Chapter 1

Introduction

This document provides a comprehensive overview of the Zappy AI's functionality, structure, and behavior within the game environment. The AI is designed to simulate intelligent player behavior, enabling autonomous decision-making and interaction with the game world.

1.1 Binary Usage

./zappy_ai -p port -n name -h machine

```
option description

-p port port number
-n name name of the team
-h machine hostname of the server (localhost by default)
```

Chapter 2

Argument Parsing

2.1 Overview

The ParseArgs class manages the validation and processing of command-line arguments for configuring the Zappy AI. It ensures that all required arguments are provided, correctly formatted, and within valid ranges.

2.2 Class Structure

The ParseArgs class includes methods for handling different aspects of argument validation and processing:

- __init__ Method: Initializes default values (localhost, -1, '', -1) for host, port, name, and ID.
- check_type Method: Validates argument types and values. Exits with an error code if validation fails.
- check_host Method: Verifies the validity of the provided host using socket.gethostbyname(). Exits with an error if the host is invalid.
- check_port Method: Ensures the provided port number is within the valid range of 0 to 65535. Exits with an error if out of range.
- check_name Method: Checks that the provided team name does not exceed 32 characters in length. Exits with an error if the name is too long.
- check_id Method: Validates that the provided ID is a non-negative integer. Exits with an error if the ID is invalid.
- parse Method: Processes command-line arguments (-p, -n, -h, -id) to set instance variables (port, name, host, id). Calls check_type() to validate inputs. Returns validated arguments.
- print_usage Method: Prints usage instructions for running the program with the correct argument format. Exits with an error after printing.
- print_invalid_argument Method: Handles errors for invalid argument formats. Prints an error message and exits with an error code.

2.3. check_host Method

The check_host method verifies the validity of the provided host using the socket.gethostbyname() function.

```
def check_host(self):
    try:
        socket.gethostbyname(self.host)
    except socket.gaierror:
        print("Error: Invalid host")
        sys.exit(84)
```

2.4. check_port Method

The check_port method validates whether the provided port number (self.port) falls within the acceptable range of 0 to 65535, inclusive.

```
def check_port(self):
    if self.port < 0 or self.port > 65535:
        print("Error: Port out of range")
        sys.exit(84)
```

2.5. check_name Method

The check_name method verifies that the provided team name (self.name) does not exceed 32 characters in length.

```
def check_name(self):
    if len(self.name) > 32:
        print("Error: Name too long")
        sys.exit(84)
```

2.6. check_id Method

The check_id method validates that the provided ID (self.id) is a non-negative integer.

```
def check_id(self):
    if self.id < 0:
        print("Error: ID must be a non-negative integer")
        sys.exit(84)
```

2.7. Conclusion

The MainClient class provides essential functionality for managing client-side interactions with the Zappy game server. By establishing connections, sending/receiving data, and managing player slots, it supports the seamless integration of AI-driven players into the game environment.

Chapter 3

MainClient

The MainClient class manages the client-side communication with the Zappy game server. It facilitates connection establishment, data transmission, and interaction with the server to manage player slots effectively.

3.1. Class Overview

The MainClient class is designed to handle the following functionalities:

- Initialization (__init__): Establishes a connection to the server using the provided host, port, and name.
- Sending Data (send): Sends encoded data to the server.
- Receiving Data (receive): Receives and decodes data from the server, handling disconnection scenarios.
- Connecting and Getting Slots (connect_and_get_slots): Connects to the server, sends the team name (name), retrieves welcome and slot availability messages, and returns the number of available slots.
- Closing Connection (close): Closes the socket connection with the server.

