5-2023

Prototyping A 3D Reconstruction of Fly Photoreceptors: From Traditional Computer Vision Techniques to 3D Visualization

Kunal Jain

PROTOTYPING A 3D RECONSTRUCTION OF FLY PHOTORECEPTORS: FROM TRADITIONAL COMPUTER VISION TECHNIQUES TO 3D VISUALIZATION

A Thesis Presented

by

KUNAL JAIN

Submitted to the Office of Graduate Studies,

University of Massachusetts Boston,

for the degree of

MASTER OF SCIENCE

May 2023

Computer Science Program

PROTOTYPING A 3D RECONSTRUCTION OF FLY PHOTORECEPTORS: FROM TRADITIONAL COMPUTER VISION TECHNIQUES TO 3D VISUALIZATION

A Thesis Presented

by

KUNAL JAIN

Approved as to style and content by:

Daniel Haehn, Assistant Professor
Chairperson of Committee

Dan Simovici, Professor
Member

Jens Rister, Associate Professor
Member

Marc Pomplun, Chair
Computer Science Department

Dan Simovici, Graduate Program Director
Computer Science Department

ABSTRACT

PROTOTYPING A 3D RECONSTRUCTION OF FLY PHOTORECEPTORS: FROM TRADITIONAL COMPUTER VISION TECHNIQUES TO 3D VISUALIZATION

May 2023

Kunal Jain, B.Tech., National Institute of Technology, Hamirpur
M.S., University of Massachusetts Boston

Directed by Professor Daniel Haehn

The Drosophila fruit fly is a well-established model organism for studying vision and neural perception. In this research, we focused on segmenting and analyzing photoreceptor cells in the Drosophila retina, specifically rhabdomeres. To facilitate our study, we acquired a comprehensive 3D dataset of Drosophila retina EM images and developed an automated segmentation pipeline using computer vision techniques. We evaluated the performance of our pipeline using automated metrics on Wild Type CS 1 and Nina D1 Mutant Drosophila datasets and prototyped a 3D model of segmented cells. The generated segmentation masks and 3D models of photoreceptor cells contribute to a better understanding of Drosophila vision and neural perception, and may have implications for further studies in developmental biology, and neurobiology.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

LIST OF TABLES

TABLES PAGE

CHAPTER 1

INTRODUCTION

Drosophila, or fruit flies, have compound eyes composed of many tiny facets called ommatidia, which are the main visual structure of an insect and are responsible for mosaic vision. Each ommatidium is divided into two parts: a light-gathering part consisting of a cornea and crystalline cones, and a light-detecting part.

In humans, the retina contains photoreceptor cells called rods and cones, while in Drosophila, the photoreceptor cells are called rhabdomeres. Rhabdomeres contain pigments that absorb light and trigger a neural response. The structure of rhabdomeres in Drosophila forms a polygon pattern in open structure and fuses in the center in close structure, with flies typically having a hexagonal open rhabdomere structure. Drosophila has a neuronal superposition eye (Langen et al., 2015), as shown on in figure 1c on page 3; the rhabdomeres of one ommatidium point at different points in space, but the red highlighted

ones in neighboring ommatidia point at the same point.

Both Drosophila and humans have a neural pathway that sends visual information to the brain for processing. In Drosophila, this pathway involves lamina neurons, which are connected to the photoreceptors and send information about the visual scene to higher brain centers. In humans, the optic nerve carries information from the retina to the brain.

While Drosophila eyes do not have a complex structure like the human eye's iris and pupil for focusing, they do have mechanisms for adjusting the sensitivity of their photoreceptor cells to different levels of light. Drosophila eyes are smaller and simpler than human eyes, but they are still valuable for studying the principles of vision and the neural basis of perception due to their relatively simple nervous system and genetic tractability.

The study of 3D models of Drosophila photoreceptors can provide valuable insights into the structure and function of these cells and their role in vision. Drosophila, or fruit flies, are widely used as model organisms in the field of neurobiology and have been extensively studied due to their relatively simple nervous system and genetic tractability.

One of the key benefits of studying the 3D model of Drosophila photoreceptors is gaining a better understanding of their structure and function. The 3D model can reveal the intricate details of the shape, organization, and arrangement of photoreceptor cells in the eye. By visualizing the photoreceptors in three dimensions, researchers can study their morphology, such as the shape and size of the cells, the distribution of pigments, and the arrangement of cellular components. This information can help researchers decipher how these cells are

specialized for their role in vision and how they interact with each other to process visual information.

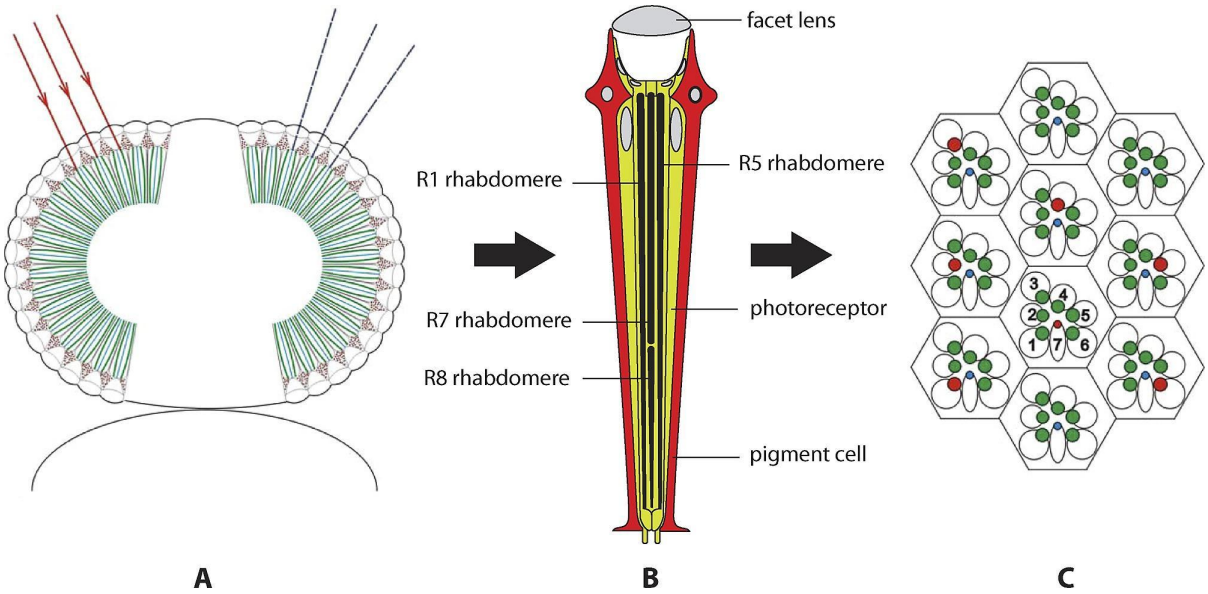

Figure 1: Diagrams of eye and retinal organization of the Fruit Fly Drosophila (https://doi.org/10.1371/journal.pbio.0060115.g001)

Furthermore, the 3D model of Drosophila photoreceptors can provide insights into the neural connectivity of these cells. Photoreceptors are just one part of the visual pathway in Drosophila, and their signals are processed by downstream neurons, such as lamina neurons, before being transmitted to higher brain centers. By visualizing the connections between photoreceptors and other neurons in the visual pathway, researchers can gain a better understanding of the neural circuitry involved in vision in Drosophila. This can help uncover the complex network of interactions between different types of neurons and how

they work together to process visual information.

Moreover, studying the 3D model of Drosophila photoreceptors can also shed light on the effects of genetics and environment on these cells. Fruit flies are widely used in genetic studies due to their well-characterized genome and the availability of various genetic tools. By comparing the 3D models of photoreceptors from different genetic backgrounds, researchers can investigate the effects of specific genes on the structure and function of these cells. Additionally, the 3D models can be used to study how environmental factors, such as light exposure or temperature, may impact the morphology and organization of photoreceptor cells.

1.1 IMAGE SEGMENTATION

In computer vision, an image is a digital representation of a visual scene or object. It is typically composed of pixels, which are tiny dots that contain information about color, intensity, and other visual attributes. Images can be captured from various sources, such as cameras or generated by computer graphics.

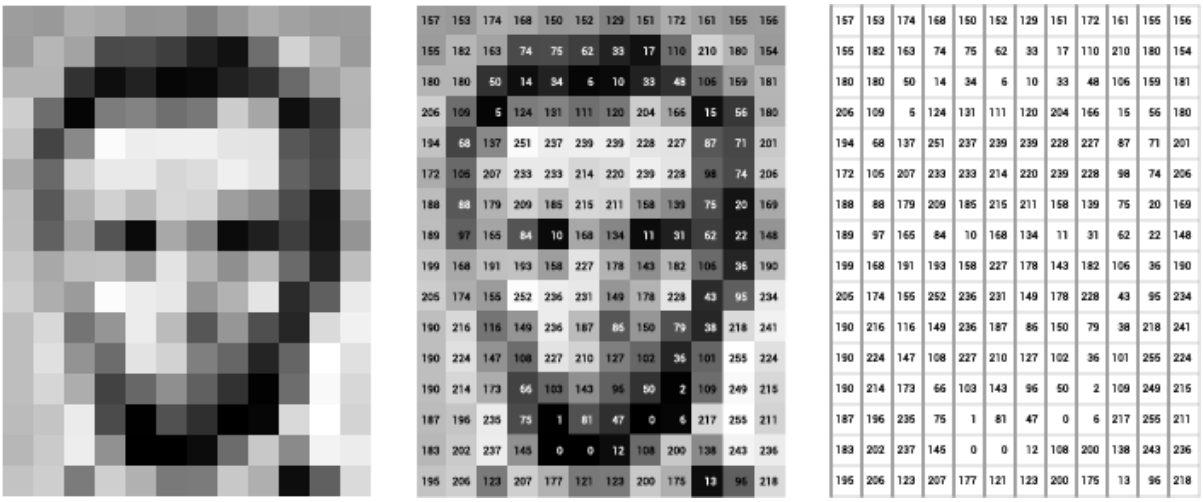

Figure 2: A widely used image of Abraham Lincoln represented as a matrix of pixels(left), pixels with intensity values(center), and matrix of pixel intensities(right) (https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html)

Image segmentation enables machines to understand and interpret the contents of an image at a more detailed level. It has applications in a wide range of fields, such as medical imaging, autonomous driving, robotics, and video surveillance, among others

Image segmentation is the process of dividing an image into meaningful and visually distinct regions or segments. The goal is to identify and isolate different objects or regions of interest within an image. These segments can be used for further analysis, such as object recognition, object tracking, or image understanding..

Image segmentation is widely used in medical imaging, such as in CT scans, MRIs, and ultrasound images. By segmenting the image, doctors can identify and isolate specific regions of interest, such as tumors, blood vessels, or organs, which can aid in diagnosis,

treatment planning, and monitoring. Image segmentation plays a crucial role in object recognition, where the goal is to identify and localize objects within an image. By segmenting an image into different parts or regions, it becomes easier to identify and differentiate different objects within the image,

Using traditional computer vision techniques or training a CNN are popular ways of achieving image segmentation, and the choice of technique depends on the specific application and the properties of the image being segmented. Traditional computer vision techniques and convolutional neural networks (CNNs) have different strengths and weaknesses when it comes to image segmentation.

Traditional computer vision techniques, such as thresholding, edge detection, and region growing, are relatively simple and computationally efficient, making them useful for many applications. They can work well for images with simple or well-defined objects or regions, and they are often easy to interpret and understand. However, they may not be as effective for complex or ambiguous images, where the segmentation boundaries are unclear, or where objects overlap or have similar textures or colors.

