TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

IMAGE PROCESSING PROJECT

IMAGE TO G-CODE GENERATOR

STUDENT: GYARMATHY CSABA
TEACHER: FLORIN IOAN ONIGA
TEACHING ASSISTANT: VARGA RÓBERT
YEAR: III.
ACADEMIC YEAR: 2022-2023.

Contents

1. Project Description

The project aims to generate G-code, a machine-readable language used in computer-aided manufacturing (CAM), from a given image's boundary. G-code is commonly used to control CNC (Computer Numerical Control) machines, such as 3D printers or milling machines.

The program starts by detecting the boundary of the image, which represents the outline of the desired shape. It uses an edge detection algorithm to identify the pixels that form the boundary. Once the boundary is obtained, the program generates G-code instructions to move a tool along the boundary, effectively recreating the shape represented by the image.

The generated G-code file includes commands to set the positioning mode, units of measurement, and home all axes. It then sets the starting point as a rapid move (G00 command) and generates G01 commands for each point along the boundary, ensuring a smooth toolpath. Finally, the G-code file concludes with commands to move to a safe Z height and an end-of-program command.

Additionally, the program provides an option to detect and handle holes within the shape. It identifies the inner contours (holes) within the boundary and generates G-code instructions to trace those contours as well. The resulting G-code file can be used to precisely reproduce the image's shape using a CNC machine.

Overall, this project combines image processing techniques with G-code generation to enable the translation of images into physical shapes using CNC machines. It provides a convenient way to convert images into machine-readable instructions for automated manufacturing processes.

2. Related work and research

There have been various approaches to solving the problem of generating G-code from an image's boundary. Researchers and developers have shared their solutions through papers, tutorials, and online resources, providing valuable insights into the process.

One common technique used in these approaches is image processing, which involves analyzing the image to identify its boundary. Edge detection algorithms, such as the Canny edge detector or the Sobel operator, are commonly employed to locate the pixels that form the edges of the shape. These algorithms detect changes in intensity or color gradients, which often indicate the presence of boundaries. By applying these algorithms to the image, the boundary can be extracted.

However, detecting and handling holes within the shape can be a challenging task. Holes in an image's boundary are typically represented as inner contours that need to be traced separately. Some solutions utilize region growing or flood-fill algorithms to identify the enclosed areas within the boundary. By examining the connectivity of pixels, these algorithms can determine which areas represent holes. Once the holes are identified, the same edge detection techniques can be applied to their boundaries to extract the inner contours.

It is important to note that these solutions typically work on binary grayscale images, where only black and white colors are used to represent the shape. This simplification allows for straightforward boundary detection and hole identification. Converting an input image to a binary format is often done by thresholding, where a threshold value is chosen to separate the foreground (shape) from the background. Adaptive thresholding techniques can also be employed to handle varying lighting conditions and noise in the image.

To gather information for this project, I've utilized the Image Processing laboratories and courses, which provided a foundation in the relevant concepts and techniques. Additionally, researching online resources such as tutorials, forums, and open-source projects offered practical examples and implementation details.

3. General Approach and High Level Architecture

In the context of using OpenCV for the project, the general approach and high-level architecture can be outlined. OpenCV is a popular computer vision library that provides a wide range of functions and algorithms for image processing and analysis. Its extensive capabilities make it well-suited for tasks such as boundary detection and hole identification.

The general approach to generating G-code from an image's boundary using OpenCV typically involves several key steps:

1. **Image Preprocessing:** The input image may require preprocessing to enhance the quality of the boundary detection process. Operations such as resizing, denoising, and contrast adjustment can be applied to improve the image's clarity and remove unwanted artifacts.
2. **Thresholding:** To convert the input image into a binary format, a thresholding technique is employed. This involves selecting an appropriate threshold value to separate the foreground (shape) from the background. OpenCV provides various thresholding methods, including global thresholding, adaptive thresholding, and many more.
3. **Edge Detection:** Once the image has been thresholded, edge detection algorithms can be applied to identify the boundaries of the shape. OpenCV offers several edge detection algorithms, such as the Canny edge detector and the Sobel operator. These algorithms analyze the intensity or color gradients in the image to detect edges.

