In the realm of photography, the quest for perfection often involves not just capturing the moment but transforming it into an immaculate visual narrative. The artistry behind the lens is continually evolving, with technology and innovation pushing the boundaries of creativity. This report delves into three key facets of modern photography enhancement: object removal, panorama creation, and analog film effects. Each aspect focuses on augmenting the photographer's toolkit, offering novel techniques and automated solutions to elevate the art of image composition and manipulation.

Our exploration of Panorama Creation delves into the intricacies of amalgamating multiple images seamlessly, resulting in breathtaking panoramic photographs. Our endeavour aims to simplify the arduous task of merging images, granting photographers the ability to craft expansive visual stories effortlessly.

Problem Description:
The code creates a panoramic image by stitching multiple input images using OpenCV for image processing tasks like loading, stitching, contour detection, and cropping.

Motivation:
Creating panoramic images is a typical problem in photography and computer vision. The goal is to generate a seamless panoramic view by automating the process of stitching several photos together. The stitching procedure is made simpler by utilizing OpenCV's image processing features.

Explanation:
The code starts by collecting image paths and loading them using OpenCV. It then stitches them together using OpenCV's Stitcher_create() function. After successful stitching, it adds a border, converts to grayscale, finds contours, and processes them to identify the region of interest (ROI) and crop the stitched image.

```python
import numpy as np
import cv2
import glob
import imutils
import tkinter as tk
from tkinter import filedialog

# Function to stitch images together
def stitch_images():
    # Get paths of all .jpg images in the 'unstitchedImages' folder
    image_paths = glob.glob('unstitchedImages/*.jpg')
    images = []

    # Load images and add them to the 'images' list
    for image in image_paths:
        img = cv2.imread(image)
        if img is not None: images.append(img)
        else:
            print(f"Error: Unable to load image {image}")

    # Check if there are at least 2 images for stitching
    if len(images) < 2:
        print("Insufficient images for stitching")
    else:
        # Create a Stitcher object and stitch the images
        imageStitcher = cv2.Stitcher_create()
        status, stitched_img = imageStitcher.stitch(images)

        # Check if stitching was successful
        if status == cv2.Stitcher_OK:
            # Save and display the stitched image
            cv2.imwrite("stitchedOutput.png", stitched_img)
            cv2.imshow("Stitched Image", stitched_img)
            cv2.waitKey(0)

            # Add a border to the stitched image
            stitched_img = cv2.copyMakeBorder(stitched_img, 10, 10, 10, 10, cv2.BORDER_CONSTANT, value=(0, 0, 0))

            # Convert the stitched image to grayscale and create a thresholded image
            gray = cv2.cvtColor(stitched_img, cv2.COLOR_BGR2GRAY)
```

```
        # Convert the stitched image to grayscale and create a thresholded image
        gray = cv2.cvtColor(stitched_img, cv2.COLOR_BGR2GRAY)
        _, thresh_img = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY)

        # Find contours in the thresholded image
        contours = cv2.findContours(thresh_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        contours = imutils.grab_contours(contours)

        # Check if contours are found
        if len(contours) > 0:
            # Get the largest contour and create a mask
            areaOI = max(contours, key=cv2.contourArea)
            mask = np.zeros(thresh_img.shape, dtype="uint8")
            x, y, w, h = cv2.boundingRect(areaOI)
            cv2.rectangle(mask, (x, y), (x + w, y + h), 255, -1)

            # Find the minimum rectangle enclosing the mask
            minRectangle = mask.copy()
            sub = mask.copy()

            # Process the minimum rectangle to remove unwanted areas
            while cv2.countNonZero(sub) > 0:
                minRectangle = cv2.erode(minRectangle, None)
                sub = cv2.subtract(minRectangle, thresh_img)

            # Find contours in the processed minimum rectangle
            contours = cv2.findContours(minRectangle, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
            contours = imutils.grab_contours(contours)

            # Check if contours are found after processing
            if len(contours) > 0:
                # Get the largest contour and crop the stitched image
                areaOI = max(contours, key=cv2.contourArea)
                x, y, w, h = cv2.boundingRect(areaOI)
                stitched_img = stitched_img[y:y + h, x:x + w]

                # Save and display the processed stitched image
                cv2.imwrite("stitchedOutputProcessed.png", stitched_img)
                cv2.imshow("Stitched Image Processed", stitched_img)
                cv2.waitKey(0)
```

```
                    # Save and display the processed stitched image
                    cv2.imwrite("stitchedOutputProcessed.png", stitched_img)
                    cv2.imshow("Stitched Image Processed", stitched_img)
                    cv2.waitKey(0)
                else:
                    print("Unable to find contours in the masked image")
            else:
                print("No contours found in the thresholded image")
        else:
            print("Stitching was unsuccessful")

# Create a Tkinter window
root = tk.Tk()
root.title("Image Stitching")

# Function to call when the button is clicked
def on_button_click():
    stitch_images()

# Create a button in the Tkinter window to trigger image stitching
button = tk.Button(root, text="Stitch Images", command=on_button_click)
button.pack()

root.mainloop()
```

