“Computer Vision”

from scratch

Brian Starkey
Pi Wars Mini Conference
October 2019

Slides link on Twitter after the talk #PiWars

Who?

BrianS on Discord, @usedbytes, blog.usedbytes.com, hackaday.io

Brian Starkey

Day job: fixed-function pixel processing hardware

Entered Mini Mouse in Pi Wars 2019

- Only **camera** + compass*
- 3rd overall in Pros category
- Fastest “Hubble Nebula”, 2nd “Blast Off”, 3rd “Canyons of Mars”

*9DoF sensor

Very briefly an introduction to me.

My name’s Brian. For my day job, I work on very low-level software for bits of pixel-processing hardware

I entered Mini Mouse in Pi Wars 2019, which only has two sensors: a camera and a compass.

I was really inspired I would say by Piradigm last year, where Mark was able to basically do everything with a camera. Frankly I didn’t have space for many sensors, so I was really keen to save space and only have a camera.

I used the camera for all of the autonomous challenges, and it went pretty well. I had the fastest hubble run overall, and came second and third in the line and maze challenges, using the algorithms I’ll talk about in a moment.

The plan?

- What and Why?
- Tools
- Algorithms
  - Thresholding
  - Edge detection
  - Line detection
- Wrap-up

I wanted to do this talk, because I feel like people have this fear of computer vision, and think it's some ultra-complex vodoo magic.

In my case, I'm running really really simple operations, which were perfectly adequate for the challenges this year, so I want to try and show that it doesn't need to be that complicated.

I'll talk a bit about the tools I'm using and then run through the specifics of essentially all the algorithms Mini Mouse has.

With a varied audience it's hard to target this talk at everyone, but I hope you all get at least something out of it.

Also, I'll need to gloss over a lot of details, but if you want more, I will gladly talk your ear off afterwards!

“Computer Vision” from scratch?

“Computer Vision”

- Too basic to really count as CV
- But still *enough* for (last) Pi Wars

From scratch:

- Efficiency and control
- Learning experience
- Go OpenCV bindings are ... immature

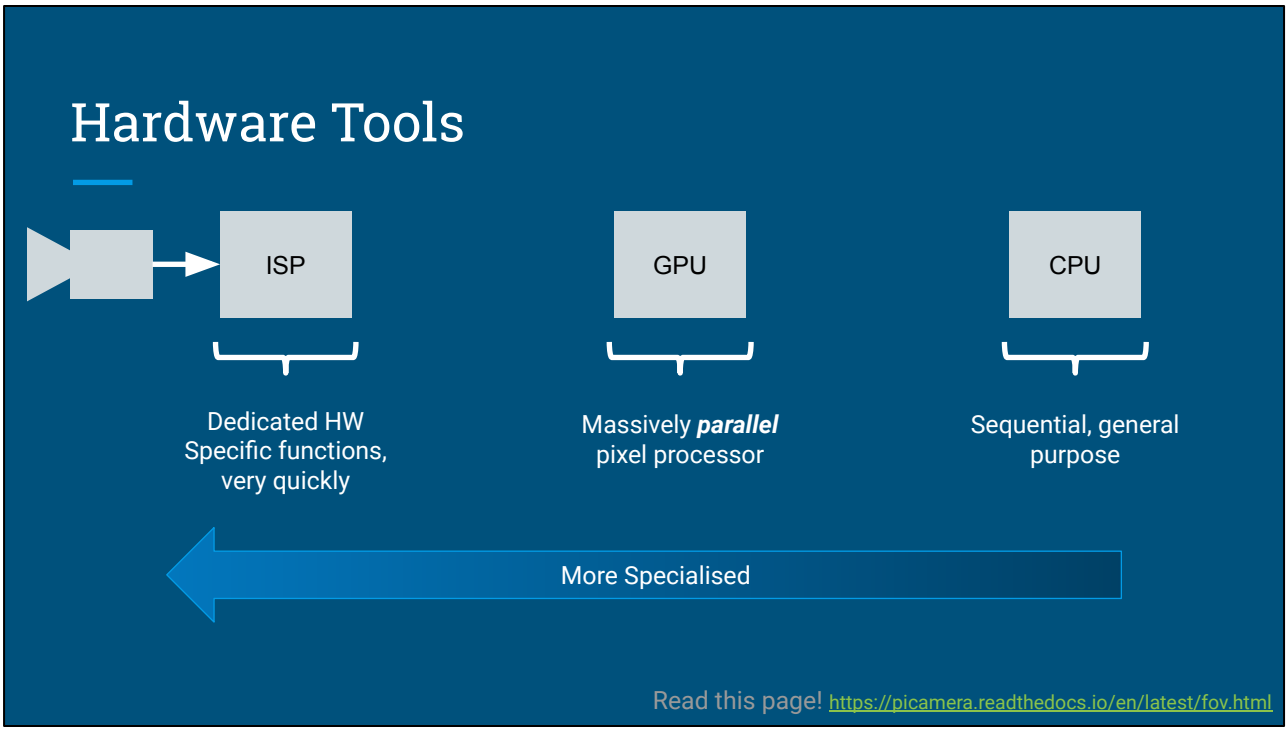
As for the title.

“Computer Vision” is in quotes, because I think this stuff is so basic it only barely qualifies. But still, it was enough for this year’s competition.

From scratch is because I’m not using any libraries like OpenCV. Not because I want to show off, but because I want to show that you really can understand and write this stuff yourself without needing to do a PhD thesis.

Tools

Before diving into the computer vision bits, I think it's worth spending a bit of time talking about what tools are useful when working on this stuff.



To start with we can look at the physical processing tools we have available to us.

Inside the main chip on the Pi, there's three main processing elements which are relevant for us. You can use all of these as part of your image processing pipeline for computer vision.

Starting all the way on the left, is the ISP or Image Signal Processor. This is the part that connects to the physical camera sensor, and it's packed full of super-specialised hardware which is there specifically to process images from the camera.

This means it's limited in what it can do - it can only do some image-processing related things - but it does those things extremely efficiently.

Because of that, it's a good idea to check if the ISP can do what you need - for example resizing or rotating your camera picture, because if you use the ISP to do it, it's basically free and won't slow anything down. I'll tweet a link to the slides - there's a few examples in there of what I mean.

All the way on the right is the CPU. This is the bit we're all familiar with, and it's the least specialised. That means it can do anything, but it won't do it as fast as

something designed to do just one job. The CPU does things sequentially, one after the other.

Somewhere in the middle is the GPU, or Graphics processor. This is still a special piece of hardware, but it's more general purpose than the ISP. The GPU is a big parallel processor with a few special bits tacked on for 3D graphics. It's very good at doing the same thing to lots of bits of data at the same time, but it's not very good at doing things one after the other.

Using the GPU is relatively more complex than the others, but again there's some links in my slides if you want to give it a go.

Using the ISP

What for?

As much as possible! If it *can* do what you want, then use it, because it's basically free.

Python cv2	ISP (with Picamera)
`cv2.resize(img,(320,340))`	`camera.capture(resize=(320, 240))`
`cv2.cvtColor(img, cv2.COLOR_BGR2YUV)`	`camera.capture(format='yuv')`
`cv2.flip(img,1)`	`camera.vflip = True`
`cv2.blur(img,(2,2))`	`camera.image_effect='blur'` `camera.image_effect_params = 2`

Read this page! https://picamera.readthedocs.io/en/latest/api_camera.html

With the ISP, the decision of whether to use it or not is quite simple:

If it *can* do what you want, you should use it. The camera feed is going through it anyway, and it's effectively "free" to turn on any of its functionality.

