ObNoDog: Shape Detection on Not a Dog

Jing Cao
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, U.S.
jingcao@mit.edu

Nicholas Cerone
Department of Mechanical Engineering
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, U.S.
ceronj26@mit.edu

***Abstract*—This project presents a shape detection system implemented on an FPGA, capable of identifying multiple geometric shapes such as circles, squares, triangles, and plus signs in a real-time video feed. The system employs a hardware-accelerated pipeline that combines efficient image processing algorithms to achieve accurate shape recognition. Connected Components Labeling (CCL) is applied to calculate the areas of shapes and track their centers of mass, ensuring each shape and its data is stored consistently and separately. Moore's Neighbor Tracing algorithm (Moore's) is utilized to detect and trace the perimeters of shapes. To classify shapes, the system uses a circularity measure, a ratio derived from the perimeter and area, to distinguish among a few different preset geometric forms. This approach leverages the parallel processing capabilities of the FPGA to deliver high-performance shape detection with minimal latency, making it suitable for applications such as object tracking, robotic vision, and industrial automation. The results demonstrate the system's effectiveness in identifying and classifying shapes with real-time processing speeds.**

***Index Terms*—Shape detection, no dog, FPGA, Moore's Neighbor Tracing, Connected Components Labeling**

I. Introduction

This project focuses on implementing a real-time shape detection system on an FPGA to identify geometric shapes such as circles, squares, triangles, and plus signs. The system is designed to process image data in hardware, leveraging the computational efficiency of FPGAs to achieve high-speed detection and classification. By combining Connected Components Labeling (CCL) for area calculation and center of mass tracking, Moore's Neighbor Tracing algorithm (Moore's) for boundary detection, and a circularity measure for shape classification, the system provides a robust pipeline for shape analysis.

The algorithm begins by using CCL to identify distinct objects in the image and computes their areas and centers, allowing for accurate tracking and localization. Objects with sufficiently large area are individually masked and then passed into Moore's, which calculates perimeter by tracing each object's boundary. Finally, we pass each object's area and perimeter into a module to calculate Circularity, a modified ratio of area to perimeter which allows us to distinguish shapes based on how circular they are (how much the area is maximized given the perimeter). This multi-step approach ensures that the system can handle noise and imperfections within certain shape classes reliably and efficiently.

The integration of these algorithms into FPGA hardware is tailored to exploit its parallel processing capabilities, enabling the system to operate at real-time speeds with minimal computational overhead. On an FPGA, this algorithm can be run to classify all distinct objects in parallel to one another. The result is a shape detection solution that is not only fast and reliable but also energy-efficient, making it ideal for embedded systems and applications where power and performance constraints are critical.

II. Image Processing

In this section, we describe the image processing pipeline implemented on the FPGA, focusing on the techniques used for shape detection and classification. The system integrates CCL for identifying and localizing shapes, Moore's for boundary extraction, and a custom image sprite display for visualization.

A. Camera Feed Masking (Nicholas)

In order to maximize available BRAM in which to store shapes for classification, we forego a traditional 24-bit wide frame buffer on raw camera input. Instead, we directly combinationally convert raw camera RGB pixel data to YCrCb values, create a mask through a threshold on Cr, then store the mask into a 1-bit wide BRAM. This mask is then displayed out via HDMI as though it were a traditional frame buffer, and this mask is what is input into the CCL module. Though this leaves us with low-resolution camera feed output which is not very helpful for debugging, the BRAM saved was critical for a robust implementation.

B. Connected Components Labeling (Jing)

The Connected Components Labeling (CCL) module is responsible for identifying distinct blobs (connected regions of pixels) in a binary image [1], assigning unique numeric labels to each blob, and calculating blob area and center of mass. This module uses a two-pass algorithm to efficiently label connected components and compute properties.

The algorithm is implemented using a finite state machine with nine states (see Fig. 6):

1) **IDLE:** Waits for a new frame to begin processing.
2) **STORE_FRAME:** Stores the masked frame into a frame buffer.

