversal

Behavior Trees implement a tree traversal algorithm executed each tick (frame). Starting from the root, the algorithm performs depth-first search, evaluating nodes based on their type:

Selector nodes implement OR logic: evaluate children left-to-right, return SUCCESS on first success, FAILURE if all fail.

Sequence nodes implement AND logic: evaluate children left-to-right, return FAILURE on first failure, SUCCESS if all succeed.

This traversal approach differs fundamentally from Finite State Machines, which maintain explicit state and require n-squared transition rules. Behavior Trees re-evaluate from root each tick, enabling reactive behavior without complex transition management.

Observer Pattern and Reactive Systems

The Observer Aborts mechanism implements the observer pattern. Decorators register as observers of Blackboard key changes. When a monitored key changes value:

1. Blackboard notifies all registered observers
2. Decorators evaluate their conditions against new values
3. If condition becomes false, decorator aborts execution of its subtree
4. Control returns to root for immediate re-evaluation

This creates zero-latency reactivity. AI responds to world changes immediately rather than waiting for current action completion.

Distance Calculations and Performance

Distance checks use Euclidean distance calculation: sqrt((x2-x1)^2 + (y2-y1)^2 + (z2-z1)^2). However, squared distance comparisons avoid the expensive square root operation. When checking if distance > threshold, we can compare squared distance > threshold^2.

Services run at intervals (every 0.5 seconds) rather than every frame, reducing computational cost from O(n * fps) to O(n * 2) per second for n AI agents. This optimization is crucial for games with many simultaneous AI agents.

Game Design Connections

The perception and pursuit system directly impacts gameplay feel:

Detection Range defines engagement distance. Shorter ranges (300-600) create ambush scenarios, longer ranges (1000-2000) create open combat.

Pursuit Distance determines how persistent enemies feel. Short pursuit creates safe zones, long pursuit creates tension and resource management (stamina, hiding spots).

Hysteresis (different engage vs disengage distances) prevents flickering behavior and creates natural-feeling transitions. This small technical detail significantly impacts perceived intelligence.

Attack Range timing affects combat feel. Tight timing (200 units) requires precise positioning. Loose timing (300+ units) is more forgiving but less tactical.

These parameters are not arbitrary technical values but fundamental game design variables that define player experience. Understanding this connection helps students design AI that serves gameplay goals rather than just implementing technical systems.

3. Implementation Analysis

Architecture Decisions

Separation of Concerns

The implementation strictly separates three concerns:

1. Perception (PawnSensing component): Responsible only for detecting stimuli and firing events. Does not contain decision logic.
2. Decision (Behavior Tree + Blackboard): Contains all behavioral logic but no sensory code or movement code.
3. Action (Tasks and MoveTo): Executes behaviors but makes no decisions about what to do.

This separation enables independent testing, easier debugging, and component reuse across different AI types.

Why PawnSensing Over AI Perception

The tutorial uses PawnSensing rather than the more advanced AI Perception system. This decision was deliberate:

PawnSensing is simpler to configure (3 parameters vs 15+), reducing cognitive load for beginners. AI Perception's additional complexity (hearing, prediction, team awareness) would distract from core Behavior Tree concepts.

Event-based detection (On See Pawn) is more intuitive than continuous query systems for beginners. Students can directly see cause and effect.

For production games, AI Perception is recommended. The tutorial notes this explicitly, treating PawnSensing as a learning tool rather than production recommendation.

Service-Based Monitoring

Distance checking could be implemented in multiple ways (decorator, task check, Event Tick). The Service approach was chosen because:

Services teach an important concept: background monitoring that is scoped to specific tree branches.

Interval-based checking (every 0.5s) is more performant than per-frame checking.

Service logic is cleanly separated from behavior execution, following single responsibility principle.

Performance Considerations

For a single enemy, performance is negligible. However, the architecture scales to multiple enemies:

Behavior Trees are instanced (multiple AI share the same tree definition in memory).

Service intervals prevent O(n*fps) scaling. With 20 enemies and 60fps, Event Tick approach costs 1200 checks/second. Service approach costs 40 checks/second (20 enemies * 2 checks/second).

NavMesh pathfinding uses A* with spatial hashing, providing logarithmic lookup time for path nodes.

Blackboard key lookups use hash tables, providing O(1) access time regardless of key count.

Scalability

The base implementation extends naturally:

Adding States: New sequences can be added to the Selector without modifying existing logic. A patrol state, search state, or flee state plugs in seamlessly.

Enemy Variations: Different enemy types inherit the same AI Controller but override parameters (detection range, speed, aggression). Behavior Tree instances share structure but read different blackboard values.

