Documentation

Setup & Preparation

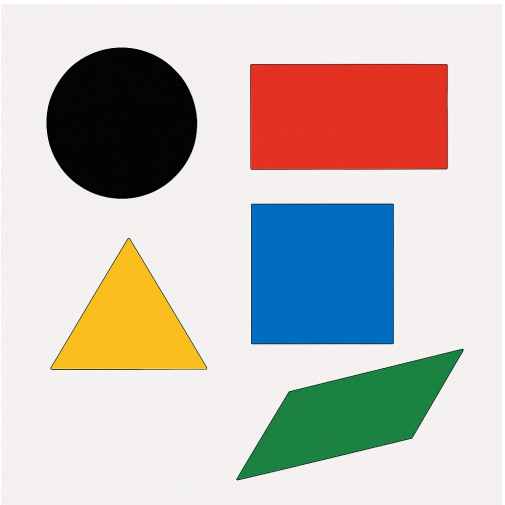
I updated the volume path for the **docker run** command inside the **README.md** to have access to the project files inside the container. I added a missing include directory in the **setup.py**. And included the **opencv_imgcodecs** module for the C++ bindings in order to be able to write images to file from the **C++** code, for debugging purposes.

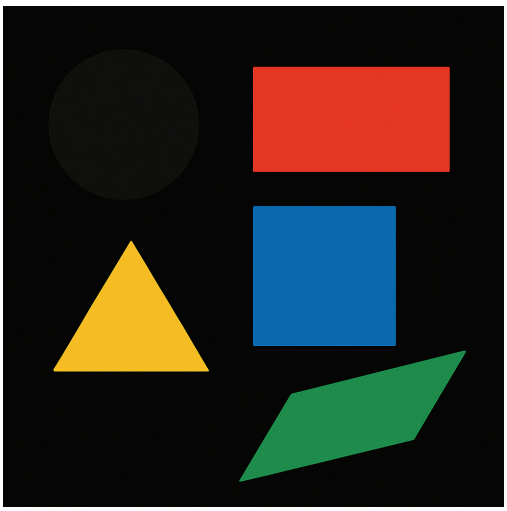
First Approach using Binary Thresholding

For the approach I used following steps:

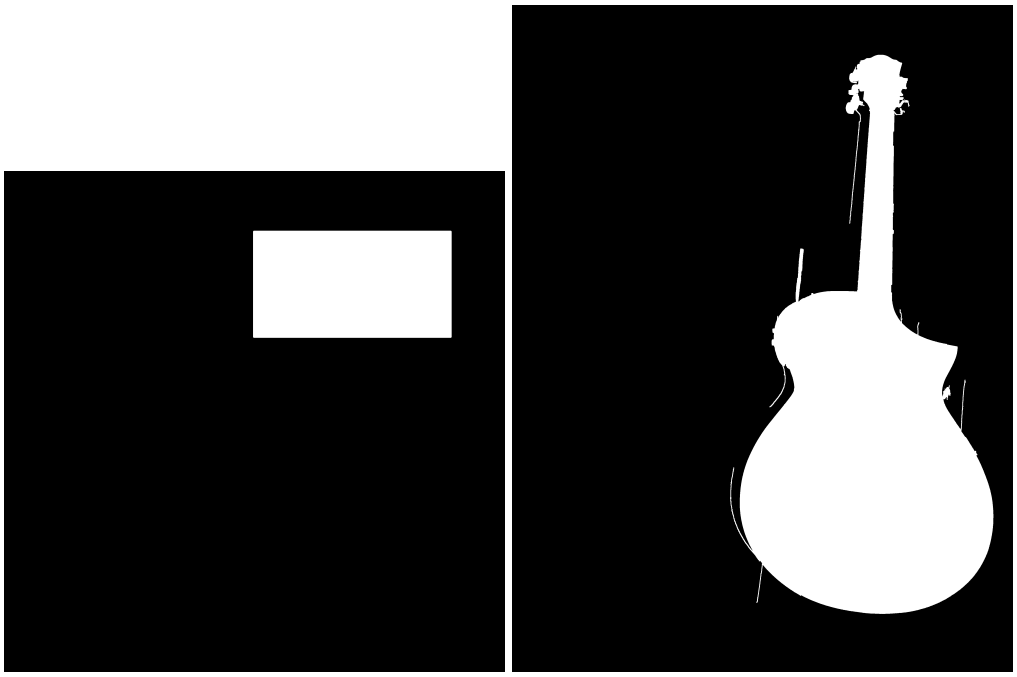
1. Convert the image to grayscale.
2. Apply binary thresholding to the image using **THRESH_BINARY_INV and THRESH_OTSU**
3. Use OpenCV's **findContours** to find the contours in the thresholded image.
4. Calculate the area for each contour and select the largest.
5. Create a mask for the largest contour, apply blurring and copy the masked area to the original image.

I used the following two images for initial testing:





I wrote debugging images after every test step to verify the results. For example after creating the mask, I created a debugging image for the masked area. This is pictured below: on the left is the mask for one of the intial test images, on the right the mask for an image from one of the latter tests.



I repeated this for every step of the process.

