HUMAN SENSORY PROCESS FOR ARTIFICIAL INTELLIGENCE

-

DOCUMENTATION

Communication Classes

Purpose and functionality of several classes used for managing TCP communication in a Unity application. The classes involved are `SocketManager`, `CommunicationInitializer`, `SendingsHandler`, `ReceiverHandler`, and `CommunicationHandler`.

Class Overview

SocketManager

Purpose:
The `SocketManager` class manages TCP socket communication. It sets up a TCP listener, accepts client connections, and handles the communication flow using dedicated handler classes.

Methods:

- SocketManager(): Constructor for the class.
- Initialize(): Initializes the TCP listener and accepts a client connection.
- StartCommunication(WorldClass board, Vector3 avatarPosition, Vector3 avatarForward): Starts the communication thread.
- GetInfo(WorldClass board, Vector3 avatarPosition, Vector3 avatarForward): Initializes the connection and continuously checks for incoming data.
- ReceiveAction(): Receives an action from the network stream.
- ReceiveNumber(): Receives a float number from the network stream.
- SendNewPosition(Vector3 newPosition): Sends a new position to the network stream.
- SendNewValue(float value): Sends a new float value to the network stream.
- SendNewList(List<IWorldObject> listToSend, string sendListName): Sends a list of `IWorldObject` to the network stream.
- SendNewList(List<ITemperatureEmitter> listToSend, string sendListName): Sends a list of `ITemperatureEmitter` to the network stream.
- SendNewList(List<ISoundEmitter> listToSend, string sendListName): Sends a list of `ISoundEmitter` to the network stream.
- SendNewPossiblePositions(List<Vector3> possiblePositions): Sends a list of possible positions (Vector3) to the network stream.

CommunicationInitializer

Purpose:
The `CommunicationInitializer` class initializes communication by sending initial game states and possible positions. It interacts with an instance of `SocketManager` to send data over the network.

Methods:

- CommunicationInitializer(SocketManager socketManager): Constructor for the class.
- InitGame(Vector3 avatarPosition, Vector3 avatarForward): Initializes the game by sending the avatar's position and forward direction.
- InitPossiblePositions(WorldClass board): Sends the possible positions on the board to the client.

SendingsHandler

Purpose:
The `SendingsHandler` class is responsible for sending various types of data (like positions, values, and lists) over a network stream using a `CommunicationHandler` instance.

Methods:

- SendingsHandler(NetworkStream networkStream): Constructor for the class.
- SendNewPosition(Vector3 newPosition): Sends a new position to the network stream.
- SendNewValue(float value): Sends a new float value to the network stream.
- SendNewList(List<IWorldObject> listToSend, string sendListName): Sends a list of `IWorldObject` to the network stream.
- SendNewPossiblePositions(List<Vector3> possiblePositions): Sends a list of possible positions (Vector3) to the network stream.
- SendObject(GameObject gameObject): Sends a GameObject's relevant data to the network stream.

ReceiverHandler

Purpose:

The `ReceiverHandler` class is responsible for receiving actions and numbers from a network stream using a `CommunicationHandler` instance.

Methods:

- ReceiverHandler(NetworkStream networkStream): Constructor for the class.
- ReceiveAction(): Receives an action from the network stream and parses it.
- ReceiveNumber(): Receives a float number from the network stream.
- ParseAction(string action): Parses the received action string.

CommunicationHandler

Purpose:
The `CommunicationHandler` class handles communication over a network stream, including sending and receiving data.

Methods:

- CommunicationHandler(NetworkStream networkStream): Constructor for the class.
- ReceiveAction(): Receives an action string from the network stream.
- ReceiveNumber(): Receives a float number from the network stream.
- SendNumber(string number): Sends a number string to the network stream.
- SendName(string name): Sends a name string to the network stream.
-

WorldClass

Class Overview

The `WorldClass` is a singleton that represents the game world in a Unity application. It manages the state of the game world, including the ambient temperature, the dimensions of the world grid, and the positions of various objects within the world. The class ensures that only one instance of `WorldClass` exists, initializes possible positions for objects, and handles the resetting of object positions.

Properties

- Instance: Static instance of the `WorldClass`, ensuring a singleton pattern.
- AmbientTemperature: The ambient temperature of the world.