Result:

Console Output: Error messages are displayed in the console if images fail to load or if the stitching process encounters issues.
Displaying Images: The stitched images (both original and processed) are displayed using OpenCV's cv2.imshow() function.
Saving Results: The code saves the stitched images in PNG format using cv2.imwrite().

Summary:
The code demonstrates an automated process for stitching images to create panoramas, including contour detection, cropping, and visualisation, with error handling and a user-friendly Tkinter interface.overall this code offers a functional image stitching implementation for creating panoramic views from multiple input images.

Secondly, to meet the resurgence of interest in retro aesthetics, we delve into analog film effects. Our implementation of these effects allows photographers to infuse their images with the nostalgic charm of analog film, enabling them to imbue character and uniqueness into their compositions.

Problem Description:
The application uses Python, OpenCV, and Tkinter to create a film effect on an image, incorporating techniques like grayscale conversion, noise addition, blur, and contrast enhancement.

Motivation:
The goal is to provide a simple interface that allows users to pick an image file and instantly view the effect, making it an easy-to-use tool for adding a film-like effect to photographs.

Explanation:
User Image Selection: Utilizes filedialog to prompt users to select an image file.The apply_film_effect image processing function grayscales the picture and applies a Gaussian blur to the picture in grayscale to creates erratic pixel intensities to add graininess and makes use of histogram equalization to improve contrast and Picture Displaying: Tkinter is used to convert the processed picture from OpenCV format to PIL format for display.

```python
import cv2
import numpy as np
import tkinter as tk
from tkinter import filedialog
from PIL import Image, ImageTk

def apply_film_effect(image):
    # Convert the image to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Apply Gaussian blur to the grayscale image
    blurred = cv2.GaussianBlur(gray, (15, 15), 0)

    # Add graininess to the image by generating random pixel intensity values
    grain = np.zeros_like(gray, dtype=np.uint8)
    cv2.randu(grain, 0, 255)
    noisy_image = cv2.addWeighted(blurred, 0.5, grain, 0.5, 0)

    # Increase contrast using histogram equalization
    equalized = cv2.equalizeHist(noisy_image)

    return equalized

def apply_effect_on_click():
    image_path = filedialog.askopenfilename()  # Ask user to select an image
    if image_path:
        image = cv2.imread(image_path)
        if image is None:
            print("Error: Unable to load the image.")
        else:
            filtered_image = apply_film_effect(image)
            # Convert the OpenCV image to PIL format for displaying in tkinter
            filtered_image_rgb = cv2.cvtColor(filtered_image, cv2.COLOR_BGR2RGB)
            img = Image.fromarray(filtered_image_rgb)
            img_tk = ImageTk.PhotoImage(image=img)
            label.configure(image=img_tk)
            label.image = img_tk  # Keep a reference to prevent garbage collection
```

```python
        image = cv2.imread(image_path)
        if image is None:
            print("Error: Unable to load the image.")
        else:
            filtered_image = apply_film_effect(image)
            # Convert the OpenCV image to PIL format for displaying in tkinter
            filtered_image_rgb = cv2.cvtColor(filtered_image, cv2.COLOR_BGR2RGB)
            img = Image.fromarray(filtered_image_rgb)
            img_tk = ImageTk.PhotoImage(image=img)
            label.configure(image=img_tk)
            label.image = img_tk  # Keep a reference to prevent garbage collection

# Create the main window
root = tk.Tk()
root.title("Film Effect App")

# Create a button to apply the effect
apply_button = tk.Button(root, text="Apply Film Effect", command=apply_effect_on_click)
apply_button.pack()

# Create a label to display the image
label = tk.Label(root)
label.pack()

# Run the application
root.mainloop()
```

