**Marmara University**

**Faculty of Engineering**

**Electrical and Electronics Engineering**

**EE4065.1: INTRODUCTION TO EMBEDDED IMAGE PROCESSING#**

**Homework #3 Report & Results**

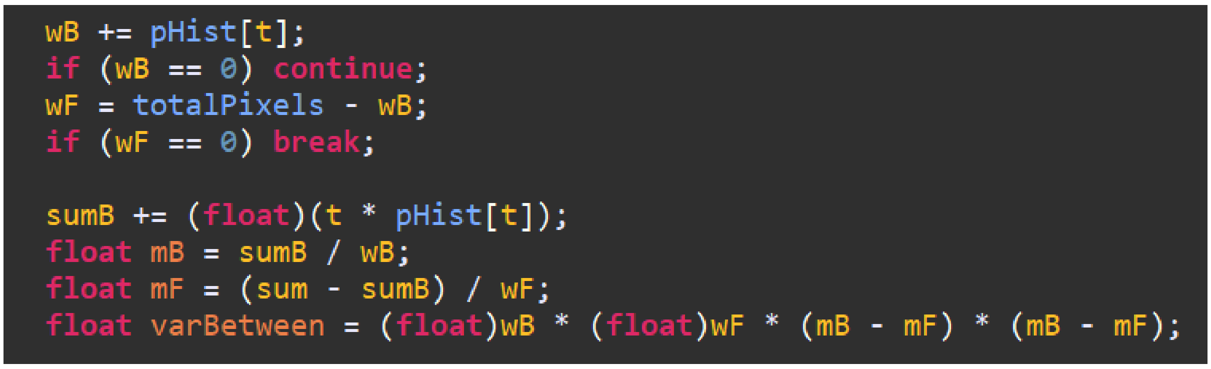
| **Name Surname** | **Student ID** |
|---|---|
| Yusuf Yiğit Söyleyici | 150723524 |
| Mehmet Açar | 150719020 |

**Introduction**

The objective of this assignment was to implement advanced image processing algorithms on the STM32 Nucleo-F446RE microcontroller. The tasks involved implementing Otsu's thresholding method for automatic binary conversion of both grayscale and color images, followed by morphological operations (Erosion, Dilation, Opening, and Closing) to process the resulting binary images. Data transfer between the PC and the microcontroller was handled via UART serial communication using a custom Python script.

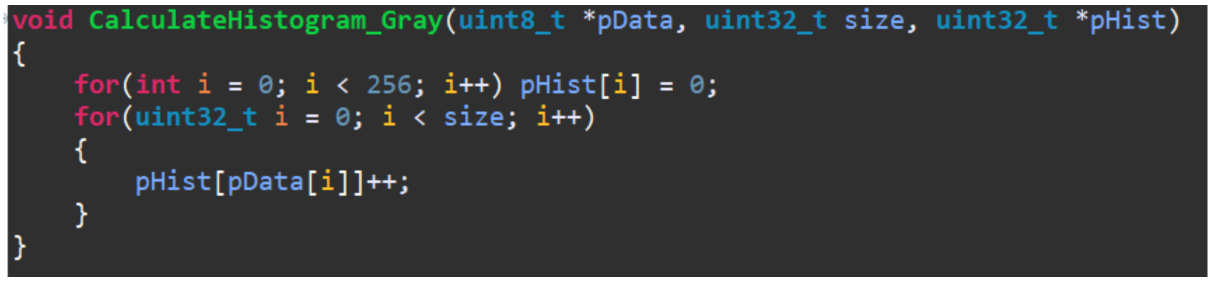
**Q1) Otsu’s Thresholding Method on Grayscale Image**

Otsu's method is a global thresholding algorithm used to automatically separate an image into foreground and background classes. It assumes that the image histogram is bi-modal (contains two distinct peaks). The algorithm iterates through all possible threshold values to find the optimal threshold that maximizes the Between-Class Variance.



