**Aquiplicity 2025 Advanced**

**Technical Overview by Tracy Rose**

This document provides a comprehensive technical analysis of **Aquiplicity 2025 (Single Threshold - Original Resolution)**, a sophisticated web-based image composition tool designed for advanced image processing. Targeted at researchers, engineers, and computer scientists with a PhD-level understanding of image processing, this manual delves into the core algorithms, focusing on the threshold mechanism, color distancing, and other technically significant features. It assumes familiarity with concepts such as image data structures, pixel manipulation, and computational geometry.

**Table of Contents**

1. System Overview
2. Core Data Structure: ImageStack Class
3. Threshold Mechanism
   * Definition and Role
   * Mathematical Formulation
   * Implementation Details
4. Color Distancing
   * Euclidean Distance in RGB Space
   * Application in Composition
5. Image Composition Algorithm
   * Smoothing Process
   * Pixel Selection Logic
   * Layer Matrix
6. Interactive Editing Features
   * Patch Click and Lasso
   * Blend Lasso
   * Gradient Lasso
7. Undo System and State Management
8. Image Removal and Stack Management
9. Canvas Interaction and Rendering
10. Performance Considerations
11. Technical Significance and Novel Aspects
12. Limitations and Future Extensions

**System Overview**

Aquiplicity 2025 is a browser-based application implemented in HTML5, CSS, and JavaScript, leveraging the HTML5 Canvas API for real-time image manipulation. It enables users to upload multiple images, compose them into a single output based on a user-defined threshold, and interactively edit the result using patching, blending, and gradient overlay tools. The system is designed to operate at the original resolution of the input images, preserving fidelity while introducing computational challenges for large datasets.

Key features include:

* **Threshold-based composition**: Combines images by comparing pixel color differences against a threshold.
* **Color distancing**: Uses Euclidean distance in RGB space to quantify pixel differences.
* **Interactive editing**: Supports patch-based layer swapping, localized blending, and gradient overlays.
* **Dynamic image management**: Allows runtime addition and removal of image layers.
* **Undo system**: Maintains a history of up to 15 states for reversible editing.
* **Responsive UI**: Adapts canvas size to screen constraints while maintaining pixel accuracy.

The system is architecturally centered around the ImageStack class, which manages image data and orchestrates composition, with a modular front-end for user interaction.

**Core Data Structure: ImageStack Class**

The ImageStack class is the backbone of Aquiplicity’s processing pipeline, encapsulating a collection of images and their metadata.

* **Attributes**:
  + images: An array of ImageData objects, each representing a layer’s pixel data in RGBA format.
  + width, height: Dimensions of the stack, set by the first loaded image and enforced across all layers.
* **Methods**:
  + addImage(file): Loads an image file, resizes it to match stack dimensions if necessary, and stores its ImageData.
  + smoothLayer(imageData, iterations): Applies iterative averaging to reduce noise in a layer.
  + colorDistance(r1, g1, b1, r2, g2, b2): Computes Euclidean distance between two RGB colors.
  + compose(threshold): Generates a composite image by selecting pixels based on color differences.

The class ensures dimensional consistency by resizing subsequent images to match the first image’s resolution, using bilinear interpolation via canvas drawing. It also handles memory management by revoking object URLs after loading.

**Threshold Mechanism**

**Definition and Role**

The threshold is a user-defined parameter controlling the sensitivity of pixel selection during composition. It represents the minimum color difference (in RGB space) required for a pixel from a non-base layer to be chosen over the base layer’s pixel. The threshold is expressed as a normalized value (0.01 to 1.0), scaled to a percentage (1% to 100%) for user interaction, and internally mapped to a 0–255 range for RGB calculations.

**Mathematical Formulation**

Let ( T ) be the user-defined threshold (0.01 ≤ ( T ) ≤ 1.0), scaled internally as:

T\_{\text{internal}} = T \times 255

For a pixel at position ((x, y)) in the smoothed base layer (index 0) with RGB values

(r\_0, g\_0, b\_0)

and a smoothed layer ( i ) (for

i \geq 1

) with RGB values

(r\_i, g\_i, b\_i)

, the color distance

D\_i

is computed as:

D\_i = \sqrt{(r\_i - r\_0)^2 + (g\_i - g\_0)^2 + (b\_i - b\_0)^2}

The pixel selection rule is:

* If

D\_i > T\_{\text{internal}}

for any layer ( i ) (scanned from highest to lowest index), select the pixel from layer ( i ).