Result:

The GUI application features an "Apply Film Effect" button, allowing users to select an image file and apply the film effect, which is displayed in the Tkinter window.

Summary:
The application provides a simple way for users to apply a film-like effect to their images.
It utilizes OpenCV for image processing and Tkinter for the graphical interface.
While the current implementation includes basic functionality, further improvements could be made, such as adding more customization options for the film effect, providing previews, or incorporating error handling for invalid image files or processing failures.

REMOVAL OF OBJECTS FROM AN IMAGE USING PYTHON.

The is one of the first part of the project we started with, using GUI and some other libraries. This code is meant to highlight a part of the image or an object in the image to be removed or replaced or cleared leaving a clear background. It was going well at first until we got into some issues with the code and the output.

Deleting a part of an image refers to the process of destroying the image data in a specific part of the image. The download can be dynamic or predefined. In most download processes, the image size will be the same as the resulting image. In fact, if you take a part of a shape, the size will change in some unknown way.

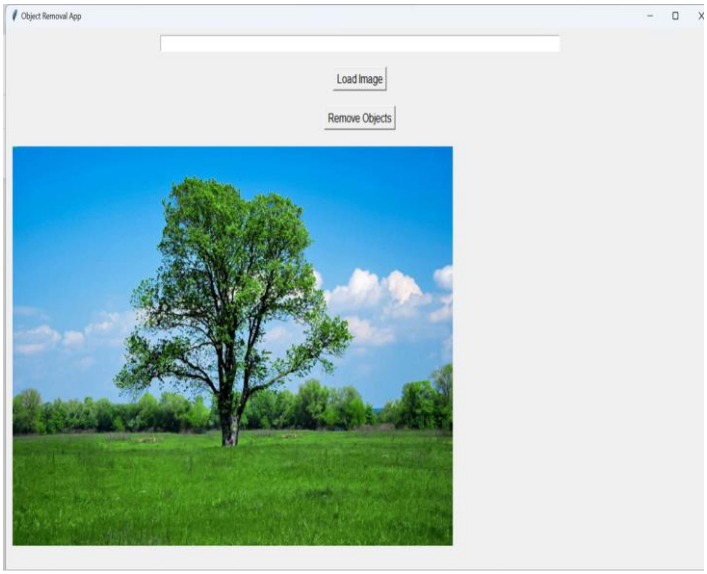
WHY OBJECT REMOVAL?

Sometimes images contain artifacts (variables) or white space. Once you know these sites, you need to identify their type, which will help you find the best way to get rid of them. Most image manipulation packages, such as Photo shop, Gimp, etc., provide tools to perform specific tasks. However, it can also be done programmatically. To extract a specific part (region) of the S-shape, the corresponding region must first be provided. Providing an ROI every time you go through the process means that ROI is difficult to measure. The ROI calculation itself (changing with different images and methods) means that ROI is dynamic. The ROI is the tuple of size 4 in the upper left and lower right of the box. To download a domain the first step is to specify the domain you want to download. The selection can be a region or a pixel value. To find an area, replace the pixel values in that area with the pixel values behind it. The background color is not uniform and will vary depending on the context in which the image is used. Most backgrounds are black and white.