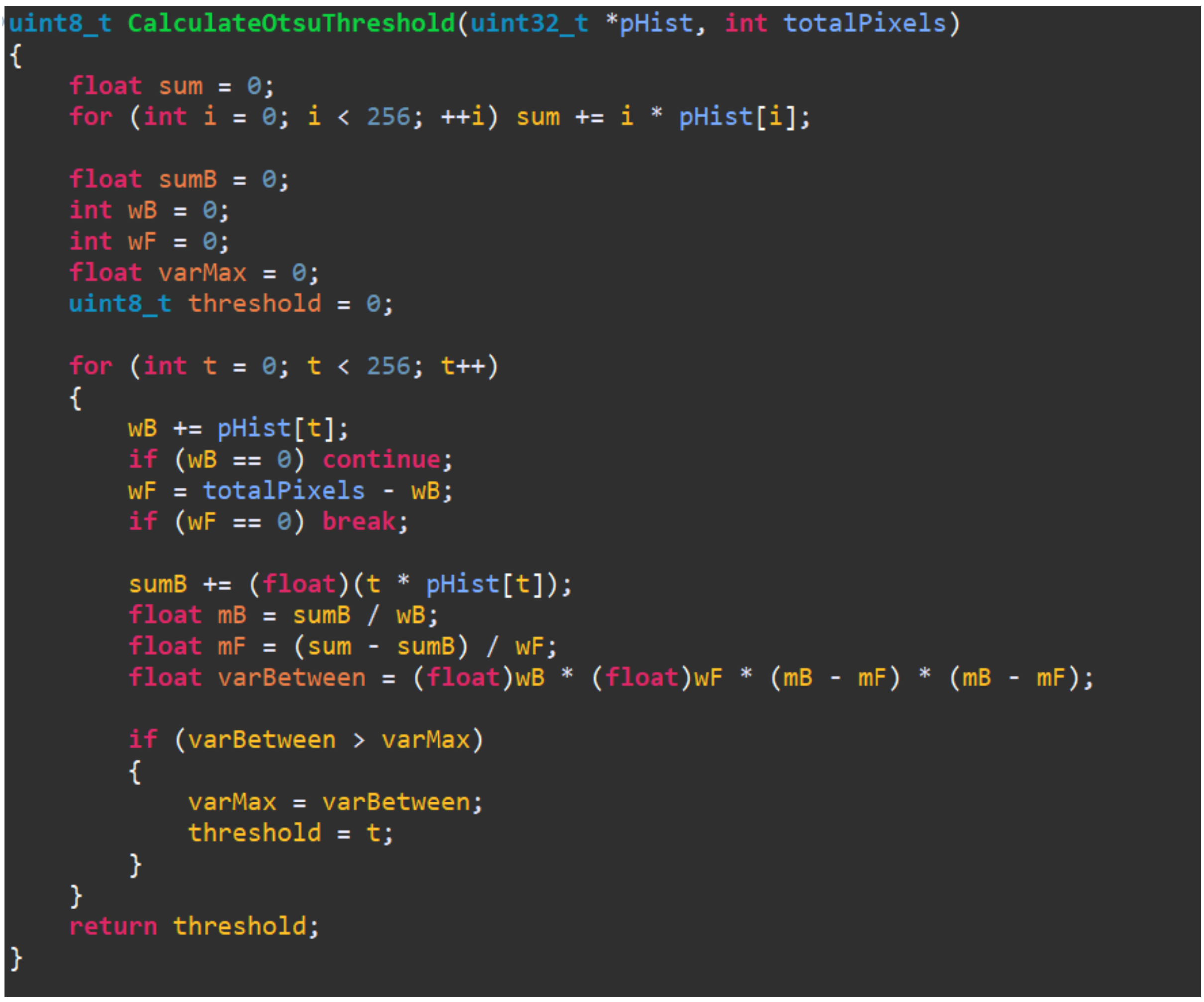
```
wB += pHist[t];
if (wB == 0) continue;
wF = totalPixels - wB;
if (wF == 0) break;

sumB += (float)(t * pHist[t]);
float mB = sumB / wB;
float mF = (sum - sumB) / wF;
float varBetween = (float)wB * (float)wF * (mB - mF) * (mB - mF);
```

**Histogram Calculation:** The MCU iterates through the received 128x128 image buffer to compute the frequency of each pixel intensity.



