The Most Polish Landscape (TMPL) - Complete System Documentation

December 31, 2024

Contents

Complete System Documentation

System Overview

Purpose

The Most Polish Landscape is an interactive art installation that creates real-time Polish landscapes based on viewer interaction. The system combines depth sensing, real-time image generation, and high-quality display technologies to create an immersive experience.

System Components

1. **Position Tracking (Viewer Detection)**
   - Real-time depth sensing using OAK-D camera
   - Viewer position analysis and tracking
   - State management and logging
2. **Mask Generation**
   - Processing of viewer position data
   - Static and dynamic mask combination
   - Real-time state monitoring
3. **Image Rendering**
   - High-performance image synthesis
   - Multi-GPU processing
   - Real-time image generation
4. **Display Management**
   - High-resolution display output
   - Smooth transitions and effects
   - Multi-monitor support

System Architecture

Component Flow

```
graph LR
    A[Position Tracking] -- Position Data --> B[Mask Generator]
    B -- Mask Files --> C[Renderer]
    C -- Generated Images --> D[Display Manager]
    D -- Visual Output --> E[Display]
```

Data Pipeline

1. **Input Stage (Position Tracking)**
   - Depth data acquisition
   - Position analysis
   - State logging
2. **Processing Stage (Mask Generation)**
   - Mask combination
   - State processing
   - File generation
3. **Rendering Stage**
   - Image synthesis
   - Multi-GPU processing
   - Output generation
4. **Display Stage**
   - Image sequence management
   - Transition handling
   - Display output

Component Integration

1. Position Tracking → Mask Generation

Data Flow

```python
# Position Tracking output (tmpl.log)
[counter1, counter2, ..., counter10]

# Mask Generator input
def process_state(state: List[int]):
    """Process viewer position state"""
    for counter in state:
        # Generate corresponding masks
```

Key Integration Points

- File-based state communication
- Real-time state monitoring
- Synchronized processing

2. Mask Generation → Renderer

Data Flow

```python
# Mask Generation output
processed_mask = cv2.imwrite(f"{next_index}.bmp", final_mask)

# Renderer input
def load_masks_batch(file_nums, input_dir, output_dir):
    """Load and process mask batch"""
    masks = []
    for file_num in file_nums:
        mask = cv2.imread(f"{file_num}.bmp", cv2.IMREAD_GRAYSCALE)
```

Key Integration Points

- File system synchronization
- Batch processing coordination
- Memory management

3. Renderer → Display Manager

Data Flow

```python
# Renderer output
cv2.imwrite(output_path, img, [cv2.IMWRITE_JPEG_QUALITY, 95])

# Display Manager input
def load_image(self, path, size, keep_aspect=True):
    """Load and prepare image for display"""
    image = Image.open(path)
    texture = self.create_texture_from_surface(surface)
```

Key Integration Points

- Synchronized file access
- Image format compatibility
- Real-time processing

Data Flow

Complete Processing Pipeline

1. **Viewer Detection**

   ```python
   # Position Tracking
   def analyze_columns(self, distances, mirror=True):
       """Detect viewer presence in columns"""
       column_presence = self.column_analyzer.analyze_columns(distances)
       self.depth_tracker.update(column_presence)
   ```

2. **Mask Generation**

   ```python
   # Mask Processing
   def process_and_save(self, state: Dict[int, List[Tuple[int, int]]]):
       """Generate masks from state"""
   ```

```python
    final_mask = self.combine_masks(masks, gray_indexes)
    cv2.imwrite(result_path, final_mask)
```

3. **Image Rendering**

```python
# Image Generation
def process_batch(model, file_nums, input_dir, output_dir, gpu_id):
    """Generate images from masks"""
    with torch.cuda.amp.autocast():
        generated = model(data_i, mode='inference')
```

4. **Display Management**

```python
# Display Processing
def _handle_image_sequence(self):
    """Handle image sequence display"""
    self._render_frame_with_overlay(texture)
```

Hardware Requirements

Complete System Requirements

1. **Position Tracking**
   - OAK-D camera
   - USB 3.2 port
   - Processing unit for depth analysis
2. **Mask Generation**
   - Multi-core CPU
   - 64GB+ RAM
   - Fast storage for mask processing
3. **Rendering**
   - NVIDIA A6000 ADA, A100, or H100 GPU
   - 32GB+ VRAM per GPU
   - NVMe SSD storage
4. **Display**