- Rows: The number of rows in the world grid.
- Columns: The number of columns in the world grid.
- Foods: List of food objects in the world.
- TotalObjects: List of all world objects.
- PossiblePositions: List of possible positions for objects in the world.

Methods

Awake()

Ensures that there is only one instance of `WorldClass`. If an instance already exists and it's not this one, the current object is destroyed. Otherwise, this instance is assigned to the static `Instance` property.

Start()

Called on the frame when a script is enabled just before any of the Update methods are called the first time. Initializes possible positions for objects and initializes objects in the world.

InitializePossiblePositions()

Populates the `possiblePositions` list based on the dimensions of the world grid (rows and columns). Each position in the grid is added as a `Vector3` to the `possiblePositions` list.

InitializeObjects()

Finds objects in the world based on their tags (e.g., "Fire", "Bulb", "Apple", "Speaker"). Removes their positions from the `possiblePositions` list and adds them to the `totalObjects` list. Additionally, specific types of objects are added to their respective lists (e.g., foods). Logs an error if a required component is not found on an object.

ResetObject(IWorldObject worldObject)

Resets the position of a given world object to a new possible position. The current position of the object is added back to the `possiblePositions` list, and a new position is randomly selected from the list of possible positions. The object is then moved to the new position, and the new position is removed from the `possiblePositions` list.ç

Avatar Classes

Avatar

- Purpose: Represents the Avatar in the game, handling its senses, drives, and movement.
- Properties:
  - `IAvatarSenses Senses { get; }`: The Avatar's senses.
  - `IAvatarDrives Drives { get; }`: The Avatar's drives.
- Methods:
  - `Awake()`: Initializes the avatar's drives, senses, and manual movement components.
  - `Start()`: Initializes the avatar's manual movement with its senses and animation controller.
  - `ChangeForward(Vector3 forward)`: Changes the forward direction of the Avatar.
  - `RotateToObject(Vector3 targetPosition, float speed)`: Rotates the Avatar to face a target position while moving towards it.
  - `MoveToObject(Vector3 targetPosition, float speed)`: Moves the Avatar towards a target position.
  - `ActualizeDrives(Tuple<bool, float, float> tuple, float temperature, float glare, float sound)`: Updates the Avatar's drives based on given inputs.
  - `OnTriggerEnter(Collider other)`: Updates the senses by adding the Collider.
  - `OnTriggerExit(Collider other)`: Updates the senses by removing the Collider.

AvatarSenses

- Purpose: Represents the senses of the Avatar, including sight, touch, and hearing.
- Properties:
  - `List<IWorldObject> SightList { get; set; }`: List of objects currently in the Avatar's sight.
  - `List<ITemperatureEmitter> TouchList { get; set; }`: List of temperature emitters currently being touched by the Avatar.
  - `List<ISoundEmitter> HearingList { get; set; }`: List of sound emitters currently being heard by the Avatar.

    - `float Glare { get; }`: Amount of glare currently felt by the Avatar.
- Methods:
    - `Start()`: Initializes the obstacle layer mask.
    - `UseSightSense(Transform avatarTransform)`: Uses the sight sense to detect objects in front of the Avatar.
    - `CalculateTemperature(float ambientTemp, Vector3 currentPosition)`: Calculates the temperature felt by the Avatar based on nearby temperature emitters.
    - `CalculateSound(Vector3 currentPosition)`: Calculates the sound intensity felt by the Avatar based on nearby sound emitters.
    - `ActualizeSensesListAdding(Collider other)`: Updates the senses list by adding a new collider.
    - `ActualizeSensesListRemoving(Collider other)`: Updates the senses list by removing a collider.
    - `ConvertToRadians(double angle)`: Converts an angle from degrees to radians.
    - `getListNotViewed()`: Returns the list of objects that were not viewed by the Avatar.

IAvatarSenses

- Purpose: Interface for defining the senses of an Avatar, including sight, touch, and hearing.
- Properties:
    - `List<IWorldObject> SightList { get; set; }`: List of objects currently in the Avatar's sight.
    - `List<ITemperatureEmitter> TouchList { get; set; }`: List of temperature emitters currently being touched by the Avatar.
    - `List<ISoundEmitter> HearingList { get; set; }`: List of sound emitters currently being heard by the Avatar.
    - `float Glare { get; }`: Amount of glare currently felt by the Avatar.
- Methods:
    - `ActualizeSensesListAdding(Collider other)`: Updates the senses list by adding a new collider.
    - `ActualizeSensesListRemoving(Collider other)`: Updates the senses list by removing a collider.
    - `UseSightSense(Transform avatarTransform)`: Uses the sight sense to detect objects in front of the Avatar.