There's a few examples in this table of operations in OpenCV and what their equivalent would be using the ISP via the picamera python API. The most common is likely to be the resizing, but things like colour conversions, flips, rotation and some blurring are also possible.

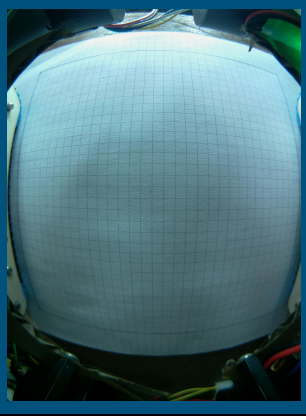
Using the GPU

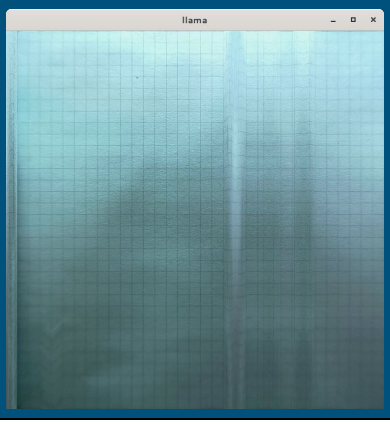
https://hackaday.io/project/26993-bot-matrix/log/65934-camera-open-gl-es-and-distortion-models

What for?

Per-pixel operations, which don't depend on (many) others.

Downside: Complex, with lots to learn





Note: Use the direct Camera→GPU texture fast-path

https://github.com/SanderVocke/picamgpu_minimal

https://github.com/usedbytes/bot_matrix-camera

For the GPU, it's a bit more complicated.

It's good at doing things where it can work on multiple pixels at the same time - which means the value of an output pixel shouldn't depend on too many input pixels.

There's a fast-path from the camera to the GPU, so on Bot Matrix, I used the GPU to correct the fish-eye distortion from my camera.

The main downside is you'll need to learn some OpenGL to use the GPU, and it's a pretty steep learning curve.

Using the CPU

- Get close using the ISP (and GPU)
- Avoid copies as much as possible
- Use (Y)UV if black-and-white is all you need
  - Reduces the work 3x!
- Reduce your pixel count obsessively!
  - Work scales by a factor of n^2

Lastly the CPU - it's not useless, and some things like "if/else" statements are really best handled here.

Whatever work you do do on the CPU, my recommendations would be:
* to avoid copying images around if you can help it
* Look in to YUV and using only the'Y' if you only need a greyscale image. That will reduce the amount of data you're processing by a factor of three
* And, reduce the size of your image to the absolute minimum you can. If you halve the width and height of the image, then you reduce the number of pixels by a factor of 4!

Tools

- Photos! Videos!
  - From robot's eye view - "what does the problem actually look like?"
    https://github.com/silvanmelchior/RPi_Cam_Web_Interface
- Prototyping
  - Hack things together to get a feel for the problem
- Testing
  - Look for "ooh, that's interesting" cases - and capture them!
- Profiling
  - Language-dependent. Go is really great here

For working on computer vision algorithms, if it's not obvious, you're really going to need pictures of the problem you're trying to solve.

I really like this RPi_Cam_Web_Interface - which lets you grab photos from the Pi via a web page, without needing to write any code at all. It makes it really easy to get pictures from your robot's point of view very early on in development.

You probably also need a way to do some quick hacky prototyping, just to get some ideas for what works and what doesn't.

Once you have an idea what your algorithm will be, you need a way to test it, and find cases where it doesn't work.

And lastly, maybe you find out your code is too slow, in which case a profiler will help you to make it faster.

Here's some of my example photos from Mini Mouse.

The top ones were taken with the web interface, before I had any camera code written for the robot.

They're much higher resolution than what I eventually used - but they help to get an idea of how the world looks from the robot's eyes

The bottom row are captured a few months later via my testing tools and are much more representative of what the robot is actually using when running.

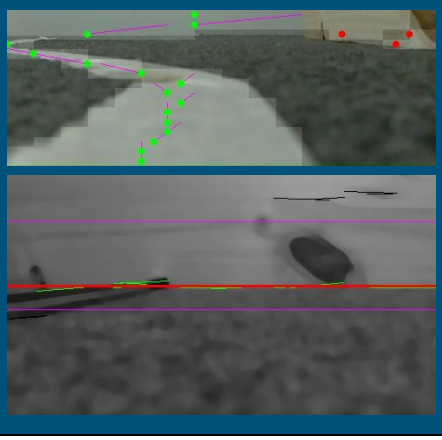
Prototyping

I played around with Python with OpenCV on a PC

- White line + horizon detection

Don't worry about speed or efficiency - just play!

Nothing like the eventual solution - but helped me understand the problem



For prototyping, I spent a couple of evenings playing with OpenCV on my PC, using the images grabbed from the web interface.

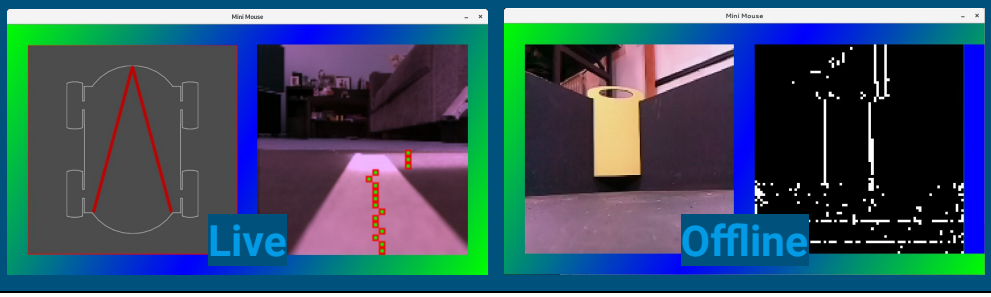
I tried to make two algorithms: One to find a white line for following, and one to find the horizon to detect walls.

The result was far too slow to actually use on the robot - but it was very quick to write, and it helped me figure out some ideas for how to approach the challenges.

Testing

Find a way to:

- View what the robot sees, and capture to an image file
- Run the same code on image files and the camera feed
- Ideally, time and/or profile the image processing code



As the robot develops, it's really useful to spend some time on developing test and debug tools.

I have two main tools:

The one on the left runs on my laptop and talks to the robot over wifi in real-time. It sends a live view from the camera, and I can press a button to save the image to my laptop.

This one also lets me run my processing algorithms on my laptop, using the real camera feed, which means I can test things without having to copy it over to the robot all the time.

With this, I can quickly test, and I can keep an eye out for cases where the algorithm breaks and capture that picture to a file for testing improvements to the algorithm.

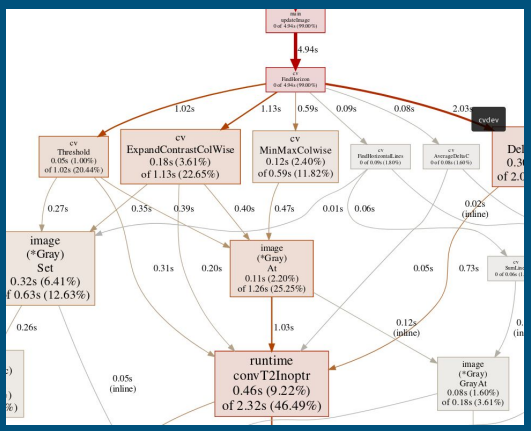
The one on the right only runs on image files. So, I can take pictures from the robot with the first tool, and then work on the algorithms by themselves without needing the robot running.

