

Embedded Systems Lab Prep 8

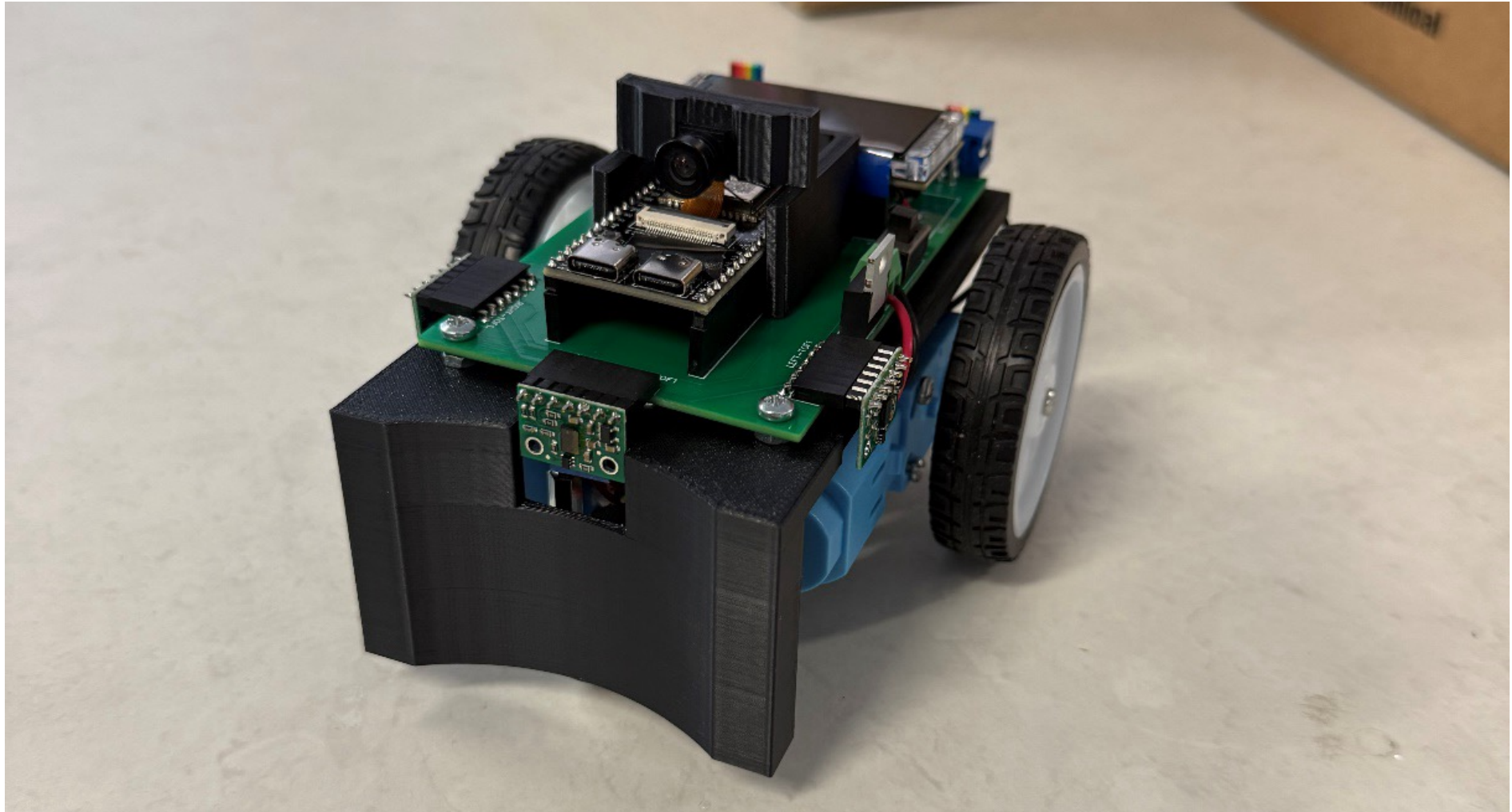
EyeBot-32 Camera

What Are the EyeBots32?

- A small mobile robot (large palm size) built around the ESP32 microcontroller.
- It features 3 PSDs, two motors with encoders, an onboard camera and an extensive software API to program it.
- This API is called RoBIOS, and it hides the internal complexity of driving and sensing, allowing you to focus on bigger picture things.



The EyeBot32



RoBIOS

- This API splits the robot into the following parts:
- LCD – text and simple graphics
- CAM – Camera
- IP - image processing functions
- VW – “vehicle” motion primitives (auto PID)
- MOTOR – direct wheel control and PID tuning
- KEY – read the two onboard buttons
- PSD – read front/left/right distance sensors in mm



Image Processing

- Images can be stored in many different ways.
- In the common RGB method, images are made up of pixels, which each have three values associated with them. Each value corresponds to a Red, Green or Blue value for that pixel.
- These three values can be stored as bytes in an array. One array for each pixel.
- The image is made a singular 1 dimensional array of size: $\text{Width} * \text{Height} * 3$.
- Before we can work on our image, we must allocate enough space for our array.
- The EyeBot-32 can only take images in a QVGA Size.
- We already have predefined constants for the image sizes:

QVGA - 160*120 pixels

- As well as predefined sizes to help initialisation of arrays:

eg. QVGA_SIZE = 160*120*3,

QVGA = 160*120

QVGA_WIDTH = 160, QVGA_HEIGHT = 120

- Which helps us initialise our arrays easily

BYTE colimage[QVGA_SIZE];



Image Processing

- In order to capture an image you then simply run:

CAMGet(colimage);

- Take this image as an example. It's an RGB image of size 20 x 15 pixels.
- Note that row and column indexes start at zero and that they start from the top left of the image.
- In each pixel is stored an array of three 'BYTE's for the RGB val.

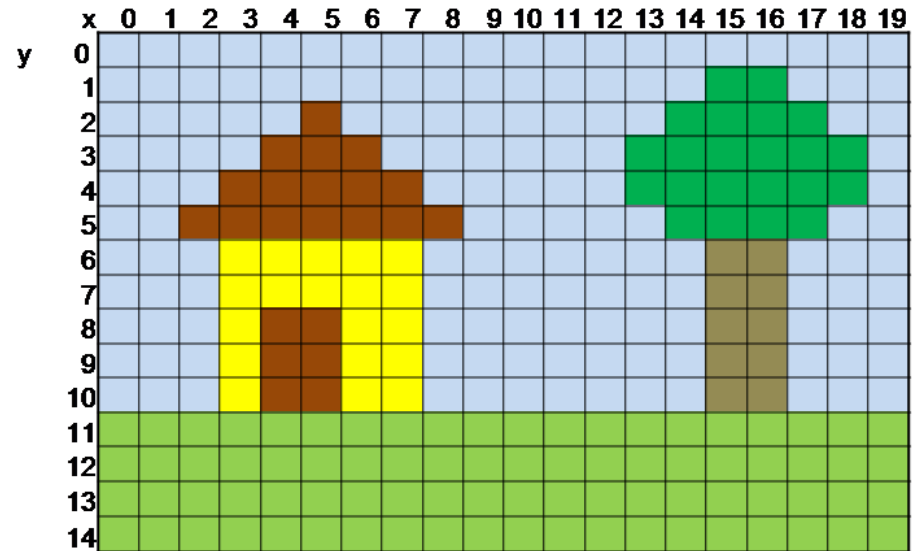


Image Processing

Let's say we want to look at the values in the pixel that makes up the top of the roof of the house.