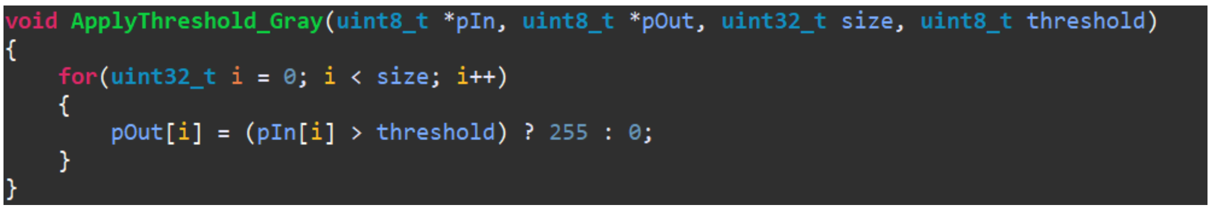
```
void CalculateHistogram_Gray(uint8_t *pData, uint32_t size, uint32_t *pHist)
{
    for(int i = 0; i < 256; i++) pHist[i] = 0;
    for(uint32_t i = 0; i < size; i++)
    {
        pHist[pData[i]]++;
    }
}
```

**Variance Maximization:** A function CalculateOtsuThreshold calculates the variance for every possible threshold. The threshold that produced the maximum variance was selected as optimal.



```c
uint8_t CalculateOtsuThreshold(uint32_t *pHist, int totalPixels)
{
    float sum = 0;
    for (int i = 0; i < 256; ++i) sum += i * pHist[i];

    float sumB = 0;
    int wB = 0;
    int wF = 0;
    float varMax = 0;
    uint8_t threshold = 0;

    for (int t = 0; t < 256; t++)
    {
        wB += pHist[t];
        if (wB == 0) continue;
        wF = totalPixels - wB;
        if (wF == 0) break;

        sumB += (float)(t * pHist[t]);
        float mB = sumB / wB;
        float mF = (sum - sumB) / wF;
        float varBetween = (float)wB * (float)wF * (mB - mF) * (mB - mF);

        if (varBetween > varMax)
        {
            varMax = varBetween;
            threshold = t;
        }
    }
    return threshold;
}
```

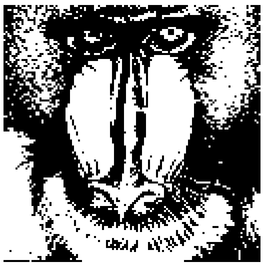
**Binarization:** The image was traversed a second time. Pixels with intensity >$T_{optimal}$ were set to 255(White), and those <=$T_{optimal}$ were set to 0(Black).



```c
void ApplyThreshold_Gray(uint8_t *pIn, uint8_t *pOut, uint32_t size, uint8_t threshold)
{
    for(uint32_t i = 0; i < size; i++)
    {
        pOut[i] = (pIn[i] > threshold) ? 255 : 0;
    }
}
```

**Input:** mandrill_grayscale.png (128x128)

**Output:** A binary image was returned to the PC, showing a clear separation of the mandrill's features from the darker regions without manual threshold selection.



**Q2) Otsu's Thresholding Method on Colorful Image**

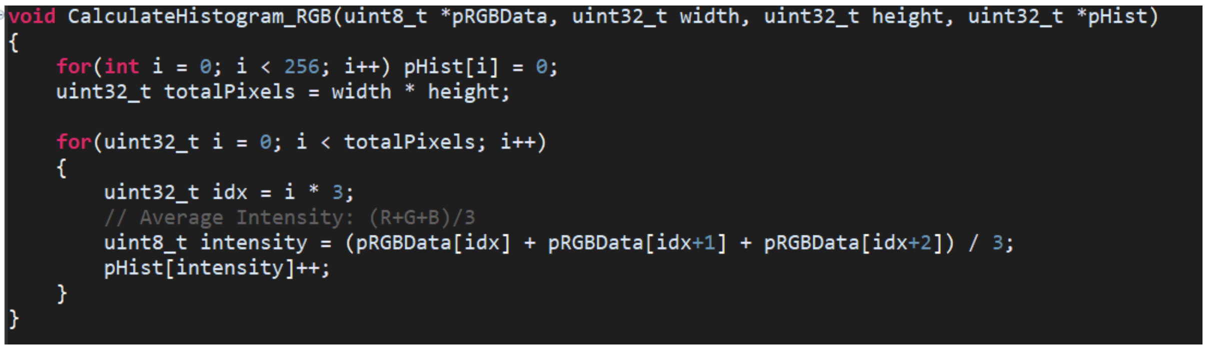
Extending Otsu's method to color images requires reducing the 3-dimensional color space (RGB) into a 1-dimensional intensity space to compute a histogram.

**Memory Management:** The input buffer size was increased to 128x128x3 bytes (48 KB) to accommodate the RGB data within the board's 128KB RAM limit.

