SpriteSheet Pipeline: A Plugin-Based System for Production-Ready 2D Sprite Generation in Stable Diffusion UIs

Open Source Community

January 21, 2026

Abstract

This paper presents a plugin-based pipeline for generating production-ready 2D spritesheets directly within Stable Diffusion user interfaces (ComfyUI and AUTOMATIC1111). The system automates the complete workflow from image generation through post-processing, including grid tiling, tile slicing, normalization, progressive downscaling, and palette reduction. The pipeline outputs engine-ready spritesheets with accompanying metadata, eliminating manual post-processing steps typically required for game development and digital art workflows. We describe the architecture, algorithms, and implementation details of both the minimum viable product (MVP) and proposed extensions, with a focus on open-source integration and practical deployment.

1 Introduction

Stable Diffusion has emerged as a powerful tool for generating digital artwork, with user interfaces like ComfyUI and AUTOMATIC1111 providing accessible workflows for both artists and developers. However, generating production-ready spritesheets for game engines and digital applications requires extensive post-processing that is currently performed manually or through separate tools. This paper documents a plugin-based extension that integrates directly into Stable Diffusion workflows to automate spritesheet generation.

The pipeline addresses the gap between AI-generated artwork and production-ready assets by implementing a complete processing chain: from initial image generation through final export. By operating as a plugin within existing Stable Diffusion interfaces, the system leverages established workflows while adding specialized spritesheet capabilities.

2 Problem Statement

Current workflows for creating spritesheets from AI-generated images involve multiple manual steps:

1. Generating individual sprite images using Stable Diffusion
2. Manually arranging images into a grid layout
3. Slicing the grid into individual tiles
4. Normalizing sprite dimensions and alignment
5. Downscaling to target resolutions while preserving quality
6. Reducing color palettes for engine compatibility
7. Exporting in formats compatible with game engines

Each step requires specialized tools and manual intervention, making the process time-consuming and error-prone. Additionally, maintaining consistency across spritesheets generated at different times or with different parameters becomes challenging.

The proposed pipeline automates these steps within the Stable Diffusion environment, providing a unified workflow that reduces manual effort and improves consistency. The system must handle variations in generated image quality, support multiple output formats, and integrate seamlessly with existing Stable Diffusion plugins and extensions.

3 Non-Goals

This paper explicitly excludes the following from the MVP scope:

- **3D sprite generation**: The pipeline focuses exclusively on 2D spritesheets
- **Animation generation**: Frame-by-frame animation creation is out of scope, though the pipeline can process pre-generated animation frames
- **Custom model training**: The system uses existing Stable Diffusion models without modification
- **Real-time generation**: The pipeline operates in batch mode, not real-time streaming
- **Multi-user collaboration**: The MVP is designed for single-user workflows
- **Cloud deployment**: Initial implementation targets local installations
- **Proprietary format support**: MVP focuses on open formats (PNG, JSON)

Future work may address these areas, but they are explicitly excluded from the initial implementation to maintain focus and deliverability.

4 User Workflow

The user workflow is designed to be intuitive and require minimal configuration:

4.1 Initial Setup

1. User installs the SpriteSheet Pipeline extension in their Stable Diffusion UI (ComfyUI or AUTOMATIC1111)
2. Extension registers as a custom node/workflow component
3. User configures default output directory and engine presets (optional)

4.2 Generation Workflow

1. User opens Stable Diffusion UI and navigates to the SpriteSheet Pipeline component
2. User provides input parameters:
   - Prompt for sprite generation (e.g., "a red mailbox on white background")
   - Number of sprites to generate (grid dimensions: $n \times m$)
   - Target sprite dimensions (e.g., 64×64 pixels)
   - Optional: ControlNet input for pose/structure guidance
   - Optional: img2img source for style transfer
3. User initiates generation
4. Extension executes the complete pipeline (detailed in Section 7)
5. Output files are written to the specified directory:

- `spritesheet.png`: Final spritesheet image
- `manifest.json`: Metadata including tile positions, dimensions, and generation parameters
- Optional: Engine-specific preset files (Unity, Godot, etc.)

6. User reviews output and can iterate with adjusted parameters

4.3 Integration Points

The extension integrates at two key points:

Generation Stage Intercepts or wraps Stable Diffusion's txt2img/img2img APIs to generate base images

Post-Processing Stage Processes generated images through the spritesheet pipeline before final output

This design allows the extension to work alongside other Stable Diffusion plugins without conflicts.