Profiling

"Premature optimisation is the root of all evil" - Donald Knuth

1. First, write the algorithm in the most obvious way
   - Make sure that it works
2. Then see how long it takes
3. ***If*** it's too slow, profile it and find out why
   - Focus on making only the slowest parts faster

↩ Repeat



https://stackoverflow.com/questions/582336/how-can-you-profile-a-python-script

https://blog.golang.org/profiling-go-programs

And then the last thing is Profiling.

When you've written some algorithms, you might find they're running too slow.

There's this famous quote that "premature optimisation is the root of all evil" - and all this means is that don't bother optimising anything until you know for a fact it's too slow.

If you do find your code is too slow, then a profiler can show you why. On the right is real output from the Go profiler running on my horizon detection code.

Each box represents a function, and the bigger and redder it is, the slower it is. It shows me that this "convT2Inoptr" is taking 46.49% of the total time of the program, and so I know I can make my code twice as fast by somehow avoiding or speeding up that function.

Quite often, the result can be surprising. This function isn't actually mine, it's part of the core language, and I would have never guessed that particular thing was going to be so slow.

Algorithms

OK! That's all the intro out of the way, let's talk about some actual algorithms

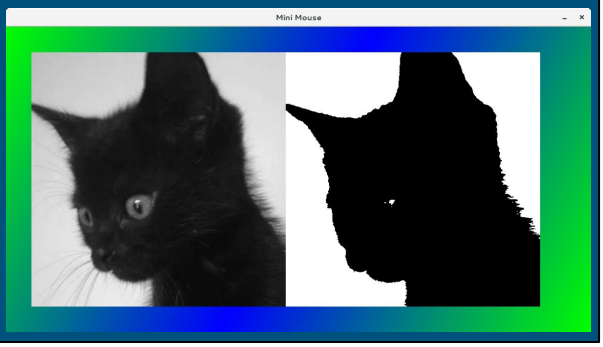
Thresholding https://blog.usedbytes.com/2018/10/quite-robust-thresholding/

Converting a colour/grayscale image, to a black and white one.

- Darker pixels → Black
- Lighter pixels → White

The meaning of black and white depends on the input image:

- Cat or not?
- Ball or not?
- Edge or not?



The first of my building blocks is “thresholding”

This is converting an image with lots of different values into one which contains only black or white - no grey.

The idea is to take all the “darker” pixels and make them black, and all the “brighter” pixels and make them white. Depending on the image that you feed in, the meaning of “black” and “white” after thresholding will vary.

In the case on the right, we’ve got a dark kitten on a light background - so after thresholding, all of our black pixels mean “cat” and all of the white ones mean “Background”

This is useful to help us throw away information which we don’t need, and just get a simple answer for each pixel: “cat or not”. Later on we’ll see some practical uses.

By the way - if you check out my blog post there, there’s a great example wasting a weekend on premature optimisation.

Thresholding

```
for pixel in image:
    if pixel > 128:
        pixel = 255
    else:
        pixel = 0
```

Threshold

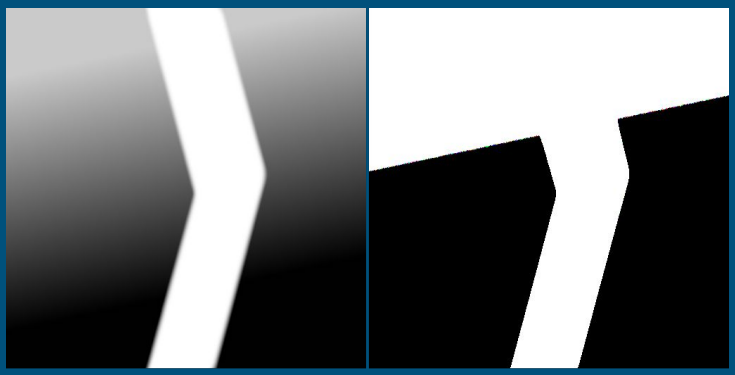
Really, it's very simple:

For each pixel in the image, we need to decide if it's a "bright" pixel or a "dark" pixel, and convert it to white or black as appropriate.

In this case our "threshold" where we decide if something is "bright" or "dark" is 128. Anything greater than 128 becomes white, and darker becomes black.

Thresholding

```
for pixel in image:
    if pixel > 128:
        pixel = 255
    else:
        pixel = 0
```



Sadly, just saying “anything greater than 128 is bright” runs into problems when the difference between bright and dark is small - as shown at the top of this picture.

As the background gets lighter, some of the “darker” pixels become greater than 128, and so they get converted to white and we lose the line

Adaptive Thresholding http://homepages.inf.ed.ac.uk/rbf/HIPR2/adpthrsh.htm

Ideas:

- Check proportion of black vs white
  - ```
    if (too_much_black) decrease_threshold()
    if (too_much_white) increase_threshold()
    ```
- Adjust threshold based on local pixel values
- Adjust pixel values, keep a static threshold

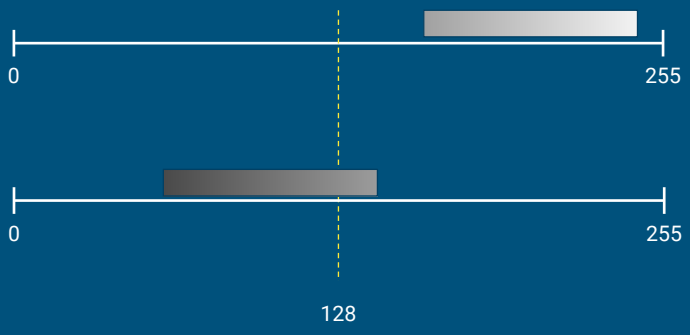
One way to fix this is to use adaptive thresholding - where instead of always checking everything against 128, we try and pick a better number based on “something”.

There’s a bunch of different ways - you could check how many black and white pixels you have after thresholding. If everything is coming out white, then you should increase your threshold, to make some of the lighter pixels get turned to black.

Another approach is to us a different threshold for different regions of the image, based on the pixel values in that region

And the one I went with is to adjust the pixel values in the image so that the darkish pixels are always much darker than the brightish ones.
    ```

Fixed threshold problem

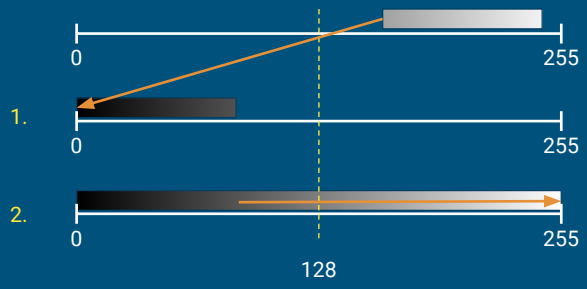


So we have this problem where in some parts of the image, all the pixels are brighter than the threshold.

We could just move the threshold, but then other parts of the image would end up being entirely less than the threshold, and come out totally black.

Contrast Expansion

Make the darkest pixel black and the brightest white, and spread out the rest in-between.



The approach I took to solve this, was to adjust each row in the image to make sure the difference between the darkest and brightest pixel was as big as possible.

That's a two-step process, first, we shift all of the values in the row down, so that this minimum value becomes zero.

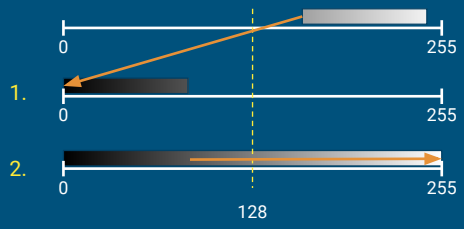
And then, to increase the difference between dark and bright, we stretch out all the values to make the brightest one white.

Contrast Expansion

