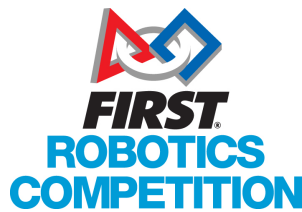


# GRIP - GRAPHICAL IMAGE PROCESSING



# Table of Contents

Articles ..... 3

GRIP Alpha Preview ..... 4

Processing images from the FRC 2014 game ..... 15

Processing images from the 2009 game ..... 18

Reading array values published by NetworkTables ..... 19

Using GRIP for the 2016 game ..... 23

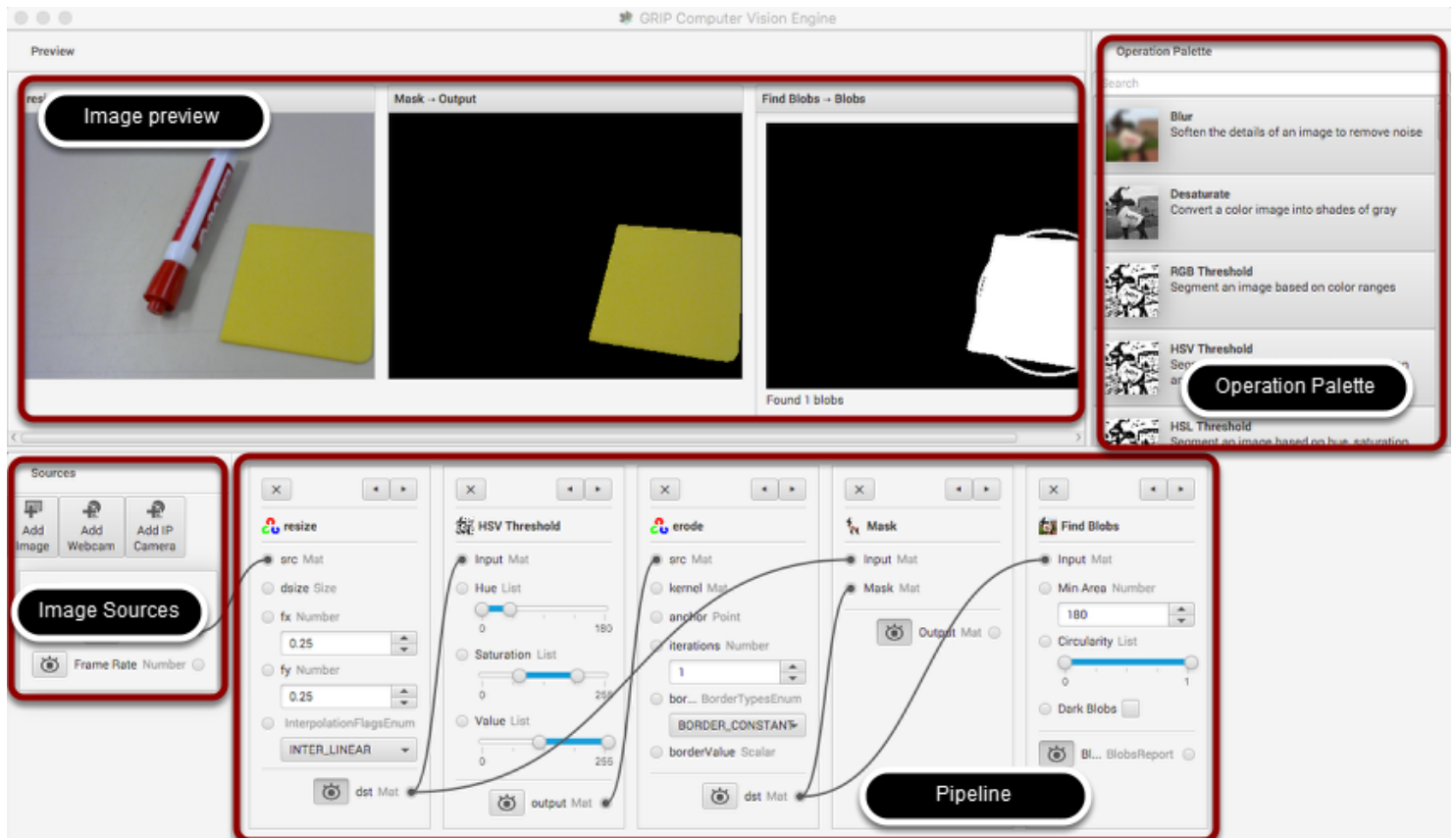
# Articles

# GRIP Alpha Preview

GRIP is a tool for developing computer vision algorithms interactively rather than through trial and error coding. After developing your algorithm you may run GRIP in headless mode on your roboRIO, on a Driver Station Laptop, or on a coprocessor connected to your robot network. With Grip you choose vision operations to create a graphical pipeline that represents the sequence of operations that are performed to complete the vision algorithm.

GRIP is based on OpenCV, one of the most popular computer vision software libraries used for research, robotics, and vision algorithm implementations. The operations that are available in GRIP are almost a 1 to 1 match with the operations available if you were hand coding the same algorithm with some text-based programming language.

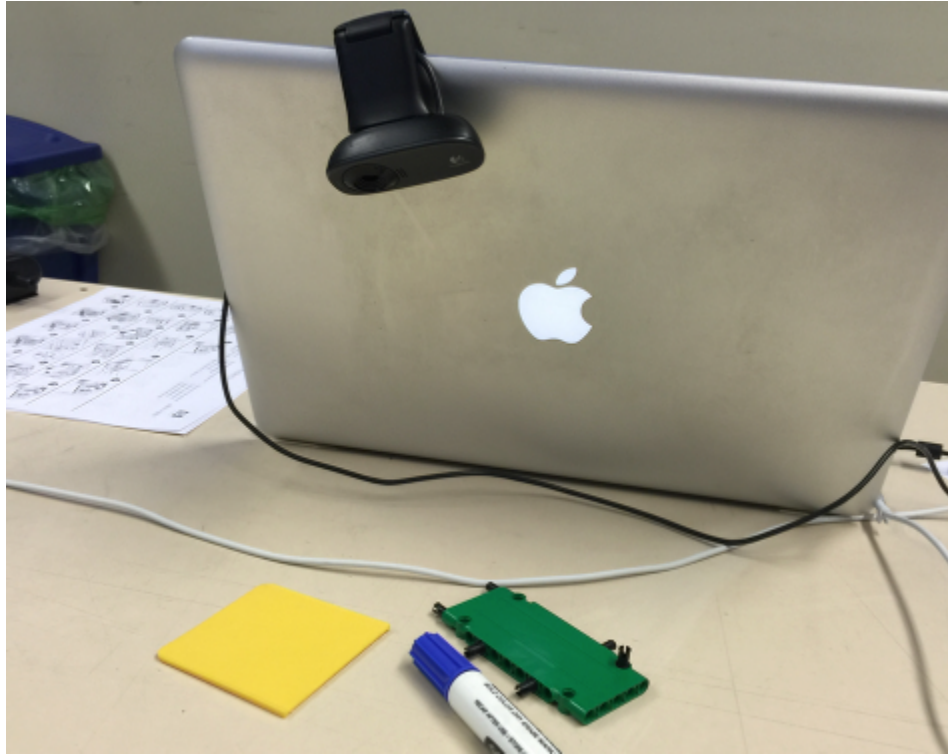
## The GRIP user interface



The GRIP user interface consists of 4 parts:

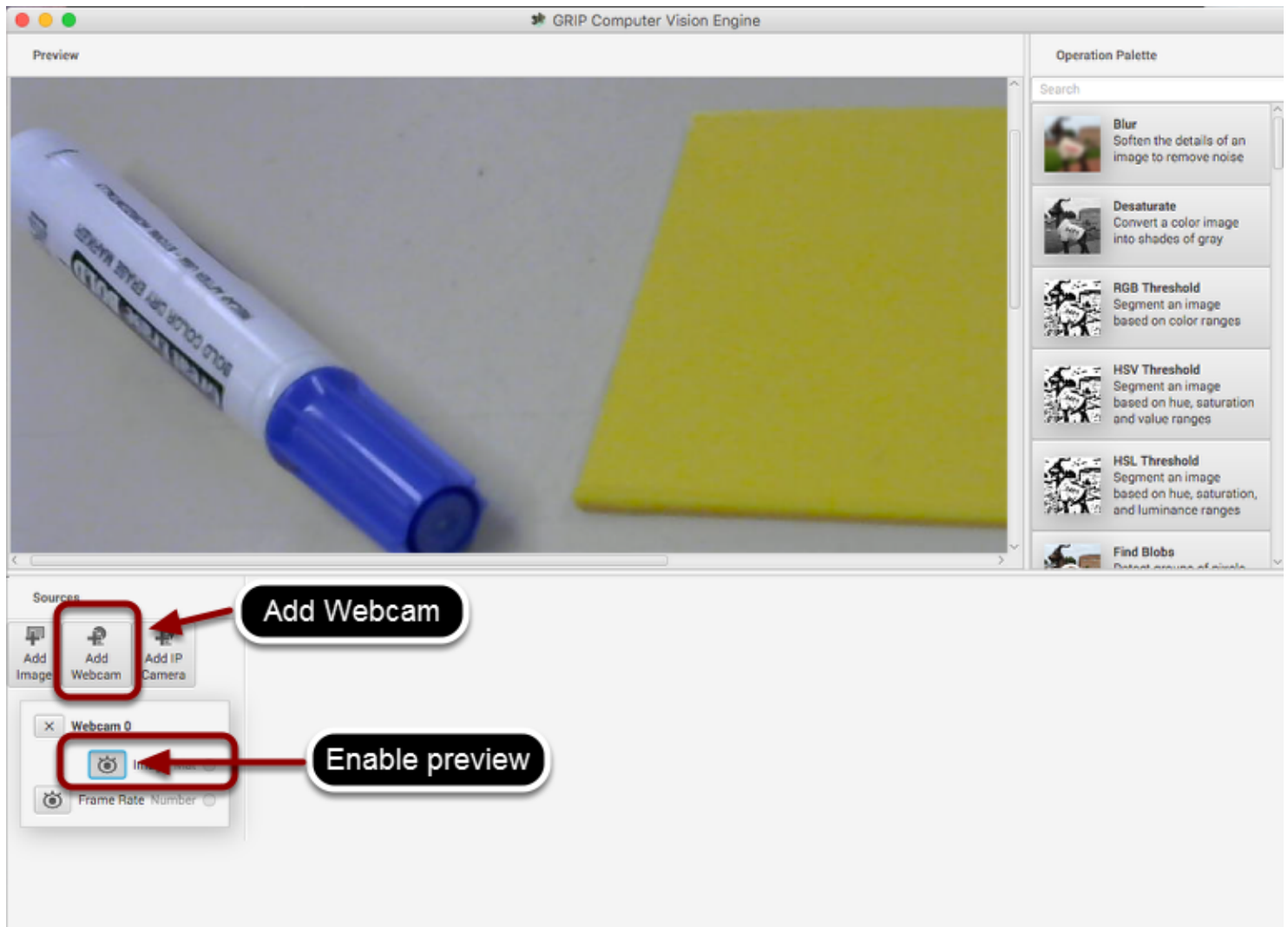
- **Image Sources** are the ways of getting images into the GRIP pipeline. You can provide images through attached cameras or files. Sources are almost always the beginning of the image processing algorithm.
- **Operation Palette** contains the image processing steps from the OpenCV library that you can chain together in the pipeline to form your algorithm. Clicking on an operation in the palette adds it to the end of the pipeline. You can then use the left and right arrows to move the operation within the pipeline.
- **Pipeline** is the sequence of steps that make up the algorithm. Each step (operation) in the pipeline is connected to a previous step from the output of one step to an input to the next step. The data flows from generally from left to right through the connections that you create.
- **Image Preview** are shows previews of the result of each step that has it's preview button pressed. This makes it easy to debug algorithms by being able to preview the outputs of each intermediate step.

