ImagoCrafter

Radu Daniel - Dumitru

June 25, 2025

Contents

Chapter 1

Introduction

ImagoCrafter is an image processing application built in C# on .NET 8.0. The goal of this project is to deliver a flexible, modular library of filters and processing techniques based on convolution kernels and statistical methods, all accessible via a command-line interface.

Chapter 2

System Architecture

The project is organized into four main modules:

- **Core**: Fundamental classes for image representation, loading, and saving.
- **Processing**: The `IImageProcessor` interface and its implementations for different filters.
- **Kernels**: Classes responsible for creating and normalizing convolution kernels.
- **Program**: The command-line entry point that ties everything together.

Chapter 3

Modules and Components

3.1 Core

In the `ImagoCrafter.Core` namespace:

Image The `Image` class serves as a container for pixel data, keeping track of width, height, and the number of channels. It provides methods to read and write individual pixel components (R, G, B) both as bytes (0–255) and normalized floats (0.0–1.0).

Key algorithms:

1. Pixel access using linear memory layout:

$$index = (y \cdot width + x) \cdot channels + channel$$

2. Safe coordinate handling:

$$x_{safe} = clamp(x, 0, width - 1)$$

$$y_{safe} = clamp(y, 0, height - 1)$$

3. Color space normalization:

$$value_{float} = value_{byte}/255.0$$

$$value_{byte} = value_{float} \cdot 255.0$$

The linear memory layout ensures efficient access while the safe variants prevent buffer overflows.

ImageLoader `ImageLoader` hides the complexity of the ImageSharp library, loading files into `Image<Rgb24>` and converting them into a linear byte array for processing.

Key algorithms:

1. Image loading:
   (a) Load image using ImageSharp
   (b) Allocate linear byte array of size $width \cdot height \cdot channels$

(c) Copy RGB components using:

$$destIndex = (y \cdot width + x) \cdot 3$$

2. Image saving:

   (a) Create ImageSharp RGB24 image

   (b) Copy components using channel mapping:

$$r = data[srcIndex]$$

$$g = channels \geq 2?data[srcIndex + 1] : r$$

$$b = channels \geq 3?data[srcIndex + 2] : r$$

This approach ensures color integrity and consistent channel order.

Edge Handling When accessing pixels near image boundaries, the following algorithms are used:

1. Clamp-to-edge:

$$x = max(0, min(x, width - 1))$$

$$y = max(0, min(y, height - 1))$$

2. Edge pixel replication for convolution:

$$pixel_{safe}(x, y) = pixel(clamp(x, 0, w - 1), clamp(y, 0, h - 1))$$

This ensures that operations near image edges produce valid results without introducing artifacts.

3.2 Processing

IImageProcessor This interface defines the contract for all filters: a `Process` method that transforms an input `Image` and returns a new one, and a `Configure` method that accepts dynamic parameters to adjust behavior without recompilation.

3.3 Filters

3.3.1 ConvolutionProcessor

Applies any two-dimensional kernel defined by a `ConvolutionKernel`. For each pixel:

$$output(x, y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} input(x + i, y + j) \cdot kernel(i + r, j + r) \cdot factor + bias$$

where r is the kernel radius. The process includes:

1. Edge handling using clamp-to-edge
2. Optional kernel normalization ensuring $\sum kernel = 1$
3. Factor and bias adjustments for contrast control

3.3.2 GaussianBlurProcessor

Implements a separable Gaussian blur by performing two passes—horizontal and vertical—using a one-dimensional kernel generated from *sigma*. This approach reduces complexity from $O(r^2)$ to $O(2r)$ where r is the kernel radius.

The 1D Gaussian kernel is generated as:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

The kernel size is determined by σ:

$$size = max(3, \lfloor 6\sigma \rfloor | 1)$$

The process applies two passes:

1. Horizontal blur using the 1D kernel
2. Vertical blur on the result of step 1

3.3.3 VignetteProcessor

Darkens image edges based on distance from the center using a radial darkening effect:

1. Calculate normalized distance from center:

$$dx = \frac{x - centerX}{centerX}$$

$$dy = \frac{y - centerY}{centerY}$$

$$distance = \sqrt{dx^2 + dy^2}$$

2. Apply falloff function:

$$vignette = 1 - min(1, \frac{distance}{radius})$$

$$vignette = 1 - strength \cdot (1 - vignette^2)$$

3. Scale pixel values:

$$pixel_{out} = pixel_{in} \cdot vignette$$

3.3.4 ResizeProcessor

Resizes images using bilinear interpolation. For each target pixel (x, y) in the output image, the algorithm:

