ICT 2403– Graphics and Image Processing

Edge Detection and Cartooning of an Image

A.M.C.B. Adikari (ICT/19/20/003)

A.N. Warnasooriya (ICT/19/20/139)

A.H.T.N. Premachandra (ICT/19/20/085)

B.K.K.S. Rodrigo (ICT/19/20/095)

D.M.I.P.B. Gunasinhe (ICT/19/20/040)

R.M.V.P.B. Udagama (ICT/19/20/138)

L.G.R.J. Lindapitiya (ICT/19/20/132)

Introduction

Image processing techniques like edge detection and image cartooning are critical in translating raw visual data into useful and compelling representations. The edges include a great deal of the information in the image. The edges give information about the locations of the objects, their sizes, their shapes and information concerning their texture. Edge detection is a process to detect such locations from images in terms of pixels where their intensity changing is abruptly. As we know there are many types of images such as medical images, satellite images, articular images, industrial images, general purpose images etc. edge detection is important in any type of image for specific tasks like image segmentation for medical diagnosis and object recognition.

Image cartooning, on the other hand, has grown in popularity as a creative way of transforming an image into its cartoon effect. Image cartooning minimizes complex color palettes, highlights sharp outlines, and adds a new creative dimension to visual content by combining edge detection with sophisticated filtering techniques. It has clean edges, smooth color, and relatively basic textures that demonstrate significance for texture description based on loss function utilized in existing techniques. Today, we can find a plethora of applications on the internet that will convert images to a cartoon effect. This is very important in digital entertainment, where it aids in the creation of compelling graphics and increases the attraction of multimedia content.

We go into the technical complexities of edge detection and image cartooning in this report. Through the implementation of the Sobel operator, we aim to explore these processes. We discuss the methods involved in these transformations and present the visual effects that follow from these operations using OpenCV and NumPy. This report underscores the significance of these techniques in both technical and artistic contexts, elucidating their impact on diverse fields.

As we progress, we will discuss the unique aspects of each process, elaborate on the underlying python code implementations, and present visual evidence of the results. Through the examination of edge detection and image cartooning, we hope to provide a comprehensive insight into the

power of image processing techniques in enhancing the interpretation and artistic expression of visual data.

Task 01: Edge Detection

Edge Detection Concept:

Edge detection is a fundamental image processing technique used to identify boundaries or transitions between different objects or regions within an image. It plays a important role in feature extraction, object recognition, and image segmentation. The idea of edge detection is to recognize abrupt changes or discontinuities in pixel intensity levels. These variations are frequently suggestive of object boundaries, corners, and other important properties.

The importance of edge detection rests in its capacity to improve visual information by emphasizing the most important elements. We can simplify complex images, reduce noise, and improve subsequent analytic processes by isolating edges. This is especially valuable in medical imaging, where accurate segmentation of organs and anomalies is critical, and in computer vision systems that require object recognition and tracking.

Sobel Operator:

The Sobel operator is a common edge detection filter that computes the gradient of an image by convolving it with two separate masks, typically in the horizontal and vertical directions. These masks highlight the changes in pixel intensities in the respective directions, effectively identifying edges.

Sobel operator for edge detection

- It works by calculating the gradient of image intensity at each pixel within the image.
- It finds the direction of the largest increase from light to dark and the rate of change in that direction.
- The result shows how abruptly or smoothly the image changes at each pixel, and therefore how likely it is that that pixel represents an edge. It also shows how that edge is likely to be oriented.
- The result of applying the filter to a pixel in a region of constant intensity is a zero vector.
- The result of applying it to a pixel on an edge is a vector that points across the edge from darker to brighter values.
- The sobel filter uses two 3 x 3 kernels. One for changes in the horizontal direction, and one for changes in the vertical direction.
- The two kernels are convolved with the original image to calculate the approximations of the derivatives. If we define Gx and Gy as two images that contain the horizontal and vertical derivative approximations respectively, the computations are:

-1	0	+1
-2	0	+2
-1	0	+1

Gx

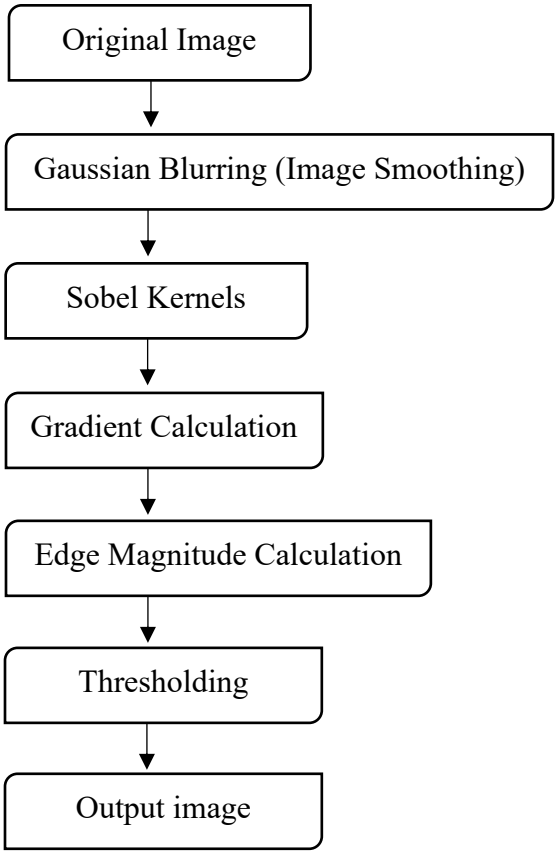
+1	+2	+1
0	0	0
-1	-2	-1

Gy

Gaussian Smoothing Filter:

Before applying the Sobel operator, it's often beneficial to employ Gaussian smoothing. Gaussian smoothing is the process of convolving an image with a Gaussian kernel to decrease noise and high-frequency distortions. By reducing unnecessary variations in pixel values, this preprocessing step aids in acquiring more precise edge information. The effect of Gaussian smoothing is to blur an image, in a similar fashion to the mean filter. The degree of smoothing is determined by the standard deviation of the Gaussian.

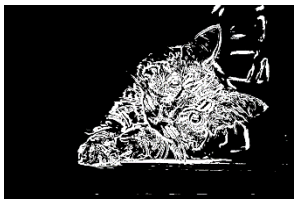
Steps in Edge Detection using the Sobel Operator:



Img 01 – Original image

Img 02 – Blurred image

Img 03 – Edge Magnitude



Img 04 – Thresholded image

1. Image Smoothing: The first step involves applying Gaussian smoothing to the input image. This is done to reduce noise and unwanted high-frequency details.

2. Gradient Calculation: The Sobel operator is applied to the smoothed image to calculate the gradients in both the horizontal and vertical directions. These gradients represent the rate of change of pixel intensities.

3. Gradient Magnitude and Direction: The magnitude of the gradient is calculated using the horizontal and vertical gradients. The direction of the gradient is also determined.

4. Edge Thresholding: By setting appropriate threshold values on the gradient magnitude, edges can be identified as locations where the gradient magnitude exceeds the threshold.

5. Non-Maximum Suppression: This step involves suppressing non-maximum values in the gradient direction to thin out the edges and keep only the most significant ones.

6. Hysteresis Thresholding: A double thresholding technique is often used to identify weak and strong edges. This helps to connect weak edges that are part of a strong edge.