## Finding the yellow square



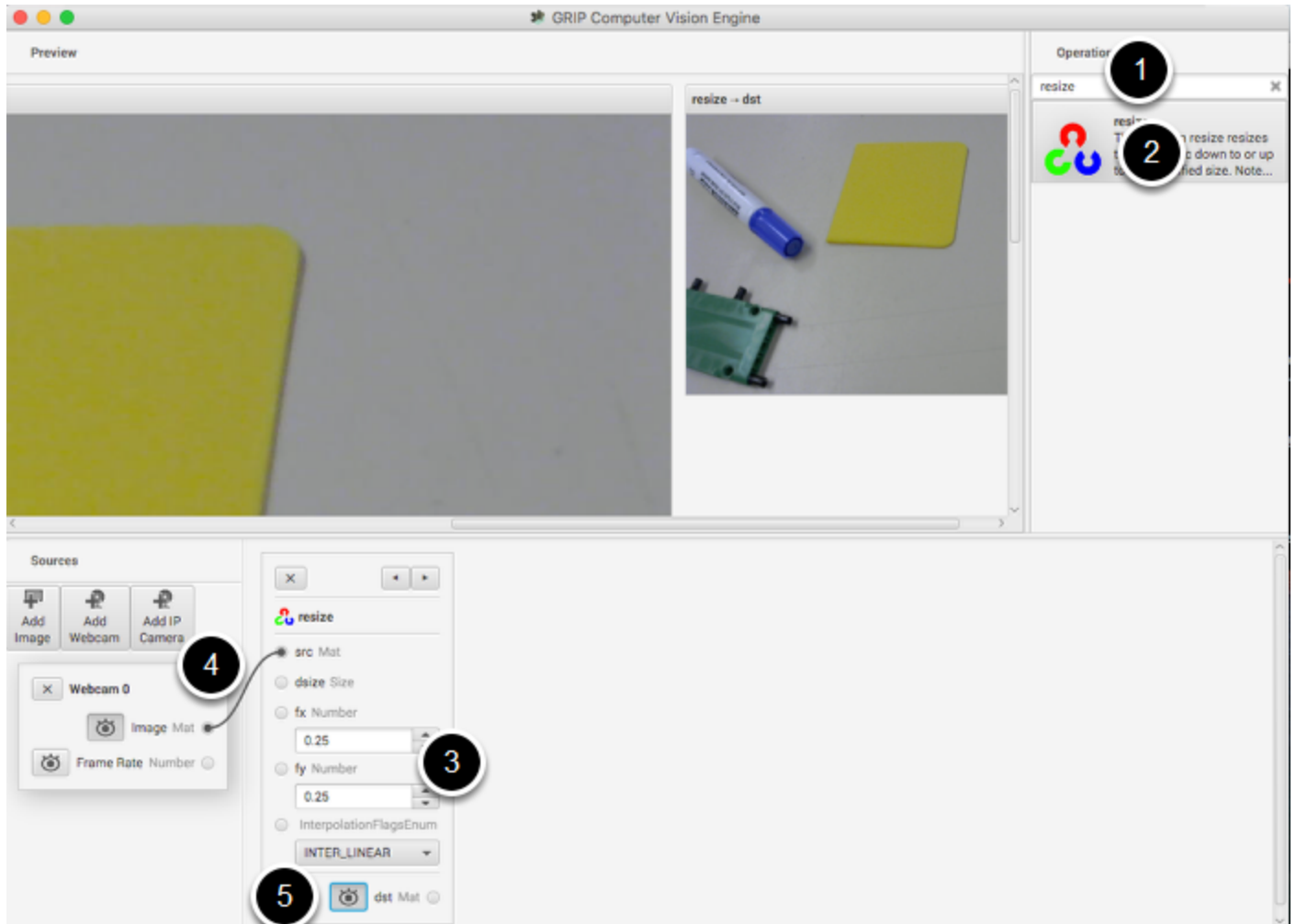
In this application we will try to find the yellow square in the image and display its position. The setup is pretty simple, just a USB web camera connected to the computer looking down at some colorful objects. The yellow plastic square is the thing that we're interested in locating in the image.

## Enable the image source



The first step is to acquire an image. To use the source, click on the "Add Webcam" button and select the camera number. In this case the Logitech USB camera that appeared as Webcam 0 and the computer monitor camera was Webcam 1. The web camera is selected in this case to grab the image behind the computer as shown in the setup. Then select the image preview button and the real-time display of the camera stream will be shown in the preview area.

## Resize the image



In this case the camera resolution is too high for our purposes, and in fact the entire image cannot even be viewed in the preview window. The "Resize" operation is clicked from the Operation Palette to add it to the end of the pipeline. To help locate the Resize operation, type "Resize" into the search box at the top of the palette. The steps are:

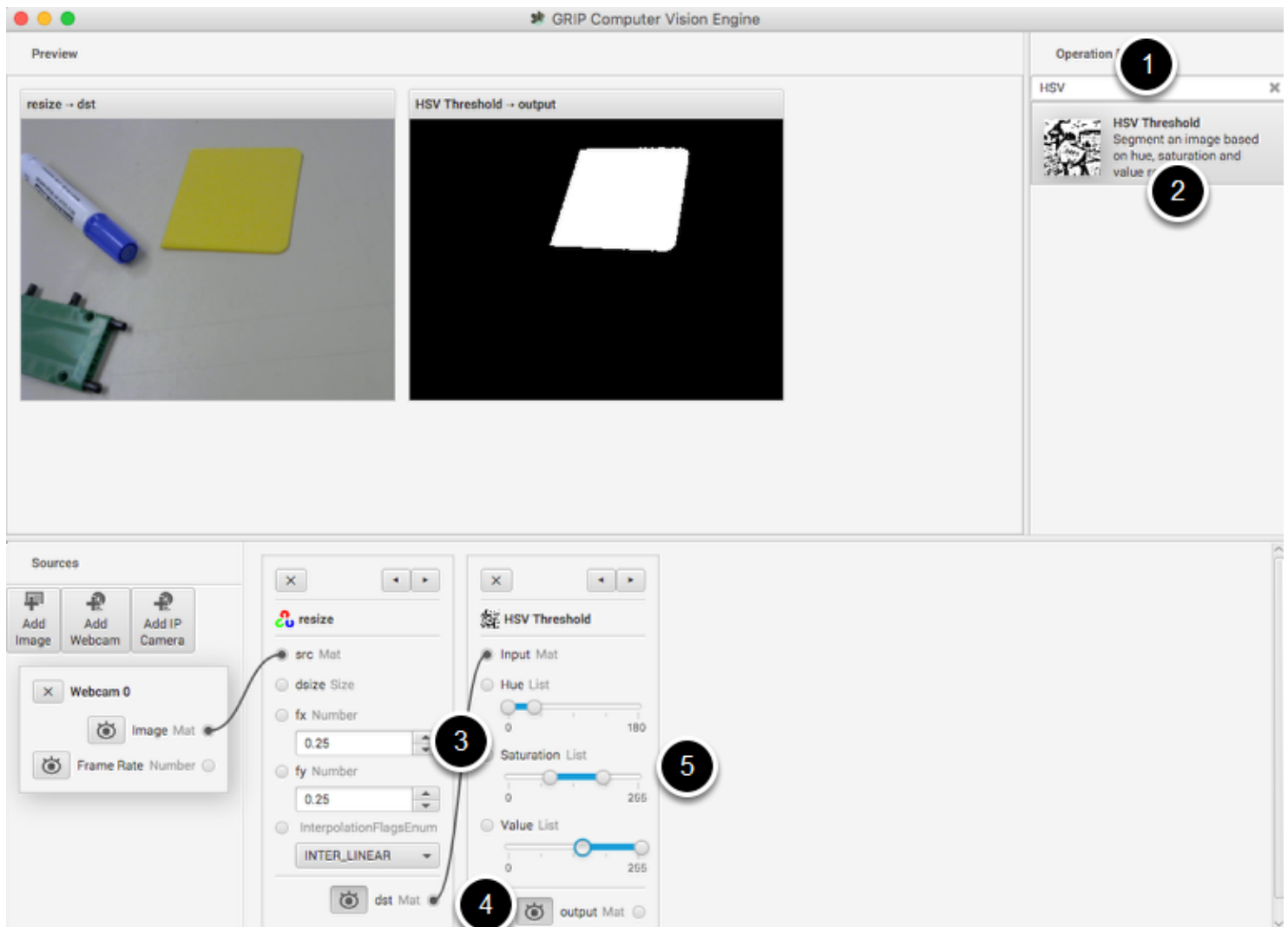
1. Type "Resize" into the search box on the palette
2. Click the Resize operation from the palette. It will appear in the pipeline.
3. Enter the x and y resize scale factor into the resize operation in the pipeline. In this case 0.25 was chosen for both.
4. Drag from the Webcam image output mat socket to the Resize image source mat socket. A connection will be shown indicating that the camera output is being sent to the resize input.



# GRIP - Graphical Image Processing

5. Click on the destination preview button on the "Resize" operation in the pipeline. The smaller image will be displayed alongside the larger original image. You might need to scroll horizontally to see both as shown.
6. Lastly, click the Webcam source preview button since there is no reason to look at both the large image and the smaller image at the same time.

## Find only the yellow parts of the image

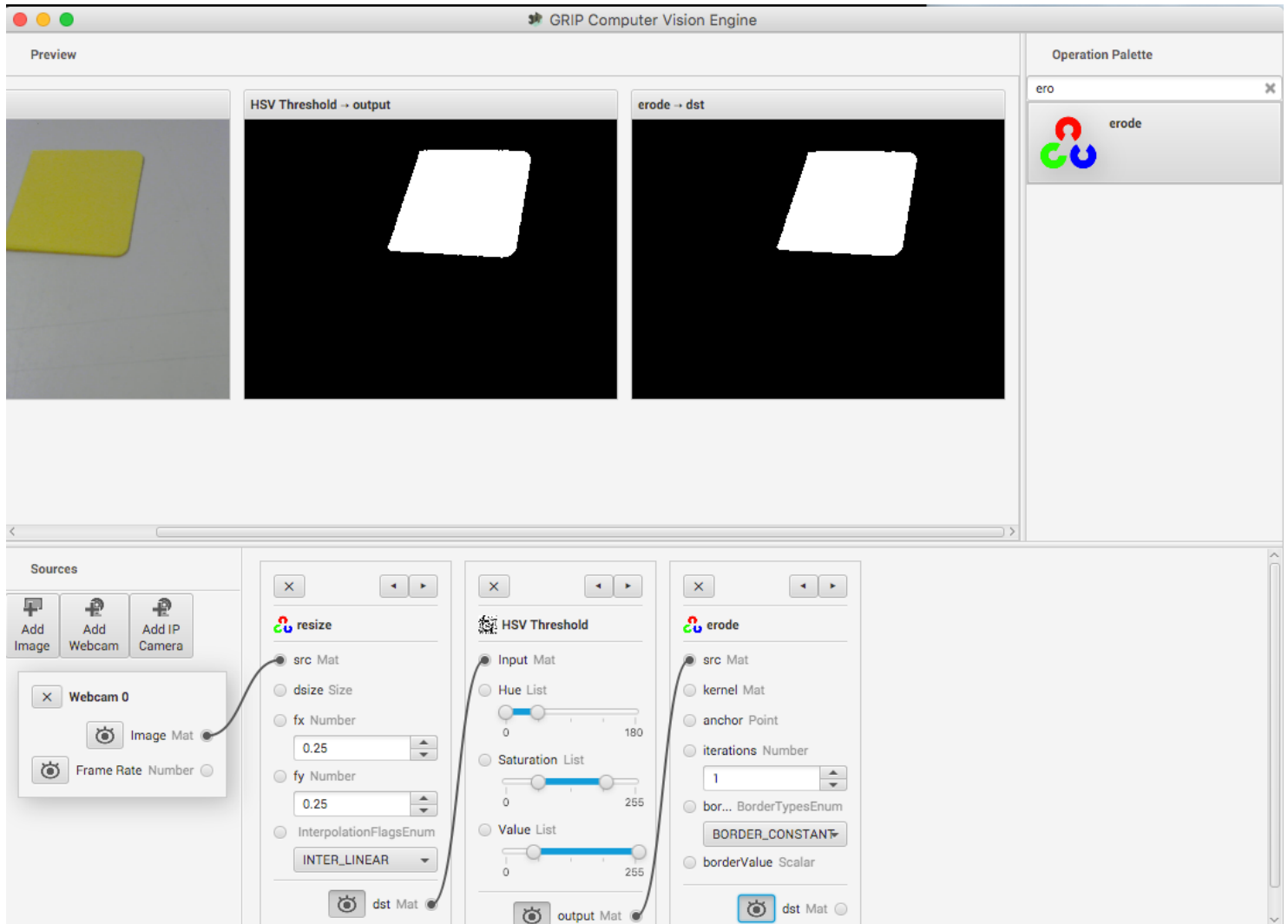


The next step is to remove everything from the image that doesn't match the yellow color of the piece of plastic that is the object being detected. To do that a HSV Threshold operation is chosen to set upper and lower limits of HSV values to indicate which pixels should be included in the resultant binary image.