1. Maps target coordinates to source coordinates:

$$srcX = x \cdot \frac{width_{src}}{width_{target}}$$

$$srcY = y \cdot \frac{height_{src}}{height_{target}}$$

2. Finds the four surrounding pixels (x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2)
3. Calculates interpolation weights:

$$xWeight = srcX - \lfloor srcX \rfloor$$

$$yWeight = srcY - \lfloor srcY \rfloor$$

4. Interpolates horizontally for both rows:

$$row1 = p_{11}(1 - xWeight) + p_{21}xWeight$$

$$row2 = p_{12}(1 - xWeight) + p_{22}xWeight$$

5. Interpolates vertically for final value:

$$pixel = row1(1 - yWeight) + row2 \cdot yWeight$$

For upscaling operations, adaptive sharpening is automatically applied:

1. Calculate upscale ratio:

$$ratio = max(\frac{width_{target}}{width_{source}}, \frac{height_{target}}{height_{source}})$$

2. Apply adaptive sharpening strength:

$$strength_{adaptive} = strength_{base} \cdot (ratio - 1.0)$$

Where $ratio > 1.0$ indicates upscaling, and the strength increases linearly with the upscale factor. For example:

- 200
- 300

This adaptive sharpening compensates for the natural softening that occurs during upscaling, with stronger sharpening applied for larger scale factors.

3.3.5 UnsharpMaskProcessor

Implements high-quality image sharpening using the unsharp masking technique:

1. Create a Gaussian blurred version of the original image with radius σ:

$$blur(x, y, c) = G_\sigma * original(x, y, c)$$

where G_σ is the Gaussian kernel and $*$ denotes convolution

2. Calculate edge mask by subtracting blur from original:

$$mask(x, y, c) = original(x, y, c) - blur(x, y, c)$$

3. Apply adaptive sharpening with threshold:

$$output(x, y, c) = \begin{cases} original(x, y, c) + mask(x, y, c) \cdot strength & \text{if } |mask(x, y, c)| > threshold \\ original(x, y, c) & \text{otherwise} \end{cases}$$

4. For upscaling operations, parameters scale with ratio:

$$strength = strength_{base} \cdot (ratio_{scale} - 1.0)$$

$$radius = 0.5 + (ratio_{scale} - 1.0) \cdot 0.5$$

$$threshold = 0.01$$

where $ratio_{scale}$ is the upscaling factor and $threshold$ prevents noise amplification.

The threshold-based masking ensures that only significant edges are enhanced while noise and low-contrast details remain unchanged. The adaptive radius ensures appropriate detail enhancement for different scaling factors.

3.4 Kernels

ConvolutionKernel The base class for all matrix-based filters. Key algorithms:

1. Kernel normalization:

$$sum = \sum_{i=0}^{size-1} \sum_{j=0}^{size-1} kernel[i, j]$$

$$kernel_{normalized}[i, j] = \frac{kernel[i, j]}{sum}$$

2. Value computation:

$$value = value \cdot factor + bias$$

The normalization ensures that the kernel preserves the image's overall brightness when $\sum kernel = 1$.

GaussianKernel Inherits from `ConvolutionKernel` and automatically builds a two-dimensional Gaussian matrix. Key algorithms:

1. Generate kernel values:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

2. Normalize kernel:

$$kernel_{normalized}[i, j] = \frac{kernel[i, j]}{\sum_{i,j} kernel[i, j]}$$

After filling each cell using the normal distribution formula, it normalizes the kernel and sets the factor to 1.

SharpenKernel Implements advanced image sharpening using a high-pass filtering approach:

1. High-pass filter for edge detection:

$$K_{hp} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

2. Combine with identity matrix for controllable sharpening:

$$K_{sharp} = I + strength \cdot \frac{K_{hp}}{8}$$

   where I is the identity matrix and $strength$ controls sharpening intensity

3. For upscaling operations, adaptive strength calculation:

$$strength_{final} = strength_{base} \cdot (ratio_{scale} - 1.0)$$

$$ratio_{scale} = \max(\frac{width_{target}}{width_{source}}, \frac{height_{target}}{height_{source}})$$

4. Final kernel normalization to preserve brightness:

$$sum = \sum_{i,j \neq center} kernel[i,j]$$

$$kernel[center] = 1.0 - sum$$

$$factor = 1.05 \text{ (brightness boost)}$$

The high-pass filter detects edges by computing the Laplacian of the image. The normalized kernel ($\frac{K_{hp}}{8}$) ensures balanced enhancement, while the identity matrix preserves original image content. The brightness boost factor compensates for potential darkening caused by edge enhancement.

Chapter 4

Command-Line Interface

The `Program` class parses console arguments to identify commands (`blur`, `vignette`, `resize`), reads input images, applies the appropriate processor, and writes output files with a ".processed" suffix. Error handling and usage messages guide the user in case of invalid input.

Usage Patterns

blur ImagoCrafter blur <input-file> [sigma]

vignette ImagoCrafter vignette <input-file> [strength] [radius]

resize ImagoCrafter resize <input-file> <width> <height>

Chapter 5

Conclusion

ImagoCrafter demonstrates a modular, extensible framework for image processing. Its clean separation of concerns allows developers to introduce new filters and kernels with minimal changes to existing code.