3) **FIRST_PASS:** Performs an initial labeling of masked pixels and builds an equivalence table for labels.
4) **SECOND_PASS:** Resolves equivalences to assign final labels.
5) **PROPERTY_CALC:** Computes area, sum of x-coordinates, and sum of y-coordinates for each blob.
6) **STORE_IN_ARRS:** Stores calculated properties in arrays.
7) **PRUNE:** Discards blobs smaller than a predefined size threshold and identifies the largest ones.
8) **TL_FRAME:** Stores masks of each blob in a frame buffer in `top_level`.
9) **OUTPUT:** Outputs the final results.

1) Two-Pass CCL Algorithm: We implement the CCL algorithm in two passes over the image.

In the first pass, the algorithm scans the masked frame in raster order. For each masked pixel encountered the module examines four neighboring pixels (west, northwest, north, northeast). If no labeled neighbors exist, the pixel is assigned a new unique numeric label. If labeled neighbors are present, the pixel inherits the smallest neighboring label. This pass results in an initial labeling where parts of the same blob may have different temporary labels.

The second pass focuses on resolving label equivalences and finalizing object properties. The algorithm identifies and merges labels that belong to the same connected component, ensuring that all pixels of a single object end up with the same label. As labels are resolved, the algorithm updates properties for each blob. The properties include the area and sums of x- and y-coordinates which are used for center of mass calcuation.

An example of this algorithm is shown in 1. We are first presented with a binary mask. In the first pass, we give an initial labeling of the blobs where blobs may have multiple different labels. In the second pass, we resolve labels that point to the same blob and get the final labeled blobs.

2) Blob Pruning and Center of Mass Calculation: After the two-pass labeling, we prune blobs with fewer pixels than a predefined minimum size. They are discarded by marking them as 0 in the intermediary frame buffers.

For each remaining blob, we calculate the center of mass using the formula

$$\text{CENTER OF MASS(X, Y)} = \left(\frac{\sum x_i}{A}, \frac{\sum y_i}{A} \right)$$

where x_i and y_i are the coordinates of the pixels in the blob and A is the total area of the blob.

C. Connecting CCL and Moore's Neighbor Tracing (Nicholas)

CCL outputs the areas and center of mass of the largest two blobs and stores them into frame buffers for usage by Moore's. Moore's requires four frame buffers to read from since it must access all eight neighbors for each pixel so CCL outputs a mask of each labeled blob into four frame buffers, making a total of eight frame buffers it outputs to since we only keep track of the largest two blobs.