Remembering back that our images are stored as a single 1D array, to obtain the correct pixel we must do maths...

```
int pixel = row*width+col;
```

```
//in this case row = 4, width = 20 and col = 3
```

As our array will store the RGB BYTES in the order RED, GREEN, BLUE

to obtain our values we can get them by:

```
BYTE p_red = colimage[pixel*3];
```

```
BYTE p_green = colimage[pixel*3+1];
```

```
BYTE p_blue = colimage[pixel*3+2];
```

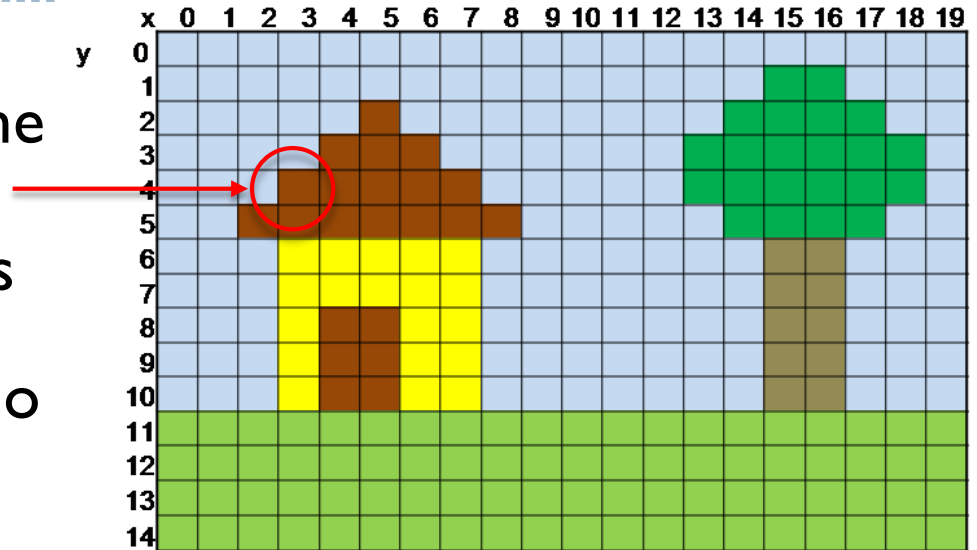


Image Processing

Let's say we want to find the location of an object in an image

- we know that this object is predominately blue
- we know our background is predominately not blue

Let's start with some code to identify the blue pixels in the image.

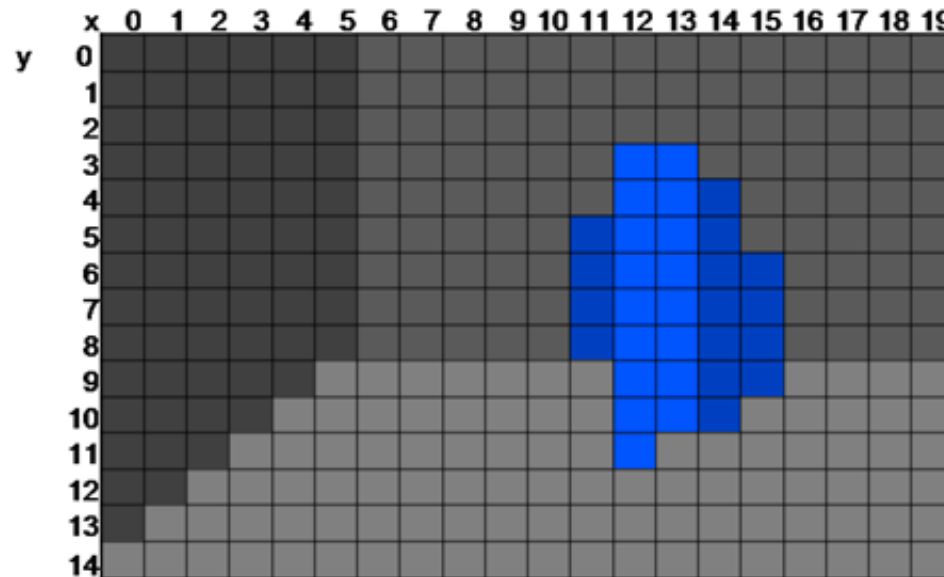


Image Processing

Start with some for loops to scan through the pixels of the image

...

```
int i,j;
```

```
for (i = 0, i < 15,i++) {
```

```
    for (j = 0, j < 20, j++) {
```

```
        /* we can now address each pixel */
```

```
        /*eg colimage[pixel*3+2] for blue */
```

```
        ...
```

```
    }
```

```
}
```

```
...
```



Image Processing

Now let's check to see if that pixel is blue. We'll need somewhere to store the results too so let's make another array.

...

```
isBlue[15][20] = {0};
```

```
int i,j;
```

```
    for (i = 0, i < 15,i++) {
```

```
        for (j = 0, j < 20, j++) {
```

```
/* -> Red pixel */        if (image[i][j][0] < 100 &&
```

```
/* -> Green pixel */      image[i][j][1] < 100  &&
```

```
/* -> Blue pixel */      image[i][j][2] > 200) {
```

```
    isBlue[i][j] = 1;
```

```
        }
```

```
    }
```

```
}
```



Image Processing

Pro tip!

