

# Infrared Insights

## Identifying Leaky Windows with Small-Sample Thermal Image Analysis Techniques

Leigha DeRango

### Introduction

This project aims to use cost-effective thermal imaging technology in combination with commodity hardware to identify windows that are “leaky”, or fail to successfully insulate their seams. Currently, this identification is done mostly through professional services, who offer to take the image and then manually analyze it and return recommendations based on their in-house conclusions. However, this type of analysis does not prevent companies from referring unnecessary procedures to clients with the goal of making money off of performing the procedure. Generally, we wish to allow such analysis to be more easily done by homeowners, providing them with the tools to decide how to best take care of their home. This involves three parts: collecting images and metadata such as direction the window is facing, outside temperature, etc; analyzing the images to identify what trends may be indicating leakiness; and finally, creating a method to take in the image, identify whether this trend is present in the image or not, and returning that determination to the user. While this project does not come close to accomplishing these goals, the goals did guide our research for handling what little preliminary data we did have. In the end, we provide a summary of thermal imaging techniques and how these methodologies guide an effective procedure to get the most useful images, along with functions to identify the shape of the window and temperature scale of the image. We also provide recommendations for reperforming this project and moving forward with image analysis.

### Problem Statement

Currently, identifying leaky windows is mostly done through manual thermal imaging and visual inspection, and does not utilize growing technology that can handle and recognize trends in thermal images that may speed up this process and put more information in the hands of homeowners. This development will help various stakeholders identify infrastructure issues, whether that be homeowners or building development administrators, save time and money by quickly and accurately identifying issues, and generally improve a homeowners’

ability to monitor the state of their home without requiring a third party. Overall, simplifying the process of analyzing thermal images will improve the quality of future investments in the insulation and window industries.

The severe limitation this study faces is small sample size. Modern computer vision techniques utilize machine learning, which requires minimum sample sizes in the thousands for very simple projects, even a binary classification of “leaky” or “not leaky”. It is because of this that we acknowledge the power of computer vision in accomplishing our goals, but cannot employ it to its fullest extent in this project.

## **Data Analysis and Methodology**

We began the project by attempting to efficiently familiarize ourselves with image processing techniques so that we could give the field researchers their procedure as soon as possible, in order to facilitate as much data collection as possible. The most important aspect of processing in this case, given the unavailability of computer vision for analysis, is getting the most straightforward and uniform images as possible. This will facilitate batch processing and reproducibility of results from one image to another. The field researchers were instructed to take images as straight on as possible and take only images of windows that would fit in the entire frame from between 1 to 6 feet away, as this was the distance recommended by the camera manufacturers for optimizing the information the camera could capture. A full copy of the document given to the field researchers is attached as Appendix A. It should be noted that we did not obtain a single image (out of about 70 received) that met the requirements listed in the procedure. The images used in this report were obtained outside of the original field research.

Image processing is split into three sequential procedures: pre-processing, processing, and computer vision. The OpenCV package in Python provides a wide range of image processing functionality and was used for the entirety of the following analysis. Image pre-processing involves reading in and cleaning the data, along with performing any necessary adjustments to make the images more conducive to later analysis. For us, image pre-processing included Gaussian blurring and contrast enhancing, before thresholding to complete the binarization of the image necessary for finding the shape of the window, called a contour. These are common pre-processing techniques, which were chosen after evaluating a variety of possibilities and what they may contribute to identifying regions of a window (Khandelwal). Gaussian blurring performs noise reduction by establishing a kernel and then applying a certain number of standard deviations of the kernel value to the pixel at the center of that kernel. Larger standard deviations result in blurrier images. This parameter could be tuned if more data was acquired. To improve contrast of the image, histogram equalization was used. Histogram equalization takes the range of pixel intensity values of the existing image and spreads out the most frequent values to create greater contrast in the image. Thresholding is used to help identify an object and separate it from the background of an image, and works by selecting a specific pixel intensity value. Any pixel with a value less than the threshold selected is set to

zero, or white, and any pixel greater is set to the maximum value chosen (often 255, or black). In essence, this technique returns a flattened image where the object of interest is one color and the background is the other. This value is another processing parameter which could be tuned given more images.