Notice that the target area is white while everything that wasn't within the threshold values are shown in black. Again, as before:

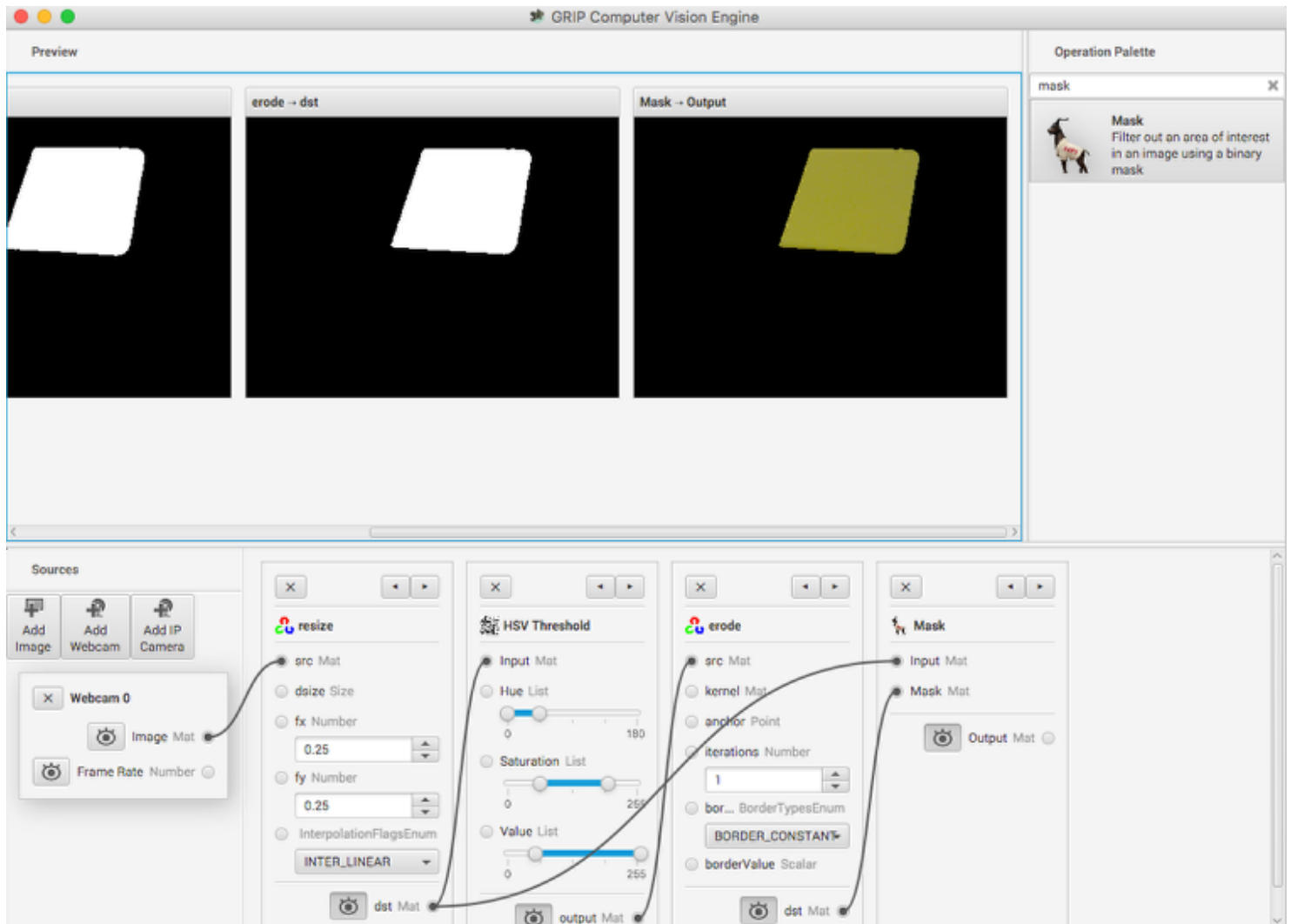
1. Type HSV into the search box to find the HSV Threshold operation.
2. Click on the operation in the palette and it will appear at the end of the pipeline.
3. Connect the dst (output) socket on the resize operation to the input of the HSV Threshold.
4. Enable the preview of the HSV Threshold operation so the result of the operation is displayed in the preview window.
5. Adjust the Hue, Saturation, and Value parameters only the target object is shown in the preview window.

## Get rid of the noise and extraneous hits



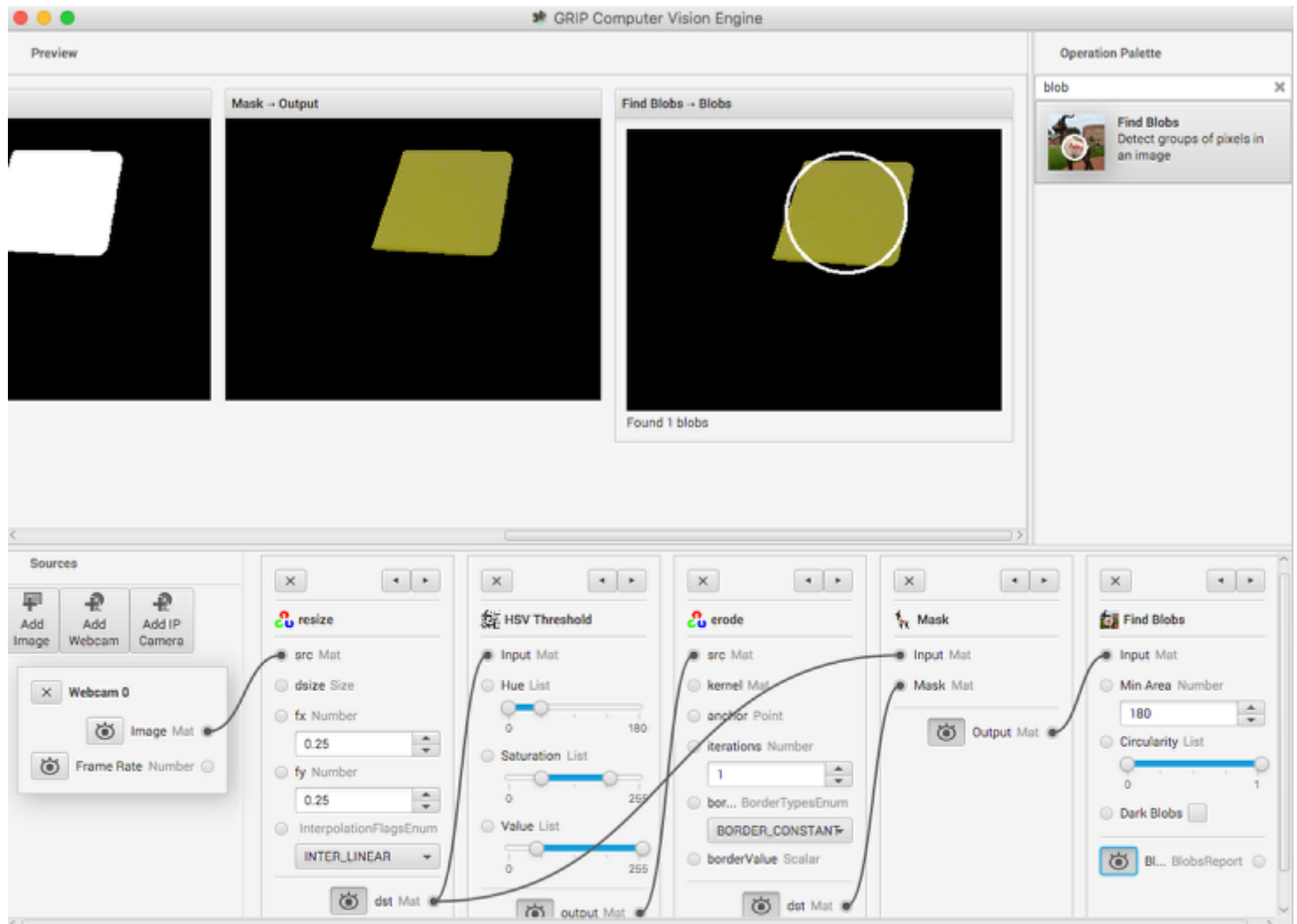
This looks pretty good so far, but sometimes there is noise from other things that couldn't quite be filtered out. To illustrate one possible technique to reduce those occasional pixels that were detected, an Erosion operation is chosen. Erosion will remove small groups of pixels that are not part of the area of interest.

## Mask the just the yellow area from the original image



Here a new image is generated by taking the original image and masking (and operation) it with the the results of the erosion. This leaves just the yellow card as seen in the original image with nothing else shown. And it makes it easy to visualize exactly what was being found through the series of filters.

## Find the yellow area (blob)



The last step is actually detecting the yellow card using a Blob Detector. This operation looks for a grouping of pixels that have some minimum area. In this case, the only non-black pixels are from the yellow card after the filtering is done. You can see that a circle is drawn around the detected portion of the image. In the release version of GRIP (watch for more updates between now and kickoff) you will be able to send parameters about the detected blob to your robot program using Network Tables.

## Status of GRIP

As you can see from this example, it is very easy and fast to be able to do simple object recognition using GRIP. While this is a very simple example, it illustrates the basic principles of using GRIP and feature extraction in general. Over the coming weeks the project team will be posting updates to GRIP as more features are added. Currently it supports cameras (Axis ethernet camera and web cameras) and image inputs. There is no provision for output yet although Network Tables and ROS (Robot Operating System) are planned.

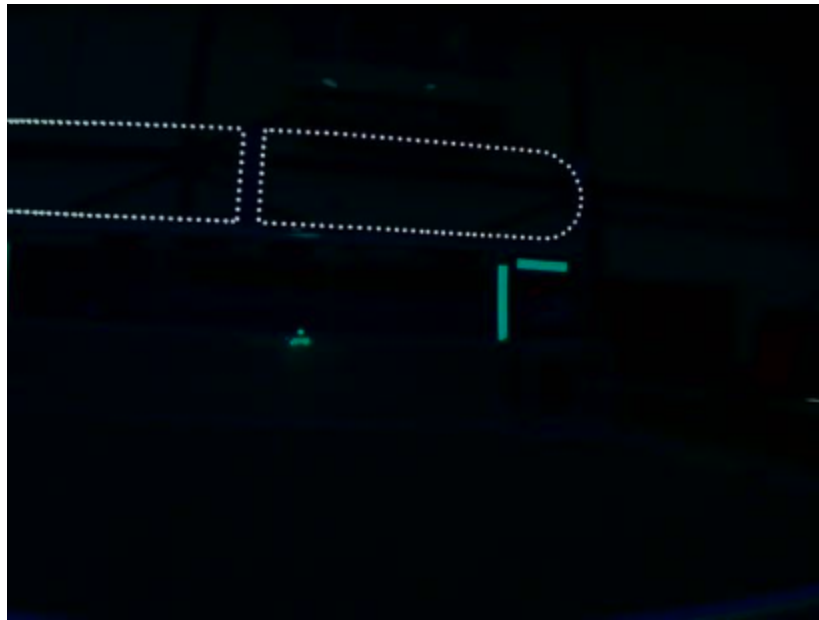
You can either download a pre-built release of the code from the github page "Releases" section (<https://github.com/WPIRoboticsProjects/GRIP>) or you can clone the source repository and built it yourself. Directions on building GRIP are on the project page. There is also additional documentation on the project wiki.

So, please play with GRiP and give us feedback here on the forum. If you find bugs, you can either post them here or as a Github project issue on the project page.

