

1. Introduction

This document describes the detailed design for the *Immersive NPC Behavior System*, which aims to create intelligent, adaptive, and immersive behaviors for non-player characters (NPCs) in-game environments. This system integrates seamlessly with the *Modular World Generation* tool to enhance interactive and dynamic gameplay experiences.

2. System Overview

The system comprises two primary components:

- **Behavior Logic Module:** Responsible for NPC decision-making, reactions, and interactions, using rule-based engines, decision trees, and storytelling integration.
- **Data and Integration Layer:** This layer stores NPC behavior configurations, and contextual data, and facilitates integration with the *Modular World Generation* tool to ensure consistency in scene logic and environmental interactions.

3. Design Considerations

- Assumes Unity engine version 2020.x or higher for compatibility with modern NavMesh and AI tools.
- Designed for single-player games with a focus on story-driven or sandbox environments.
- Uses Unity's built-in components for animation, physics, and navigation (e.g., NavMesh).

4. System Architecture

The system follows a modular architecture divided into:

1. **Behavior Logic:**
 - Rule-based decision-making for immediate responses.
 - Decision trees for multi-layered decision-making processes.
 - Integration with a storytelling package to align NPC behaviors with narrative goals.
2. **Integration Layer:**
 - Interfaces with the *Modular World Generation* tool to ensure NPC behaviors align with scene elements like terrain, walkable areas, and objects.
3. **Data Handling:**
 - Stores NPC configurations, state variables, and contextual triggers to manage dynamic responses.

5. Component Design

- **Behavior Logic Module:**
 - **Purpose:** Generates and manages adaptive NPC behaviors.
 - **Input:** Environmental data (terrain, objects, player actions) and NPC state variables.
 - **Output:** Contextually appropriate NPC actions (e.g., movement, interactions, reactions).
 - **Dependencies:** Unity's NavMesh, Animator, and AudioSource components.
- **Data and Integration Layer:**
 - **Purpose:** Links NPC behaviors to environmental elements and story components.
 - **Input:** Scene configuration data from *Modular World Generation* and narrative triggers.
 - **Output:** Updated NPC states and actions synchronized with the scene.
 - **Dependencies:** Modular World Generation tool and storytelling package.

6. Data Design

- **NPC Behavior Schema:**
 - **NPC_ID (int):** Unique identifier for each NPC.
 - **Personality_Type (enum):** Defines NPC traits (e.g., Friendly, Hostile).
 - **Situation_State (enum):** Current context for decision-making (e.g., Normal, Panic).

- **Behavior_Tree_ID (int)**: Links to a pre-defined decision tree for complex logic.
- **Position (Vector3)**: NPC's current location.
- **Story Triggers Schema:**
 - **Trigger_ID (int)**: Unique identifier for story events.
 - **Associated_NPCs (list)**: NPCs affected by the trigger.
 - **Action (string)**: Behavioral changes tied to the trigger (e.g., flee, assist).

7. Detailed Class/Function Design

Class: NPC

- **Properties:**
 - **PersonalityType**: Friendly, Hostile, Neutral.
 - **SituationState**: Normal, Panic, Emergency.
 - **Health, Speed, Animations, NavMeshAgent**.
- **Key Functions:**
 - **UpdateSituation(SituationType newSituation)**: Adjusts NPC's behavior based on the current context.
 - **Interact(GameObject interactor)**: Handles player interactions.
 - **TakeDamage(int damage)**: Updates NPC's health and triggers appropriate responses.

```
class NPC {
    // Enum definitions for PersonalityType and SituationState
    enum PersonalityType { FRIENDLY, HOSTILE, NEUTRAL }
    enum SituationState { NORMAL, PANIC, EMERGENCY, HOSTILE }

    // NPC Properties
    PersonalityType personalityTag;
    SituationState situationTag;
    int health;
    float walkingSpeed;
    float runningSpeed;

    // Physical Properties
    Vector3 position;
    NavMeshAgent navMeshAgent;
    Animator animator;

    // Runtime Variables
    bool isInteracted;
```

```
bool isRunning;
```

```
// Initialize NPC components
```

```
void Awake() {  
    navMeshAgent = GetComponent<NavMeshAgent>();  
    animator = GetComponent<Animator>();  
}
```

```
// Updates the situation of the NPC
```

```
void UpdateSituation(SituationState newSituation) {  
    situationTag = newSituation;  
    switch (newSituation) {  
        case SituationState.NORMAL:  
            navMeshAgent.speed = walkingSpeed;  
            isRunning = false;  
            break;  
        case SituationState.PANIC:  
        case SituationState.EMERGENCY:  
            navMeshAgent.speed = runningSpeed;  
            isRunning = true;  
            break;  
        case SituationState.HOSTILE:  
            PrepareForHostile();  
            break;  
    }  
    UpdateAnimationState();  
}
```

```
// Handles player interaction
```

```
bool Interact(GameObject interactor) {  
    if (isInteracted) return false;  
    isInteracted = true;  
    HandleInteraction(interactor);  
    return true;  
}
```

```
// Handles taking damage
```

```
bool TakeDamage(int damage) {  
    health -= damage;  
    if (health <= 0) {  
        Die();  
        return false;  
    }  
}
```

```

    PlayAnimation("Hurt");
    return true;
}

// NPC death behavior
void Die() {
    navMeshAgent.enabled = false;
    PlayAnimation("Death");
}

// Updates animation state based on NPC status
void UpdateAnimationState() {
    if (animator != null) {
        animator.SetBool("IsRunning", isRunning);
        animator.SetInteger("SituationState", (int)situationTag);
    }
}

// Prepares the NPC for hostile actions
void PrepareForHostile() {
    PlayAnimation("HostileReady");
}

// Plays an animation by name
void PlayAnimation(string animationName) {
    if (animator != null) animator.Play(animationName);
}
}

