main.py

```
import time
import cv2
import numpy as np
import pyautogui
import pygetwindow as gw
import torch
from ai import GettingOverItAI
from constants import GAME_TITLE, BATCH_SIZE, SAVE_INTERVAL, BUFFER_MAXLEN
from models import SimpleModel, CuriosityModel
from preprocessing import preprocess_frame
from buffer import ReplayBuffer
from torch import optim

# Initialize the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Game details
window = gw.getWindowsWithTitle(GAME_TITLE)[0]
x, y, width, height = window.left, window.top, window.width, window.height

# The number of color channels in the input images
# This is 3 for RGB images or 1 for grayscale images
channels = 3  # Replace with the number of color channels in your preprocessed
images

# The number of frames stacked together
STACK_SIZE = 4  # This is a common value, but you should replace it with the actual
number you're using

input_shape = (STACK_SIZE * channels, height, width)
model = SimpleModel(input_shape)


def main():
    curiosity_model = CuriosityModel()
    ai = GettingOverItAI(model, curiosity_model)
    replay_buffer = ReplayBuffer(BUFFER_MAXLEN)

    last_save_time = time.time()

    # Define an optimizer
    optimizer = optim.Adam(model.parameters())

    while True:
        try:
            # Capture a screenshot of the game
            game_screen = pyautogui.screenshot(region=(x, y, width, height))
            game_screen_np = np.array(game_screen)
```

```
            # Preprocess the game screen
            preprocessed_screen_np = preprocess_frame(game_screen_np)

            # Convert the 12-channels image to multiple 3-channels (RGB) images for
visualization
            for i in range(0, preprocessed_screen_np.shape[2], 3):  # Assume the
shape is (height, width, 12)
                visualization = preprocessed_screen_np[:, :, i:i + 3]  # Take three
channels at a time

                # Scale the visualization to range 0-255 if it's not already in
that range
                visualization = cv2.normalize(visualization, None, alpha=0.0,
beta=255.0, norm_type=cv2.NORM_MINMAX,
                                              dtype=0)
                cv2.imshow(f'AI View Channels {i + 1}-{i + 3}', visualization)
                cv2.waitKey(1)  # Refresh the display every 1 millisecond

            # Convert the numpy array to a tensor and move it to the device
            state_tensor =
torch.from_numpy(preprocessed_screen_np).float().to(device)

            # Convert the tensor back to a numpy array
            state = state_tensor.cpu().numpy()

            # Feed the preprocessed screen into the AI model
            predicted_action = ai.predict(state)

            # Save the current screen and action to the buffer
            replay_buffer.push(preprocessed_screen_np, predicted_action, 0, 0,
False)

            if len(replay_buffer) >= BATCH_SIZE:
                # When replay buffer has enough experiences
                state, action, reward, next_state, done =
replay_buffer.sample(BATCH_SIZE)

                # Set the gradients to zero
                optimizer.zero_grad()

                # Compute the loss
                loss = ai.compute_loss(state, action, reward, next_state, done)

                # Backpropagation
                loss.backward()

                # Optimization step
                optimizer.step()
```

```python
                # Save the model at a regular interval
                if time.time() - last_save_time >= SAVE_INTERVAL:
                    ai.save_model()
                    last_save_time = time.time()

            time.sleep(0.1)

        except Exception as e:
            print(f"An error occurred: {e}")
            break

        finally:
            cv2.destroyAllWindows()

    ai.save_model()


if __name__ == "__main__":
    main()
```

ai.py

```python
import logging
import cv2
import numpy as np
import pyautogui
import torch
import torch.nn as nn
import torch.optim as optim
from skopt import gp_minimize
from torch import device
from torchvision.transforms import ToTensor
from buffer import ReplayBuffer
from constants import BUFFER_MAXLEN, EPSILON, EPSILON_DECAY, EPSILON_MIN,
OUTPUT_SIZE, BATCH_SIZE