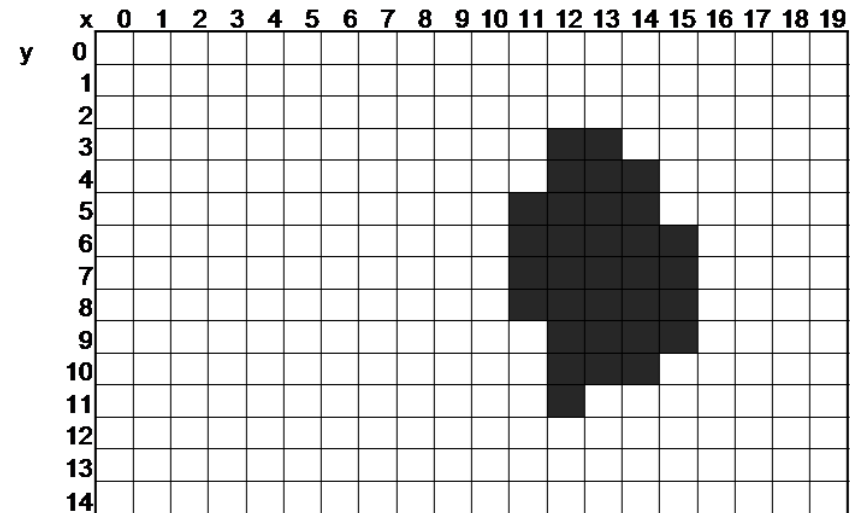
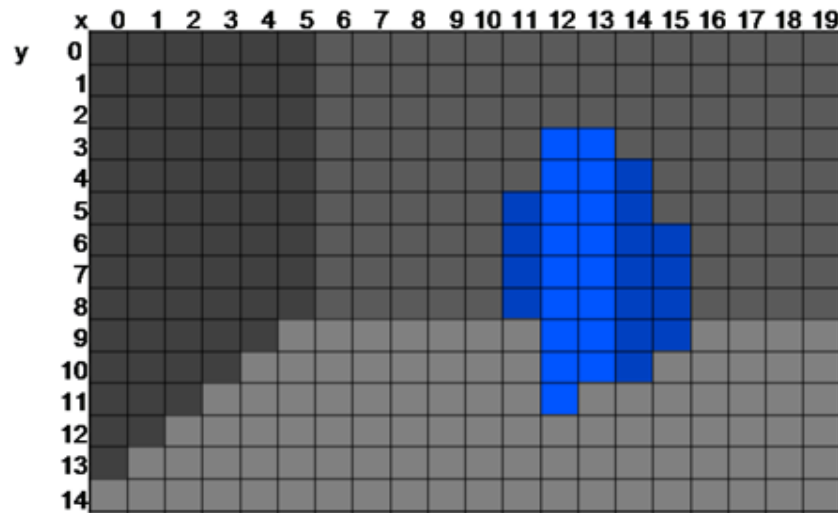
Do not use the values from the previous slide as your thresholds for matching a colour.

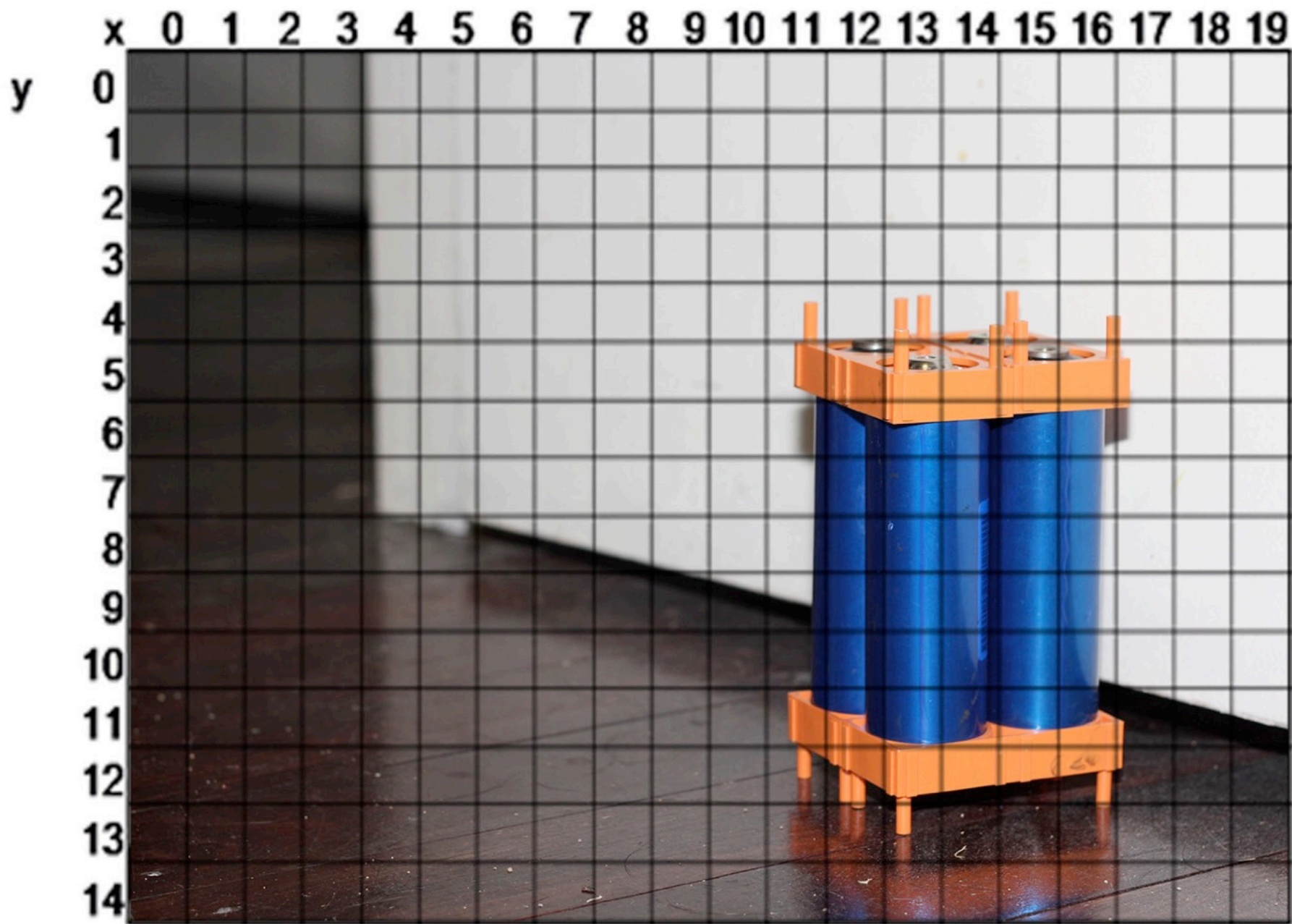
You will need to determine your own values based on camera/lighting/etc.