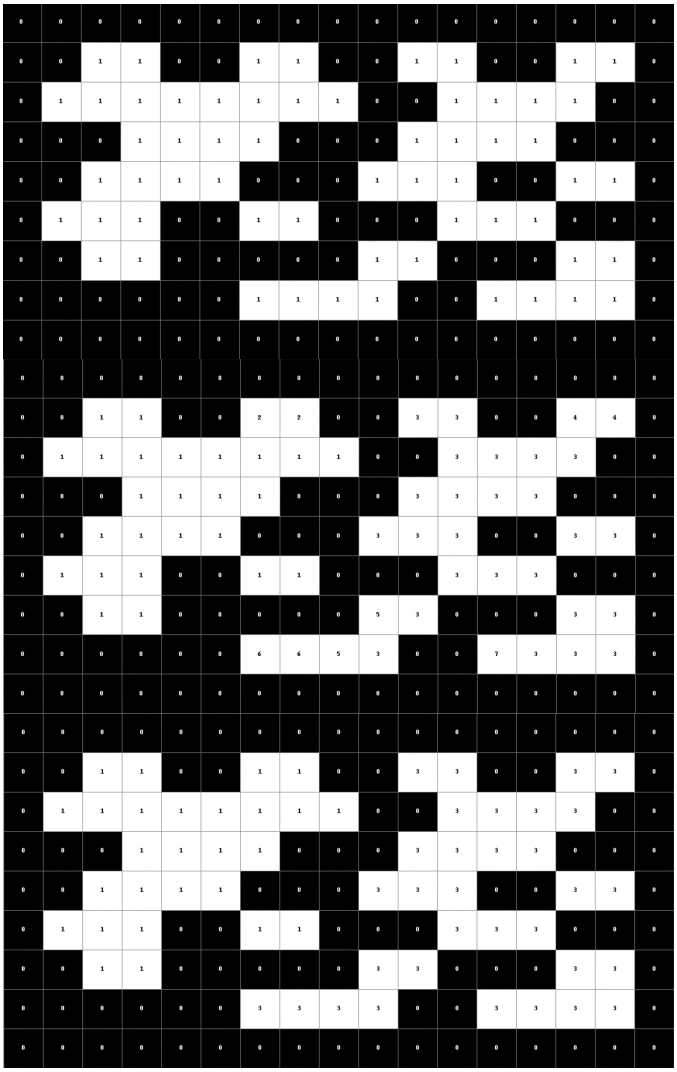

Fig. 1: Example walkthrough of connected components labeling algorithm [2].

D. Moore's Neighbor Tracing (Nicholas)

Moore's Neighbor Tracing algorithm (Moore's) is a boundary-following algorithm used to identify the perimeters of shapes in a binary image. This algorithm begins by locating the first boundary pixel in the raster scan and traces the contour by iteratively checking the neighboring pixels in a predefined order based on the previous neighbor's location [3]. It continues until it returns to the starting pixel, effectively extracting the entire perimeter of the shape.

This version of Moore's is implemented to be integrated with CCL; the Moore's module outputs addresses which are used to read from the BRAMs in `top_level` where CCL stores the blob masks. This is an efficient method because it means that Moore's does not use extra resources storing the frame in local BRAMs while also allowing for CCL to easily write to the BRAMs as they are in `top_level`.

The algorithm is implemented using a finite state machine with four distinct states (see: Fig. 7):

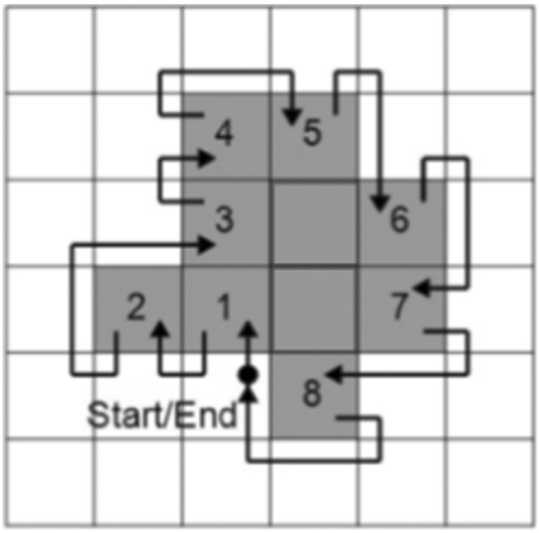

Fig. 2: Example walkthrough of Moore's Neighbor Tracing algorithm [4].

1) **IDLE:** The module waits for a new frame signal to begin storing the frame.
2) **SEARCHING:** The module scans the frame line-by-line to identify the first boundary pixel in the object, if such an object exists on the screen. Note that we need not store the frame in local BRAMs, as all memory accesses happen to BRAMs in top level which set prior to providing the new frame signal.
3) **TRACING:** After identifying the first boundary pixel, the module performs Moore's Neighbor Tracing algorithm. During each cycle, eight adjacent locations to the current pixel of interest are pulled from BRAM and checked in a specified order based on the previous neighbor[3] to determine the next pixel of interest. This process continues until the module returns to the first found boundary pixel.
4) **OUTPUT:** Once the object has been fully traced, the module provides a valid output signal and outputs the calculated perimeter of the shape.

E. Transparent Image Sprite (Nicholas)

To facilitate visual feedback and debugging, the processed data is displayed on a monitor using a custom image sprite renderer. This renderer overlays detected shapes with their centroid markers, and each shape is annotated with its corresponding classification result in text (e.g., circle, square, triangle, plus) are displayed for clarity. This is done through a modified Image Sprite module, whereby input images are binary masked and only pixels which are masked in the image are actually displayed onscreen. This effectively allows for the display of transparent images (the custom shape centroid markers and classification labels) overtop of the camera display without any artifacts. As for setup, this required writing a custom Python file to convert `.png` images into `.mem` files of 1-bit width based on whether the background is transparent at that pixel or not. The sprite visualization helps to create an interactive and informative user interface, not only for debugging, but for outputting results in an intuitive manner.