from preprocessing import preprocess_frame, PLAYER_COLOR

logging.basicConfig(filename='ai.log', level=logging.ERROR)


# Initialize the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")


class GettingOverItAI:
    def __init__(self, model, curiosity_model):
        self.model = model
```

```
        self.curiosity_model = curiosity_model
        self.loss_function = nn.MSELoss()
        self.optimizer = optim.Adam(self.model.parameters(), lr=0.00025)
        self.curiosity_optimizer = optim.Adam(self.curiosity_model.parameters(),
lr=0.00025)
        self.transform = ToTensor()
        self.total_reward = 0.0
        self.previous_position = None
        self.buffer = ReplayBuffer(BUFFER_MAXLEN)
        self.epsilon = EPSILON
        self.steps_since_progress = 0
        self.max_position = None
        self.previous_mouse_position = None
        self.previous_position = 0.0
        self.gamma = 0.99  # Discount factor for future rewards

    @staticmethod
    def get_screen_shot():
        try:
            # Capture the entire screen
            screen = np.array(pyautogui.screenshot())
            # Convert the image from BGR to RGB color space
            screen = cv2.cvtColor(screen, cv2.COLOR_BGR2RGB)
            return screen
        except Exception as e:
            logging.error(f'Error capturing screen: {e}')

    @staticmethod
    def perform_mouse_movement(action):
        try:
            # Get the current mouse position
            current_position = np.array(pyautogui.position())
            # Calculate the new mouse position
            new_position = current_position + action
            # Ensure the new position is within the screen bounds
            new_position = np.clip(new_position, 0, np.array(pyautogui.size()))
            # Move the mouse to the new position
            pyautogui.moveTo(new_position[0], new_position[1], duration=0.1)
        except Exception as e:
            logging.error(f'Error performing mouse movement: {e}')

    def simulate_game(self, num_episodes=50):
        total_reward = 0.0
        self.previous_position = 0  # Initialize to a default value

        for _ in range(num_episodes):
            done = False

            # Get the initial game screen
            game_screen = self.get_screen_shot()
```

```
            # Preprocess the game screen
            state = preprocess_frame(game_screen)

            while not done:
                action = self.get_mouse_movement()
                # Execute the action
                self.perform_mouse_movement(action)

                # Get the next game screen
                next_game_screen = self.get_screen_shot()

                # Preprocess the next game screen
                next_state = preprocess_frame(next_game_screen)

                # Determine if the game is "done"
                current_position = self.get_player_position(state)
                done = self.is_game_done(current_position)

                reward = self.get_reward(current_position)
                self.add_experience_to_buffer(state, action, reward, next_state,
done)
                self.train(BATCH_SIZE)

                # Update previous_position
                self.previous_position = current_position

                state = next_state
                total_reward += reward

        average_reward = total_reward / num_episodes
        return average_reward

    def compute_loss(self, states, actions, rewards, next_states, dones):
        # Convert data to tensors
        states = torch.FloatTensor(states).to(device)
        next_states = torch.FloatTensor(next_states).to(device)
        actions = torch.LongTensor(actions).unsqueeze(-1).to(device)  # Ensure that
actions are a column vector
        rewards = torch.FloatTensor(rewards).to(device)
        dones = torch.FloatTensor(dones).to(device)

        # Compute Q-values for the current states and next states
        curr_q_values = self.model(states)
        curr_q = curr_q_values.gather(1, actions).squeeze(-1)  # Gather along the
action dimension
        next_q_values = self.model(next_states)
        next_q = next_q_values.max(1)[0]

        # Compute target Q-values
```