WHAT WE DID

So we started by implementing buttons to load in the image and a button to use to remove objects. Cv2, numpy, tensorflow, tkinter, and PIL were the libraries used in this code. Now the issue we ran into first was accommodating the dimensions of the picture in the window of the program when it loads the image. We didn't want a situation where if you uploaded a different image, the program window will collapse or just show half of it because it is bigger than the dimensions required. We were able to fix that changing the windows size and adjusting the space for the picture or image. Then ran into the second problem, the removing object part. The buttons worked for the blank screen before we add the image because we had to change the cursor to show we had started the drawing mode or the removing

object phase; but unfortunately it didn't work for the loaded image this is the screenshot of the code and the result.




```python
import cv2
import numpy as np
import tensorflow as tf
import tkinter as tk
from tkinter import filedialog, ttk
from PIL import Image, ImageTk

# Load the pre-trained DenseNet201 model
model = tf.keras.applications.DenseNet201(input_shape=(None, None, 3), include_top=False)
model.trainable = False

# Variables to store drawing information
drawing_mode = False
img_original = None
img_modified = None
drawn_boxes = []

# Function to remove objects from an image with bounding boxes
def remove_objects(image_path, output_path):
    global img_original, img_modified

    # Load the image
    img = cv2.imread(image_path)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Draw bounding boxes on the original image
    for box in drawn_boxes:
        x, y, w, h = box
        cv2.rectangle(img_rgb, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

```
        # Fill the ROI with white to remove the object
        img_rgb[y:y+h, x:x+w] = [255, 255, 255]

    # Save the modified image
    cv2.imwrite(output_path, cv2.cvtColor(img_rgb, cv2.COLOR_RGB2BGR))

    # Update the modified image
    img_modified = Image.fromarray(cv2.cvtColor(img_rgb, cv2.COLOR_BGR2RGB))
    img_modified.thumbnail((800, 600))
    img_modified = ImageTk.PhotoImage(image=img_modified)

    return img_rgb

# Function to handle mouse events for drawing bounding boxes
def on_canvas_click(event):
    global drawing_mode
    if drawing_mode:
        x, y = canvas_original.canvasx(event.x), canvas_original.canvasy(event.y)
        canvas_original.create_rectangle(x, y, x, y, outline="red", width=2, tags="current_box")

# Function to handle mouse motion for drawing bounding boxes
def on_canvas_drag(event):
    global drawing_mode
    if drawing_mode:
        x, y = canvas_original.canvasx(event.x), canvas_original.canvasy(event.y)
        canvas_original.coords("current_box", x, y, x, y)

# Function to finalize the current bounding box
```

```
        # Fill the ROI with white to remove the object
        img_rgb[y:y+h, x:x+w] = [255, 255, 255]

    # Save the modified image
    cv2.imwrite(output_path, cv2.cvtColor(img_rgb, cv2.COLOR_RGB2BGR))

    # Update the modified image
    img_modified = Image.fromarray(cv2.cvtColor(img_rgb, cv2.COLOR_BGR2RGB))
    img_modified.thumbnail((800, 600))
    img_modified = ImageTk.PhotoImage(image=img_modified)

    return img_rgb

# Function to handle mouse events for drawing bounding boxes
def on_canvas_click(event):
    global drawing_mode
    if drawing_mode:
        x, y = canvas_original.canvasx(event.x), canvas_original.canvasy(event.y)
        canvas_original.create_rectangle(x, y, x, y, outline="red", width=2, tags="current_box")

# Function to handle mouse motion for drawing bounding boxes
def on_canvas_drag(event):
    global drawing_mode
    if drawing_mode:
        x, y = canvas_original.canvasx(event.x), canvas_original.canvasy(event.y)
        canvas_original.coords("current_box", x, y, x, y)

# Function to finalize the current bounding box
def finalize_current_box(event):
    global drawing_mode, drawn_boxes
```

```
import cv2
import numpy as np
import tensorflow as tf
import tkinter as tk
from tkinter import filedialog, ttk
from PIL import Image, ImageTk

# Load the pre-trained DenseNet201 model
model = tf.keras.applications.DenseNet201(input_shape=(None, None, 3), include_top=False)
model.trainable = False

# Variables to store drawing information
drawing_mode = False
img_original = None
img_modified = None
drawn_boxes = []

# Function to remove objects from an image with bounding boxes
def remove_objects(image_path, output_path):
    global img_original, img_modified