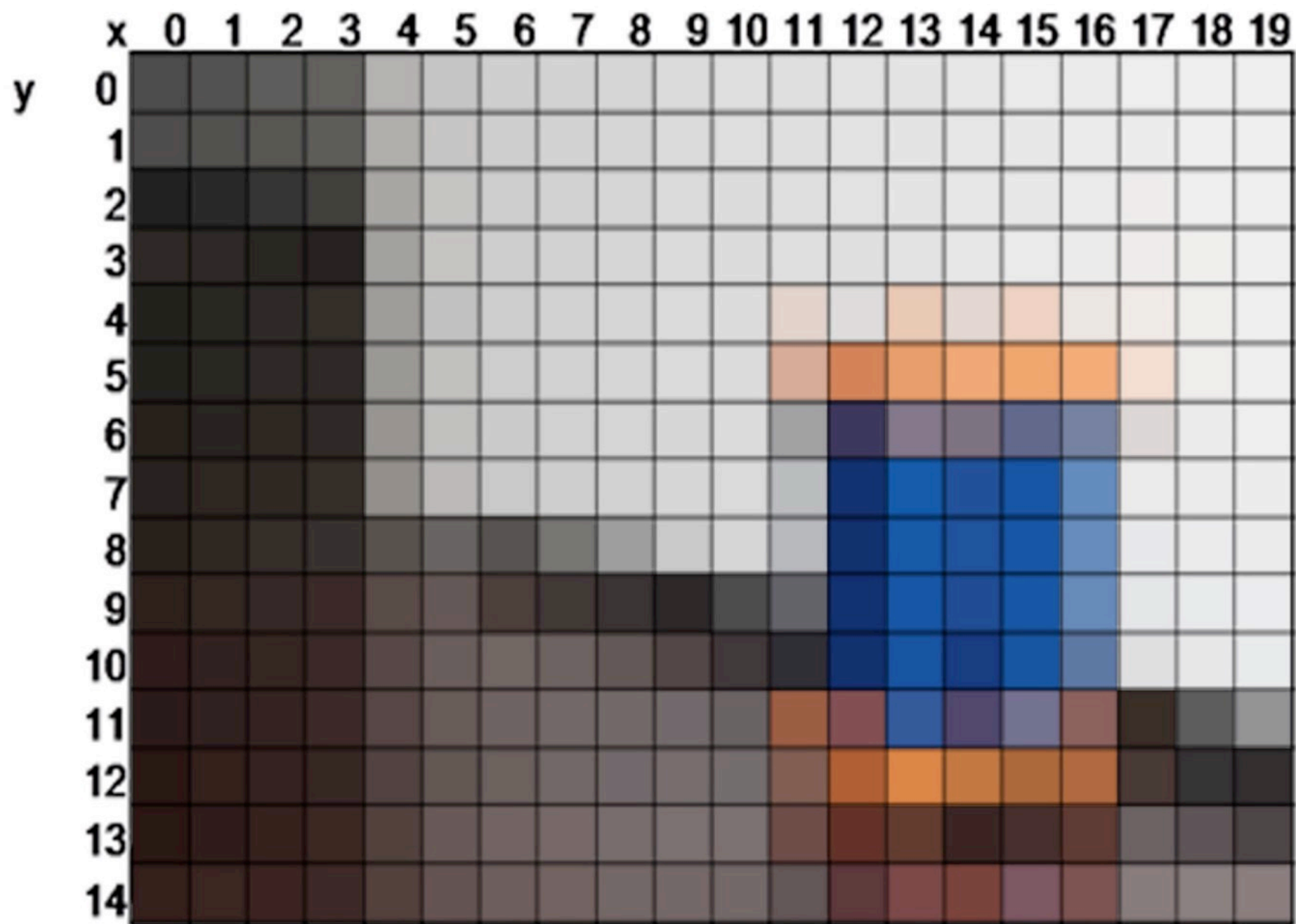


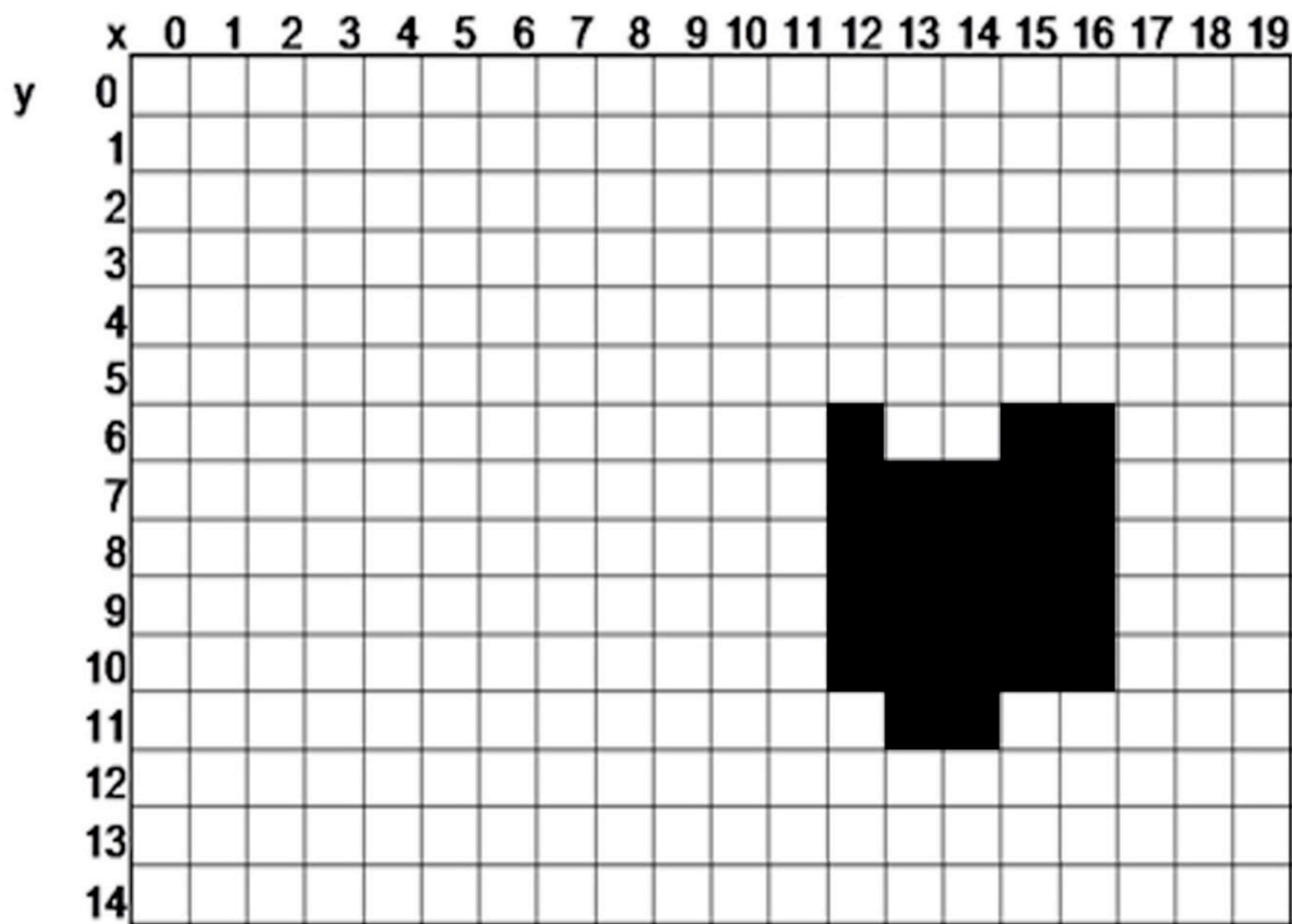
Image Processing

- We now have an array of array of a integers where pixels that are 'blue' are now '1's and those that aren't are '0's.
- If we take the left image as input and were to display the output of this code as a binary image we'd have the image on the right (where '1's are displayed as black and '0's as white)