Python code with using Gaussian filter

```
import cv2
import numpy as np

image_path = 'cat.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

blurred_image = cv2.GaussianBlur(image, (5, 5), 0)

sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

gradient_x = cv2.filter2D(blurred_image, cv2.CV_64F, sobel_x)
gradient_y = cv2.filter2D(blurred_image, cv2.CV_64F, sobel_y)

edge_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
edge_magnitude = np.uint8(edge_magnitude)
```

```
threshold_value = 50
edge_thresholded = cv2.threshold(edge_magnitude, threshold_value, 255,
cv2.THRESH_BINARY)[1]

cv2.imshow('Original Image', image)
cv2.imshow('Blurred Image', blurred_image)
cv2.imshow('Edge Magnitude', edge_magnitude)
cv2.imshow('Edge Thresholded', edge_thresholded)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

1. Import Libraries:

- The `cv2` library is imported, which is OpenCV for Python.

- The `numpy` library is imported, used for numerical computations.

2. Read the Image:

- An image named 'cat.jpg' is read using the `cv2.imread()` function. It's read in grayscale mode using the `cv2.IMREAD_GRAYSCALE` flag.

3. Sobel Kernels:

- Two 3x3 Sobel kernels are defined: `sobel_x` for horizontal gradient detection and `sobel_y` for vertical gradient detection.

4. Gradient Computation: -

The `cv2.filter2D()` function is used to apply the Sobel kernels (`sobel_x` and `sobel_y`) on the blurred image to compute the horizontal and vertical gradients (`gradient_x` and `gradient_y`).

5. Edge Magnitude Calculation:

- The edge magnitude is calculated by taking the square root of the sum of squared gradient values: `edge_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)`.

- The `np.uint8()` function is used to convert the edge magnitude values to an 8-bit unsigned integer format.

7. Thresholding:

- A threshold value of 50 is defined (`threshold_value = 50`).

- The `cv2.threshold()` function is applied to the edge magnitude image to create a binary image (`edge_thresholded`) based on the threshold value. Pixels with values greater than the threshold are set to 255 (white), and pixels below or equal to the threshold are set to 0 (black).

8. Display Results:

- The original image, blurred image, edge magnitude image, and edge thresholded image are displayed using the `cv2.imshow()` function.

- The `cv2.waitKey(0)` function waits for a key press, and then the `cv2.destroyAllWindows()` function closes all open windows.

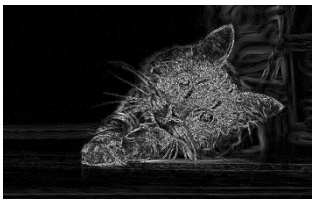
Python code without using Gaussian smoothing filter

```
import cv2
import numpy as np

image_path = 'cat.jpg'
image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)

sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
sobel_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])

gradient_x = cv2.filter2D(image, cv2.CV_64F, sobel_x)
gradient_y = cv2.filter2D(image, cv2.CV_64F, sobel_y)

edge_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
edge_magnitude = np.uint8(edge_magnitude)

threshold_value = 50
edge_thresholded = cv2.threshold(edge_magnitude, threshold_value, 255,
cv2.THRESH_BINARY)[1]

cv2.imshow('Original Image', image)
cv2.imshow('Edge Magnitude', edge_magnitude)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Comparison: Edge Detection with and without Smoothing:

The comparison between edge detection with and without Gaussian smoothing highlights the importance of this preprocessing step. Applying edge detection without smoothing can lead to noisy and spurious edge responses, while edge detection after smoothing provides cleaner and more accurate edge information.

Img 05 – Edge Detection with Smoothing



Img 05 – Edge Detection without Smoothing

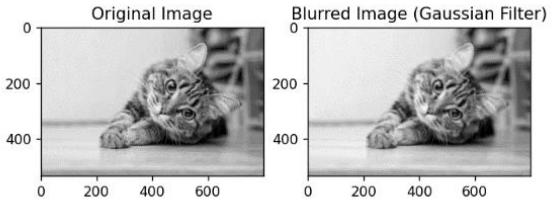
Edge Detection without Gaussian Smoothing:

When edge detection is performed directly on the original image without smoothing, it can result in noisy and unreliable edge responses. Noise in the image can be mistaken for edges, leading to spurious or false detections. The edges might appear fragmented or jagged due to the noise, making it harder to interpret the actual object boundaries.

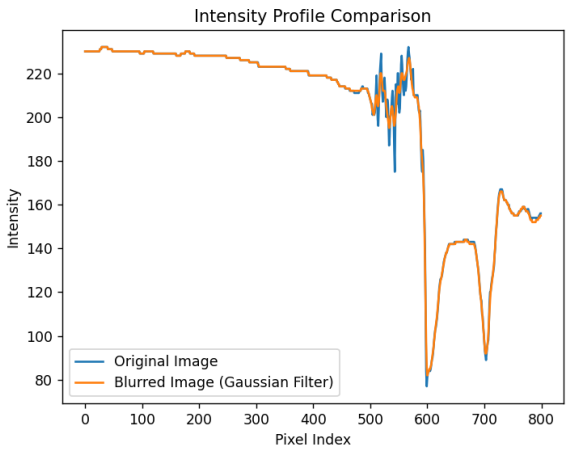
Edge Detection with Gaussian Smoothing:

Applying Gaussian smoothing to the image before edge detection helps to remove noise and unwanted high-frequency details. The smoothed image provides a more consistent and even representation of the image content. When edge detection is performed on the smoothed image, the resulting edges are cleaner and more accurate.

