AI Scene Maker - Comprehensive Review & Installation Guide

📋 Table of Contents

🚀 Installation Guide

Prerequisites

- Python 3.8 or higher
- Git
- FFmpeg (optional but recommended for video processing)
- API Keys:
  - OpenAI API Key
  - FAL.ai API Key

Step-by-Step Installation

1. Clone the Repository

```bash
git clone https://github.com/[username]/ai-scene-maker-3-models.git
cd ai-scene-maker-3-models
```

2. Create a Virtual Environment (Recommended)

```bash
# Windows
python -m venv venv
venv\Scripts\activate

# macOS/Linux
python3 -m venv venv
source venv/bin/activate
```

3. Install Dependencies

```bash
pip install -r requirements.txt
```

4. Install FFmpeg (Optional but Recommended)

Windows:

1. Download FFmpeg from ffmpeg.org
2. Extract the archive
3. Add the `bin` folder to your system PATH
4. Verify: `ffmpeg -version`

macOS:

```bash
brew install ffmpeg
```

Linux (Ubuntu/Debian):

```bash
sudo apt update
sudo apt install ffmpeg
```

5. Set Up API Keys

Create a `.env` file in the root directory:

```bash
cp .env.example .env
```

Edit `.env` and add your API keys:

```env
# API Keys
OPENAI_API_KEY=your_openai_api_key_here
FAL_API_KEY=your_fal_api_key_here

# Optional Configuration
OUTPUT_DIR=./outputs
MAX_RETRIES=3
DEFAULT_RESOLUTION=720p
DEFAULT_INFERENCE_STEPS=40
DEFAULT_SAFETY_CHECKER=False
```

6. Run the Application

```bash
python app.py
```

The application will launch in your browser at `http://localhost:7860`

Troubleshooting

Issue: Missing API Keys Error

- Ensure your `.env` file exists and contains valid API keys
- Restart the application after adding keys

Issue: FFmpeg Not Found

- The app will still work but video stitching may be limited
- Install FFmpeg following the steps above

Issue: ImportError

- Ensure you're in the virtual environment
- Run `pip install -r requirements.txt` again

📂 File Structure Breakdown

Root Files

`app.py`

- **Purpose**: Main entry point for the application
- **Key Functions**:
  - Sets up logging configuration with custom filters
  - Suppresses verbose HTTP request logs
  - Creates and launches the Gradio UI

`config.py`

- **Purpose**: Central configuration management
- **Key Functions**:
  - Loads environment variables from `.env`
  - Validates required API keys
  - Sets default values for application settings
  - Creates output directories

`requirements.txt`

- **Purpose**: Lists all Python dependencies
- **Key Packages**:
  - `gradio`: Web UI framework
  - `fal-client`: FAL.ai API client
  - `openai`: OpenAI API client

- `langchain`: LLM orchestration
- `moviepy`: Video processing
- `opencv-python`: Computer vision operations

UI Module (`ui/`)

`gradio_ui.py`

- **Purpose**: Complete UI implementation
- **Key Components**:
  - `create_ui()`: Main UI builder with tabs for generation and API setup
  - `on_image_upload()`: Handles image analysis when uploaded
  - `ui_start_chain_generation()`: Manages the video generation process
  - `start_chain_generation_with_updates()`: Core chain generation logic
  - Progress tracking and real-time updates
  - Gallery display for individual video chains

Utils Module (`utils/`)

`fal_client.py`

- **Purpose**: Interface with FAL.ai API
- **Key Functions**:
  - `generate_video_from_image()`: Creates videos from images using various models (WAN, Pixverse, LUMA, Kling)
  - `upload_file()`: Uploads images to FAL.ai storage
  - `download_video()`: Downloads generated videos
  - Model-specific parameter handling

`openai_client.py`

- **Purpose**: Interface with OpenAI API
- **Key Functions**:
  - `analyze_image_structured()`: Extracts theme, background, subject, tone, and action
  - `image_to_text()`: Basic image description
  - `generate_scene_vision()`: Creates cohesive scene descriptions
  - `determine_optimal_chain_count()`: Auto-calculates video chains needed

`langchain_prompts.py`

- **Purpose**: Advanced prompt generation using LangChain
- **Key Functions**:

- `generate_cinematic_prompt()`: Creates cinematic prompts maintaining continuity
- Considers story phases (Establishing, Setup, Development, Resolution)
- Implements cinematography techniques

`video_processing.py`

- **Purpose**: Video and image processing operations
- **Key Functions**:
  - `extract_simple_last_frame()`: Gets the last frame for continuity
  - `auto_adjust_image()`: Fixes brightness/saturation issues
  - `stitch_videos()`: Combines multiple videos using FFmpeg or moviepy
  - `enhance_frame_quality()`: Improves frame quality between chains

`helpers.py`

- **Purpose**: General utility functions
- **Key Functions**:
  - File management utilities
  - Logging helpers
  - Filename sanitization

🎯 Character Consistency Improvements

Current Challenges

The main issue is that each video generation through FAL.ai may interpret the character differently, leading to inconsistencies across the chain.

Proposed Solutions

1. Enhanced Frame Extraction with Character Detection

python

```python
# Add to video_processing.py
def extract_character_region(frame_path, character_description):
    """
    Use object detection to identify and crop the main character
    """
    # Implement using YOLOv8 or similar for person detection
    # Then use OpenAI Vision to verify it matches character_description
    pass

def create_character_reference_sheet(video_paths, character_description):
    """
    Extract best character shots from all videos to create a reference
    """
    character_frames = []
    for video in video_paths:
        frames = extract_character_frames(video, character_description)
        character_frames.extend(frames)

    # Create a collage of character appearances
    reference_sheet = create_collage(character_frames)
    return reference_sheet
```

2. Prompt Engineering for Character Consistency

python

```python
# Modify langchain_prompts.py
def generate_cinematic_prompt_with_character_lock(
    action_direction, scene_vision, frame_description,
    character_details, current_chain, total_chains
):
    # Add explicit character descriptors
    character_lock = f"""
    CRITICAL CHARACTER CONSISTENCY:
    - Exact appearance: {character_details['appearance']}
    - Clothing: {character_details['clothing']}
    - Distinctive features: {character_details['features']}
    - Must match EXACTLY the character from the previous frame
    """
    # Include in prompt generation
```

3. Post-Processing Character Validation

python

```python
# Add to utils/openai_client.py
def validate_character_consistency(frame1_path, frame2_path, character_description):
    """
    Compare two frames to ensure character consistency
    """
    # Use OpenAI Vision to compare characters
    prompt = f"""
    Compare the main character in these two images.
    Character should be: {character_description}

    Rate consistency from 1-10 and list any differences.
    """
    # Return consistency score and differences
```

4. Adaptive Frame Selection

python

```python
# Enhance video_processing.py
def extract_best_character_frame(video_path, character_description, previous_frame=None):
    """
    Extract frame with best character representation
    """
    frames = extract_top_frames(video_path, num_frames=10)

    if previous_frame:
        # Score frames based on character similarity to previous
        scores = []
        for frame in frames:
            score = calculate_character_similarity(frame, previous_frame, character_descriptior
            scores.append(score)

        best_idx = scores.index(max(scores))
        return frames[best_idx]
    else:
        # Use quality metrics for first frame
        return select_highest_quality_frame(frames)
```

🤦 Face Replacement Options

1. Face Restoration Post-Processing

python

```python
# New file: utils/face_enhancement.py
import cv2
from gfpgan import GFPGANer

class FaceEnhancer:
    def __init__(self):
        self.restorer = GFPGANer(
            model_path='GFPGANv1.4.pth',
            upscale=2,
            arch='clean',
            device='cpu'
        )

    def enhance_faces_in_video(self, video_path, output_path):
        """
        Process video frame by frame to enhance faces
        """
        cap = cv2.VideoCapture(video_path)
        # Process and write enhanced frames
```

2. Reference Face Injection

python

```python
# Add to utils/face_replacement.py
class FaceConsistencyManager:
    def __init__(self, reference_image_path):
        self.reference_face = self.extract_face(reference_image_path)

    def extract_face(self, image_path):
        """Extract face region using MediaPipe or dlib"""
        pass

    def create_face_prompt_modifier(self):
        """
        Generate prompt modifications to maintain face consistency
        """
        return "maintaining exact facial features from the reference"

    def validate_face_match(self, frame_path, threshold=0.8):
        """
        Check if face in frame matches reference
        """
        frame_face = self.extract_face(frame_path)
        similarity = self.calculate_face_similarity(self.reference_face, frame_face)
        return similarity > threshold
```

