LiquidX Programmer Test Documentation:

Contents

System Design and Structure:

State Machine Design: The code uses a state machine design pattern, which is a behavioral design pattern that provides a systematic and loosely coupled way to manage states and transitions between states. This is evident in the ChangeState method in the Character class, which changes the current state of the NPC.

Component-Based Design: The code follows a component-based design where different functionalities are encapsulated in separate classes. For example, the SoundDetectorSensor class is responsible for sound detection, and the FieldOfViewSensor class is responsible for field of view detection. This design allows for high cohesion and low coupling, making the code more maintainable and scalable.

Use of Interfaces: The code uses interfaces like IHasPathFinding and ICanHear to define contracts for classes. This ensures that the classes implementing these interfaces will have certain methods, providing a level of abstraction and making the code more flexible and extensible.

Expandability:

Modular NPC Behaviors: The NPC behaviors are designed to be modular and reusable. Each behavior is encapsulated in a separate class that extends NpcBehaviorBase<T>. This design allows new behaviors to be easily added in the future without having to make major changes to the existing system.

Expandable Sensor System: The sensor system is designed to be expandable. New types of sensors can be easily added by creating a new class that extends the SensorBase class and implements the required interfaces.

Flexible Pathfinding: The pathfinding system is designed to be flexible. Different pathfinding algorithms can be easily implemented and used by changing the PathfindingAlgorithmEnum.

how to use and expand the classes:

1. Character Class:

This class serves as the base class for all characters in the game. It contains properties and methods that are common to all characters, such as `ChangeState()`, which changes the current state of the character, and `GetStateByName()`, which retrieves a state by its name.

To use this class, you would typically create a new class that inherits from `Character` and override the necessary methods to implement the specific behavior for that character.To expand this class, you could add new properties or methods that are common to all characters. For example, you could add a `Health` property if all characters have health, or a `TakeDamage()` method if all characters can take damage.

2. NPC Class:

This class inherits from `Character` and represents a non-player character (NPC) in the game. It contains additional properties and methods that are specific to NPCs, such as `CurrentTarget`, which represents the current target position that the NPC is moving towards, and `SetCurrentTarget()`, which sets the current target position.

To use this class, you would typically create a new class that inherits from `NPC` and override the necessary methods to implement the specific behavior for that NPC.

To expand this class, you could add new properties or methods that are specific to NPCs. For example, you could add a `PatrolRoute` property if all NPCs have a patrol route, or a `FollowPlayer()` method if all NPCs can follow the player.

3. SensorController Class:

This class is responsible for managing the sensors of an NPC. It contains a list of `SensorBase` objects, which represent the sensors attached to the NPC, and methods for initializing and updating these sensors.

To use this class, you would typically create a new instance of `SensorController` and add the necessary sensors to it. Then, you could call the `InitializeSensor()` method to initialize a sensor with a set of detection events, and the `UpdateSensor()` method to update the sensors.

To expand this class, you could add new methods for managing the sensors. For example, you could add a `RemoveSensor()` method if you need to remove sensors from the NPC, or a `GetSensorByName()` method if you need to retrieve a sensor by its name.

4. SensorBase Class:

This class represents a sensor that can detect certain events. It contains properties for the sensor's name and detection events, and a `Detect()` method that detects the events.

To use this class, you would typically create a new class that inherits from `SensorBase` and override the `Detect()` method to implement the specific detection logic for that sensor.

To expand this class, you could add new properties or methods that are common to all sensors. For example, you could add a `Range` property if all sensors have a range, or a `SetRange()` method if you need to set the range of a sensor.

5. StateMachine Class:

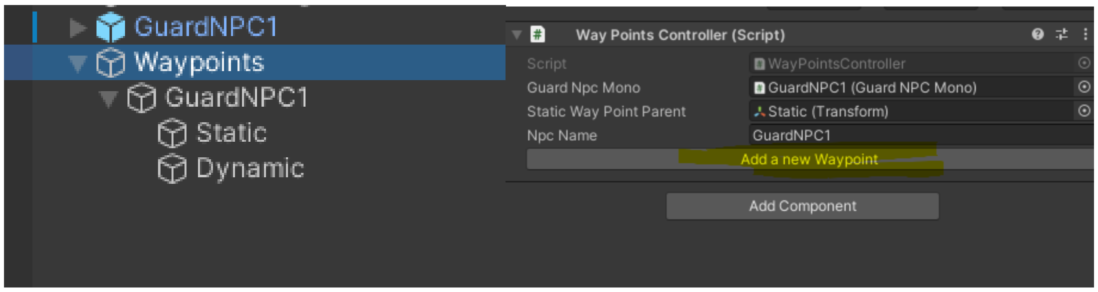
This class represents a state machine that can transition between different states. It contains a list of `IBaseState` objects, which represent the states of the state machine, and methods for changing the current state and updating the state machine.