```
        target_q = rewards + (1 - dones) * self.gamma * next_q

        # Compute loss
        loss = self.loss_function(curr_q, target_q.detach())
        return loss

    def objective(self, params):
        # Unpack the parameters
        num_episodes, = params

        # Simulate the game
        average_reward = self.simulate_game(num_episodes)

        # We're minimizing the objective function, so return the negative of the
reward
        return -average_reward

    def optimize(self):
        # Optimize the AI
        res = gp_minimize(self.objective, [(1, 100)], n_calls=50, random_state=0)

        # Print the result
        print(f'Best reward: {-res.fun}')

    def get_mouse_movement(self):
        if np.random.rand() < self.epsilon:
            # Perform a random action
            action = np.random.randint(OUTPUT_SIZE)
        else:
            # Get the best action according to the model
            with torch.no_grad():
                state_tensor =
torch.from_numpy(self.previous_position).float().unsqueeze(0).to(device)
                q_values = self.model(state_tensor)
                action = torch.argmax(q_values).item()

        # Decay the epsilon value
        self.epsilon = max(self.epsilon * EPSILON_DECAY, EPSILON_MIN)
        return action

    def get_player_position(self, state):
        # Function to determine the player's current position
        # This depends on the specifics of your game and may need to be adjusted
        player_pixels = np.where(np.all(state == PLAYER_COLOR / 255, axis=-1))
        if player_pixels[0].size > 0:
            return np.array([player_pixels[0].mean(), player_pixels[1].mean()])
        else:
            return self.previous_position

    def is_game_done(self, current_position):
```

```
        # Function to determine if the game is "done" or not
        # This will need to be adjusted based on the specifics of your game
        done = False
        if self.previous_position is not None:
            # Consider the game done if the player has not made progress in the
last 200 steps
            self.steps_since_progress += 1
            if np.any(current_position > self.previous_position):
                self.steps_since_progress = 0
            if self.steps_since_progress > 200:
                done = True
        return done

    def get_reward(self, current_position):
        # Function to calculate the reward based on the player's current position
        # This will need to be adjusted based on the specifics of your game
        if self.previous_position is not None:
            reward = np.maximum(current_position - self.previous_position,
0.0).sum()
        else:
            reward = 0.0
        return reward

    def add_experience_to_buffer(self, state, action, reward, next_state, done):
        # Adds an experience to the replay buffer
        state = torch.from_numpy(state).float().to(device)
        next_state = torch.from_numpy(next_state).float().to(device)
        self.buffer.push(state, action, reward, next_state, done)

    def predict(self, state):
        with torch.no_grad():
            state_tensor = torch.from_numpy(state).float().unsqueeze(0).to(device)
            q_values = self.model(state_tensor)
            action = torch.argmax(q_values).item()
            return action

    def train(self, batch_size):
        # Trains the AI based on experiences from the replay buffer
        if len(self.buffer) >= batch_size:
            states, actions, rewards, next_states, dones =
self.buffer.sample(batch_size)
            targets = self.model(states).tolist()
            next_q_values = self.model(next_states).detach().numpy().max(axis=1)
            for i, done in enumerate(dones):
                if done:
                    targets[i][actions[i]] = rewards[i]
                else:
                    targets[i][actions[i]] = rewards[i] + next_q_values[i]
            targets = torch.tensor(targets).to(device)
            self.optimizer.zero_grad()
```

```
            loss = self.loss_function(self.model(states), targets)
            loss.backward()
            self.optimizer.step()

    def save_model(self, model_file_path='model.pth',