3.2. __init__ Method

The __init__ method initializes the MainClient object with the specified host, port, and name, establishing a TCP connection to the server using socket.socket.

```
def __init__(self, host, port, name):
    self.host = host
    self.port = port
    self.name = name
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.socket.connect((self.host, self.port))
```

3.3. send Method

The send method encodes and sends data to the server using the established socket connection.

```
def send(self, data):
    self.socket.send(data.encode())
```

3.4. receive Method

The receive method retrieves and decodes data received from the server, handling cases where the server closes the connection.

```
def receive(self):
    data = self.socket.recv(1024).decode().strip()
    if not data:
        print("Server closed the connection")
        sys.exit(84)
    return data
```

3.5. connect_and_get_slots Method

The connect_and_get_slots method orchestrates the initial interaction with the server, sending the team name (self.name) and retrieving information about available player slots.

```
def connect_and_get_slots(self):
    self.send(self.name + '\n')
    welcome_msg = self.receive()
    available_slots_msg = self.receive()
    available_slots = int(available_slots_msg.split('\n')[0].strip())
    return available_slots + 1
```

3.6. close Method

The close method terminates the socket connection with the server, releasing network resources.

```
def close(self):
    self.socket.close()
```

3.7. Conclusion

The MainClient class provides essential functionality for managing client-side interactions with the Zappy game server. By establishing connections, sending/receiving data, and managing player slots, it supports the seamless integration of AI-driven players into the game environment.

Chapter 4

Client

The Client class manages the network communication and AI behavior for interacting with the Zappy game server. It establishes a TCP connection, sends and receives data, handles server responses, and executes AI-driven actions to participate in the game environment effectively.

4.1. Class Overview

The Client class is designed to handle the following functionalities:

- Initialization (__init__): Establishes a TCP connection to the server using the provided host, port, team name, and client ID. Initializes necessary attributes such as the socket, selector for asynchronous I/O operations, and AI instance.
- Sending Data (send): Sends encoded data to the server through the established socket connection.
- Receiving Data (receive): Receives and processes data from the server, handling disconnection scenarios and managing incoming messages.
- Sending Queue (send_queue): Sends a sequence of queued commands to the server, updating the AI state based on received responses.
- Sending Broadcast (send_broadcast): Sends a broadcast message to the server, triggering specific actions based on the message content.
- Revive Client (revive_client): Handles client reconnection to the server in case of disconnection, resetting client state and re-establishing the socket connection.
- Handling Broadcast (handle_broadcast): Processes incoming broadcast messages from the server, updating AI states and triggering appropriate actions based on the message content.
- Closing Connection (close): Terminates the socket connection and releases associated resources.
- Connection Initialization (connect): Establishes the initial connection with the server, sending the team name and handling initial server responses.
- Main Execution Loop (run): Manages the main execution loop of the client, determining AI actions based on the current mode and game conditions.

4.2. __init__ Method

The __init__ method initializes the Client object with the specified host, port, team name, and client ID, establishing a TCP connection to the Zappy game server using socket.socket.

```
def __init__(self, host, port, name, id):
    self.host = host
    self.port = int(port)
    self.name = name
    self.id = id
    self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    self.socket.connect((self.host, self.port))
    self.queue = []

    self.selector = selectors.DefaultSelector()
    self.selector.register(self.socket, selectors.EVENT_READ, self.receive)

    self.ai = AI(self.id)
```

4.3. send Method

The send method encodes and sends data to the server using the established socket connection.

```
def send(self, data):
    self.socket.send(data.encode() + b'\n')
```

4.4. receive Method

The receive method retrieves and processes data received from the server, handling disconnection scenarios and managing incoming messages.

```
def receive(self):
    buffer = ""
    while True:
        data = self.socket.recv(512).decode().strip()

        if not data:
            print("Server closed the connection")
            sys.exit(0)

        buffer += data
        while '\n' in buffer:
            message, buffer = buffer.split('\n', 1)
            if message.startswith('message'):
                self.handle_broadcast(message)
            else:
                self.ai.update_state(message)

        if '\n' not in buffer:
            break

    count = 0
    while buffer.startswith('message') and count < 3:
        self.handle_broadcast(buffer)
        buffer = self.socket.recv(512).decode().strip()
        count += 1

    return buffer
```

4.5. connect Method

The connect method initializes the connection with the server, sending the team name and handling initial server responses.

```
def connect(self):
    data = self.receive()  # Initial connection response (welcome message)
    self.send(self.name)
    data = self.receive()  # Additional server response (map size)
    if data == 'ko':
        sys.exit(84)
```

4.6. run Method

The run method manages the main execution loop of the client, determining AI actions based on the current mode and game conditions. (To be continued with the implementation of the run method and the AI class in the next chapter.)