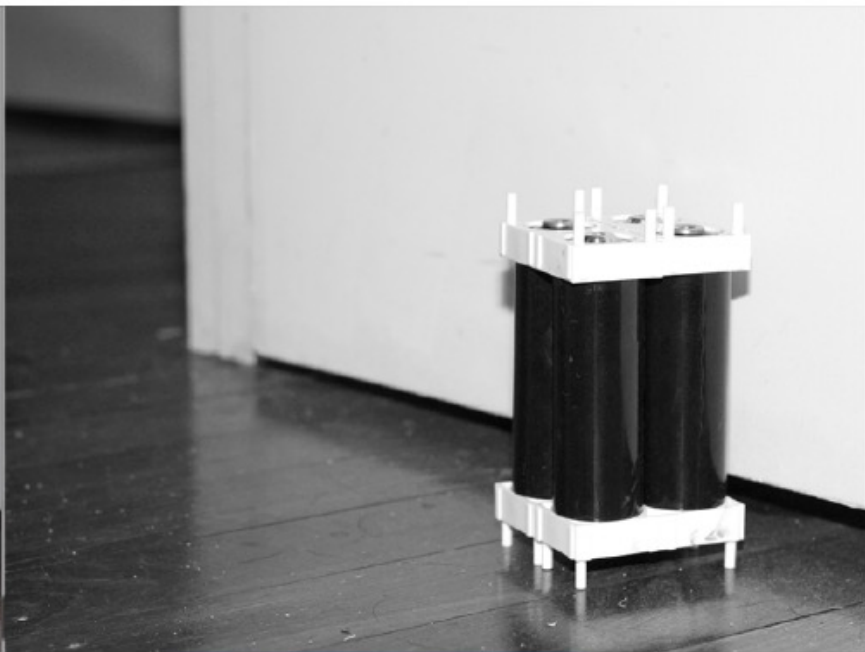




R
G
B



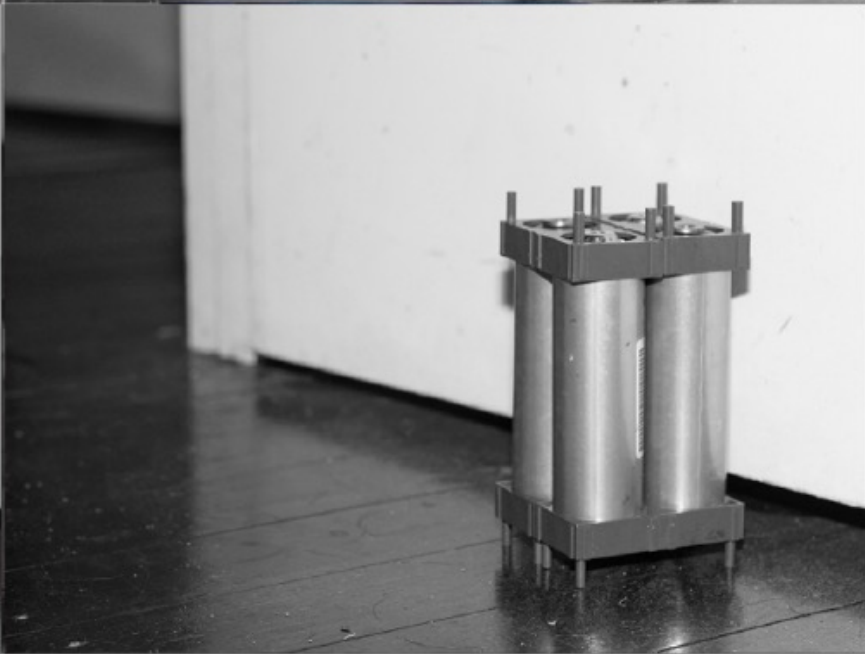
R
E
D



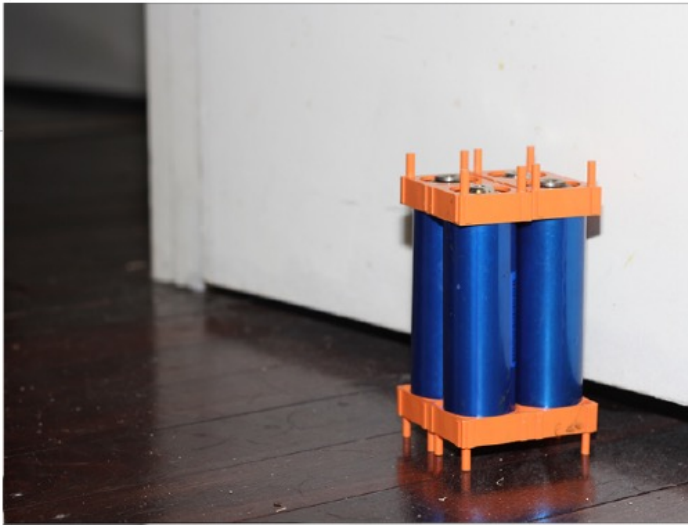
G
R
E
E
N



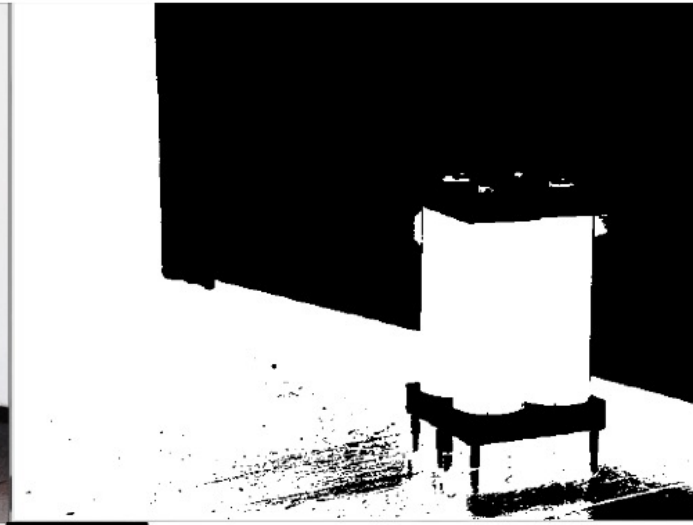
B
L
U
E



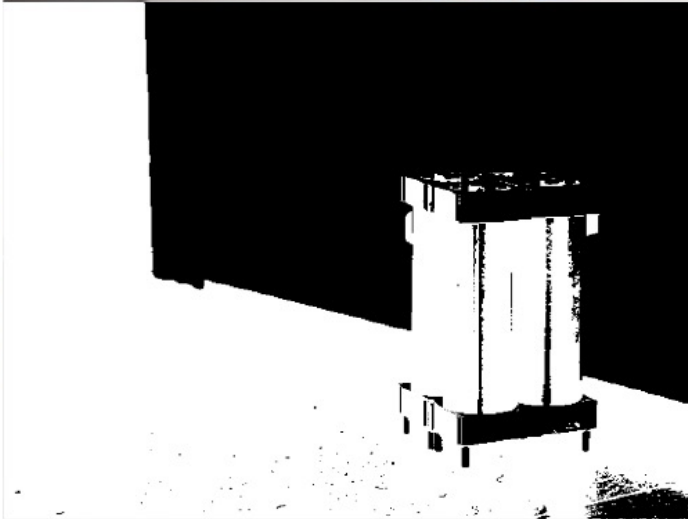
R
G
B



R
E
D



G
R
E
E
N



B
L
U
E



```
if ( image[i][j][0] < 100 && /* -> Red pixel */
      image[i][j][1] < 100 && /* -> Green pixel */
      image[i][j][2] > 200) { /* -> Blue pixel */
    isBlue[i][j] = 1;
  }
```