        - `CalculateTemperature(float ambientTemp, Vector3 currentPosition)`: Calculates the temperature felt by the Avatar based on nearby temperature emitters.
        - `CalculateSound(Vector3 currentPosition)`: Calculates the sound intensity felt by the Avatar based on nearby sound emitters.

IAvatarDrives

- Purpose: Interface for defining the drives (motivations or needs) of an Avatar, such as health and hunger.
- Properties:
    - `float Health { get; }`: The current health level of the Avatar.
    - `float Hunger { get; }`: The current hunger level of the Avatar.
- Methods:
    - `ActualizeDrives(Tuple<bool, float, float> tuple, float temperature, float glare, float sound)`: Updates the Avatar's drives based on the given inputs.
    - `CheckRestart()`: Checks whether the Avatar needs to restart, based on its current drives.

AvatarDrives

- Purpose: Represents the drives (motivations or needs) of an Avatar, such as health and hunger. Implements the IAvatarDrives interface.
- Properties:
    - `float Health { get; }`: The current health level of the Avatar.
    - `float Hunger { get; }`: The current hunger level of the Avatar.
- Methods:
    - `ActualizeDrives(Tuple<bool, float, float> tuple, float temperature, float glare, float sound)`: Updates the Avatar's drives based on the given inputs.
    - `CheckRestart()`: Checks whether the Avatar needs to restart, based on its current drives.
    - `TempCurve(float tempValue)`: Calculates the health loss due to temperature.
    - `GlareCurve(float glareValue)`: Calculates the health loss due to glare.
    - `SoundCurve(float soundValue)`: Calculates the health loss due to sound.

AvatarManualMovement

- Purpose: Handles the manual movement and rotation of the Avatar, including interactions such as eating.
- Properties:
  - `bool isMoving`: Indicates whether the Avatar is currently moving.
  - `AvatarAnimationController animationController`: Reference to the Avatar's animation controller.
  - `IAvatarSenses avatarSenses`: Reference to the Avatar's senses.
- Methods:
  - `Initialize(IAvatarSenses avatarSenses, AvatarAnimationController animationController)`: Initializes the manual movement by setting up input detection and references.
  - `MoveRight()`: Moves the Avatar to the right.
  - `MoveDown()`: Moves the Avatar downwards.
  - `MoveLeft()`: Moves the Avatar to the left.
  - `MoveUp()`: Moves the Avatar upwards.
  - `RotateUp()`: Rotates the Avatar to face upwards.
  - `RotateDown()`: Rotates the Avatar to face downwards.
  - `RotateLeft()`: Rotates the Avatar to face left.
  - `RotateRight()`: Rotates the Avatar to face right.
  - `Eat()`: Makes the Avatar attempt to eat an object in front of it.
  - `ExecuteMovement(Vector3 direction)`: Executes movement in the specified direction if the path is not blocked.
  - `RotateTowardsTarget(Vector3 direction)`: Rotates the Avatar towards the target direction.
  - `IsFacingTarget(Quaternion targetRotation)`: Checks if the Avatar is facing the target direction.
  - `MoveAndRotateTowardsTarget(Vector3 direction)`: Moves and rotates the Avatar towards the target position.
  - `PathIsBlocked(Vector3 offset)`: Checks if the path in the specified direction is blocked.
  - `ChangeForward(Vector3 forward)`: Changes the forward direction of the Avatar.
  - `RotateToObject(Vector3 targetPosition, float speed)`: Rotates the Avatar to face the target position while moving.
  - `MoveToObject(Vector3 targetPosition, float speed)`: Moves the Avatar towards the target position.

AvatarAnimationController

- Purpose: Controls the animations of the Avatar.
- Properties:
  - `Animator anim`: Reference to the Animator component.
- Methods:
  - `Start()`: Initializes the Animator component.
  - `PlayWholeIdleAnimation()`: Plays the idle animation by gradually setting the Speed parameter to 0.
  - `PlayWholeDismissAnimation()`: Plays the dismiss animation by gradually setting the Speed parameter to 2.
  - `PlayWholePickingUpAnimation()`: Plays the picking up animation by gradually setting the Speed parameter to 1.
  - `SetWalkingAnimation()`: Sets the walking animation by setting the Speed parameter to 0.5.

