HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN 1

ĐỀ TÀI:

Xây dựng công cụ chuyển ảnh thành tranh vẽ

Giảng viên hướng dẫn : **Phạm Hoàng Việt**

Nhóm Lớp : **XLA N12**

Nhóm bài tập lớn : **00**

Nhóm sinh viên thực hiện : **Dương Hải Lưu – B22DCCN513**
Nguyễn Nhật Quang – B22DCCN645

Hà Nội – 2025

Mục Lục

Đề tài: Xây dựng công cụ chuyển ảnh thành tranh vẽ

I. CÔNG CỤ VÀ THƯ VIỆN

1. NumPy (numpy)

- Công dụng: Thư viện tính toán khoa học, xử lý mảng đa chiều
- Vai trò: Biểu diễn ảnh dưới dạng ma trận số, thực hiện các phép toán ma trận
- Sử dụng: Lưu trữ và xử lý dữ liệu pixel của ảnh
- Các hàm chính:
  + np.zeros(): Tạo mảng toàn số 0
  + np.pad(): Thêm padding cho ảnh
  + np.sum(): Tính tổng các phần tử
  + np.exp(): Tính hàm mũ e^x
  + np.sqrt(): Tính căn bậc hai
  + np.clip(): Giới hạn giá trị trong khoảng
  + np.random.normal(): Tạo nhiễu Gaussian
  + np.power(): Tính lũy thừa (gamma correction)

2. SciPy (scipy.ndimage)

- Công dụng: Thư viện xử lý ảnh khoa học
- Vai trò: Cung cấp các hàm xử lý ảnh tối ưu
- Sử dụng:
  + scipy.ndimage.zoom(): Thay đổi kích thước ảnh (downsampling/upsampling)
- Lưu ý: Không bắt buộc, có thể dùng phương pháp thủ công thay thế

3. PIL/Pillow (PIL.Image, ImageTk)

- Công dụng: Thư viện xử lý ảnh Python
- Vai trò: Đọc/ghi file ảnh, chuyển đổi định dạng
- Sử dụng:
  + Mở file ảnh từ đĩa

+ Lưu kết quả xử lý

+ Chuyển đổi sang ImageTk để hiển thị trong Tkinter

4. Tkinter (tkinter)

- Công dụng: Thư viện tạo giao diện đồ họa (GUI)
- Vai trò: Tạo cửa sổ ứng dụng, nút bấm, hiển thị ảnh
- Sử dụng:

+ Xây dựng giao diện người dùng thân thiện

+ Button, Frame, Label, Scale để tạo các điều khiển

+ Canvas để hiển thị ảnh

5. Time

- Công dụng: Đo lường thời gian thực thi
- Vai trò: Theo dõi hiệu suất các bước xử lý
- Sử dụng: time.time() để tính thời gian xử lý từng bước

II. QUY TRÌNH XỬ LÝ ẢNH

BƯỚC 1: CHUYỂN ĐỔI SANG ẢNH XÁM (GRAYSCALE)

Hàm: chuyen_rgb_sang_xam()

Công thức:

$$Gray = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

Giải thích:

- Chuẩn: ITU-R BT.601 (chuẩn quốc tế cho chuyển đổi màu)
- Lý do hệ số khác nhau:

+ Mắt người nhạy cảm với màu xanh lá (Green) nhất

→ hệ số cao nhất (0.587)

+ Màu đỏ (Red) → hệ số 0.299

+ Màu xanh dương (Blue) → hệ số thấp nhất (0.114)

- Trong code:

```
b = anh[:, :, 0].astype(np.float32)
g = anh[:, :, 1].astype(np.float32)
```

r = anh[:, :, 2].astype(np.float32)

anh_xam = 0.114 * b + 0.587 * g + 0.299 * r

Công dụng:

- Giảm độ phức tạp từ 3 kênh màu xuống 1 kênh
- Tăng tốc xử lý gấp 3 lần
- Phù hợp cho phát hiện biên vì biên phụ thuộc vào cường độ sáng, không phải màu sắc