4. **Hole Identification:** To detect and handle holes within the shape, additional processing steps are required. OpenCV provides functions like contour detection, contour approximation, and contour hierarchy analysis, which can be used to identify inner contours representing the holes. By analyzing the connectivity and hierarchy of the contours, holes can be distinguished from the outer boundary.
5. **G-code Generation:** Once the boundary and hole contours have been identified, the next step is to convert them into G-code instructions that can be understood by CNC machines or 3D printers. The specific G-code generation process may vary depending on the desired machine and application. However, it typically involves mapping the image coordinates to the appropriate machine coordinates and generating the necessary movement commands (e.g., G0, G1) to trace the boundary and inner contours.

Regarding high-level architecture or module organization, the project can be divided into several components or modules:

1. **Input Module:** This module handles the input image and any necessary preprocessing operations, such as resizing or denoising.
2. **Thresholding Module:** Responsible for converting the preprocessed image into a binary format using appropriate thresholding techniques.
3. **Edge Detection Module:** This module applies edge detection algorithms to detect the boundaries of the shape based on the binary image.
4. **Hole Identification Module:** Handles the identification and extraction of inner contours representing holes within the shape.
5. **G-code Generation Module:** Converts the extracted contours into G-code instructions, considering the specific requirements of the target machine or printer.

It's worth noting that the exact architecture and module organization can vary depending on the specific implementation and requirements of the project. In my case, there is no image preprocessing or thresholding, as the selected input images are already binary grayscale images.

4. Implementation Details

4.1. Basic Edge Detection Algorithm

```
function calculate_perimeter(img)
    boundary = empty list

    // Find P_0
    P_0 = None
    for i in range(img.rows):
        for j in range(img.cols):
```

```
            if img(i, j) == 0:
                P_0 = (i, j)
                break
        if P_0 is not None:
            break

    boundary.push_back(P_0)

    n8_di = [0, -1, -1, -1, 0, 1, 1, 1]
    n8_dj = [1, 1, 0, -1, -1, -1, 0, 1]

    dir = 7
    P_current = P_0

    while True:
        new_dir = None
        if dir % 2 == 0:
            new_dir = (dir + 7) % 8
        else:
            new_dir = (dir + 6) % 8

        for k in range(8):
            new_i = P_current.y + n8_di[new_dir]
            new_j = P_current.x + n8_dj[new_dir]

            if img(new_i, new_j) == 0:
                P_current = (new_i, new_j)
                boundary.push_back((new_i, new_j))
                dir = new_dir
                break

            new_dir = (new_dir + 1) % 8

        if boundary.size() > 2 and boundary[0] == boundary[boundary.size() -
2] and boundary[boundary.size() - 1] == boundary[1]:
            break

    // Draw boundary on the image
    boundary_img = create image with same size as img
    for i in range(boundary.size() - 1):
        line(boundary_img, (boundary[i].second, boundary[i].first),
(boundary[i + 1].second, boundary[i + 1].first), (0, 0, 255), 1)

    display boundary_img

    return boundary
```

The function takes a binary grayscale image `img` as input and initializes an empty list `boundary` to store the boundary coordinates. It searches for the starting point `P_0` by iterating through the rows and columns of the image until it finds the first black pixel (value 0). `P_0` is added to the `boundary` list. The variables `n8_di` and `n8_dj` represent the change in row and column indices, respectively, for eight different directions (neighbors). The function enters a loop that continues until the boundary is closed, i.e., the last two points in the `boundary` list match the first two points. Inside the loop, the function calculates a new direction `new_dir` based on the current direction `dir`. It iterates through the eight directions, checking the neighboring pixels starting from the calculated new direction. If a neighboring pixel is black (value 0), the current position is updated, and the coordinates are added to the `boundary` list. The direction is also updated. After iterating through all directions, the function checks if the boundary is closed by comparing the first and last two points in the `boundary` list. Once the boundary is determined, a new image `boundary_img` is created with the same size as the input image. The `boundary_img` is drawn by connecting consecutive points in the `boundary` list using red lines. The `boundary_img` is displayed, and the function returns the `boundary` list.