III. Shape Classification

Our current shape classification algorithm utilizes a naive and computationally efficient method based on circularity, which is a measure of how closely the shape approximates a perfect circle. Circularity is defined as the ratio of the area (A) to the square of its perimeter (P) scaled by 4π:

$$\text{Circularity} = \frac{4\pi A}{P^2}$$

To simplify the computation and avoid handling decimals, we multiply this ratio by a factor of 100 (letting $\pi \cdot 100 = 314$) and account for any noise which may cause the value to exceed 100 in our thresholds. Circularity provides a normalized metric that helps differentiate between geometric shapes based on their boundary and area characteristics.

The larger the circularity, the more likely it is to be a circle. We have four thresholds to differentiate between shapes:

1) Circularity $>$ 128: classify as a circle
2) Circularity $>$ 80: classify as a square
3) Circularity $>$ 73: classify as a triangle
4) Circularity $\leq$ 73: classify as a plus sign

These thresholds were identified through calculation then refined through trial and error.

IV. Results

The current implementation demonstrates the feasibility and computational efficiency of using FPGAs for real-time shape detection and classification of multiple objects. Our implementation can label and analyze up to two shapes utilizing the CCL algorithm, perform Moore's to calculate the perimeter of detected shapes, and classify shapes based on their circularity values. Images of final results are shown in Fig. 3.

Improving on our preliminary results, the implementation can now detect and classify more than one shape at a time and prunes noise from labeling in CCL. This allows it to often classify shapes correctly, even when multiple shapes are on the screen at once.

V. Evaluation

A. Classification Accuracy

The accuracy of our shape classification system is influenced by several key factors.

1) **Orientation:** The orientation of shapes relative to the image plane significantly impacts classification accuracy. When shapes are rotated such that their edges become diagonal relative to the pixel grid, their measured perimeter decreases compared to shapes aligned with the image axes. This occurs because diagonal edges create fewer pixel transitions than horizontal or vertical edges, reducing the measured perimeter while the area remains constant. Consequently, the circularity

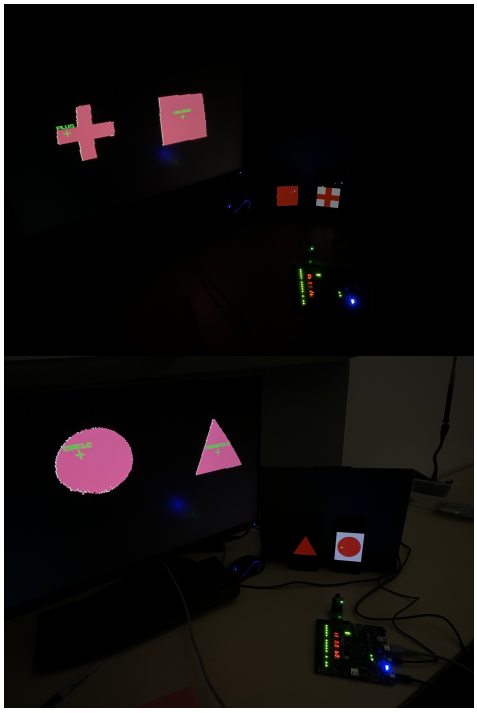

Fig. 3: Results of testing on multi-shape detection.

metric increases for rotated shapes, potentially leading to misclassification.

2) **Distance from camera:** Objects positioned closer to the camera yield more reliable classification results due to improved spatial resolution. This is because more precise area and perimeter measurements due to reduced quantization error, better definition of shape boundaries, reducing edge detection ambiguity, and improved separation between adjacent shapes, minimizing the risk of merged component labeling.
3) **Circularity thresholds:** The system's classification accuracy was heavily dependent on carefully calibrated circularity thresholds. These thresholds define the decision boundaries between different shape categories based on their calculated circularity values. Through iterative testing, we determined optimal threshold values that maximize classification accuracy across our test dataset. However, this manual threshold calibration process highlights a limitation in the system's adaptability to varying environmental conditions.

