Image Processing project

Introduction

The objective of this project is to develop a classical computer vision pipeline that can analyze jigsaw puzzle pieces and prepare them for automatic matching and assembly. In Milestone 1, we focus on preprocessing the input images and enhancing their edges so that individual pieces and their boundaries are clearly represented. Using only traditional image processing techniques (without any machine or deep learning), we design and evaluate a sequence of operations that convert the raw puzzle tiles into robust edge maps. These outputs will be used in Milestone 2 to match different pieces together using those edges.

Pre-Processing

Description:

The goal of the preprocessing phase is to extract meaningful edges from the puzzle tile images so they can be reliably used in the next stage for piece matching and assembly. In particular, we aim to obtain accurate but not overly detailed edge maps that emphasize the main outlines of objects and puzzle boundaries, while suppressing inner textures and small noisy details that would complicate matching. The preprocessing pipeline is therefore designed to enhance local contrast, reduce noise in flat regions,

and produce clean, stable edges that capture the essential structure of each tile in a way that is robust across different images and lighting conditions.

Steps:

1)We convert the image into a grayscale image so to prepare it for the edge detection filter.

2)Edge density detection, here we use fixed parameter canny filter for the original image then we count the percentage of white pixels in the result, this gives us a good indicator of how busy the image is this later on helps us base our parameters on the image itself. If the image is too busy we want to blur more and only detect the very strong edges but if it's a simple plain image then we don't want the blurring to be too high and the threshold for the edge detection can be a bit lower.

2)We apply histogram equalization using Clahe (Contrast Limited Adaptive Histogram Equalization) . It differs from normal histogram equalization in the sense that instead of applying to the whole image at once it divides the image into segments and applies histogram equalization on each segment alone so edges and details inside each segment are clearer it performs better than normal adaptive histogram equalization when it comes to flat regions in the image as normal histogram equalization leads to a lot of noise when it comes to these areas, then the enhanced segments are blended smoothly so we can't tell that they were separated.

3)We apply a bilateral filter, to blur the image yet still preserve edges. This is a very important step in our pipeline the bilateral filter works by taking to things into consideration when looking at the nearby pixels in the kernel, intensity and how close they are. If a pixel's intensity is a lot higher than the current pixel's intensity then that likely means it's an edge so the bilateral filter gives that pixel a smaller weight which helps in preserving edges, same happens with closer pixels they are given a higher weight than pixels that are further away from the current pixel in the kernel.

4)We then use an optimized version of canny edge detector to detect the edges in the image called auto canny. The way auto canny works is that for each image we calculate the median from all the pixel values, we choose median over mean so noise pixels aren't considered. The median gives us an indicator at whether the image is dark, bright, or has an average intensity. This way we can determine the values for the lower and upper threshold of the canny edge detector.

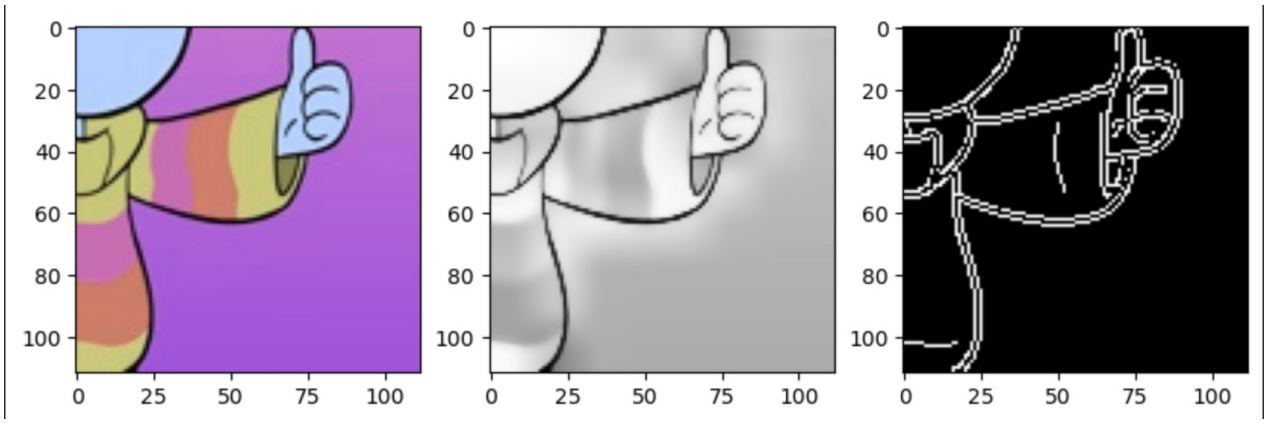
Image samples:

This pipeline performance varies in the images that we use these are samples of some of the images, the issues that arise in them and some of our solutions to these problems.

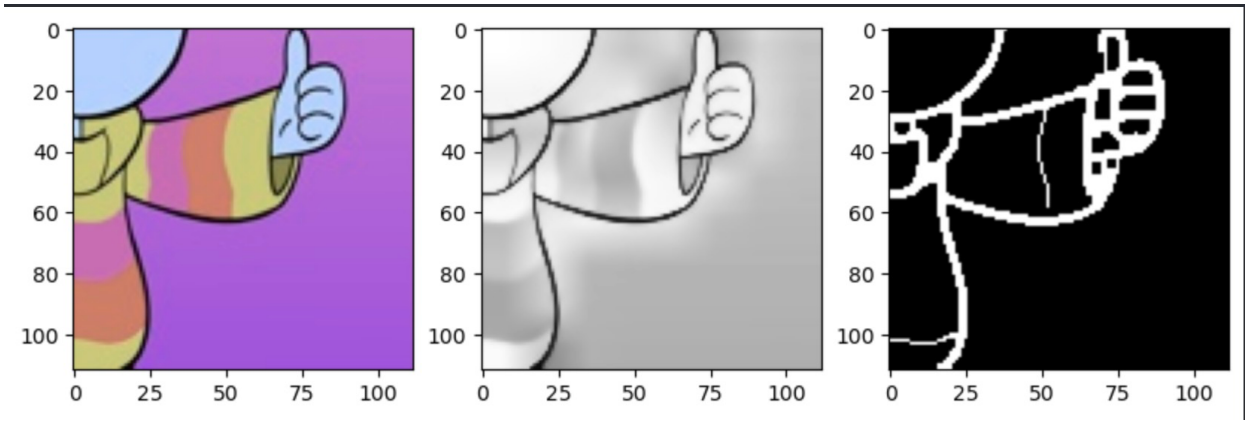
1)Some of the

1) In this image the pipeline successfully extracts the edges from the image , but it also has a problem that double edges appear because of the black stroke in the original image.

We had two solutions to this:

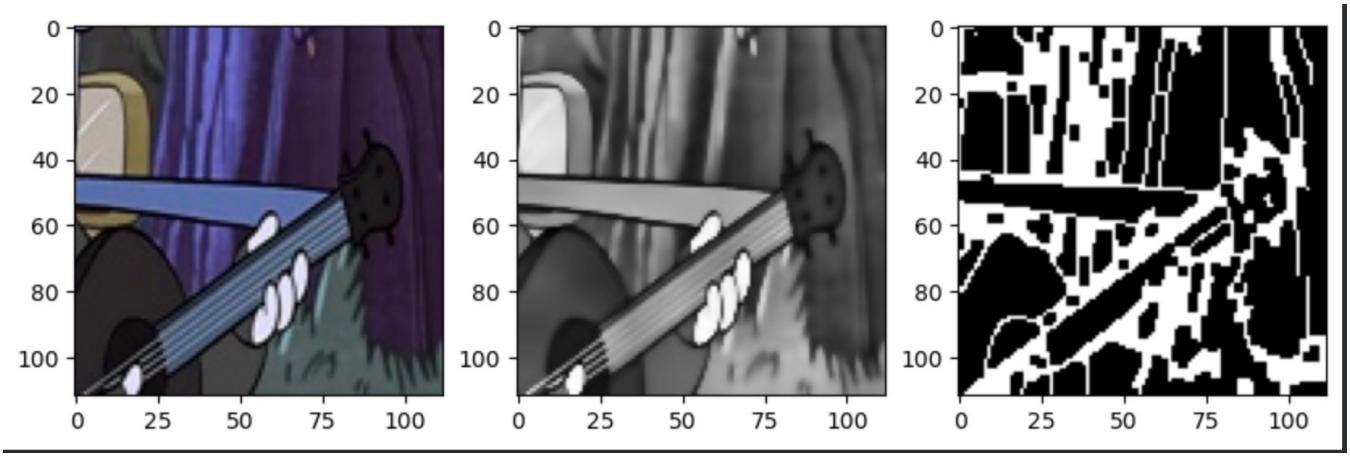


1.1) Apply morphological closing to this result to merge the two edges together.