curiosity_model_file_path='curiosity_model.pth'):
        torch.save(self.model.state_dict(), model_file_path)
        torch.save(self.curiosity_model.state_dict(), curiosity_model_file_path)
```

models.py

```
import torch
from constants import OUTPUT_SIZE
import torch.nn as nn
import numpy as np


class SimpleModel(nn.Module):
    def __init__(self, input_shape):  # Remove the default value
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=5, stride=2),
nn.BatchNorm2d(32), nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=5, stride=2), nn.BatchNorm2d(64),
nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=5, stride=2), nn.BatchNorm2d(64),
nn.ReLU(),
        )

        # Create a dummy input with the correct input shape
        dummy_input = torch.zeros(1, *input_shape)

        # Pass the dummy input through the conv layers to get the output size
        conv_output = self.conv(dummy_input)

        # Calculate the number of features from the output size
        num_features = int(np.prod(conv_output.size()))

        self.fc = nn.Sequential(
            nn.Linear(num_features, 512), nn.ReLU(),
            nn.Linear(512, OUTPUT_SIZE)
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)  # Flatten the tensor
        return self.fc(x)
```

```python
class CuriosityModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.inverse_model = nn.Sequential(
            nn.Linear(4480*2, 512), nn.ReLU(),
            nn.Linear(512, OUTPUT_SIZE)
        )
        self.forward_model = nn.Sequential(
            nn.Linear(4480+OUTPUT_SIZE, 512), nn.ReLU(),
            nn.Linear(512, 4480)
        )

    def forward(self, state, action, next_state):
        state_action = torch.cat([state, action], 1)
        predicted_next_state = self.forward_model(state_action)

        state_next_state = torch.cat([state, next_state], 1)
        predicted_action = self.inverse_model(state_next_state)

        return predicted_action, predicted_next_state
```

constants.py

```python
# Game details
GAME_TITLE = 'Getting Over It'

# Hyperparameters
OUTPUT_SIZE = 2
BATCH_SIZE = 64
EPSILON = 0.9
EPSILON_DECAY = 0.999
EPSILON_MIN = 0.1
BUFFER_MAXLEN = int(1e5)
REWARD_THRESHOLD = 100
SAVE_INTERVAL = 600  # Save model every 10 minutes

# Miscellaneous
MODEL_SAVE_PATH = 'model_save.pth'
```

preprocessing.py

```python
from collections import deque
```

```python
import cv2
import numpy as np
import pyautogui
import pygetwindow as gw
import matplotlib.pyplot as plt

# Game details
game_title = 'Getting Over It'
window = gw.getWindowsWithTitle(game_title)[0]
x, y, width, height = window.left, window.top, window.width, window.height

# Frame stacking
stack_size = 4  # Adjust this based on your game's temporal dynamics
frame_stack = deque(maxlen=stack_size)

# Define the color for each object in the abstracted version
PLAYER_COLOR = np.array([255, 0, 0])  # Red
obstacle_color = np.array([0, 255, 0])  # Green
environment_color = np.array([0, 0, 255])  # Blue

# Data augmentation
noise_std = 0.1  # Adjust this based on the scale and nature of your data


def add_noise(frame):
    noise = np.random.normal(0, noise_std, frame.shape)
    frame += noise
    frame = np.clip(frame, 0, 1)  # Ensure the added noise doesn't cause data to go
out of range
    return frame


def rotate_frame(frame, angle):
    # Get image height and width
    (h, w) = frame.shape[:2]

    # Define the center of the image
    center = (w / 2, h / 2)

    # Perform the rotation
    m = cv2.getRotationMatrix2D(center, angle, 1.0)
    rotated = cv2.warpAffine(frame, m, (w, h))

    return rotated


def visualize_mask(mask, title):
    plt.figure(figsize=(10, 10))
    plt.imshow(mask, cmap='hot')
    plt.title(title)
```

```
    plt.show()


def visualize_histogram(data, title):
    plt.figure(figsize=(10, 10))
    plt.hist(data.ravel(), bins=256, color='orange', )
    plt.title(title)
    plt.show()


def visualize_frame_stack(frame_stack):
    for i, frame in enumerate(frame_stack):
        # Transpose the frame back to its original shape
        frame = np.transpose(frame, (1, 2, 0))

        # Rescale the frame to the range 0-255
        frame = (frame * 255).astype(np.uint8)

