Name: Patrick Quijano

Image Toolkit Documentation

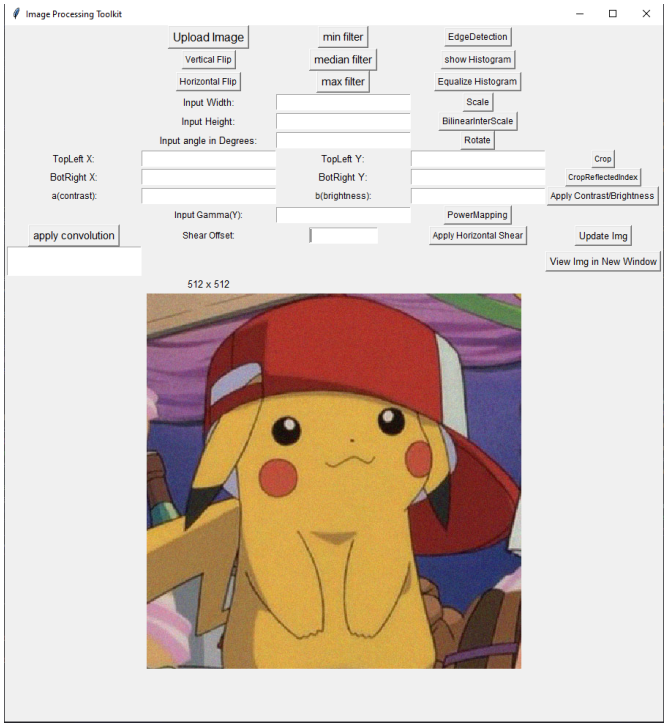
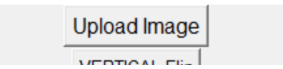


Table of Contents

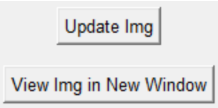
User-Manual

Toolbox capabilities(What it does/ how to use)

When first launching up the application in order to perform any of the listed operations the user will need to upload a image first



IMAGES will change and saved when performing operations - To see changes to image after performing any operations User has option to update current image in view or to open image in new window

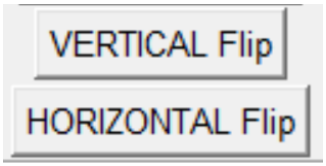


Crop (zero-padding/circular/reflected indexing)

Does/Suppose to do: To crop the current image the user can input the top left coordinates (x and y values) and the bottom right coordinates (x and y values) of the part of the image that they want to keep everything outside of these 2 coordinates will be cropped out and will either be replaced with zero-padding, reflected, or circular indexing based on the users preference(3 cropping buttons: *crop* or reflected crop or Circular crop). After cropping your uploaded image will be updated to be the cropped image.

Flip (Horizontal/Vertical)

Does/Suppose to do: To flip the current image the user can simply click the vertical flip or horizontal flip button options, this results in the images to be flipped horizontally or vertically.



Scale (nearest neighbor/ bilinear interpolation)

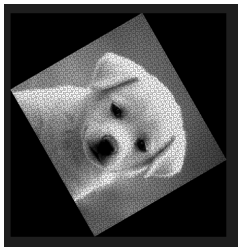
Does/Suppose to do: To scale the current image the user can input new desired Width and Height of the image. Clicking Scale will scale the image to the new specified dimensions using nearest neighbor scaling. Clicking the bilinearInterScale button will scale the image to the new specified dimensions using bilinear interpolation providing a more accurate image upon scaling.

Horizontal Shearing

Does/Suppose to do: To shear the current image the user can input a desired offset value. Upon clicking the "Apply Horizontal Shear" the image will be modified to have a shearing effect.

Rotate

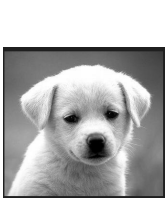
Does/Suppose to do: To rotate the current image the user can input the desired angle to rotate by providing the angle in degrees. As a result the Image will be rotated upon the middle of the image and depending if a rotation other than 90, 180, 270, 360 etc.. is given then the image dimensions will be modified in order to avoid any loss of data or parts of the image being cut off.



Does not do/Not supposed to do: when rotating at an angle that requires padding only zero-padding is used by default, circular indexing and reflected indexing are not provided as an option.