CNNs, on the other hand, are much more powerful and flexible than traditional computer vision techniques. They can learn complex patterns and relationships in the data, making them better suited for tasks such as image segmentation. CNNs can also handle noisy or ambiguous images and can adapt to a wide range of data distributions, making them more generalizable across different domains. However, CNNs require large amounts of labeled

data and significant computational resources for training, and they may not be as interpretable or transparent as traditional computer vision techniques.

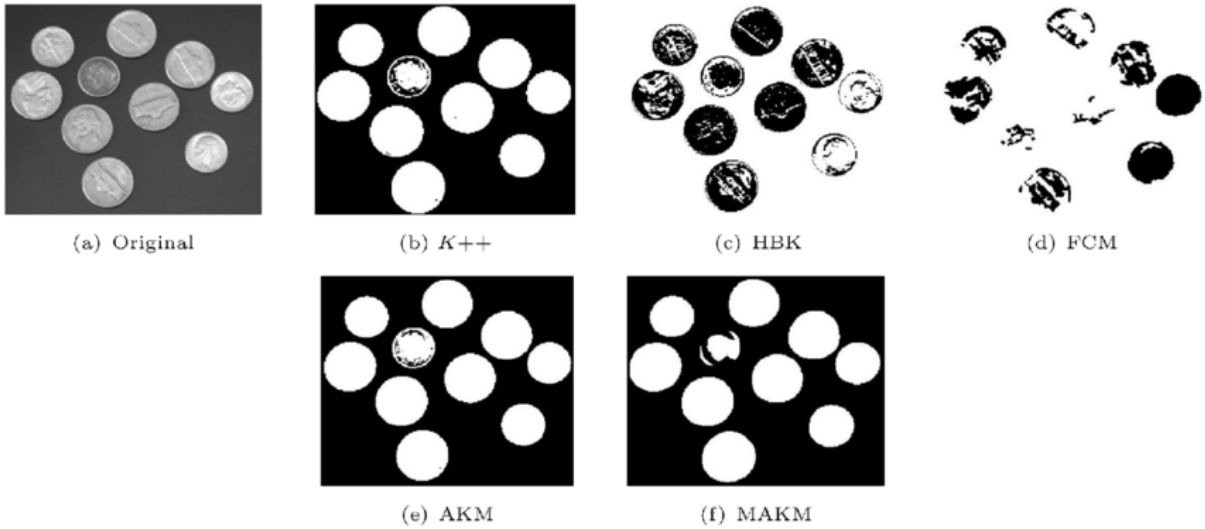

Figure 3: Coin-labeled image segmentation using various approaches (https://doi.org/10.1007/s41095-019-0151-2)

1.2 TRADITIONAL COMPUTER VISION

Traditional computer vision techniques refer to methods and algorithms that process and analyze images using mathematical operations and statistics without the use of machine learning. Some examples of traditional computer vision techniques used for image processing and analysis:

1. Thresholding: This technique involves selecting a threshold value to segment an image into binary regions based on the intensity of pixels. It is often used to extract objects of interest from the background or to separate objects with different intensities.

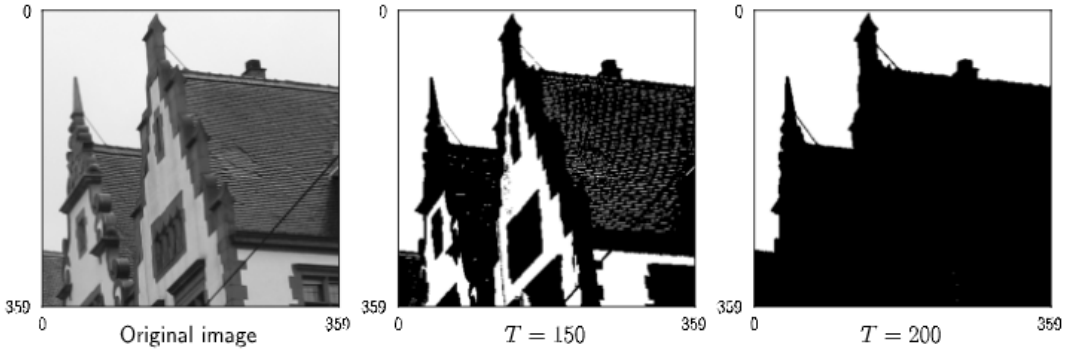

Figure 4: Example of binarization for two thresholds.
(https://vincmazet.github.io/bip/segmentation/histogram.html)

2. Edge detection: This technique involves finding the boundaries of objects in an image by detecting sudden changes in intensity. Edge detection is used for feature extraction

and object recognition.

Figure 5: Example of edge detection using canny edge detector
(http://vision.cs.arizona.edu/nvs/research/image_analysis/edge.html)

3. Morphological operations: This technique involves using mathematical operations to manipulate and analyze the shape and structure of objects in an image. It is often used for tasks such as noise removal and object segmentation.

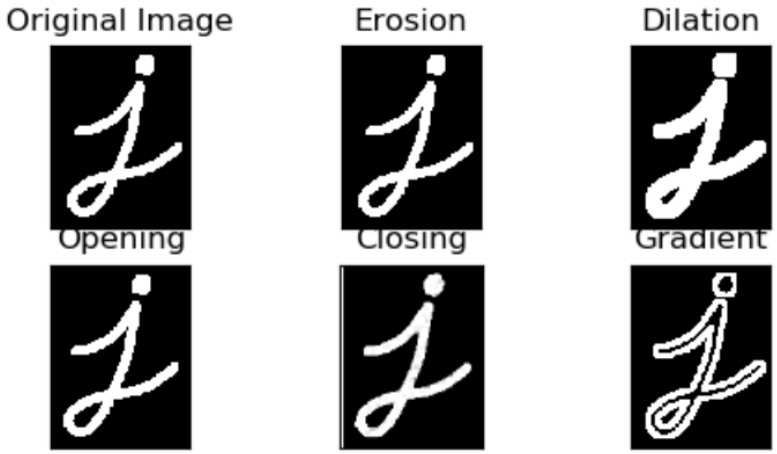

Figure 6: Example showing morphological operations applied on an image
(https://buffml.com/morphological-operations/)

CHAPTER 2

EXISTING RELATED WORK

The Drosophila visual system is a well-studied model system in neuroscience due to the high degree of genetic conservation between flies and mammals and the relatively simple and well-defined structure of the fly visual system.

Photoreceptors are sensory neurons that detect light and transform it into electrical signals, which then get transmitted to the brain. Rhabdomeres are small, light-sensitive structures that extend from the photoreceptor cell membrane and contain specialized proteins called opsins.

In some insects, such as flies, compound eyes have two categories of photoreceptors and rhabdomeres that differ in morphology and location. The outer or peripheral category consists of photoreceptors R1 to R6 and their rhabdomeres, which are large, cylindrical and take up most of the ommatidial facet. The inner or central category consists of photoreceptors R7 and R8 and their rhabdomeres, which are small, and located in the center of the ommatidium. Photoreceptors R7 and R8 are arranged in tandem and so a

cross-section only shows one photoreceptor.

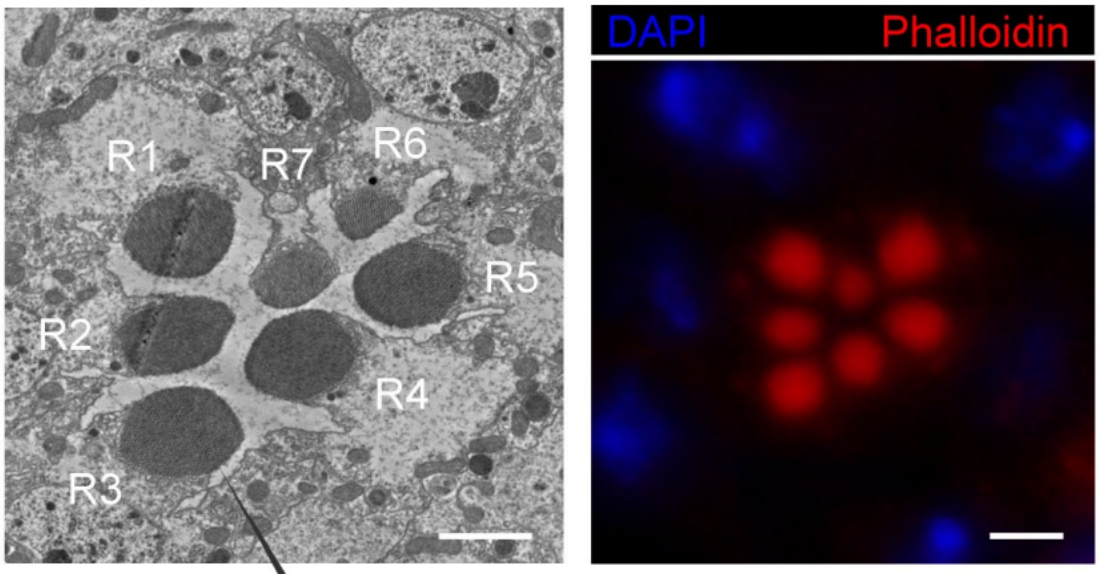

Figure 7: Structure and arrangement of rhabdomeres from Drosophila photoreceptor cells. EM Image of a cross section through photoreceptor cells(left), and light microscopic fluorescence image of an ommatidial cross section (Schopf et al., 2019)

When seen under electron microscope, the compound eye of Drosophila is made up of about 800 ommatidia. Each ommatidium has its own lens that can be seen on the surface of the eye, and this lens focuses light onto the photoreceptor cells beneath it. Each ommatidium contains eight photoreceptor neurons that are located under the lens. These photoreceptors have specialized structures called rhabdomeres, which are loaded with Rhodopsin proteins that detect photons. The rhabdomeres of the 'outer' photoreceptors (R1-R6) span the entire length of the ommatidium and are arranged in a trapezoid shape. On

the other hand, the rhabdomeres of the 'inner' photoreceptors (R7 and R8) are located above each other, with R7 being farther from the lens compared to R8, and they share a common light path facing the same point in space. The rhabdomere of the 'inner' photoreceptors is positioned in the center of the trapezoid formed by the 'outer' photoreceptors (Rister et al., 2013)

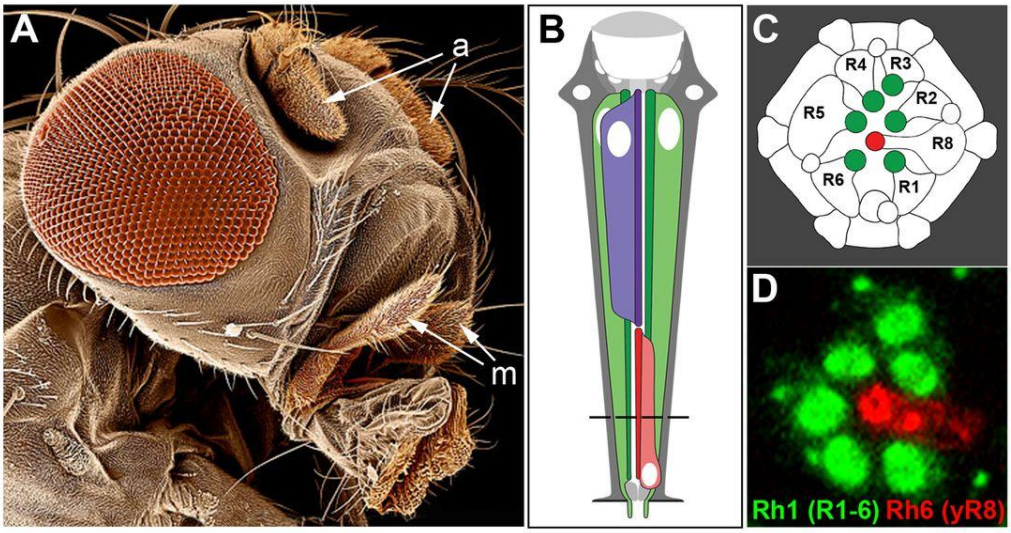

Figure 8: (A)Scanning electron microscope image of Drosophila head. (B) Schematic side view of an ommatidium. (C) Schematic of a section through an ommatidium at the level indicated by the dashed line in B. (D) Image corresponding to schematic in C showing R1-R6 photoreceptors in green and R8 photoreceptors in red. (Rister et al., 2013)

It is seen that Vitamin A deprivation affects Drosophila photoreceptor structure and Rhodopsin expression (Dewett et al., 2021). Since vitamin A is essential for the formation of visual pigments, the absence of vitamin A leads to a complete lack of visual pigments in the rhabdomere membranes. This absence of visual pigments in the rhabdomeres of the fly

causes severe defects in the size and structure of the rhabdomeres.