The next step is image processing, which performs analysis but does not necessarily generate insight about the image, as a neural net may be able to do. Using simple image processing techniques and OpenCV, the values of a thermal image can be used to identify the region of the window in the image, also called the “contour”. Once the region of the window is identified, we examine different methods that may begin to identify indicators of leakiness. FindContours() identifies the boundary of points between the black and white regions identified in pre-processing. This process can be visualized with Figure 1, which shows our test image at four stages: raw, gray-scaled and blurred, thresholded, and finally with the largest contour overlaid on the raw image.



Figure 1: Progression of Image Processing

To better account for variation around the edges of a leak, we also utilized OpenCV’s approxPolyDP() to simplify the contours and ideally better identify a normally-shaped window. The function takes an additional input parameter, epsilon, which is a threshold for how important a point on the contour must be in order to be retained in the simplified polygon, as determined by the Ramer–Douglas–Peucker algorithm. A higher epsilon results in less edges in the final contour (Open Source Computer Vision). The result of this approximation on the sample image can be seen in Figure 2.

Using this framework, we created a series of scripts to process a collection of images at one time. Knowing that our end goal would be to make this process of classifying a window as ‘leaky’ or ‘not leaky’ as autonomous as possible, scripting the image processing steps is the first step towards being able to gain enough data to utilize a neural net or other image classification model. We begin with a function to approximate a four sided contour in the input image, approx\_four\_sides(). To account for potential temperature variation due to leakiness or an off-axis photo, the function loops through increasing values of epsilon until the returned shape has four vertices, in order to best approximate rectangular windows. This does render the analysis inadequate for non-rectangular windows, though future work may allow for a larger sample size and more room to account for a variety of window shapes. If the loop does not find a four-sided contour by the time epsilon reaches 0.15, it returns an error message with