- Display support for 3840x1200 resolution
- Graphics card with hardware acceleration
- Multi-monitor support

Installation and Setup

Complete System Setup

1. **Position Tracking Setup**

```bash
# Install position tracking
git clone https://github.com/speplinski/tmpl-viewer-position-tracking.git
cd tmpl-viewer-position-tracking
pip install -r requirements.txt
```

2. **Mask Generator Setup**

```bash
# Install mask generator
git clone https://github.com/speplinski/tmpl-generator.git
cd tmpl-generator
pip install -e .
```

3. **Renderer Setup**

```bash
# Install renderer
git clone https://github.com/speplinski/tmpl-benchmark-app.git
cd tmpl-benchmark-app
pip install -r requirements.txt
```

4. **Display Manager Setup**

```bash
# Install display manager
git clone https://github.com/speplinski/tmpl-app.git
cd tmpl-app
pip install -e .
```

System Configuration

1. **Position Tracking Configuration**

```python
# config.py
MIN_THRESHOLD = 0.4  # Detection distance
MAX_THRESHOLD = 1.8
nH = 10  # Horizontal divisions
```

2. **Mask Generator Configuration**

```json
{
    "static_masks": {
        "10": 1,
        "50": 3
    },
    "sequence_masks": {
        "35": 2
    }
}
```

3. **Renderer Configuration**

```
python gen.py --batchSize 4 --gpu_ids 0,1
```

4. **Display Configuration**

```python
final_resolution = (3840, 1200)
final_resolution_model = (3840, 1280)
```

Operations Guide

Starting the System

1. Start Position Tracking:

```
python main.py  # In tmpl-viewer-position-tracking
```

2. Start Mask Generator:

```
python main.py P1100142  # In tmpl-generator
```

3. Start Renderer:

```
python gen.py --gpu_ids 0,1  # In tmpl-benchmark-app
```

4. Start Display Manager:

```
tmpl-app --monitor 1  # In tmpl-app
```

Monitoring

1. **Position Tracking**
    - Real-time depth heatmap
    - Column presence indicators
    - FPS counter
2. **Mask Generation**
    - Memory usage statistics
    - Processing status
    - File generation logs
3. **Renderer**
    - GPU utilization
    - Processing times
    - Batch statistics
4. **Display**
    - Playback statistics
    - Frame rates
    - Memory usage

Maintenance

System Monitoring

1. **Performance Monitoring**
    - Track processing times across components

- Monitor memory usage
- Check GPU utilization

2. **Error Handling**
   - Log file monitoring
   - Error recovery procedures
   - Component restart protocols
3. **Resource Management**
   - Storage cleanup
   - Memory optimization
   - Cache management

Troubleshooting Guide

1. **Position Tracking Issues**
   - Check camera connection
   - Verify lighting conditions
   - Monitor USB bandwidth
2. **Mask Generation Issues**
   - Verify file permissions
   - Check disk space
   - Monitor memory usage
3. **Rendering Issues**
   - Check GPU status
   - Monitor VRAM usage
   - Verify NCCL setup
4. **Display Issues**
   - Check monitor connection
   - Verify resolution settings
   - Monitor frame rates

Position Tracking

Overview

Purpose

The TMPL Position Tracking module is a real-time viewer position tracking system using the OAK-D camera. It's designed to detect and track viewer presence in The Most Polish Landscape installation, enabling interactive responses to viewer movements and positions.

