In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import os
import torch
import torch.nn as nn
import torch.optim as optim
import random
import sys
from collections import deque
from matplotlib import cm
from torch.cuda.amp import autocast, GradScaler
from qutip import Bloch, Qobj
```

In [2]:
```
# Hyperparameters
HIDDEN_FEATURES = 64
LEARNING_RATE = 1e-4
GAMMA = 0.95
EPSILON = 1.0
EPSILON_DECAY = 0.995
MIN_EPSILON = 0.001
MEMORY_SIZE = 1000
BATCH_SIZE = 128
TARGET_UPDATE = 10
EPISODES = 10000
MAX_STEPS = 10
FIDELITY_THRESHOLD = 1e-5
PATIENCE = 5000
HADAMARD = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
T_GATE = np.array([[1, 0], [0, np.exp(1j * np.pi / 4)]], dtype=np.complex128
CNOT = np.array([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]])
```

Quantum State Transfer

- **NUM_QUBITS:** This parameter specifies the number of qubits in the quantum system. Each qubit can represent two states, |0> and |1>, allowing the system to exist in a superposition of these states.
- **STATE_SIZE:** Calculated as 2^{QUBITS}, this defines the dimension of the state vector in the Hilbert space for the quantum system. It represents the total number of basis states the system can have. For instance, with 1 qubit, STATE_SIZE is 2, corresponding to the two possible states |0> and |1>.
- **ACTION_SIZE:** The number of possible actions the agent can take. In this case, it represents three possible rotations on the Bloch sphere: Rx, Ry, and Rz, which correspond to rotations around the x, y, and z axes, respectively. They are achieved by a general unitary operator obtained by combining amplitud, phase, and duration parameters.

Neural Network Features

- **INPUT_FEATURES:** Calculated as $2^{(QUBITS+1)}$, this determines the input size for the neural network. It accounts for both the real and imaginary parts of the quantum state. For 1 qubit, INPUT_FEATURES is 4, reflecting that both the real and imaginary components of the two basis states are considered.
- **HIDDEN_FEATURES:** The number of neurons in the hidden layer of the neural network. This parameter defines the capacity of the network to learn complex patterns.

Training Hyperparameters

- **LEARNING_RATE:** The rate at which the model's weights are updated during training. A smaller learning rate can lead to more precise convergence, while a larger rate can speed up training but may miss the optimal solution.
- **GAMMA:** The discount factor in the Q-learning algorithm. It determines the importance of future rewards versus immediate rewards. A value close to 1 encourages long-term gains.
- **EPSILON:** The initial exploration rate for the epsilon-greedy policy, which defines the probability of choosing a random action versus the best-known action. It starts at 1.0, meaning all actions are chosen randomly initially.
- **EPSILON_DECAY:** The rate at which EPSILON decreases over time. This allows the agent to gradually shift from exploration to exploitation.
- **MIN_EPSILON:** The minimum value that EPSILON can reach, ensuring that the agent continues to explore to some degree even in later stages of training.
- **MEMORY_SIZE:** The size of the replay memory, which stores past experiences for training the neural network. This helps in breaking correlations between consecutive experiences and stabilizes learning.
- **BATCH_SIZE:** The number of experiences sampled from memory to update the model at each step. This affects the stability and speed of learning.
- **TARGET_UPDATE:** The frequency (in episodes) at which the target network's weights are updated to match the current model's weights. This helps stabilize learning by providing a consistent target.
- **EPISODES:** The total number of episodes the agent will be trained for. Indicates the training process's duration.
- **MAX_STEPS:** The maximum number of steps the agent can take in a single episode.

- **FIDELITY_THRESHOLD:** A threshold for the fidelity of the quantum state, used as a criterion for early stopping. When the fidelity between the target and final states exceeds this value, training can stop early.
- **PATIENCE:** The number of episodes to wait without improvement in fidelity before triggering early stopping. This prevents premature stopping and ensures adequate exploration.

GPU Management

In [3]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [4]:
```
!set 'PYTORCH_CUDA_ALLOC_CONF=max_split_size_mb:2'
os.environ["CUDA_LAUNCH_BLOCKING"] = "1"
```

In [5]:
```
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
torch.backends.cudnn.enabled = True
torch.backends.cudnn.benchmark = False
torch.cuda.empty_cache()
```

In [6]:
```
print("__Python VERSION:", sys.version)
print("__pyTorch VERSION:", torch.__version__)
```

```
__Python VERSION: 3.11.9 (main, Jun 13 2024, 16:50:48) [GCC 11.4.0]
__pyTorch VERSION: 2.3.1+cu121
```

In [7]:
```
print("__CUDA VERSION")
!nvidia-smi
```

```
__CUDA VERSION
Wed Aug  7 15:09:52 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.90.07              Driver Version: 550.90.07      CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce GTX 1650 Ti     Off |   00000000:01:00.0  On |                  N/A |
| N/A   67C    P5              8W /   50W |      96MiB /   4096MiB |     30%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A      9689      G   /usr/lib/xorg/Xorg                             91MiB |
+-----------------------------------------------------------------------------------------+
```

In [8]:
```python
print("__CUDNN VERSION:", torch.backends.cudnn.version())
!nvcc --version
```

```
__CUDNN VERSION: 8902
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2024 NVIDIA Corporation
Built on Thu_Mar_28_02:18:24_PDT_2024
Cuda compilation tools, release 12.4, V12.4.131
Build cuda_12.4.r12.4/compiler.34097967_0
```

In [9]:
```python
print("__Number CUDA Devices:", torch.cuda.device_count())
print("__Devices")
print("Available devices ", torch.cuda.device_count())
print("Active CUDA Device: GPU", torch.cuda.current_device())
```

```
__Number CUDA Devices: 1
__Devices
Available devices  1
Active CUDA Device: GPU 0
```

In [10]:

```python
class QuantumGateEnv:
    """
    A class to represent the environment for quantum gate control using reir

    Attributes:
    ----------
    gate : str
        The type of quantum gate (e.g., 'H', 'T', 'CNOT').
    control_pulse_params : dict
        Dictionary containing amplitude, phase, and duration parameters for
    initial_state : np.ndarray
        The initial quantum state of the system.
    state : np.ndarray
        The current quantum state of the system.
    target : np.ndarray
        The target quantum gate matrix.
    theoretical_state : np.ndarray
        The expected quantum state after applying the target gate.
    time_step : int
        The current time step in the episode.
    max_steps : int
        The maximum number of steps in an episode.
    state_history : list
        A history of quantum states during an episode.
    """
    def __init__(self, gate):
        """
        Initializes the QuantumGateEnv with the specified gate type.