R
G
B



I
S
B
L
U
E



Alternative Colour Spaces

RGB is one way to represent colour and is probably the most intuitive but we can represent colours with other systems that have various advantage and disadvantages

- HSI/HSL/HSV

- Hue, saturation, intensity/luminance/value (same thing)
- Great for separating out bands of colour (hue)
- Difficult to represent visually

- CMYK

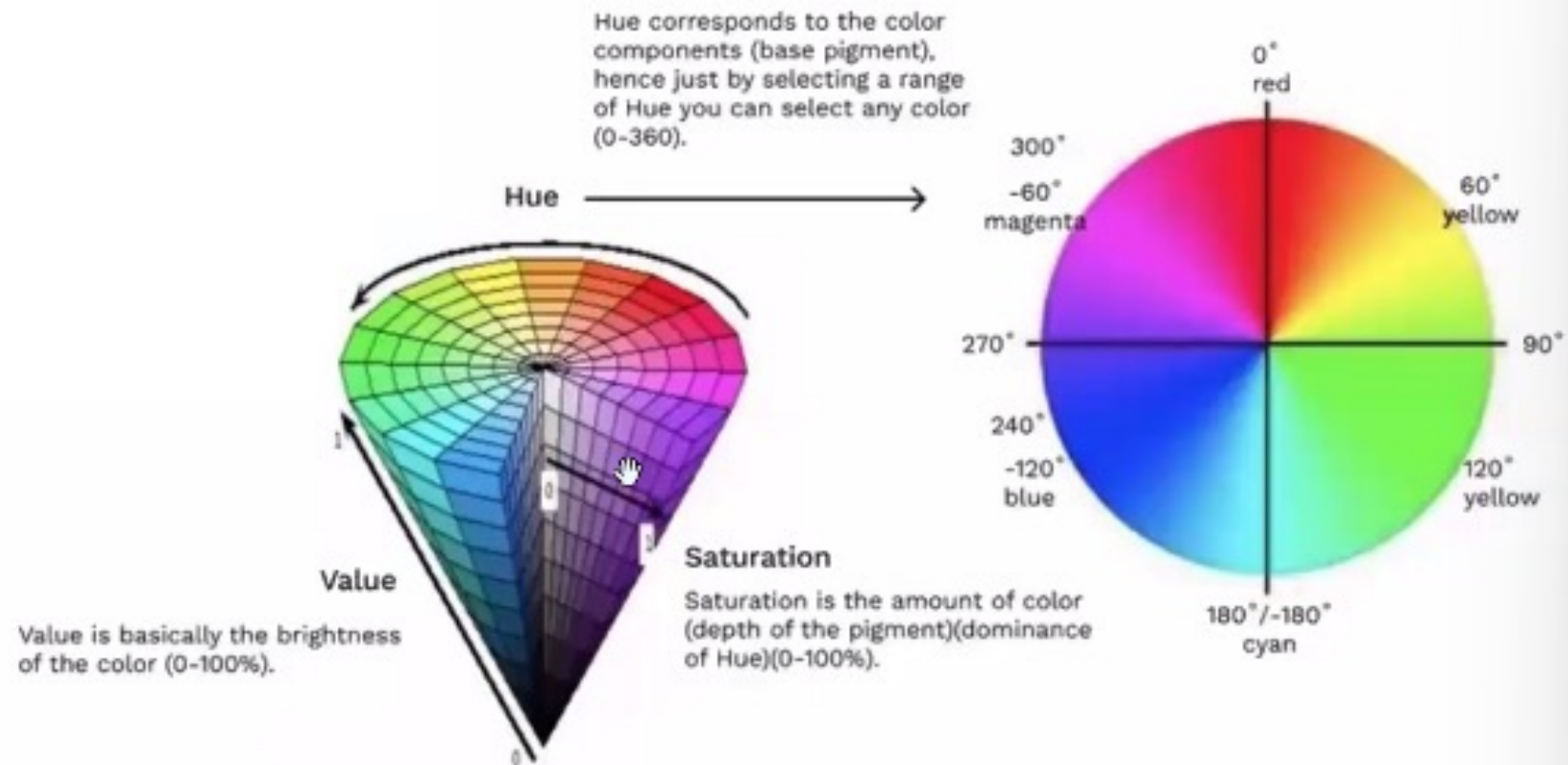
- Cyan, magenta, yellow, black
- Used in printers

- YPbPr/YCbCr

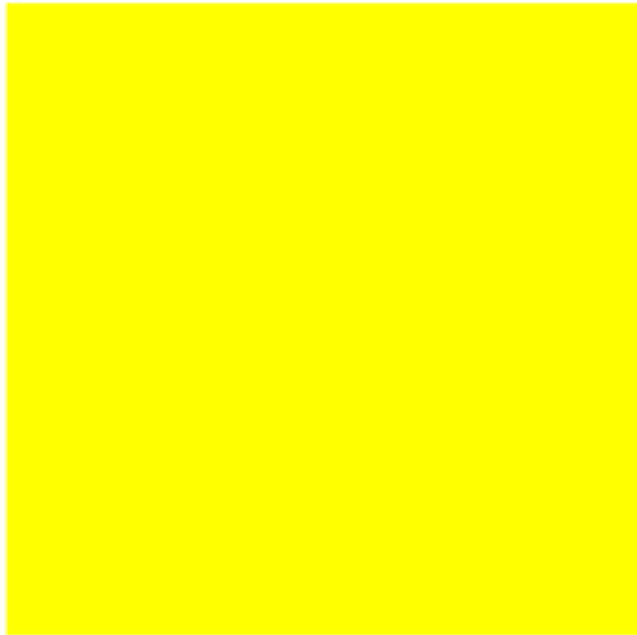
- Used in analogue TV signals / JPEG and MPEG encoding
- Converting between colour spaces is computationally trivial.