```
for each row in image:
    darkest = min(row)
    brightest = max(row)
    diff = brightest - darkest
    for each pixel in row:
        pixel = (pixel - darkest) * (255 / diff)
```

1. Make the darkest pixel black

2. Stretch everything else out, making `brightest` → 255



And then, threshold with 128

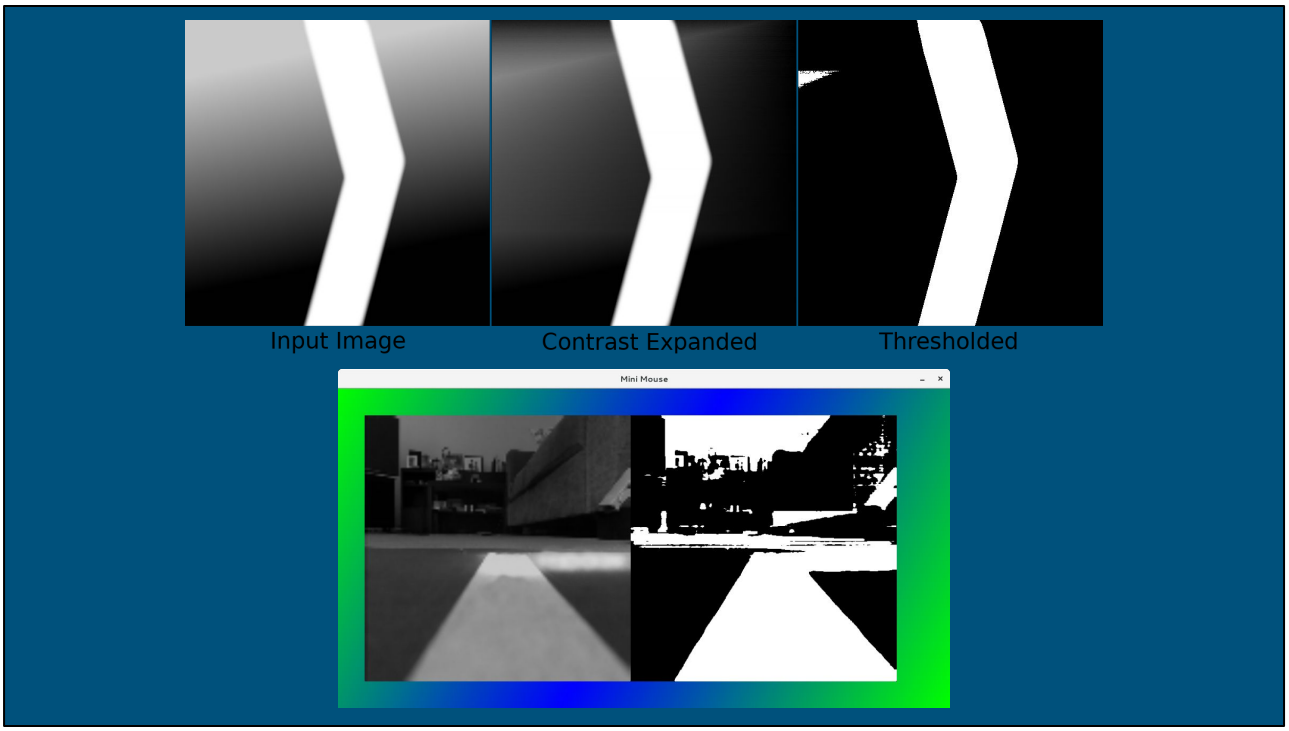
In code, it looks like this:

For each row in the image,

Find the darkest pixel, and find the brightest pixel, then find the difference between them

Then, for each pixel in the row, shift everything down to make the darkest pixel black, and then divide by the difference to stretch everything out

~~Of course, you can achieve the same thing by using the min and max values to calculate a different threshold for each row, but I preferred this approach to let me use a global threshold after pre-processing the image.~~



This is the result.

On the left is the input, with the contrast getting lower towards the top of the image.

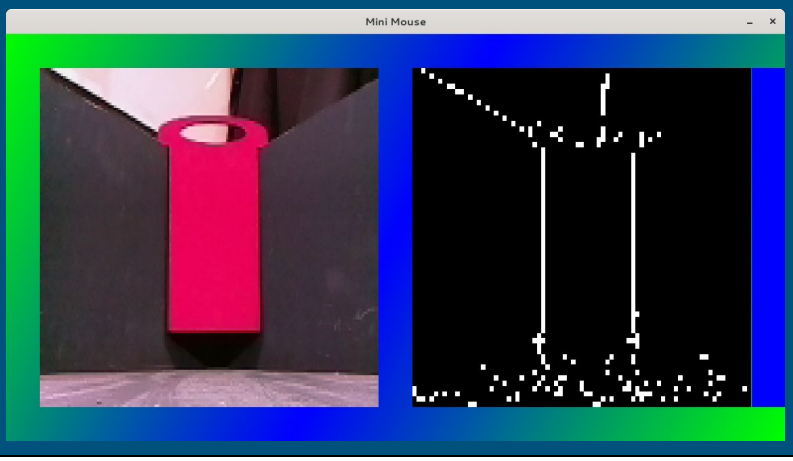
In the middle, is how it looks after contrast expansion - the background gets darker, preserving the bright line.

And on the right is after applying a threshold of 128 to the contrast-expanded image. It’s not perfect, with a little white spot on the left - but it’s far better than before where the whole top half of the image was white.

So this thresholding algorithm is all I needed in terms of image processing for the white line following, but it’s also going to be used as a part of the next algorithms.

Edge Detection

Finding edges is a fundamental operation for identifying “things” in images



Next, we’ll look at edge detection.

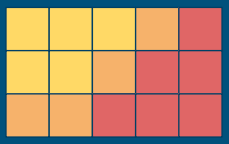
If you want to find a “thing” in an image, like this red board, you need to find where it starts and ends - and that normally means finding edges.

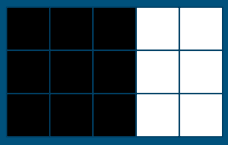
Hopefully you’re mostly familiar - but one of the challenges this year required identifying these coloured boards in the four corners of a square arena.

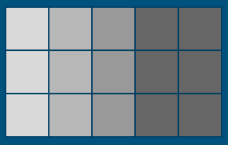
What's an Edge?

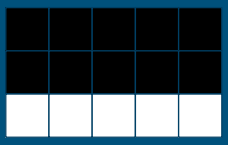
Simply put - a difference in value between pixels

- But which pixels?
- And how much difference?
- And how do we measure difference?









First things first, what's an edge?

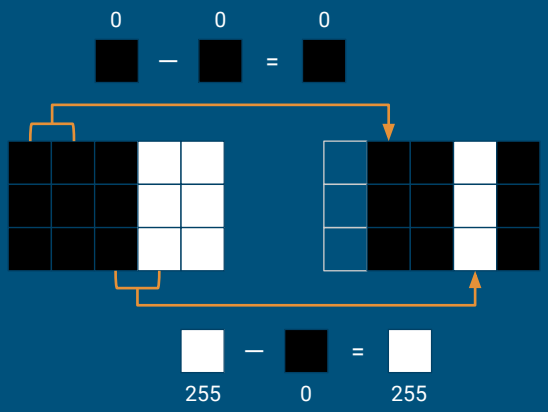
It's basically just a difference between pixels which are near to each other

but we need to decide which pixels to compare, how to compare them, and how different they need to be to say "yes, this is an edge!"

So this is clearly an edge, but this middle one is less obvious. And of course it doesn't have to be black and white - here we've got an edge between two different colours.

Simple vertical edge detector

Find the difference between each pixel and the one to its left



Here's a very simple vertical edge detector.

All we do is compare each pixel to the pixel to its left, and store the difference into an output image.

This is just a black and white image to make it simple, but of course you can compare colour pixels to each other in the same way, too.

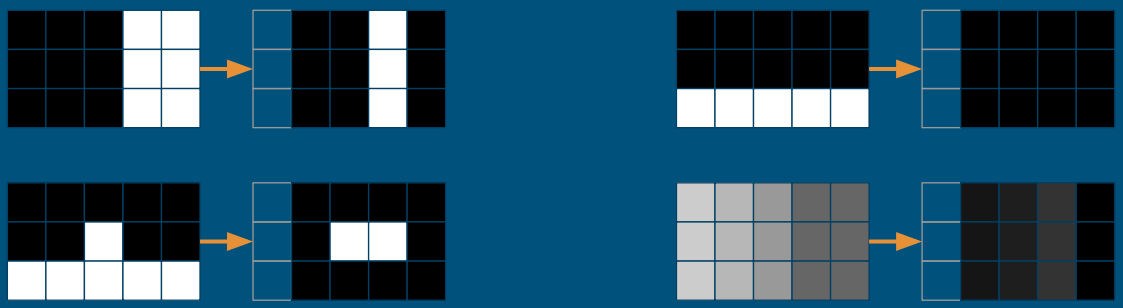
So, for these two pixels - black minus black is black - so the output pixel is black

White minus black is white, so the output pixel is white.

We can do this for every pair of pixels in the image, and that gives us this output with a white line where the edge was in the input image.

Simple vertical edge detector