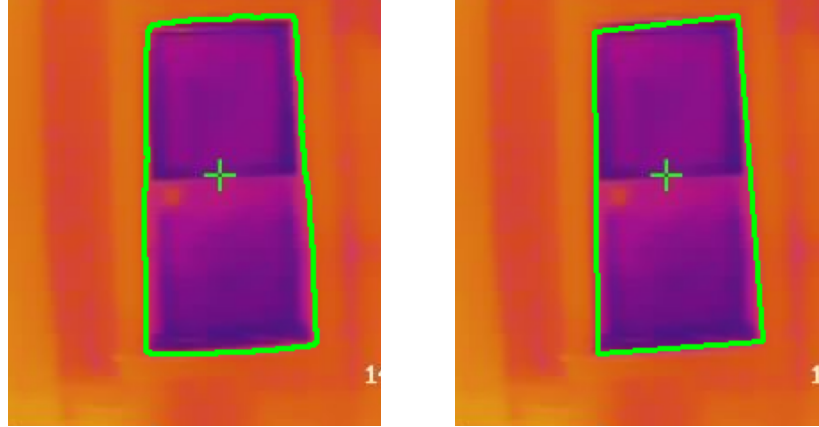


Figure 2: Result of Contour Approximation on Region Definition

a list of the number of edges it found for each. The function returns a list of the identified epsilon value and the contour object it corresponds to.

```
# function to read a (ideally cropped, but handles all images) opencv format image and return
#that produces an approximated 4-edge contour
def approx_four_sides(image):

    # put image into thresholded form
    imgRGB = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # translate from CV to RGB format
    imgGray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # make image Grayscale
    imgBlur = cv2.GaussianBlur(imgGray,(7,7),1)
    # apply Gaussian Blur
    imgEqualized = cv2.equalizeHist(imgBlur)
    # increase contrast
    imgBlurer = cv2.GaussianBlur(imgEqualized,(15,15),1)
    # apply stronger Gaussian Blur

    # threshold
    _, threshold = cv2.threshold(imgBlurer, 110, 270, cv2.THRESH_BINARY)
    threshold_inverted = cv2.bitwise_not(threshold)
    # invert (contours looks for white space)

    # generate original contour
    contours, _ = cv2.findContours(threshold_inverted, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    # filter out contours based on area to get the largest
```

```

orig_contour = max(contours, key=cv2.contourArea)

# initialize
eps = 0.01
epsilon = eps * cv2.arcLength(orig_contour, True)
approx = cv2.approxPolyDP(orig_contour, epsilon, True)
len_vec = []
len_vec.append(len(approx))

while len(approx) != 4:
# for each value of epsilon from (0.01 - 0.15 by 0.005)

    eps = eps + 0.005
    #generate approximation of original contour
    epsilon = eps * cv2.arcLength(orig_contour, True)
    approx = cv2.approxPolyDP(orig_contour, epsilon, True)
    len_vec.append(len(approx))

# if num edges is 4 (rectangle)
if eps >= 0.15:
    msg = 'No four sided figure could be identified, search went: {}'.format(len_vec)

    return [msg, 0]

return [eps, approx]

```

In theory, with this function to generate an approximated contour for each image, we can take a list of images, generate their contours, and calculate the mean grayscale value in the region inside and outside of the window, which can be used in future temperature calculations. With the end goal of being able to evaluate “leakage” of a window, these grayscale values may serve as a reference point for that evaluation.

Also in line with this goal, the grayscale values still need to be translated to temperature values. In order to maximize the amount of information we could get from each image, the cameras given to the researchers were set to a variable scale. This means that the RGB scale is the same for every image, but the temperatures it represents change from image to image. However, the HIKMicro E1L camera does not store any form of metadata, only the jpeg. Along with our very small sample of images, this means that to translate the RGB to a temperature, we must use an existing optical character recognition (OCR) package to scrape the values off of the image. This process is made available due to the constant location of the scale on each image. If the scale location was variable, this method would have to be adapted. Using Tesseract and their Python driver pyTesseract, either end of the scale can be pulled out and

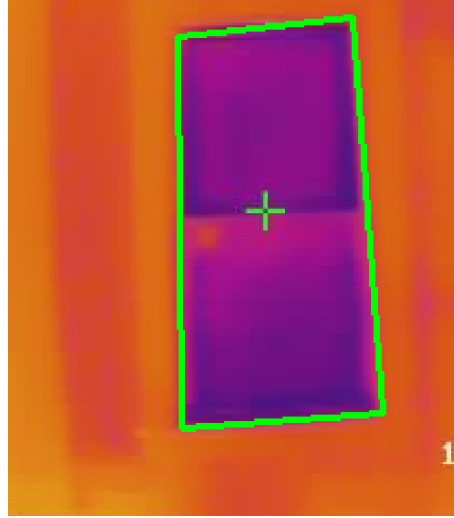


Figure 3: Resulting Four Sided Contour

run through `image_to_string()` to return a string of characters.