        plt.figure(figsize=(10, 10))
        plt.imshow(frame, cmap='gray')
        plt.title(f'Frame {i+1}')
        plt.show()


# Define the size for the low-res, pixelated version
low_res_size = (64, 64)  # Adjust this based on your needs


def preprocess_frame(frame, frame_stack):
    # Convert the frame to HSV color space
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    # Define the color range for each object
    player_range = (np.array([0, 100, 100]), np.array([10, 255, 255]))  # Red range
    obstacle_range = (np.array([35, 100, 100]), np.array([85, 255, 255]))  # Green
range
    environment_range = (np.array([110, 100, 100]), np.array([130, 255, 255]))  #
Blue range

    # Create a binary mask for each object
    player_mask = cv2.inRange(hsv, player_range[0], player_range[1])
    obstacle_mask = cv2.inRange(hsv, obstacle_range[0], obstacle_range[1])
    environment_mask = cv2.inRange(hsv, environment_range[0], environment_range[1])

    # Combine the masks into a single image
    preprocessed = np.zeros_like(frame)
    player_color = (255, 255, 255)  # example RGB color for the player
    preprocessed[player_mask > 0] = player_color
    preprocessed[obstacle_mask > 0] = obstacle_color
    preprocessed[environment_mask > 0] = environment_color
```

```
    # Resize the preprocessed frame to a lower resolution
    preprocessed = cv2.resize(preprocessed, low_res_size)

    # Normalize the preprocessed frame
    preprocessed = preprocessed / 255.0

    # Add noise for data augmentation
    preprocessed = add_noise(preprocessed)

    # Transpose the frame to have channels as the first dimension
    preprocessed = np.transpose(preprocessed, (2, 0, 1))

    # Add the preprocessed frame to the frame stack
    frame_stack.append(preprocessed)

    # Stack the frames along the channel dimension
    if len(frame_stack) < stack_size:
        # If stack is not full, pad with zeros
        stacked_frames = np.concatenate(
            [np.zeros(((stack_size - len(frame_stack)) * preprocessed.shape[0],
                       preprocessed.shape[1], preprocessed.shape[2])),
*frame_stack], axis=0)
    else:
        # If stack is full or overfull, drop the oldest frames and use the newest
ones
        stacked_frames = np.concatenate(list(frame_stack), axis=0)[-stack_size *
preprocessed.shape[0]:]

    return stacked_frames, frame_stack


def display_preprocessed_frame(frame):
    # Convert frame to a NumPy array if it's not already
    if not isinstance(frame, np.ndarray):
        frame = np.array(frame)

    # Undo the normalization and transpose back to the original shape
    display_frame = (frame * 255.0).astype(np.uint8)
    display_frame = np.transpose(display_frame, (1, 2, 0))

    # Use only the first three channels of the image
    if display_frame.shape[2] > 3:
        display_frame = display_frame[:, :, :3]

    # Create a figure with subplots
    fig, ax = plt.subplots(figsize=(10, 10))

    # Show the frame with a color map
    im = ax.imshow(display_frame, cmap='hot')
```

```python
    # Add a color bar
    cbar = fig.colorbar(im, ax=ax)
    cbar.set_label('Pixel Intensity')

    # Add a title
    ax.set_title('Preprocessed Frame')

    # Display the figure
    plt.show()


# Capture a screenshot of the game
while True:
    game_screen = pyautogui.screenshot(region=(x, y, width, height))
    game_screen_np = np.array(game_screen)
    preprocessed_frame, frame_stack = preprocess_frame(game_screen_np, frame_stack)

    display_preprocessed_frame(preprocessed_frame)  # Show the processed frame

    # Break the loop when 'q' is pressed
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cv2.destroyAllWindows()
```

buffer.py

```python
import random
from collections import deque
import numpy as np


class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        state = np.expand_dims(state, 0)
        next_state = np.expand_dims(next_state, 0)

        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sample(self.buffer,
```

```
batch_size))
        return np.concatenate(state), action, reward, np.concatenate(next_state),
done

    def __len__(self):
        return len(self.buffer)
```