* Otherwise, retain the pixel from the base layer (index 0).

This creates a binary decision boundary in RGB space, where the threshold defines a hypersphere centered on the base pixel’s color.

**Implementation Details**

* **User Input**: The threshold is adjusted via a slider (<input type="range">), with values mapped to a percentage display (e.g.,

T = 0.15 \rightarrow 15\%

).

* **Internal Scaling**: The slider value is multiplied by 255 in getThreshold() to align with RGB intensity ranges.
* **Composition Impact**: Lower thresholds (e.g., 5%) favor the base layer, producing conservative composites, while higher thresholds (e.g., 50%) increase contributions from other layers, emphasizing color variations.
* **Status Reporting**: The composition process reports the effective threshold as a percentage of the maximum RGB distance (255), e.g.,

T\_{\text{internal}} / 255 \times 100

.

**Color Distancing**

**Euclidean Distance in RGB Space**

Color distancing is implemented using the Euclidean distance metric in the RGB color space, treating each pixel’s RGB values as a 3D vector. For two pixels with RGB values

(r\_1, g\_1, b\_1)

and

(r\_2, g\_2, b\_2)

, the distance is:

D = \sqrt{(r\_1 - r\_2)^2 + (g\_1 - g\_2)^2 + (b\_1 - b\_2)^2}

This is computed in the colorDistance method:

javascript

colorDistance(r1, g1, b1, r2, g2, b2) {

const dr = r1 - r2;

const dg = g1 - g2;

const db = b1 - b2;

return Math.sqrt(dr \* dr + dg \* dg + db \* db);

}

* **Range**: Since RGB values are in [0, 255], the maximum distance is

\sqrt{3 \times 255^2} \approx 441.67

.

* **Precision**: The method uses Math.sqrt for exact computation, avoiding approximations that could skew threshold comparisons.

**Application in Composition**

Color distancing is central to the composition algorithm:

* **Smoothed Layers**: Distance is computed on smoothed versions of each layer to mitigate noise, ensuring robust comparisons.
* **Layer Priority**: Layers are evaluated in reverse order (highest index first), allowing later-added images to override earlier ones when the threshold is exceeded.
* **Performance Optimization**: Distances are computed per pixel, with early termination if a layer satisfies the threshold, reducing unnecessary calculations.

This approach assumes RGB space as a perceptually uniform metric, which is a simplification (see Limitations (#limitations-and-future-extensions)).

**Image Composition Algorithm**

The compose method in ImageStack orchestrates the creation of a composite image. It requires at least two images and operates as follows:

**Smoothing Process**

Each layer is smoothed to reduce noise and stabilize color comparisons:

* **Algorithm**: A 4-neighbor averaging filter is applied iteratively (default: 6 iterations).
* **Implementation**:

javascript

smoothLayer(imageData, iterations = 6) {

const data = new Float32Array(imageData.data);

const temp = new Float32Array(data.length);

const width = imageData.width;

const height = imageData.height;

for (let iter = 0; iter < iterations; iter++) {

for (let y = 0; y < height; y++) {

for (let x = 0; x < width; x++) {

const idx = (y \* width + x) \* 4;

if (x === 0 || x === width - 1 || y === 0 || y === height - 1) {

for (let c = 0; c < 4; c++) temp[idx + c] = data[idx + c];

} else {

for (let c = 0; c < 3; c++) {

temp[idx + c] = (

data[((y - 1) \* width + x) \* 4 + c] +

data[((y + 1) \* width + x) \* 4 + c] +

data[(y \* width + (x - 1)) \* 4 + c] +

data[(y \* width + (x + 1)) \* 4 + c]

) / 4;

}

temp[idx + 3] = data[idx + 3]; *// Preserve alpha*

}

}

}

data.set(temp);

}

const result = new Uint8ClampedArray(data.length);

for (let i = 0; i < data.length; i++) {

result[i] = Math.round(Math.min(255, Math.max(0, data[i])));

}

return new ImageData(result, width, height);

}

* **Boundary Handling**: Edge pixels are preserved to avoid artifacts.
* **Precision**: Computations use Float32Array to prevent rounding errors, with final clamping to Uint8ClampedArray for canvas compatibility.
* **Effect**: Smoothing reduces high-frequency noise, making the threshold-based selection more consistent across regions.

