Image Cartoonizer

Image and Video Processing Project Report

Group Members:-

Gursimrat Singh Kalra **(B420024)**
Kabir Bhagat **(B420061)**
Yash Gupta **(B420066)**

Introduction to the project

Image processing is a rapidly developing branch of computer science that deals with analysing, manipulating, and interpreting images. It has a wide variety of applications in various fields such as medical, engineering, entertainment, and security. Image cartoonization, a subfield of image processing, is the process of transforming digital images into cartoon-style images, usually with simplified features and exaggerated representations.
Caricatures have grown significantly in popularity in recent years due to their various practical uses in entertainment, marketing, and education. In marketing, it can be used to create unique and eye-catching styles of advertising and promotional materials. In education, it can be used to create cartoon-style images for teaching materials such as textbooks and online courses.

The proposed image cartoonizer aims to provide a solution to the challenges encountered in manually creating cartoon-style images, which are time-consuming and require considerable artistic skill. Instead, it offers a more efficient and automated approach to converting digital images into cartoon-style images. This project uses various image processing techniques such as edge detection, thresholding, smoothing, blurring and gradient adjustment to produce high quality cartoon style images. This report provides a detailed overview of the proposed image cartooning tool, its usefulness, the proposed workflow, and the frameworks and tools used in its implementation. Implementation results, results, and their discussion are presented, including the strengths and weaknesses of the proposed approach and suggestions for future improvements. In addition, the report outlines each member's contribution to the development of picture cartoonists.

Utility of the project

The proposed image cartoonizer has various practical applications in various fields such as entertainment, marketing and education. It provides an efficient and automated approach to transforming digital images into cartoon-style images with simplified features and exaggerated representations. This approach saves time and effort compared to manual cartoonization and can produce high-quality cartoon-style images with minimal noise and artifacts.

Here are some specific points outlining the Utility of the project:

- The image cartoonizer can be used to create funny and whimsical drawings for social media, memes, and other online content.
- It can be used to create unique and eye-catching styles of advertising and promotional materials.
- Image Cartooniser can be used to create cartoon-style images for teaching materials such as textbooks and online courses.
- It can be used to create cartoon-style images for animations and games.
- Image cartoonizers can be used in the medical industry to enhance images for diagnostic purposes.
- In the security industry, it can be used for facial recognition and identification purposes.
- The image cartoonizer can be used to create personalised cartoon style avatars for use in social media and gaming applications.
- It can be used to enhance the visual appeal of websites, blogs, and other digital media by adding cartoon-style images.

Proposed workflow in the project

To create a cartoon effect, we need to pay attention to two things:

1. The first difference is that the colours in the cartoon image are more homogeneous as compared to the normal image.
2. The second difference is noticeable within the edges that are much sharper and more pronounced in the cartoon.

Those are what make the differences between a photo and a cartoon. To adjust that two main components, there are four main steps that we will go through:

1. Detecting and emphasising edges
    - Convert the original colour image into grayscale
    - Using adaptive thresholding to detect and emphasise the edges in an edge mask.
    - Apply a median blur to reduce image noise.
2. Image filtering
    - Apply a bilateral filter to create homogeneous colours on the image.
3. Creating a cartoon effect
    - Use a bitwise operation to combine the processed colour image with the edge mask image
4. Creating a cartoon effect using colour quantization

1. Detecting and emphasising edges

Let's begin by importing the necessary libraries and loading the input image.

```
# Necessary imports
import cv2
import numpy as np
#  Importing function cv2_imshow necessary for programming in
Google Colab
from google.colab.patches import cv2_imshow
```

Now, we are going to load the image.

```
img = cv2.imread("Superman.jpeg")
cv2_imshow(img)
```

The next step is to detect the edges. For that task, we need to choose the most suitable method. Remember, our goal is to detect clear edges. There are several edge detectors that we can pick. Our first choice will be one of the most common detectors, and that is the Canny edge detector. But unfortunately, if we apply this detector we will not be able to achieve desirable results. We can proceed with Canny, and yet you can see that there are too many details captured. This can be changed if we play around with Canny's input parameters (numbers 100 and 200).

```
edges = cv2.Canny(img, 100, 200)
cv2_imshow(edges)
```

Although Canny is an excellent edge detector that we can use in many cases in our code we will use a threshold method that gives us more satisfying results. It uses a threshold pixel value to convert a grayscale image into a binary image. For instance, if a pixel value in the original image is above the threshold, it will be assigned to 255. Otherwise, it will be assigned to 0 as we can see in the following image.