In essence, the preprocessing step of Gaussian smoothing significantly improves the quality and reliability of edge detection. It's like wipe the car windscreen before driving a long journey to see through it – the view becomes clearer.



Img 06 - Original image with Gaussian filter



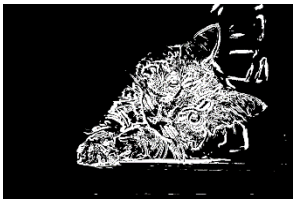
Img 07 - With Gaussian filter intensity value

Sample Images and Results:

Img 08 – Original image

Img 09 – Blurred image

Img 10 – Edge Magnitude



Img 11 – Thresholded image

In the sample images provided, the benefits of applying Gaussian smoothing before edge detection using the Sobel operator become evident. The smoothed images exhibit clearer and more distinct edges, while the unsmoothed images may show jagged and fragmented edge responses.

Task 02: Cartooning of an image

Img 12 – Original image

Img 13 – Cartooned image

What is cartooning?

Cartooning refers to the process of transforming an image into a cartoon-like representation. It involves using particular methods and effects to change the original image, producing a pleasing aesthetic result that is frequently comical or whimsical. People can use cartooning as a creative outlet to express themselves in a special and original way.

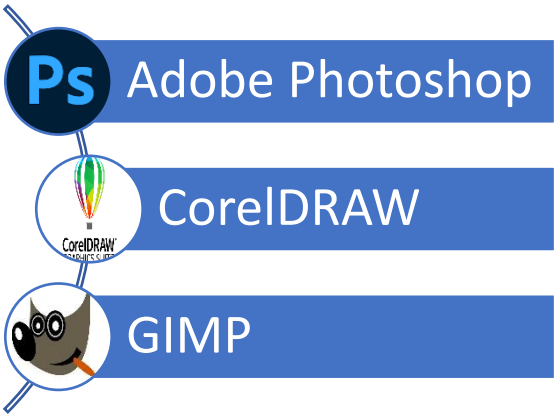
The purpose of cartooning techniques extends beyond mere entertainment. It serves as a means of communication, storytelling, and self-expression in various domains. Cartooning is a flexible and interesting method of visual communication that may be used for a variety of reasons, including personal use, social media profiles, brand promotion, and educational ones.

Cartoons have a long history, with the earliest examples appearing in medieval texts and prehistoric cave drawings. Cartoons have changed and altered over time in response to various artistic trends, cultural influences, and technical developments. They have influenced a lot of popular culture, from political cartoons in newspapers to animated cartoons on television and the internet.

The Cartoonizer-software

It uses algorithms to analyze the original image/video's colors, movement, and lighting to create an interesting cartoon-like effect. The cartoonizer-software plays a crucial role in the process of cartoonization. These tools provide a range of features and options to apply filters, effects, and settings that give images a cartoon-like appearance.

Popular software applications like:



When cartooning images, certain filters, effects, and settings are commonly used to achieve the desired visual style. For example, adjusting brightness and contrast, enhancing outlines, simplifying colors, and adding halftone patterns are some techniques used to create a cartoon effect. Other effects like cell shading, crosshatching, or stippling can also be applied to enhance the cartoon-like qualities of the image.

The Process of Cartooning Images

Two steps are followed when cartooning an image.

1. Edge detection
2. Bilateral filtering

Bilateral Filtering

Introduction:

Bilateral filtering is an advanced image processing method that smooths images while safeguarding edges and details. It stands out by considering both spatial distance and intensity similarity among pixels, emphasizing similarity for smoother transitions between areas. Through a weighted average based on these aspects, bilateral filtering reduces noise while retaining key image features. Its versatility is evident in denoising, texture preservation, and stylization applications. However, it can be resource-intensive for larger filter windows, prompting optimizations like separable kernels. In summary, bilateral filtering is a potent technique, balancing noise reduction and edge preservation for diverse image enhancement tasks.