Key Features

- Real-time depth sensing and position tracking
- Terminal-based visualization with heatmap display
- OpenCV window support for visual debugging
- Configurable detection zones and thresholds
- Real-time state logging for installation interaction
- Memory-efficient processing
- Visual feedback through heatmap display
- Position state persistence

Technical Specifications

Hardware Requirements

- OAK-D camera
- Computing device with USB 3.2 support
- Processor with support for real-time processing

Software Requirements

- Python 3.8 or newer
- Operating System: Linux or macOS
- Required packages:

- depthai 2.21.1
- numpy 1.21.0
- opencv-python 4.7.0

System Architecture

Core Components

1. Depth Sensing System

- OAK-D camera integration
- Stereo depth calculation
- Spatial location processing

2. Analysis System

- Column-based presence detection
- Position tracking
- State management

3. Visualization System

- Terminal-based UI
- OpenCV window support
- Real-time heatmap generation

Class Documentation

Core Classes

DepthApplication Main application class coordinating all components.

```python
class DepthApplication:
    def __init__(self):
        self.config = Config()
        self.depth_tracker = DepthTracker()
```

```python
        self.visualizer = Visualizer(self.config)
        self.column_analyzer = ColumnAnalyzer(self.config)
```

Key Methods

- `create_pipeline()`: OAK-D pipeline setup
- `run()`: Main application loop
- `handle_key()`: User input processing
- `update_stats()`: Statistics display update

`ColumnAnalyzer` Analyzes depth data to detect presence in columns.

```python
class ColumnAnalyzer:
    def __init__(self, config):
        self.config = config

    def analyze_columns(self, distances, mirror=True):
        """Analyze columns for object presence"""
```

`DepthTracker` Tracks viewer positions and manages state logging.

```python
class DepthTracker:
    def __init__(self):
        self.threshold_time = 3.0
        self.increment_interval = self.config.COUNTER_INCREMENT_INTERVAL
        self.position_timers = {}
        self.position_counters = [0] * 10
```

Key Methods

- `update()`: State update and logging
- `log_state()`: State persistence

`Visualizer` Handles visual output and heatmap generation.

```python
class Visualizer:
    def __init__(self, config):
        self.config = config

    def create_heatmap(self, distances, mirror=True):
        """Create OpenCV heatmap"""

    def create_console_heatmap(self, distances, prev_buffer, mirror=True):
        """Create terminal-based heatmap"""
```

Configuration Class

`Config` Application configuration and constants.

```python
class Config:
    def __init__(self):
        # Distance thresholds
        self.MIN_THRESHOLD = 0.4  # 40 cm
        self.MAX_THRESHOLD = 1.8  # 1.8 meters

        # Display configuration
        self.DISPLAY_WINDOW = False
        self.SHOW_STATS = True
        self.MIRROR_MODE = True

        # Grid dimensions
        self.nH = 10  # Horizontal divisions
        self.nV = 6   # Vertical divisions
```

Utility Classes

`TerminalUtils` Terminal management utilities.

```python
class TerminalUtils:
    @staticmethod
    def init_terminal():
        """Terminal initialization"""

    @staticmethod
    def get_key():
        """Non-blocking key input"""

    @staticmethod
    def move_cursor(x: int, y: int):
        """Cursor positioning"""
```

Technical Processes

Depth Processing Pipeline

1. **Camera Setup**

```python
def create_pipeline(self):
    pipeline = dai.Pipeline()
    # Configure stereo cameras
    monoLeft = pipeline.create(dai.node.MonoCamera)
    monoRight = pipeline.create(dai.node.MonoCamera)
    stereo = pipeline.create(dai.node.StereoDepth)
```

2. **Spatial Analysis**

```python
def analyze_columns(self, distances, mirror=True):
    heatmap = np.array(distances).reshape(self.config.nV, self.config.nH)
    mask = (heatmap >= self.config.MIN_THRESHOLD) &
           (heatmap <= self.config.MAX_THRESHOLD)
```

3. **Position Tracking**

```python
def update(self, column_presence):
    current_time = time.time()
    for i in range(10):
        if column_presence[i] == 1:
            # Track presence and update counters
```

Visualization System

1. **Console Heatmap**

```python
def create_console_heatmap(self, distances, prev_buffer, mirror=True):
    heatmap = np.array(distances).reshape(self.config.nV, self.config.nH)
    # Generate visual representation
```

2. **OpenCV Heatmap**

```python
def create_heatmap(self, distances, mirror=True):
    heatmap = np.array(distances).reshape(self.config.nV, self.config.nH)
    # Apply color mapping and scaling
```

