**A Technical Architecture and Methodological Analysis of the Aquiplicity 2025 Image Composition Engine by Tracy Rose**

**Abstract**

Aquiplicity 2025 is a single-page, client-side application for procedural image composition. It leverages a deterministic algorithm based on colorimetric differentiation to synthesize a single master image from a stack of source layers. The system's core innovation lies in its generation of a layerMatrix, a data structure that maps each pixel of the composed image to its source layer. This matrix decouples the computationally intensive composition phase from the subsequent real-time, interactive manipulation phase, enabling a suite of performant post-composition editing tools. This document provides a detailed analysis of the system's architecture, data structures, and the mathematical foundations of its composition and tooling algorithms.

**1.0 System Architecture Overview**

The application operates entirely within the client's web browser, utilizing the standard HTML5 Canvas API, DOM, and JavaScript (ES6+). It follows a single-threaded execution model, with asynchronous operations managed via Promises and setTimeout calls to prevent UI blocking during intensive computations. The architecture can be logically divided into three primary components:

* **Data Layer:** Consists of the in-memory ImageStack class, which manages an array of ImageData objects, and the two principal output buffers: masterImageData (the visual output) and the layerMatrix (the positional source map). State management for undo/redo functionality is handled by a simple LIFO stack of ImageData objects.
* **Logic Layer (Composition Engine):** Contains the core algorithms for image composition, layer smoothing, color distance calculation, and the implementation of all interactive editing tools.
* **Presentation Layer:** The HTML structure and CSS, which provide the user interface controls (sliders, buttons) and the primary canvas element for rendering output.

**2.0 Data Ingestion and Pre-processing**

The workflow begins with user-provided image files.

1. **File Handling:** The ImageInput element accepts multiple files. These are processed using the URL.createObjectURL() method to create a direct memory reference for an <img> element.
2. **Rasterization and Normalization:** Each image is drawn onto a temporary <canvas> element to rasterize its data. The ImageData is then extracted via ctx.getImageData(). The dimensions of the first image loaded establish the canonical dimensions (W, H) for the entire session. All subsequent images are resized to these dimensions during this rasterization step to ensure dimensional homogeneity across the layer stack.
3. **Alpha Channel Sanitization:** The alpha channel for every pixel is explicitly set to 255 (fully opaque) to ensure that the color distancing algorithm operates solely on RGB values and is not affected by transparency masks in the source files.
4. **Data Structure:** The processed ImageData objects are stored in an ordered array within the ImageStack instance.

**3.0 The Core Composition Algorithm**

The composition is a deterministic process that generates the masterImageData and the corresponding layerMatrix.

**3.1 Layer Smoothing (Pre-computation)**  
Before the main composition loop, each layer's ImageData is processed by a smoothing filter. This is a crucial pre-computation step designed to reduce high-frequency noise and create more contiguous color regions. The implemented filter is a simple 3x3 box blur, where each pixel's new RGB value is the arithmetic mean of its and its eight immediate neighbors' values.

P'(x,y) = (1/9) \* Σ(i=-1 to 1) Σ(j=-1 to 1) P(x+i, y+j)

It is critical to note that this smoothed data is used **only for the decision-making process**; the final masterImageData is constructed using pixels from the original, unsmoothed source layers.

**3.2 Pixel-wise Colorimetric Differentiation**  
The engine iterates through every pixel coordinate p(x, y) from (0, 0) to (W-1, H-1). For each coordinate, it performs a top-down layer comparison.

1. **Reference Color:** The color C\_base is sampled from the smoothed base layer, S\_0.
2. **Comparison Loop:** The algorithm iterates from the topmost smoothed layer S\_(N-1) down to S\_1.
3. **Color Distance Calculation:** In each iteration, it calculates the Euclidean distance ΔE between the current layer's pixel color C\_i and the base layer's pixel color C\_base in 3D RGB space.

ΔE = sqrt((R\_i - R\_base)² + (G\_i - G\_base)² + (B\_i - B\_base)²)

This metric provides a scalar value representing the perceptual difference between the two colors.

**3.3 The Threshold Parameter (τ)**  
A user-defined scalar value, τ (tau), serves as the decision boundary. This value is derived from the UI slider, which provides a normalized value v in the range [0.0, 1.0]. This is mapped to the range [0, 255] for the comparison:

τ = v \* 255

If ΔE > τ, the pixel p(x, y) from layer i is considered "sufficiently different." The comparison loop for that coordinate is terminated, and layer i is selected as the source.

**3.4 Generation of Output Buffers**  
Upon selecting a source layer L for a coordinate (x, y) (where L defaults to 0 if no layer satisfies the threshold condition):

* The RGBA quadlet from the **original, non-smoothed** ImageData of layer L at (x, y) is written to the masterImageData buffer.
* The index L is written to the layerMatrix buffer at the corresponding position: layerMatrix[y \* W + x] = L.

Upon completion of the main loop, these two buffers represent the complete state of the composed image.

<h4><strong>4.0 Post-Composition Interactive Tooling: Leveraging the <code>layerMatrix</code></strong></h4>

The existence of the layerMatrix allows for highly performant editing without re-running the O(W \* H \* L) composition algorithm. All tools operate directly on the masterImageData buffer.

**4.1 Patch Brush and Patch Lasso**  
These tools implement a "pixel replacement" strategy guided by the layerMatrix.

1. **Source Layer Identification:** On mousedown, the tool samples the layerMatrix at the initial cursor coordinates (x\_0, y\_0) to retrieve the source layer index L\_src.
2. **Pixel Transfer:** For every subsequent pixel (x\_i, y\_i) within the drawn brush stroke or lasso boundary, the tool reads the RGBA quadlet from ImageStack.images[L\_src] at (x\_i, y\_i) and writes it directly into the masterImageData at the same coordinate. This operation is effectively a masked memory copy from one buffer to another.

**4.2 Blend/Feather Lasso**  
This tool applies a spatial convolution filter limited to the lasso's boundary.

1. A copy of the masterImageData within the lasso's bounding box is created to serve as a static read source.
2. For each pixel p(x, y) inside the lasso polygon, a new color is calculated as the arithmetic mean of the colors of its neighboring pixels (within a fixed radius) that are *also* inside the polygon.
3. This averaged color is then written to the masterImageData, resulting in a localized blur or "feathering" effect.

**4.3 Gradient Overlay Lasso**  
This tool applies a procedural, non-photorealistic gradient.

1. **Color Sampling:** The algorithm samples pixels from two small regions at the start and end of the lasso's primary axis (horizontal or vertical) to calculate two average colors, C\_start and C\_end.
2. **Linear Interpolation (Lerp):** For each pixel p(x, y) inside the lasso, an interpolation factor t in [0, 1] is calculated based on its normalized position along the primary axis.
3. **Color Calculation:** A new gradient color is calculated via linear interpolation: C\_grad = (1-t)C\_start + tC\_end.
4. **Alpha Blending:** This C\_grad is then alpha-blended with the pixel's original color from masterImageData using a fixed opacity (GRADIENT\_LASSO\_OPACITY), and the result is written back to the buffer.

**5.0 Conclusion**

The Aquiplicity 2025 engine demonstrates an effective client-side architecture for complex procedural image synthesis. Its key strength lies in the generation of the layerMatrix, a metadata structure that successfully decouples the initial, computationally expensive analysis from subsequent, real-time user interactions. This architectural decision allows a feature-rich editing experience to be built on top of a simple yet powerful core composition algorithm, all within the resource constraints of a standard web browser. Future work could explore multi-threading with Web Workers for the composition loop and more sophisticated color difference models like CIEDE2000 for threshold comparisons.