5 Inputs and Outputs

5.1 Input Specifications

The pipeline accepts the following inputs:

Parameter	Type	Description
`prompt`	string	Text prompt for Stable Diffusion generation
`negative_prompt`	string	Negative prompt for exclusion guidance
`grid_width`	integer	Number of sprites per row (n)
`grid_height`	integer	Number of sprite rows (m)
`target_width`	integer	Target sprite width in pixels
`target_height`	integer	Target sprite height in pixels
`controlnet_image`	image (opt.)	ControlNet guidance image
`img2img_source`	image (opt.)	Source image for img2img mode
`downscale_levels`	integer	Number of progressive downscale passes
`palette_size`	integer	Target color palette size (0 = no reduction)
`engine_preset`	string (opt.)	Target engine (unity, godot, generic)

Table 1: Input Parameters

5.2 Output Specifications

The pipeline produces the following outputs:

File	Description
`spritesheet.png`	Final spritesheet image with all sprites arranged in grid
`manifest.json`	Metadata including tile coordinates, dimensions, generation parameters
`engine_preset.json`	Engine-specific configuration (if preset specified)

Table 2: Output Files

The manifest.json structure:

```
{
  "version": "1.0",
  "grid_dimensions": {"width": 4, "height": 4},
```

```
  "sprite_dimensions": {"width": 64, "height": 64},
  "spritesheet_dimensions": {"width": 256, "height": 256},
  "tiles": [
    {
      "id": 0,
      "x": 0,
      "y": 0,
      "width": 64,
      "height": 64,
      "generation_params": {...}
    },
    ...
  ],
  "generation_timestamp": "2024-01-01T00:00:00Z",
  "pipeline_version": "1.0.0"
}
```

6 System Architecture

The pipeline is architected as a modular plugin system with clear separation of concerns:

Plugin Interface Layer Handles integration with ComfyUI/AUTOMATIC1111 APIs, manages user interface components, and coordinates workflow execution

Generation Module Interfaces with Stable Diffusion's generation APIs (txt2img, img2img) and optional ControlNet integration

Processing Pipeline Core processing modules: rasterization, slicing, normalization, downscaling, palette reduction

Export Module Handles file output, manifest generation, and engine preset formatting

Configuration Manager Manages user settings, presets, and pipeline parameters

The architecture follows a pipeline pattern where each stage processes data and passes it to the next stage. This design enables:

- Easy extension with additional processing stages
- Independent testing of each module
- Parallel processing where applicable (e.g., tile normalization)
- Graceful error handling at stage boundaries

Communication between modules uses standardized data structures (image arrays, metadata dictionaries) to maintain compatibility across different Stable Diffusion UI implementations.

7 Algorithm

The core pipeline algorithm consists of nine sequential stages, each performing a specific transformation on the input data.

Algorithm 1 Generation Stage

Require: Prompt p, grid dimensions (n, m), optional ControlNet image c, optional img2img source s
Ensure: Array of generated images $I = [I_1, I_2, \ldots, I_{n \times m}]$

```
I ← []
count ← n × m
for i = 1 to count do
    if img2img mode and s provided then
        I_i ← img2img(s, p, seed = i)
    else if ControlNet enabled and c provided then
        I_i ← controlnet_txt2img(c, p, seed = i)
    else
        I_i ← txt2img(p, seed = i)
    end if
    I.append(I_i)
end forreturn I
```

7.1 Generation Stage

The generation stage interfaces with Stable Diffusion to produce base images. The algorithm supports both txt2img and img2img modes, with optional ControlNet guidance.

The generation stage uses sequential seeds $(1, 2, \ldots, n \times m)$ to ensure variation while maintaining reproducibility. ControlNet integration allows users to provide structural guidance (e.g., pose sketches, edge maps) for consistent sprite composition.

7.2 Rasterization and Grid Tiling

Generated images are arranged into a grid layout. The rasterization stage determines optimal grid dimensions and arranges sprites in row-major order.

Algorithm 2 Rasterization and Grid Tiling

Require: Array of images I, grid width n, grid height m
Ensure: Composite spritesheet image S

```
sprite_w, sprite_h ← get_dimensions(I[0])
sheet_w ← n × sprite_w
sheet_h ← m × sprite_h
S ← create_canvas(sheet_w, sheet_h, background = transparent)
idx ← 0
for row = 0 to m − 1 do
    for col = 0 to n − 1 do
        x ← col × sprite_w
        y ← row × sprite_h
        blit(S, I[idx], x, y)
        idx ← idx + 1
    end for
end forreturn S
```

The algorithm assumes all generated images have uniform dimensions. If dimensions vary, the normalization stage (Section 7.5) handles resizing before rasterization.