    # Load the image
    img = cv2.imread(image_path)
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # Draw bounding boxes on the original image
    for box in drawn_boxes:
        x, y, w, h = box
        cv2.rectangle(img_rgb, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

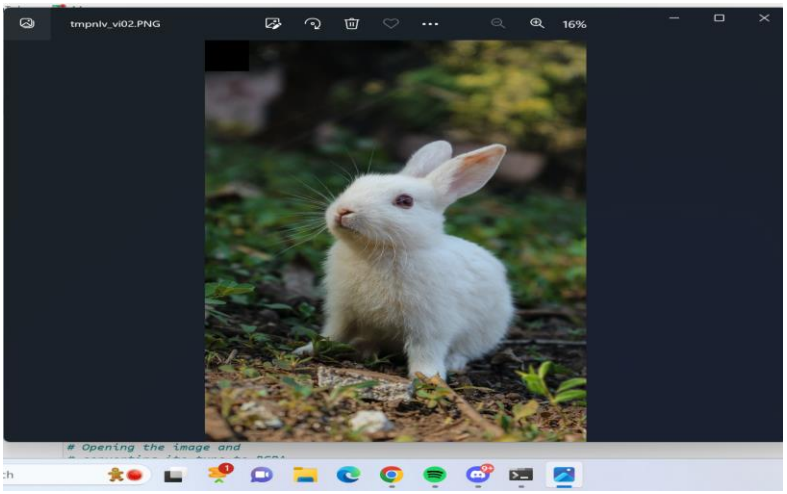
Then we tried it another way using pil and numpy libraries. This really didn't change much, it more or less just duled the image. and it was a shorter code.

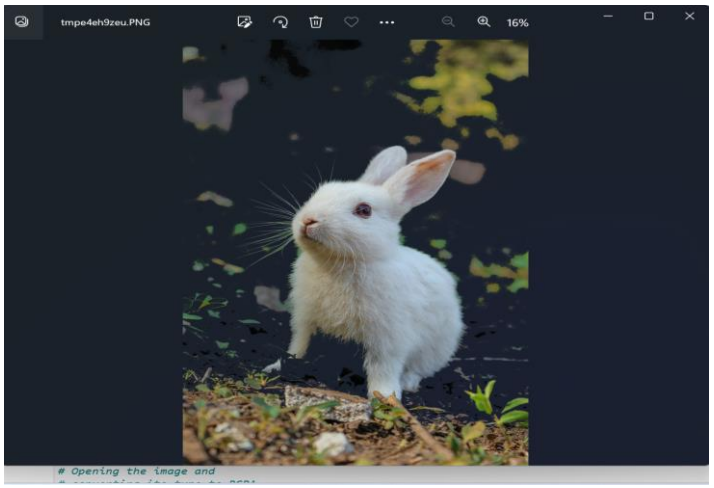
```
from PIL import Image
import numpy as np

# Opening the image and converting
# it to RGB color mode
# IMAGE_PATH => Path to the image
img = Image.open(r"rabbit.jpg").convert('RGB')

# Extracting the image data &
# creating an numpy array out of it
img_arr = np.array(img)

# Turning the pixel values of the 400x400 pixels to black
img_arr[0 : 400, 0 : 400] = (0, 0, 0)

# Creating an image out of the previously modified array
img = Image.fromarray(img_arr)

# Displaying the image
img.show()
```

This was the code we used but we also ran another code similar to this but isued image draw. It gave us a darker background output.
This is the second code.

```
# Importing ImageDraw for
# using floodfill function
from PIL import Image, ImageDraw

# Opening the image and
# converting its type to RGBA
img = Image.open(r"rabbit.jpg").convert('RGBA')

# Location of seed
seed = (0, 0)

# Pixel Value which would
# be used for replacement
rep_value = (0, 0, 0, 0)

# Calling the floodfill() function and
# passing it image, seed, value and
# thresh as arguments
ImageDraw.floodfill(img, seed, rep_value, thresh = 100)

img.show()
```

below are the two outputs for the code repectfully.





Conclusion:
The project highlights the evolving landscape of photography, highlighting the potential of Python libraries for creative image manipulation. Further exploration and refinement can lead to robust tools and automated solutions.