The complication of some of these images is that the coolest temperature is lower than the camera can register, which generally happens when the camera captures part of the sky or something much further away than the focus of the image. These values will be imputed to -20 C, which is the minimum temperature in Celcius the camera can register. Though the researchers were instructed to set the camera scale to be in Fahrenheit, most of the original images received were in Celcius, so we will continue with Celcius for the time being. To complete the scale to temperature translation, we grayscale the image. By grayscaling, each pixel value is in a single dimension, a value from 0 to 255 which progressively gets darker, to represent the one dimension variable we are trying to measure, temperature, as opposed to a three dimension RGB value. This allows for the translation of grayscale to temperature values using a simple linear expression:  $temperature = (grayscaleval/255) * (upperscalelimit - lowerscalelimit) + lowerscalelimit$ . This involves dividing the grayscale value of the region by 255 in order to find the scale point of interest, then multiplying by the range of the scale in order to get the change in temperature in degrees per a one unit increase in grayscale value, and finally adding the lower scale value to put the temperature in the context of the image. The following function takes in a list of OpenCV image objects and extracts all of this information into a dataframe with nine columns: the original image object, epsilon to generate a four-sided contour, the approximated four-sided contour object, the mean grayscale value inside the window region, the temperature of this inner region, the mean grayscale value outside the window region, the temperature of this outer the bottom scale value, and the top scale value.

```

# function to take in list of cv image objects and return a dataframe of [the uncropped image

def bulk_contours(img_list):

    # initialize pytesseract location
    pytesseract.pytesseract.tesseract_cmd = r'C:/Program Files/Tesseract-OCR/tesseract.exe'

    img_post = [] # initialize list to hold returned values from each image

    for image in img_list:
        # crop scale out of image to assist with contouring
        crop_image = image[60:300, 0:210]
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        #generate four-sided contour
        returned_eps, approx_contour = approx_four_sides(crop_image)

        if len(approx_contour) != 4:
            image = returned_eps = approx_contour = inner_gray = outer_gray = 'NA'
            scale_lower = scale_upper = 'NA'
            print('warning, rectangular window not identified in image')

        else:

            gray_crop_image = gray_image[60:300, 0:210] # grayscale whole image

            ## generate grayscale value in/out of region from approximated contour
            # use a mask to identify the region of interest
            mask = np.zeros(gray_crop_image.shape[:2], np.uint8)
            cv2.drawContours(mask, [approx_contour], 0, 255, -1)
            inner_gray = cv2.mean(gray_crop_image, mask = mask)[0] ## mean values

            # invert the mask in order to pick the values in the other region
            newImage = gray_crop_image[:, :]
            outer_mask = cv2.bitwise_not(newImage, mask)
            outer_gray = cv2.mean(gray_crop_image, mask = outer_mask)[0]

            ## get scale
            # crop numbers at top and bottom of scale
            scale_upper_crop = image[30:60, 190:240]
            scale_lower_crop = image[260:280, 170:240]

```

```

# translate image to strings
scale_upper = pytesseract.image_to_string(scale_upper_crop)
scale_lower = pytesseract.image_to_string(scale_lower_crop)

try:
    scale_upper_numeric = float(scale_upper.strip())
except:
    scale_upper_numeric = 1000
    # if upper string is not able to be converted, impute unreasonable number
try:
    scale_lower_numeric = float(scale_lower.strip())
except:
    scale_lower_numeric = -20
    # if lower string is not able to be converted, impute lowest camera val

scale_diff = scale_upper_numeric - scale_lower_numeric

inner_temp = (inner_gray/215)*scale_diff + scale_lower_numeric
outer_temp = (outer_gray/215)*scale_diff + scale_lower_numeric
# temp = prop/grayscale range*numeric scale range + lower end of scale

img_post.append([image, returned_eps, approx_contour, inner_gray, inner_temp, outer_gray])

#turn list img_post into a dataframe and return
df = pd.DataFrame(img_post, columns = ["image", "eps_for4", "approx_contour", "inner_gray"])

return df

```

We must remember that the single-dimension scale values do not necessarily span the entire (0,255) range, and adjust the temperature calculations based on the true scale difference. The translation from RGB to grayscale values is found in the `imgproc` module documentation (Open Source Computer Vision).

```

image = cv2.imread(r"images/IMG000001.jpeg")
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

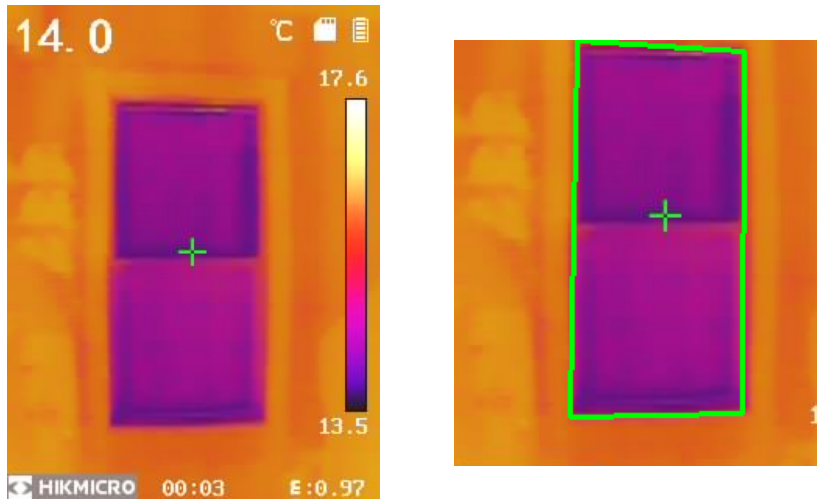
# find grayscale values from uncropped and translate to gray scale
r_u, g_u, b_u = image[256,225]
gray_upper_val = (0.299*r_u) + (0.587*g_u) + (0.114*b_u)