7.3 Sheet Slicing

The slicing stage extracts individual tiles from the composite spritesheet. This operation is the inverse of rasterization and enables per-tile processing.

Algorithm 3 Sheet Slicing

```
function SLICESHEET(S, n, m, tile_w, tile_h)
Require: Spritesheet S, grid dimensions (n, m), tile dimensions (tile_w, tile_h)
Ensure: Array of tile images T = [T_1, T_2, ..., T_{n×m}]
    T ← []
    for row = 0 to m − 1 do
        for col = 0 to n − 1 do
            x ← col × tile_w
            y ← row × tile_h
            tile ← extract_region(S, x, y, tile_w, tile_h)
            T.append(tile)
        end for
    end forreturn T
end function
```

The slicing operation uses exact pixel boundaries to ensure tiles align correctly. Edge cases (partial tiles, overlapping regions) are handled by the normalization stage.

7.4 Tile Selection

The MVP implements order-based tile selection, where tiles are processed in the order they appear in the grid (row-major). Future versions will support CLIP-based selection for quality filtering.

Algorithm 4 Tile Selection (MVP: Order-Based)

```
Require: Array of tiles T, selection mode mode
Ensure: Filtered array of tiles T'
  if mode = "order" then
    T' ← T                                          ▷ No filtering in MVP
  else if mode = "clip" then
    T' ← []
    threshold ← 0.7                                 ▷ Quality threshold
    for each tile in T do
        score ← clip_quality_score(tile)
        if score ≥ threshold then
            T'.append(tile)
        end if
    end for
  end ifreturn T'
```

CLIP-based selection (Pro feature) uses a pre-trained CLIP model to score tiles based on prompt alignment and visual quality. The MVP processes all tiles without filtering to maintain simplicity.

7.5 Normalization

Normalization ensures all tiles have uniform dimensions and consistent alignment. This stage handles resizing, centering, and padding.

Normalization preserves aspect ratio while ensuring all tiles fit within the target dimensions. Centering ensures consistent alignment across the spritesheet.

Algorithm 5 Tile Normalization

function NORMALIZETILE($tile$, $target\_w$, $target\_h$)
Require: Tile image $tile$, target dimensions $(target\_w, target\_h)$
Ensure: Normalized tile image $tile'$
$w, h \leftarrow$ get_dimensions($tile$)
$aspect\_ratio \leftarrow w/h$
$target\_aspect \leftarrow target\_w/target\_h$
if $aspect\_ratio > target\_aspect$ **then**
$new\_h \leftarrow target\_h$
$new\_w \leftarrow$ round$(target_h \times aspect_ratio)$
else
$new\_w \leftarrow target\_w$
$new\_h \leftarrow$ round$(target_w/aspect_ratio)$
end if
$tile\_resized \leftarrow$ resize$(tile, new_w, new_h,$ filter = LANCZOS$)$
$tile' \leftarrow$ create_canvas$(target_w, target_h,$ background = transparent$)$
$offset\_x \leftarrow (target\_w - new\_w)/2$
$offset\_y \leftarrow (target\_h - new\_h)/2$
blit$(tile', tile_resized, offset_x, offset_y)$ **return** $tile'$
end function

7.6 Progressive Downscaling

Progressive downscaling reduces image resolution through multiple intermediate steps, preserving visual quality better than single-pass downscaling. The algorithm uses AREA/BOX filters for intermediate passes and NEAREST for the final pass to maintain pixel art aesthetics.

Algorithm 6 Progressive Downscaling

function DOWNSCALEPROGRESSIVE($tile$, $target\_w$, $target\_h$, $levels$)
Require: Tile image $tile$, target dimensions $(target\_w, target\_h)$, number of levels $levels$
Ensure: Downscaled tile image $tile'$
$current \leftarrow tile$
$w, h \leftarrow$ get_dimensions($tile$)
$scale\_factor \leftarrow (target\_w/w)^{1/levels}$
for $i = 1$ to $levels - 1$ **do**
$intermediate\_w \leftarrow$ round$(w \times scale_factor^i)$
$intermediate\_h \leftarrow$ round$(h \times scale_factor^i)$
$current \leftarrow$ resize$(current, intermediate_w, intermediate_h,$ filter = AREA$)$

$tile' \leftarrow$ resize$(current, target_w, target_h,$ filter = NEAREST$)$ **return** $tile'$
end function

The AREA filter (or BOX filter as fallback) provides smooth downscaling for intermediate steps, while NEAREST ensures crisp pixel boundaries in the final output. The number of levels is configurable, with a default of 3 for most use cases.