# Processing images from the FRC 2014 game

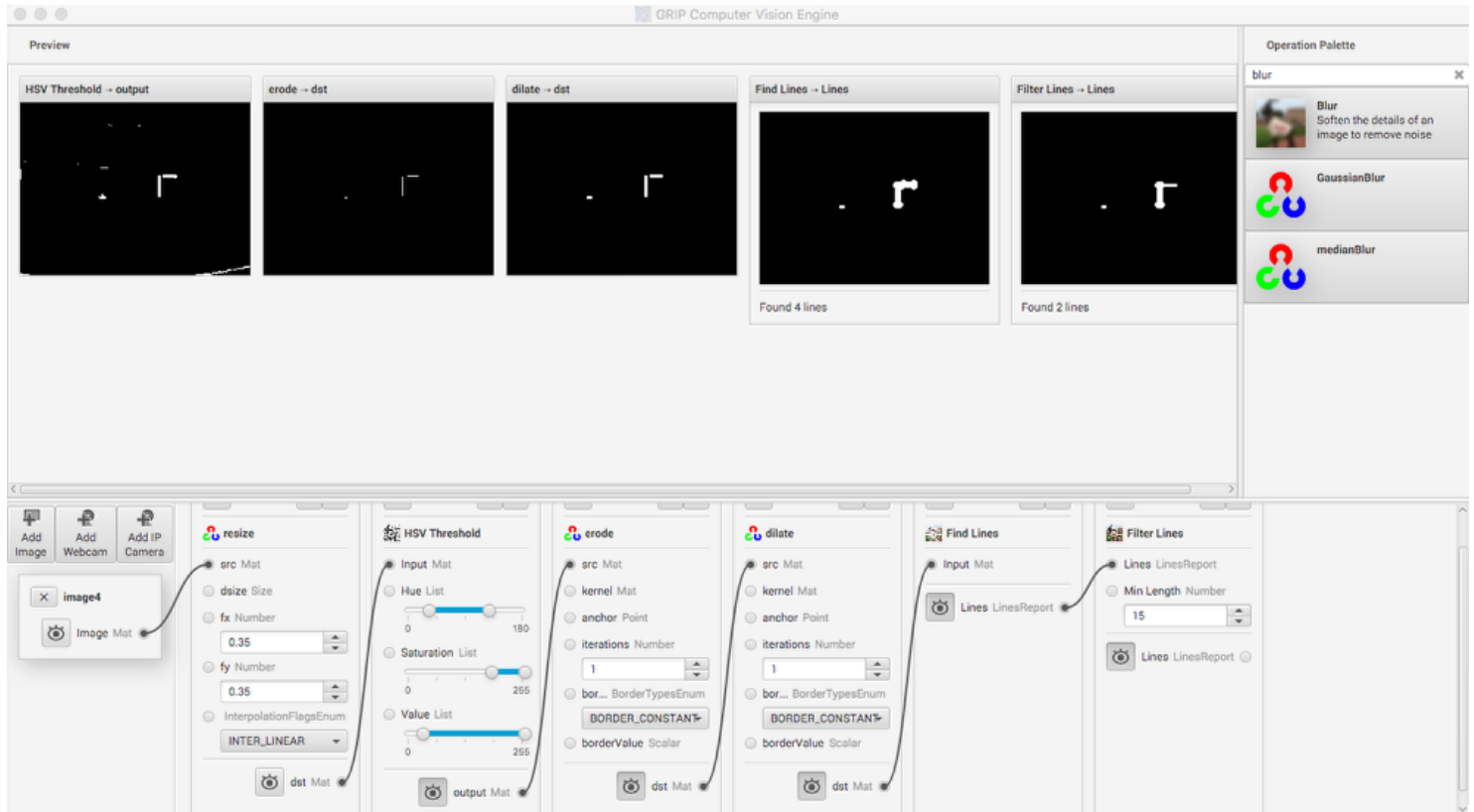
This is a quick sample using GRIP to process a single image from the FRC 2014 Aerial Assist. Keep in mind that this is just a single image (all that I could find in a hurry) so it is not necessarily a particularly robust algorithm. You should really take many pictures from different distances, angles, and lighting conditions to ensure that your algorithm will perform well in all those cases.

## The sample image (I'll get more later)



The original image is shown here and represents the hot goal being indicated by the retroreflective tape on the moveable board. When the goal is hot, the horizontal piece of tape is facing the robot. When it is not hot, the board that the tape is attached is flipped up so it doesn't reflect and that horizontal line would not be present.

## The actual algorithm depicted in GRIP



This shows the pipeline as it was developed in GRIP. It took only a few minutes to do this and was the first attempt:

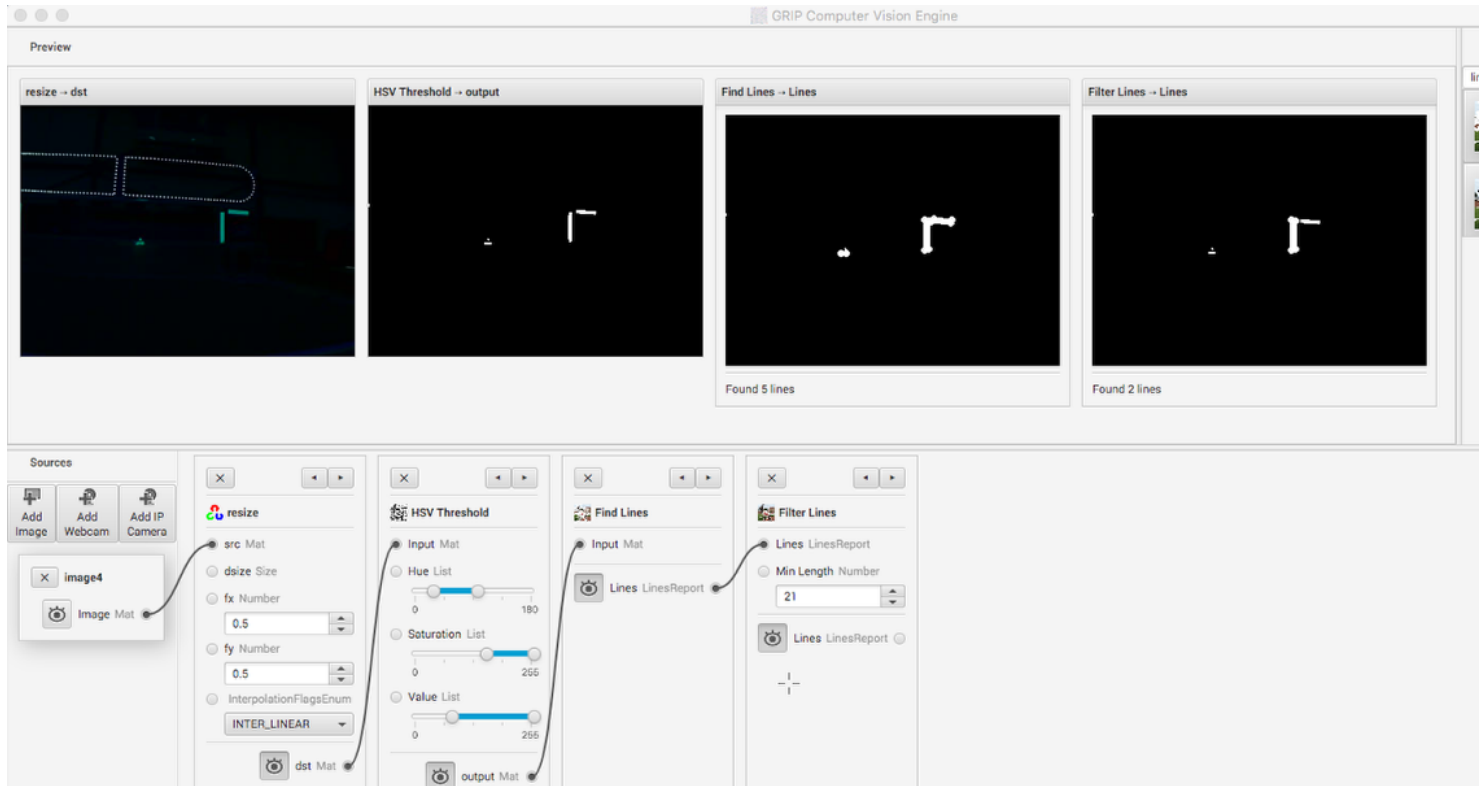
1. The image was resized by 0.35 to make it fit better for this presentation.
2. A HSV threshold operation is done to filter out everything but the blue-green lines that are shown in the initial image.
3. An erosion operation was done to reduce a bunch of the small artifacts that the threshold was not able to filter. This also reduced the line thickness.
4. A dilation is done to make the lines a little thicker in hopes of better detecting them, although this might not have been necessary.
5. Then a line detection operation found 4 lines in this case, the few artifacts registered as lines as well as the actual lines.
6. The smaller artifacts were filtered using the Filter Lines operation as shown in the final step. Filter lines allows the specification of the minimum line size so the extra mis-detected lines are removed from the list.



# GRIP - Graphical Image Processing

In the final release, the output of the filter lines operation can be sent to network tables to make it accessible to the robot program to decide whether or not to take the shot.

## A simpler approach

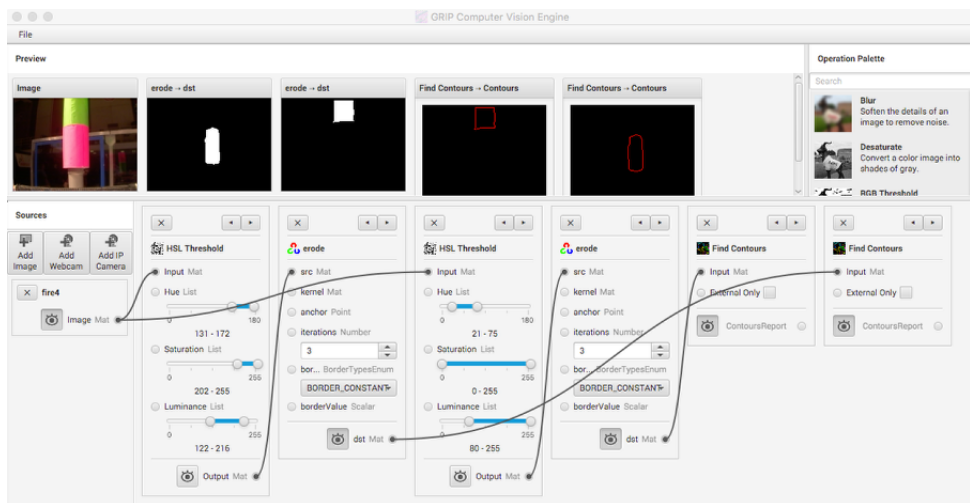


It seemed like the erosion and dilation might not be necessary since there was very little noise showing up in the images. So for this image (again - in real life test this with more images), the same algorithm was attempted successfully without those extra steps. In this case the algorithm is shorter and would run more quickly, but might not be as robust as the one with the potentially better filtering. And again, it also found 2 lines representing the hot goal targets.

## Processing images from the 2009 game

In the FRC 2009 game, Lunacy, robots were required to put orbit balls into the trailers of opponents robots. To differentiate robots on each of the two alliances, a "flag" was attached to the center of the goal. The flag was a cylinder that was green on top and red on the bottom or red on top with green on the bottom. During the autonomous period, robots could look for opponent robots and shoot balls into their trailer using the vision targets.