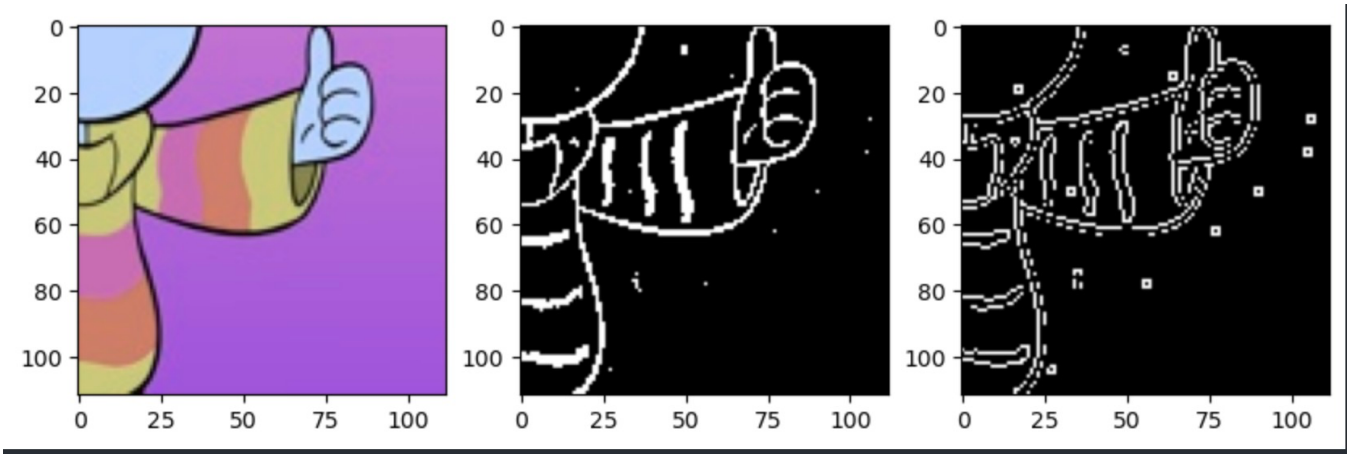


3

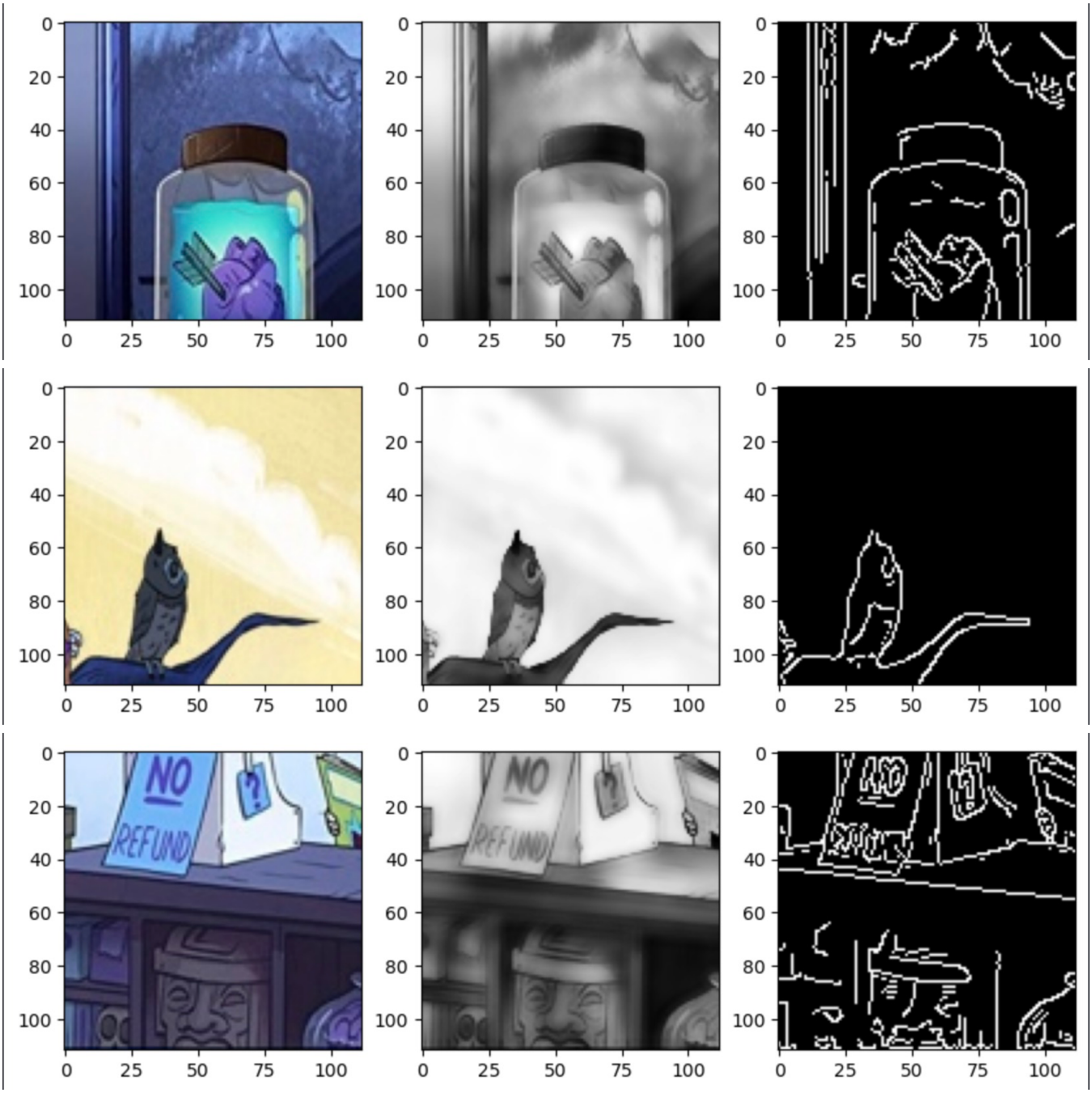
This merges the two edges that were causing the problem but for other images it messes up the edges.