7.7 Palette Reduction

Palette reduction quantizes colors to a target palette size, reducing file size and ensuring compatibility with engines that require limited color palettes.

The algorithm uses k-means clustering to identify representative colors, then maps each pixel to the nearest palette color. Floyd-Steinberg dithering reduces banding artifacts in gradients. If palette size is 0, the stage is skipped.

Algorithm 7 Palette Quantization

function QUANTIZEPALETTE(*tile*, *palette_size*)
Require: Tile image *tile*, target palette size *palette_size*
Ensure: Quantized tile image $tile'$
  if $palette\_size = 0$ **then return** *tile* ▷ No reduction requested
  end if
  $colors \leftarrow$ extract_colors(*tile*)
  $palette \leftarrow$ kmeans_quantize($colors, palette\_size$)
  $tile' \leftarrow$ map_colors(*tile*, *palette*)
  $tile' \leftarrow$ apply_dithering($tile'$, method = FLOYD_STEINBERG) **return** $tile'$
end function

7.8 Post-Processing Cleanup

Post-processing cleanup handles edge cases and final adjustments: removing artifacts, cleaning up transparency, and applying optional filters.

Algorithm 8 Post-Processing Cleanup

Require: Array of processed tiles T
Ensure: Cleaned array of tiles T'
  $T' \leftarrow []$
  for each *tile* in T **do**
    tile $\leftarrow$ remove_artifacts(*tile*)
    tile $\leftarrow$ clean_transparency(*tile*)
    if sharpening enabled **then**
      tile $\leftarrow$ apply_sharpen(*tile*, strength = 0.1)
    end if
    T'.append(*tile*)
  end forreturn T'

Artifact removal uses morphological operations to eliminate stray pixels. Transparency cleaning ensures consistent alpha channel handling. Optional sharpening can be applied to compensate for downscaling blur.

7.9 Export

The export stage assembles the final spritesheet and generates metadata files.

Algorithm 9 Export Stage

Require: Array of tiles T, grid dimensions (n, m), output directory *dir*, engine preset *preset*
Ensure: Files written to *dir*
  $sheet \leftarrow$ rasterize(T, n, m)
  save_image($sheet$, path = dir + "/spritesheet.png")
  $manifest \leftarrow$ generate_manifest(T, n, m)
  save_json($manifest$, path = dir + "/manifest.json")
  if $preset \neq$ null **then**
    $engine\_config \leftarrow$ generate_preset($preset, manifest$)
    save_json($engine\_config$, path = dir + "/engine_preset.json")
  end ifreturn

The export stage re-rasterizes processed tiles into the final spritesheet. The manifest includes tile coordinates, dimensions, and generation parameters for downstream tools. Engine presets provide format-specific configurations (e.g., Unity sprite import settings).

8 Quality Metrics

The pipeline's effectiveness is measured through several quality metrics:

Metric	Description
Visual Consistency	Perceptual similarity across sprites in a sheet (SSIM, LPIPS)
Dimensional Accuracy	Percentage of sprites matching target dimensions exactly
Palette Efficiency	Actual palette size vs. target (lower is better)
File Size	Output spritesheet size in bytes
Processing Time	End-to-end pipeline execution time
User Satisfaction	Subjective quality ratings from test users

Table 3: Quality Metrics

Target values for MVP:

- Visual Consistency: SSIM > 0.85 within sprite variations
- Dimensional Accuracy: 100% (enforced by normalization)
- Processing Time: < 5 minutes for a 4×4 grid at 64×64 resolution
- File Size: Reasonable for target resolution (typically < 1 MB for 256×256 sheets)

9 MVP Scope

The Minimum Viable Product includes the following features:

- **Core Pipeline**: All nine algorithm stages implemented and functional
- **UI Integration**: Basic plugin interface for ComfyUI and AUTOMATIC1111
- **Input Support**: txt2img, img2img, and ControlNet integration
- **Output Formats**: PNG spritesheets and JSON manifests
- **Tile Selection**: Order-based selection (all tiles processed)
- **Downscaling**: Progressive downscaling with AREA/NEAREST filters
- **Palette Reduction**: k-means quantization with Floyd-Steinberg dithering
- **Documentation**: User guide and API documentation

Excluded from MVP:

- CLIP-based tile selection (Pro feature)
- Engine-specific presets beyond basic JSON
- Batch processing of multiple prompts
- Advanced post-processing filters
- Cloud storage integration
- Multi-language UI support

The MVP focuses on core functionality to validate the approach and gather user feedback before implementing advanced features.