```
for row = range(0, height):
    for column = range(1, width):
        out[row][column] =
          abs(in[row][column] - in[row][column - 1])
```



In code it looks like this, so for each row and each column, we set the output pixel to the the input pixel minus the pixel on its left.

We need this “abs()” operation here, to make sure that if we have something like black minus white, we get +255 and not -255.

You’ll also see that we don’t have any values for this column, because we don’t have anything on the left of these pixels to compare against.

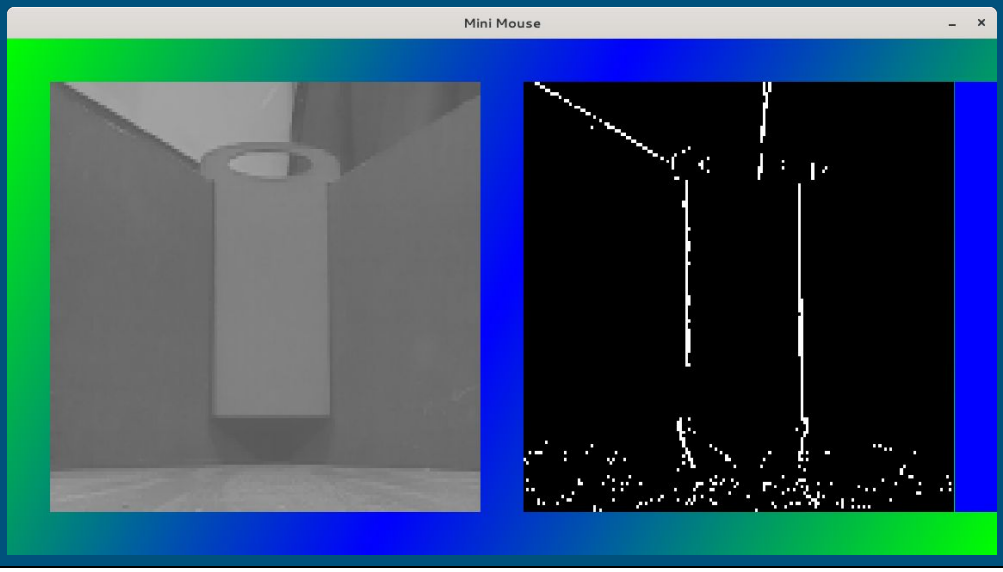
Because we’re comparing left/right pixels, this won’t find horizontal edges. So you can see here, that there’s definitely a horizontal edge in the input, but the output is totally black.

Here, there is an edge, but the difference isn’t very big. So we do get this grey line in the output, but because it’s quite a weak edge, it’s not very bright.

Real Image

If you apply that to a real image, then you can see it’s already doing a pretty good job.

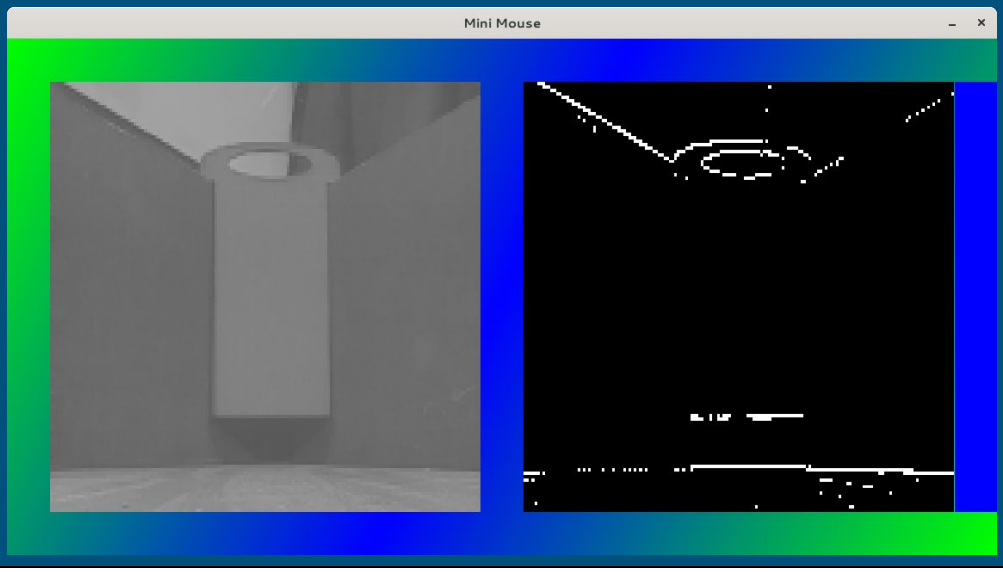
After thresholding



As you may have guessed already, after running the edge detection, we can apply thresholding to make the weak edges much more pronounced.

Now, anywhere we found a fairly strong edge, we have a white pixel, and everywhere else is black.

Horizontal edges - compare with pixel above



Of course, if you compare pixels above/below each other, instead of left/right, then you detect horizontal edges instead of vertical ones.

Note that these diagonal lines show up in both cases - because they have both horizontal and vertical components.

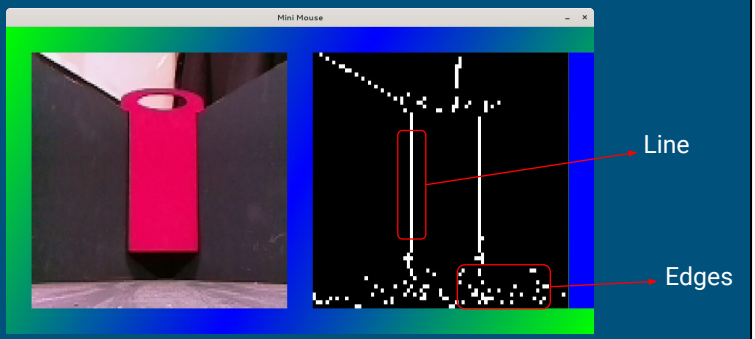
There are much better and more complex edge detection algorithms you can find, but this super simple pixel comparison was enough for me for Pi Wars.

Line Detection

"Edges" happen anywhere where there is texture.

"Lines" imply more structure

To find objects, we often want to find "Lines" rather than simply edges.



The last thing I want to talk about is lines. We've found edges, but that's really all that useful.

You find edges anywhere the image has "texture", like all the speckles at the bottom caused by marks on the floor.

What we really want to find are "lines" - because the object we're looking for is a rectangle made up of lines.