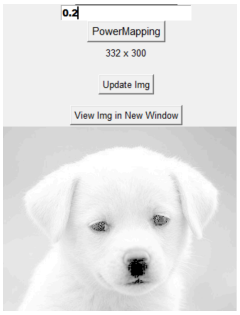
Linear Mappings

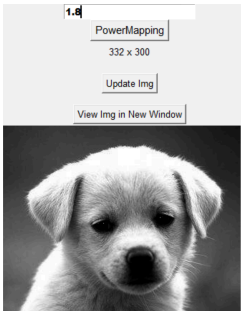
Does/Suppose to do: To apply different linear mappings to an image the user is given the option to input a choice of a(contrast) and b(brightness). The given a and b can be integer or decimal values negative or positive.



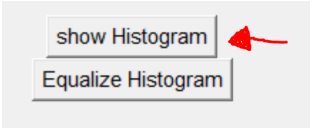
Power Law Mappings

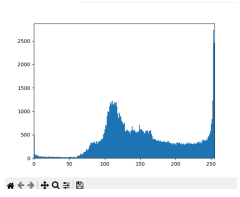
Does/Suppose to do: To apply power low mappings to an image the user is given the option to input a choice of Gamma(y). When y < 1 then the contrast in the darker areas of the image will Increase and when y > 1 the contrast in the darker areas will decrease.





Calculate Histogram

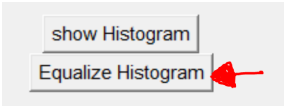




Does/Suppose to do: To view/calculate the current images histogram simply click the show Histogram button option, upon clicking a window popup view of the current histogram will be displayed.

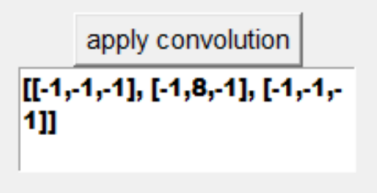
Histogram Equalization

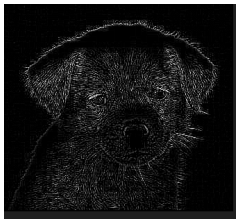
Does/Suppose to do: To equalize the the current histogram of the image the user can simply click the Equalize Histogram button, this will equalize the current image and the user will be prompted a warning popup indicating that any RGB images will be converted to grayscale since the histogram equalization will only be applied to grayscale images. To view the changes made by equalization, users can click the "view Img in New Window" or "Update Img" and also check the new histogram view "show Histogram" option.



Convolution (zero padding)

Does/Suppose to do: To apply convolution to an image the user can provide a convolution kernel as a list of lists, this will as a result apply the kernel to the current image. Ex kernels for edge detection:
(`[[-1,-1,-1], [-1,8,-1], [-1,-1,-1]]`)
or maybe
(`[[-2,-1,0], [-1,1,1], [0,1,2]]`) etc..

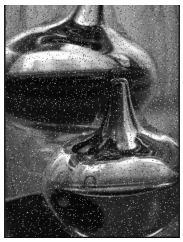




Does not do/Not supposed to do: When providing kernels any kernels that contain real numbers must be given as decimals (ex. 0.1), fractions will not be accepted (ex. 1/9).

Filtering (Min, Median, Max)

Does/Suppose to do: To apply minimum, median, or maximum filtering the user simply has to click 3 of the provided options of either "min filter", "median filter", "max filter", upon clicking the image will then be removed of any salt, pepper or salt and pepper.



Edge Detection

Does/Suppose to do: The toolbox offers a button in order to perform edge detection. Upon executing the operation the image will then be updated such that all the edges in the y and x in the image will be highlighted and indicated in white and everything else will be set to black.

Technical Discussion(how it does what it does)

Crop (zero-padding/reflected/circular indexing)

Zero-padding Crop: To achieve the zero padding effect I simply take in the coordinates of the top left and the bottom right of the image that the user wants to keep then looping through the image if the pixel is not within that range given by the user,convert the pixel value to black(0).

Reflected Index Crop: to achieve reflected indexing the code loops through the pixels of the image $\forall (x,y) \in Z2$, f R (x,y) = f(h(x mod 2M),k(y mod 2N)), therefore the pixel at x, h(x)=x if $x \in 0..M-1$ and h(x)=2M−1−x if $x \in M..2M-1$. And the pixel at y k(y)=y if $y \in 0..N-1$ and k(y)=2N−1−y if $y \in N..2N-1$.