Installation and Setup

Basic Installation

```bash
# Clone the repository
git clone https://github.com/speplinski/tmpl-viewer-position-tracking.git
cd tmpl-viewer-position-tracking

# Create virtual environment
python -m venv venv
source venv/bin/activate

# Install requirements
pip install -r requirements.txt
```

Hardware Setup

1. Connect OAK-D camera via USB 3.0
2. Position camera for optimal viewing angle
3. Ensure proper lighting conditions
4. Verify camera access permissions

Usage Guide

Basic Operation

```
# Start the application
python main.py
```

Control Keys

- 'q'- Exit application
- 'w'- Toggle OpenCV window
- 's'- Toggle statistics
- 'm'- Toggle mirror mode

Display Elements

- Real-time depth heatmap
- FPS counter
- Mirror mode status
- Column presence indicators
- Position counters

Data Output

- State changes logged to `tmpl.log`
- Array format: `[counter1, counter2, ..., counter10]`
- Real-time updates on position changes

Maintenance and Troubleshooting

Common Issues

1. Camera Access
   - Verify USB connection
   - Check device permissions
   - Monitor system resources
2. Performance Issues
   - Check USB bandwidth
   - Monitor CPU usage
   - Verify lighting conditions
3. Detection Problems
   - Adjust distance thresholds
   - Check camera positioning
   - Verify lighting conditions

Performance Optimization

- Monitor FPS for optimal performance
- Adjust grid dimensions if needed
- Balance detection sensitivity

Generator

Overview

Purpose

The TMPL Generator is a specialized system designed to process and manage image masks for The Most Polish Landscape project. It handles both static masks and dynamic sequences of panoramic images, providing real-time processing capabilities for the interactive art installation.

Key Features

- Processing of static masks and dynamic sequence frames
- Real-time state monitoring
- Automatic mask combining with proper layering
- Memory usage monitoring and optimization
- Support for multiple image formats (PNG, BMP)
- Configurable mask mappings
- Dynamic configuration based on directory content

Technical Specifications

System Requirements

Hardware Requirements

- CPU: Multi-core processor recommended
- Storage: Sufficient space for mask storage and processing
- Display: Support for high-resolution image processing

Software Requirements

- Python 3.7 or higher
- Required packages:
    - numpy: Array processing

  - opencv-python: Image processing
  - psutil: System monitoring

System Architecture

Core Components

1. Monitoring System

- File state monitoring
- System resource tracking
- Real-time updates handling

2. Mask Processing

- Static mask management
- Sequence frame handling
- Binary mask operations

3. Configuration Management

- Dynamic configuration generation
- Mask mapping management
- Directory structure handling

Class Documentation

Core Classes

TMPLMonitor Main monitoring system coordinating all operations.

```python
class TMPLMonitor:
    def __init__(self, panorama_id: str, mask_configs: List[MaskConfig]):
        self.panorama_id = panorama_id
        self.base_paths = {
            'base': Path(f'./landscapes/{panorama_id}'),
```

```
            'sequences': Path(f'./landscapes/{panorama_id}/sequences'),
            'output': Path(f'./landscapes/{panorama_id}'),
            'results': Path('./results')
        }
```

Key Methods

- `run()`: Main monitoring loop
- `process_state()`: State processing
- `_initialize_system()`: System initialization

`MaskManager` Handles mask loading, caching, and processing operations.

```
class MaskManager:
    def __init__(self, config: MaskConfig, panorama_id: str, base_paths: Dict[str, Path]):
        self.config = config
        self.panorama_id = panorama_id
        self.base_paths = base_paths
        self.mask_cache = {}
        self.sequence_frames = {}
```

Key Methods

- `load_static_masks()`: Static mask loading
- `load_sequence_frames()`: Sequence frame loading
- `process_and_save()`: Mask processing and saving
- `get_frame()`: Frame retrieval

`ImageProcessor` Handles image loading and processing operations.

```
class ImageProcessor:
    TARGET_SIZE = (1280, 3840)
    BINARY_THRESHOLD = 127
```

```python
    @staticmethod
    def load_and_resize_image(image_path: Path) -> Optional[np.ndarray]:
        """Load and resize image to standard dimensions"""

    @staticmethod
    def combine_masks(masks: Dict[int, np.ndarray], gray_indexes: Dict[int, int]) -> np.ndar
        """Combine multiple masks with proper indexing"""
```