```



```
r_l, g_l, b_l = image[64, 225]
gray_lower_val = (0.299*r_l) + (0.587*g_l) + (0.114*b_l)
```

The summary of this sequence of functions on an image from an HIKMicro Camera can be seen here. There is work to be done in validating the results of the OCR used to calculate the scale of the image (the function was unable to recognize 13.5 as the bottom of the scale), but this is easily accomplished with the collection of new data and exploration of advanced binarization methods.



Data Sample

	image	eps_for4	\	
0	[[[220, 148, 28], [220, 149, 23], [222, 150, 1...	0.01		
	approx_contour	inner_gray	inner_temp	\
0	[[[68, 0]], [[65, 211]], [[164, 208]], [[162, ...	69.67	-7.816	
	outer_gray	outer_temp	lower_scale_val	upper_scale_val
0	89.794	-4.297	-20	17.6

Description of DataTypes

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1 entries, 0 to 0
Data columns (total 9 columns):
#   Column          Non-Null Count  Dtype
---  -
0   image            1 non-null      object
```

```

1  eps_for4          1 non-null    float64
2  approx_contour    1 non-null    object
3  inner_gray        1 non-null    float64
4  inner_temp        1 non-null    float64
5  outer_gray        1 non-null    float64
6  outer_temp        1 non-null    float64
7  lower_scale_val   1 non-null    int64
8  upper_scale_val   1 non-null    float64
dtypes: float64(6), int64(1), object(2)
memory usage: 204.0+ bytes

```

## Recommendations

This project is a work in progress and has much to be done before it could serve as a fully functioning product.

To begin, there are plenty of adjustments to be made to the existing code. In the image pre-processing, we would also recommend an exploration into different methods of thresholding, such as Gaussian adaptive or mean adaptive, which account for images like the first sample image where the lighting across the image is not constant by allowing for different threshold values on each pixel based on the region around it. Using one of these methods may create a better outline of the shape of the window, regardless of any leakiness around it, in case of any unevenness in the image's orientation. We would also encourage exploration of the pre-processing parameters such as the Gaussian blurring standard deviations and the thresholding value used to generate the initial contour. Of utmost priority would be fixing the `approx_four_sides()` function to check if the vertices of the contour result in intersecting lines (see appendix B(a) for more context). `FindContours()` returns an ordered list of points, so some calculation of the slopes between alternating points may be appropriate. This would return two lines between which the intersect could be calculated, and the function could continue looping through epsilon values until either the maximum epsilon is reached or the vertices do not result in intersecting lines.

Additionally, a method to adjust the temperature methods in the `bulk_contours()` function based on the scale (Fahrenheit or Celcius) of the image would contribute to the quality of data produced, and allow some flexibility in the types of images the function can receive. In line with this goal of data quality, it would also be useful to implement some check for the values generated by Tesseract are reasonable temperature values. This may mean a flag for an unusually wide range in the scale values or some threshold of raw temperatures that the scale cannot exceed.

With additional images, we would also be interested in exploring methods of identifying leakiness that do not require as many images as a neural network. Some suggestions include sectioning some region around the window contour (such as a certain number of pixels around