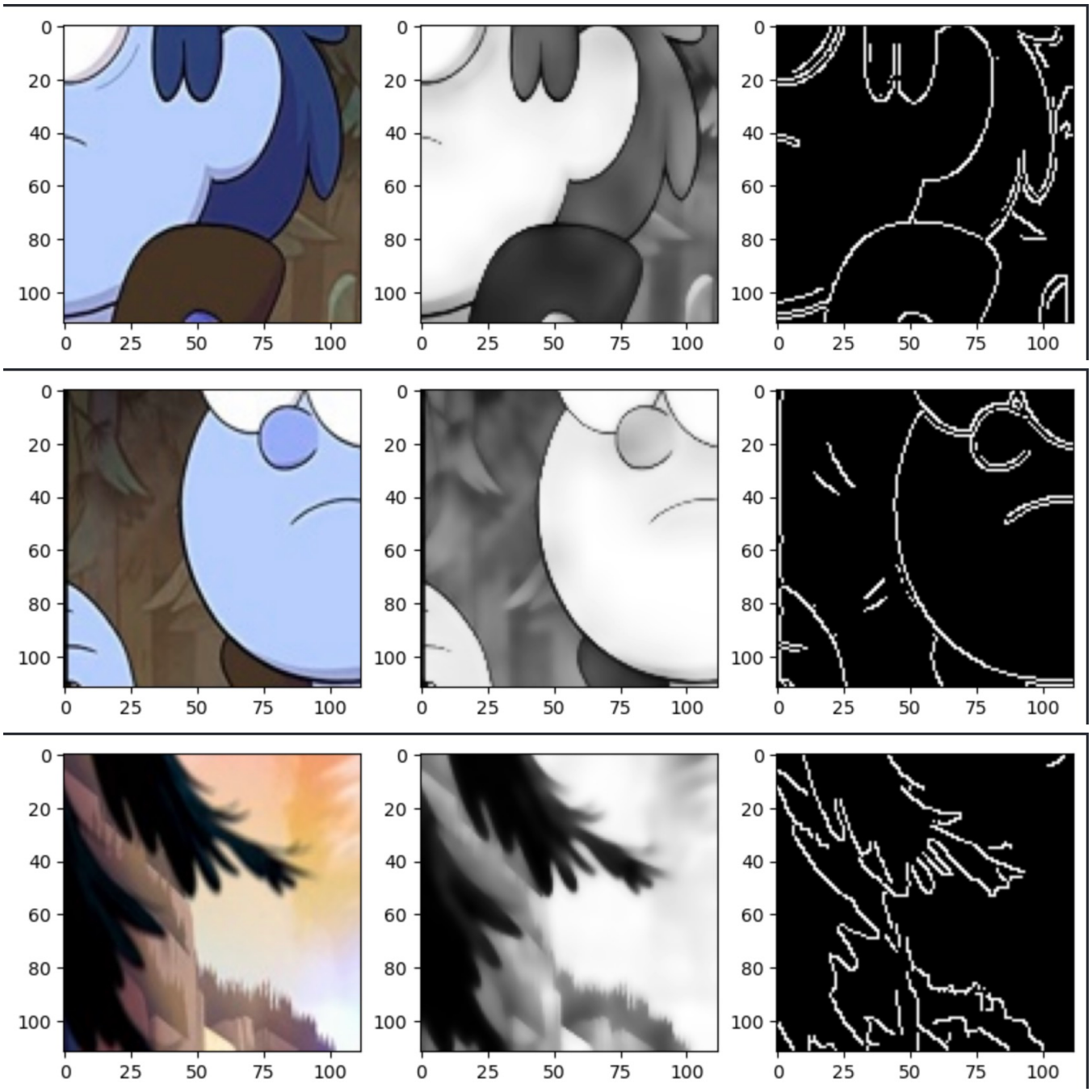


1.2) We also thought of applying thresholding before the canny edge detector to remove the black strokes that caused the double edges in the first place but that messed up the results.