EXTRA CLASSES

InputsDetector

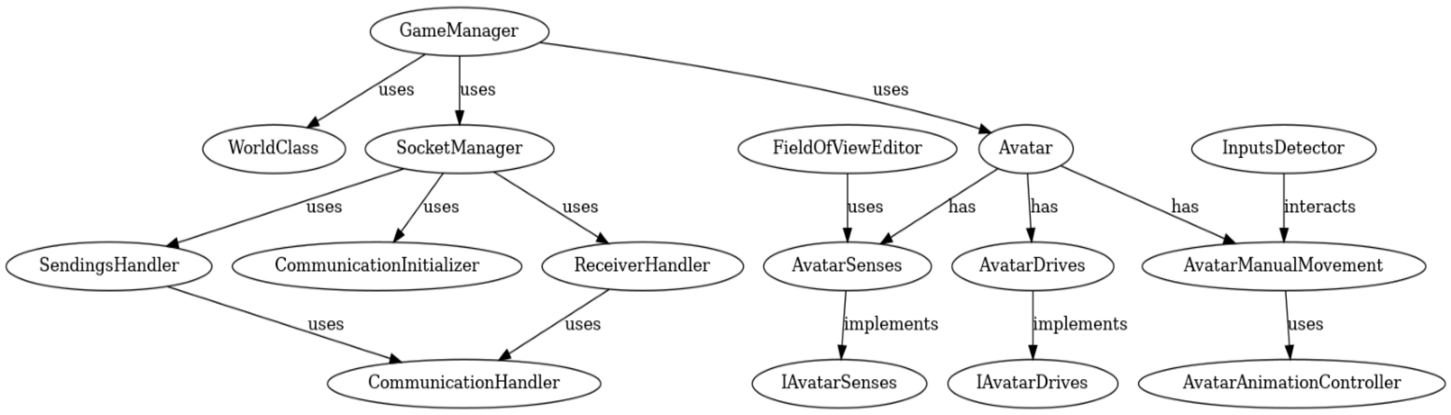
- Purpose: Singleton class to detect user input and invoke corresponding actions.
- Properties:
  - `static InputsDetector Instance { get; private set; }`: Singleton instance.
  - `Action LHasBeenPressed`: Action to be invoked when 'L' key is pressed.
  - `Action JHasBeenPressed`: Action to be invoked when 'J' key is pressed.
  - `Action KHasBeenPressed`: Action to be invoked when 'K' key is pressed.
  - `Action IHasBeenPressed`: Action to be invoked when 'I' key is pressed.
  - `Action WHasBeenPressed`: Action to be invoked when 'W' key is pressed.
  - `Action AHasBeenPressed`: Action to be invoked when 'A' key is pressed.
  - `Action SHasBeenPressed`: Action to be invoked when 'S' key is pressed.
  - `Action DHasBeenPressed`: Action to be invoked when 'D' key is pressed.

    - `Action EHasBeenPressed`: Action to be invoked when 'E' key is pressed.
    - `Action SpaceHasBeenPressed`: Action to be invoked when 'Space' key is pressed.
- Methods:
    - `Awake()`: Ensures that there is only one instance of InputsDetector.
    - `Update()`: Checks for specific key presses and invokes corresponding actions if assigned.

FieldOfViewEditor

- Purpose: Custom editor for the `AvatarSenses` component to visualize the field of view in the scene view.
- Methods:
    - `OnSceneGUI()`: Called when the scene view is being drawn. This method is used to draw custom gizmos in the scene view.
    - `DirectionFromAngle(float eulerY, float angleInDegrees)`: Calculates a direction vector from an angle.

CLASS DIAGRAM:


GameManager
uses
uses
uses
WorldClass
SocketManager
FieldOfViewEditor
Avatar
InputsDetector
uses
uses
uses
uses
has
has
has
interacts
SendingsHandler
CommunicationInitializer
ReceiverHandler
AvatarSenses
AvatarDrives
AvatarManualMovement
uses
uses
implements
implements
uses
CommunicationHandler
IAvatarSenses
IAvatarDrives
AvatarAnimationController