In normal or Vitamin A replete fly eyes, the visual pigments in the rhabdomeres are responsible for detecting and responding to light, allowing the flies to see and perceive their visual environment. However, in a Vitamin A deprived fly eye, the absence of visual pigments due to the lack of vitamin A results in impaired visual function and severe abnormalities in the size and shape of the rhabdomeres.

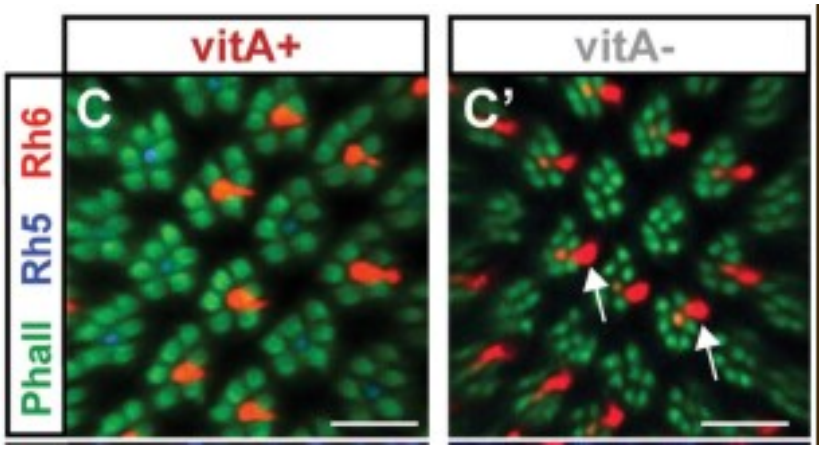

Figure 9: Vitamin A replete wild-type adult eye on left and Vitamin A deprived mutant fly eye on the right. The photoreceptors in Vitamin A deprived fly are very small in shape and size as compared to that of Vitamin A replete fly. (Dewett et al., 2021)

CHAPTER 3

DATA EXPLORATION AND VISUALIZATION

In this thesis research we used the flyEM photoreceptor datasets we received from researchers at Indiana University. Researchers at Indiana University have used serial block-face electron microscopy to image the photoreceptors of Drosophila. These images provide detailed information about the structure and organization of photoreceptors at the cellular level, including their shape, size, and ultrastructure. The electron microscope used in SBF-EM employs a focused electron beam to create images of samples at high resolution and can be used to visualize the ultrastructure of cells and tissues in three dimensions. It is possible to use electron microscopy to image photoreceptors, the cells in the eye that detect light and play a crucial role in vision, in a variety of organisms, including flies. The goal of using serial block-face EM technique is to obtain a 3D nanoscale ultrastructure so that the data can be analyzed in 3D rather than in isolated sections(He et al., 2018).

It is evident from the dataset that the photoreceptors are organized into a regular, hexagonal array. These hexagonal structures are called the rhabdomeres and these are the specialized input regions in Drosophila. Along with photoreceptors, images also contain numerous organelles, including mitochondria, endoplasmic reticulum, and Golgi apparatus, which are involved in various cellular processes such as energy production and protein synthesis.

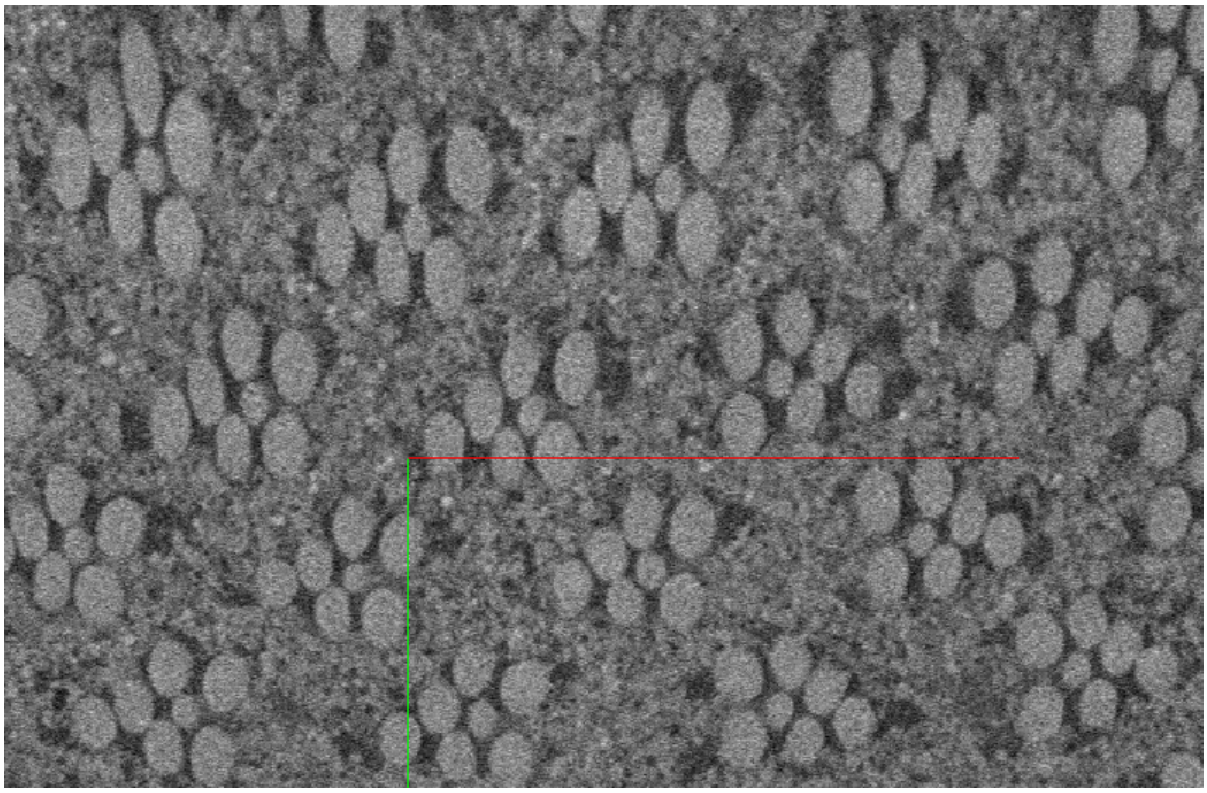

Figure 10: An image from flyEM dataset showing rhabdomeres structures. The image is from the Wild Type CS1 Overview Dataset

We received EM image datasets for four types of Drosophila. For each fly type, we have an overview tileset and data tileset. Data tileset has multiple stacks with different voxel offsets. All datasets have a very high resolution of 6144 px X 4096 px with Overview tileset of wild type CS 2 being an exception, which has a resolution of 3072 px X 2048 px. The voxel size of each dataset is 10 X 10 X 40 nm^3. In total the datasets take up 1038 GBs on disk.

All datasets are available on https://mpsych.org/rister/ where we have deployed it using Google's Neuroglancer software.

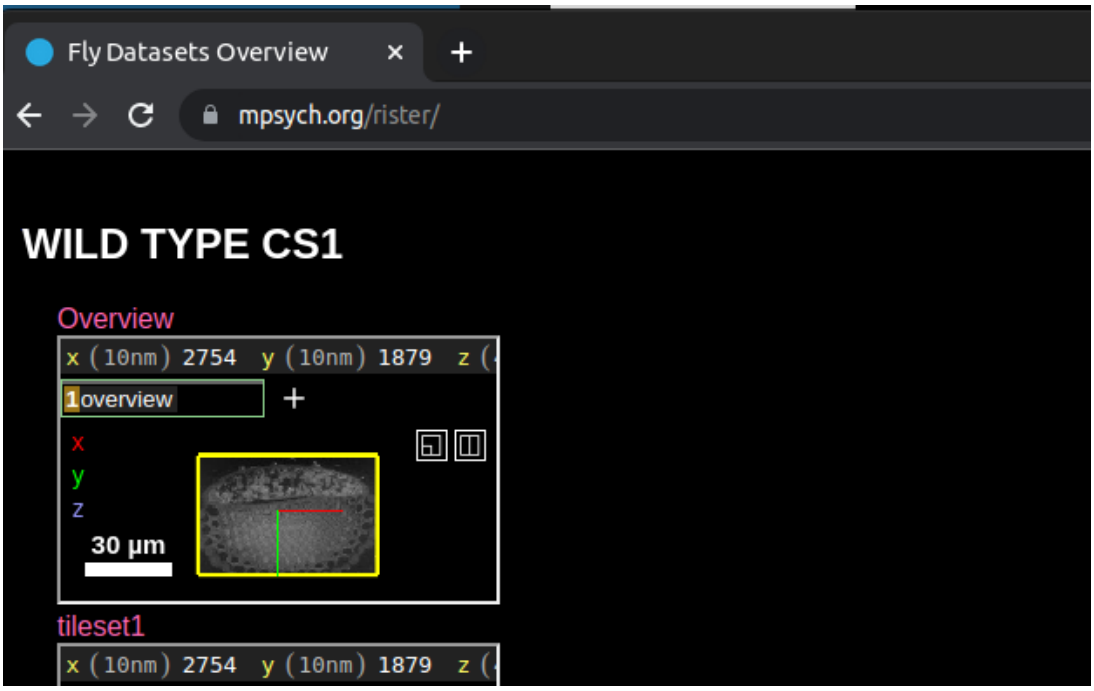

Figure 11: All datasets are deployed on https://mpsych.org/rister

Fly type	Dataset	Resolution	Number of slices	Dataset size
Wild Type CS 1	VSOverviewTileSet	6144 X 4096	862	43.4 GB
	VSDataTileSet1	6144 X 4096	3444	173.5 GB
	VSDataTileSet2	6144 X 4096	862	43.4 GB
Wild Type CS 2	VSOverviewTileSet	3072 X 2048	1268	16 GB
	NewData1	6144 X 4096	1268	63.8 GB
	NewData2	6144 X 4096	1268	63.8 GB
	NewData3	6144 X 4096	1268	63.8 GB
Nina D1 MUTANT	VSOverviewTileSet	6144 X 4096	2168	109.2 GB
	VSDataTileSet1	6144 X 4096	4	1.2 GB
	VSDataTileSet2	6144 X 4096	2168	327.1 GB
MPSDel	VSOverviewTileSet	6144 X 4096	1600	80.5GB
	VSDataTileSet	6144 X 4096	1600	322.3 GB
Total				**1308 GB**