FileMonitor Handles file monitoring and state reading.

```python
class FileMonitor:
    def __init__(self, filename: str = LOG_FILENAME):
        self.filename = filename
        self.last_modified = 0
        self.last_state = None
```

Key Methods

- `get_last_state()`: State reading
- `check_for_updates()`: Update monitoring

Configuration Classes

MaskConfig Configuration for mask types and mappings.

```python
@dataclass
class MaskConfig:
    name: str
    gray_values: List[int]
    gray_indexes: Dict[int, int]
```

Utility Classes

SystemMonitor System resource monitoring.

```python
class SystemMonitor:
    @staticmethod
    def get_memory_usage() -> float:
        """Returns memory usage in MB"""

    @staticmethod
    def print_memory_status():
        """Prints current memory status"""
```

Technical Processes

Mask Processing Pipeline

1. **Static Mask Loading**

```python
def load_static_masks(self):
    for gray_value in self.config.gray_values:
        mask_path = self.base_paths['base'] / f"{self.panorama_id}_{gray_value}.bmp"
        if mask_path.exists():
            mask = ImageProcessor.load_and_resize_image(mask_path)
            if mask is not None:
                self.mask_cache[gray_value] = mask
```

2. **Sequence Frame Loading**

```python
def load_sequence_frames(self) -> int:
    for gray_value in self.config.gray_values:
        gray_dir = self.base_paths['base'] / f"{self.panorama_id}_{gray_value}"
        if gray_dir.exists():
            # Process sequence directories
            for seq_dir in seq_dirs:
                # Load and process frames
                frame = ImageProcessor.load_and_resize_image(frame_path)
```

3. **Mask Combination**

```python
def process_and_save(self, state: Dict[int, List[Tuple[int, int]]]) -> Optional[Path]:
    final_mask = np.full(target_size, 255, dtype=np.uint8)
    for gray_value in sorted_gray_values:
        if gray_value in self.mask_cache:
            # Combine static and sequence masks
            # Apply proper indexing
```

Memory Management

1. **Resource Monitoring**

```python
def print_memory_status():
    memory_mb = SystemMonitor.get_memory_usage()
    total_memory = psutil.virtual_memory().total / 1024 / 1024
    print(f"Memory usage: {memory_mb:.1f} MB")
```

2. **Efficient Image Processing**

```python
def load_and_resize_image(image_path: Path) -> Optional[np.ndarray]:
    # Load and process images efficiently
    image = cv2.imread(str(image_path), cv2.IMREAD_GRAYSCALE)
    # Resize and optimize
```

Configuration System

Mask Mapping Configuration

```json
{
    "P1100142": {
        "static_masks": {
            "10":  1,
            "50":  3
        },
        "sequence_masks": {
            "35":  2,
```

```
            "170": 6
        }
    }
}
```

Dynamic Configuration Generation

```python
def create_dynamic_config(panorama_id: str) -> Dict:
    gray_values, gray_indexes = scan_directory(panorama_id)
    return {
        'name': "dynamic",
        'gray_values': gray_values,
        'gray_indexes': gray_indexes
    }
```

Installation and Setup

Basic Installation

```bash
# Clone the repository
git clone https://github.com/speplinski/tmpl-generator.git
cd tmpl-generator

# Install the package
pip install -e .
```

System Dependencies

```bash
# Install required packages
pip install numpy opencv-python psutil
```

Usage Guide

Basic Usage

```
# Run with specific panorama ID
python main.py P1100142
```

Directory Structure

```
tmpl_generator/
  landscapes/         # Panorama data
  results/            # Processed masks
  mask_mapping.json   # Configuration
  tmpl.log            # State log
```

Maintenance and Troubleshooting

Common Issues

1. Memory Management
   - Monitor system resources
   - Check image dimensions
   - Verify mask cleanup
2. File Processing
   - Verify file permissions
   - Check file formats
   - Validate directory structure
3. Performance Optimization
   - Buffer size adjustment
   - Process monitoring
   - Resource allocation

Performance Monitoring

The system includes built-in monitoring: - Memory usage tracking - Processing time measurement - Resource utilization statistics

Renderer

Overview

Purpose

The TMPL Renderer is a high-performance semantic image synthesis system designed for The Most Polish Landscape installation. It evaluates and utilizes enterprise-grade GPUs to generate realistic landscape images from semantic masks in real-time.