        Parameters:
        ----------
        gate : str
            The type of quantum gate (e.g., 'H', 'T', 'CNOT').
        """
        self.control_pulse_params = {
            'amplitude': np.linspace(0, 1, 12),  # Example amplitudes
            'phase': np.linspace(-np.pi, np.pi, 12),  # Example phases
            'duration': np.linspace(0.1, 1.0, 120)  # Example durations
        }
        self.gate = gate
        self.initial_state, self.target_state,self.state_size, self.action_s
        self.reset()

    def set_quantum_info(self):
        """
        Sets the quantum information for the environment based on the gate t

        Returns:
        -------
        tuple
            A tuple containing the initial state, target_state, state size,
```

```
        """
        action_size = len(self.control_pulse_params['amplitude']) * len(self

        if self.gate in ["H", "T"]:
            num_qubits = 1
            state_size = 2**num_qubits
            initial_states = [np.array([1, 0], dtype=np.complex128), np.arra
            initial_state = random.choice(initial_states)
            input_features = 2 ** (
                num_qubits + 1
            )  # Number of states in the input space
            unitary = HADAMARD if self.gate == "H" else T_GATE
            target_state = np.dot(initial_state, unitary)

        elif self.gate == 'CNOT':
            num_qubits = 2
            state_size = 2**num_qubits
            initial_states = [
            np.array([1, 0, 0, 0], dtype=np.complex128),
            np.array([0, 1, 0, 0], dtype=np.complex128),
            np.array([0, 0, 1, 0], dtype=np.complex128),
            np.array([0, 0, 0, 1], dtype=np.complex128),
            ]
            initial_state = random.choice(initial_states)
            input_features = 2 ** (
                num_qubits + 1
            )  # Number of states in the input space
            unitary = CNOT
            target_state = np.dot(initial_state, unitary)

        return  initial_state, target_state, state_size, action_size, input_

    def reset(self):
        """
        Resets the environment to the initial state.

        Returns:
        -------
        np.ndarray
            The initial quantum state.
        """
        self.state = self.initial_state.copy()
        self.time_step = 0
        self.max_steps = 10  # Simplified time horizon
        #self.state_history = []
        return self.state

    def step(self, action):
        """
        Takes a step in the environment using the given action.

        Parameters:
        ----------
        action : int
            The action to be taken by the agent.
```

```
        Returns:
        -------
        tuple
            A tuple containing the next state, reward, and done flag.
        """
        self.time_step += 1

        # Decode action into control pulse parameters
        amplitude_index = action // (
            len(self.control_pulse_params["phase"])
            * len(self.control_pulse_params["duration"])
        )
        phase_index = (action // len(self.control_pulse_params["duration"]))
            self.control_pulse_params["phase"]
        )
        duration_index = action % len(self.control_pulse_params["duration"])

        # Set control pulse parameters
        amplitude = self.control_pulse_params["amplitude"][amplitude_index]
        phase = self.control_pulse_params["phase"][phase_index]
        duration = self.control_pulse_params["duration"][duration_index]

        # Apply control pulse
        control_matrix = self._apply_control_pulse(amplitude, phase, duratic
        next_state = np.dot(control_matrix, self.state)
        self.state = next_state

        # Calculate reward
        reward = -self.infidelity(next_state) if self.time_step == self.max_
        done = self.time_step == self.max_steps

        # Store state history
        # self.state_history.append(self.state)
        # self.amplitude_history.append(amplitude)
        # self.phase_history.append(phase)
        # self.duration_history.append(duration)

        return next_state, reward, done, amplitude, phase, duration

    def _apply_control_pulse(self, amplitude, phase, duration):
        """
        Applies a control pulse to the quantum gate.

        Parameters:
        ----------
        amplitude (float): The amplitude of the control pulse.
        phase (float): The phase shift of the control pulse.
        duration (float): The duration of the control pulse.

        Returns:
        -------
        np.ndarray
            The resulting unitary matrix after applying the control pulse.
        """
        # Calculate the rotation angle
        theta = amplitude * np.pi * duration
```

```
        # Apply control pulse for single-qubit gates
        if self.gate in ["H", "T"]:
            return np.array([
                        [np.cos(theta / 2), -1j * np.sin(theta / 2) * np.exp
                        [-1j * np.sin(theta / 2) * np.exp(-1j * phase), np.c
                    ])
        elif self.gate == "CNOT":
            # Define the rotation matrix on the target qubit
            R = np.array([
                    [np.cos(theta / 2), -1j * np.sin(theta / 2) * np.exp(1j
                    [-1j * np.sin(theta / 2) * np.exp(-1j * phase), np.cos(t
            ])
            # Define the identity matrix for the control qubit
            I = np.eye(2)

            # Create the full matrix by combining the control and target ope
            # Note: This assumes the control is the first qubit and target i
            return np.block([
                [I, np.zeros_like(I)],
                [np.zeros_like(I), R]
            ])
        else:
            raise ValueError("Unsupported gate type")

    def infidelity(self, final_state):
        """
        Calculates the infidelity between the final state and the theoretica

        Parameters:
        ----------
        final_state (np.ndarray): The final quantum state after applying the

        Returns:
        -------
        float
            The infidelity between the final state and the theoretical state
        """
        fidelity = np.abs(np.dot(np.conjugate(self.target_state), final_stat
        return 1 - fidelity
```

In [11]:
```
# Dueling Double Deep Q-Network
class DDDQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DDDQN, self).__init__()
        self.fc1 = nn.Linear(env.input_features, HIDDEN_FEATURES)
        self.fc2 = nn.Linear(HIDDEN_FEATURES, HIDDEN_FEATURES)
        self.value_fc = nn.Linear(HIDDEN_FEATURES, 1)
        self.advantage_fc = nn.Linear(HIDDEN_FEATURES, env.action_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        value = self.value_fc(x)
        advantage = self.advantage_fc(x)
```

```
        q_vals = value + (advantage - advantage.mean(dim=1, keepdim=True))
        return q_vals
```

In [12]:
```
# Replay Memory
class ReplayMemory:
    """
    A class to represent replay memory for storing experiences during reinfo

    Attributes:
    ----------
    memory : deque
        A deque to store the experiences with a fixed maximum length.
    """

    def __init__(self, capacity):
        """
        Initializes the replay memory with a specified capacity.