```

Class: BehaviorTree

- **Purpose:** Encapsulates complex decision-making processes.
- **Methods:**
 - **EvaluateNode():** Processes current conditions and returns the next action.
 - **UpdateTree():** Updates decision-making logic based on environmental inputs.

```
class BehaviorTree {  
    // Node structure for the decision tree  
    class Node {  
        public string Condition;  
        public Action ExecuteAction;  
        public Node TrueNode;  
        public Node FalseNode;  
  
        public Node(string condition, Action action) {  
            Condition = condition;  
            ExecuteAction = action;  
        }  
    }  
    Node rootNode;  
  
    // Evaluates the decision tree based on the current state  
    void EvaluateNode(Node currentNode) {  
        if (currentNode == null) return;  
        bool conditionMet = CheckCondition(currentNode.Condition);  
        if (conditionMet) {  
            currentNode.ExecuteAction?.Invoke();  
            EvaluateNode(currentNode.TrueNode);  
        } else {  
            EvaluateNode(currentNode.FalseNode);  
        }  
    }  
  
    // Updates the behavior tree logic dynamically  
    void UpdateTree(Node newRootNode) {  
        rootNode = newRootNode;  
    }  
    // Placeholder for condition evaluation  
    bool CheckCondition(string condition) {  
        // Evaluate the condition (to be implemented based on specific game logic)  
        return true;  
    }  
}
```

Class: StoryManager

- **Purpose:** Links NPC behaviors with narrative events.
- **Methods:**
 - `TriggerEvent(int triggerID)`: Executes associated NPC behaviors.

```
class StoryManager {  
    // Story trigger structure  
    class StoryTrigger {  
        public int TriggerID;  
        public List<NPC> AssociatedNPCs;  
        public Action TriggerAction;  
  
        public StoryTrigger(int id, List<NPC> npcs, Action action) {  
            TriggerID = id;  
            AssociatedNPCs = npcs;  
            TriggerAction = action;  
        }  
    }  
}  
  
List<StoryTrigger> triggers = new List<StoryTrigger>();  
  
// Adds a new story trigger  
void AddTrigger(int id, List<NPC> npcs, Action action) {  
    StoryTrigger newTrigger = new StoryTrigger(id, npcs, action);  
    triggers.Add(newTrigger);  
}  
  
// Executes a story trigger  
void TriggerEvent(int triggerID) {  
    StoryTrigger trigger = triggers.Find(t => t.TriggerID == triggerID);  
    if (trigger == null) return;  
  
    trigger.TriggerAction?.Invoke();  
    foreach (NPC npc in trigger.AssociatedNPCs) {  
        npc.UpdateSituation(NPC.SituationState.NORMAL); // Example behavior  
    }  
}
```

interface IDynamicBehavior:

// NPC-related functions

UpdateSituation(newSituation: NPC.SituationType)

Interact(interactor: GameObject) -> bool

SetDestination(newDestination: Vector3)

TakeDamage(damageAmount: int) -> bool

Die()

HandleFriendlyInteraction(interactor: GameObject)

HandleHostileInteraction(interactor: GameObject)

HandleNeutralInteraction(interactor: GameObject)

FindSafeSpot()

PrepareForHostile()

SetFriendlyBehavior()

PlayAnimation(animationName: string)

PlaySound(soundName: string)

// Vehicle-related functions

StartDriving(targetDestination: Vector3)

StopDriving()

StartParking()

FinishParking()

UpdateVehicle()

MoveTowardsDestination()

UpdateRotation()

CheckArrival()

// Tool-related functions

TryGrab() -> bool

Release()

MoveTo(newPosition: Vector3, newRotation: Vector3) -> bool

Break() -> bool

PlayBreakEffects()

8. User Interface Design

The system leverages Unity's built-in UI tools to provide developers with:

- A behavior editor for creating and assigning NPC decision trees.
- Visual tools for mapping NPC interactions to environmental or narrative triggers.
- Debugging overlays to test NPC reactions in real time.

Immersive NPC Behavior System

Number Of overall NPCs

Number Of NPC Types

Type "A"
⋮

128

Type "A"
⋮

Default Routine ▼

Generate

* A "Type" field will pop up based on the number in "Number Of NPC Types".

* A slider for each type of NPC can determine the amount of said NPC

* For each NPC type the user may pre-determine the routine. Otherwise, the routine is set to "Default"

* Changes to routines and behaviors can be modified later on.