B. Resource Usage

We use 40 RAMB36s (2x for the camera input mask frame buffer, 2x for each of the 2 shape classifier sprites, 8x for each of the 2 Moore's input masks, 2x for each of 2 label masks for display, 14x for CCL) and 1 RAMB18 for registers. By efficiently routing BRAMs and removing extraneous features, we are well below the limit for available memory usage, even with two shape classifiers.

C. Latency

Our design successfully meets timing with

$$\text{WNS} = +0.131\text{ns}$$

. In addition, to meet visual latency for the user, we required that our design could complete all of the shape classifications within a single frame, or within 0.033s at 30fps. With a clock at 10ns, this means our design must complete all computation in 3 million cycles. We assumed that 3 million clock cycles would be more than enough time for our design to classify shapes based on the following analysis; since each of CCL and Moore's runs in non-deterministic time (CCL requires divisions to find centers of mass while Moore's traces the perimeter of the masked object, which could be different length frame-to-frame), we decided to analyze them based on algorithmic complexity:

- CCL: for n total pixels on the screen and a hard-coded maximum number of labels $m < n$, CCL first stores the frame over n pixels, then does a first pass over n pixels, then loops to resolve m equivalences m times, then calculates properties for m labels, then stores the m labels into proper arrays, then prunes over m labels, then stores the n pixels into the `top_level` BRAM, then outputs. Thus, CCL takes $O(n + m^2)$ linear time on the size of the screen and quadratic time on the maximum number of labels. Since we have $n = 320 \cdot 180$ and we hard-code $m = 64$, we assume the runtime of this module does not blow up.
- Moore's: for n total pixels on the screen, Moore's first scans through all n pixels to find the start point, then traces the perimeter (at worst it could scan a spiral of length $n/2$), then outputs. Thus, Moore's takes $O(n)$ linear time on the size of the screen. Since we have $n = 320 \cdot 180$, we assume the runtime of this module does not blow up.

All other parts of the algorithm have comparatively insignificant runtimes. Additionally, this runtime is not dependent on the number of shapes, since all shapes can be classified in parallel due to the FPGA's parallel processing capabilities. Finally, we recognize that even if CCL and Moore's together did exceed the 3 million clock cycle limit, then we would simply need to skip the frame. In the end, we find that the latency of our algorithm does not exceed the functional maximum, as desired.

D. Meeting Goals

We successfully met both our minimum and stretch goals posed in the original design review presentation. Our original goal was to implement rudimentary noise canceling and then use area computation, Moore's, and circularity to classify just a single shape, which we achieved by our preliminary report. However, through the implementation of CCL and routing Moore's to read from BRAM in `top_level`, we effectively parameterize the number of shapes that can be classified at once, limited only by the memory capabilities

of the hardware (not latency, as the FPGA can classify in parallel). Since ObNoDog is able to classify multiple shapes at once with individual shape identification, sophisticated noise pruning, and efficiently utilized BRAMs, we successfully met our stretch goal of classifying a variable number of multiple shapes.

VI. Discussion

A. Challenges

Several challenges emerged through development.

Firstly, memory proved to be a scarce resource. With the current implementation, CCL requires BRAM to store a mask of the frame alongside the label of each pixel, Moore's reads from the BRAM masks in `top_level` in order to compute neighbors, additional masks are required to output individual blobs to the display, and more. To get our model within the BRAM limit, we had to remove extraneous features (e.g. the initial frame buffer of camera input), alongside trade less memory usage for more latency in some modules (e.g. we increased CCL latency from an original design by 4x to reduce the amount of memory used by 4x).

Secondly, the reliance on circularity alone for classification makes it difficult to distinguish between shapes with similar values. For instance, a rounded square may be misclassified as a circle, and irregularities in the boundary can distort circularity values. This limitation becomes particularly evident in noisy or imperfectly segmented images, where boundary artifacts skew the calculated metrics. By pruning blobs under a certain area (single pixel noise), CCL reduces the likelihood that shapes are misclassified, though jagged shape edges will still likely overinflate the perimeter calculation.

Thirdly, debugging and test-benching CCL proved to be much more challenging and time-consuming that we originally anticipated. In part, this is due to having to reconfigure BRAM usage during development multiple times (CCL stores pixel labels into BRAM, a large sink for our original BRAM budget). Another reason was that CCL has a large number of states in the FSM, and since a number of these states needed to interact with `top_level` in intelligent ways (e.g. writing to `top_level` BRAMs), a large amount of test-benching was required to ensure robust outputs.