Steps of Bilateral Filtering:

Bilateral filtering involves several steps to achieve its noise reduction and edge-preserving effects. Here's a breakdown of the typical steps involved in bilateral filtering:

1. **Parameters Setup:**
    - **num_down = 2**: This parameter controls how many times the image will be downsampled.
    - **num_bilateral = 7**: This parameter controls how many iterations of bilateral filtering will be performed.

2. **Downsampling:** The input image is downsampled multiple times using the **cv2.pyrDown()** function. Downscaling the image helps to reduce the computational load of the bilateral filter while maintaining the overall structure.

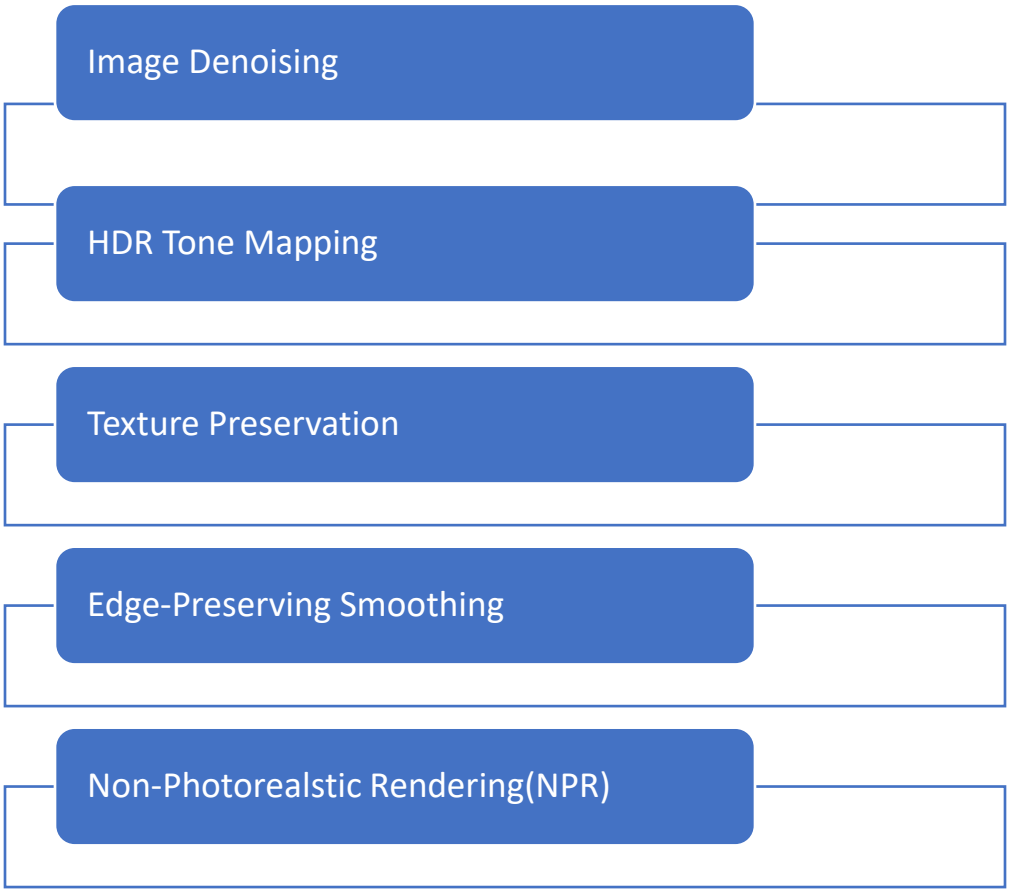
3. **Bilateral Filtering Iterations:** For each iteration in the range of **num_bilateral**, bilateral filtering is applied to the downsampled image.

    a. **Bilateral Filtering Parameters:**
    - **d = 9**: This parameter defines the diameter of each pixel neighborhood used for filtering. Larger values include pixels farther away from the central pixel.
    - **sigmaColor = 50**: This parameter affects the color similarity factor in the filter. A higher value allows more color variation within the neighborhood.
    - **sigmaSpace = 50**: This parameter influences the spatial proximity factor. A higher value results in a wider spatial range being considered during filtering.