Finding curves, or lines at angles is quite complicated, and there's much more sophisticated algorithms for those tasks.

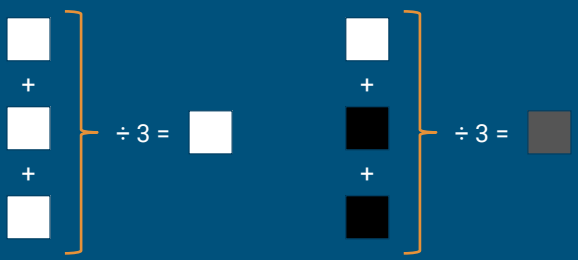
But finding vertical or horizontal lines is pretty easy. And that's fortunate - because I'm only interested in finding these two vertical lines at the sides of the board.

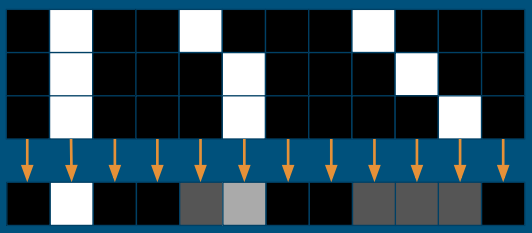
So what does a vertical line look like? Well, it just looks like a column here, with lots of white pixels in.

Vertical Line Detection

Vertical (horizontal) lines are relatively easy to find:

Look for columns (rows) with lots of edges.





Brighter pixels represent longer, more vertical lines

So we simply need to find columns in the image with lots of edges - or lots of white, after we've done edge detection.

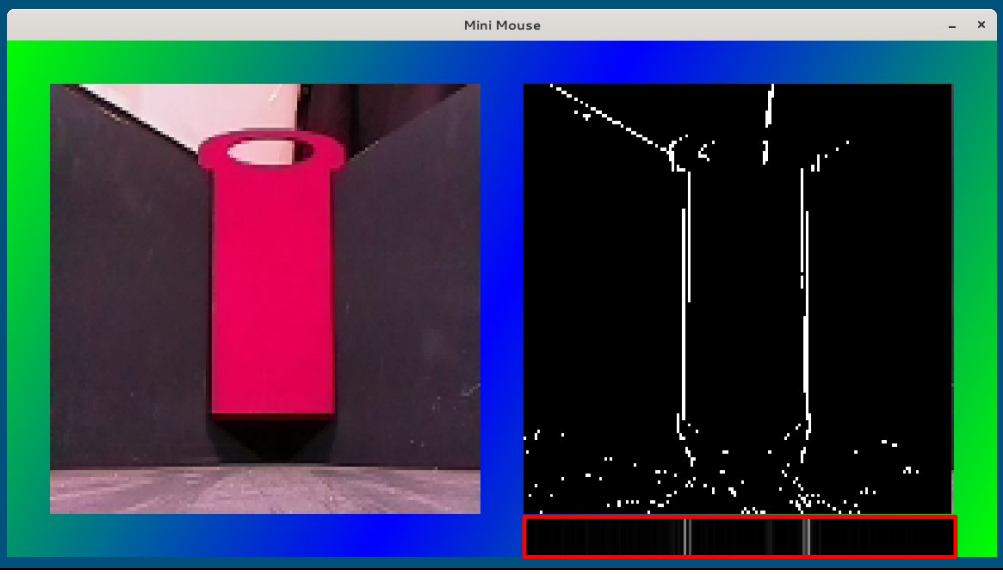
If we find the average of each column in the edge-detected image, then the more edges there are, the higher the average and the more likely it is that we have a vertical line.

Over here is an example image.

These columns have no edges, so the mean is zero and there can't be a line
This column has a perfectly vertical line, and so the mean value is white

As the line gets further away from vertical, the mean values of the columns get lower and so the pixels get darker.

Vertical Lines



Here’s the result running on the “real” image. At the bottom you see the the mean value of each column, shown bigger on the next slide.

Vertical Line Detection

If the line isn't perfectly vertical, it "spreads out":

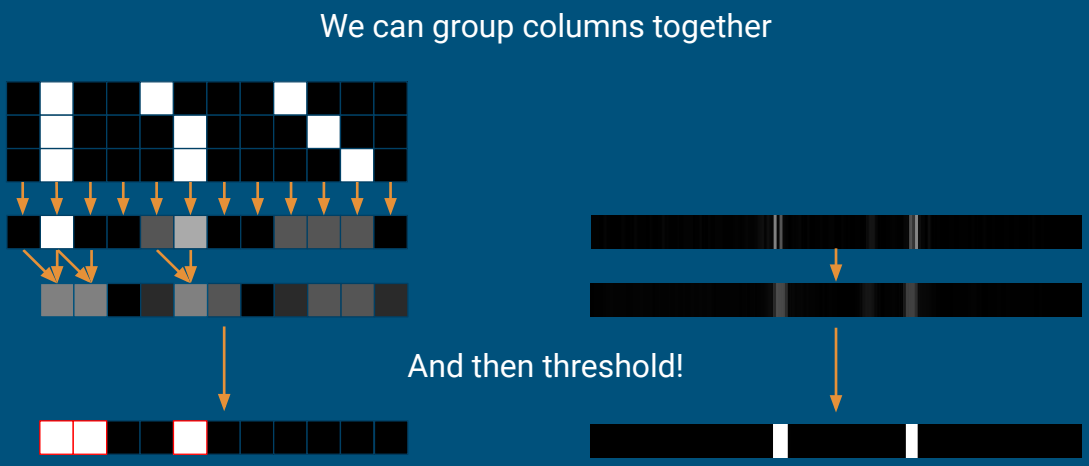
- Either a "wider" and darker smudge → Makes lines harder to find
- Or maybe even several thin marks → Shows false lines!

Already there's a couple of possible issues.

We've seen that if the line isn't perfectly vertical, then instead of one bright mean pixel, we get several darker ones - which makes the line harder to identify

And in this image, the left hand edge gets picked out as two edges, giving two bright spots in the result. That could make us think that there's more lines than there really is.

Vertical Line Detection



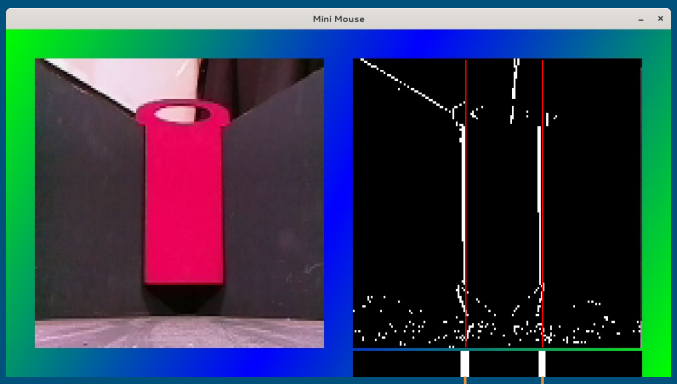
To help with this, we can tweak the code slightly to group columns together.

Instead of getting an average for each column, we can do it for pairs of columns, or groups of three.