        Parameters:
        ----------
        capacity : int
            The maximum number of experiences the memory can hold.
        """
        self.memory = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        """
        Adds an experience to the replay memory.

        Parameters:
        ----------
        state : np.ndarray
            The state observed before taking the action.
        action : int
            The action taken by the agent.
        reward : float
            The reward received after taking the action.
        next_state : np.ndarray
            The state observed after taking the action.
        done : bool
            Whether the episode has ended.
        """
        self.memory.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        """
        Samples a batch of experiences from the replay memory.

        Parameters:
        ----------
        batch_size : int
            The number of experiences to sample.

        Returns:
        -------
        list
```

```
            A list of sampled experiences.
        """
        return random.sample(self.memory, batch_size)

    def __len__(self):
        """
        Returns the current size of the replay memory.

        Returns:
        -------
        int
            The number of experiences currently stored in the replay memory.
        """
        return len(self.memory)
```

In [13]:
```
# DDDQN Agent
class DDDQNAgent:
    """
    A Dueling Double Deep Q-Network (DDDQN) agent for reinforcement learning

    Attributes:
    ----------
    state_size : int
        The size of the state space.
    action_size : int
        The size of the action space.
    epsilon : float
        The exploration rate for the epsilon-greedy policy.
    memory : ReplayMemory
        The replay memory to store experiences.
    model : DDDQN
        The Q-network model for learning the Q-values.
    target_model : DDDQN
        The target Q-network model for stable learning.
    optimizer : torch.optim.Adam
        The optimizer for training the model.
    scaler : torch.cuda.amp.GradScaler
        The gradient scaler for mixed precision training.
    loss : torch.nn.MSELoss
        The loss function for training the model.
    """

    def __init__(self, state_size, action_size):
        """
        Initializes the DDDQNAgent with the given state and action sizes.

        Parameters:
        ----------
        state_size : int
            The size of the state space.
        action_size : int
            The size of the action space.
        """
        self.state_size = state_size
        self.action_size = action_size
        self.epsilon = EPSILON
```

```python
        self.memory = ReplayMemory(MEMORY_SIZE)
        self.model = DDDQN(state_size, action_size).to(device)
        self.target_model = DDDQN(state_size, action_size).to(device)
        self.optimizer = optim.Adam(
            self.model.parameters(),
            lr=LEARNING_RATE,
            amsgrad=True,
            weight_decay=LEARNING_RATE * 0.1,
        )
        self.scaler = GradScaler()
        self.update_target_model()
        self.loss = nn.MSELoss().to(device)

    def update_target_model(self):
        """
        Updates the target model by copying the weights from the current moc
        """
        self.target_model.load_state_dict(self.model.state_dict())

    def act(self, state):
        """
        Selects an action using the epsilon-greedy policy.

        Parameters:
        ----------
        state : np.ndarray
            The current state.

        Returns:
        -------
        int
            The action selected by the agent.
        """
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = (
            torch.FloatTensor(np.concatenate([state.real, state.imag]))
            .unsqueeze(0)
            .to(device)
        )
        q_vals = self.model(state)
        return torch.argmax(q_vals).item()

    def remember(self, state, action, reward, next_state, done):
        """
        Stores an experience in the replay memory.

        Parameters:
        ----------
        state : np.ndarray
            The state before taking the action.
        action : int
            The action taken by the agent.
        reward : float
            The reward received after taking the action.
        next_state : np.ndarray
```

```python
            The state after taking the action.
        done : bool
            Whether the episode has ended.
        """
        self.memory.push(state, action, reward, next_state, done)

    def replay(self):
        """
        Trains the model by replaying a batch of experiences from the replay
        """
        if len(self.memory) < BATCH_SIZE:
            return
        batch = self.memory.sample(BATCH_SIZE)
        states, actions, rewards, next_states, dones = zip(*batch)

        self.model.train()
        self.target_model.eval()

        states = torch.FloatTensor(
            np.array([np.concatenate([s.real, s.imag]) for s in states])
        ).to(device)
        actions = torch.LongTensor(actions).to(device)
        rewards = torch.FloatTensor(rewards).to(device)
        next_states = torch.FloatTensor(
            np.array([np.concatenate([s.real, s.imag]) for s in next_states]
        ).to(device)
        dones = torch.FloatTensor(dones).to(device)

        with autocast():
            q_vals = self.model(states).gather(1, actions.unsqueeze(1)).sque
            next_q_vals = self.target_model(next_states).max(1)[0]
            target_q_vals = rewards + (GAMMA * next_q_vals * (1 - dones))
            loss = self.loss(q_vals, target_q_vals)

        self.optimizer.zero_grad(set_to_none=True)
        self.scaler.scale(loss).backward()
        self.scaler.step(self.optimizer)
        self.scaler.update()

        if self.epsilon > MIN_EPSILON:
            self.epsilon *= EPSILON_DECAY
```

In [14]:

```python
# Training the agent
def train_agent(agent, env, episodes, target_update, fidelity_threshold, pat

    total_rewards = []
    fidelities = []
    state_history = []
    amplitudes_history = []
    phases_history = []
    durations_history = []
    best_fidelity = 0
    patience_counter = 0

    for e in range(episodes):
        state = env.reset()
```

```
        total_reward = 0
        for time in range(env.max_steps):
            action = agent.act(state)
            next_state, reward, done, amplitudes, phases, durations = env.st
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward

            if done:
                break
        agent.replay()

        if e % target_update == 0:
            agent.update_target_model()

        state_history.append(state)
        amplitudes_history.append(amplitudes)
        phases_history.append(phases)
        durations_history.append(durations)
        total_rewards.append(total_reward)
        current_fidelity = 1 - env.infidelity(state)
        fidelities.append(current_fidelity)

        if e % 100 == 0:
            print(
                f"Episode: {e}/{episodes},  Total Reward: {total_reward:.5f}
            )
        # Early stopping check
        if current_fidelity >= best_fidelity:
            best_fidelity = current_fidelity
            patience_counter = 0  # Reset patience counter if fidelity impro
        else:
            patience_counter += 1

        if best_fidelity >= (1 - fidelity_threshold) or patience_counter >=
            print(
                f"Early stopping triggered. Achieved fidelity: {best_fidelit
            )
            break

    print("Training finished.")
    return total_rewards, fidelities, state_history, amplitudes_history, pha
```

In [15]:

```
def plot_results(total_rewards, fidelities, amplitudes, phases, durations):
    """
    Plots the results of the training process, including total rewards, fide

    Parameters:
    ----------
    total_rewards : list
        A list of total rewards per episode.
    fidelities : list
        A list of fidelities per episode.
    amplitudes : list
        A list of amplitudes of control pulses per episode.
    phases : list
```

```
        A list of phases of control pulses per episode.
    durations : list
        A list of durations of control pulses per episode.
    """
    plt.figure(figsize=(15, 10))

    # Plot total rewards
    plt.subplot(3, 1, 1)
    plt.plot(total_rewards, label="Total Reward per Episode")
    plt.xlabel("Episode")
    plt.ylabel("Total Reward")
    plt.title("Training Progress")
    plt.legend()
    plt.grid()

    # Plot fidelities
    plt.subplot(3, 1, 2)
    plt.plot(fidelities, label="Fidelity per Episode")
    plt.xlabel("Episode")
    plt.ylabel("Fidelity")
    plt.legend()
    plt.grid()

    # Plot control pulse parameters
    plt.subplot(3, 1, 3)
    plt.plot(amplitudes, label="Amplitude per Episode", color='r')
    plt.plot(phases, label="Phase per Episode", color='g')
    plt.plot(durations, label="Duration per Episode", color='b')
    plt.xlabel("Episode")
    plt.ylabel("Control Pulse Parameters")
    plt.legend()
    plt.grid()

    plt.tight_layout()
    plt.show()
```

H-Gate

In [16]:
```
# Initialize environment and agent
env = QuantumGateEnv(gate='H')
agent = DDDQNAgent(env.state_size, env.action_size)
```

In [17]:
```
# Compile the model (requires PyTorch 2.0 or later)
if torch.__version__ >= "2.0.0":
    agent.model = torch.compile(agent.model)
    agent.target_model = torch.compile(agent.target_model)
```

In [18]:
```
# Train the agent
(
    total_rewards,
    fidelities,
    state_history,
    amplitudes_history,
    phases_history,
```

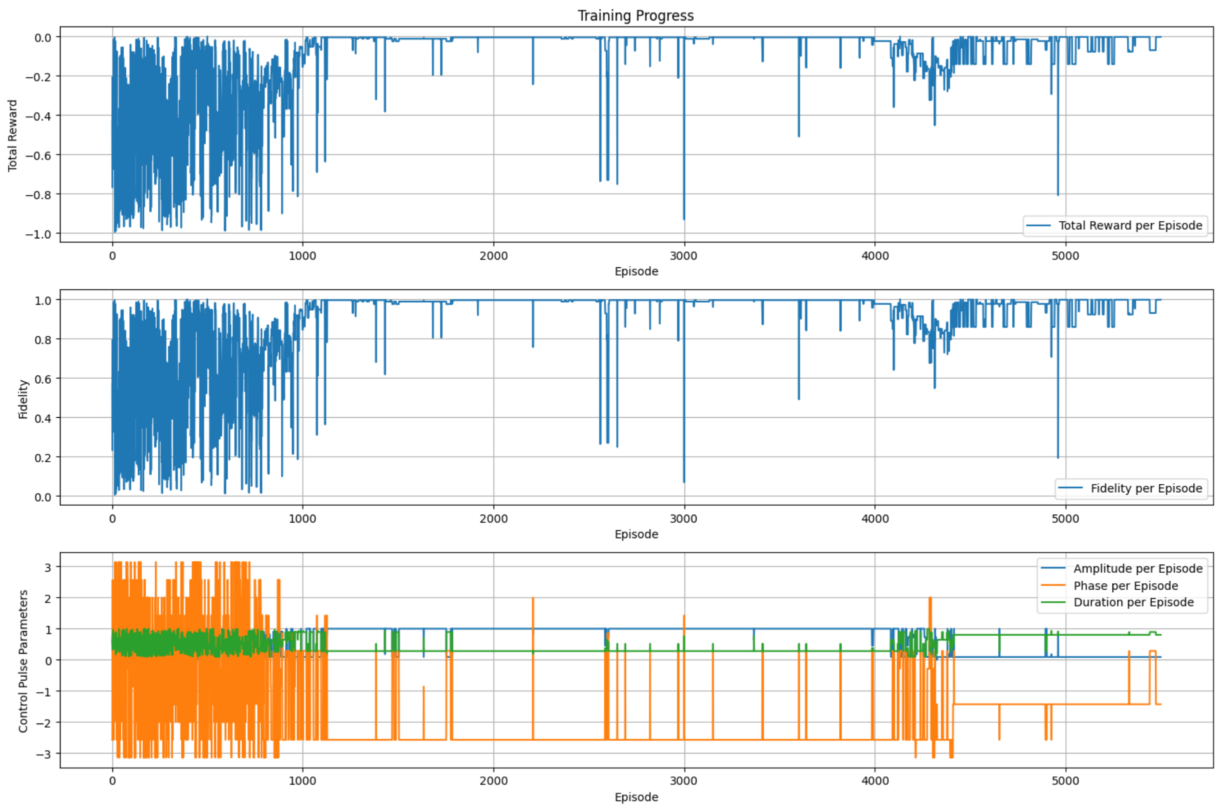
```
    durations_history,
) = train_agent(agent, env, EPISODES, TARGET_UPDATE, FIDELITY_THRESHOLD, PAT
```

```
Episode: 0/10000,  Total Reward: -0.43165, Fidelity: 0.56835,  Epsilon: 1.00
0
Episode: 100/10000,  Total Reward: -0.16610, Fidelity: 0.83390,  Epsilon: 0.
640
Episode: 200/10000,  Total Reward: -0.16474, Fidelity: 0.83526,  Epsilon: 0.
388
Episode: 300/10000,  Total Reward: -0.84419, Fidelity: 0.15581,  Epsilon: 0.
235
Episode: 400/10000,  Total Reward: -0.43772, Fidelity: 0.56228,  Epsilon: 0.
142
Episode: 500/10000,  Total Reward: -0.22449, Fidelity: 0.77551,  Epsilon: 0.
086
Episode: 600/10000,  Total Reward: -0.20590, Fidelity: 0.79410,  Epsilon: 0.
052
Episode: 700/10000,  Total Reward: -0.08214, Fidelity: 0.91786,  Epsilon: 0.
032
Episode: 800/10000,  Total Reward: -0.20501, Fidelity: 0.79499,  Epsilon: 0.
019
Episode: 900/10000,  Total Reward: -0.51021, Fidelity: 0.48979,  Epsilon: 0.
012
Episode: 1000/10000,  Total Reward: -0.15556, Fidelity: 0.84444,  Epsilon:
0.007
Episode: 1100/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.004
Episode: 1200/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.003
Episode: 1300/10000,  Total Reward: -0.00629, Fidelity: 0.99371,  Epsilon:
0.002
Episode: 1400/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 1500/10000,  Total Reward: -0.02487, Fidelity: 0.97513,  Epsilon:
0.001
Episode: 1600/10000,  Total Reward: -0.01204, Fidelity: 0.98796,  Epsilon:
0.001
Episode: 1700/10000,  Total Reward: -0.01204, Fidelity: 0.98796,  Epsilon:
0.001
Episode: 1800/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 1900/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2000/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2100/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2200/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2300/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2400/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2500/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2600/10000,  Total Reward: -0.73095, Fidelity: 0.26905,  Epsilon:
0.001
Episode: 2700/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
```