10 Tech Stack

The pipeline is implemented using the following technologies:

Python 3.9+ Core implementation language, chosen for compatibility with Stable Diffusion ecosystems

Pillow (PIL) Image processing operations (resizing, compositing, format conversion)

NumPy Array operations for image data manipulation

scikit-image Advanced image processing (morphological operations, filters)

ComfyUI API Integration layer for ComfyUI workflows

AUTOMATIC1111 API Integration layer for AUTOMATIC1111 web UI

ControlNet Optional dependency for structural guidance

JSON Metadata serialization format

The implementation follows Python packaging standards and can be installed via pip. Dependencies are managed through `requirements.txt` with version pinning for stability.

Example dependency specification:

```
Pillow>=10.0.0
numpy>=1.24.0
scikit-image>=0.21.0
```

The plugin architecture allows the pipeline to work with any Stable Diffusion UI that provides a Python API, though initial support targets ComfyUI and AUTOMATIC1111.

11 Test Plan

Testing is organized into three categories: unit tests, integration tests, and user acceptance tests.

11.1 Unit Tests

Each algorithm stage is tested independently:

- **Generation Stage**: Mock Stable Diffusion API calls, verify image array output
- **Rasterization**: Test grid layout correctness, verify tile positioning
- **Slicing**: Verify inverse relationship with rasterization, test edge cases
- **Normalization**: Test aspect ratio preservation, centering accuracy
- **Downscaling**: Verify quality preservation, test filter application
- **Palette Reduction**: Test quantization accuracy, verify palette size limits
- **Export**: Verify file generation, test manifest structure

11.2 Integration Tests

End-to-end pipeline tests with realistic inputs:

- **Full Pipeline**: Execute complete workflow with sample prompts (e.g., "red mailbox", "desk lamp", "trash can")
- **ControlNet Integration**: Test with pose/edge guidance images
- **Error Handling**: Test behavior with invalid inputs, missing dependencies
- **UI Integration**: Test plugin registration and workflow execution in both ComfyUI and AUTOMATIC1111

11.3 User Acceptance Tests

Real-world usage scenarios:

- **Game Asset Generation**: Generate spritesheets for common game objects
- **Style Consistency**: Verify visual consistency across multiple generation runs
- **Performance**: Measure processing time for various grid sizes and resolutions
- **Output Quality**: Subjective evaluation of final spritesheet quality

Test coverage target: $> 80\%$ for core pipeline code, 100% for critical path functions (generation, slicing, export).

12 Future Work

Proposed extensions beyond the MVP:

1. **CLIP-Based Tile Selection**: Implement quality scoring using CLIP models to filter low-quality tiles automatically
2. **Advanced Post-Processing**: Add optional filters (outline generation, shadow casting, color grading)
3. **Animation Support**: Extend pipeline to process animation frame sequences into sprite animations
4. **Engine Presets**: Expand support for Unity, Godot, Unreal Engine with format-specific optimizations
5. **Batch Processing**: Support processing multiple prompts in a single execution
6. **Cloud Integration**: Optional cloud storage for generated assets
7. **Web UI**: Standalone web interface for users who prefer not to use Stable Diffusion UIs directly
8. **Model Fine-Tuning**: Tools for fine-tuning Stable Diffusion models on sprite-specific datasets
9. **Multi-Language Support**: Internationalization for UI components
10. **Performance Optimization**: GPU acceleration for processing stages, parallel tile processing

Priority is given to features that improve quality (CLIP selection) and usability (engine presets, batch processing) based on user feedback from MVP deployment.

13 Conclusion

This paper has documented a plugin-based pipeline for generating production-ready 2D spritesheets within Stable Diffusion user interfaces. The system automates the complete workflow from image generation through final export, eliminating manual post-processing steps and improving consistency.

The modular architecture enables easy extension and integration with existing Stable Diffusion ecosystems. The MVP focuses on core functionality while establishing a foundation for advanced features like CLIP-based quality filtering and engine-specific optimizations.

By operating as a plugin within ComfyUI and AUTOMATIC1111, the pipeline leverages established workflows while adding specialized spritesheet capabilities. The open-source implementation ensures community-driven development and broad compatibility.

Future work will expand the pipeline's capabilities based on user feedback and emerging requirements from game development and digital art communities. The system's architecture supports these extensions without requiring fundamental redesign.