Team Behaviors: Blackboards can be shared between AI agents for squad coordination. One enemy's detection can alert others through shared keys.

Complex Behaviors: Subtrees can be extracted and reused. A "search area" subtree could be shared across enemy types, stealth games, and investigation mechanics.

4. Assessment and Effectiveness

Validation Criteria

Students successfully learned the material if they can:

Technical Competency:

• Create a working enemy AI that chases and stops appropriately (minimal requirement)

• Debug and fix provided broken implementations (comprehension)
• Modify detection ranges and explain behavioral impact (parameter understanding)
• Add a new state (patrol or search) without guidance (synthesis)

Conceptual Understanding:

• Explain why Behavior Trees are preferred over FSMs in modern games
• Describe the role of Blackboard in decoupling AI subsystems
• Identify which component is responsible when AI behavior fails (perception vs decision vs action)

Assessment methods include practice exercises (graded), debugging challenges (must-pass), and optional extension projects (bonus).

Expected Student Challenges

Based on common patterns in game development education, students will likely struggle with:

1. Blackboard Key Name Mismatches (90% of students)

Students will create keys with one name and reference them with another (targetActor vs TargetActor). The tutorial addresses this by:

• Explicitly warning about case sensitivity before the issue occurs
• Showing exact spelling in screenshots
• Including this as the first item in the debugging guide
• Recommending dropdown selection instead of manual typing

2. Observer Aborts Misconfiguration (75% of students)

Students will leave Observer Aborts set to None, causing AI to ignore blackboard changes. The tutorial addresses this by:

• Explaining why this setting matters before configuration
• Including a screenshot showing the correct setting
• Listing this as a primary debugging check

3. Service Not Attached (60% of students)

Students will create the service but forget to attach it to the Behavior Tree node. The tutorial addresses this by:

• Including explicit attachment steps with screenshots
• Testing the service immediately after creation
• Providing debug prints to verify service execution

4. NavMesh Missing (50% of students)

Students will forget to place NavMesh Bounds Volume, causing AI to teleport or get stuck. The tutorial addresses this by:

• Including NavMesh setup as a mandatory step early in the process
• Teaching the P key visualization for immediate visual confirmation
• Providing a dedicated debugging section for NavMesh issues

5. Conceptual: Perception vs Decision Confusion (40% of students)

Students will struggle to understand which component is responsible for which behavior (e.g., why doesn't PawnSensing make the AI stop chasing?). The tutorial addresses this by:

• Beginning with a clear architecture diagram showing separation of concerns
• Explaining the perception-decision-action pipeline explicitly
• Using consistent terminology throughout (sensing vs thinking vs acting)

How This Tutorial Addresses These Challenges

The tutorial structure incorporates error prevention strategies:

1. Explicit Warnings: Common mistakes are flagged before students encounter them

2. Incremental Testing: Each component is tested immediately after creation, isolating errors
3. Visual Confirmation: Screenshots show exactly what correct implementation looks like
4. Debugging Guide: Systematic troubleshooting process for each common issue
5. Print String Strategy: Students learn to add debug output to identify where logic fails

Success Metrics

Tutorial effectiveness can be measured through:

Completion Rate: Percentage of students who successfully build working AI (target: 85%+)
Time to Completion: Average time from start to working implementation (target: 2-3 hours)
Error Recovery: Percentage of students who resolve issues using debugging guide without additional help (target: 70%+)
Extension Projects: Percentage of students who successfully complete practice exercises and add new features (target: 50%+)
Knowledge Retention: Ability to explain core concepts one week after completion (measured through quiz or discussion)

Conclusion

This tutorial represents a pedagogically-informed approach to teaching game AI concepts. By combining theoretical foundations with hands-on implementation, scaffolded complexity progression, and comprehensive debugging support, it aims to make advanced AI techniques accessible to intermediate developers.

The choice of Behavior Trees as the teaching topic is strategic: they represent the industry standard, scale from simple to complex implementations, and teach transferable concepts (hierarchical decision-making, reactive systems, separation of concerns) applicable beyond game development.

Success is measured not just by technical implementation but by conceptual understanding. Students who complete this tutorial should be able to extend the system independently, debug novel issues systematically, and apply these architectural patterns to new problems.

The tutorial acknowledges that different students learn differently, providing multiple learning modalities: text explanation, visual diagrams, hands-on implementation, video demonstration, and practice exercises. This multimodal approach increases accessibility and retention.

Future iterations could enhance the tutorial with interactive debugging exercises, video walkthroughs of each implementation step, and a community forum for peer learning. However, the current structure provides a solid foundation for self-directed learning in game AI development.

End of Pedagogical Report