BƯỚC 2: ĐẢO NGƯỢC ẢNH XÁM LẦN 1

Hàm: dao_nguoc_anh()

Công thức:

Output_pixel = 255 - Input_pixel

Giải thích:

- Đảo ngược mọi giá trị pixel
- Trắng (255) → Đen (0)
- Đen (0) → Trắng (255)
- Xám (128) → Xám (127)

Công dụng:

- Chuẩn bị cho bước làm mờ
- Tạo hiệu ứng âm bản (negative)
- Là bước trung gian quan trọng cho Color Dodge Blending

BƯỚC 3: LÀM MỜ GAUSSIAN

Hàm: lam_mo_gaussian(), tao_kernel_gaussian()

Công thức tạo kernel:

$G(x,y) = \exp(-(x^2 + y^2) / (2 \times \sigma^2))$

Chuẩn hóa:

G_normalized = G / Σ(G)

Tham số mặc định:

- kernel_gaussian = 15: Kích thước kernel (15×15 pixels)
- sigma_gaussian = 3: Độ lệch chuẩn

Giải thích:

- Tạo kernel Gaussian 2D có dạng hình chuông
- Trọng số giảm dần từ tâm ra ngoài theo phân phối Gaussian
- Chuẩn hóa để tổng trọng số = 1 (bảo toàn độ sáng)

Ví dụ kernel 5×5 với σ=1:

[0.003 0.013 0.022 0.013 0.003]

[0.013 0.059 0.097 0.059 0.013]

[0.022 0.097 0.159 0.097 0.022]

[0.013 0.059 0.097 0.059 0.013]

[0.003 0.013 0.022 0.013 0.003]

Cơ chế hoạt động:

- Với mỗi pixel trong ảnh:
  + Lấy vùng lân cận kích thước kernel
  + Nhân từng pixel trong vùng với trọng số tương ứng
  + Cộng tất cả lại
  + Kết quả là giá trị mới của pixel trung tâm

Công dụng:

- Làm mờ ảnh, giảm nhiễu
- Làm mịn các chi tiết nhỏ
- Tạo hiệu ứng mờ đều toàn bộ ảnh (khác với Bilateral Filter)
- Chuẩn bị cho các bước xử lý tiếp theo

BƯỚC 4: LÀM MỊN BILATERAL FILTER (TỐI ƯU)

Hàm: bo_loc_song_phuong_toi_uu()

Công thức:

Trọng số không gian:

$w\_spatial(i,j) = \exp(-(i^2 + j^2) / (2 \times \sigma\_spatial^2))$

Trọng số màu:

$w\_color = \exp(-(I\_neighbor - I\_center)^2 / (2 \times \sigma\_color^2))$

Trọng số tổng hợp:

$w\_total = w\_spatial \times w\_color$

Pixel kết quả:

$I\_result[i,j] = \Sigma(w\_total \times I\_neighbor) / \Sigma(w\_total)$

Tham số mặc định:

- d = 5: Đường kính vùng lân cận (5×5 pixels)
- sigma_mau = 50: Độ lệch chuẩn màu (kiểm soát độ mịn theo cường độ)
- sigma_khong_gian = 50: Độ lệch chuẩn không gian

Giải thích chi tiết:

- Trọng số không gian (Spatial weight):
  + Pixel gần trung tâm có trọng số cao hơn
  + Giống như Gaussian blur
  + Pre-computed 1 lần để tăng tốc
- Trọng số màu (Range weight):
  + Pixel có cường độ tương tự trung tâm có trọng số cao
  + Đây là điểm khác biệt chính với Gaussian blur
  + Bảo toàn biên vì pixel khác màu có trọng số thấp
- Ví dụ thực tế:

Nếu đang xét pixel sáng (200) trong vùng chứa:

+ Pixel sáng tương tự (195, 205) → trọng số cao

+ Pixel tối (50, 60) → trọng số rất thấp

→ Pixel kết quả vẫn sáng, biên được bảo toàn

Các kỹ thuật tối ưu hóa:

a) Downsampling tự động (ảnh > 500px)

- Công thức: ty_le = 500 / max(chieu_cao, chieu_rong)
- Ví dụ: Ảnh 2000×1500 → resize xuống 500×375
- Tăng tốc: x16 (do độ phức tạp $O(n^2)$)

b) Pre-compute spatial weights

- Tính ma trận trọng số không gian 1 lần duy nhất
- Tái sử dụng cho mọi pixel
- Giảm phép tính từ $O(n \times m \times d^2)$ xuống $O(d^2 + n \times m)$

c) Vectorization với NumPy

- Lấy toàn bộ vùng một lần: vung = anh_padding[i:i+d, j:j+d]
- Tính vectorized: chenh_lech = vung - gia_tri_tam
- Nhanh hơn vòng lặp Python 10-100 lần

d) Batch processing

- Xử lý 50 dòng một lần
- Giảm overhead của vòng lặp
- Cache-friendly memory access

e) Upsampling kết quả

- Phóng to ảnh về kích thước gốc
- Dùng scipy.ndimage.zoom() hoặc phương pháp thủ công

Công dụng:

- Làm mịn nhiễu trong ảnh (giảm các điểm nhiễu nhỏ)

- Bảo toàn biên: Không làm mờ các cạnh sắc nét
- Tạo hiệu ứng "cartoonize": Làm phẳng các vùng đồng nhất
- Giúp Sobel filter hoạt động tốt hơn bằng cách giảm nhiễu
- Khác với Gaussian blur: Giữ được các chi tiết quan trọng

BƯỚC 5: ĐẢO NGƯỢC ẢNH ĐÃ LÀM MỜ

Hàm: dao_nguoc_anh()

Công thức:

Output_pixel = 255 - Input_pixel

Công dụng:

- Chuyển từ ảnh âm bản về dạng dương
- Chuẩn bị cho Color Dodge Blending
- Tạo điều kiện cho việc tạo hiệu ứng phác thảo

BƯỚC 6: PHÁT HIỆN CẠNH SOBEL

Hàm: phat_hien_canh()

Kernel Sobel:

Sobel Gx (phát hiện biên dọc): Sobel Gy (phát hiện biên ngang):

[-1 0 1] [-1 -2 -1]

[-2 0 2] [0 0 0]

[-1 0 1] [1 2 1]

Công thức:

Gradient theo X:

$Gx[i,j] = \Sigma \Sigma vung[m,n] \times Sobel\_Gx[m,n]$

Gradient theo Y:

$Gy[i,j] = \Sigma \Sigma vung[m,n] \times Sobel\_Gy[m,n]$

Độ lớn gradient:

$G = \sqrt{(Gx^2 + Gy^2)}$

Tăng cường:

$G_{enhanced} = G \times 2.5$

Giải thích chi tiết:

- Sobel Gx:
  + Phát hiện thay đổi cường độ theo hướng ngang (X)
  + Tìm các cạnh dọc (vertical edges)
  + Hệ số -2, -1, 1, 2 tạo độ nhạy cao ở trung tâm
  + Cột giữa là 0 → không ảnh hưởng
- Sobel Gy:
  + Phát hiện thay đổi cường độ theo hướng dọc (Y)
  + Tìm các cạnh ngang (horizontal edges)
  + Hàng giữa là 0 → không ảnh hưởng
- Độ lớn gradient:
  + Kết hợp cả hai hướng để có độ mạnh tổng thể của biên
  + Dùng định lý Pythagoras: $\sqrt{(Gx^2 + Gy^2)}$
  + Cho kết quả không phụ thuộc vào hướng của cạnh
- Tăng cường cạnh:
  + Nhân với 2.5 để làm cạnh rõ nét hơn
  + Tạo nét vẽ đậm hơn cho phác thảo

Ví dụ thực tế:

Với vùng ảnh 3×3:

[50 50 50]

[50 50 200]

[50 50 200]

Gx tại tâm:

= -1×50 + 0×50 + 1×50 + (-2)×50 + 0×50 + 2×200 + (-1)×50 + 0×50 + 1×200

= 350 (phát hiện cạnh dọc mạnh)

Gy tại tâm:

= -1×50 + (-2)×50 + (-1)×50 + 0×50 + 0×50 + 0×200 + 1×50 + 2×50 + 1×200

= 150 (phát hiện cạnh ngang yếu hơn)

G = √(350² + 150²) ≈ 381

Công dụng:

- Phát hiện đường viền của các đối tượng trong ảnh
- Tìm các vùng thay đổi cường độ đột ngột (biên)
- Tạo hiệu ứng phác họa: Biên → nét vẽ bút chì
- Sobel tốt hơn các toán tử đơn giản nhờ:
  + Có trọng số (hệ số 2 ở giữa)
  + Kết hợp làm mịn và đạo hàm
  + Ít nhạy với nhiễu hơn

BƯỚC 7: COLOR DODGE BLENDING

Hàm: tron_mau_color_dodge()

Công thức:

blend_inverted = 255 - blend

Nếu blend_inverted == 0:

result = 255

Ngược lại:

result = (base / blend_inverted) × 255

Cuối cùng:

result = clip(result, 0, 255)

Giải thích:

- Đây là blending mode được sử dụng trong Photoshop
- Công thức toán học:
  + Đảo ngược ảnh blend (tạo mặt nạ)
  + Chia ảnh base cho mặt nạ
  + Nhân với 255 để chuẩn hóa
- Hiệu ứng:

+ Vùng blend tối (0) → blend_inverted cao (255) → result gần base

+ Vùng blend sáng (255) → blend_inverted thấp (0) → result = 255

+ Tạo hiệu ứng "dodge" (làm sáng) giống trong phòng tối ảnh

- Ví dụ:

base = 100, blend = 200

→ blend_inverted = 55

→ result = (100/55) × 255 = 463.6 → clip = 255 (rất sáng)

base = 100, blend = 50

→ blend_inverted = 205

→ result = (100/205) × 255 = 124.4 (tăng sáng nhẹ)

Công dụng:

- Kết hợp ảnh xám gốc với ảnh mờ đảo ngược
- Tạo hiệu ứng phát sáng, làm nổi bật các vùng sáng
- Giảm bóng tối, tăng độ sáng tổng thể
- Tạo hiệu ứng giống tranh phác thảo bút chì
- Làm nền trắng, nét vẽ rõ ràng

BƯỚC 8: KẾT HỢP NÉT VẼ CẠNH

Hàm: nhan_hai_anh()

Công thức:

1. Đảo ngược cạnh:

   canh_dao = 255 - canh

2. Chuẩn hóa:

   canh_dao_chuan_hoa = canh_dao / 255.0

3. Power transformation (gamma correction):

   canh_dao_chuan_hoa = canh_dao_chuan_hoa^0.6

4. Nhân với phác thảo:

   phac_thao = (phac_thao / 255.0) × canh_dao_chuan_hoa

   phac_thao = phac_thao × 255

Giải thích:

- Bước 1: Đảo cạnh để cạnh mạnh → giá trị cao
- Bước 2: Chuẩn hóa về [0,1] để tính toán
- Bước 3: Gamma correction với γ=0.6
  + γ < 1: Làm sáng các giá trị tối
  + Công thức: output = input^γ
  + Ví dụ: 0.5^0.6 ≈ 0.66 (tăng từ 0.5 lên 0.66)
  + Làm nét vẽ đậm hơn, rõ ràng hơn
- Bước 4: Nhân element-wise
  + Vùng không có cạnh (=1) → giữ nguyên phác thảo
  + Vùng có cạnh (nhỏ hơn 1) → làm tối phác thảo → tạo nét vẽ