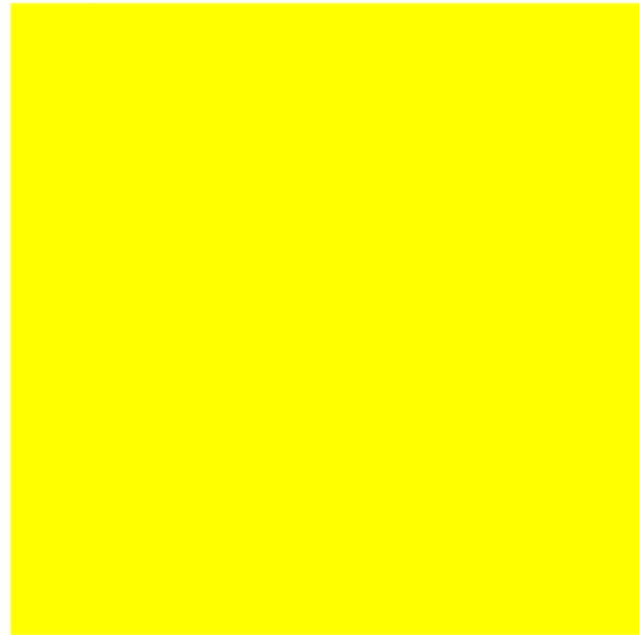
HSV (Hue, Saturation and Value)



HSV Representation (H=60.0°, S=255.0, V=255.0)



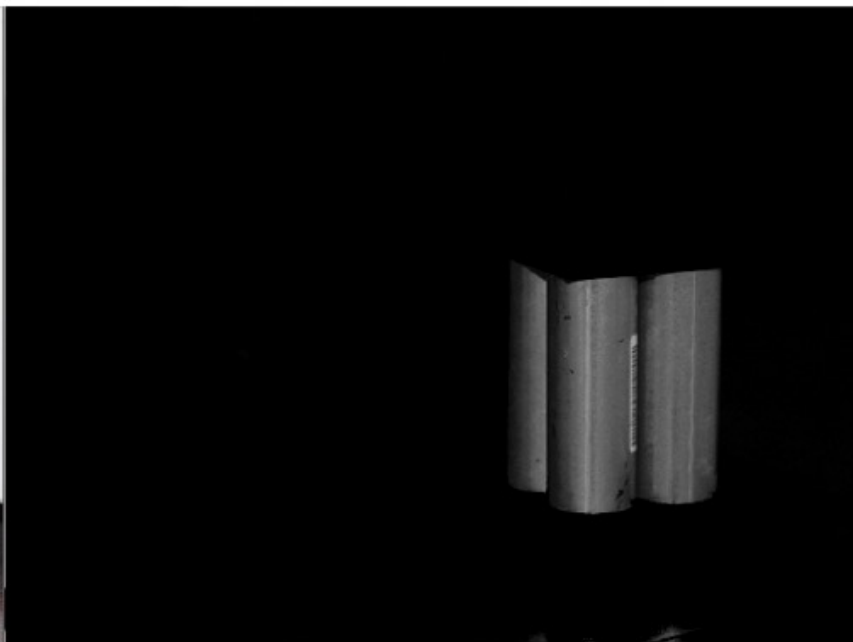
RGB Representation (R=255, G=255, B=0)



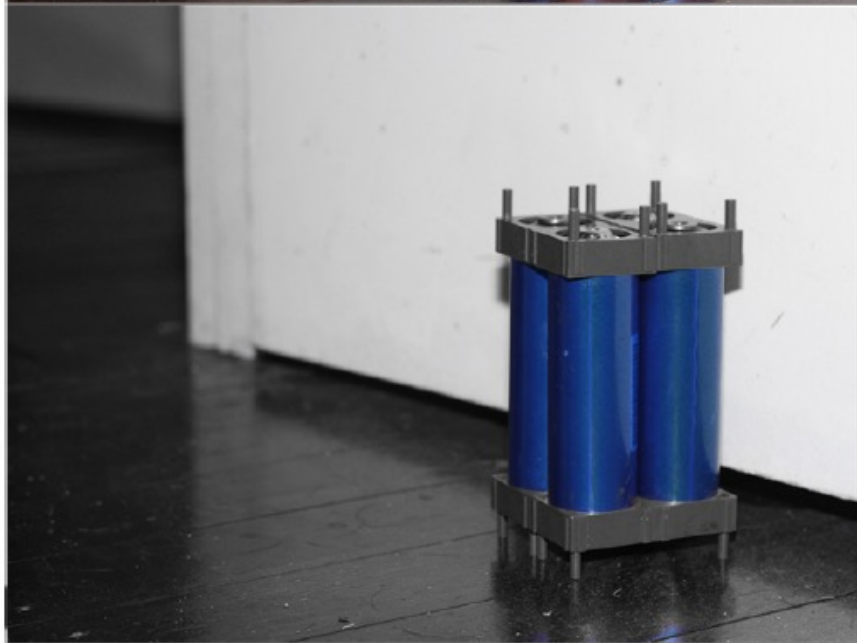
H
S
L



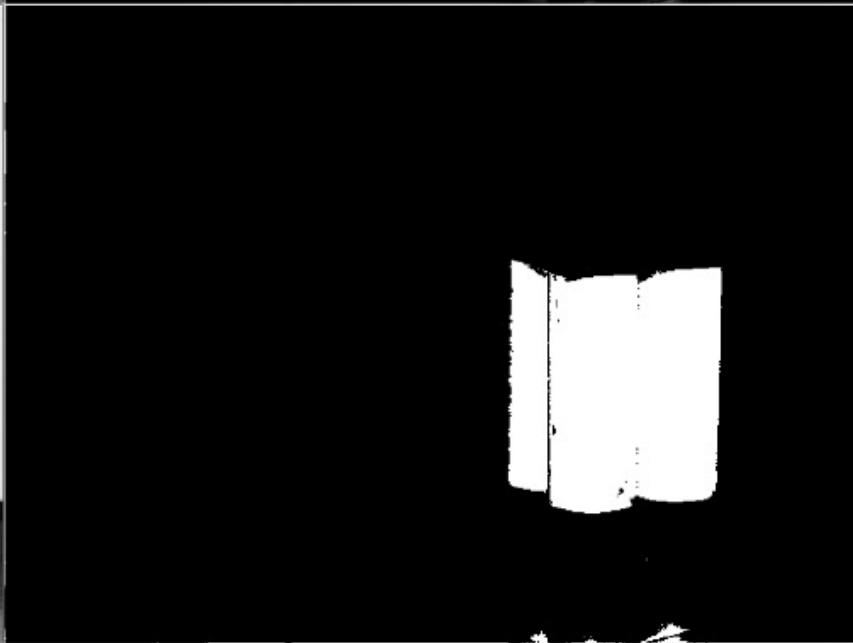
H
U
E



O
V
E
R
L
A
Y



I
S
B
L
U
E



Pointers

Sometimes it is not convenient or possible to pass through variables into another function. In particular, we have no easy way of returning multiple variables.