B. Future Work

To address the identified challenges and expand the system's capabilities, future work will focus on the following areas:

Latency: Optimizing the finite state machine logic and memory access patterns will help reduce processing time. For example, instead of writing outputs from the CCL module into separate BRAMs and reloading them in the Moore's Neighbor Tracing module, the design will share BRAMs across modules by instantiating them in the `top_level`. This eliminates unnecessary read/write cycles and minimizes memory access delays. Implementing pipelined architectures can allow multiple steps of the processing pipeline to operate concurrently, significantly improving throughput. Parallelizing computationally intensive tasks, such as perimeter calculation and equivalence resolution in the CCL module, can further reduce latency.

Improved shape classification: The current classification system will be expanded by modifying Moore's to output an array of boundary pixel coordinates. This additional data will enable the calculation of properties like number of vertices, to distinguish between polygons with similar circularity, and edge detection, which may provide metrics like aspect ratio or edge uniformity. While vertex and edge detection introduce their own set of challenges in boundary noise and orientation considerations, incorporating these features alongside circularity will create a more nuanced multi-dimensional classification metric, improving accuracy and robustness.

n-shape detection: The current system is designed to trace and classify up to two shapes at a time. This is due to limits in the FPGA's available BRAM. However, the current code can be extended to any number of shapes, since all shapes are classified in parallel. On the other hand, storing all pixel values and masks DRAM instead of BRAM would immediately loosen the restriction on memory, allowing storage of many more shapes at the potential expense of latency.

VII. Acknowledgments

We would like to express our heartfelt gratitude to our TA, Jan, for her unwavering support and assistance, especially during the critical days leading up to the deadline. Her guidance in structuring our system, debugging issues, and identifying design flaws that we had overlooked was invaluable.

Additionally, we extend our thanks to Prof. Joe Steinmeyer for delivering an exceptional class that enriched our learning experience.

VIII. Appendix

GitHub Repository: https://github.com/jca0/ObNoDog-final

References

[1] Image Analysis - connected components labeling. (n.d.). https://homepages.inf.ed.ac.uk/rbf/HIPR2/label.htm

[2] Wikimedia Foundation. (2024, November 5). Connected-component labeling. Wikipedia. https://en.wikipedia.org/wiki/Connected-component_labeling

[3] Ghuneim, A. (n.d.). Contour tracing. https://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/moore.html

[4] Adjustable method based on body parts for improving the accuracy of 3D reconstruction in visually important body parts from silhouettes - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/An-example-of-Moore-Neighbor-boundary-tracing-found-1-8-pixels-as-a-boundary-12_fig3_372683288 [accessed 12 Dec 2024]