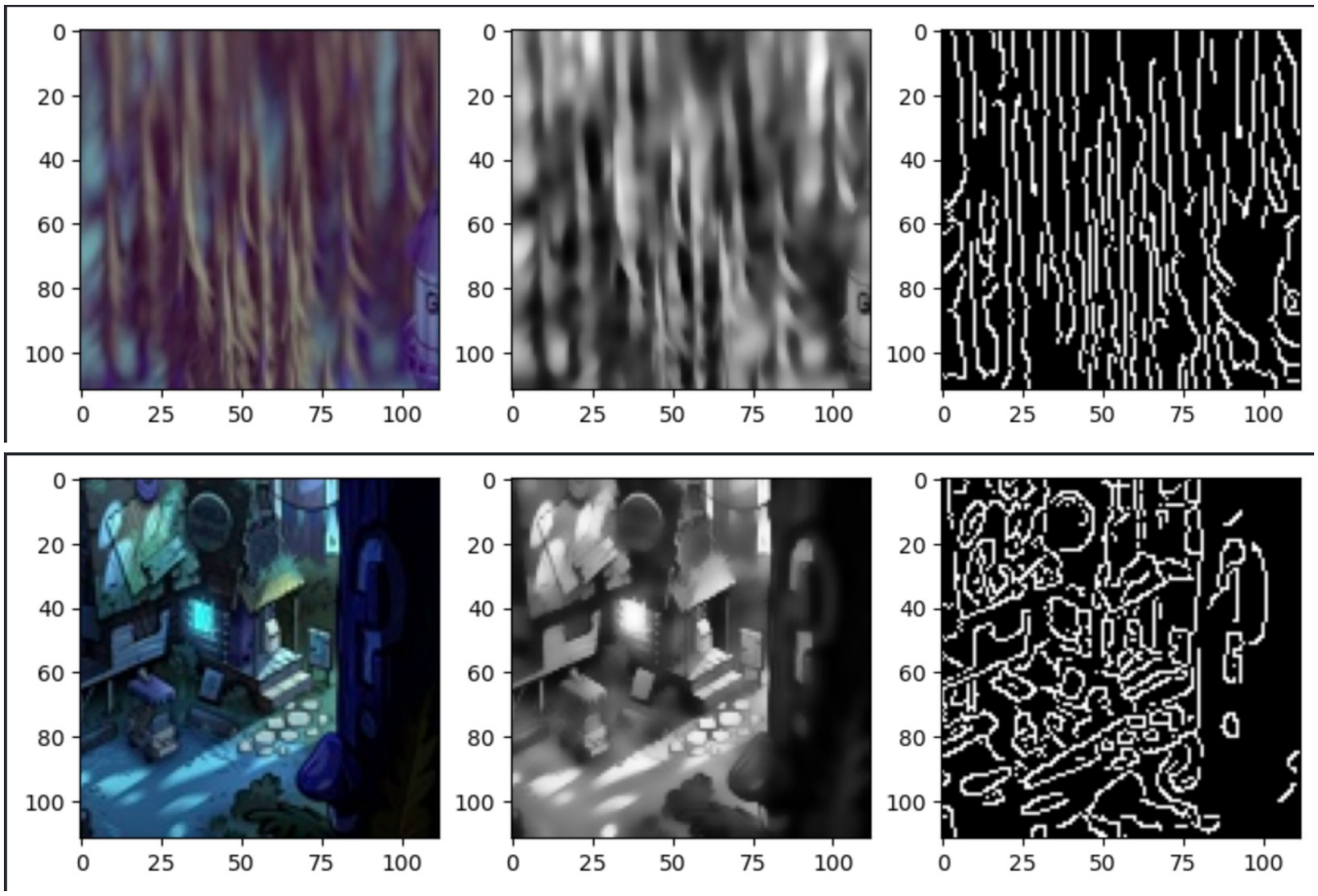


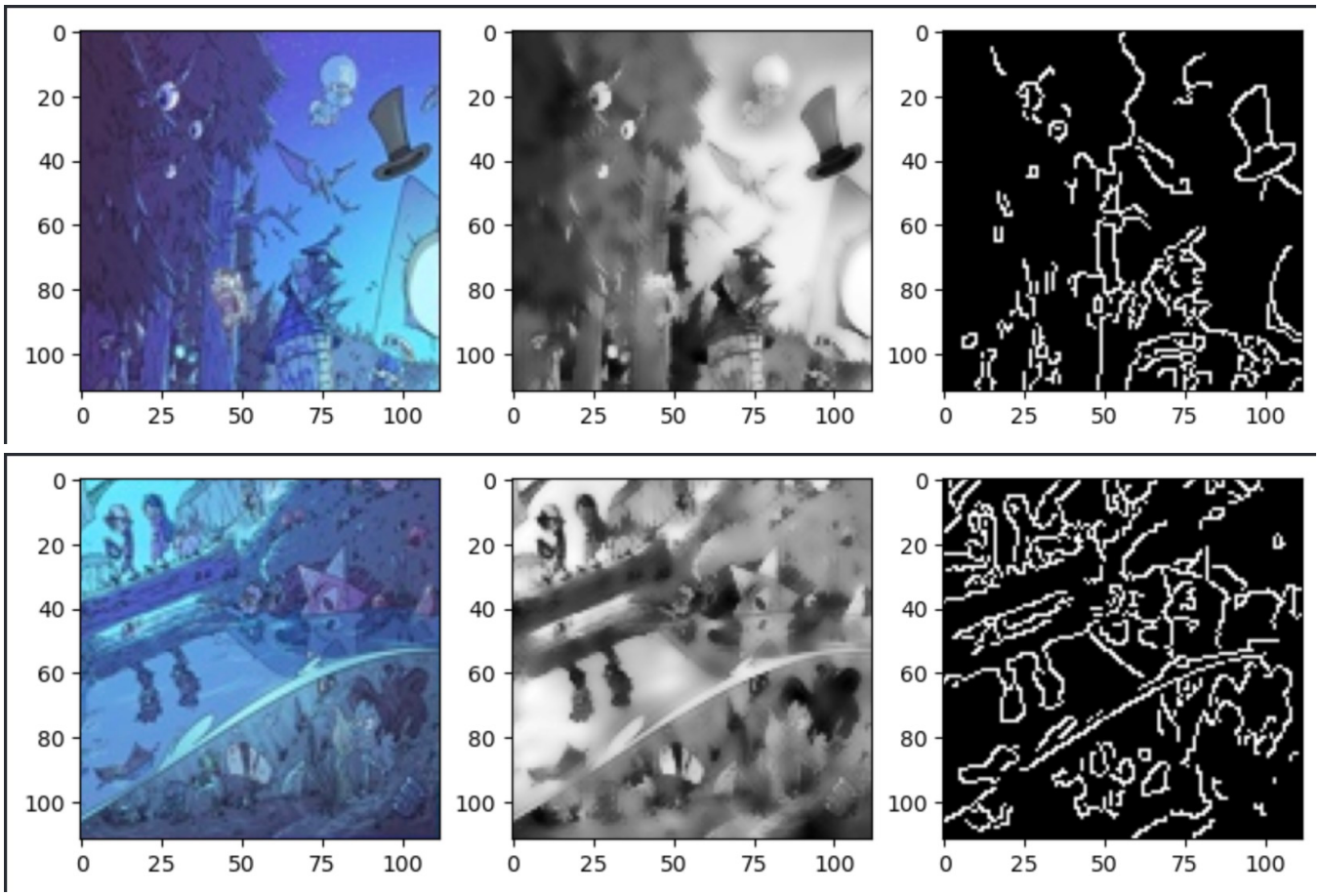
2) Some of the images produced good results without any double edges