Table 1: Table displaying information about all the datasets received from Indiana university. The received datasets are of four different types of Drosophila. Each dataset has Overview tileset and Data tilesite. All datasets have 6144 X 4096 pixel resolution with the exception of Wild Type CS 2 Overview tileset which has resolution of 3072 X 2048 pixels

3.1. DATASET CONVERSION FOR NEUROGLANCER WEB VISUALIZATION

The conversion of given raw data into neuroglancer-compatible data consists of 4 steps -

1. Copying the images from the input folder into a temporary folder for safe conversion and not tampering with the original data

```
only_images = tempfile.TemporaryDirectory()
only_images_path = only_images.name
print("Path of Temporary folder one is {}".format(only_images_path))
```

2. Converting the copied image files, which are in TIFF format, into JPEG using the Imagemagick tool

```
images = [f for f in sorted(os.listdir(only_images_path)) if '.tif' in f.lower()]

for i in images:
    img_input_path = only_images_path + "/" + i
    img_output_path = jpeg_path + "/" + i.replace("tif", "jpeg")
```

3. Read the info from .info of the image file to generate Neuroglancer JSON File.

```
with open(path + "/" + images[0]) as f:
    lines = f.readlines()
    da = [i for i in lines if "pixelsize" in i.lower()]
    la = [i for i in lines if "tif" in i.lower()]
    ofs = [i for i in lines if "offset" in i.lower()]
    offset = ofs[0].split()
    offset_a = offset[1]
    offset_b = offset[2]
    offset_c = 0
```

```
pixel_size = da[0].split()
pixel_x = pixel_size[1]
pixel_y = pixel_size[2]

z_axis = la[0].split()
z_value = float(z_axis[1])
f.close()
```

4. After reading the required data from .info files, we align the data with the standard Neuroglancer JSON format before starting the conversion.

```
counter = [f for f in sorted(os.listdir(jpeg_path)) if '.jpeg' in f.lower()]

slices = len(counter)
image = mahotas.imread(jpeg_path + "/" + counter[0])
height = image.shape[0]
width = image.shape[1]

if image.ndim == 2:
    channels = 1  # grayscale

if image.ndim == 3:
    channels = image.shape[-1]
```

The Neuroglancer JSON file includes information about the volume data, such as the file format and location of the data, the dimensions and resolution of the volume, and any additional metadata associated with the data. It also includes information about how the data should be displayed and interacted with in Neuroglancer, such as the default views and layers and any custom annotations or labels that will be added to the data.

```
# JSON FILE
array_out = {
    "type": "image",
    "data_type": str(image.dtype),
    "num_channels": channels,
    "scales": [
        {
            "chunk_sizes": [],
            "encoding": "jpeg",
            "key": "full",
            "resolution": [int(pixel_x), int(pixel_y), volume_data],
            "size": [width, height, slices],
            "voxel_offset": [int(offset_a), int(offset_b), int(offset_c)]
        }
    ]

}
```

Figure 12: Neuroglancer json file format

The Neuroglancer JSON file helps to load and display the volume data in Neuroglancer. Neuroglancer allows users to view and explore the data in three dimensions and to interact with and annotate the data using Neuroglancer's various tools and features.

The Neuroglancer script then reads the images and JSON files and converts them into a neuroglancer readable stack. A Neuroglancer stack is a collection of volumetric data that users can visualize and interact with in Neuroglancer. Neuroglancer supports several different data formats, including the Multiscale Volume Format (MVF), specifically designed for efficient visualization and exploration of large volumetric data sets. To convert

our data into an MVF stack that Neuroglancer can read, we use the Neuroglancer Precomputed Python library. This library provides tools for creating and manipulating MVF stacks, as well as tools for visualizing the data in Neuroglancer.

```
os.system("generate-scales-info {}/data.json {}".format(json_path, OUTFOLDER))
os.system("slices-to-precomputed --flat --input-orientation RPS {} {}".format(jpeg_path, OUTFOLDER))
os.system("compute-scales --flat {}".format(OUTFOLDER))
```

Figure 13: Code to run the neuroglancer conversion script

The Multiscale Volume Format (MVF) is a file format developed specifically for efficient visualization and exploration of large volumetric data sets in Neuroglancer. MVF files are typically used to store and transfer volumetric data too large to be stored in memory or transmitted over the network simultaneously. MVF files consist of a hierarchy of scale levels, each representing a progressively downsampled data version. The highest scale level represents the full resolution of the data, while the lower scale levels represent progressively lower resolutions. Neuroglancer can access and visualize the data in an MVF file by loading only the scale levels needed to display the data at the desired resolution.

Figure 14: MVF files generated after a successful conversion

CHAPTER 4

TRADITIONAL COMPUTER VISION TECHNIQUE PIPELINE

Traditional computer vision techniques seek to equip computers with the ability to interpret, analyze, and understand images much like humans do. This involves the use of a range of techniques for tasks such as segmentation, object detection, and feature extraction from images. Images themselves are stored as an array of numerical values, with each number representing the color intensity value at a specific pixel. Through various manipulations of these arrays, computer vision techniques can extract meaningful information from images. For segmenting photoreceptors from the fly retina EM image, we started with preprocessing the data to segment photoreceptor cells in the fly EM images. Depending on the quality of the images, we applied various preprocessing techniques. The preprocessing techniques gave us both good and bad results for segmentation. To get a concrete outcome of how accurate the pipeline, we calculated the accuracy of the created mask. We then saved the

most accurate mask and finally determined if the mask is good based on the number of ommatidia present in the mask.

The source code of the pipeline follows the Model View Controller design pattern, and we use jupyter notebooks as drivers for the pipeline. The model represents the segmentation logic and the view represents the logic for display and saving of various image plots. All techniques used in the pipeline are encapsulated in functions present in the utility function file. Finally the controller acts as an intermediary between the driver notebook and the model and the view component.

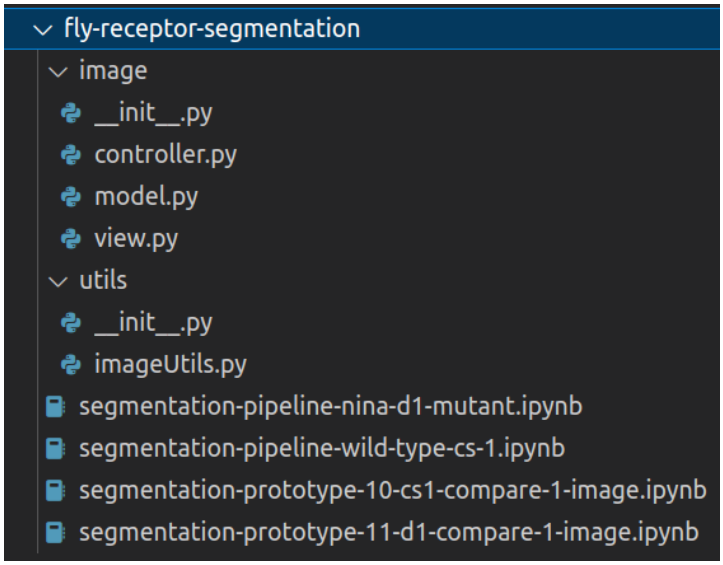

Figure 15: The code for segmentation pipeline structured as an MVC model.
Before running the pipeline, we need to set up logging, directory paths, and few dataset dependent variables.

```
data_dir = '/raid/mpsych/RISTERLAB/fly_segmentation_experiments/data_dir/wild_type_raw/'
save_dir = '/raid/mpsych/RISTERLAB/fly_segmentation_experiments/data_dir/04-14/wild_type_cs_1/binary_mask_9/'
```

```
LOGGING_IN_NOTEBOOK = True
log_path = save_dir + 'log.txt'
crop_path = save_dir + 'original_crop/'
binary_mask_path = save_dir + 'binary_mask/'
compare_plot_path = save_dir + 'compare_plots/'
compare_labeled_plot_path = save_dir + 'compare_labeled_plots/'
```

Figure 16: Code snippet for setting up files. data_dir contains all raw files from the dataset. save_dir is a directory where all output images are saved. The crop of the original image is saved in crop_path. A comparison plot comparing original image, intermediate image and segmented mask image is saved in compare_labeled_plot_path. A comparison plot of original image and segmented mask image is saved in compare_polt_path. The final segmented mask is saved in binary_mask_path.

```
# NOTE: one single ommatidium has 7 rhabdomeres.
# Value NUM_RHABDOMERE and values related to threshold are not to be changed
NUM_RHABDOMERE = 7

THRESHOLD_BRACKET = 40
LEAST_POSSIBLE_THRESHOLD = 1
MOST_POSSIBLE_THRESHOLD = 255

# Change these values as per different datasets
# Values for Wild_Type_CS_1
#     REMOVE_SMALL_REGION = True
#     REGION_SIZE = 750
#     LEAST_PIXEL_COUNT = 20000
#     LEAST_OMMATIDIA_COUNT = 1
# Values for NINA_D1_MUTANT
#     CROP_SIZE = (512, 512)
#     REMOVE_SMALL_REGION = False
#     REGION_SIZE = 750
#     LEAST_PIXEL_COUNT = 20000
#     LEAST_OMMATIDIA_COUNT = 12
CROP_SIZE = (512, 512)
REMOVE_SMALL_REGION = True
REGION_SIZE = 750
LEAST_PIXEL_COUNT = 20000
LEAST_OMMATIDIA_COUNT = 1
```

Figure 17: Code snippet for setting up image parameters. Since one ommatidium contains seven rhabdomeres, NUM_RHABDOMERE is set as seven. THRESHOLD_BRACKET is set for varying threshold values. Other parameters are dataset dependent and are chosen carefully for both Wild Type CS 1 Overview dataset and Nina D1 Mutant Overview dataset

Once the environment is set up we then run all the raw images that we have through the pipeline. The steps of the pipeline are as follows -

1. Load file from the image file path - To load an image from a given file path we used the imread function of the mahotas python library. The imread function loads an image as a numpy array.

```
data = ImageController.read(file_path)
ImageController.display(data)
```

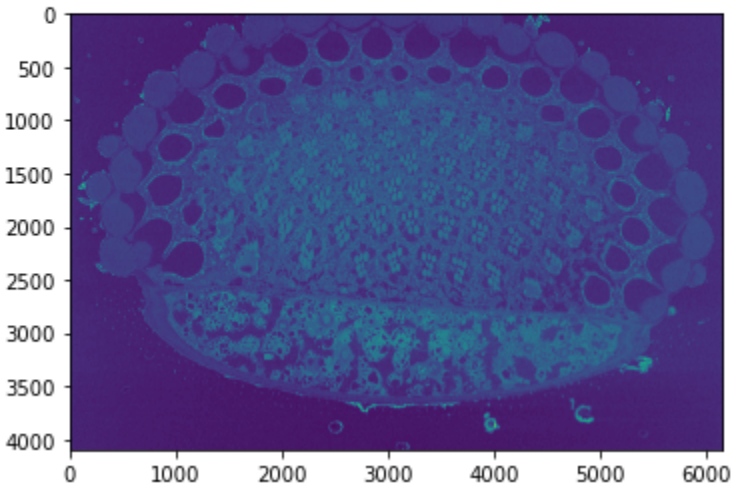

Figure 18: Code to load one image from the dataset. The above image is loaded from Wild Type CS 1 Overview dataset. The image resolution is 6144 X 4096 pixels