Key Features

- Real-time semantic image synthesis
- Multi-GPU processing support
- High-performance batch processing
- Distributed processing capabilities
- Performance benchmarking
- Automated file handling and processing

Hardware Requirements

Primary supported GPUs: - NVIDIA A6000 ADA - NVIDIA A100 - NVIDIA H100

Performance Targets

- Minimum 4 images per second generation rate
- Real-time processing capability
- Efficient multi-GPU utilization
- Optimized memory usage

Technical Specifications

System Requirements

Hardware Requirements

- GPU: NVIDIA A6000 ADA, A100, or H100
- VRAM: Minimum 32GB per GPU
- RAM: 32GB minimum recommended
- Storage: NVMe SSD recommended for optimal I/O

Software Requirements

- Python 3.0+
- PyTorch 1.0+
- CUDA compatible with GPU
- Required packages:

```
torch>=1.0.0
torchvision
dominate>=2.3.1
dill
scikit-image
opencv-python
```

System Architecture

Core Components

1. Generator Pipeline

- Multi-GPU coordination
- Distributed processing
- Batch management
- Memory optimization

2. Model Management

- SPADE network architecture
- Synchronized batch normalization
- Distributed model handling
- Checkpoint management

3. File Processing

- Input mask handling
- Output image management
- File locking mechanism
- Batch processing coordination

Class Documentation

Core Classes

Pix2PixModel Main model class for image generation.

```
class Pix2PixModel(torch.nn.Module):
    def __init__(self, opt):
        self.netG, self.netD, self.netE = self.initialize_networks(opt)
```

Key Methods

- `forward()`: Generation pipeline
- `generate_fake()`: Image synthesis
- `initialize_networks()`: Network setup
- `preprocess_input()`: Data preparation

Distributed Processing System

```
def setup_distributed(gpu_id, world_size):
    """Setup distributed training environment"""
    os.environ['MASTER_ADDR'] = 'localhost'
```

```python
    os.environ['MASTER_PORT'] = '12355'
    dist.init_process_group(
        backend='nccl',
        init_method='env://',
        world_size=world_size,
        rank=gpu_id
    )
```

Batch Processing System

```python
def process_batch(model, file_nums, input_dir, output_dir, gpu_id):
    """Process a batch of images on specific GPU"""
    data_i = load_masks_batch(file_nums, input_dir, output_dir)
    with torch.cuda.amp.autocast():
        generated = model(data_i, mode='inference')
```

Configuration System

BaseOptions Configuration management class.

```python
class BaseOptions:
    def initialize(self, parser):
        parser.add_argument('--gpu_ids', type=str, default='0')
        parser.add_argument('--batchSize', type=int, default=1)
        parser.add_argument('--label_nc', type=int, default=182)
```

Performance Optimization

Memory Management

```python
def process_batch(model, file_nums, input_dir, output_dir, gpu_id):
    try:
        with torch.cuda.amp.autocast():
            generated = model(data_i, mode='inference')
```

```python
        # Clear memory
        del generated, data_i
        torch.cuda.empty_cache()
```

Multi-GPU Coordination

```python
def main():
    world_size = len(opt.gpu_ids)
    if world_size > 1:
        mp.spawn(process_on_gpu, args=(world_size, opt),
                nprocs=world_size)
```

Installation and Setup

Basic Installation

```bash
# Clone repository
git clone https://github.com/speplinski/tmpl-benchmark-app.git
cd tmpl-benchmark-app/

# Install requirements
pip install -r requirements.txt
```

Dataset Preparation

```bash
cd datasets/
./download_dataset.sh
cd ..
```

Model Setup

```bash
cd checkpoints
./download_ckpts.sh
cd ..
```

Usage Guide

Basic Usage

```
# Single GPU mode
python gen.py --batchSize 4 --gpu_ids 0

# Multi-GPU mode
python gen.py --batchSize 8 --gpu_ids 0,1
```