```
Episode: 2800/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 2900/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3000/10000,  Total Reward: -0.93198, Fidelity: 0.06802,  Epsilon:
0.001
Episode: 3100/10000,  Total Reward: -0.01204, Fidelity: 0.98796,  Epsilon:
0.001
Episode: 3200/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3300/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3400/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3500/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3600/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3700/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3800/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 3900/10000,  Total Reward: -0.00483, Fidelity: 0.99517,  Epsilon:
0.001
Episode: 4000/10000,  Total Reward: -0.02377, Fidelity: 0.97623,  Epsilon:
0.001
Episode: 4100/10000,  Total Reward: -0.10220, Fidelity: 0.89780,  Epsilon:
0.001
Episode: 4200/10000,  Total Reward: -0.10744, Fidelity: 0.89256,  Epsilon:
0.001
Episode: 4300/10000,  Total Reward: -0.00996, Fidelity: 0.99004,  Epsilon:
0.001
Episode: 4400/10000,  Total Reward: -0.05633, Fidelity: 0.94367,  Epsilon:
0.001
Episode: 4500/10000,  Total Reward: -0.00190, Fidelity: 0.99810,  Epsilon:
0.001
Episode: 4600/10000,  Total Reward: -0.02369, Fidelity: 0.97631,  Epsilon:
0.001
Episode: 4700/10000,  Total Reward: -0.02369, Fidelity: 0.97631,  Epsilon:
0.001
Episode: 4800/10000,  Total Reward: -0.14122, Fidelity: 0.85878,  Epsilon:
0.001
Episode: 4900/10000,  Total Reward: -0.06898, Fidelity: 0.93102,  Epsilon:
0.001
Episode: 5000/10000,  Total Reward: -0.00280, Fidelity: 0.99720,  Epsilon:
0.001
Episode: 5100/10000,  Total Reward: -0.00280, Fidelity: 0.99720,  Epsilon:
0.001
Episode: 5200/10000,  Total Reward: -0.07780, Fidelity: 0.92220,  Epsilon:
0.001
Episode: 5300/10000,  Total Reward: -0.00280, Fidelity: 0.99720,  Epsilon:
0.001
Episode: 5400/10000,  Total Reward: -0.00280, Fidelity: 0.99720,  Epsilon:
0.001
Early stopping triggered. Achieved fidelity: 0.99983, Episode: 5499, Patienc
```

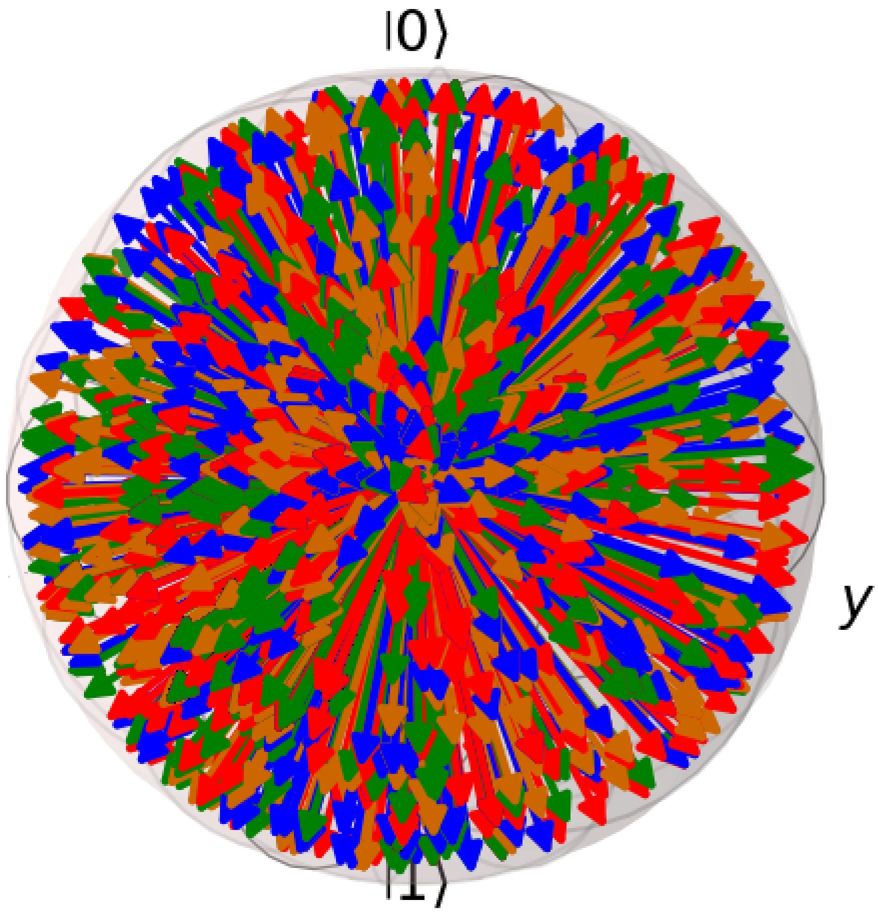
```
e: 5000, Total Reward: -0.00280, Epsilon: 0.001
Training finished.
```

In [19]:
```python
# Plot the results
plot_results(
    total_rewards,
    fidelities,
    amplitudes_history,
    phases_history,
    durations_history
)
```



In [20]:
```python
def plot_bloch_sphere_trajectory_qutip(initial_state, states):
    bloch = Bloch()
    # num_states = len(states) + 1  # Include the initial state