However, a simple threshold may not be good if the image has different lighting conditions in different areas. In this case, we opt to use `cv2.adaptiveThreshold()` function which calculates the threshold for smaller regions of the image. In this way, we get different thresholds for different regions of the same image. That is the reason why this function is very suitable for our goal. It will emphasise black edges around objects in the image.

So, the first thing that we need to do is to convert the original colour image into a grayscale image. Also, before the threshold, we want to suppress the noise from the image to reduce the number of detected edges that are undesired. To accomplish this, we will apply the median filter which replaces each pixel value with the median value of all the pixels in a small pixel neighbourhood. The function `cv2.medianBlur()` requires only two arguments: the image on which we will apply the filter and the size of a filter.

The next step is to apply the `cv2.adaptiveThreshold()` function. As the parameters for this function we need to define:

- **max value** which will be set to 255
- **cv2.ADAPTIVE_THRESH_MEAN_C** : a threshold value is the mean of the neighbourhood area.
- **cv2.ADAPTIVE_THRESH_GAUSSIAN_C** : a threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.
- **Block Size** – It determines the size of the neighbourhood area.
- **C** – It is just a constant which is subtracted from the calculated mean (or the weighted mean).

For better illustration, let's compare the differences when we use a median filter, and when we do not apply one.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 5)
```

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray_1 = cv2.medianBlur(gray, 5)
edges = cv2.adaptiveThreshold(gray_1, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 5)
```

As you can see we will obtain much better results when we apply a median filter. Naturally, edge detection obviously is not perfect. One idea that we will not explore here and that you can try on your own is to apply morphological operations on these images. For instance, erosion can assist us here to eliminate small tiny lines that are not a part of a large edge.

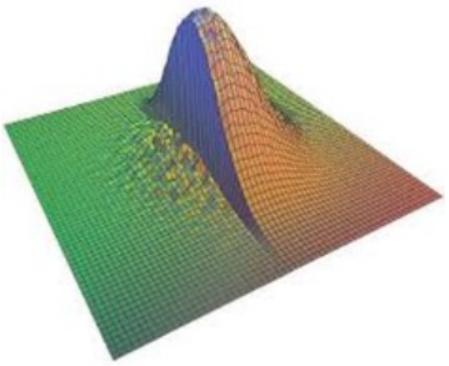
2. Image filtering

Now we need to choose a filter that is suitable for converting an RGB image into a colour painting or a cartoon. There are several filters that we can use. For example, if we choose to use `cv2.medianBlur()` filter we will obtain a solid result. We will manage to blur the colours of the image so that they appear more homogeneous. On the other hand, this filter will also blur the edges and this is something that we want to avoid.

The most suitable filter for our goal is a bilateral filter because it smooths flat regions of the image while keeping the edges sharp.

Bilateral filter

Bilateral filter is one of the most commonly used edge-preserving and noise-reducing filters. In the following image you can see an example of a bilateral filter in 3D when it is processing an edge area in the image.



Similarly to the Gaussian, bilateral filter replaces each pixel value with a weighted average of nearby pixel values. However, the difference between these two filters is that a bilateral filter takes into account the variation of pixel intensities in order to preserve edges. The idea is that two nearby pixels that occupy nearby spatial locations also must have some similarity in the intensity levels.

To better understand this let's have a look in the following equation:

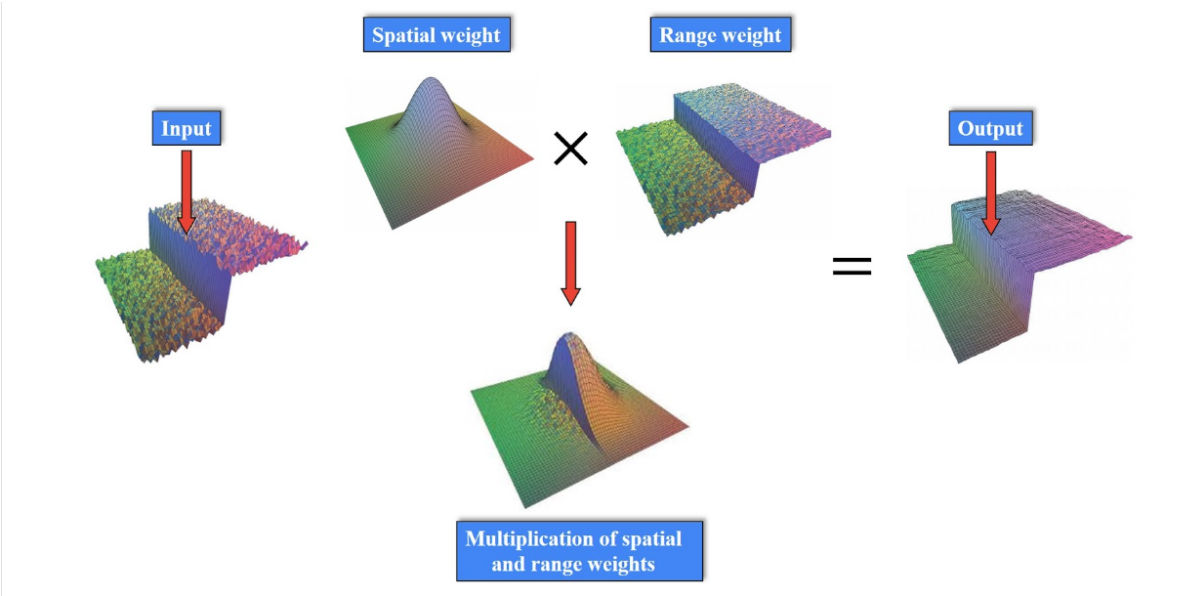
$$BF[I]_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) G_{\sigma_r}(I_{\mathbf{p}} - I_{\mathbf{q}}) I_{\mathbf{q}}$$

Where,

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in S} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|) G_{\sigma_r}(I_{\mathbf{p}} - I_{\mathbf{q}})$$

Here the term $1/Wp$ is a normalised weighted average of nearby pixels p and q.

Parameters σs and σr control the amount of filtering. $G\sigma s$ is a spatial Gaussian function that controls the influence of distant pixels, and $G\sigma r$ is a range Gaussian function that controls the influence of pixels with an intensity value different from the central pixel intensity Ip.
So, this function makes sure that only those pixels with similar intensities to the central pixel are considered for smoothing. Therefore, it will preserve the edges since pixels at edges will have large intensity variation.
Now, to visualise this equation let's have a look at the following image. On the left we have an input image represented in 3D. We can see that it has one sharp edge. Then, we have a spatial weight and a range weight function based on pixel intensity. Now, when we multiply range and spatial weights we will get a combination of these weights. In that way the output image will still preserve the sharp edges while flat areas will be smoothed.



There are three arguments in `cv2.bilateralFilter()` function:

- **d** – Diameter of each pixel neighbourhood that is used during filtering.
- **sigmaColor** – the standard deviation of the filter in the colour space. A larger value of the parameter means that farther colours within the pixel neighbourhood will be mixed together, resulting in larger areas of semi-equal colour.
- **sigmaSpace** –the standard deviation of the filter in the coordinate space. A larger value of the parameter means that farther pixels will influence each other as long as their colours are close enough.

```
colour = cv2.bilateralFilter(img,
d=9,sigmaColor=200,sigmaSpace=200)
cv2_imshow(colour)
```

3. Creating a cartoon effect

Our final step is to combine the previous two: We will use `cv2.bitwise_and()` the function to mix edges and the colour image into a single one.

```
cartoon = cv2.bitwise_and(colour, colour, mask=edges)
cv2_imshow(cartoon)
```

This is our final result, and you can see that indeed we do get something similar to a cartoon or a comic book image.

4. Creating a cartoon effect using colour quantization

Another interesting way to create a cartoon effect is by using the colour quantization method. This method will reduce the number of colours in the image and that will create a cartoon-like effect. We will perform colour quantization by using the K-means clustering algorithm for displaying output with a limited number of colours.