Circular Index Crop: to achieve circular indexing the algorithm used loops through each pixel and calculates the new intensity level using this formula:

4.a) formal definition of f^c

1. $\forall (x,y) \in 0..M-1 \times 0..N-1, f^c(x,y) = f(x,y)$

2. $\forall (x,y) \in 0..M-1 \times 0..N-1, \forall (i,j) \in Z^2, f^c(x+iM, y+jN) = f(x,y)$

Flip (Horizontal/Vertical)

In order to flip images horizontally and vertically I simply looped through each pixel of an image and reversed the image's pixel values by replacing the current pixel value with the opposite end pixel value, As seen in Example 2.2 in order to reverse the pixel we get the opposite end pixel value by getting the opposite end Y-value (height-j-1) and vice versa for flipping vertically.

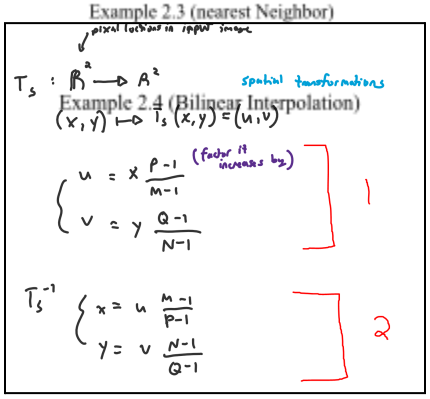
Example 2.2

```
temp1 = img_pixls[i,j]
img_pixls[i,j] = img_pixls[i,height-j-1]
img_pixls[i,height-j-1] = temp1
```

Scale (nearest neighbor/ bilinear interpolation)

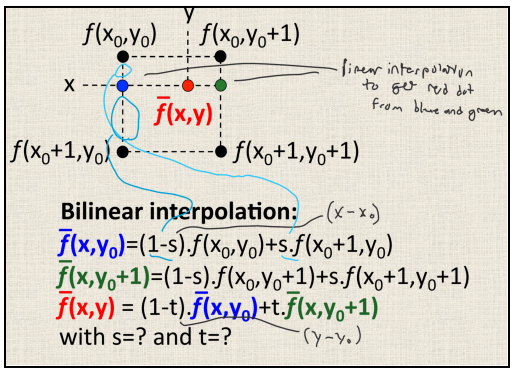
Nearest Neighbor: In order to achieve this I took in the current uploaded image and a new width and height specified by the user. Using this info a new empty image is created with the new given width and height and looping through the new image we can scale the image using the formula found in example 2.3 which takes the ratio between the original and new image width/height then multiplies that with each current x and y coordinate to get the new coordinate value.

Example 2.3 (nearest Neighbor)



Bilinear Interpolation: Bilinear interpolation creates new pixel values to maintain the image quality. In order to achieve this as when looping through the new image I translate the pixels in the original image to the scaled image by multiplying the current x and y by the ratio between the original and new image width and height I then get the surrounding 4 pixel coordinates of the current pixel, get their gray levels then get the estimated point using the 4 surrounding points using the formula from example 2.4.

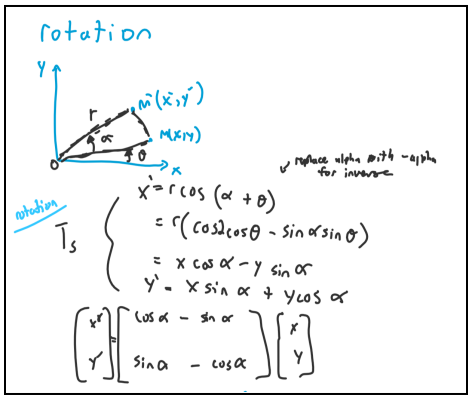
Example 2.4 (Bilinear Interpolation)



Rotate

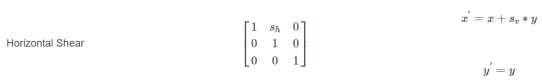
Example 2.5

```
newWidth = (img.width*abs(cos(newRotateBy)) + img.height*abs(sin(newRotateBy)))
newHeight = (img.height*abs(cos(newRotateBy)) + img.width*abs(sin(newRotateBy)))
```