To use this class, you would typically create a new instance of `StateMachine` and add the necessary states to it. Then, you could call the `ChangeState()` method to change the current state, and the `UpdateStateMachine()` method to update the state machine.

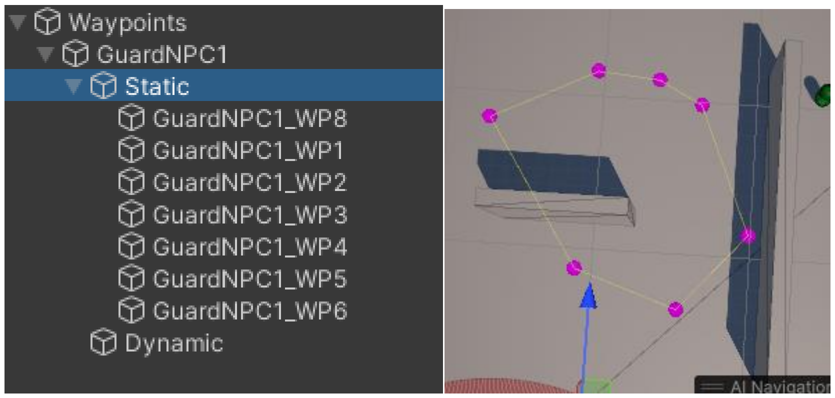
To expand this class, you could add new methods for managing the states. For example, you could add a `RemoveState()` method if you need to remove states from the state machine, or a `GetStateByName()` method if you need to retrieve a state by its name.

Create a WayPoints for NPC

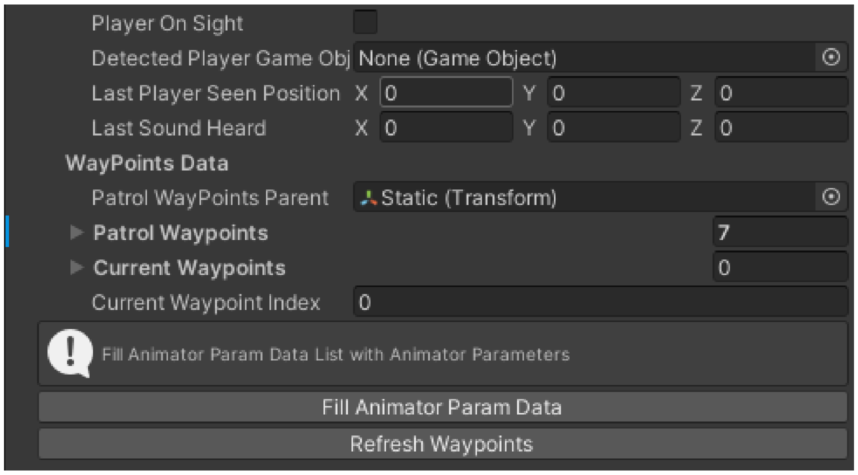
Added GuardNPC1 prefab to the scene, based on the name of NPC prefab in the below game objects been created in Scene, To add a new Waypoint click on the “Add a new Waypoint” button as the below images.



Script	WayPointsController
Guard Npc Mono	GuardNPC1 (Guard NPC Mono)
Static Way Point Parent	Static (Transform)
Npc Name	GuardNPC1



Sometimes you need refresh manually to the created waypoints show in the Editor by use the editor use the “Refresh Waypoint” button on the NPC prefab.



Player On Sight	
Detected Player Game Obj	None (Game Object)
Last Player Seen Position	X 0 Y 0 Z 0
Last Sound Heard	X 0 Y 0 Z 0
WayPoints Data	
Patrol WayPoints Parent	Static (Transform)
Patrol Waypoints	7
Current Waypoints	0
Current Waypoint Index	0

Fill Animator Param Data List with Animator Parameters