Phase 3: Colour Processing Logic

1 Overview

This phase details the architecture of the image processing pipeline found in `core/color_processing.py`. This module is the "Orchestrator." It does not perform the raw FFT math itself; rather, it prepares image data, routes it to the math engine, and reconstructs the final output.

Its primary responsibility is ensuring that **Grayscale** and **Color (RGB)** images are handled correctly, preserving their data types and color fidelity according to the project requirements.

2 The Channel Orchestrator: `_process_channel`

Before discussing specific Grayscale or Color pipelines, we must understand the internal worker function `_process_channel`. This function implements the standard High-Boost sequence for a *single* 2D array. Both the grayscale and color functions call this internally.

```
def _process_channel(channel, mask_builder, mask_kwargs, r, ...):
    # 1. Range Preservation
    orig_dtype = channel.dtype
    ch_float, vmin, vmax = _to_float_and_range(channel)

    # 2. Forward FFT Pipeline
    F = compute_fft(ch_float)
    F_shifted = fft_shift(F)

    # 3. Mask Generation
    shape = F_shifted.shape
    L = filters.build_lowpass_mask(shape=shape, **mask_kwargs)

    # 4. High-Boost Application
    Hb = compose_highboost_mask(L, r)
    G_shifted = F_shifted * Hb  # Apply filter

    # 5. Inverse FFT Pipeline
    G = ifft_shift(G_shifted)
    out_float = compute_ifft(G)

    # 6. Restoration (Clipping & Casting)
    out = _clip_and_cast(out_float, orig_dtype, vmin, vmax)

    return out
```

Key Engineering Detail: Range Preservation Input images are typically `uint8` (0-255). However, the FFT process involves complex numbers and high-precision floats.

- We convert inputs to `float64` before processing to avoid overflow/underflow errors during spectral manipulation.
- After the Inverse FFT, pixel values might exceed 255 (due to boosting) or dip below 0 (overshoot). The function `_clip_and_cast` clamps these values back to the valid range $[0, 255]$ before returning the result.

3 The Grayscale Pipeline

The grayscale pipeline is straightforward because a grayscale image is already a single 2D matrix ($Height \times Width$).

```
def process_grayscale(image, ...):
    if image.ndim != 2:
        raise ValueError("process_grayscale expects a 2D array."
            )

    # Delegate directly to the single-channel processor
    return _process_channel(image, ...)
```

Why this wrapper exists: Even though it seems simple, this wrapper provides a layer of safety. It validates the input shape to ensure we don't accidentally pass a 3D RGB array into the FFT engine, which would result in a mathematically nonsensical "3D Fourier Transform" instead of a spatial image transform.

4 The Color (RGB) Pipeline

Handling color is more complex. A standard color image is a 3D array of shape $(H, W, 3)$. The Fourier Transform of a 3D block is *not* the same as the Fourier Transform of three 2D planes. To preserve color fidelity and prevent "bleeding" between channels, we must isolate them.

```
def process_color_rgb(image_rgb, ...):
    # 1. Validation
    if image_rgb.ndim != 3 or image_rgb.shape[2] != 3:
        raise ValueError("process_color_rgb expects HxWx3 array.
            ")

    chans_out = []

    # 2. Channel Splitting & Loop
    # We iterate 3 times: once for Red, Green, and Blue
    for idx in range(3):
        # Slicing: Extract the 2D plane for the current color
        ch = image_rgb[:, :, idx]

        # Process this specific color plane independently
        ch_out = _process_channel(ch, ...)

        chans_out.append(ch_out)

    # 3. Channel Merging (Stacking)
    # Re-assemble the three processed 2D arrays back into a 3D
        block
    stacked = np.stack(chans_out, axis=2)

```

```
23      return stacked
```

Deep Dive: Why Split Channels?

- **Independence:** Edges in the Red channel might not exist in the Blue channel. By separating them, we allow the High-Boost filter to sharpen red details based solely on red frequency information.
- **Mathematical Correctness:** `fft2` operates on the last two axes. Explicitly looping ensures we are performing exactly three distinct 2D transforms, which is the standard method for color image processing.

5 Advanced Feature: sRGB Linearization

The project includes an optional "sRGB Linearize" feature in the GUI. This handles the Gamma Correction problem.

```
1  def srgb_to_linear(img):
2      # Convert non-linear sRGB (gamma ~2.2) to Linear Light
3      return np.where(img <= 0.04045, img / 12.92, ((img + 0.055)
           / 1.055) ** 2.4)
4
5  def linear_to_srgb(img_lin):
6      # Convert Linear Light back to sRGB for display
7      return np.where(img_lin <= 0.0031308, ...)
```

The Logic:

1. **Input:** Standard image files are Gamma Encoded (non-linear).
2. **The Problem:** Mathematically adding "Boost" to a non-linear pixel value is physically inaccurate. It can shift perceived hues and brightness incorrectly.
3. **The Fix:**
   - **Step 1:** Convert 0-255 pixels to 0.0-1.0 floats.
   - **Step 2:** Apply `srgb_to_linear` to undo the gamma curve.
   - **Step 3:** Run the High-Boost Filter in this "Linear Physics" space.
   - **Step 4:** Apply `linear_to_srgb` to re-encode the result for the monitor.

This ensures that the "energy" added by the boost factor corresponds to actual physical light intensity, resulting in more natural-looking edges with fewer dark halo artifacts.