This blurs lines together, tending towards showing fewer, darker regions - and then we can apply thresholding to pick out the strongest results - giving this final result with just two white spots representing the two edges of the red board.

Vertical Line Detection

Finally threshold to find the very strongest lines:



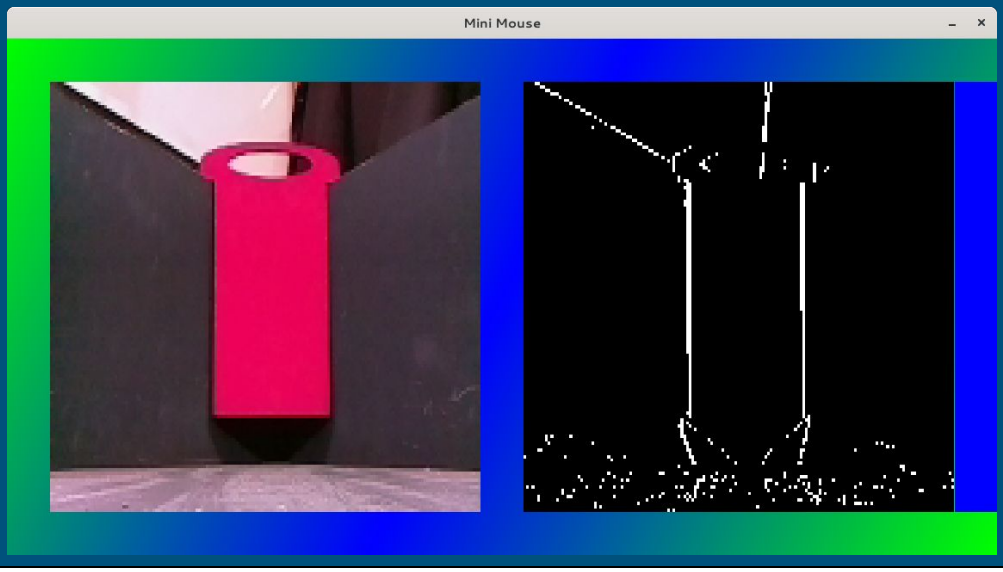
Each group of white pixels represents a vertical line

And that’s it, right? Mission accomplished. After that’s all done, we’re left with a nice clean row of pixels, which tells us exactly where the left and right edges of the board are. And the whole thing only took around 15 minutes to implement!

1) Find edges in the image

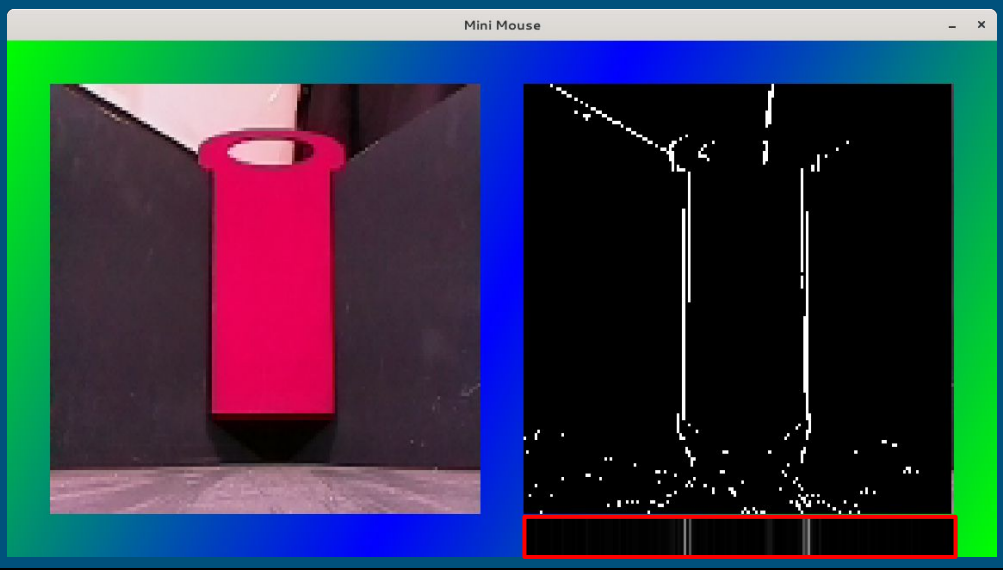
First, we found edges by just comparing adjacent pixels

2) Threshold to keep only the strongest



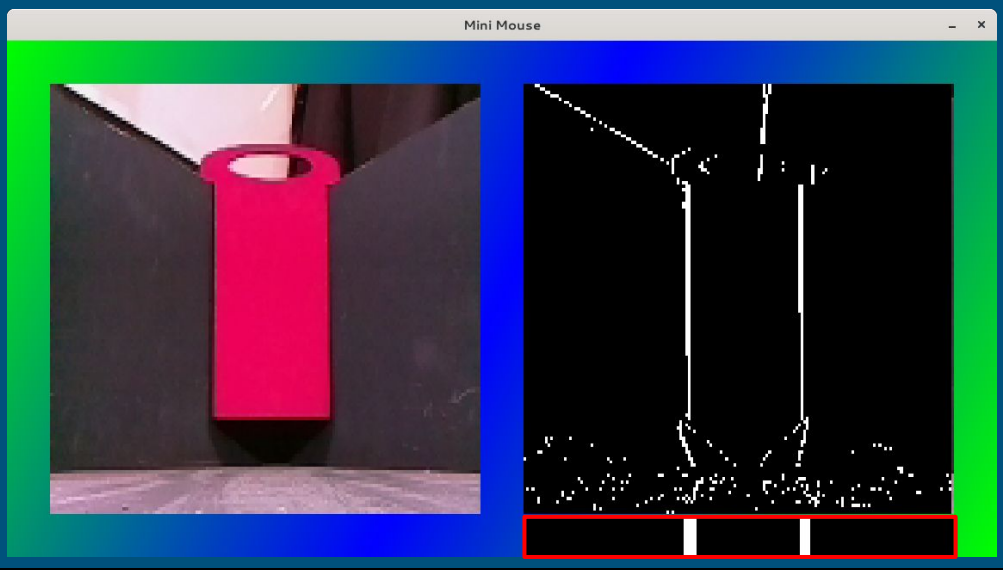
Then, we applied a simple threshold to find the strongest edges

3) Find vertical lines in the edges



Then, add together columns to find vertical lines

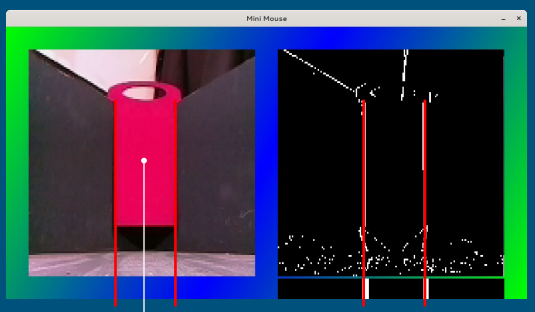
4) Threshold to keep only the strongest



And finally, threshold again to find only the longest, most vertical lines.

Hubble Challenge

1. Turn to where you think a corner is
2. Run this algorithm to find board location
3. Sample the middle pixel to find the colour
4. Turn to the next corner



With those steps, we've got everything we need to base a Hubble challenge solution on

We can turn to where we think the corner is. It doesn't need to be exact, as long as the board is in-view of the camera

Then we can run the algorithm, which will tell us exactly where the two edges of the board are. If needed, we can use that to centre properly on the corner

Then, we can sample the middle of the board to find out its color, and turn to the next corner.

Wrap Up

So hopefully I haven't lost you all with my onslaught of diagrams. Let me try and draw some conclusions.

Conclusions

1. Keep it simple!
   a. Pi Wars challenges are well-defined, and well controlled
2. Use the smallest number of pixels possible
   a. Line follow: **32 x 32** pixels, grayscale only
   b. Maze: **40 x 80** pixels, YUV
   c. Hubble: **100 x 100** pixels, YUV
3. Invest time in tools and testing
   a. It's much easier to understand when you can see what's going on

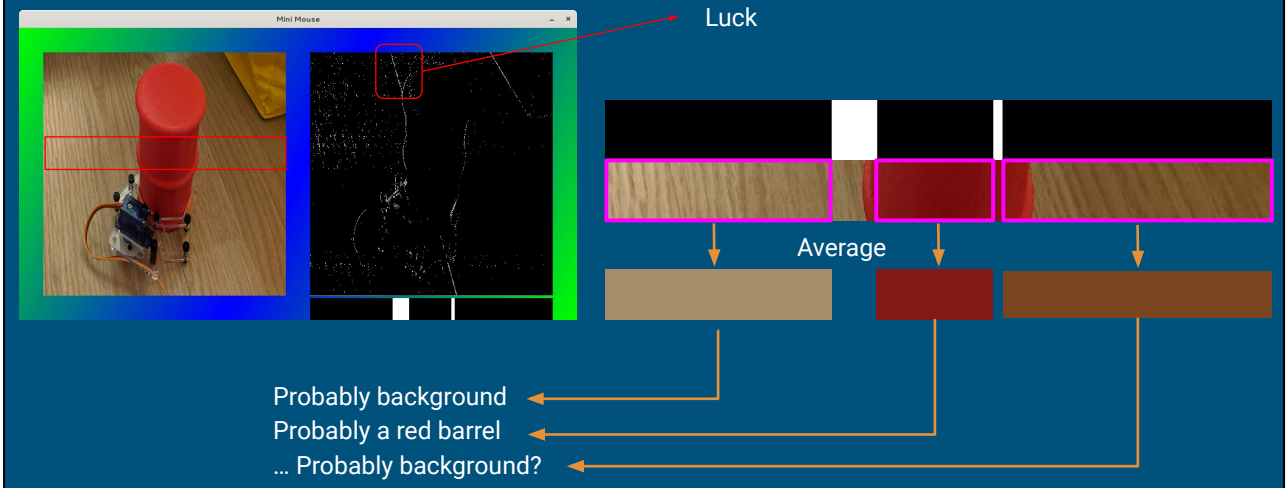