the contour) and taking the variability of temperature in that area. We would like to note that this should be taken as the variability in terms of the temperature of that region, not of the grayscale color value in order to gain this information on the same scale for all photos. Taking the variability of the grayscale value would result in images with a smaller temperature scale having higher variability than appropriate, since the change in temperature for an increase in grayscale units is not uniform across images. Alternatively, once the four-sided contours are fleshed out, there is potential information in the difference in area between the original contour and its four-sided approximation. It is possible that a leaky window would have a larger area difference, if the leakiness manifests as a sort of intrusion outside of the window frame. We would look forward to exploring these suggestions with more appropriate images.

Along with sample size, the limitation of this method that finds a contour and approximates a four-sided shape from it is that not being able to find a rectangular window when it is indeed in the photo might be a sign of leakiness. As the method stands now, this information would be censored, which is especially negative when the number of images to work with is already so small.

## **Conclusion**

While we did not reach a stage in which fully automated image processing was implemented, our methodologies developed and preliminary findings serve as a foundational piece to continue future research and development on. Due to time and data constraints we were unable to rigorously test and evaluate the efficacy of our process, however, going forward we recommend implementing the suggested improvements in the data collection process. We are optimistic that in continuing our work, these implementations will assist in the establishment of an automated ‘leaky window’ detection process. As we look to the future, the lessons learned from this project will undoubtedly inform and improve ongoing efforts in the field of at-home thermal image analysis, driving us closer to our overall goal of making this technology a practical tool for homeowners to assess and enhance their window insulation efficiency effectively.

## Acknowledgements

Thank you to Leah Boger and Brandon Nguy who contributed to a preliminary version of this report.

## References

“Glass and Thermal Insulation.” Saint-Gobain High Performing Glass Solutions.  
<https://www.saint-gobain-glass.co.uk/en-gb/glass-and-thermal-insulation#:~:text=The%20normal%20emissivi.02.&text=A%20surface%20will%20exchange%20heat,to%20its%20surroundings%20by%20radiation.>  
 Accessed 28 April 2024.

“How Does Emissivity Affect Thermal Imaging.” Teledyne FLIR, 1 November 2021. <https://www.flir.com/discover/professional-tools/how-does-emissivity-affect-thermal-imaging/>. Accessed 28 April 2024.

Khandelwal, Neetika. “Image Processing in Python: Algorithms, Tools, and Methods You Should Know.” neptune.ai, MLOps Blog, 25 August 2023, <https://neptune.ai/blog/image-processing-python>.

Open Source Computer Vision. “Image Processing (imgproc module).” OpenCV Tutorials, 4.10.0.dev, [https://docs.opencv.org/4.x/d7/da8/tutorial\\_table\\_of\\_content\\_imgproc.html](https://docs.opencv.org/4.x/d7/da8/tutorial_table_of_content_imgproc.html).

Tuychiev, Bex. “A Comprehensive Tutorial on Optical Character Recognition (OCR) in Python With Pytesseract.” DataCamp, April 2024, <https://www.datacamp.com/tutorial/optical-character-recognition-ocr-in-python-with-pytesseract>.

Weil, Stephen. “Tesseract Open Source OCR Engine.” Github, <https://github.com/tesseract-ocr/tesseract>.