## Using the Find Contours operation to find targets



In this example the image, in this case a file, you can see the green and red halves of the vision target. The strategy is to:

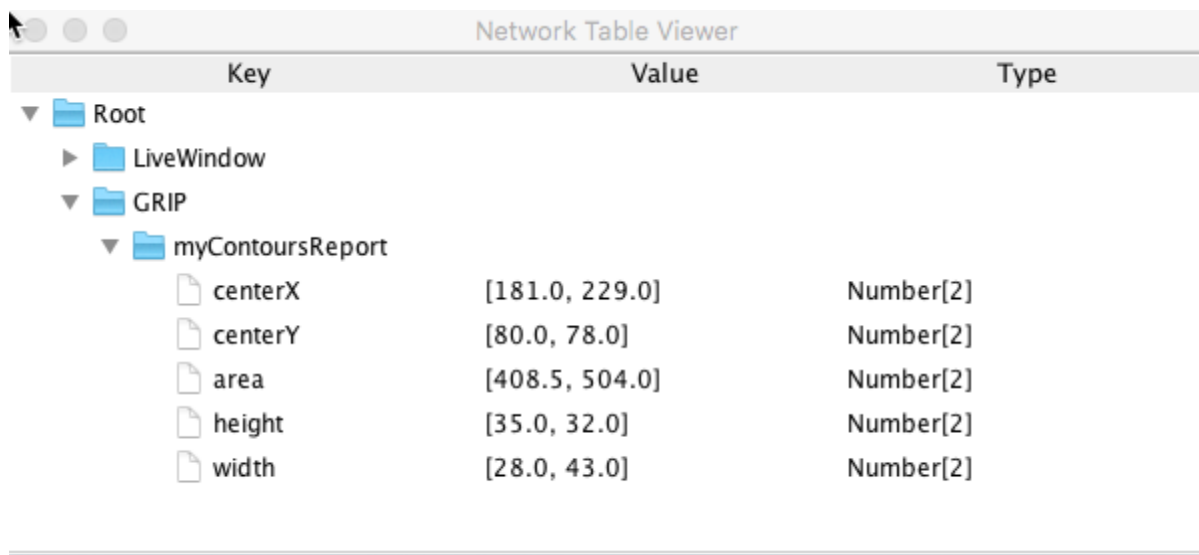
1. Look for objects that are either green or red using two different HSL Threshold operations, one for each color. Setting the appropriate parameters on the threshold allow it to detect the two halves of the target.
2. Erode the images to get rid of any very small objects that slipped through
3. Find contours in each of the resultant binary images
4. Send the red and green contours lists to the robot, It will look for objects with the correct aspect ratio, proximity, and orientation with respect to each other. From this the robot program can determine which sets are targets.

# Reading array values published by NetworkTables

This article describes how to read values published by NetworkTables using a program running on the robot. This is useful when using computer vision where the images are processed on your driver station laptop and the results stored into NetworkTables possibly using a separate vision processor like a raspberry pi, or a tool on the robot like GRIP, or a python program to do the image processing.

Very often the values are for one or more areas of interest such as goals or game pieces and multiple instances are returned. In the example below, several x, y, width, height, and areas are returned by the image processor and the robot program can sort out which of the returned values are interesting through further processing.

## Verify the network table keys being published



The screenshot shows the 'Network Table Viewer' application. On the left is a tree view with the following structure:

- Root
  - LiveWindow
    - GRIP
      - myContoursReport
        - centerX
        - centerY
        - area
        - height
        - width

On the right is a table with three columns: Key, Value, and Type.

| Key     | Value          | Type      |
|---------|----------------|-----------|
| centerX | [181.0, 229.0] | Number[2] |
| centerY | [80.0, 78.0]   | Number[2] |
| area    | [408.5, 504.0] | Number[2] |
| height  | [35.0, 32.0]   | Number[2] |
| width   | [28.0, 43.0]   | Number[2] |

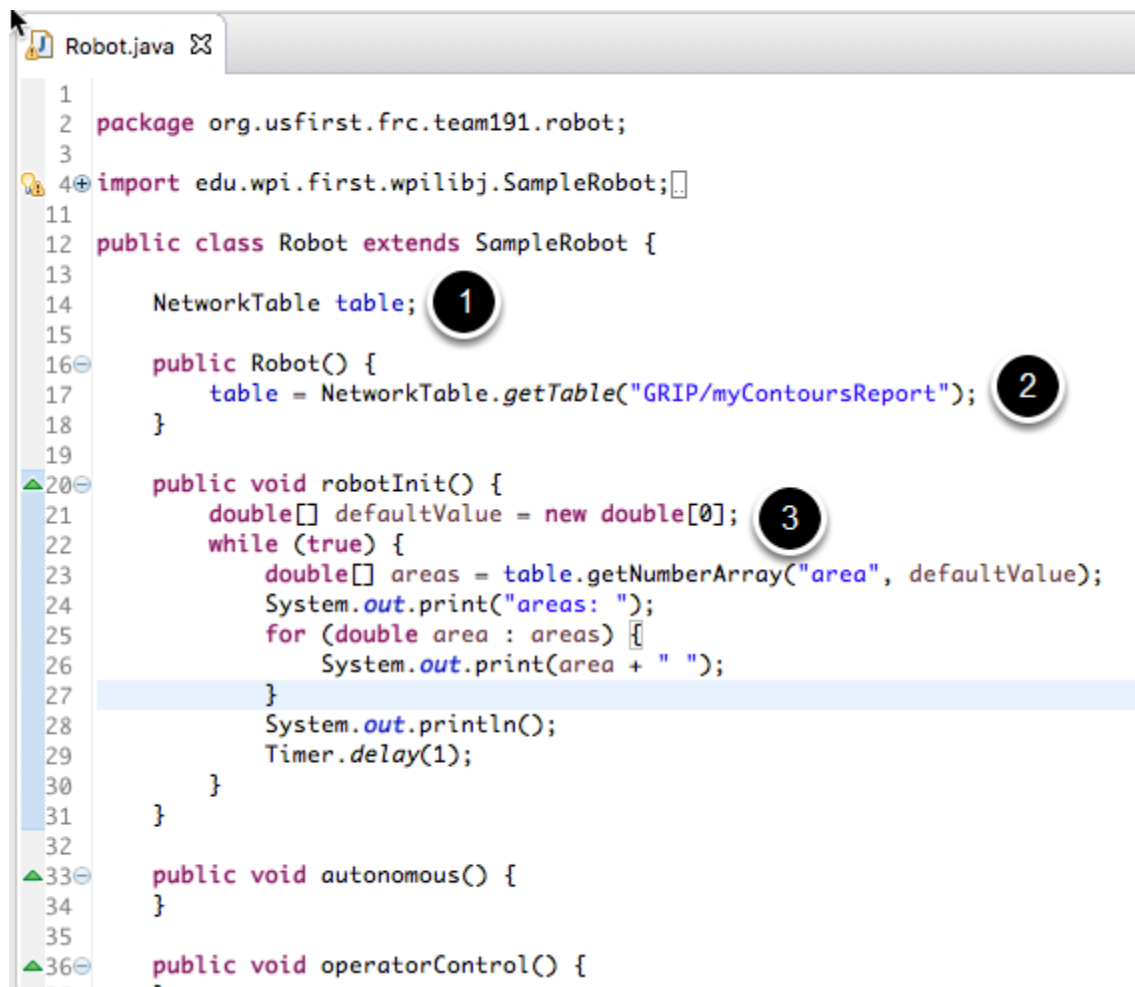
You can verify the names of the network table keys used for publishing the values by using the Network Table Viewer application. It is a java program in your user directory in the wpilib/tools folder. The application is started by selecting the "WPILib" menu in eclipse then "OutlineViewer". In this example, with the image processing program running (GRIP) you can see the values being put into NetworkTables.

# GRIP - Graphical Image Processing

In this case the values are stored in a table called GRIP and a sub-table called myContoursReport. You can see that the values are in brackets and there are 2 values in this case for each key. The network table key names are centerX, centerY, area, height and width.

*Both of the following examples are extremely simplified programs that just illustrate the use of NetworkTables. All the code is in the robotInit() method so it's only run when the program starts up. In your programs, you would more likely get the values in code that is evaluating which direction to aim the robot in a command or a control loop during the autonomous or teleop periods.*

## Writing a Java program to access the keys



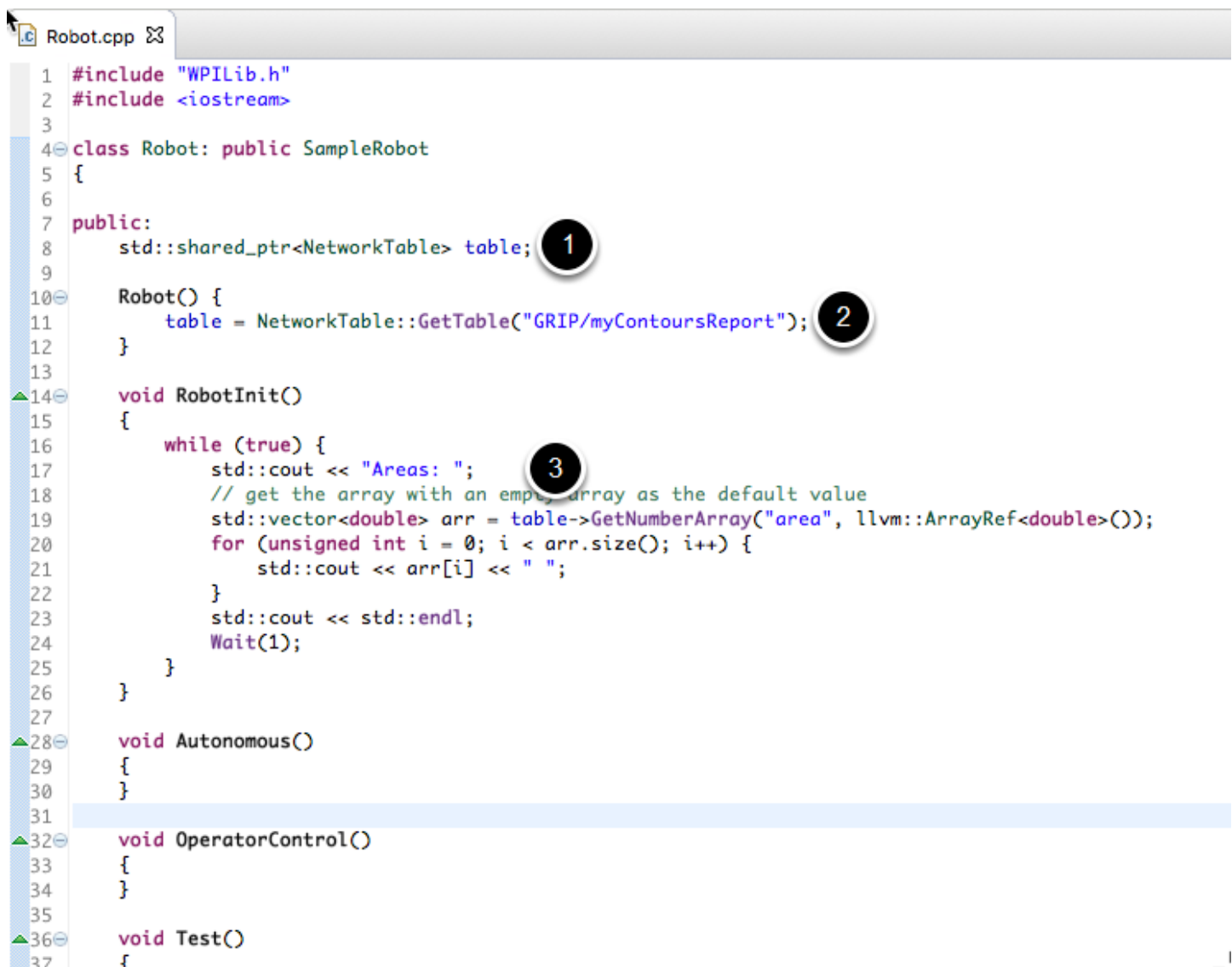
```
1 package org.usfirst.frc.team191.robot;
2
3
4 import edu.wpi.first.wpilibj.SampleRobot;
5
6
7
8
9
10
11 public class Robot extends SampleRobot {
12
13     NetworkTable table; 1
14
15     public Robot() {
16         table = NetworkTable.getTable("GRIP/myContoursReport"); 2
17     }
18
19     public void robotInit() {
20         double[] defaultValue = new double[0]; 3
21         while (true) {
22             double[] areas = table.getNumberArray("area", defaultValue);
23             System.out.print("areas: ");
24             for (double area : areas) {
25                 System.out.print(area + " ");
26             }
27             System.out.println();
28             Timer.delay(1);
29         }
30     }
31
32     public void autonomous() {
33     }
34
35     public void operatorControl() {
36     }
```

The steps to getting the values and, in this program, printing them are:

1. Declare the table variable that will hold the instance of the subtable that have the values.

2. Initialize the subtable instance so that it can be used later for retrieving the values.
3. Read the array of values from NetworkTables. In the case of a communicating programs, it's possible that the program producing the output being read here might not yet be available when the robot program starts up. To avoid issues of the data not being ready, a default array of values is supplied. This default value will be returned if the network table key hasn't yet been published. This code just loops forever and reads values and prints them to the console.