Performance Optimization Tips

1. Batch Size Optimization

   ```
   # For A100 GPU (40GB variant)
   python gen.py --batchSize 6 --gpu_ids 0

   # For multi-GPU setup
   python gen.py --batchSize 12 --gpu_ids 0,1
   ```

2. Memory Management

   - Monitor VRAM usage
   - Adjust batch size accordingly
   - Enable automatic mixed precision

3. Multi-GPU Setup

   - Ensure proper NCCL setup
   - Balance load across GPUs
   - Monitor GPU utilization

Performance Benchmarking

Expected Performance Metrics

1. Single GPU Performance (A100)

    - Batch Size: 4
    - Images per second: 4-5
    - Memory usage: ~32GB VRAM
2. Dual GPU Performance (A100)
    - Batch Size: 8 (4 per GPU)
    - Images per second: 8-10
    - Memory usage: ~32GB VRAM per GPU

Benchmarking Procedure

1. Initialization

```python
def benchmark_performance(model, batch_size, num_iterations):
    total_time = 0
    for i in range(num_iterations):
        start_time = time.time()
        process_batch(...)
        total_time += time.time() - start_time
```

2. Metrics Collection

    - Processing time per batch
    - Memory usage monitoring
    - GPU utilization tracking
    - I/O performance measurement

Maintenance and Troubleshooting

Common Issues

1. Memory Management
    - Out of memory errors
    - Memory fragmentation
    - Batch size optimization
2. Performance Issues

    - GPU utilization
    - I/O bottlenecks
    - Process coordination
3. Multi-GPU Problems
    - NCCL configuration
    - Load balancing
    - Process synchronization

Performance Monitoring

Built-in monitoring provides: - Processing times - Memory usage - GPU utilization - Batch statistics

Acknowledgments

Based on: - pix2pixHD - SPADE - Synchronized Batch Normalization implementation

Application

Overview

Purpose

The Most Polish Landscape (TMPL) is an innovative interactive art installation that creates real-time Polish landscapes based on audience interaction. The project combines artificial intelligence (SPADE models) with interactive display technology to generate unique landscape interpretations.

Value Proposition

- Creates unique, AI-generated Polish landscapes in real-time
- Provides interactive art experience
- Delivers professional-grade visual quality
- Supports multi-display configurations for flexible installation

Key Features

- Real-time landscape visualization
- Smooth transitions between generated images
- Professional-grade display quality (4KE resolution)
- Multi-monitor support
- Automated sequence management
- Performance monitoring and statistics

Technical Specifications

System Requirements

Hardware Requirements

- Display: Support for 3840x1200 resolution
- GPU: Graphics card with hardware acceleration support
- Storage: Sufficient space for image sequences and video files

Software Requirements

- Operating System: Linux (Ubuntu recommended) or macOS
- Python 3.7 or higher
- SDL2 libraries and dependencies
- Required Python packages:
  - av (10.0.0): Video processing
  - numpy (1.24.0): Numerical computations
  - Pillow (10.0.0): Image processing
  - pysdl2 (0.9.16): Display and rendering
  - pysdl2-dll (2.28.0): SDL2 bindings

System Architecture

Core Components Overview

1. **Application Core (`main.py`)**
   - Main application loop
   - Event handling
   - State management
   - Rendering coordination
2. **SDL Application Layer (`sdl_app.py`)**
   - Window management
   - Renderer initialization
   - Multi-monitor support
   - Text rendering
3. **Texture Management (`texture_manager.py`)**
   - Image loading and processing
   - Texture creation and handling
   - Memory management
4. **Transition System (`transition_manager.py`)**
   - Smooth transitions between states
   - Fade effects management

- White transition handling

5. **Media Players**
   - Image Sequence Player (`image_sequence.py`)
   - Video Player (`video.py`)
6. **Statistics System (`playback_stats.py`)**
   - Performance monitoring
   - Frame rate tracking
   - Playback statistics

Class Documentation

Core Classes

`Application` Class Main application class coordinating all components.

```
class Application:
    def __init__(self, monitor_index):
        self.config = AppConfig()
        self.sdl_app = SDLApp(monitor_index)
        self.texture_manager = TextureManager(self.sdl_app.renderer)
        self.sequence_player = ImageSequencePlayer(self.config, self.texture_manager)
        self.transition_manager = TransitionManager(self.sdl_app.renderer, self.config)
        self.stats = PlaybackStatistics()
```