3. Face Swapping Integration

python

```python
# Using InsightFace for post-processing
class FaceSwapper:
    def __init__(self):
        self.swapper = insightface.model_zoo.get_model('inswapper_128.onnx')

    def swap_faces_in_video(self, video_path, reference_face_path, output_path):
        """
        Replace all faces in video with reference face
        """
        # Implementation for face swapping
        pass
```

🎨 Additional Enhancement Methods

1. Style Consistency Network

python

```python
# utils/style_consistency.py
class StyleConsistencyEnhancer:
    def __init__(self):
        self.style_reference = None

    def extract_style_features(self, image_path):
        """Extract color palette, lighting, and style features"""
        img = cv2.imread(image_path)

        # Color histogram
        color_hist = self.calculate_color_histogram(img)

        # Lighting analysis
        lighting = self.analyze_lighting(img)

        # Texture features
        texture = self.extract_texture_features(img)

        return {
            'color_palette': color_hist,
            'lighting': lighting,
            'texture': texture
        }

    def create_style_lut(self, reference_frames):
        """Create Look-Up Table for color grading"""
        pass

    def apply_style_transfer(self, video_path, style_reference):
        """Apply consistent style across all frames"""
        pass
```

2. Temporal Smoothing

python

```python
# utils/temporal_smoothing.py
def smooth_transitions(video_paths, output_path):
    """
    Add smooth transitions between video segments
    """
    clips = []

    for i, path in enumerate(video_paths):
        clip = VideoFileClip(path)

        if i > 0:
            # Add crossfade transition
            clip = clip.crossfadein(0.5)

        if i < len(video_paths) - 1:
            clip = clip.crossfadeout(0.5)

        clips.append(clip)

    final = concatenate_videoclips(clips, method="compose")
    final.write_videofile(output_path)
```

3. Motion Interpolation

python

```python
# utils/motion_enhancement.py
class MotionInterpolator:
    def __init__(self):
        self.flow_model = cv2.optflow.createOptFlow_DeepFlow()

    def interpolate_between_chains(self, video1_path, video2_path):
        """
        Create smooth motion transition between chain videos
        """
        # Extract last frame of video1 and first frame of video2
        # Generate interpolated frames using optical flow
        # Return transition frames
        pass
```

4. Quality Metrics Dashboard

python

```python
# utils/quality_metrics.py
class QualityAnalyzer:
    def analyze_chain_quality(self, video_paths):
        """
        Comprehensive quality analysis of the chain
        """
        metrics = {
            'character_consistency': [],
            'style_consistency': [],
            'motion_smoothness': [],
            'resolution_quality': []
        }

        for i in range(len(video_paths) - 1):
            # Analyze transitions between videos
            metrics['character_consistency'].append(
                self.check_character_consistency(video_paths[i], video_paths[i+1])
            )
            # ... other metrics

        return metrics

    def generate_quality_report(self, metrics):
        """Create visual report of quality metrics"""
        pass
```

5. Automated Quality Control

python

```python
# utils/quality_control.py
class QualityController:
    def __init__(self, thresholds):
        self.thresholds = thresholds

    def validate_chain_video(self, video_path, previous_video=None):
        """
        Validate video meets quality standards
        """
        checks = {
            'resolution': self.check_resolution(video_path),
            'brightness': self.check_brightness(video_path),
            'sharpness': self.check_sharpness(video_path),
            'artifacts': self.check_artifacts(video_path)
        }

        if previous_video:
            checks['continuity'] = self.check_continuity(previous_video, video_path)

        return all(checks.values()), checks

    def suggest_corrections(self, failed_checks):
        """Suggest parameter adjustments for failed checks"""
        pass
```

Implementation Priority

1. **High Priority**:
   - Character consistency validation
   - Enhanced prompt engineering for character locking
   - Post-processing face enhancement
2. **Medium Priority**:
   - Style consistency LUT
   - Temporal smoothing
   - Quality metrics dashboard
3. **Future Enhancements**:
   - Face swapping integration
   - Motion interpolation
   - Advanced AI upscaling

These improvements would significantly enhance the consistency and quality of the generated video chains while working within the constraints of the FAL.ai API system.