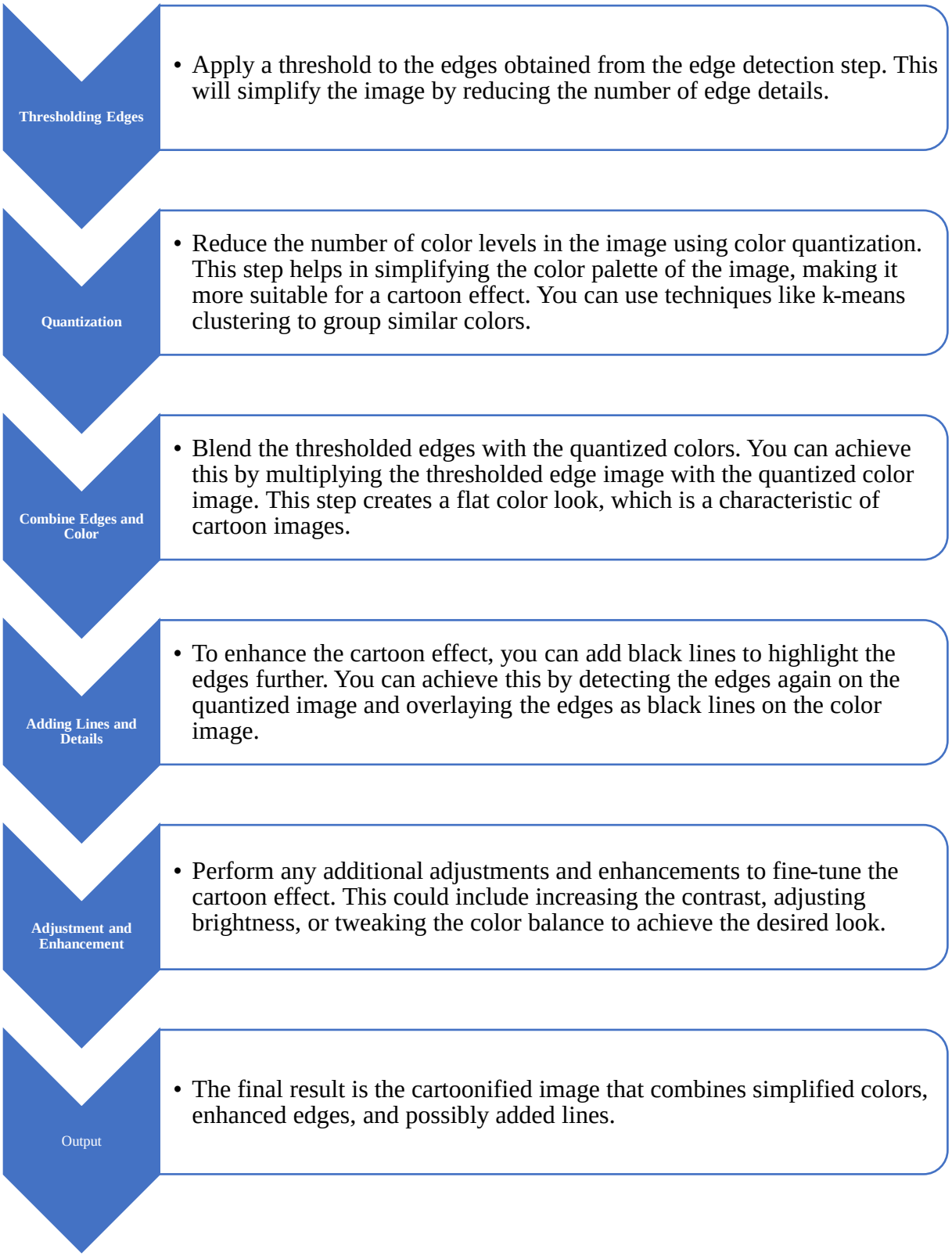
    b. **Bilateral Filtering Process:** During each iteration, the **cv2.bilateralFilter()** function applies bilateral filtering to the image. This filtering considers both the color and spatial distances between pixels. Pixels with similar colors and located close to each other are preserved, while noise is reduced.

4. **Upsampling:** After completing the bilateral filtering iterations, the image is upsampled back to its original size using the **cv2.pyrUp()** function. This step restores the image to its original dimensions while maintaining the smoothing effects achieved through bilateral filtering.
5. **Output Image:** The resulting image after bilateral filtering is the smoothed version of the input image. It will have reduced noise while preserving edges and important details.