Rotating an image is achieved by first calculating what the width and height of what the image would be after rotating; this will avoid image rotation with certain parts "eaten off" because after rotating some parts of the image could be out of the original image dimensions and so we need to account for that. Using the trig equations from example 2.5 we effectively calculate the max width and height of the new image. In our case we are rotating the image around the center point so for each x and y value we subtract from them the center of the image (x,y) - This makes it so when rotating the center is the (0,0) coordinate. We then can loop through this new image calculating the new pixel positions using newX = (x cos theta - y sin theta) + middleXlocation and new y = (x sin theta + y cos theta) + middleYlocation - when getting the new Image x and y coordinates we add back to each the *newImage* center coordinates so that the intensity values are being set correctly in the new image since we have already rotated our image around the center for the current pixel.

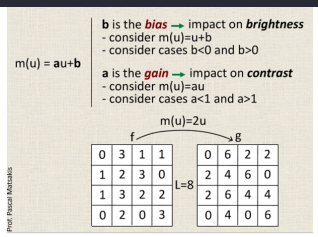
Shear



Horizontal shearing is achieved by looping through the current images pixels and updating the current intensity values with the intensity values from image pixel coordinates x' = x + offset * y and y' = y. Setting each pixel value to the values of these new coordinates will result in the horizontal shear effect.

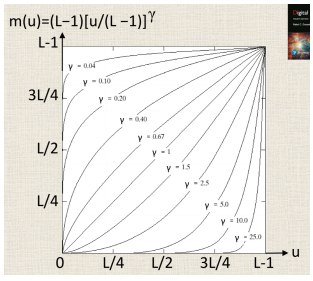
Linear Mappings

```
r, g, b = img.getpixel((x,y))
r = aVal * r + bVal
g = aVal * g + bVal
b = aVal * b + bVal
```



Linear mappings is achieved by simply using the equation M(u) = **a**u + **b** where a is the "gain" and makes changes to the image contrast and b is the "bias" which impacts the brightness and so the code takes in a given A and B value and loops through each pixel in the image applying that formula and setting the result back to the current pixel, this in turn gives us our resultant image.

Power Law Mappings



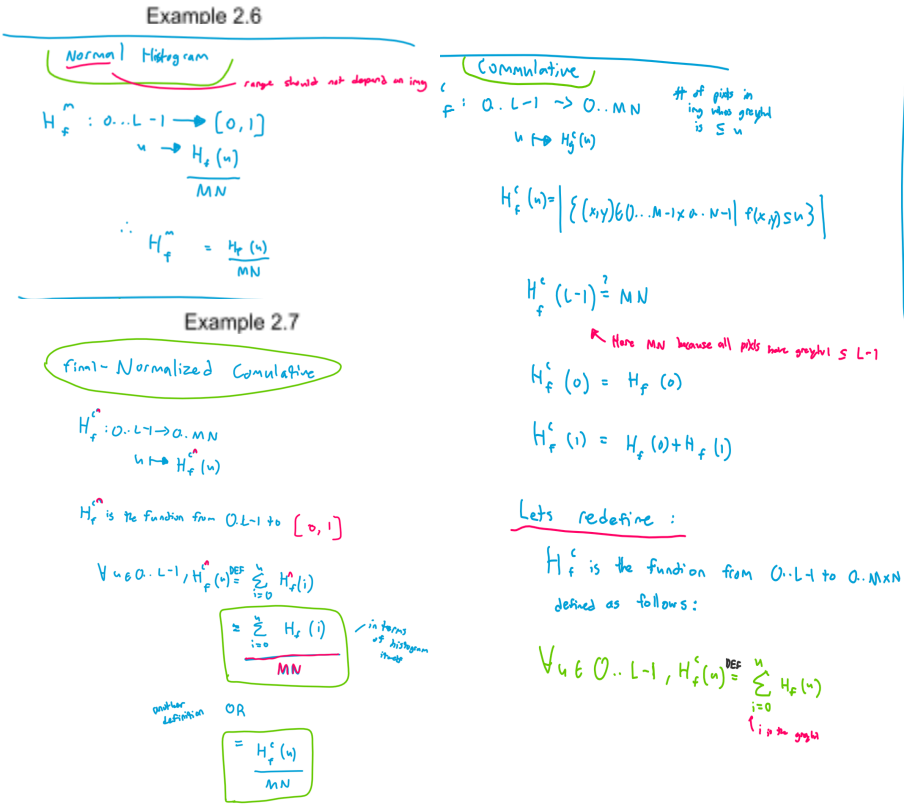
Power law mapping is achieved in my program by using the equation M(u) = (L-1)[u/(L-1)]^y, where L-1 is the max gray level in a our image and y is the gamma value, y = 1 would be no change in the image while y < 1 would increase the contrast in the darker regions of the image and y > 1 would decrease the contrast in the darker areas of the image, and so the code takes in a Gamma(y) value and loops through each pixel in the image calculating this

equation and setting the current pixel value to the result, this in turn gives us our resultant image.