    # # Define colors for each state, ensuring enough colors are provided
    # colors = ["r", "g", "b", "m", "y", "c"]
    # colors = colors * (num_states // len(colors)) + colors[: num_states %

    # Add initial state with the first color
    bloch.add_states(Qobj(initial_state))

    # Add other states with different colors
    for i, state in enumerate(states):
        bloch.add_states(Qobj(state))

    bloch.show()
    plt.show()
```

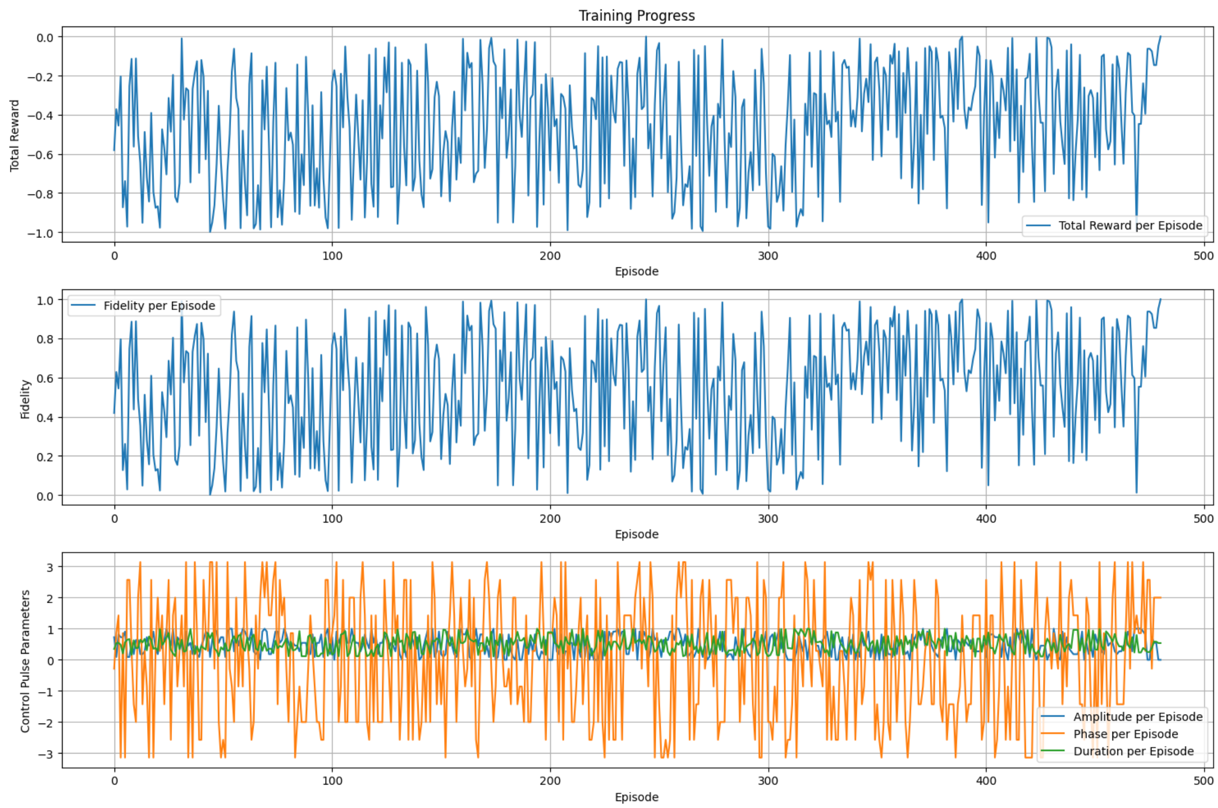
In [21]:
```python
plot_bloch_sphere_trajectory_qutip(env.initial_state, state_history)
```

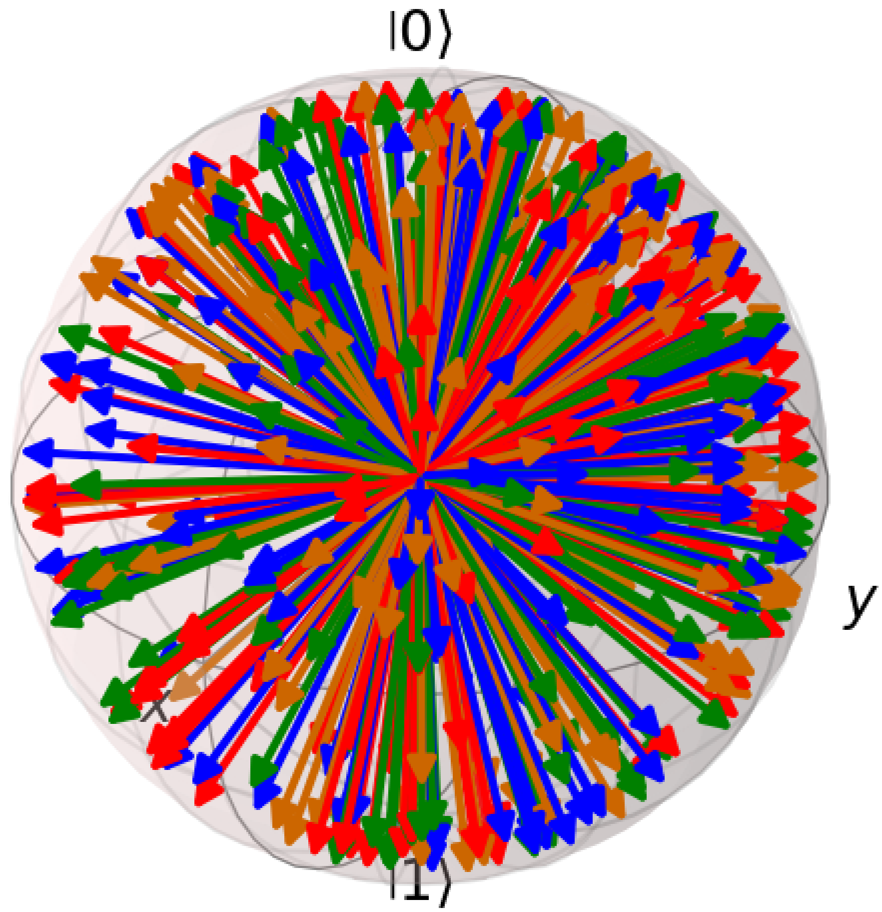


T-Gate