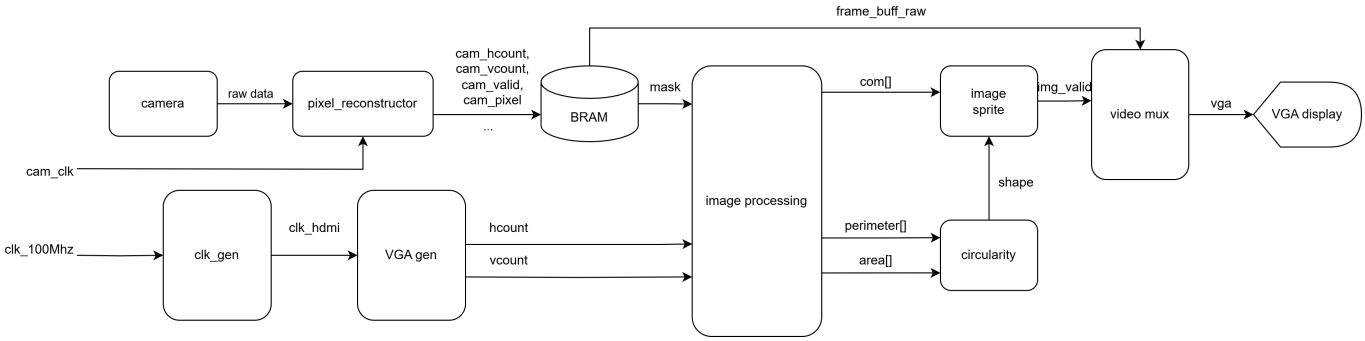

Fig. 4: Block diagram of top level module

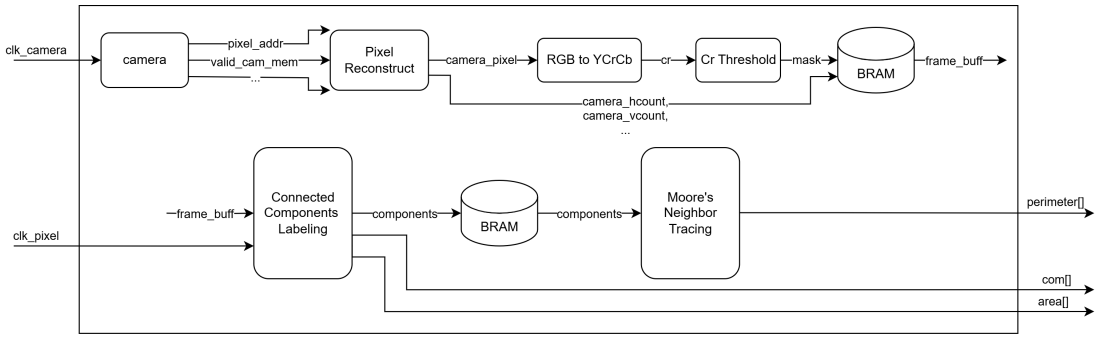

Fig. 5: Block diagram for image processing

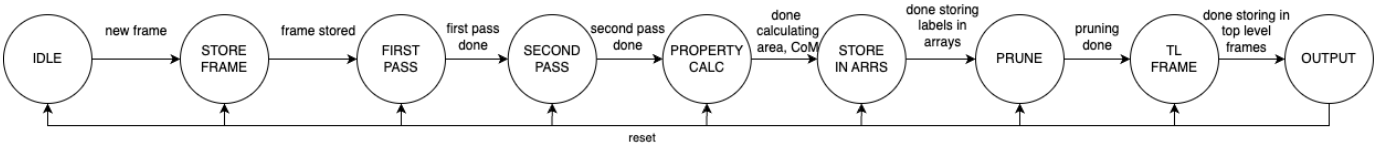

Fig. 6: FSM diagram for Connected Components Labeling module

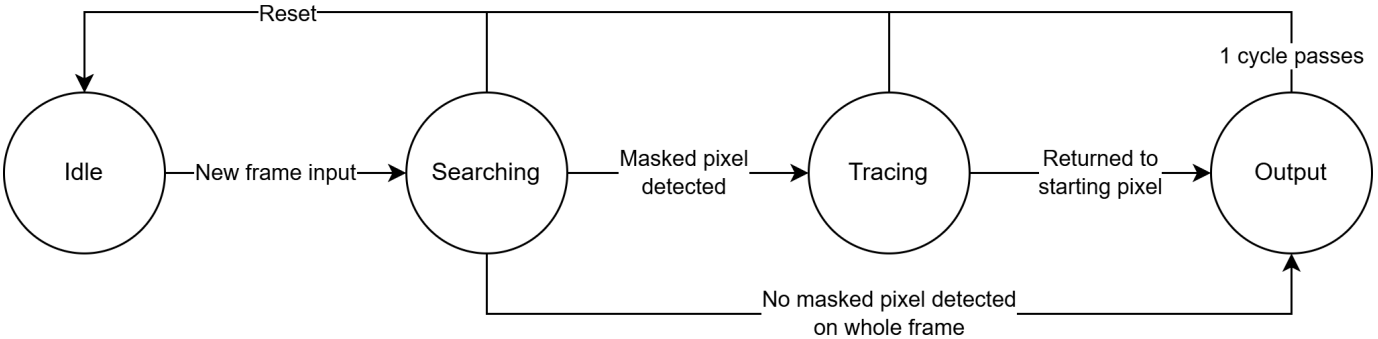

Fig. 7: FSM diagram for Moore's Neighbor Tracing module