Ví dụ thực tế:

phac_thao = 200 (sáng)

canh = 100 (cạnh yếu)

→ canh_dao = 155

→ canh_dao_chuan_hoa = 155/255 = 0.608

→ sau gamma: 0.608^0.6 = 0.716

→ phac_thao_moi = 200 × 0.716 = 143.2 (tối hơn một chút)

phac_thao = 200 (sáng)

canh = 200 (cạnh mạnh)

→ canh_dao = 55

→ canh_dao_chuan_hoa = 55/255 = 0.216

→ sau gamma: 0.216^0.6 = 0.372

→ phac_thao_moi = 200 × 0.372 = 74.4 (tối đậm → nét vẽ rõ)

Công dụng:

- Làm đậm các nét vẽ cạnh
- Tạo độ tương phản cao giữa nét vẽ và nền
- Làm cho phác thảo giống vẽ tay hơn
- Kết hợp thông tin từ cả color dodge và edge detection

BƯỚC 9: ĐIỀU CHỈNH CONTRAST & BRIGHTNESS VÀ THÊM NHIỄU

Hàm: dieu_chinh_tuong_phan()

Công thức Contrast:

output = (input - 128) × contrast_factor + 128

Tham số mặc định:

- tuong_phan = 1.1: Tăng tương phản 10%
- do_sang = 50: Tăng sáng 50 đơn vị

Công thức Brightness:

output = input + brightness

Công thức Noise:

noise = random.normal(mean=0, std=2, shape)

output = input + noise

Giải thích chi tiết:

- Điều chỉnh Contrast:
  + Công thức hoạt động quanh điểm neo 128 (giữa thang xám)
  + contrast > 1.0: Tăng khoảng cách với 128
    * Pixel sáng (>128) → sáng hơn
    * Pixel tối (<128) → tối hơn
    * Kết quả: Ảnh có độ tương phản cao hơn
  + Ví dụ với contrast=1.1:
    * input=200 → (200-128)×1.1+128 = 207.2 (sáng hơn)

* input=50 → (50-128)×1.1+128 = 42.2 (tối hơn)

- Điều chỉnh Brightness:

+ Cộng trực tiếp vào mọi pixel

+ brightness=50 → mọi pixel sáng thêm 50 đơn vị

+ Làm cho ảnh sáng hơn tổng thể

- Thêm Nhiễu Gaussian:

+ Phân phối chuẩn: N(0, 2)

+ mean=0: Trung bình nhiễu = 0 (không làm đổi độ sáng trung bình)

+ std=2: Độ lệch chuẩn nhỏ (nhiễu nhẹ)

+ Mục đích:

* Tạo texture tự nhiên như giấy

* Phá vỡ độ mịn quá mức

* Giống giấy vẽ thực tế hơn

+ 68% nhiễu nằm trong [-2, +2]

+ 95% nhiễu nằm trong [-4, +4]

Ví dụ thực tế:

Pixel ban đầu = 150

Sau contrast (1.1):

→ (150-128)×1.1+128 = 152.2

Sau brightness (+50):

→ 152.2+50 = 202.2

Sau noise (giả sử +1.5):

→ 202.2+1.5 = 203.7

Clip về [0,255]:

→ 204 (uint8)

Công dụng:

- Contrast: Làm rõ nét, tăng độ sắc nét của nét vẽ

- Brightness: Làm sáng tổng thể, giống giấy trắng hơn

- Noise: Tạo texture tự nhiên, giống giấy vẽ thật

III. CÁC PHƯƠNG PHÁP HỖ TRỢ

1. THAY ĐỔI KÍCH THƯỚC ẢNH - BILINEAR INTERPOLATION

Hàm: thay_doi_kich_thuoc_anh()

Công thức:

Tỷ lệ scale:

ty_le_y = chieu_cao_goc / chieu_cao_moi

ty_le_x = chieu_rong_goc / chieu_rong_moi

Tìm vị trí trong ảnh gốc:

nguon_y = i × ty_le_y

nguon_x = j × ty_le_x

Tìm 4 điểm lân cận:

y0 = floor(nguon_y), y1 = y0 + 1

x0 = floor(nguon_x), x1 = x0 + 1

Trọng số:

dy = nguon_y - y0

dx = nguon_x - x0

Bilinear interpolation:

f(x,y) = f(x0,y0)×(1-dx)×(1-dy) + f(x1,y0)×dx×(1-dy) +

f(x0,y1)×(1-dx)×dy + f(x1,y1)×dx×dy

Giải thích:

- Dùng 4 điểm lân cận để tính giá trị pixel mới
- Trọng số phụ thuộc vào khoảng cách
- Cho kết quả mịn hơn nearest neighbor

Công dụng:

- Downsampling: Giảm kích thước để tăng tốc

- Upsampling: Phóng to kết quả về kích thước gốc

2. CONVOLUTION 2D

Hàm: ap_dung_tich_chap()

Công thức:

$$\text{Output}[i,j] = \sum_m \sum_n \text{Input}[i+m, j+n] \times \text{Kernel}[m,n]$$

Giải thích:

- Zero-padding: Thêm viền xung quanh ảnh (reflect mode)

- Sliding window: Trượt kernel qua từng vị trí

- Element-wise multiplication: Nhân từng phần tử

- Sum: Cộng tất cả lại

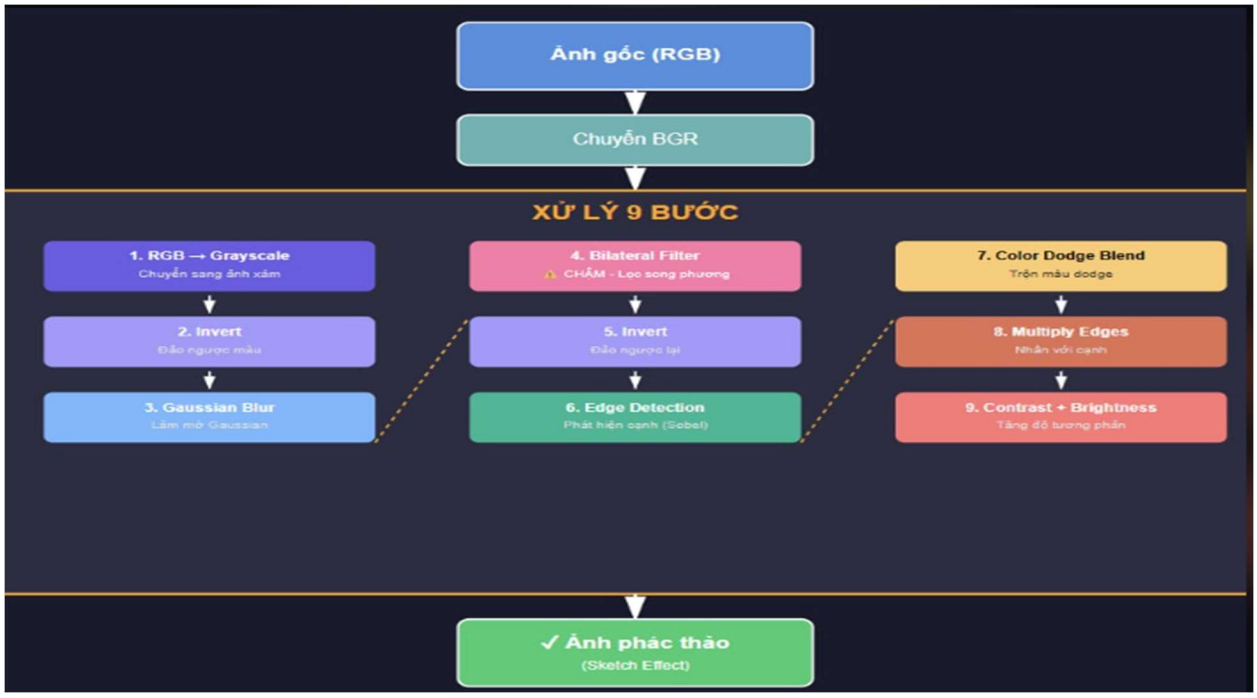
Công dụng:

- Là nền tảng cho mọi bộ lọc

- Áp dụng các phép biến đổi cục bộ

- Sử dụng cho làm mịn, phát hiện biên, làm sắc nét

IV. LUỒNG XỬ LÝ HOÀN CHỈNH



V. ƯU ĐIỂM CỦA PHƯƠNG PHÁP

1. Code thủ công (From Scratch)
   - Không phụ thuộc OpenCV hay thư viện nặng
   - Hiểu rõ từng bước xử lý
   - Dễ tùy chỉnh và cải tiến
   - Có thể thay thế scipy nếu cần
2. Bilateral Filter với tối ưu hóa
   - Làm mịn nhưng không mất biên
   - Tạo hiệu ứng "cartoon" tự nhiên
   - Giảm nhiễu hiệu quả
   - Downsampling tự động tăng tốc x16
   - Pre-compute spatial weights
   - Vectorization với NumPy
   - Batch processing (50 dòng/lần)
3. Sobel Operator
   - Nhanh và hiệu quả
   - Kết hợp cả hai hướng gradient
   - Phát hiện biên chính xác
   - Có trọng số trung tâm
   - Ít nhạy với nhiễu
4. Color Dodge Blending
   - Tạo hiệu ứng phát sáng tự nhiên
   - Giống Photoshop blending mode
   - Làm nền trắng, nét vẽ rõ ràng
5. Kết quả
   - Ảnh phác thảo rõ nét
   - Nét vẽ liền mạch
   - Giống tranh vẽ bút chì thực tế
   - Có texture giấy tự nhiên (nhờ noise)

VI. SO SÁNH VỚI CÁC PHƯƠNG PHÁP KHÁC

1. SO VỚI GAUSSIAN BLUR

Gaussian Blur:

- Làm mờ đều cả ảnh
- Mất biên, cạnh bị mờ
- Nhanh ($O(n \times k^2)$)
- Đơn giản, chỉ phụ thuộc khoảng cách

Bilateral Filter:

- Giữ biên, chỉ làm mịn vùng đồng nhất
- Phức tạp hơn ($O(n \times d^2)$)
- Phụ thuộc cả khoảng cách và cường độ
- Kết quả tự nhiên hơn cho phác thảo

Kết luận:

- Gaussian: Phù hợp làm mờ chung
- Bilateral: Phù hợp khi cần giữ chi tiết

2. SO VỚI CANNY EDGE DETECTION

Sobel:

- 2 bước: Gradient X, Y → Magnitude
- Không có non-maximum suppression
- Không có hysteresis thresholding
- Đơn giản, nhanh
- Cạnh dày hơn

Canny:

- 5 bước: Gaussian → Sobel → NMS → Threshold → Hysteresis
- Cạnh mỏng, chính xác hơn
- Phức tạp hơn
- Chậm hơn

Kết luận:

- Sobel: Đủ tốt cho phác thảo (cần cạnh dày)

- Canny: Tốt hơn cho phát hiện biên chính xác

3. SO VỚI PREWITT/ROBERTS

Roberts (2×2):

[1 0] [0 1]

[0 -1] [-1 0]

- Nhỏ nhất, nhanh nhất

- Nhạy với nhiễu

- Ít chính xác

Prewitt (3×3):

[-1 0 1] [-1 -1 -1]

[-1 0 1] [0 0 0]

[-1 0 1] [1 1 1]

- Không có trọng số trung tâm

- Trung bình giản đơn

Sobel (3×3):

[-1 0 1] [-1 -2 -1]

[-2 0 2] [0 0 0]

[-1 0 1] [1 2 1]

- Có trọng số trung tâm (×2)

- Cân bằng tốt giữa độ chính xác và tốc độ

- Ít nhạy với nhiễu hơn Prewitt

Kết luận:

- Roberts: Quá đơn giản

- Prewitt: Trung bình

- Sobel: Lựa chọn tốt nhất cho bài toán này

4. SO VỚI CÁC PHƯƠNG PHÁP KHÁC CHUYỂN ẢNH THÀNH PHÁC THẢO

Phương pháp đơn giản:

- Chỉ dùng edge detection + inversion

- Nhanh nhưng kết quả kém tự nhiên
- Thiếu hiệu ứng ánh sáng

Phương pháp này:

- Kết hợp nhiều kỹ thuật
- Color dodge tạo hiệu ứng ánh sáng
- Bilateral giữ chi tiết
- Gamma correction làm rõ nét
- Noise tạo texture giấy
- Kết quả rất giống vẽ tay

VII. CÁC VẤN ĐỀ VÀ GIẢI PHÁP

1. VẤN ĐỀ: XỬ LÝ CHẬM VỚI ẢNH LỚN

Nguyên nhân:

- Bilateral filter có độ phức tạp $O(n^2)$
- Ảnh 2000×2000 = 4 triệu pixel × 25 (kernel 5×5) = 100M phép tính

Giải pháp:

a) Downsampling tự động

- Resize ảnh xuống 500px trước khi xử lý
- Upsampling sau khi xong
- Tăng tốc x16 với minimal quality loss

b) Pre-compute spatial weights

- Tính 1 lần, dùng cho mọi pixel
- Giảm phép tính lặp lại

c) Vectorization

- Dùng NumPy thay vì vòng lặp Python
- Tăng tốc 10-100 lần

d) Batch processing

- Xử lý nhiều dòng cùng lúc
- Tốt cho cache

2. VẤN ĐỀ: CẠNH QUÁ YẾU HOẶC QUÁ MẠNH

Giải pháp:

- Điều chỉnh hệ số tăng cường (hiện tại: 2.5)
- Điều chỉnh sigma của Bilateral filter
- Điều chỉnh gamma correction (hiện tại: 0.6)

3. VẤN ĐỀ: ẢNH QUÁ TỐI HOẶC QUÁ SÁNG

Giải pháp:

- Điều chỉnh độ sáng (brightness)
- Điều chỉnh độ tương phản (contrast)
- Cả hai đều có tham số điều chỉnh

4. VẤN ĐỀ: KẾT QUẢ KHÔNG TỰ NHIÊN

Giải pháp:

- Thêm nhiễu Gaussian nhẹ (N(0,2))
- Tạo texture giấy tự nhiên
- Không quá mịn, không quá sắc nét

VIII. KẾT LUẬN

Phương pháp này sử dụng kết hợp 9 kỹ thuật xử lý ảnh:

1. Chuyển đổi màu (RGB → Grayscale)
2. Đảo ngược ảnh (Inversion)
3. Làm mờ Gaussian
4. Làm mịn Bilateral (bảo toàn biên)
5. Phát hiện cạnh Sobel
6. Color Dodge Blending
7. Gamma Correction
8. Điều chỉnh Contrast & Brightness
9. Thêm nhiễu Gaussian

Các tối ưu hóa chính:

- Downsampling/Upsampling tự động
- Pre-computation

- Vectorization với NumPy
- Batch processing

Kết quả:

- Ảnh phác thảo bút chì tự nhiên
- Nét vẽ rõ ràng, liền mạch
- Có texture giấy
- Giống tranh vẽ tay thực tế

Thời gian xử lý:

- Ảnh nhỏ (<500px): 2-5 giây
- Ảnh trung bình (500-1000px): 5-15 giây
- Ảnh lớn (>1000px): Tự động downsampling, vẫn 5-15 giây

Ưu điểm nổi bật:

- Code từ đầu, không phụ thuộc OpenCV
- Hiểu rõ từng bước
- Dễ tùy chỉnh tham số
- Kết quả chất lượng cao
- Có tối ưu hóa hiệu suất