2. Normalize and crop image - Image normalization is a technique used to adjust the pixel values of an image so that they fall within a specific range. This can help to improve the performance of image processing algorithms by reducing the effects of

lighting and contrast variations. We used MinMax normalization in the pipeline. In MinMax normalization, the minimum and maximum pixel values of the image are determined, and the pixel values are scaled to fall between a specified minimum and maximum value. Once we have normalized the image, we then crop the image as per the size mentioned before starting the pipeline. We performed center cropping on the image, in which the center region of the image is cropped and saved for further processing. We achieved cropping using numpy array slicing.

```
crop_data = ImageController.center_crop(normalized_data)
ImageController.display(crop_data)
```

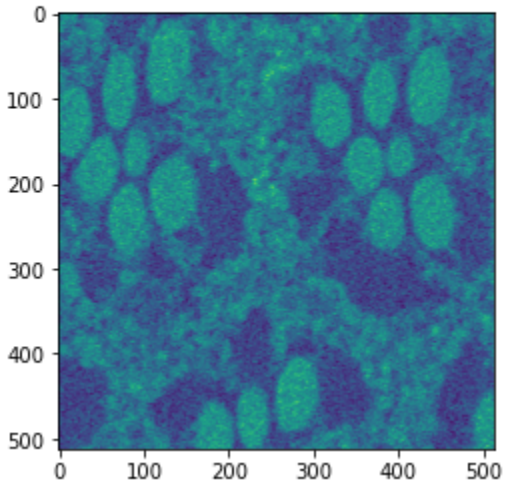

Figure 19: Code snippet for cropping of image. The original image is first normalized and then cropped to the resolution of 512 X 512 pixels.

```
@staticmethod
def crop_image(original_image, crop_size):
    # returns cropped image of crop size at the center of the image
    height, width = original_image.shape
    center_x, center_y = height//2, width//2
    new_height, new_width = crop_size
    start_x, start_y = center_x - (new_height//2), center_y - (new_width//2)
    end_x, end_y = center_x + (new_height//2), center_y + (new_width//2)
    result_image = original_image[start_x:end_x, start_y:end_y]
    return result_image
```

Figure 20: Code snippet of crop_image function. The function crops images from the center, irrespective of input image resolution.

3. Smoothen Image - A Gaussian filter is a filter commonly used to smooth out an image and reduce noise. It works by applying a Gaussian distribution to the values in the image, which effectively blurs the image and reduces the high-frequency components. A Gaussian filter is applied to an image by convolving the image with a 2D Gaussian kernel. This involves sliding the kernel over the image and computing the weighted sum of the pixel values in the kernel window at each location. The resulting value is the filtered value for the center pixel of the kernel window. We used the gaussian_filter function from mahotas library to smoothen the image.

```
smoothed_data = ImageController.smooth(crop_data)
ImageController.display(smoothed_data)
```

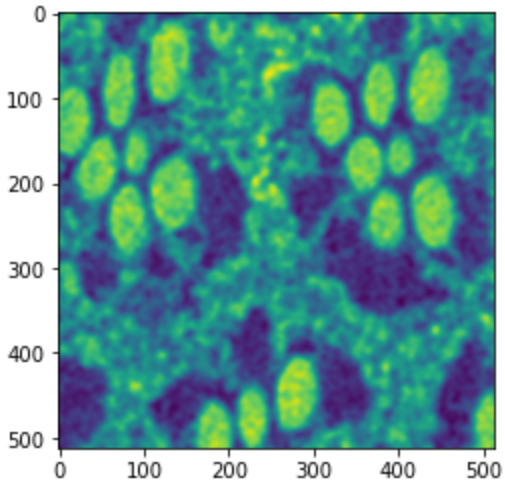

Figure 21: The cropped image is smoothed by applying the gaussian filter

4. Optimum Threshold Pipeline - The Optimum Threshold Pipeline (OTP) is a component in which we loop the image through the pipeline with varying threshold values. Since the images in our dataset had different levels of contrast, choosing one static threshold did not give best segmentation results for the image. We set the mean value of the smoothened image as the default threshold value. We add and subtract fourty from the mean value, and keep these as limits of the threshold value loop. Then for each threshold, we apply region labeling and create a binary mask of the image. We finally check the accuracy of the mask and check if it satisfies the least pixel count required in a good mask, as per the earlier set up parameters. We keep track of

the best threshold value along the loop. The computer vision techniques used in the OTP component are -

a. Thresholding - Thresholding involves converting a grayscale or color image into a binary image. The basic idea behind thresholding is to set all pixel values above a certain threshold value to one and all pixel values below the threshold value to zero. To achieve thresholding in python, we set all pixel values in the image array that are greater than or equal to the threshold value to 0, effectively converting pixels to black.

```
threshold_data = ImageController.threshold(smoothed_data, threshold_value=83)
ImageController.display(threshold_data)
```

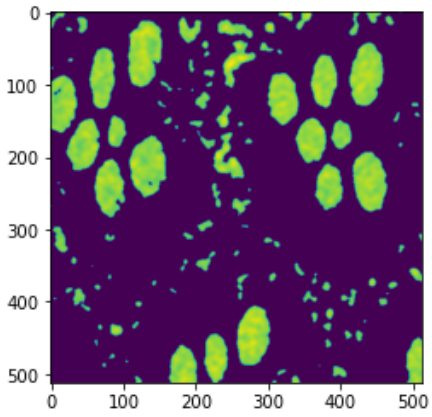

Figure 22: Thresholding applied on the smoothened image. The smoothed image is used as input and then thresholding is applied on the image. The threshold value in this figure is 83

b. Region labeling and size based filtering - Region labeling is a technique to identify and label contiguous regions in an image with similar properties. It assigns a unique label to each region and groups together pixels with the same label. This helps in finding connected components, such as photoreceptors in fly retina EM images. Size-based filtering is used to remove small regions or objects from an image or isolate the largest objects. The image is labeled using region labeling, and the labeled image is used to measure region sizes. Regions are then removed based on size requirements.

```
selected_labeled_data = ImageController.remove_regions(labeled_data,
                                                       min_region_size=750,
                                                       max_region_size=750000)
ImageController.display(selected_labeled_data)
```

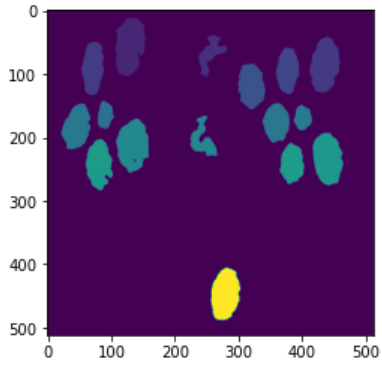

Figure 23: Thresholded image after applying size filtering. The regions smaller than min_region_size and bigger than max_region_size are removed in size filtering operation

c. Closing - Closing is a morphological operation that is used to fill in small holes and gaps in an image while preserving the overall shape and size of the objects in the image. It is a dilation followed by an erosion operation, where the dilation expands the objects in the image and the erosion shrinks them back, resulting in the filling of small gaps between the objects. The closing operation is useful for smoothing out the boundaries of objects in an image and for removing small noise or speckles from the image.

d. Mask Accuracy Calculator(MAC) - We calculate the accuracy of the mask by finding the possible number of rhabdomeres(circular/elliptical in shape) and dividing it with the number of regions present in the mask. The assumption used for calculating the accuracy as above is that a perfect mask image should only have rhabdomeres and no other regions. To find the circular/elliptical structures, we first get three properties for all the regions - area, perimeter and bounding box. We get these properties using the measure.regionprops function from the skimage python library. We then calculate circularity and squareness of each region in the image. Since the rhabdomeres are neither perfectly circular nor their bounded box is a perfect square, we keep the limits of both circularity and squareness as 0.4 to 2. If a region satisfies both circularity and squareness, then we count it as a rhabdomere. Finally we compare the number of rhabdomeres found with the number regions present to calculate the

accuracy.