I encountered some issues here with the first approach, the approach did not work for the image with the black and the white background at the same time, the thresholded image had to be inverted depending on the background color.

Second Approach using Dynamic Thresholding

I tested some methods for dynamically determining the use of **THRESH_BINARY** or **THRESH_BINARY_INV**. First I used a method that check the average color of the borders of the thresholded image, depending on the outcome of this check I used either **THRESH_BINARY** or **THRESH_BINARY_INV**. This worked for the two initial testing images, however I felt that this was not very robust as a solution, that there would be plenty of examples where this approach would fail.

Next I tried an approach where I apply both thresholding methods and then calculate areas for both results, and pick the absolute largest area from both results. This again worked for the two initial testing images, however I had to filter out areas where the entire selection rectangle was selected as the shape. This again felt not very robust and I switched away from using thresholding.

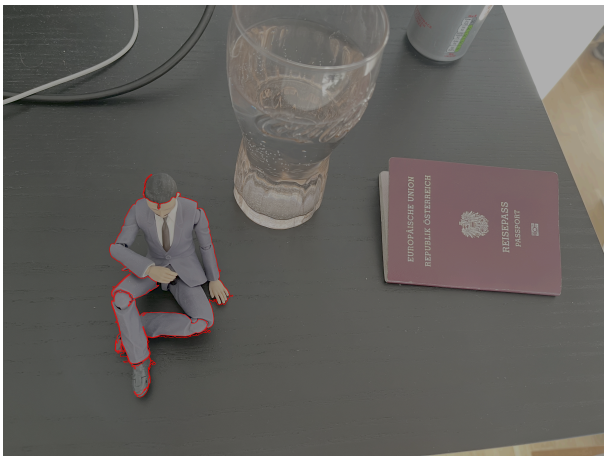
Third Approach using Canny Edge Detection

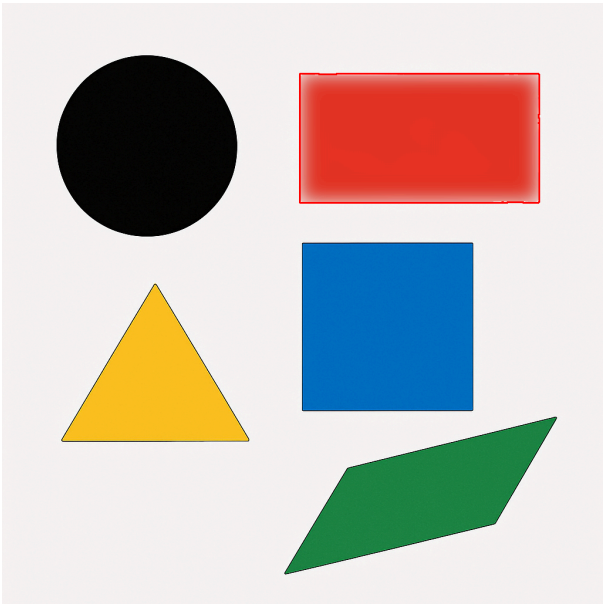
I switched to **Canny edge detection**, and additionally added a preprocessing step applying **Gaussian blur** to the grayscale image before running the edge detection. So the steps for the third approach are:

1. Convert the image to grayscale.
2. Apply Gaussian blur.
3. Apply Canny edge detection to the blurred image.
4. Run dilation on the edges to close gaps.
5. Use OpenCV's **findContours** to find the contours in the dilated edges.
6. Calculate the area for each contour and select the largest.
7. Create a mask for the largest contour, apply blurring and copy the masked area to the original image.

This approach worked immediately for the two initial testing images and I started extending the test by selecting a more diverse set of images and changing the **C++** code to not apply a blur and instead draw a red border around the largest area.

I pictured below some of the results of the extended testing:





As can be seen from the test output the approach is not perfect, however I liked how well it worked for the initial images and that it worked well for the real life images where the objects are not overlapping.

Further Improvements

I created a test branch where I experimented with a few different approaches using different parameters. I tested an approach using **approxPolyDP** to approximate complex polygons from the detected contours. This did not work well, and in addition I extended the existing **C++** function to receive more arguments to allow testing different parameters for example for the Kernel size of the **Gaussian blur** or the **Canny edge detection** thresholds.

I also wrote a shell script that runs the **python script** using different parameters and for every set of parameters creates a separate output directory and in this directory it also save a text file containing the parameters used for the run.

This can be found in the **test** branch of the repository. However, I did not do any extensive testing using this method.

Conclusion

I did not try more sophisticated approaches using machine learning techniques, this would probably have been the next step if I would continue with this work. And for sure having a more concrete use case would also decide what direction to take with the next iterations.

Finally, I added a detection in the **python script** to check if the selection rectangle is larger than the image, in this case the selection is trimmed to fit the image boundaries.

Full disclosure: I used ChatGPT and Gemini throughout the process, mostly as a search engine, however when I used code suggestions I checked everything provided by the AI to ensure that I have a full understanding of the code and the decisions made inside the code. For producing this documentation I did not use any AI tools.

All the images in this documentation can be found in the doc_images folder.