**Pixel Selection Logic**

For each pixel ((x, y)):

1. Compute the color distance between the smoothed base layer and each smoothed non-base layer.
2. Select the first layer (highest index) where

D\_i > T\_{\text{internal}}

, or default to the base layer.

1. Copy the original (unsmoothed) pixel from the selected layer to the output buffer.
2. Store the selected layer index in the layerMatrix.

The output is an ImageData object (masterImageData) and a Uint8Array (layerMatrix) mapping each pixel to its source layer.

**Layer Matrix**

* **Structure**: A 1D Uint8Array of length

W \times H

, where each element is the index of the source layer for pixel ((x, y)).

* **Purpose**: Enables interactive editing by tracking the provenance of each pixel, crucial for patching operations.
* **Memory Efficiency**: Uses 8-bit integers, sufficient for up to 256 layers (Aquiplicity limits to 15).

The composition process is computationally intensive, with complexity

O(W \times H \times L \times I)

, where ( W ) and ( H ) are image dimensions, ( L ) is the number of layers, and ( I ) is the number of smoothing iterations.

**Interactive Editing Features**

Aquiplicity supports three editing modes, each leveraging the layerMatrix and masterImageData for precise manipulation.

**Patch Click and Lasso**

* **Patch Click**:
  + **Trigger**: Mouse click without modifiers.
  + **Operation**: Replaces a

2 \times 2

pixel region centered at the click point with pixels from the layer indicated by layerMatrix at that point.

* + **Implementation**: Iterates over the patch region, copying RGBA values from the source layer’s original ImageData.
  + **Complexity**: ( O(1) ), as the patch size is fixed (PATCH\_SIZE = 2).
* **Patch Lasso**:
  + **Trigger**: Ctrl + click and drag.
  + **Operation**: Defines a polygonal region via mouse points, then replaces all pixels inside with those from the layer at the lasso’s starting point.
  + **Point-in-Polygon Test**: Uses the ray-casting algorithm (isPointInLasso) to determine if a pixel lies within the lasso:

javascript

isPointInLasso(x, y, points) {

let inside = false;

for (let i = 0, j = points.length - 1; i < points.length; j = i++) {

const xi = points[i].x, yi = points[i].y;

const xj = points[j].x, yj = points[j].y;

const intersect = ((yi > y) !== (yj > y)) && (x < (xj - xi) \* (y - yi) / (yj - yi) + xi);

if (intersect) inside = !inside;

}

return inside;

}

* + **Complexity**:

O(W \times H \times P)

in the worst case, where ( P ) is the number of lasso points, though bounding box optimization reduces this to

O((maxX - minX) \times (maxY - minY) \times P)

.

**Blend Lasso**

* **Trigger**: Alt + click and drag.
* **Operation**: Applies a localized averaging filter within the lassoed region, blending pixels with their neighbors within a radius (BLEND\_LASSO\_RADIUS = 2).
* **Algorithm**:
  + For each pixel ((x, y)) inside the lasso, compute the average RGB values of neighboring pixels also inside the lasso, within a

5 \times 5

window (radius 2).