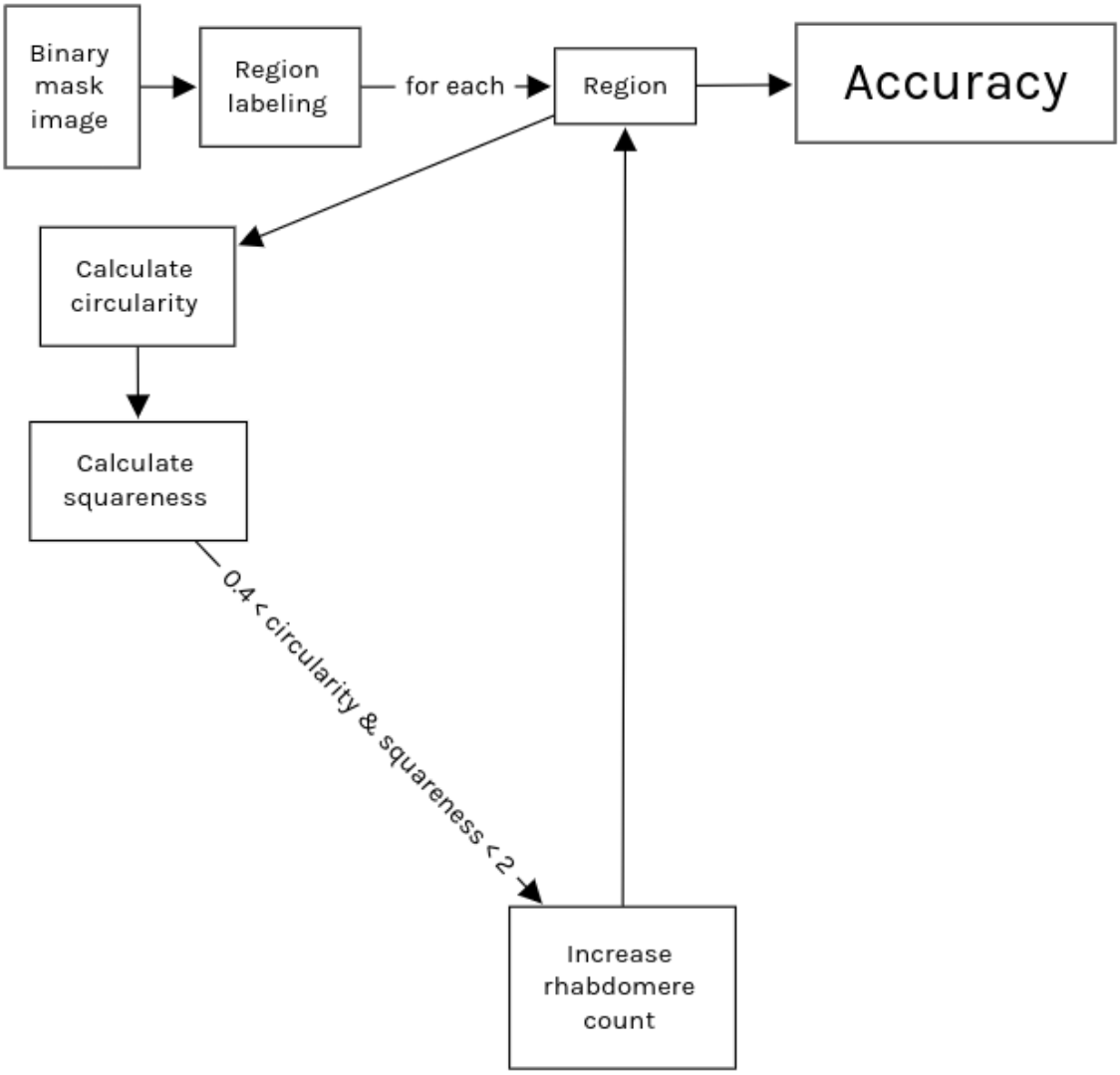

Figure 24: Flow chart explaining the Mask Accuracy Calculator(MAC) Component. First region labeling is performed on a binary mask to identify all regions. Then for each region, circularity and squareness is calculated. If both circularity and squareness is between 0.4 and 2 then the region is counted as a rhabdomere. Accuracy is calculated by dividing the number of rhabdomeres by the total number of regions in the binary image.

```
@staticmethod
def check_accuracy(binary_mask):
    # count number of rhabdoemeres
    # then calculate accuracy of mask
    # accuracy = number of rhabdomeres / total regions in mask
    # a mask that only has rhabdomeres as regions is a mask with accuracy = 1
    num_circles = 0
    num_sq = 0
    num_circle_sq = 0
    binary_mask_closed = ImageUtils.get_closed_binary_mask(binary_mask)
    labeled_binary, nr_objects_binary = ImageUtils.apply_region_labelling(binary_mask)
    # get all regions in image
    region_sizes = measure.regionprops(labeled_binary, intensity_image=binary_mask_closed)
    # check following two properties of regions
    # 1. circularity
    # 2. squareness of bounding box
    # if a region satifies both circularity and squareness
    # then count it as a rhabdomere
    for region in region_sizes:
        x1, y1 = region.centroid
        minr, minc, maxr, maxc = region.bbox
        x = maxc - minc
        y = maxr - minr
        bx = (minc, maxc, maxc, minc, minc)
        by = (minr, minr, maxr, maxr, minr)
        circularity = (region.perimeter ** 2) / (4 * math.pi * region.area);
        if circularity < 2 and circularity > 0.4:
            num_circles += 1
        if x/y >= .4 and x/y <= 2:
            num_sq += 1
        if circularity < 2 and circularity > 0.4 and x/y >= .4 and x/y <= 2:
            num_circle_sq += 1
    # if region_sizes has no elements then set accuracy as 0.0001
    if (len(region_sizes)) != 0:
        accuracy = num_circle_sq/len(region_sizes)
    else:
        accuracy = .0001
    return accuracy, num_circle_sq
```

Figure 25: Code for checking accuracy of a binary image. The code performs the steps shown in flowchart figure 24.

5. With the best threshold value that we got from OTP, we processed the initial smoothened image. We first filter filter pixel values according to the best threshold value. Then region labeling and size filtering is done to remove smaller or larger regions based on the dataset. Finally closing of holes is done on the image, and a binary mask is created for the image.

```
threshold_data = ImageController.threshold(smoothed_data, threshold_value=best_threshold)
labeled_data, nr_count = ImageController.label(threshold_data)
selected_labeled_data = ImageController.remove_regions(labeled_data,
                                                       min_region_size=MIN_REGION_SIZE,
                                                       max_region_size=MAX_REGION_SIZE)
binary_mask_data = ImageController.binary_mask(selected_labeled_data)
closed_binary_mask_data = ImageController.close_binary_mask(binary_mask_data)
segmented_image = ImageController.binary_image(closed_binary_mask_data)
```

Figure 26: Processing original image with the best threshold received from OTP. Code snippet shows the steps performed once the optimum threshold is calculated.

6. Our pipeline saves all the plots in the directories we set up before running the pipeline. Comparison plots with the circular and square labeled data is saved for all images in the dataset.

```
ImageController.display_and_save_compare_labeled(data_crop,
                                                 binary_mask_data,
                                                 segmented_image,
                                                 plot_path,
                                                 show_in_notebook=True)
```

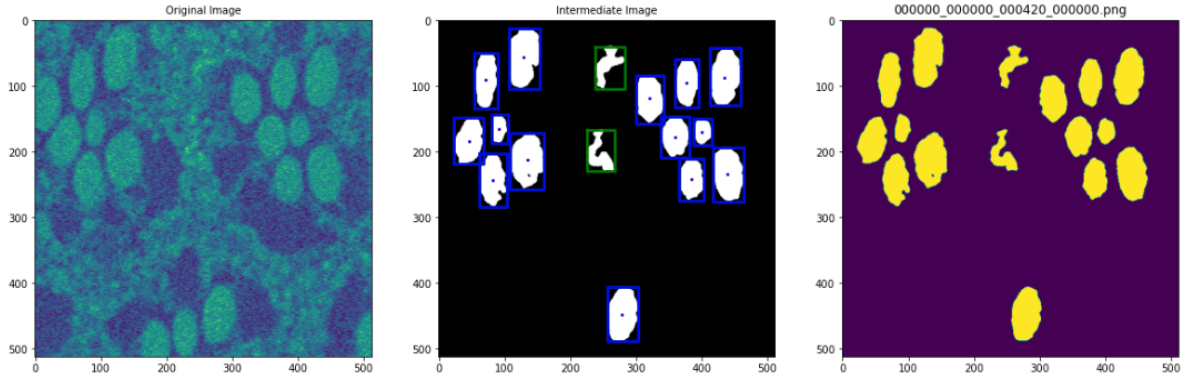

Figure 27: Pipeline results displayed for one image from Wild Type CS 1 dataset. 512 X 512 crop of original image on left. Intermediate image in middle. Plot shows regions which are both circular and enclosed in square bounded boxes are labeled in blue color. Final segmentation mask on the right.

7. Since one ommatidium consists of 7 rhabdomeres, an approximate number of ommatidia present in the mask is calculated by dividing the number of rhabdomeres by seven.

```
num_ommatidia = round(num_rhabdomeres / NUM_RHABDOMERE)
# if number of ommatidia in image and number of pixels in image
# satisfy least count, then segmentation mask is good.
if num_pixels > LEAST_PIXEL_COUNT and num_ommatidia >= LEAST_OMMATIDIA_COUNT:
    good_mask_count += 1
    print("Good segmentation mask")
    # Save binary mask and compare plot only for good mask image
    ImageController.save_compare(data_crop, segmented_image, compare_plot_name)
    ImageController.save(binary_mask_name, segmented_image)
else:
    bad_mask_count += 1
    print("Bad segmentation mask")
```

Figure 28: Code snippet showing condition for good or bad mask. If the number of ommatidia and number of pixels in the segmentation mask exceed the minimum required number then the segmentation mask is deemed as good.

8. Finally we count the number of pixel values in the binary mask. We use the two parameters - number of ommatidia and number of pixel values - to judge whether the mask is good or not. The threshold value of both the parameters are set before the pipeline is run, while setting up the pipeline environment. The binary mask and a comparison plot of binary mask and original image is saved at the directory path.

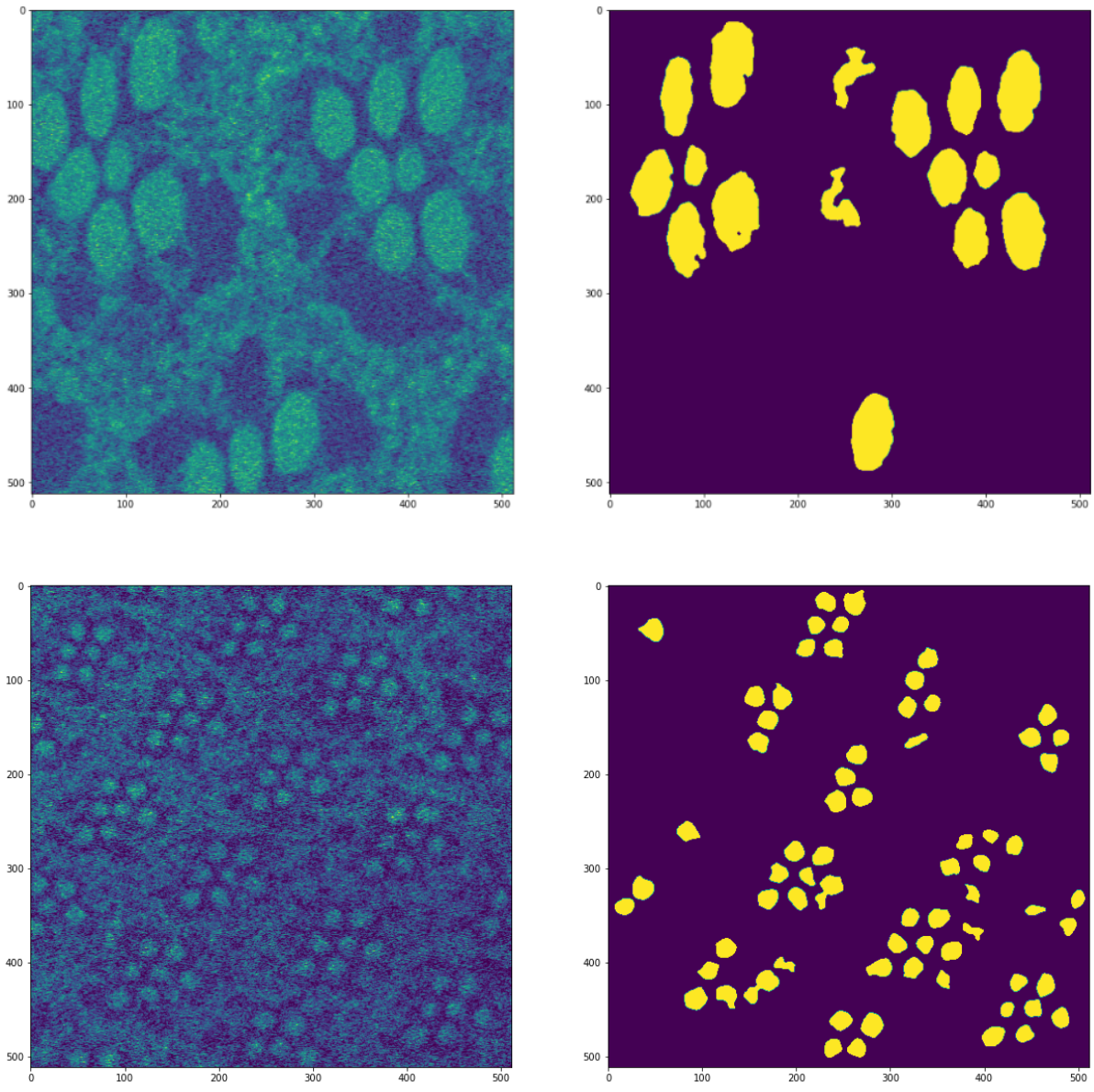

Figure 29: A side-by-side comparison of 512 X 512 cropped Wild Type CS 1 Overview dataset image on left and its segmentation mask on the right in the top row. Same comparison for Nina D1 Mutant Overview in the bottom row.