First and foremost - keep it simple! The challenges for PiWars - at least in the past - have been quite well-controlled arenas. This means your code doesn't need to worry too much about what's in the background, because it should be pretty plain. You can make a lot of assumptions, like just looking for the two biggest vertical lines - because you know exactly what the environment will be.

Use a very small number of pixels. This helps to naturally blur things, which is actually quite useful, and also makes everything faster. Remember if you halve the width and height, the number of pixels goes down by a factor of four.

My highest resolution image was 100x100 for the Hubble challenge.

And, really try and invest some time in developing tools. Everything is much easier to develop and debug when you can actually see what's going on with the robot in real-time, instead of trying to guess what went wrong.

Still applicable?



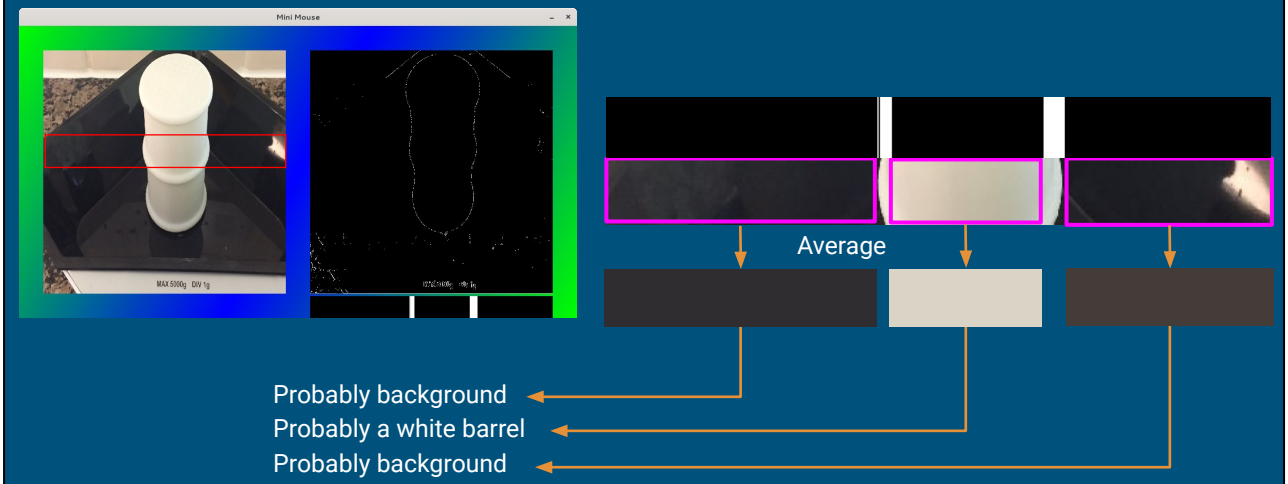
Thanks to @CannonFodder for the image

Now, Hubble isn't in the competition next year, I thought I'd have a go at applying my hubble code to finding barrels, just to see if any of this is still applicable.

I stole this picture from Brian C on twitter. The same algorithm we've just looked at does manage to find two lines around the edges of the barrel, but I think it's somewhat down to luck, as the servo stuff would have thrown it off if not for this edge it finds in the grain of the table.

If you assumed that each vertical line marks the edge of a barrel, then you could split the image at the lines, and then find the average colour for each segment. From there, it doesn't seem too outlandish to identify red and green segments as barrels.

Still applicable?

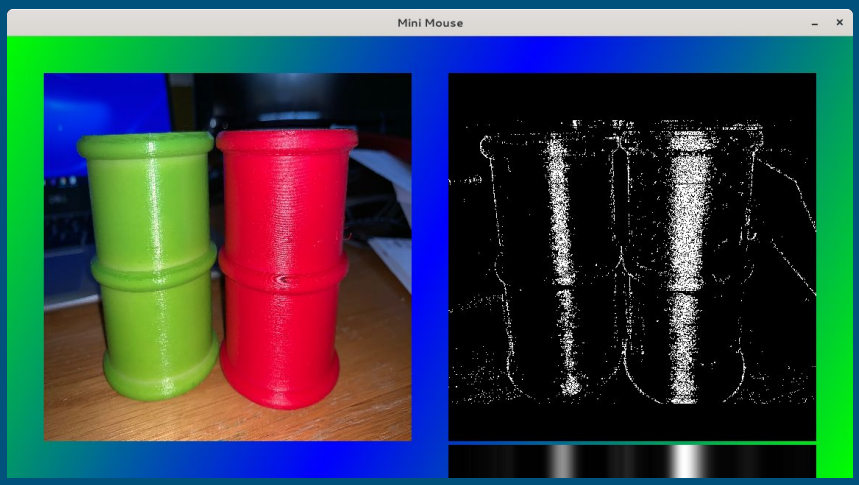


Thanks to @TeamCurrington for the image

This one works a bit better - the barrel is high-contrast against its background, and there's really no doubt that the sides of it have been found correctly.

This shows the advantage of having a uniform background (or well-controlled arena like in Pi Wars) versus trying to find things in the "real world"

Still applicable?



Thanks to @PiWarsRobotics and @PiDrogen for the image

Lastly, Mike uploaded this picture of “the real deal”

And this one, fails miserably

Here, the strongest lines in the image, by far, are the bright glare lines from the camera flash. So, I guess this shows the importance of testing, and having nice even lighting if possible :-)

And that’s it, thanks so much for listening!