**On-the-Fly Conversion:** To conserve memory, we did not create a separate grayscale buffer. Instead, the intensity was calculated dynamically during the histogram loop using the average method:

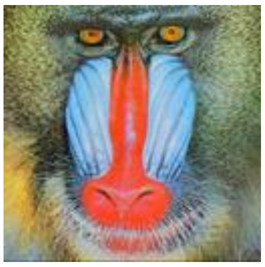
$$I = \frac{R + G + B}{3}$$

**Thresholding:** The optimal threshold was calculated from this intensity histogram. The final binary output was generated by reading the RGB source again, calculating intensity, and comparing it to the threshold.

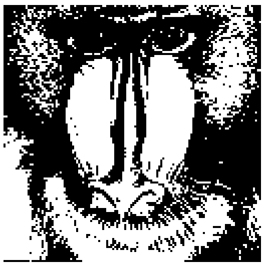


```
void CalculateHistogram_RGB(uint8_t *pRGBData, uint32_t width, uint32_t height, uint32_t *pHist)
{
    for(int i = 0; i < 256; i++) pHist[i] = 0;
    uint32_t totalPixels = width * height;

    for(uint32_t i = 0; i < totalPixels; i++)
    {
        uint32_t idx = i * 3;
        // Average Intensity: (R+G+B)/3
        uint8_t intensity = (pRGBData[idx] + pRGBData[idx+1] + pRGBData[idx+2]) / 3;
        pHist[intensity]++;
    }
}
```

**Input:** mandrill_colorful.png (128x128)



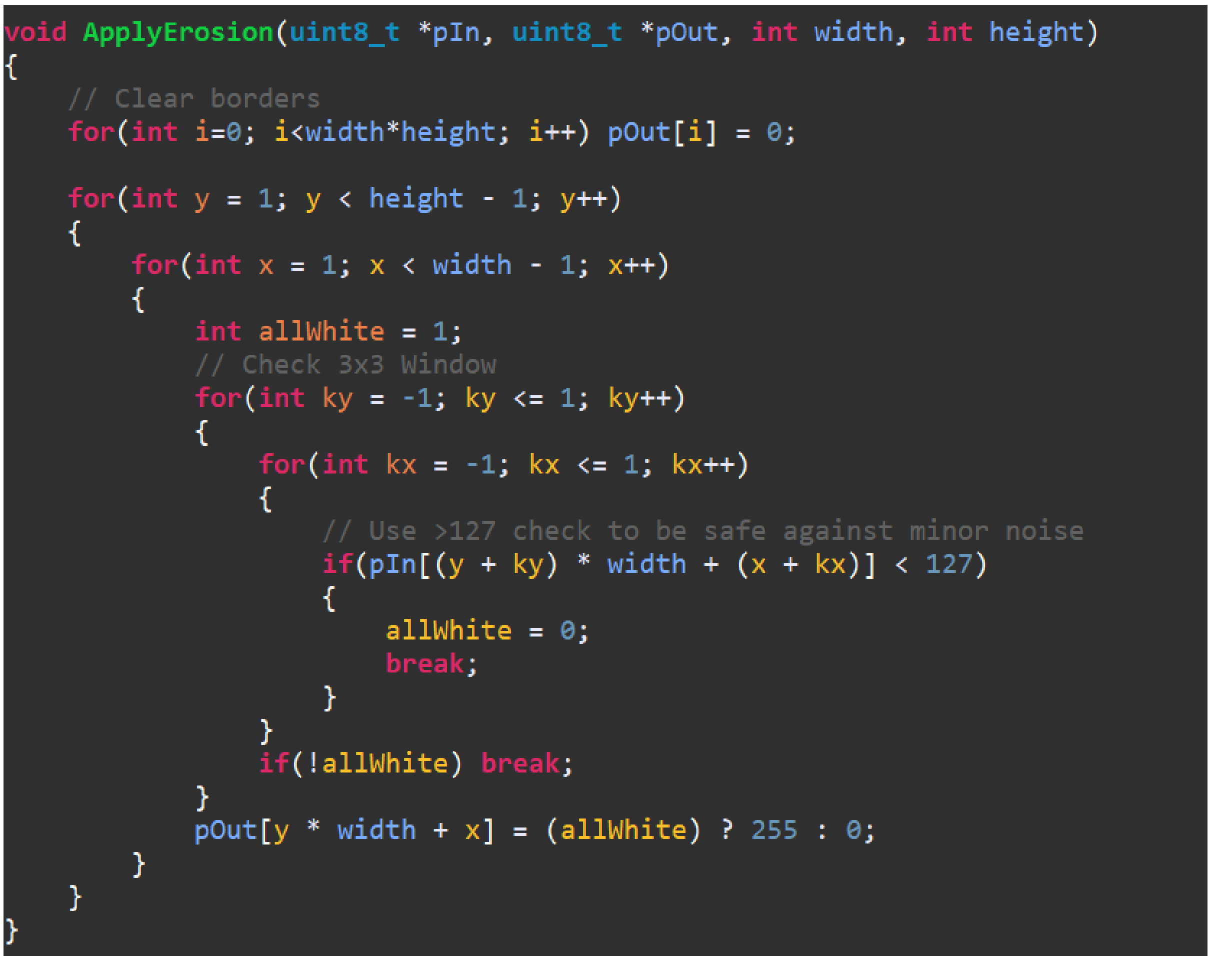
**Output:** A binary mask representing the significant visual features of the color image. The algorithm successfully segmented the color image by treating it as an intensity map.