Calculate Histogram

The histogram of an image is essentially just a graph representing the amount of pixels that share a certain gray level value and so in order to create the histogram, we first had to loop through the whole image saving the pixels and their gray levels in a list. With the list we can simply create a graph where the x axis represents the current gray level (0-255) and the y axis represents the amount of pixels for each given gray level.

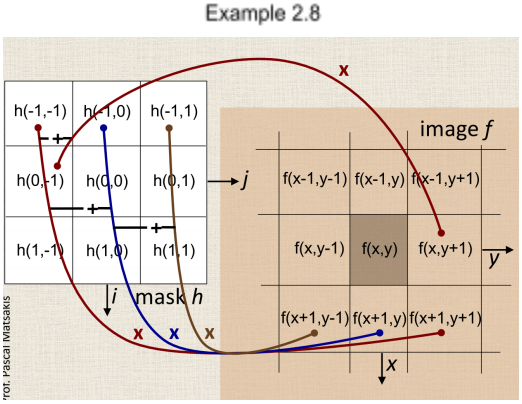
Histogram Equalization



In order to equalize the histogram of the image I first get an array/list of the histogram where the size of the array is 256 and each index keeps the # of occurrences of each intensity. We then want to normalize our histogram so that all the values are in between [0,1], in order to achieve this we use the equation as seen in example 2.6, here we divide each value in our histogram array by the total amount of pixels. After the histogram has been normalized we then get the normalized cumulative histogram. And so we then use the summation formula from example 2.7. By doing this each index in our histogram array will now be a summation of the previous two indices but still within the range [0,1]. The only thing left to do is equalize the histogram using the equation: `m(u) = Hgcn(v) * (L-1) = Hfcn(u) * (L-1)` this equation puts all our values back in the range [0,255] but should now be the equalized values.

Convolution (zero padding)

In order to apply convolution (zero padding), one should take in the kernel in the form of a 2d array/list. In order to apply convolution we first loop through each pixel in the image, at each pixel we then loop through the kernel. While looping through the kernel we calculate the new value of the current pixel in the image by using the equation: $g(x,y) = \sum_{i=-m-1/2}^{m-1/2} \sum_{j=-n-1/2}^{n-1/2} h(i,j) f(x-i, y-j)$



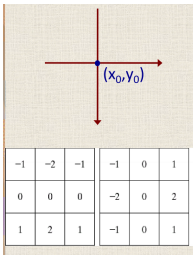
This sums up the multiplication between the current value in the kernel with the opposite pixel value in the actual image in the neighborhood(of the currently focused pixel) matching the kernel dimensions (refer to ex 2.8). After looping through the whole kernel, the same process is repeated for each pixel value in the image and the kernel is superimposed onto each pixel to calculate the new value of the pixel.

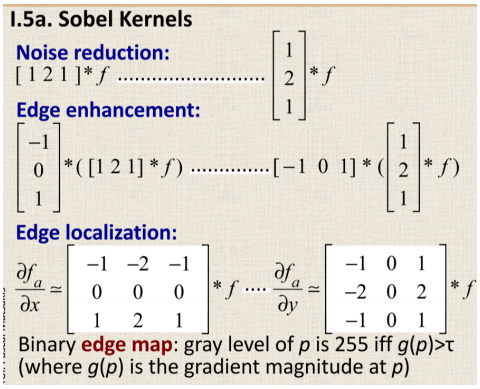
Filtering (Min, Median, Max)

To apply Min, Median and Max filtering we can apply the same general idea to each. We can take the chessboard neighborhood around each pixel in the image and put all the values into a sorted list. From this list we can take the minimum, maximum or median depending on what noise we want to get rid of, whether it's salt, pepper or salt and pepper.