First, we need to define the color_quantization() function.

```
def color_quantization(img, k):
# Defining input data for clustering
  data = np.float32(img).reshape((-1, 3))
# Defining criteria
  criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
20, 1.0)
# Applying cv2.kmeans function
  ret, label, center = cv2.kmeans(data, k, None, criteria, 10,
cv2.KMEANS_RANDOM_CENTERS)
  center = np.uint8(center)
  result = center[label.flatten()]
  result = result.reshape(img.shape)
  return result
```

Different values for \(K \) will determine the number of colours in the output picture. So, for our goal, we will reduce the number of colours to 7. Let’s look at our results.

```
img_1 = color_quantization(img, 7)
cv2_imshow(img_1)
```

Now, let’s see what we will get if we apply the median filter on this image. It will create more homogeneous pastel-like colouring.

```
blurred = cv2.medianBlur(img_1, 3)
cv2_imshow(blurred)
```

And finally, let’s combine the image with detected edges and this blurred quantized image.

```
cartoon_1 = cv2.bitwise_and(blurred, blurred, mask=edges)
cv2_imshow(cartoon_1)
```

Framework and tools used

The proposed image cartoonizer uses various frameworks and tools to implement the image processing algorithms needed to transform digital images into cartoon-style images. The frameworks and tools used to implement the project are:

❖ PROGRAMMING LANGUAGE

- **Python** : Python is a high-level, interpreted programming language known for its simplicity, readability, and versatility. It is popularly used in web development, data analysis, artificial intelligence, and machine learning, among other applications. Python's syntax is easy to read and write, making it an ideal choice for beginners and experienced developers alike. It has a large and active community of developers, with a vast collection of open-source libraries and frameworks for various purposes, such as scientific computing, data visualisation, and image processing.

❖ Framework

- ➢ **OpenCv** : OpenCV (Open Source Computer Vision Library) is a free open source library of computer vision algorithms and functions used for image and video processing. It offers a wide range of image and video processing functions such as object detection, face detection, image filtering, segmentation, feature extraction and transformation. OpenCV is widely used in various applications such as robotics, augmented reality, and driver assistance systems. It supports multiple platforms such as Windows, Linux, macOS, Android, and iOS, making it a broad choice for cross-platform computer vision applications.

Implementation outcome - results and their discussion

Image cartoonizer has been successfully implemented with promising results using two different techniques:
Quantization and bilateral filtering.

Quantization techniques reduce the number of colours in an image. This is characteristic of cartoon-style images. A quantization technique was applied to the input images using the k-means clustering algorithm with k=7. The output image has a limited number of colours and hand-drawn effects. However, quantization techniques may not preserve important features of the input image, and some details may be lost in the process.

Bilateral filtering techniques, on the other hand, smooth the image while preserving edges. A bilateral filtering technique was applied to the input image to create a cartoon-like image. The output image has smoother edges and less detail characteristic of cartoon-style images. However, bilateral filtering techniques may not produce images with sufficient colour variation.

To demonstrate the results of each step in the workflow, we will provide output images for each part:

1. **Input image/Original Image**

2. **Edge detection**

3. IMAGE CARTOONIZATION

QUANTIZATION

IMAGE QUANTIZATION

IMAGE BLUR

COMBINE THE **EDGES** WITH **QUANTIZED+BLURRED** IMAGE

BILATERAL FILTER

APPLYING BILATERAL FILTER TO IMAGE

COMBINE THE **EDGES** WITH **THE BILATERAL FILTER** IMAGE

X = ORIGINAL IMAGE	Y = BILATERAL + BLURRED	Z = Y + EDGES
		
U = QUANTISED	**V = U + BLURRED**	**W = V + EDGES**
		

FINAL CONCLUSION

After comparing the results of the two techniques, both techniques were found to be effective in creating cartoon-like images. However, quantization techniques tend to produce images with a more limited range of colours, while bilateral filtering techniques tend to produce images with smoother edges.

Contribution Done By Individual Members

Kabir Bhagat (B420061)	Gursimrat Singh Kalra (B520024)	Yash Gupta (B420066)
Image Preprocessing Image Segmentation **Details** Image preprocessing and segmentation. Different ways to find edges of a given figure using Canny edge detector Adaptive Thresholding	Feature extraction Object recognition Visualisation **Details** Image cartoonization using Bilateral Filter - Bilateral Filter - Combine With Edges	Feature extraction Object recognition Visualisation **Details** Image cartoonization using Quantization - Image Quantization - Image Blur - Combine With Edges

Code/Github of the project

The code of the project can be found in the following Google collab link:

https://colab.research.google.com/drive/1adFgNgP5fmBFWBTfCeabOvMuPfo3Ss9H?usp=sharing

The Github repository related to this project can be found in the following link:

https://github.com/conqryash007/Image-Cartoonizer