## Q3) Morphological Operations

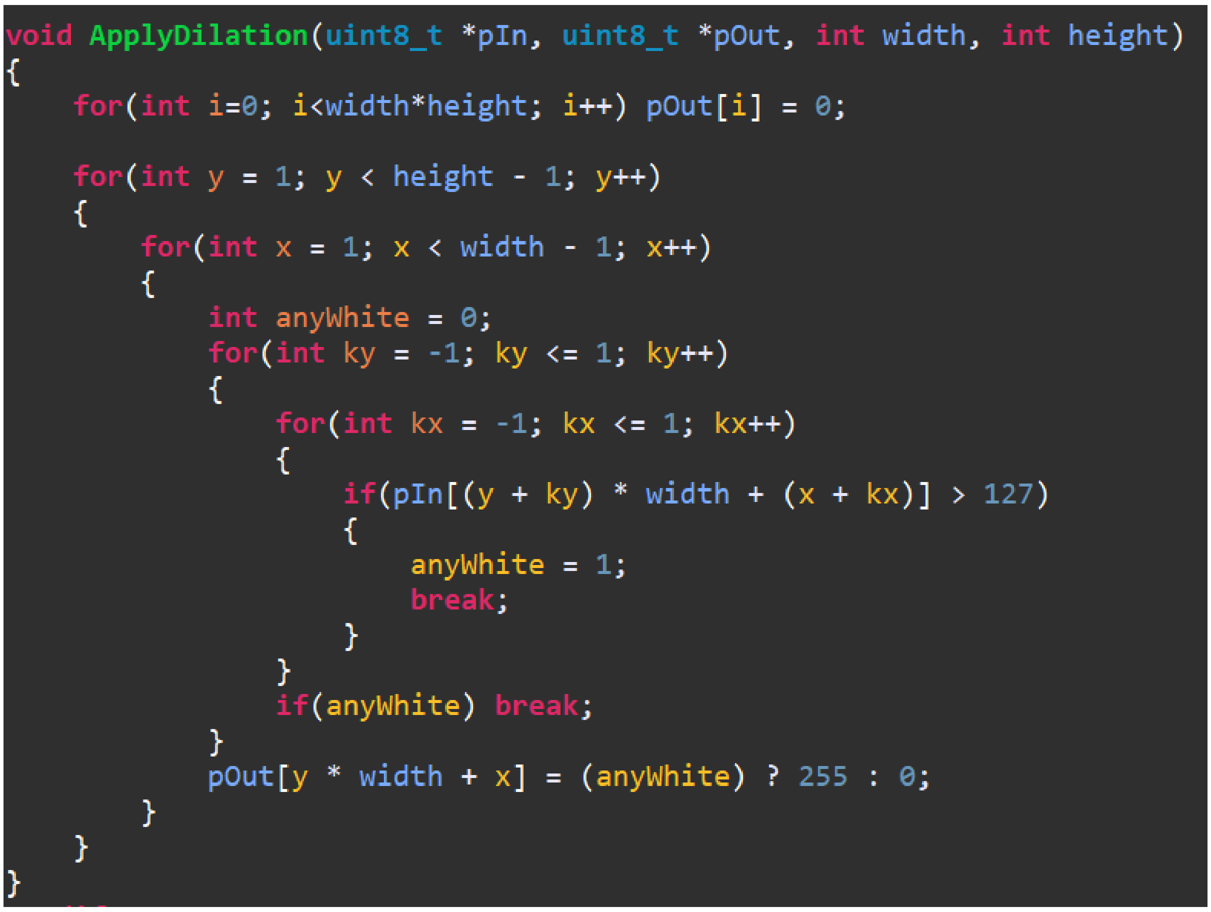
Morphological operations process images based on shapes. They work by applying a structuring element (kernel) to an input binary image.