```
In [22]: env = QuantumGateEnv(gate="T")
         agent = DDDQNAgent(env.state_size, env.action_size)
         (
             total_rewards,
             fidelities,
             state_history,
             amplitudes_history,
             phases_history,
             durations_history,
         ) = train_agent(agent, env, EPISODES, TARGET_UPDATE, FIDELITY_THRESHOLD, PAT
         plot_results(
             total_rewards, fidelities, amplitudes_history, phases_history, durations
         )
         plot_bloch_sphere_trajectory_qutip(env.initial_state, state_history)
```

```
Episode: 0/10000,  Total Reward: -0.58114, Fidelity: 0.41886,  Epsilon: 1.00
0
Episode: 100/10000,  Total Reward: -0.23426, Fidelity: 0.76574,  Epsilon: 0.
640
Episode: 200/10000,  Total Reward: -0.68450, Fidelity: 0.31550,  Epsilon: 0.
388
Episode: 300/10000,  Total Reward: -0.97158, Fidelity: 0.02842,  Epsilon: 0.
235
Episode: 400/10000,  Total Reward: -0.11959, Fidelity: 0.88041,  Epsilon: 0.
142
Early stopping triggered. Achieved fidelity: 1.00000, Episode: 480, Patienc
e: 0, Total Reward: 0.00000, Epsilon: 0.095
Training finished.
```





CNOT-Gate

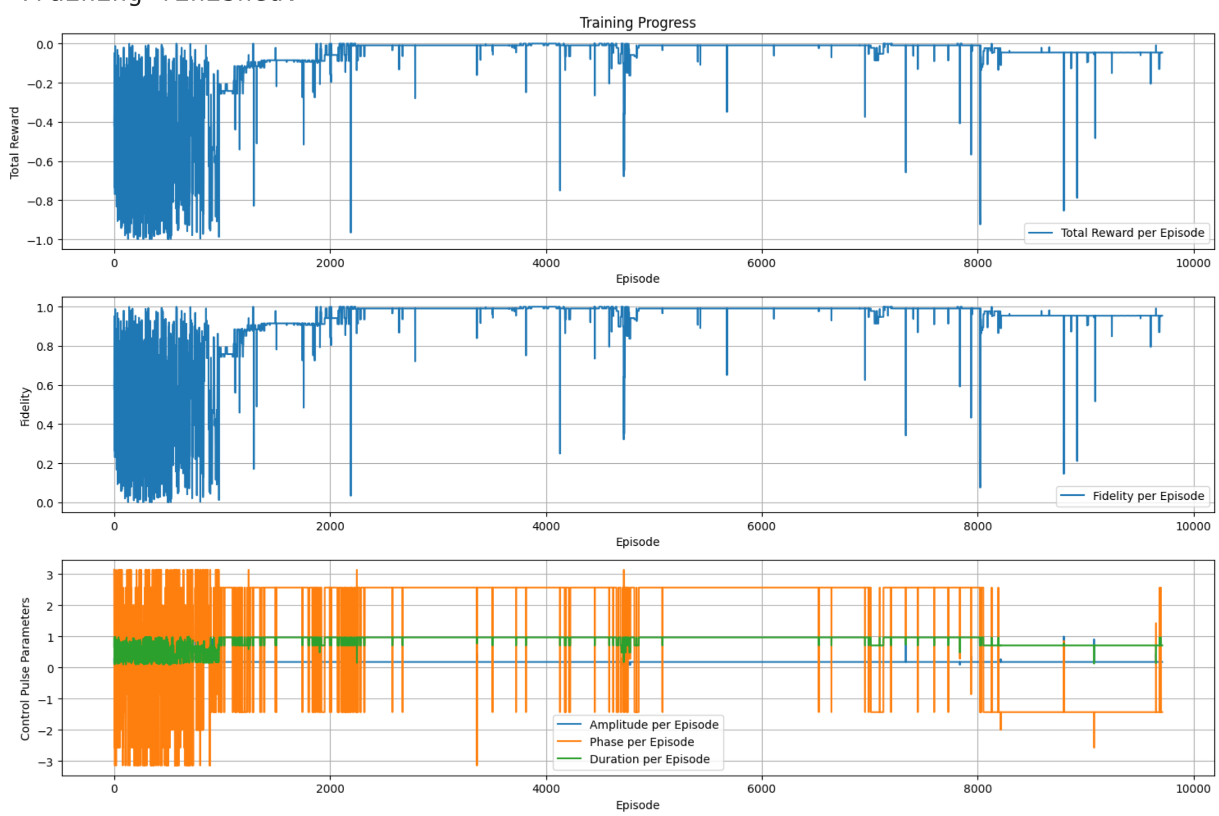
```
In [23]: env = QuantumGateEnv(gate="CNOT")
agent = DDDQNAgent(env.state_size, env.action_size)
(
    total_rewards,
    fidelities,
    state_history,
    amplitudes_history,
    phases_history,
    durations_history,
) = train_agent(agent, env, EPISODES, TARGET_UPDATE, FIDELITY_THRESHOLD, PAT
plot_results(
    total_rewards, fidelities, amplitudes_history, phases_history, durations
)
```