Chapter 5

AI

The AI class is responsible for managing the behavior and decision-making of the Zappy AI within the game environment. It contains various methods that handle different aspects of the AI's functionality, such as resource management, movement, and incantation.

5.1. map_resource Method

The map_resource method takes an integer ID as an argument and returns the corresponding resource name as a string.

```
resources = {
    0: 'sibur',
    1: 'phiras',
    2: 'deraumere',
    3: 'mendiane',
    4: 'linemate',
    5: 'thystame'
}

def map_resource(self, id):
    return resources.get(id, 'Unknown resource')
```

5.2. has_all_resources_dict_func Method

The has_all_resources_dict_func method checks if the AI has collected all required resources for a specific level.

```
def has_all_resources_dict_func(self):
    for resource, value in self.has_all_resources_dict.items():
        if value == False:
            return False
    return True
```

5.3. parse_inventory Method

The parse_inventory method processes the inventory response received from the server and updates the AI's internal inventory representation.

```
def parse_inventory(self, response):
    if self.detect_type_of_response(response) != 'inventory':
        return None
    inventory = {}
    response = response.strip('[]')
    list_of_items = response.split(', ')
    for item in list_of_items:
        parts = item.split()
        if len(parts) == 2:
            key, value = parts
            inventory[key] = int(value)
        else:
            print(f"Unexpected inventory item format: {item}")
    return inventory
```

5.4. parse_vision Method

The parse_vision method processes the vision response received from the server and updates the AI's internal representation of the game world.

```
def parse_vision(self, input_message):
    vision = self._parse_input_list(input_message.strip('[]'))
    list_of_dicts = self._create_list_of_dicts(vision)
    self._fill_empty_elements(list_of_dicts)
    padded_list = self._pad_rows(list_of_dicts)
    final_list = self._convert_elements_to_dicts(padded_list)
    return final_list
```

5.5. get_type_inventory_look Method

The get_type_inventory_look method determines whether a given response is an inventory or look response.

```
def get_type_inventory_look(self, response):
    try:
        inventory_string = response.strip("[] ")
        items = inventory_string.split(", ")
        inventory_dict = {}

        for item in items:
            key, value = item.split(" ")
            inventory_dict[key] = int(value)
        return 'inventory'
    except:
        return 'look'
```

5.6. detect_type_of_response Method

The detect_type_of_response method identifies the type of a given response from the server.

```
def detect_type_of_response(self, response):
    if response.startswith('ok'):
        return 'ok'
    if response.startswith('ko'):
        return 'ko'
    if response.startswith('['):
        return self.get_type_inventory_look(response)
    if response.startswith('message'):
        return 'broadcast'
    if response.startswith('dead'):
        return 'dead'
    if response.startswith('Elevation underway'):
        return 'elevation'
    if response.startswith('Current level:'):
        return 'level up'

    return 'unknown'
```

5.7. update_state Method

The update_state method processes server responses and updates the AI's internal state accordingly.

```
def update_state(self, response):
    type_of_response = self.detect_type_of_response(response)
    if type_of_response == 'inventory':
        self.inventory = self.parse_inventory(response)
    if type_of_response == 'look':
        self.vision = self.parse_vision(response)
    if type_of_response == 'level up':
        self.level = int(response.split()[-1])
        print(f"Level up! New level: {self.level}")
```