## Appendix A

### Windows Thermal Imaging Procedure

Hi all, we are very excited to collaborate with you on this thermal imaging project! In order to get the cleanest data possible, we have created a procedure to help streamline the picture taking process.





Below is a procedure that we would appreciate you following, as well as a link to a google form that we would like you to fill out for each picture you take. While we've done our best to make this procedure accurate and easy to follow, we have much less hands-on time with these cameras than you, so please feel free to reach out with any questions or comments on how this could be smoother for you (our emails are on the top right corner).

Important notes:

- Only take pictures of windows
  - Try to take pictures of windows from as many different buildings as possible (on and off campus). We only need one picture of one window for each building
- Only take pictures of windows that fit fully in frame when standing 1-6 ½ feet away
- Take picture as straight-on as possible (try to avoid taking from an angle)
- Only take pictures when outside temperature is above 14
  - The camera will only read surface temperatures down to -4 , which will be an issue if the photos are not sufficiently detailed (you will notice a large part of the photo that is one color, even though you believe it should be many different colors)
- Only take pictures from outside

#### Camera Set-Up

**Steps:**

- 1 In the live view interface, press  to show the menu bar.
- 2 Press / to select desired setting bar.
- 3 Press  to go to the setting interface.

- Meas. Range: put in Auto Switch mode
- Distance: estimate distance between you and your target and enter here
- Rule: select Hot Spot, Cold Spot

- Unit: Fahrenheit

Procedure:



**Steps:**

1. In the live view interface, pull the trigger to capture snapshot. The live view freezes

4



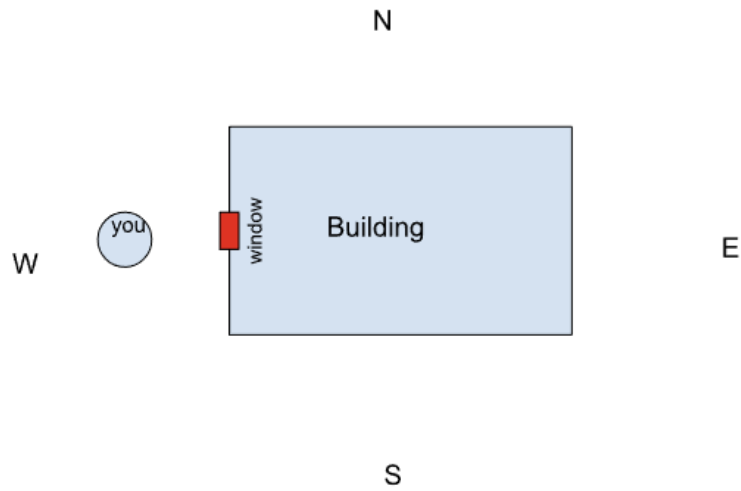
and device displays the snapshot with temperature information.

2. Press  to save the picture, press  to cancel it.
- 1.
2. Fill out google form with proper information (fill out immediately, as a lot of the information is time sensitive)

Google Form

Click the link below to fill out a form for each picture you take. Below are detailed notes for some of the pieces of information we are asking for, but feel free to reach out with any questions at all.

- Longitude/Latitude Apple maps makes this easy, simply hold down on the location you are at until a pin is dropped. Click the image that pops up, and scroll down to details. Under coordinates, longitude and latitude should be displayed. Please try to be as accurate as possible here
- Ordinal Direction Please record the direction that the window is FACING. Pick the most accurate answer possible. For example, for the window below, you would record “West” as the direction the window is facing



## Appendix B

### Known Issues

a) `approx_four_sides()`

Due to the binarization of the image before identifying the contour, if the pane of a window is not uniformly colored, the contour may not identify the whole window based on the threshold values.

An example of this can be seen in Figure 4, where the contour indents into the middle of the window because of the angle of the image. This can result in issues when we try to approximate a four-sided contour, as seen in Figure 5, where although the contour is defined by four edges, those four edges do not form an open figure.

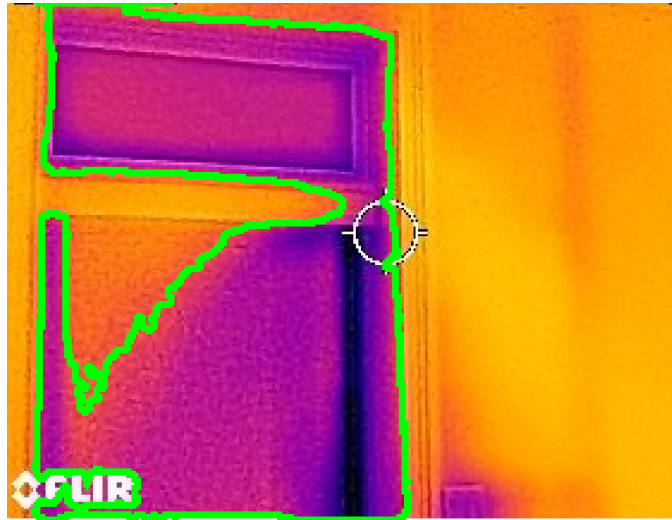


Figure 4: Resulting Contour of an Off-Axis Image



Figure 5: Resulting Four Sided Contour of an Off-Axis Image