When we send a variable as an argument to a function, a copy is made rather than the original. Any changes to this copy have no effect on the original.

```
void addOne(int x) { x = x + 1; }  
int n = 6;  
addOne(n);  
printf("%d", n);
```

What does the above print?



Pointers

In these situations, we need to use pointers.

When we declare a variable in C, it is given an address in memory.

A pointer, put simply, is just a memory address.

Be warned – messing up pointers is the most sure fire way to create unstable code.



Pointers

We already know how to declare a variable.

```
int x;
```

How do we declare a pointer? It's actually just a variable with some special notation.

```
int *x_pnt;
```

The '*' before the variable name tells the compiler that this variable is a pointer (in this case, to an int).

To store the address of x in x_pnt we use the '&' symbol – called the 'address-of-operator'.

```
x_pnt = &x;
```

To retrieve the value stored at the memory location we use the '*' symbol again – the 'dereferencing operator'

```
int y = *x_pnt;
```

The dereferencing operator can also be used in an assignment operation

```
*x_pnt = 3;
```



Pointers

Make sure you pay attention with your pointers

`x = y;`

`x = &y;`

`x = *y;`

`*x = y;`

`*x = &y;`

`*x = *y;`

`&x = y;`

`&x = &y;`

`&x = *y;`

Under differing circumstances, these are all valid but all mean very different things.



Pointers

So now we have the address of `x` stored in `x_pnt`, we can send that as an argument to a function and it can operate on it and write to it without having to have the variable within its scope.

```
void doubler_func(int *addr) {  
    *addr = *addr * 2;  
    return;  
}  
  
...  
int x = 1;  
int *x_pnt = &x;  
LCDPrintf("%d\n", x); // this will print '1'  
doubler_func(x_pnt);  
LCDPrintf("%d\n", x); // this will print '2'
```



Memory Allocation

- The TTGO has a DRAM and an external PSRAM.
- By default, your TTGO saves all variables to your DRAM. Your external RAM doesn't get used.
- However, the DRAM is tiny (327680 bytes). Half of this is used for your libraries and frameworks anyway.
- So, when we want to save images, this overflows very quickly.



Memory Allocation

- rgbBuf 57,600
- img 57,600
- Graybuffer 57,600
- combinedImage 57,600
- hue, sat, intensity, Grayimg = $4 \times 19,200 = 76,800$
- **Total $\approx$ 307 KB just in global arrays.**
 - That's basically all of DRAM, and the compiler also needs space for FreeRTOS, WiFi stacks, heap, etc. → overflow.
- Your compiler will tell you beforehand if it detects an overflow, or your TTGO will just crash.



Memory Allocation

- ❑ Solution: use external memory.
- ❑ The TTGO has about 6mb of PSRAM which we can use to save large variables like images.
- ❑ How to use:
 1. Declare the pointer without declaring a size of the array.
 2. In your **setup**, allocate the memory using `ps_malloc`.
 3. Since you're dynamically allocating memory, you can't assume what's held in the memory is clear ie. 0. So, we use the `memset` command to clear everything held in the memory.



Memory Allocation

□ Also note you will need to declare the following in your .ini file:

```
board_build.esp-idf.psram_enable = yes
```

Inside of your build flags:

```
-DBOARD_HAS_PSRAM
```

We have declared these for you in a fresh iteration of the eyebotproject file too.



Memory Allocation

```
BYTE *veryBigImage; // declare pointer

void setup() {

    veryBigImage = (BYTE*)ps_malloc(QQVGA_SIZE);
    // allocate memory to psram

    memset(veryBigImage, 0, QQVGA_SIZE);
    // clear anything contained in these memory
    locations by default.
}
```



Header Issue

- A couple IP functions have their function declarations slightly incorrect (missing a const in the parameters).
- Quick fix – download the new header file on <https://roblab.org/courses/embedded/EyeBot32/> under eyebotlibrary0310. Copy the entire contents of eyebot -> include -> eyebot.h and paste over your existing library folder.
- Long fix – download the project zip file which is now updated with the new header file.



Camera Functions – Taking an Image

Main functions required:

```
CAMGet (BYTE *img) ;
```

Captures a colour image (RGB888) of the size specified when initialising the camera

Remember that the size is 3x the pixel count

```
CAMGetGray (BYTE *img)
```

Captures a greyscale image (0-255)

Notice that this only requires an image of the same size as the pixel count.

**REMEMBER TO INITIALISE YOUR IMAGES TO
THE CORRECT SIZE FIRST!!!!**



Camera Functions – Displaying an Image

`LCDImage(BYTE *img);`

`//displays a colour image (RGB)`

`LCDImageGray(BYTE *img);`

`//displays a greyscale image (0-255)`

`LCDImageBinary(BYTE *img, int threshold);`

`//displays a binarized version of a greyscale image.`

`LCDImageStart(size, x_start, y_start, x_size, y_size);`

`//sets the starting position of the image`

`//useful for printing multiple images to the screen`



Image Processing Functions

`IPCol2HSI(const BYTE* img,BYTE* h,BYTE* s,BYTE* i);`

`// Converts an entire RGB image into its HSI (Hue,`

`// Saturation, Intensity) representation.`

`IPOverlay(BYTE* c1,BYTE* c2,BYTE* cOut);`

`//Overlays a colour image onto another one, where a black //pixel in C2 is perceived as transparency.`

`IPCol2Gray(const BYTE* imgIn, BYTE* grayOut);`

`//Converts an RGB888 image to a grayscale image.`

`IPGray2Col(const BYTE* imgIn, BYTE* colOut);`

`// Converts a grayscale image back to RGB format (still grey though).`

`IPOverlayGray(BYTE* g1,BYTE* g2,BYTE r, BYTE g,BYTE b,BYTE* cOut);`

`// Overlays a coloured mask onto a grayscale image.`

`// not recommended for use in your lab, just use IPGray2Col -> IPOverlay`



Other useful functions

`LCDSetPrintf(row, col, message);`

//prints a string at a position on the screen

`LCDLine(x_start, y_start, x_end, y_end, COLOR)`

//prints a line on the LCD.

`LCDArea(int x1,int y1,int x2,int y2,COLOR col,bool fill);`

// draws a rectangle with optional fill.

Colors are already predefined through the TFT library e.g.
`TFT_BLACK`, `TFT_WHITE`, `TFT_RED` etc.



Eyesim Version 2

- ☐ Can be installed from here: <https://roblab.org/eyesim/>
- ☐ Also uses RoBIOS, so all the exposed functions are the same as the ones on the EyeBot32 and work the same way too.
- ☐ Follow the instructions to install.
- ☐ It is a 3D simulator, and it is quite realistic so it is useful in determining whether your logic or ideas will work.
- ☐ You are not obligated to use it, but it might be useful for you if you want to do some prep at home and test your code before the labs.