Key Methods

- `run()`: Main application loop
- `handle_events()`: Event processing
- `_handle_fade_transition()`: Transition management
- `_handle_video_playback()`: Video state handling
- `_handle_image_sequence()`: Image sequence management

`AppConfig` Class Configuration management and settings storage.

```python
class AppConfig:
    def __init__(self):
        self.sequences = [...]
        self.final_resolution = (3840, 1200)
        self.buffer_size = 12
        self.frame_step = 4
        self.source_fps = 0.5
        self.frames_to_interpolate = 30
```

Key Methods

- `get_current_sequence()`: Current sequence retrieval
- `next_sequence()`: Sequence advancement

Display and Rendering Classes

`SDLApp` Class SDL2 initialization and window management.

```python
class SDLApp:
    def __init__(self, monitor_index=1):
        self._init_sdl()
        self.window, self.renderer = self._create_window_and_renderer(monitor_index)
        self.font = self._init_font()
```

Key Methods

- `_create_window_and_renderer()`: Display setup
- `render_text()`: Text rendering
- `_init_font()`: Font initialization

`TextureManager` Class Texture loading and management.

```python
class TextureManager:
    def __init__(self, renderer):
        self.renderer = renderer
```

Key Methods

- `load_image()`: Image loading and scaling
- `create_texture_from_surface()`: Texture creation

Media Player Classes

`ImageSequencePlayer` Class Image sequence management and playback.

```
class ImageSequencePlayer:
    def __init__(self, config, texture_manager):
        self.config = config
        self.texture_manager = texture_manager
        self.frame_buffer = Queue(maxsize=config.buffer_size)
```

Key Methods

- `start_loader_thread()`: Background loading initiation
- `_buffer_loader_thread()`: Frame loading management
- `set_directory()`: Directory management

`VideoPlayer` Class Video playback management.

```
class VideoPlayer:
    def __init__(self, video_path, renderer, texture_manager=None):
        self.video_path = video_path
        self.renderer = renderer
        self.texture_manager = texture_manager
```

Key Methods

- `_init_video()`: Video decoder initialization
- `get_next_frame_texture()`: Frame extraction

Monitoring Classes

`PlaybackStatistics` Class Performance monitoring and statistics.

```python
class PlaybackStatistics:
    def __init__(self):
        self.playback_time = 0.0
        self.total_source_frames = 1
        self.total_displayed_frames = 1
```

Key Methods

- `format_stats()`: Statistics formatting
- `update_playback_time()`: Timing updates
- `start_playback()`: Playback control
- `pause_playback()`: Playback pause

Installation and Setup

Basic Installation

```bash
# Clone the repository
git clone https://github.com/speplinski/tmpl-app.git
cd tmpl-app

# Install the package
pip install -e .
```

System Dependencies

For Ubuntu/Debian:

```bash
sudo apt update
sudo apt install -y libsdl2-2.0-0
```

For macOS:

```
brew install sdl2
```

Asset Preparation

```
chmod +x download_assets.sh
./download_assets.sh
```

Usage Guide

Basic Usage

```
# Run on primary display
tmpl-app --monitor 0

# Run on secondary display
tmpl-app --monitor 1
```

Configuration

Configuration is managed through `app_config.py`. Key parameters include:

```
# Frame rates and interpolation
source_fps = 0.5
frames_to_interpolate = 30

# Buffer settings
buffer_size = 12
frame_step = 4

# Display settings
final_resolution = (3840, 1200)
final_resolution_model = (3840, 1280)
```

Maintenance and Troubleshooting

Common Issues and Solutions

1. Display Configuration
   - Monitor selection issues
   - Resolution compatibility
   - SDL2 installation problems
2. Performance Issues
   - Resource monitoring
   - Buffer optimization
   - Hardware acceleration verification
3. Asset Loading
   - Directory structure
   - File permissions
   - Format compatibility

Performance Monitoring

Built-in statistics display shows: - Playback duration - Frame rates (source and display) - Frame counts - Performance metrics