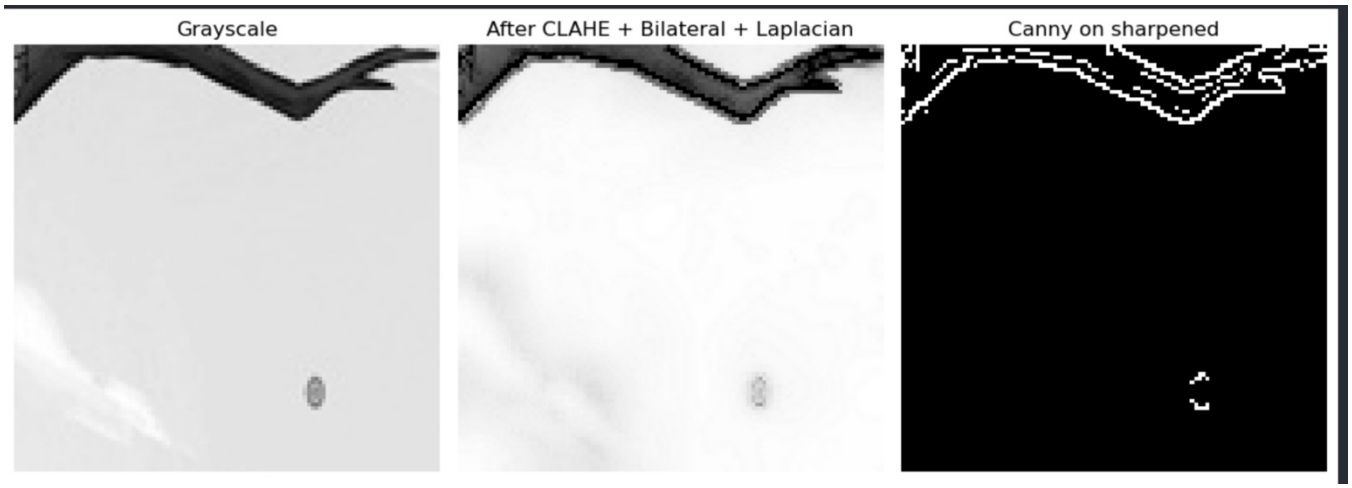

0
20
40
60
80
100
0
25
50
75
100

2) And finally some of the images were just difficult to extract the edges from as they were too detailed:





At some point we also thought about applying sharpening to the images to enhance the edges before they went into the canny edge detector but that caused the output images to have too much noise which is the opposite of what we wanted:



Segmentation

The project was given to use as folders of different format puzzle pieces 2x2 , 4x4 and 8x8 we segmented the images in these folders manually to produce 4 images from the 2x2 , 16 from the 4x4 and 64 from the 8x8 , we then apply the pipeline to each of them.

The way we did this is that the function that segments the image has the size of the image passed in (2x2 , 4x4, 8x8) and based on it the function can calculate the size of individual puzzle pieces inside the image it gets and split the image at those places to get the individual pieces returning an array of all the individual pieces from the images.

The other approach we tried was to automatically split the image into its 4 segments by detecting the edges between the 4 images and using contours to split the images at those edges. Our pipeline did the following:

As an alternative to manually splitting the puzzle images into 4 sub-tiles using fixed grid coordinates, we initially attempted to detect the boundaries between tiles automatically. In this approach, we first converted the image to grayscale and applied a bilateral filter to suppress noise while preserving edges, then ran Canny edge detection. To emphasize long grid lines, we performed morphological closing with elongated structuring elements in the horizontal and vertical directions, and then applied a probabilistic Hough transform (HoughLinesP) to detect long near-horizontal and near-vertical lines whose midpoints lay close to the image center. The idea was to use these detected lines (and subsequently contours) as the splitting boundaries between the four sub-images. However, in practice the puzzle artwork contained many strong internal edges, which generated numerous spurious line candidates; the Hough detection became highly sensitive to parameter choices, and the true tile boundaries could not be reliably isolated across the dataset. For this reason, we abandoned this automatic splitting method and reverted to a simpler geometric partitioning based on the known 2×2 / 4×4 layout of the input images.

Conclusion

The conclusion we reached is that it is basically impossible to find a single pipeline that works for all images so we tried to find one that worked for the largest portion of images so that it can be used in phase 2.

Code

You can find the code in the following github repo : https://github.com/AmirTamer-27/Gravity-Falls-Jigsaw.git