**Erosion:** A pixel is set to 1 (White) only if all pixels in its neighborhood are 1. This shrinks white regions and removes small noise.



```
void ApplyErosion(uint8_t *pIn, uint8_t *pOut, int width, int height)
{
    // Clear borders
    for(int i=0; i<width*height; i++) pOut[i] = 0;

    for(int y = 1; y < height - 1; y++)
    {
        for(int x = 1; x < width - 1; x++)
        {
            int allWhite = 1;
            // Check 3x3 Window
            for(int ky = -1; ky <= 1; ky++)
            {
                for(int kx = -1; kx <= 1; kx++)
                {
                    // Use >127 check to be safe against minor noise
                    if(pIn[(y + ky) * width + (x + kx)] < 127)
                    {
                        allWhite = 0;
                        break;
                    }
                }
                if(!allWhite) break;
            }
            pOut[y * width + x] = (allWhite) ? 255 : 0;
        }
    }
}
```

**Dilation:** A pixel is set to 1 (White) if any pixel in its neighborhood is 1. This expands white regions and fills holes.



```c
void ApplyDilation(uint8_t *pIn, uint8_t *pOut, int width, int height)
{
    for(int i=0; i<width*height; i++) pOut[i] = 0;

    for(int y = 1; y < height - 1; y++)
    {
        for(int x = 1; x < width - 1; x++)
        {
            int anyWhite = 0;
            for(int ky = -1; ky <= 1; ky++)
            {
                for(int kx = -1; kx <= 1; kx++)
                {
                    if(pIn[(y + ky) * width + (x + kx)] > 127)
                    {
                        anyWhite = 1;
                        break;
                    }
                }
                if(anyWhite) break;
            }
            pOut[y * width + x] = (anyWhite) ? 255 : 0;
        }
    }
}
```

**Opening:** Erosion followed by Dilation.

**Closing:** Dilation followed by Erosion.

**Input:** The binary image generated in Q1 (mandrill_binary.png).

**Buffering Strategy:** Morphological operations cannot be performed "in-place" because updating a pixel affects the calculation of its neighbors. We utilized a "Ping-Pong" buffering strategy:

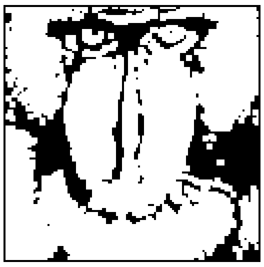
**Buffer A (img_input):** Held the source image.

**Buffer B (img_output):** Held the destination result.

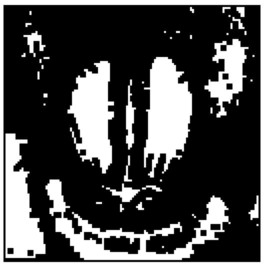
**Workflow:**

For **Opening**: The image was read from Buffer A, eroded into Buffer B, and then dilated back into Buffer A for transmission.

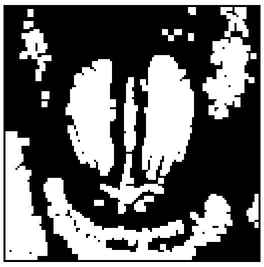
a. **Dilation**



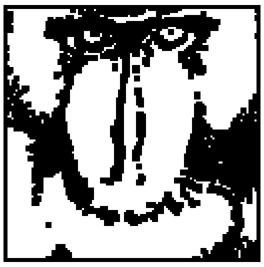
b. **Erosion**



c. **Opening**



d. **Closing**



**Conclusion**

The project successfully demonstrated the capability of the STM32F446RE to handle multi-stage image processing pipelines. By managing memory efficiently (using on-the-fly calculations for color images and double-buffering for morphology), we implemented algorithms that typically require significant computational resources. The combination of Otsu's automatic thresholding and morphological filtering provided a robust method for segmenting and cleaning images on an embedded platform.