```
Episode: 0/10000,  Total Reward: -0.41871, Fidelity: 0.58129,  Epsilon: 1.00
0
Episode: 100/10000,  Total Reward: -0.98032, Fidelity: 0.01968,  Epsilon: 0.
640
Episode: 200/10000,  Total Reward: -0.86409, Fidelity: 0.13591,  Epsilon: 0.
388
Episode: 300/10000,  Total Reward: -0.74080, Fidelity: 0.25920,  Epsilon: 0.
235
Episode: 400/10000,  Total Reward: -0.97531, Fidelity: 0.02469,  Epsilon: 0.
142
Episode: 500/10000,  Total Reward: -1.00000, Fidelity: 0.00000,  Epsilon: 0.
086
Episode: 600/10000,  Total Reward: -0.71287, Fidelity: 0.28713,  Epsilon: 0.
052
Episode: 700/10000,  Total Reward: -0.55994, Fidelity: 0.44006,  Epsilon: 0.
032
Episode: 800/10000,  Total Reward: -0.51084, Fidelity: 0.48916,  Epsilon: 0.
019
Episode: 900/10000,  Total Reward: -0.52924, Fidelity: 0.47076,  Epsilon: 0.
012
Episode: 1000/10000,  Total Reward: -0.24256, Fidelity: 0.75744,  Epsilon:
0.007
Episode: 1100/10000,  Total Reward: -0.25627, Fidelity: 0.74373,  Epsilon:
0.004
Episode: 1200/10000,  Total Reward: -0.14943, Fidelity: 0.85057,  Epsilon:
0.003
Episode: 1300/10000,  Total Reward: -0.09428, Fidelity: 0.90572,  Epsilon:
0.002
Episode: 1400/10000,  Total Reward: -0.09428, Fidelity: 0.90572,  Epsilon:
0.001
Episode: 1500/10000,  Total Reward: -0.08555, Fidelity: 0.91445,  Epsilon:
0.001
Episode: 1600/10000,  Total Reward: -0.09428, Fidelity: 0.90572,  Epsilon:
0.001
Episode: 1700/10000,  Total Reward: -0.09428, Fidelity: 0.90572,  Epsilon:
0.001
Episode: 1800/10000,  Total Reward: -0.08917, Fidelity: 0.91083,  Epsilon:
0.001
Episode: 1900/10000,  Total Reward: -0.08555, Fidelity: 0.91445,  Epsilon:
0.001
Episode: 2000/10000,  Total Reward: -0.15738, Fidelity: 0.84262,  Epsilon:
0.001
Episode: 2100/10000,  Total Reward: -0.00049, Fidelity: 0.99951,  Epsilon:
0.001
Episode: 2200/10000,  Total Reward: -0.00049, Fidelity: 0.99951,  Epsilon:
0.001
Episode: 2300/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 2400/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 2500/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 2600/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 2700/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
```

```
Episode: 2800/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 2900/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3000/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3100/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3200/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3300/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3400/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3500/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3600/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3700/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 3800/10000,  Total Reward: -0.00049, Fidelity: 0.99951,  Epsilon: 0.001
Episode: 3900/10000,  Total Reward: -0.00049, Fidelity: 0.99951,  Epsilon: 0.001
Episode: 4000/10000,  Total Reward: -0.00049, Fidelity: 0.99951,  Epsilon: 0.001
Episode: 4100/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 4200/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 4300/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 4400/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 4500/10000,  Total Reward: -0.00049, Fidelity: 0.99951,  Epsilon: 0.001
Episode: 4600/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 4700/10000,  Total Reward: -0.05240, Fidelity: 0.94760,  Epsilon: 0.001
Episode: 4800/10000,  Total Reward: -0.05871, Fidelity: 0.94129,  Epsilon: 0.001
Episode: 4900/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 5000/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 5100/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 5200/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 5300/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 5400/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
Episode: 5500/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon: 0.001
```

```
Episode: 5600/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 5700/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 5800/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 5900/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6000/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6100/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6200/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6300/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6400/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6500/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6600/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6700/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6800/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 6900/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7000/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7100/10000,  Total Reward: -0.05240, Fidelity: 0.94760,  Epsilon:
0.001
Episode: 7200/10000,  Total Reward: -0.00189, Fidelity: 0.99811,  Epsilon:
0.001
Episode: 7300/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7400/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7500/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7600/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7700/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7800/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 7900/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 8000/10000,  Total Reward: -0.00949, Fidelity: 0.99051,  Epsilon:
0.001
Episode: 8100/10000,  Total Reward: -0.03738, Fidelity: 0.96262,  Epsilon:
0.001
Episode: 8200/10000,  Total Reward: -0.02379, Fidelity: 0.97621,  Epsilon:
0.001
Episode: 8300/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
```

```
Episode: 8400/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 8500/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 8600/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 8700/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 8800/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 8900/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 9000/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 9100/10000,  Total Reward: -0.04666, Fidelity: 0.95334,  Epsilon:
0.001
Episode: 9200/10000,  Total Reward: -0.04666, Fidelity: 0.95334,  Epsilon:
0.001
Episode: 9300/10000,  Total Reward: -0.04666, Fidelity: 0.95334,  Epsilon:
0.001
Episode: 9400/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 9500/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 9600/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Episode: 9700/10000,  Total Reward: -0.04619, Fidelity: 0.95381,  Epsilon:
0.001
Early stopping triggered. Achieved fidelity: 0.99994, Episode: 9711, Patienc
e: 5000, Total Reward: -0.04619, Epsilon: 0.001
Training finished.
```



In [24]:
```python
def plot_q_sphere(states, initial_state):
    """
    Plot the QSphere for a list of quantum states using Matplotlib, includir
    """
    fig = plt.figure(figsize=(8, 8))
    ax = fig.add_subplot(111, projection="3d")

    # Draw the QSphere
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x = np.outer(np.cos(u), np.sin(v))
    y = np.outer(np.sin(u), np.sin(v))
    z = np.outer(np.ones(np.size(u)), np.cos(v))

    ax.plot_surface(x, y, z, color="r", alpha=0.1)

    # Plot initial state with a distinct color
    alpha_init, beta_init = initial_state[2], initial_state[3]
    theta_init = 2 * np.arccos(np.abs(alpha_init))
    phi_init = np.angle(beta_init) - np.angle(alpha_init)
    x_init = np.sin(theta_init) * np.cos(phi_init)
    y_init = np.sin(theta_init) * np.sin(phi_init)
    z_init = np.cos(theta_init)
    ax.plot([0, x_init], [0, y_init], [0, z_init], color="black", linestyle=
    ax.scatter(
        x_init,
        y_init,
        z_init,
        color="black",
        s=100,
    )

    # Plot other states
    for i, state in enumerate(states):
        # Extract amplitudes
        alpha, beta = state[2], state[3]

        # Calculate spherical coordinates
        theta = 2 * np.arccos(np.abs(alpha))  # Polar angle
        phi = np.angle(beta) - np.angle(alpha)  # Azimuthal angle

        # Cartesian coordinates
        x = np.sin(theta) * np.cos(phi)
        y = np.sin(theta) * np.sin(phi)
        z = np.cos(theta)

        # Color based on phase
        color = cm.hsv((phi + np.pi) / (2 * np.pi))

        ax.plot([0, x], [0, y], [0, z], color=color)
        ax.scatter(x, y, z, color=color, s=100)

    # Set axis limits
    ax.set_xlim([-1, 1])
    ax.set_ylim([-1, 1])
```

```python
    ax.set_zlim([-1, 1])

    # Set labels
    ax.set_xlabel("Re(α)")
    ax.set_ylabel("Im(α)")
    ax.set_zlabel("Re(β)")
    plt.show()
```

In [25]:
```python
plot_q_sphere(state_history, env.initial_state)
```