Applications of Bilateral Filtering



Steps to be followed after Bilateral Filtering and Edge Detection



Steps to be followed when cartooning an image

Img 14 – Original image

Img 15 – Bilateral Filter

Img 16 – Grayscale image

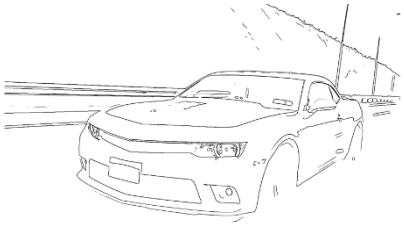
Img 17 – Edge image

Img 18 – Cartoon image

Img 19 – Original image

Img 20 – Bilateral Filter

Img 21 – Grayscale image



Img 22 – Edge image

Img 23 – Cartoon image

Python code for Cartooning an image

```
import cv2
import numpy as np

def cartoonize_image(input_image_path, output_size=(800, 600)):

    input_image = cv2.imread(input_image_path)


    num_down = 2
    num_bilateral = 7

    for _ in range(num_down):
        input_image = cv2.pyrDown(input_image)

    for _ in range(num_bilateral):
        input_image = cv2.bilateralFilter(input_image, d=9, sigmaColor=50,
sigmaSpace=50)

    for _ in range(num_down):
        input_image = cv2.pyrUp(input_image)


    gray_image = cv2.cvtColor(input_image, cv2.COLOR_BGR2GRAY)


    edges = cv2.adaptiveThreshold(gray_image, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, blockSize=9, C=2)


    kernel = np.ones((5, 5), np.uint8)
    edges = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel, iterations=1)


    cartoon_image = cv2.bitwise_and(input_image, input_image, mask=edges)


    cartoon_image_resized = cv2.resize(cartoon_image, output_size)

    return cartoon_image_resized

def display_combined_images(input_image, cartoon_image):

    input_image_resized = cv2.resize(input_image, (cartoon_image.shape[1],
cartoon_image.shape[0]))

    combined_image = np.hstack((input_image_resized, cartoon_image))
    cv2.imshow('Input vs Cartoonized', combined_image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

```
if __name__ == "__main__":
    input_image_path = 'man.jpg'
    output_size = (800, 600)
    output_image = cartoonize_image(input_image_path, output_size)

    if output_image is not None:
        input_image = cv2.imread(input_image_path)
        display_combined_images(input_image, output_image)
    else:
        print("Cartoonization failed.")
```

1. Import Libraries:

The code imports the necessary libraries: cv2 (OpenCV) for image processing and numpy for numerical operations.

2. Cartoonize Function (cartoonize_image):

This function takes an input image and an output size as parameters. The input image is loaded using cv2.imread(). It reduces the size of the image (num_down times) using Gaussian pyramid downsampling to speed up the processing.

3. Bilateral Filtering:

The code applies bilateral filtering (num_bilateral times) to the downsized image. Bilateral filtering preserves edges while smoothing other areas.

4. Image Upsampling:

The downsized image is then upsampled (num_down times) to restore the original dimensions.

5. Grayscale Conversion:

The upscaled image is converted to grayscale using cv2.cvtColor().

6. Edge Detection:

Adaptive thresholding is applied on the grayscale image using cv2.adaptiveThreshold(). This creates a binary image with black and white regions representing edges.

7. Morphological Operation:

A morphological closing operation (cv2.morphologyEx()) is performed to close gaps in the edges and enhance their connectivity.

8. Cartoon Image Generation:

The final cartoon image is created by performing a bitwise AND operation between the original image and the edge-detected image (cv2.bitwise_and()).

9. Image Resizing:

The resulting cartoon image is resized to the specified output size using cv2.resize().

10. Display Function (display_combined_images):

This function takes the input image and the cartoon image as parameters. It resizes the input image to match the dimensions of the cartoon image. The two images are horizontally stacked using np.hstack() to create a side-by-side comparison.

The combined image is displayed using cv2.imshow() and waits for a key press before closing using cv2.waitKey() and cv2.destroyAllWindows().

11. Main Execution:

If the script is run directly (not imported), it specifies the input image path, output size, and calls the cartoonize_image() function.

If cartoonization is successful, it loads the input image and displays both the original and cartoonized images using the display_combined_images() function.