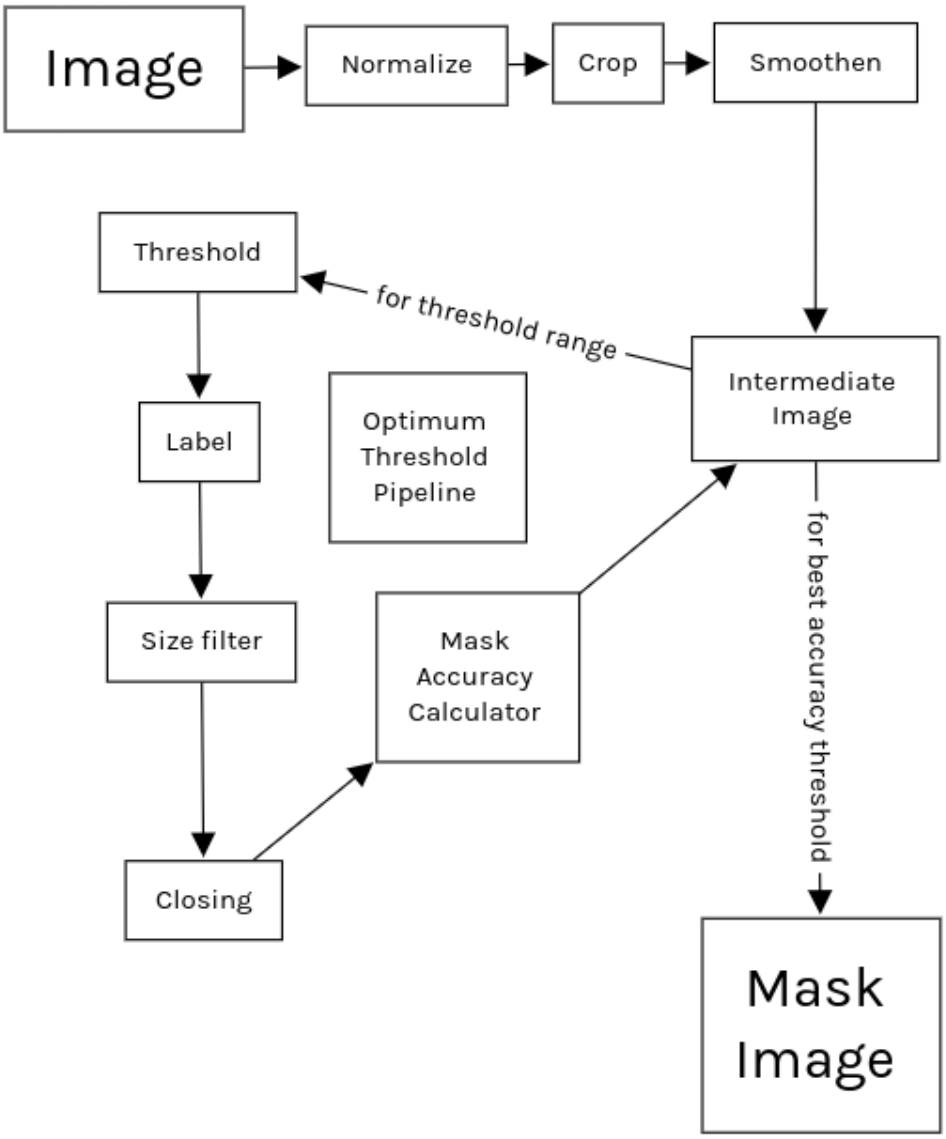

Figure 30: Flow chart explaining complete segmentation pipeline. Image is loaded from the dataset. Normalization and cropping operation is applied on the image. The image is then smoothed using the gaussian filter. Then for the range of threshold in threshold bracket, OTP is run to find the optimum threshold for segmentation. Once the optimum threshold is found, a binary mask for the image is generated using the optimum threshold value.

CHAPTER 5

EVALUATION

We evaluated the above pipeline on two datasets -

1. Wild type CS 1 Overview tileset - Vitamin A replete fly
2. Nina D1 mutant overview tileset - Vitamin A deprived mutant fly

We subjected 300 images of fly retina EM images from both the datasets to the segmentation pipeline, which produced a binary mask, if the segmentation was found good. We defined a few parameters to fine tune the pipeline. The values of each parameter was chosen as per the features of the dataset. Following are the parameters that we set along with the chosen values for each parameter -

1. CROP_SIZE - We set the crop size for both the dataset as (512, 512). The small crop size is computationally efficient as it is easier to apply image processing techniques on small image arrays.

2. MIN_REGION_SIZE and MAX_REGION_SIZE - The minimum and maximum size of the region are kept and rest regions are removed when performing size filtering. The value of MIN_REGION_SIZE for the Nina D1 Mutant dataset having (512, 512) crop size is set to 150 and MAX_REGION_SIZE is set to 750. This means that once regions are labeled, we remove any region that has size smaller than 150 and bigger than 750. The same values for Wild Type CS 1 dataset having (52, 512) crop size are 750 and 750000 respectively.
3. LEAST_PIXEL_COUNT - This is the least number of pixels that should be present in the binary mask. This value helps in judging whether the mask is good or not. For both the datasets, we found that the LEAST_PIXEL_COUNT as 20000 works well.
4. LEAST_OMMATIDIA_COUNT - The least number of ommatidia present in the binary mask. Since the resolution of both the datasets was different, we found that (512, 512) crops of the Wild Type CS 1 should at least have one ommatidia and that of Nina D1 Mutant should have at least six ommatidia.

Although the datasets are large with the Wild Type CS 1 having 861 slices and Nina D1 Mutant having 2164 slices, we ran the pipeline on 300 images from both the datasets. A small number was chosen to make pipeline processing computationally faster. All images are randomly spread over all the slices of the dataset.

As the pipeline runs, we also log relevant information like the best threshold value for the image, accuracy of the mask, possible number of rhabdomeres in the image, and number of good and bad masks to name a few.

```
-----------------------------------------------------------
Image Name: 000000_000000_002168_000000.tif
Threshold mean:  41.45377669635975
Threshold best:  0
Accuracy of mask:  0
Bad segmentation mask
Possible number of rhabdomeres:  0
Total number of pixels in the mask:  0
Total Time Taken:  2.7849934101104736  seconds
Number of good masks so far:  257
Number of bad masks so far:  43
Total files processed so far:  300
-----------------------------------------------------------
Number of good masks:  257
Number of bad masks:  43
Total processed images: 300
```

Figure 31: A screenshot of logs generated on successful run of the pipeline on 300 images from Nina D1 Mutant Overview dataset. The logs are extensive and show all relevant information for both good and bad segmentation mask images.

We got following results from the pipeline -

Dataset	Total Images	Good mask	Bad mask
Wild Type CS 1	300	162	138
Nina D1 Mutant	300	257	43

Table 2: A table showing results of the pipeline run on 300 images from both Wild Type CS 1 Overview dataset and Nina D1 Mutant Overview dataset.

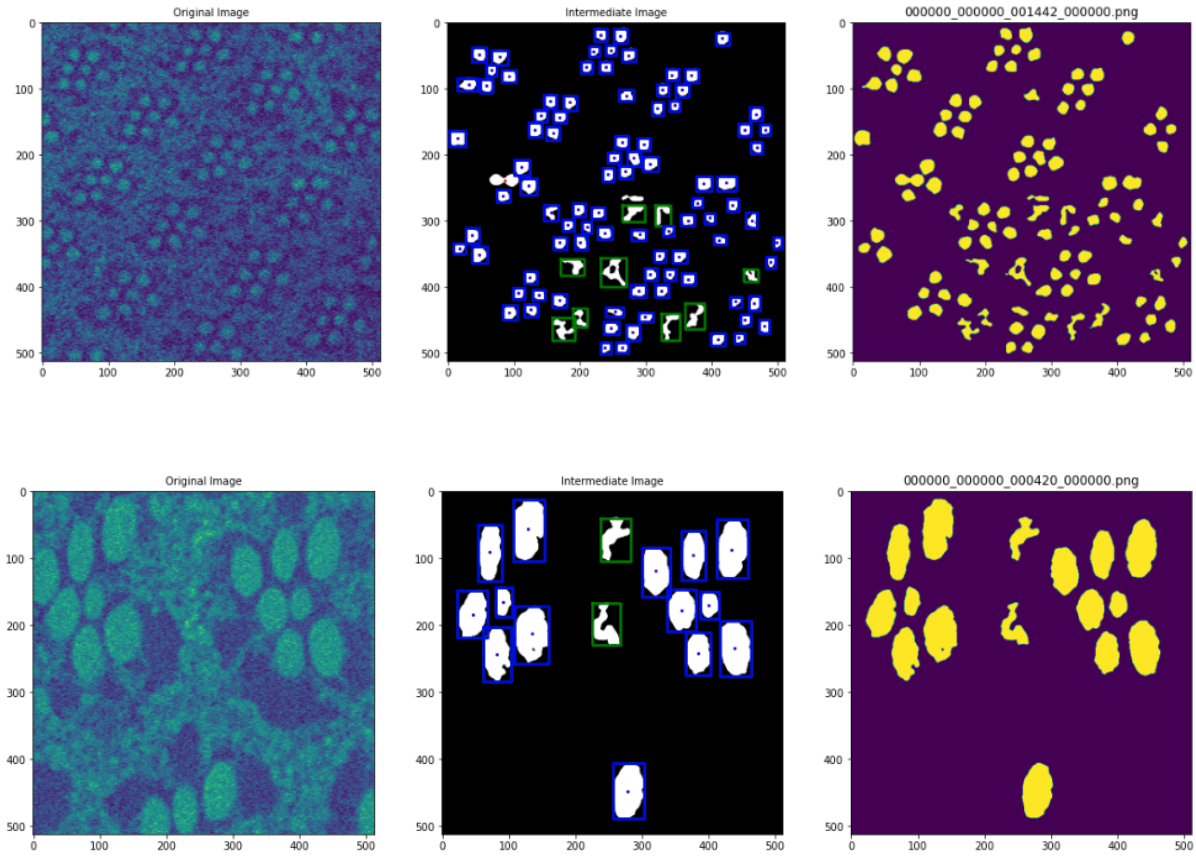

Figure 32: Pipeline results displayed for one image from the Nina D1 Mutant dataset in top row. 512 X 512 crop of original image on left. Intermediate image in middle. Plot shows regions which are both circular and enclosed in square bounded boxes are labeled in blue color. Final segmentation mask on the right. Same plots are shown for a one image from the Wild Type CS 1 dataset in bottom row. Although the segmentation pipeline results look good, the masks generated for the Nina D1 Mutant dataset are not the best and contain noise around the circular rhabdomere structures. The pipeline is not able to perform at its best due to varying contrast levels and multiple artifacts in the dataset images. The smaller size of rhabdomere in the Nina D1 dataset indicates Vitamin A deficiency in the mutant.