* + Update the pixel’s RGBA values with the computed average.
* **Implementation**:

javascript

applyLassoBlend() {

const w = masterImageData.width;

const h = masterImageData.height;

const originalData = new Uint8ClampedArray(masterImageData.data);

const targetData = masterImageData.data;

const minX = Math.max(0, Math.floor(Math.min(...lassoPoints.map(p => p.x))));

const maxX = Math.min(w - 1, Math.ceil(Math.max(...lassoPoints.map(p => p.x))));

const minY = Math.max(0, Math.floor(Math.min(...lassoPoints.map(p => p.y))));

const maxY = Math.min(h - 1, Math.ceil(Math.max(...lassoPoints.map(p => p.y))));

let updatedPixels = 0;

const radius = BLEND\_LASSO\_RADIUS;

for (let y = minY; y <= maxY; y++) {

for (let x = minX; x <= maxX; x++) {

if (isPointInLasso(x + 0.5, y + 0.5, lassoPoints)) {

let r = 0, g = 0, b = 0, a = 0, cnt = 0;

for (let ky = -radius; ky <= radius; ky++) {

for (let kx = -radius; kx <= radius; kx++) {

const nx = x + kx;

const ny = y + ky;

if (nx >= 0 && nx < w && ny >= 0 && ny < h) {

if (isPointInLasso(nx + 0.5, ny + 0.5, lassoPoints)) {

const idx = (ny \* w + nx) \* 4;

r += originalData[idx];

g += originalData[idx + 1];

b += originalData[idx + 2];

a += originalData[idx + 3];

cnt++;

}

}

}

}

if (cnt > 0) {

const idx = (y \* w + x) \* 4;

targetData[idx] = Math.round(r / cnt);

targetData[idx + 1] = Math.round(g / cnt);

targetData[idx + 2] = Math.round(b / cnt);

targetData[idx + 3] = Math.round(a / cnt);

updatedPixels++;

}

}

}

}

*// ...*

}

* **Effect**: Produces a feathered, smooth transition, ideal for softening layer boundaries.
* **Complexity**:

O((maxX - minX) \times (maxY - minY) \times R^2 \times P)

, where ( R ) is the radius.

**Gradient Lasso**

* **Trigger**: Shift + click and drag.
* **Operation**: Applies a linear gradient overlay within the lassoed region, blending with the original image at 25% opacity (GRADIENT\_LASSO\_OPACITY = 0.25).
* **Algorithm**:
  + Determine the lasso’s bounding box and orientation (horizontal if width ≥ height, else vertical).
  + Sample average colors at the start and end of the gradient direction (10% of the lasso’s width/height).
  + Interpolate colors linearly across the lasso:

c(t) = c\_1 \cdot (1 - t) + c\_2 \cdot t, \quad t \in [0, 1]

where

c\_1, c\_2

are start and end RGB colors, and ( t ) is the normalized position along the gradient axis.

* + Blend with the original pixel using:

c\_{\text{out}} = c\_{\text{original}} \cdot (1 - \alpha) + c\_{\text{gradient}} \cdot \alpha, \quad \alpha = 0.25

* **Implementation**:

javascript

applyLassoGradient() {

const w = masterImageData.width;

const h = masterImageData.height;

const originalData = new Uint8ClampedArray(masterImageData.data);

const targetData = masterImageData.data;

const minX = Math.max(0, Math.floor(Math.min(...lassoPoints.map(p => p.x))));

const maxX = Math.min(w - 1, Math.ceil(Math.max(...lassoPoints.map(p => p.x))));

const minY = Math.max(0, Math.floor(Math.min(...lassoPoints.map(p => p.y))));

const maxY = Math.min(h - 1, Math.ceil(Math.max(...lassoPoints.map(p => p.y)));

const gradWidth = maxX - minX;

const gradHeight = maxY - minY;

const isHorizontal = gradWidth >= gradHeight;

const len = isHorizontal ? gradWidth : gradHeight;

const sampleW = Math.max(1, Math.floor(Math.min(gradWidth, gradHeight) \* 0.1));

let sR = 0, sG = 0, sB = 0, sA = 0, sCnt = 0;

let eR = 0, eG = 0, eB = 0, eA = 0, eCnt = 0;