## Writing a C++ program to access the keys



```
1 #include "WPIlib.h"
2 #include <iostream>
3
4 class Robot: public SampleRobot
5 {
6
7 public:
8     std::shared_ptr<NetworkTable> table; 1
9
10    Robot() {
11        table = NetworkTable::GetTable("GRIP/myContoursReport"); 2
12    }
13
14    void RobotInit()
15    {
16        while (true) { 3
17            std::cout << "Areas: ";
18            // get the array with an empty array as the default value
19            std::vector<double> arr = table->GetNumberArray("area", llvm::ArrayRef<double>());
20            for (unsigned int i = 0; i < arr.size(); i++) {
21                std::cout << arr[i] << " ";
22            }
23            std::cout << std::endl;
24            Wait(1);
25        }
26    }
27
28    void Autonomous()
29    {
30    }
31
32    void OperatorControl()
33    {
34    }
35
36    void Test()
37    {
```

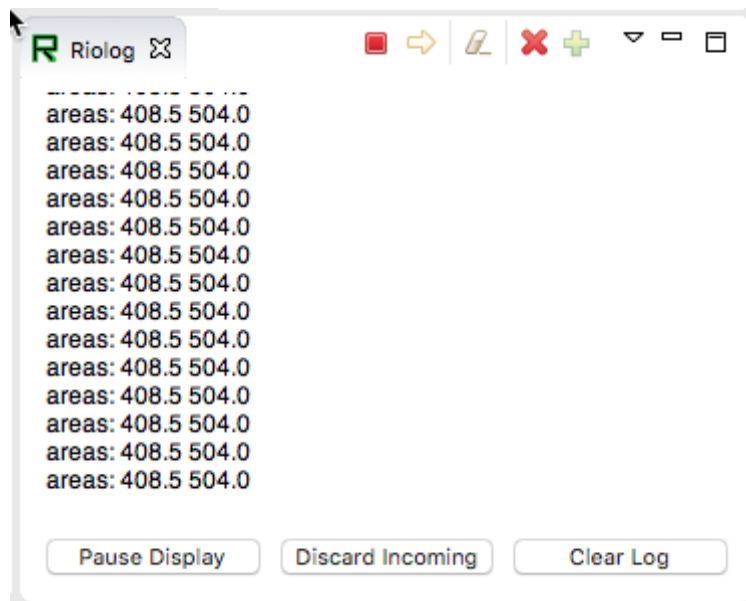
The steps to getting the values and, in this program, printing them are:

# GRIP - Graphical Image Processing

---

1. Declare the table variable that will hold the instance of the subtable that have the values. It is a shared pointer where the library takes care of allocation and deallocation automatically.
2. Initialize the subtable instance so that it can be used later for retrieving the values.
3. Read the array of values from NetworkTables. In the case of a communicating programs, it's possible that the program producing the output being read here might not yet be available when the robot program starts up. To avoid issues of the data not being ready, a default array of values is supplied. `llvm::ArrayRef<double>` creates this temporary array reference of zero length that would be returned if the network table key hasn't yet been published. This code just loops forever and reads values and prints them to the console.

## Program output

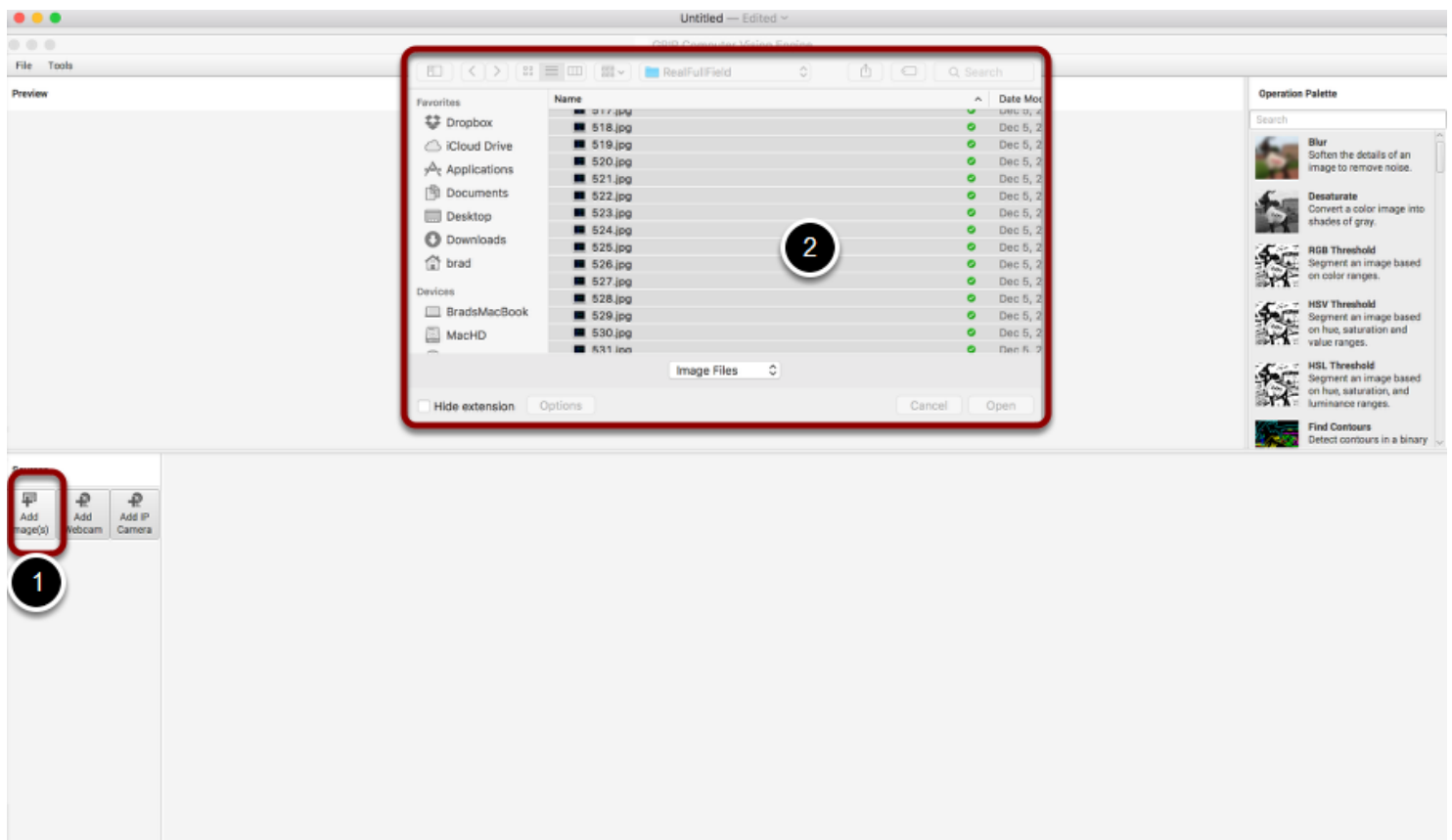


In this case the program is only looking at the array of areas, but in a real example all the values would more likely be used. Using the Riolog in eclipse or the DriverStation log you can see the values as they are retrieved. This program is using a sample static image so they areas don't change, but you can imagine with a camera on your robot, the values would be changing constantly.

## Using GRIP for the 2016 game

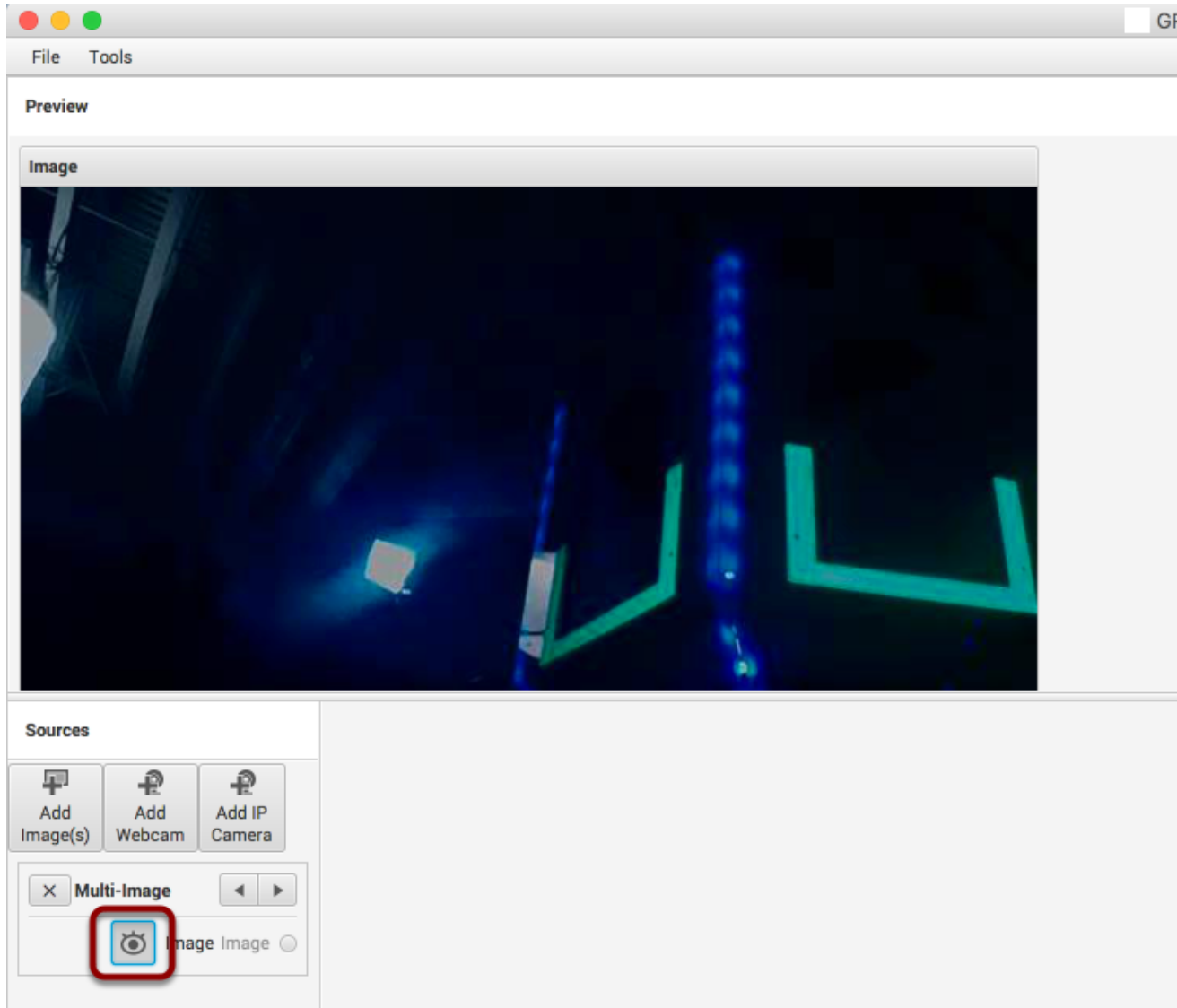
GRIP can be used to locate goals for the FIRST Stronghold by using a series of thresholding and finding contours. This page describes the algorithm that was used. You should download the set of sample images from the WPILib project on <http://usfirst.collab.net>. The idea is to load up all the images into a multi image source and test your algorithms by cycling through the pictures.

### Select all the vision samples as multi image source



Click the Add images button to create the multi-image source, then select all the images from the directory where they were unzipped.