Edge Detection

The edge detection operation within the toolbox makes use of the sobel kernels. In order to detect the edges we first define two kernels 1 kernel which will handle edges in the x and another kernel which will handle the edges in the y specifically: [[1, 2, 1],[0, 0, 0],[-1, -2, -1]] and [[1, 0, -1],[2, 0, -2],[1, 0, -1]]. We then convolve our image with respect to x and then with respect to y. I then get the magnitude of the gradient vector by taking the absolute values of the 2 convulsions together. Lastly we threshold to control how sensitive we want the edge detection to be, the local maximums compared to our chosen threshold determines whether we have edges or not.





Discussion of results and Future work

What doesn't function correctly

For the majority of the program all previously mentioned operations available in the toolbox should work and function as intended, there are only some weird cases that unfortunately don't function completely correctly.

Rotation: rotating an image will technically work correctly however when choosing to rotate an image other than 90°, 180°, 270°, 360° etc.. the image will be slightly modified with a strange pattern. This issue is called aliasing. Multiplying by *sines* and *cosines* on the integer coordinates of the uploaded image results in real numbers. Rounding the result sometimes causes certain pixels to be missed completely which causes the weird patterns. Also in order to maintain the bounding box around the image when rotating extra functionality was implemented, unfortunately this functionality causes a saving issue when other operations are applied and mixed in with rotating multiple times.

Convolution: the convolution works almost fully however rather than using zero padding the implemented code instead ignores the outer border pixels, And so for very small images the effect of not using zero padding may be more apparent.

Reflected Indexing: the reflected indexing for cropping works for certain cases and doesn't for others depending on what parts of the image is chosen to be cropped. This issue is a result of some sort of issue with the algorithm being used and not implemented correctly.

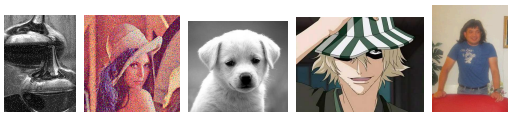
Testing

How did I test: For testing the different operations. I would first test a newly built option on its own with various images, (grayscale and RGB). I would then run the newly added operation in combination with the previously made operations, Testing all currently functioning operations and applying them all onto the same image varying orders. This is to find any unexpected behavior with the updating of the image. In order to ensure the operations were giving the correct results I compared my own methods with that of the built in methods (.rotate() , .scale() etc..) and online image editing tools to compare my results
(ex. https://generic-github-user.github.io/Image-Convolution-Playground/src/).

Ex Flows:

- upload Image, verticalFlip, crop, power mapping, minfilter, show Histogram, equalize, scale, rotate, apply convolution
- upload image rotate, scale, rotate, median filter, rotate

Test Data used: In order to test my programs various operations I made use of a variety of images specifically:



Program performances/strengths

My current image toolkit is very simple and easy to use out of the box, no learning curve is needed. All the options are provided on a single page so it's easy to quickly find any desired operations/options. The program doesn't contain any major bugs that won't hinder the user experience. The toolkit provides most of the major operations that are widely used.

What would I remedy for the future

My current image toolkit contains most of the common operations that are required in terms of image processing however it lacks the variety and additional interesting operations that could be used. Another issue is that the UI, although simple and easy to read, is not the best looking / appealing user interface.

What I would add in the future

First of all in the future I would like to redesign the User interface and perhaps create a dedicated website making the application easily accessible for anyone to use. Some other functionalities/ operations that I would like to add include:

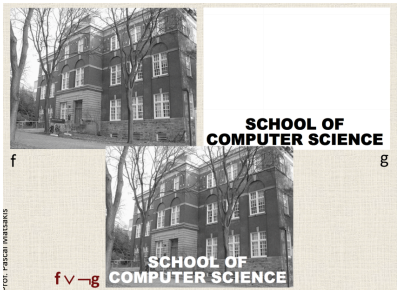


Image segmentation: allowing the user the option to divide their current image into different partitions/ parts and regions depending on different criteria such as color, intensity, texture etc..
Object recognition: allowing the user the option to identify different objects that are present in the image, by analyzing different criteria like looking at the edges or colors etc..
Image morphing: allowing the user to slowly transform their current image into another completely different image.
Image Blending: allowing the user to combine different images into a new composite image made from 2 or more different images as seen in the given image.