*// Sample colors*

for (let y = minY; y <= maxY; y++) {

for (let x = minX; x <= maxX; x++) {

if (isPointInLasso(x + 0.5, y + 0.5, lassoPoints)) {

const idx = (y \* w + x) \* 4;

const cR = originalData[idx], cG = originalData[idx + 1], cB = originalData[idx + 2], cA = originalData[idx + 3];

if (isHorizontal) {

if (x < minX + sampleW) { sR += cR; sG += cG; sB += cB; sA += cA; sCnt++; }

else if (x > maxX - sampleW) { eR += cR; eG += cG; eB += cB; eA += cA; eCnt++; }

} else {

if (y < minY + sampleW) { sR += cR; sG += cG; sB += cB; sA += cA; sCnt++; }

else if (y > maxY - sampleW) { eR += cR; eG += cG; eB += cB; eA += cA; eCnt++; }

}

}

}

}

const c1 = { r: sCnt > 0 ? sR / sCnt : 128, g: sCnt > 0 ? sG / sCnt : 128, b: sCnt > 0 ? sB / sCnt : 128, a: sCnt > 0 ? sA / sCnt : 255 };

const c2 = { r: eCnt > 0 ? eR / eCnt : 128, g: eCnt > 0 ? eG / eCnt : 128, b: eCnt > 0 ? eB / eCnt : 128, a: eCnt > 0 ? eA / eCnt : 255 };

let updatedPixels = 0;

const op = GRADIENT\_LASSO\_OPACITY;

const invOp = 1 - op;

for (let y = minY; y <= maxY; y++) {

for (let x = minX; x <= maxX; x++) {

if (isPointInLasso(x + 0.5, y + 0.5, lassoPoints)) {

let t = 0;

if (len > 0) {

t = isHorizontal ? (x - minX) / len : (y - minY) / len;

t = Math.max(0, Math.min(1, t));

}

const gradR = c1.r \* (1 - t) + c2.r \* t;

const gradG = c1.g \* (1 - t) + c2.g \* t;

const gradB = c1.b \* (1 - t) + c2.b \* t;

const idx = (y \* w + x) \* 4;

const oR = originalData[idx], oG = originalData[idx + 1], oB = originalData[idx + 2], oA = originalData[idx + 3];

targetData[idx] = Math.round(oR \* invOp + gradR \* op);

targetData[idx + 1] = Math.round(oG \* invOp + gradG \* op);

targetData[idx + 2] = Math.round(oB \* invOp + gradB \* op);

targetData[idx + 3] = oA;

updatedPixels++;

}

}

}

*// ...*

}

* **Effect**: Adds a semi-transparent color wash, enhancing creative control.
* **Complexity**: Similar to Blend Lasso, but with additional sampling overhead.

**Undo System and State Management**

* **Mechanism**: Stores up to MAX\_HISTORY = 15 copies of masterImageData in historyStack before each edit.
* **Implementation**:
  + saveStateForUndo(actionDescription): Creates a deep copy of masterImageData using new ImageData.
  + undoAction(): Pops the last state, restores masterImageData, and redisplays the canvas.
* **Memory Usage**: Each state consumes

W \times H \times 4

bytes, so for a 1920x1080 image, one state is ~8.3 MB, and 15 states are ~124.5 MB.

* **Constraints**: History is cleared after image removal or composition, as these invalidate prior states.
* **Complexity**:

O(W \times H)

for saving/restoring states.

**Image Removal and Stack Management**

* **Function**: removeImage(indexToRemove) removes an image from imageStack.images and its thumbnail from the UI.
* **Process**:
  + Saves the current masterImageData for undo (though history is cleared post-removal).
  + Updates imageStack by splicing the target image.
  + Removes the corresponding thumbnail and reindexes remaining thumbnails.
  + Clears masterImageData and layerMatrix, requiring recomposition.
* **Complexity**: ( O(1) ) for array splicing and UI updates,

O(W \times H)

if saving state.

* **Significance**: Enables dynamic layer management without reloading, preserving workflow continuity.