4.2. Edge Detection with Holes

```
function findHolesAndContours(img)
    totalHoles = 0
    holeLabels = create empty matrix with same size as img
    contour = create empty matrix with same size as img
    contourPoints = empty list
    holes = empty list

    function fillLabel(source, x, y)
        // Similar to flood fill algorithm
        // Fills a connected hole with a unique label
        // ...

    function traceHole(source, x, y)
        // Traces the points of an inner hole using chain code
        // ...

    function traceContour(source, x, y)
        // Traces the outer contour of the image using chain code
        // ...

    // Iterate through the image to find holes and contours
    for each pixel (x, y) in img:
        if img(x, y) == 255 and holeLabels(x, y) == 0:
            fillLabel(img, x, y)
            if the filled region is a hole:
```

```
                hole = traceHole(img, x, y)
                holes.push_back(hole)
            else:
                traceContour(img, x, y)

    return holes, contourPoints
```

The `findHolesAndContours` function processes an input image to find holes and contours. It initializes variables to store the total number of holes, labels for holes, contour information, and lists to store hole points and hole contours. The `fillLabel` function performs a flood-fill-like operation to fill a connected hole with a unique label. It starts from a given pixel and expands to neighboring pixels until the entire hole is labeled. The `traceHole` function traces the points of an inner hole using chain code. It starts from a given pixel and follows a specific path determined by the chain code until it completes a closed loop. The `traceContour` function traces the outer contour of the image using chain code. It starts from a given pixel and follows a specific path determined by the chain code until it completes a closed loop. The main `findHolesAndContours` function iterates through each pixel of the image. If the pixel is white (255) and has not been labeled as part of a hole, it calls the `fillLabel` function to label the connected hole. If the labeled region forms a hole, it calls the `traceHole` function to trace the hole points and adds them to the list of holes. Otherwise, it calls the `traceContour` function to trace the outer contour of the image. Finally, the function returns the list of holes and the list of contour points.

4.3. Canny Edge Detection

The Canny edge detection method is a very common and known image processing algorithm used all over the field. You can find detailed implementation details in the source material provided by the teachers, but here are the general steps of the algorithm:

1. **Gradient calculation:** Convolve the image with Sobel kernels in the x and y directions to obtain the gradient values (`dx` and `dy`).
2. **Magnitude and angle calculation**: Compute the magnitude and angle of the gradient vectors using the `dx` and `dy` values.
3. **Quantize the angles:** Map the angle values to one of eight directions (0 to 7) based on their orientation.
4. **Edge thinning:** Compare the magnitude values of each pixel with its neighbors in the gradient direction (`qi`) and suppress non-maximum values.
5. **Thresholding:** Apply thresholding to identify strong and weak edges based on the magnitude values. Pixels with a magnitude above the high threshold (`t1`) are considered strong edges. Pixels with a magnitude between the low threshold (`t2`) and high threshold (`t1`) are considered weak edges only if they have at least one strong edge neighbor.
6. **Output visualization:** Display the resulting edge map.

4.4. G-code Generation

```
function generate_gcode_holes(filename, source, scale, zHeight, feedRate)
    file = open(filename, 'w')

    write G-code header commands to file

    # Trace the outer contour
    if outerContour has enough points:
        move to starting point of outerContour
        for each point in outerContour (excluding first and last):
            write G-code command to move to point

    # Trace the inner holes
    for each hole in holes:
        if hole has enough points:
            move to starting point of hole
            for each point in hole (excluding first and last):
                write G-code command to move to point

    write G-code footer commands to file

    close the file

    display generated G-code file message

end function
```

Generally speaking, creating a G-code file is the same process after the borders of an image have been traced. First, we set some parameters that the CNC identifies in the header, such as feed rate, height of the cutting tool, etc. Then, we trace the outer contour, and later the inner holes, and generate the commands that yield the resulting shapes. And lastly, we finish the generation by writing the footer of the file, that usually just contains code to restore the machine into its initial state.

Note: My G-code generation involves using only two commands, G00 (position cutting head to given coordinates) and G01 (straight line between the previous and given coordinates). I could not figure out how to create clockwise (G02) and counter-clockwise (G03) movements, as that would have involved really high level computation.