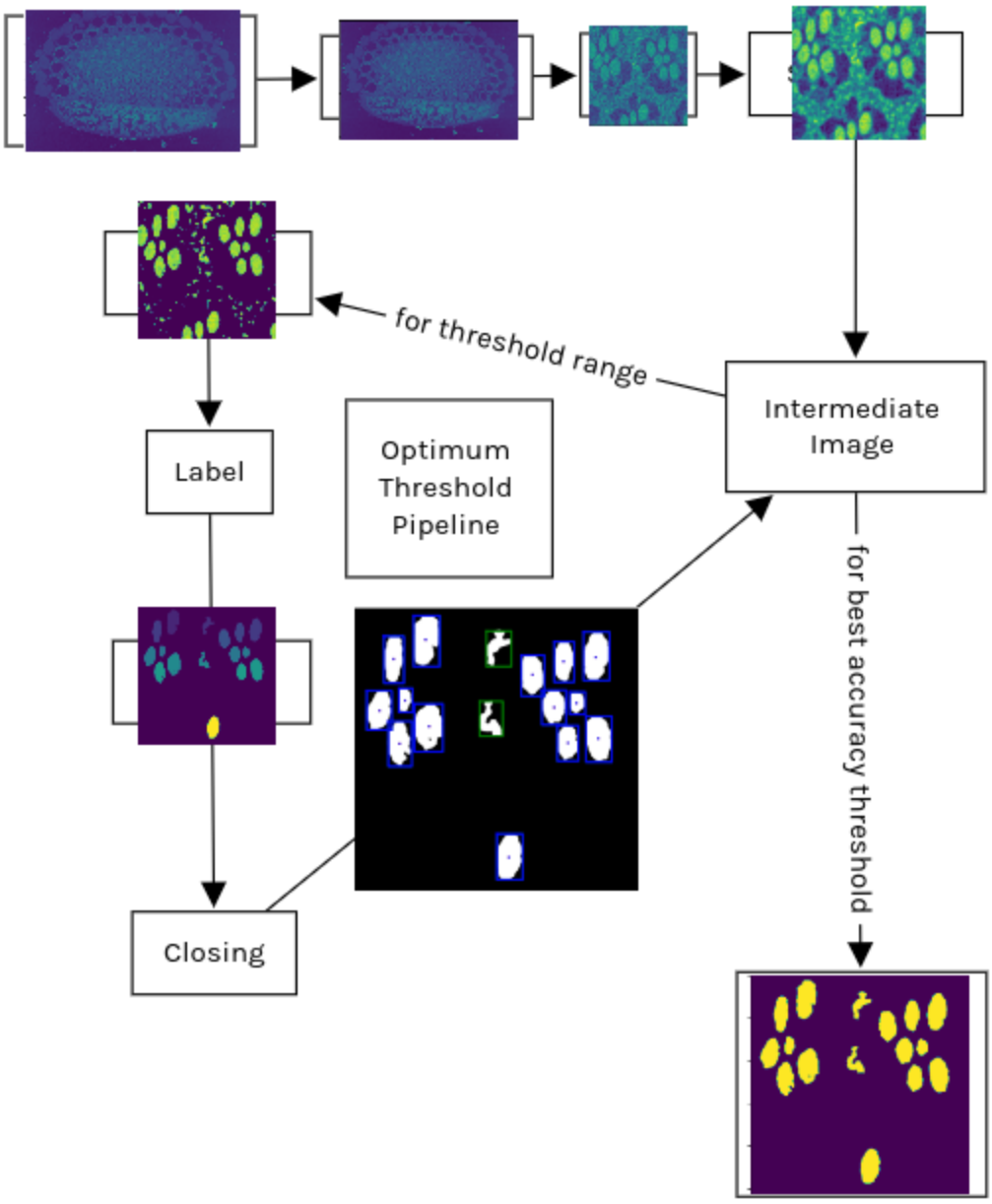

Figure 33: Pipeline from Figure 28, shown with images for different steps. The image shown is an image taken from the Wild Type CS 1 Overview dataset.

CHAPTER 6

CONCLUSION AND FUTURE WORK

The segmentation pipeline results indicate that traditional computer vision techniques are useful in image segmentation. Traditional methods, such as thresholding, region growing, and morphological operations, are easy to interpret, computationally efficient, and effective for images with simple or well-defined objects or regions. With correct threshold values and appropriate region size filtering we were able to achieve good segmentation results for both Wild Type CS 1 and Nina D1 Mutant Overview tileset datasets. By counting the approximate number of rhabdomeres and the total number of foreground pixels present in the segmentation mask, we created a metric to identify if the generated mask is good or not. Also, it is evident from the segmentation mask of Nina D1 Mutant that it is Vitamin A deprived fly as there is no visual pigment in the rhabdomeres which causes severe rhabdomere size defects as compared to that of Wild Type CS 1 fly.

However, seeing the results it is quite evident that the effectiveness of traditional computer vision techniques is limited when dealing with images that have multiple artifacts such as noise, uneven illumination, and background variations. Thresholding, edge detection, and region growing may not be effective in processing these images due to variations in background intensity and size filtering challenges. To find the most optimum threshold, we had to loop through multiple threshold values and check accuracy for every threshold value. Even after finding the most optimum threshold, the resulting masks still contain small noises around the circular rhabdomere structures.

Due to various artifacts in the dataset, a more robust approach for segmentation could be one which uses Convolutional Neural Network(CNN) and deep learning models to predict the region of interest in the image. CNNs are more powerful and flexible, capable of learning complex patterns and relationships in the data, and handling noisy or ambiguous images. They have performed well on tasks involving image segmentation, especially when dealing with complex or ambiguous images. The U-Net CNN architecture is one such model which is suitable for image segmentation where image features, noise, and contrast ratios are not uniform. It can handle ambiguous images and adapt to a wide range of data distributions, making them more generalizable across different domains. However, training U-Net models requires large amounts of labeled data and significant computational resources. They may also be less interpretable or transparent than traditional computer vision techniques.

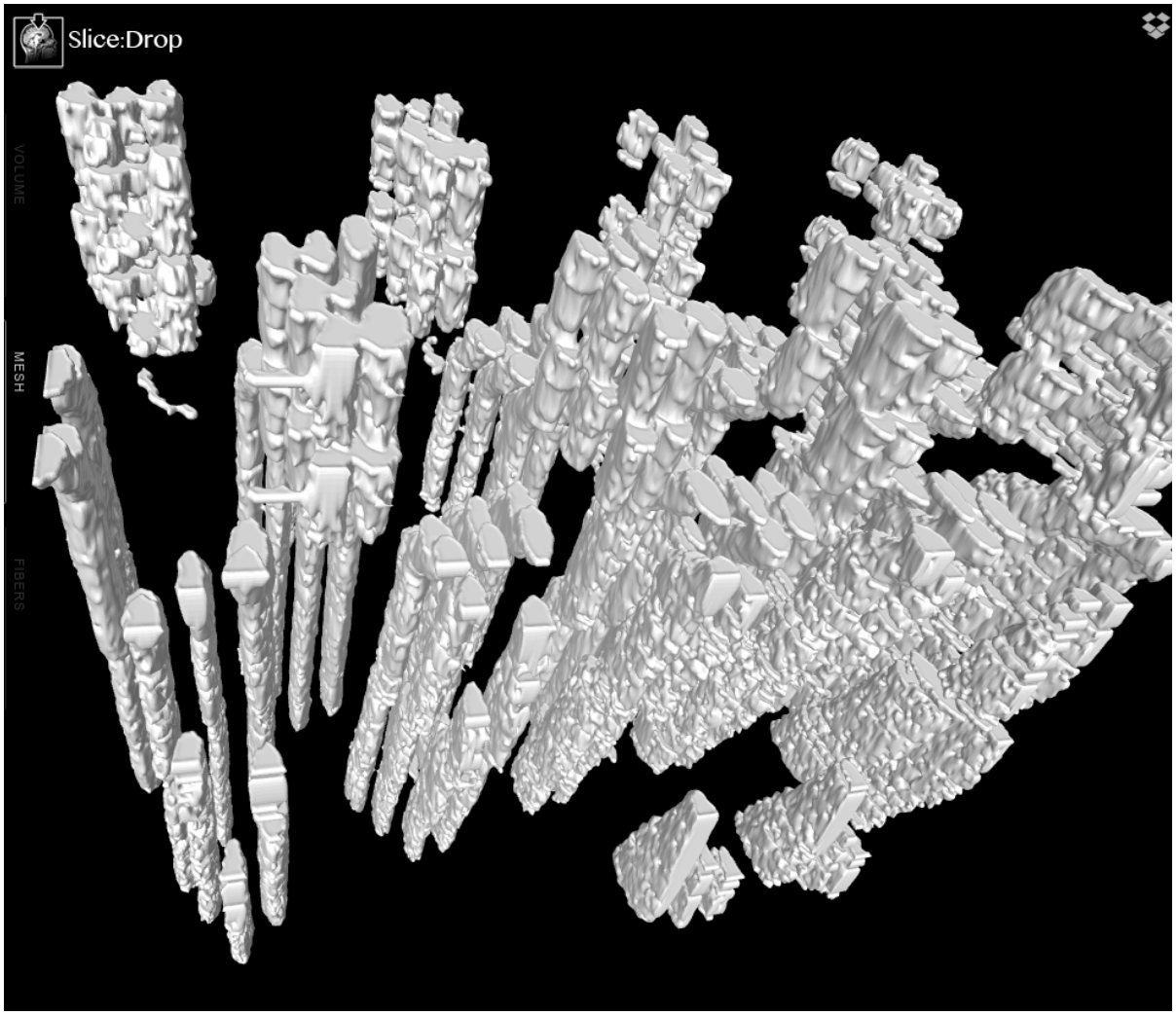

Figure 34: 3D reconstruction of drosophila photoreceptors constructed using the binary mask obtained from the computer vision pipeline

The research thesis also aims to develop a prototype for image segmentation of Drosophila photoreceptors in all fly EM datasets received from Indian University. Once we have all the segmentation masks, we can use them to reconstruct and visualize hexagonal rhabdomeres

in 3D and study the biological structure of fly photoreceptors in detail. 3D reconstruction of Drosophila photoreceptors usually involves using computer algorithms to combine the segmentation masks into a single 3D model. The 3D model can provide detailed information about the shape and structure of the cells, which can be useful for understanding their function and role in the nervous system.

All the data and source codes are available on GitHub:

https://github.com/jainkhere/flyem

REFERENCES

Deepshe Dewett, Khanh Lam-Kamath, Clara Poupault, Heena Khurana, Jens Rister. Mechanisms of vitamin A metabolism and deficiency in the mammalian and fly visual system. *Developmental Biology 476 (2021)*.

Deepshe Dewett, Maryam Labaf, Khanh Lam-Kamath, Kourosh Zarringhalam, Jens Rister. Vitamin A deficiency affects gene expression in the Drosophila melanogaster head. *G3 (Bethesda) (2021)*.

Marion Langen, Egemen Agi, Dylan J. Altschuler, Lani F. Wu, Steven J. Altschuler, Peter Robin Hiesinger. The Developmental Rules of Neural Superposition in Drosophila. *Cell Volume 162 (2015)*.

Jens Rister, Claude Desplan, Daniel Vasiliauskas. Establishing and maintaining gene expression patterns: insights from sensory receptor patterning. *Development Volume 140 (2013)*.

Krystina Schopf, Thomas K. Smylla, and Armin Huber. Immunocytochemical Labeling of Rhabdomeric Proteins in Drosophila Photoreceptor Cells Is Compromised by a Light-dependent Technical Artifact. *Journal of Histochemistry & Cytochemistry 2019*.

Q. He, M. Hsueh, G. Zhang, D. C. Joy, R. D. Leapman. Biological serial block face scanning electron microscopy at improved z-resolution based on Monte Carlo model. *Scientific Reports (2018)*.

Davies, E., R. Machine Vision: Theory, Algorithms, Practicalities (Signal Processing and Its Applications Series). *Academic Press, 1996*.

Luis Pedro Coelho. Mahotas: Open source software for scriptable computer vision. *arXiv:1211.4907v2 [cs.CV] 5 Jan 2013*.

Diagrams of the Eye and Retinal Organization of the Fruit Fly Drosophila (https://doi.org/10.1371/journal.pbio.0060115.g001)

Scikit image processing - Region properties
(https://scikit-image.org/docs/stable/auto_examples/segmentation/plot_regionprops.html)

Google Neuroglancer - WebGL based viewer for volumetric data
(https://github.com/google/neuroglancer)

Neuroglancer volumes - 3D image volumes on neuroglancer
(https://github.com/google/neuroglancer/blob/master/src/neuroglancer/datasource/precomputed/volume.md)

Neuroglancer scripts - Conversion of 3D volume into neuroglancer format
(https://github.com/HumanBrainProject/neuroglancer-scripts/blob/master/docs/script-usage.rst)

Neuroglancer python integration - Running neuroglancer scripts in python
(https://pypi.org/project/neuroglancer/)

Numpy Python (https://numpy.org/doc/stable/)

Geeks for geeks - Computer Vision techniques
(https://www.geeksforgeeks.org/computer-vision-introduction/)

Mahotas documentation
(https://buildmedia.readthedocs.org/media/pdf/mahotas/release-1.0/mahotas.pdf)

Introduction to Computer vision (https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html)