**Canvas Interaction and Rendering**

* **Canvas Setup**: Uses a 2D context with willReadFrequently: true for optimized getImageData/putImageData.
* **Scaling**: The fitToScreen function computes display dimensions based on the working area’s size, preserving aspect ratio:

\text{newWidth} = \min(\text{imgWidth}, \text{maxWidth}), \quad \text{newHeight} = \text{newWidth} / \text{aspectRatio}

If

\text{newHeight} > \text{maxHeight}

, adjust:

\text{newHeight} = \text{maxHeight}, \quad \text{newWidth} = \text{newHeight} \times \text{aspectRatio}

* **Event Handling**:
  + Mouse events (mousedown, mousemove, mouseup) are scaled to canvas coordinates:

x\_{\text{canvas}} = (x\_{\text{client}} - \text{rect.left}) \times \frac{\text{canvas.width}}{\text{rect.width}}

* + Modifier keys (Ctrl, Alt, Shift) trigger distinct lasso modes.
* **Lasso Rendering**: Draws polylines with semi-transparent fills (green for patch, yellow for blend, pink for gradient) using ctx.globalAlpha.

**Performance Considerations**

* **Composition**:

O(W \times H \times L \times I)

, dominated by smoothing and pixel comparisons. For a 1920x1080 image with 10 layers and 6 iterations, ~1.24 billion operations.

* **Lasso Operations**: Bounded by the lasso’s bounding box, reducing effective complexity.
* **Memory**: Large images (e.g., 4000x4000) consume significant RAM (~64 MB per layer, plus history).
* **Optimizations**:
  + Asynchronous updates (setTimeout) prevent UI blocking.
  + Bounding box checks limit pixel iterations.
  + Early termination in distance calculations.
* **Bottlenecks**: High-resolution images or many layers can overwhelm browser memory or CPU, addressed by warnings for resolutions >4000 pixels.

**Technical Significance and Novel Aspects**

1. **Threshold-Based Composition**:
   * Combines global thresholding with per-pixel layer selection, offering a balance between automation and control.
   * Smoothed comparisons enhance robustness against noise, unlike naive pixel differencing.
2. **Color Distancing**:
   * Simple yet effective use of Euclidean RGB distance, optimized for real-time performance.
   * Layer-priority scanning ensures deterministic outcomes.
3. **Interactive Editing**:
   * Patch Lasso leverages layerMatrix for seamless layer swapping, akin to advanced compositing in professional software.
   * Blend Lasso implements localized Gaussian-like smoothing, optimized for sparse regions.
   * Gradient Lasso’s adaptive color sampling creates context-aware overlays, a novel feature for web-based tools.
4. **Dynamic Layer Management**:
   * Runtime image removal with UI synchronization is non-trivial in a canvas-based system.
   * Maintains state consistency despite asynchronous operations.
5. **Undo System**:
   * Efficiently balances memory usage and functionality, critical for iterative workflows.
6. **Browser-Based Execution**:
   * Achieves near-native performance using JavaScript and Canvas, pushing the boundaries of web applications.

**Limitations and Future Extensions**

1. **Color Space**:
   * **Limitation**: Euclidean distance in RGB is not perceptually uniform, potentially leading to unintuitive blends.
   * **Extension**: Adopt CIELAB or HSV color spaces for better human-aligned distance metrics.
2. **Threshold Granularity**:
   * **Limitation**: Single global threshold lacks spatial adaptability.
   * **Extension**: Implement per-region thresholds or machine learning-based segmentation.
3. **Smoothing**:
   * **Limitation**: Fixed 4-neighbor averaging may over-smooth fine details.
   * **Extension**: Use adaptive filters (e.g., bilateral) or user-configurable iterations.
4. **Performance**:
   * **Limitation**: Large images strain browser resources.
   * **Extension**: Offload computation to WebGL or Web Workers for parallelization.
5. **Undo Scope**:
   * **Limitation**: Image removal clears history, limiting reversibility.
   * **Extension**: Store full stack states, though at higher memory cost.
6. **Gradient Control**:
   * **Limitation**: Fixed opacity and linear interpolation limit expressiveness.
   * **Extension**: Allow user-defined opacity, non-linear gradients, or multi-point color stops.

This manual provides a rigorous exploration of Aquiplicity 2025’s technical underpinnings, highlighting its algorithmic sophistication and potential for advanced image processing research. The system’s blend of real-time interactivity, robust composition logic, and extensible design makes it a compelling platform for further study and development.