5. User Manual and Testing

To control the application, a very basic console interface was created. The user is presented with the following choices when running the application:

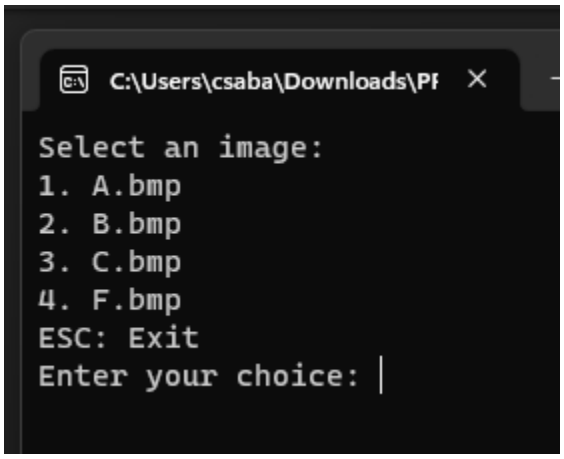


Figure 5.1. Console user interface

After an image had been selected, the program shows, after each key press, one of the implemented border tracing algorithms.

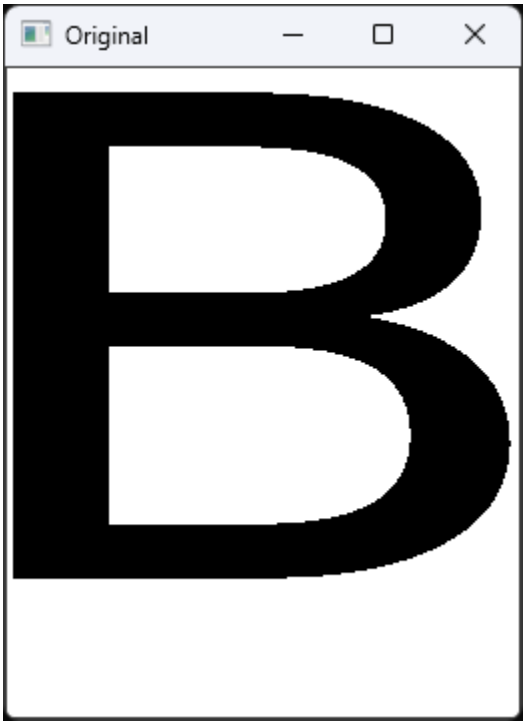


Figure 3.2 Original image

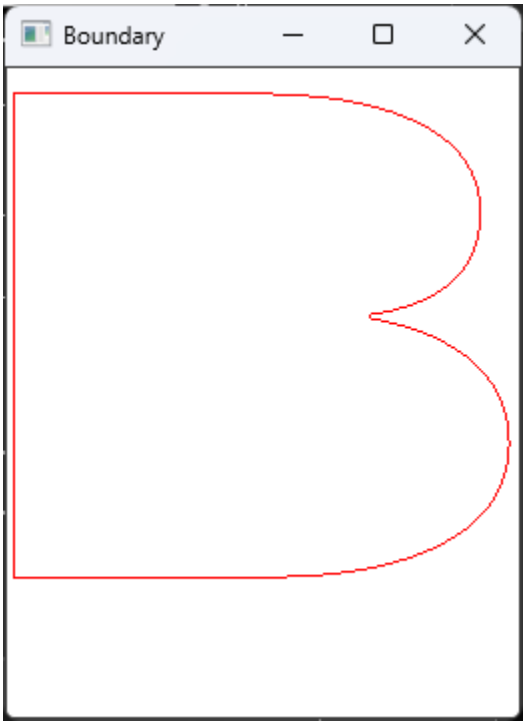


Figure 3.3. Basic edge detection

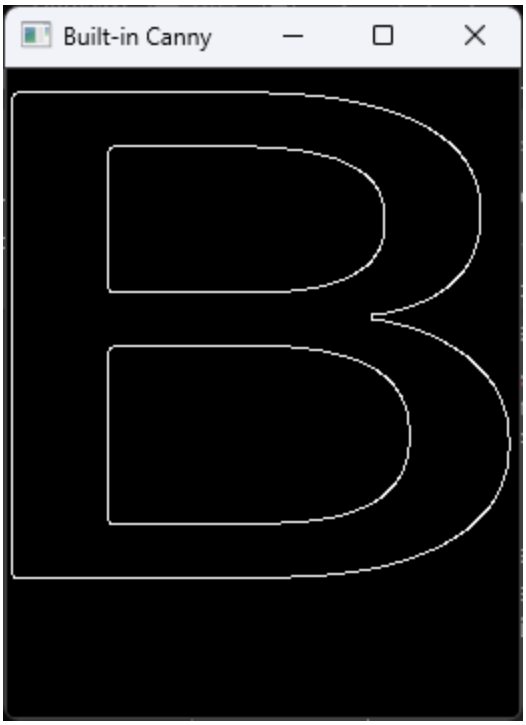


Figure 1.4. Built-in Canny

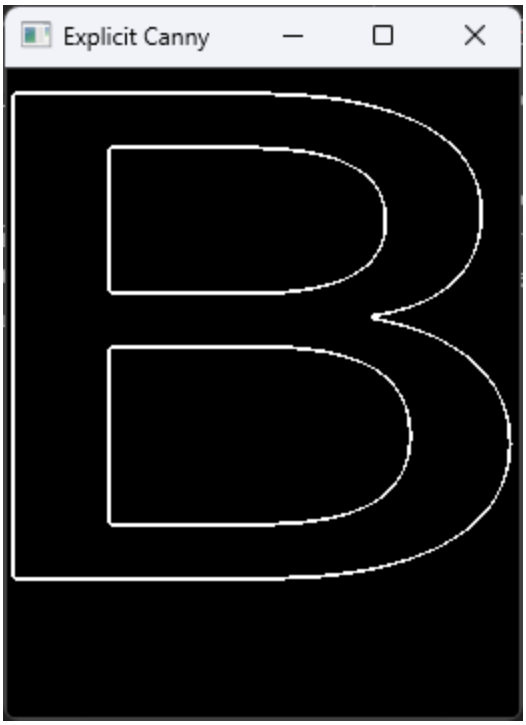


Figure 4.5. Explicit Canny

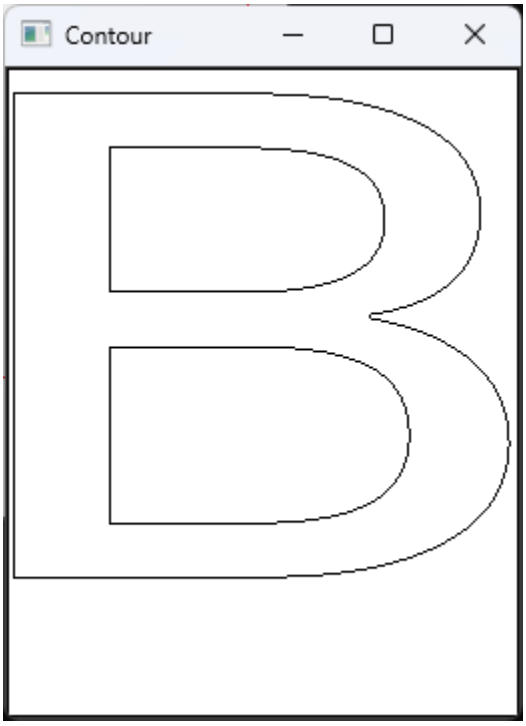
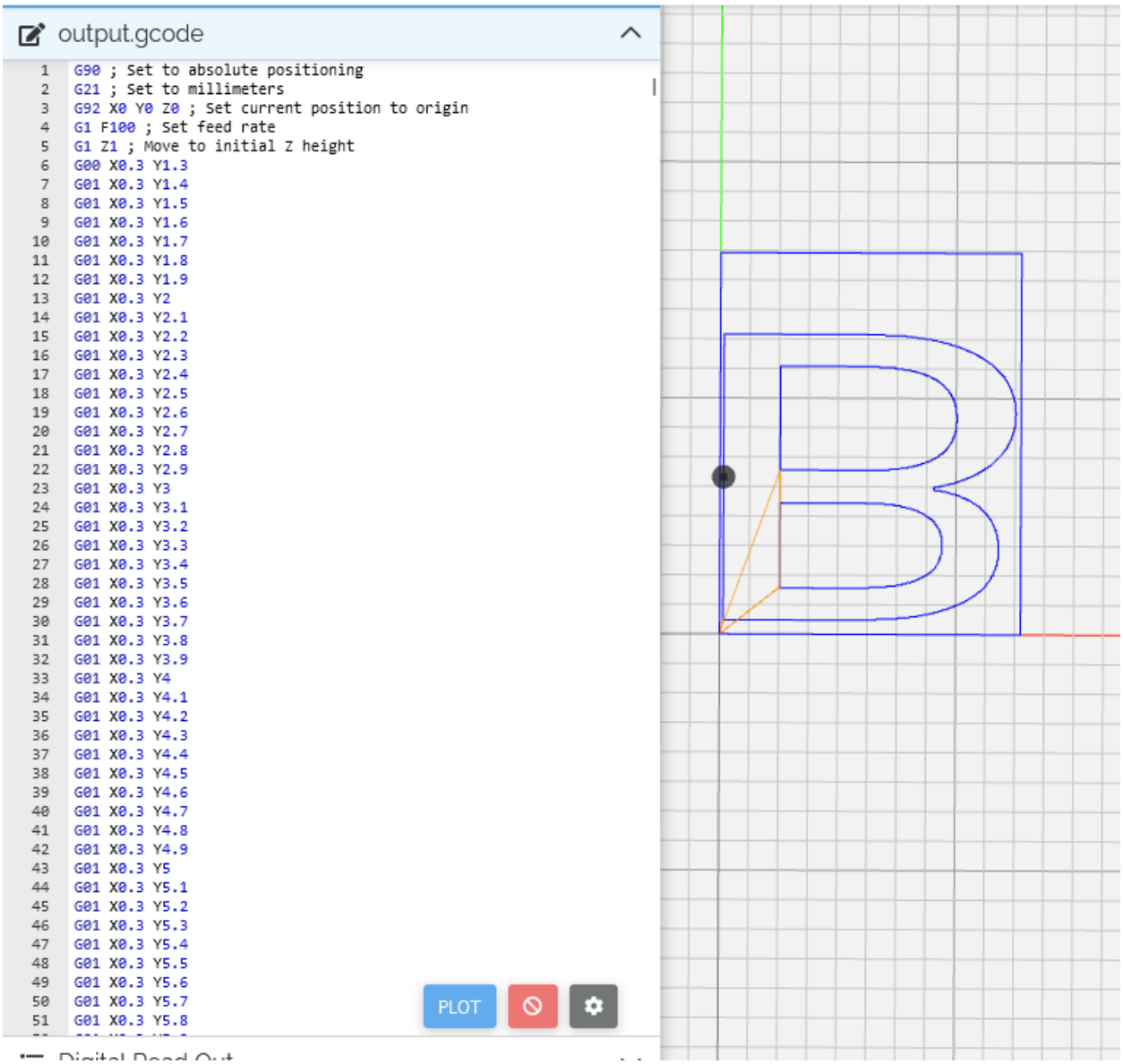


Figure 4.6. Edge detection with holes

After the next button press, two .gcode files will be created. One for the basic edge detection, and another one for the edge detection with holes. The user can verify the accutacy of these files by uploading them to an online G-code viewer and plotter, like NCViewer.



6. Conclusions

Despite the project premise being quite clear and easy, the implementation was difficult, the most difficult being the hole detection. It felt good finally working on something I am passionate about, minus the C++ part. Overall, I am satisfied with the outcome.

The basic edge detection algorithm only outputs the outer contour, while the more complex edge detection also outputs the contour of the inner holes. The Canny edge detection is built into OpenCV, but unfortunately, I was not able to generate a proper G-code output from it.

The G-code generation is not as smooth as it should be. A new command is created pixel by pixel, instead of going from one endpoint to another. This just makes the G-code file harder to read and understand. However, it does not make any difference to the CNC, as it can execute these commands sequentially in a matter of milliseconds.

7. Bibliography

Ionel Giosan, Robert Varga - Image Processing Laboratories

Florin Ioan Oniga - Image Processing Courses

https://learnopencv.com/filling-holes-in-an-image-using-opencv-python-c/

https://learnopencv.com/edge-detection-using-opencv/

https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html

Other C++ and OpenCV tutorials