5.8. set_direction Method

The set_direction method updates the AI's current direction based on the desired goal direction.

```
def set_direction(self, goal_direction):
    if self.direction == goal_direction:
        return

    idx_directions = ['Left', 'Up', 'Right', 'Down']
    idx_goal = idx_directions.index(goal_direction)
    idx_current = idx_directions.index(self.direction)
    diff = idx_goal - idx_current
    if diff == 1 or diff == -3:
        self.queue.append('Left')
    elif diff == 2 or diff == -2:
        self.queue.append('Right')
        self.queue.append('Right')
    elif diff == 3 or diff == -1:
        self.queue.append('Right')
    self.direction = goal_direction
```

5.9. move_direction Method

The move_direction method updates the AI's current position and direction based on the desired goal direction.

```
def move_direction(self, goal_direction):
    self.set_direction(goal_direction)
    self.queue.append('Forward')
```

5.10. move_to Method

The move_to method moves the AI to a specific (x, y) coordinate within the game world.

```
def move_to(self, goal_x, goal_y):

    vertical_direction = 'Down' if goal_y > self.y else 'Up'
    horizontal_direction = 'Right' if goal_x > self.x else 'Left'

    while self.y != goal_y:
        self.move_direction(vertical_direction)
        if vertical_direction == 'Down':
            self.y += 1
        else:
            self.y -= 1

    while self.x != goal_x:
        self.move_direction(horizontal_direction)
        if horizontal_direction == 'Right':
            self.x += 1
        else:
            self.x -= 1
```

5.11. take_resources Method

The take_resources method attempts to collect a specific resource from the game world.

```
def take_resources(self, resource):
    state = False
    for i, row in enumerate(self.vision):
        for j, cell in enumerate(row):
            try:
                if cell[resource] > 0:
                    self.move_to(j, i)
                    for _ in range(cell[resource]):
                        self.queue.append(f'Take {resource}')
                    state = True
            except:
                print(f"{resource} doesn't exist")
    return state
```

5.12. has_all_resources Method

The has_all_resources method checks if the AI has collected all required resources for a specific level.

```
def has_all_resources(self):
    for resource, amount in self.inventory.items():
        if resource == 'food':
            continue
        if amount < self.resources_to_get[resource]:
            return False
    return True
```

5.13. take_resources_to_get Method

The take_resources_to_get method attempts to collect all required resources for a specific level.

```
def take_resources_to_get(self):
    state = False

    goal_inventory = self.resources_to_get
    # goal_inventory['food'] = 40
    resources_to_collect = {k: v for k, v in goal_inventory.items() if v > self.inventory[k]}

    for resource, amount_needed in resources_to_collect.items():
        amount_to_collect = amount_needed - self.inventory[resource]
        if amount_to_collect <= 0:
            continue

        for i, row in enumerate(self.vision):
            for j, cell in enumerate(row):
                if cell.get(resource, 0) > 0:
                    self.move_to(j, i)
                    for _ in range(min(amount_to_collect, cell[resource])):
                        self.queue.append(f'Take {resource}')
                        amount_to_collect -= 1
                    if amount_to_collect <= 0:
                        state = True
                        break
            if amount_to_collect <= 0:
                break
    return state
```

5.14. reset_direction Method

The reset_direction method resets the AI's current direction to the default 'Down' direction.

```
def reset_direction(self):
    self.direction = 'Down'
```

5.15. linemate_level1 Method

The linemate_level1 method attempts to perform the incantation for level 1 using linemate resources.

```
def linemate_level1(self):
    state = False
    for i, row in enumerate(self.vision):
        for j, cell in enumerate(row):
            try:
                if cell['linemate'] > 0:
                    self.move_to(j, i)
                    self.queue.append('Incantation')
                    state = True
                    return state
            except:
                print(f"linemate doesn't exist")
    return state
```

5.16. drop_resources Method

The drop_resources method drops all collected resources from the AI's inventory.

```
def drop_resources(self):
    for resource, amount in self.inventory.items():
        if resource == 'food':
            continue
        for _ in range(amount):
            self.queue.append(f'Set {resource}')
    self.inventory = inventory
    return self.queue
```