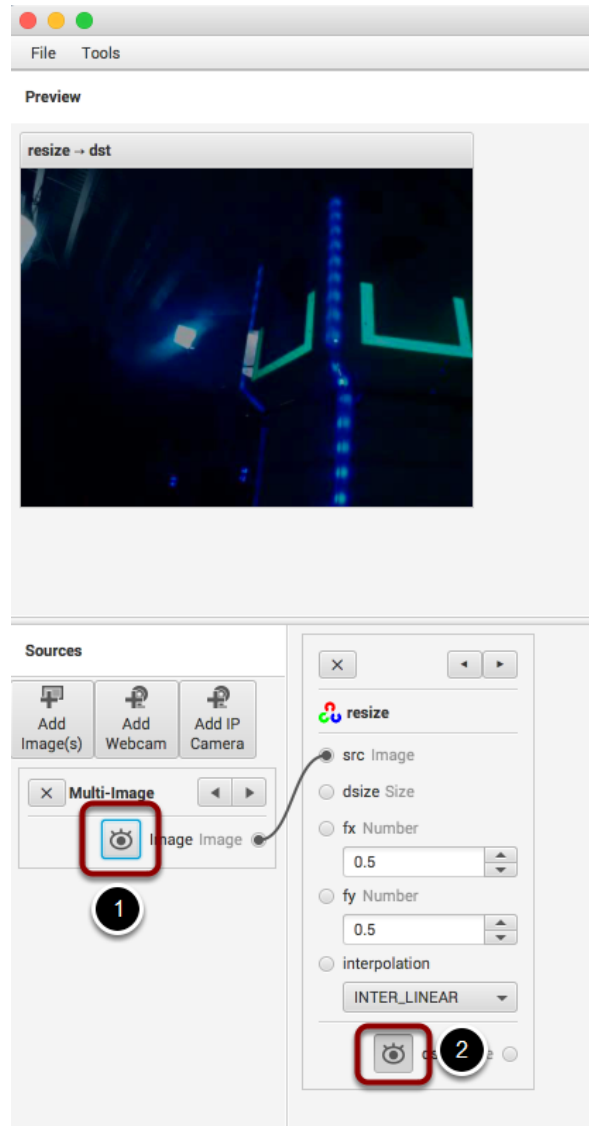
## Images are too big (maybe)



I decided that in this example the images were too big and would work just as well if they were resized. That may or may not be true, but it made the pictures small enough to capture easily for this set of examples. You can decide based on your processor and experience whether you want to resize them or not. The images are previewed by pressing the preview button shown above.

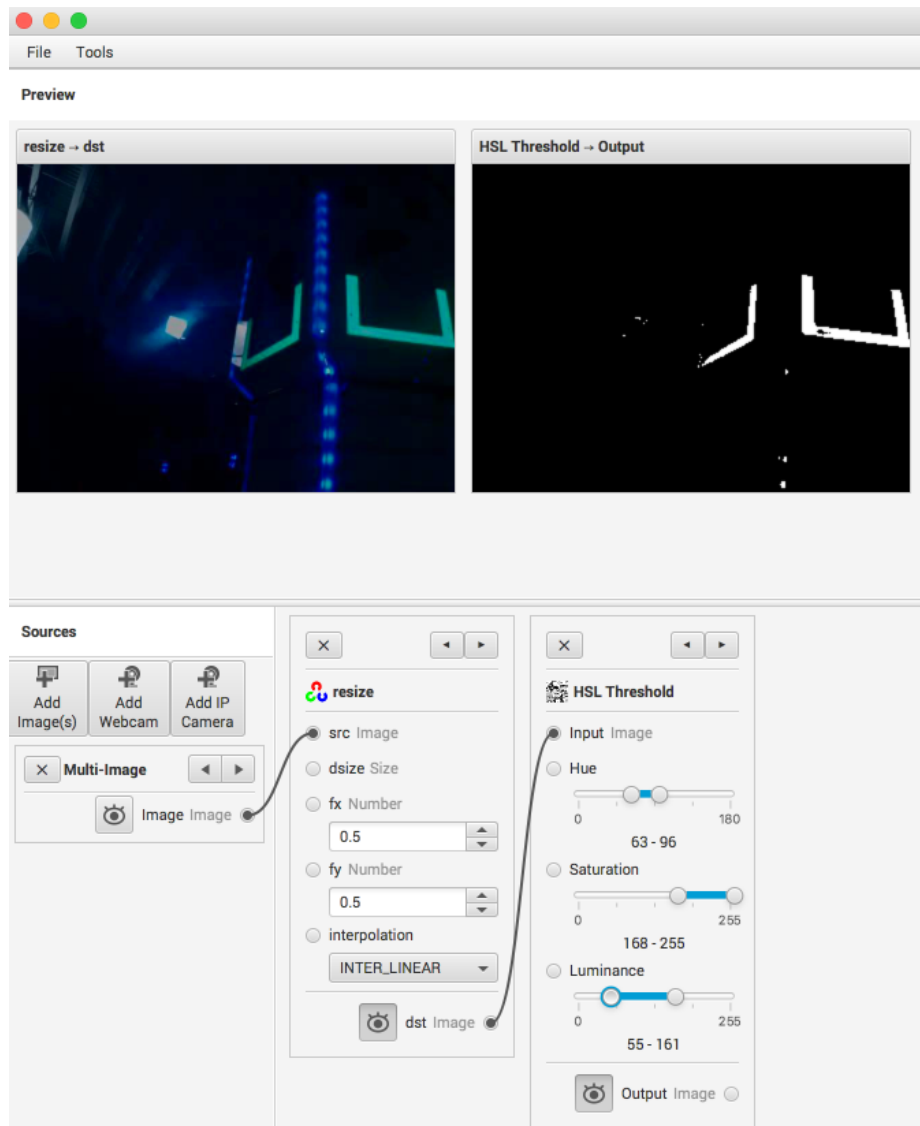


## Resized images



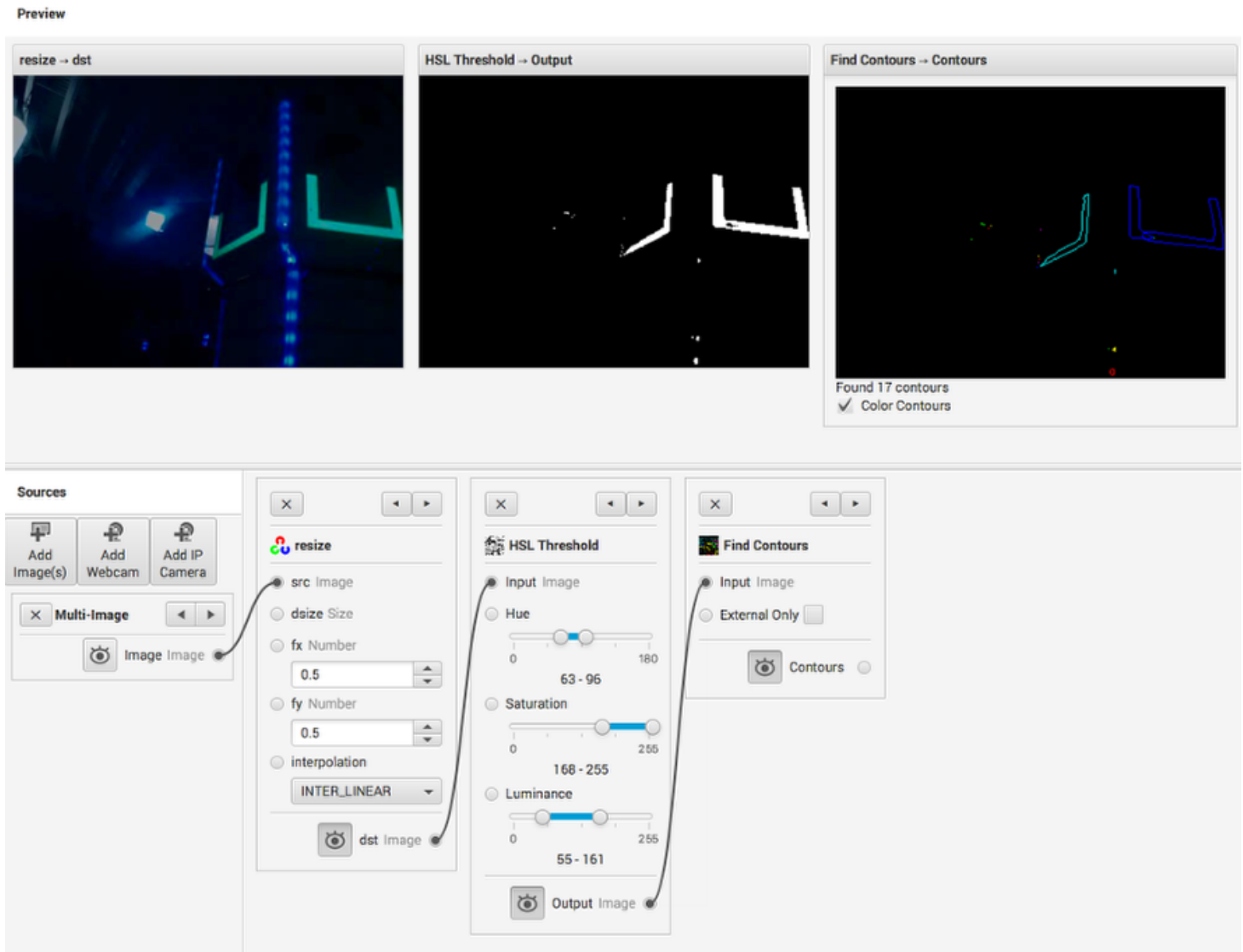
Changing the scale factor by 0.5 reduces the images to 1/4 the number of pixels and makes processing go much faster, and as said before, fits much better for this tutorial. In this example, the full size image preview is turned off and the smaller preview image is turned on. And, the output of the image source is sent to the resize operation.

## Use HSV Threshold operation to detect the targets



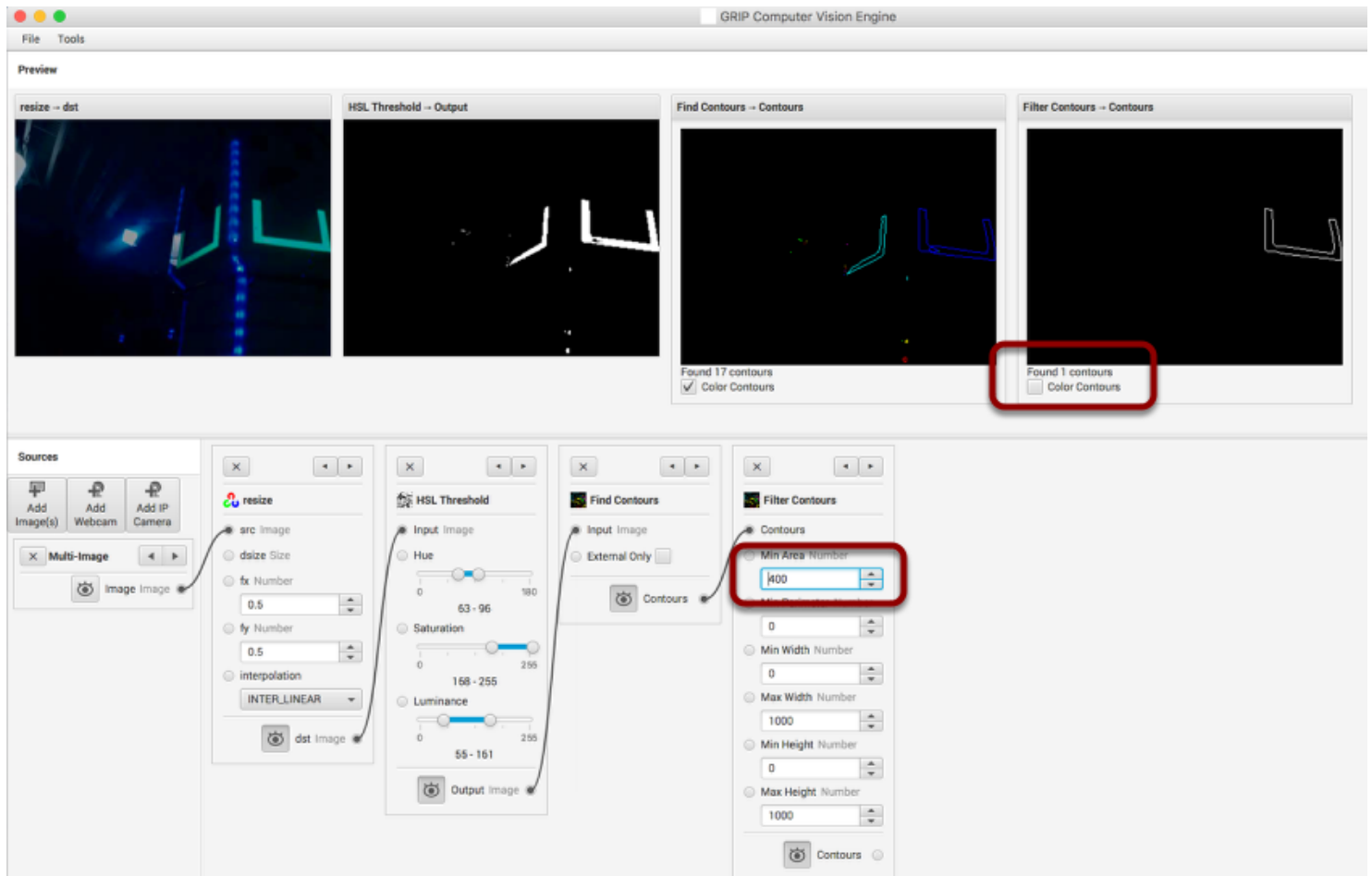
Since the targets were illuminated with a constant color ringlight, it's possible to do a threshold operation to detect only the pixels that are that color. What's left is a binary image, that is an image with 1's where the color matched the parameters and 0's where it didn't. By narrowing the H, S, and L parameters, only the target is included in the output image. You can see that there are some small artifacts left over, but mostly, it is just the vision targets in the frame.

## Find contours in the image



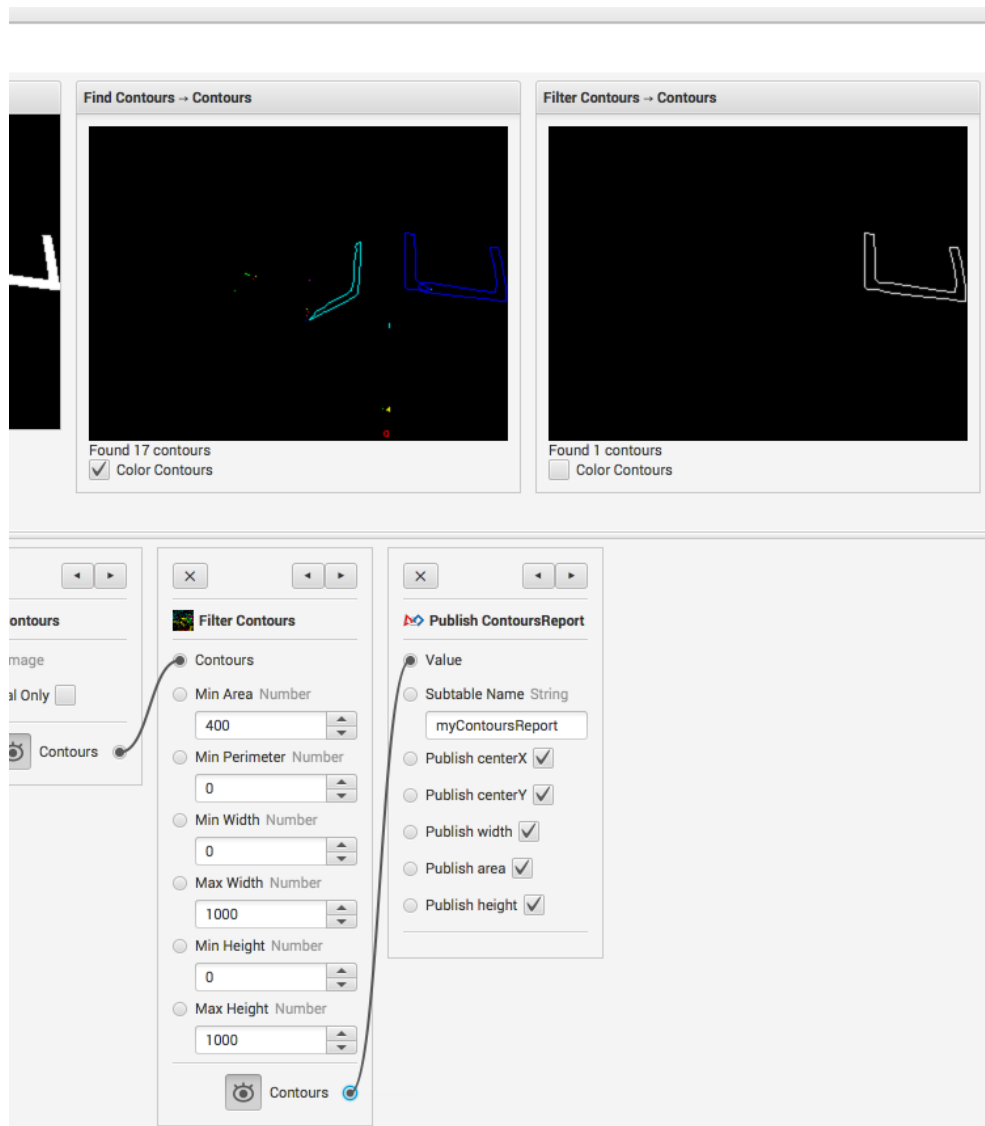
Now we can find contours in the image - that is looked for connected sets of pixels and the surrounding box. Notice that 17 contours were found that included all the smaller artifacts in the image. It's helpful to select the Color Contours checkbox to more distinctly color them to see exactly what was detected.

## Filtering the discovered contours



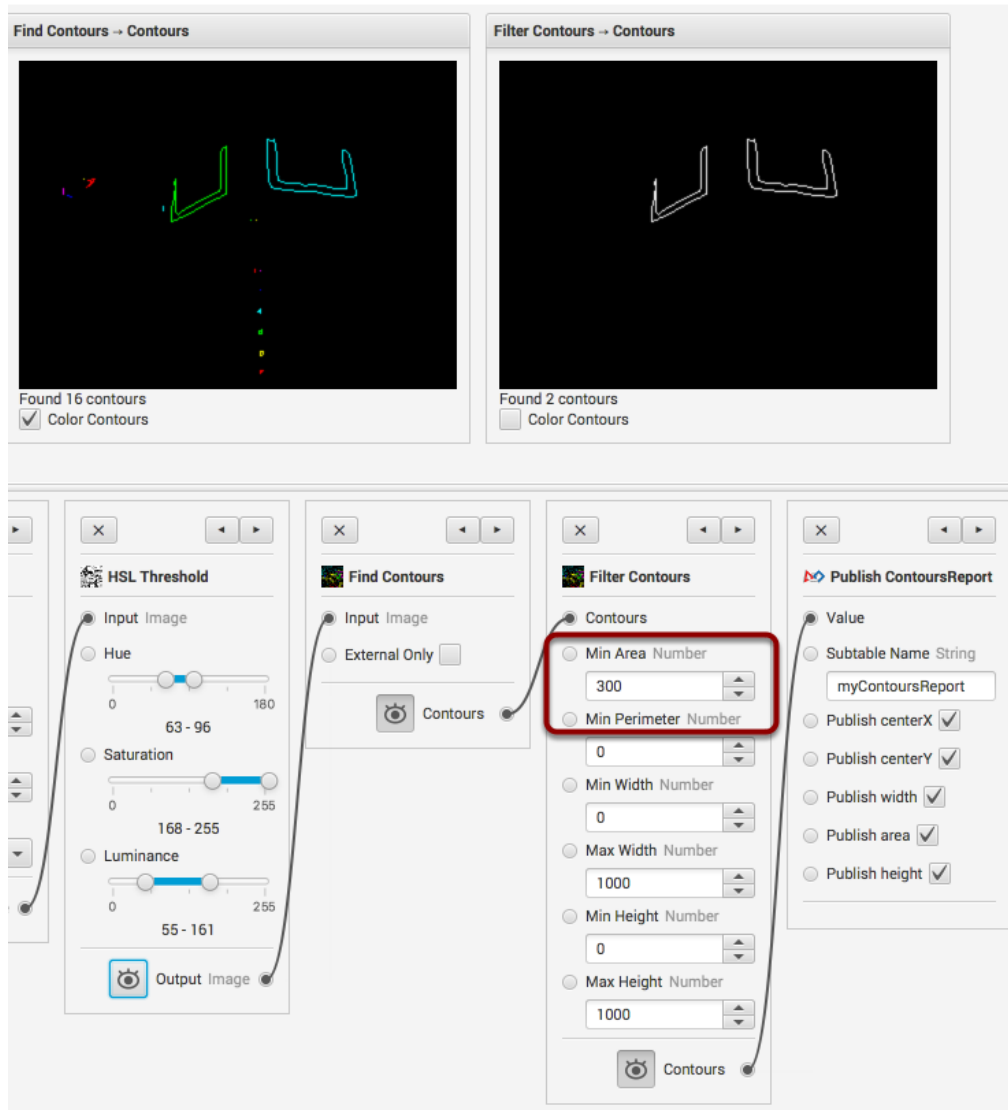
Filtering the contours with a minimum area of 400 pixels gets down to a single contour found. Setting these filters can reduce the number of artifacts that the robot program needs to deal with.

## Publishing the results to network tables



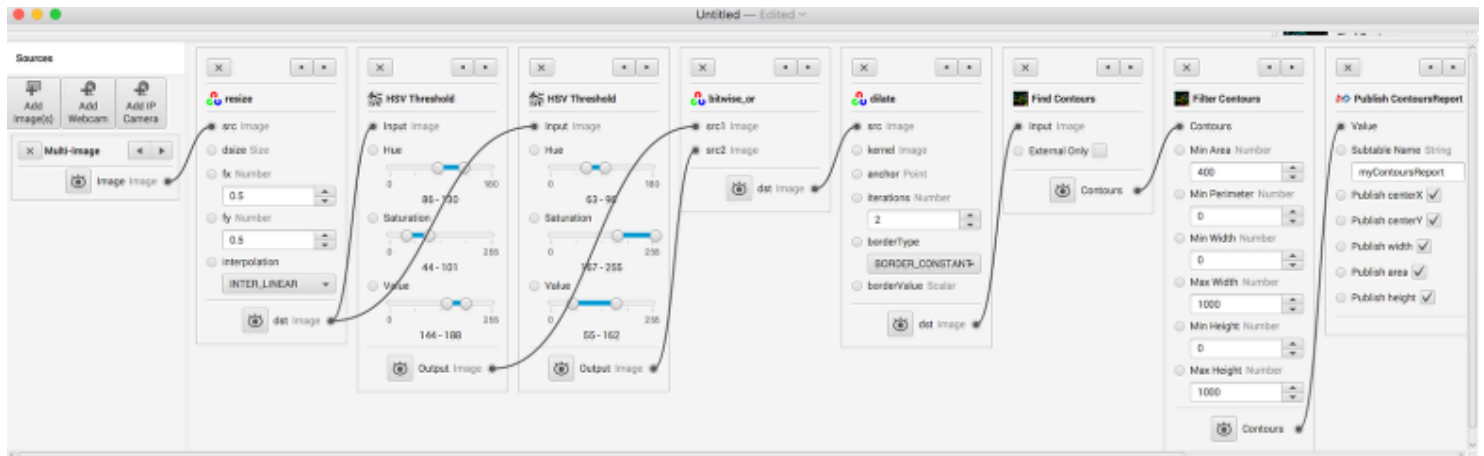
The Publish ContoursReport step will publish the selected values to network tables. A good way of determining exactly what's published is to run the Outline Viewer program that's one of the tools in the <username>/wpilib/tools directory after you installed the eclipse plugins.

## Changing filter values



Reducing the minimum area from 400 to 300 got both sides of the tower targets, but for other images it might let in artifacts that are undesirable.

## Allowing other colors



In some of the images, the two faces were slightly different colors. One way of handling this case is to open up the parameters enough to accept both, but that has the downside of accepting everything in between. Another way to solve this is to have two separate filters, one for the first color and one for the second color. Then two binary images can be created and they can be OR'd together to make a single composite binary image that includes both. That is shown in this step. The final pipeline is shown here. A dilate operation was added to improve the quality of the results which adds additional pixels around broken images to try to make them appear as a single object rather than multiple objects.