Embedded Systems Lab 9
Image Processing, Pointers, Camera Functions

Tutor: Marcus Pham
marcus.pham@uwa.edu.au

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: Width*Height*3.
- Before we can work on our image, we must allocate enough space for our array. We already have predefined constants for the image sizes:

  QQVGA - 160*120 pixels
  QVGA - 320*240 pixels
  VGA - 640*480 pixels
- As well as predefined sizes to help initialisation of arrays:
  eg. QQVGA_SIZE = 160*120*3,
  QQVGA_PIXELS = 160*120
  QQVGA_X = 160, QQVGA_Y = 120
- Which helps us initialise our arrays easily

```
BYTE colimage[QQVGA_SIZE];
```

Image Processing

- In order to ensure our camera collects the images of the correct size we must also initialise our camera to the corresponding size:
  eg. CAMInit(QQVGA); //using QQVGA ie. 160*120
- 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.

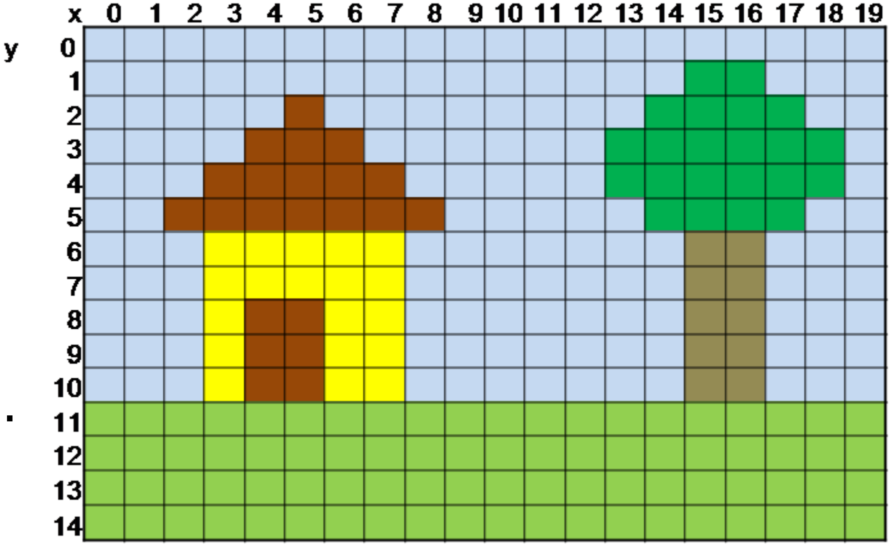
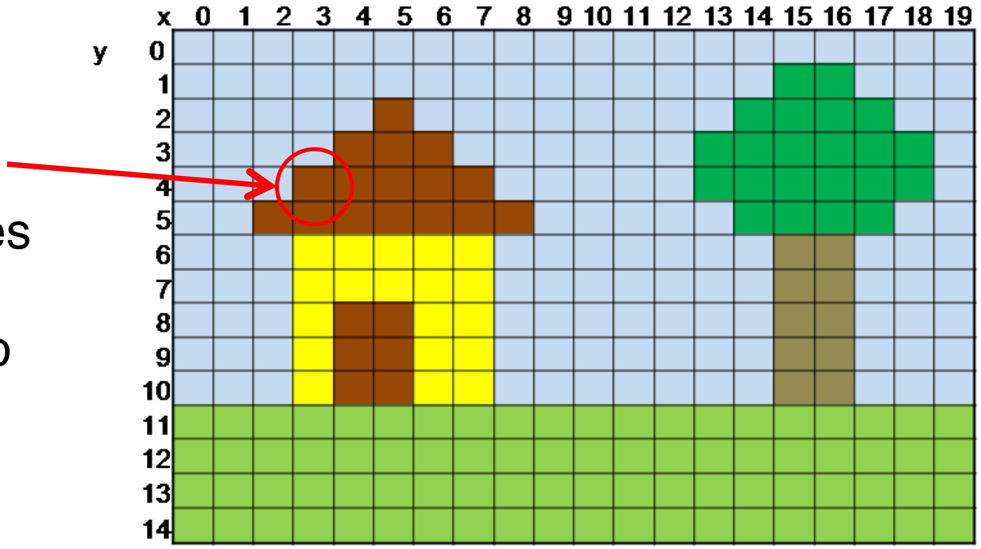


Image Processing

- 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.

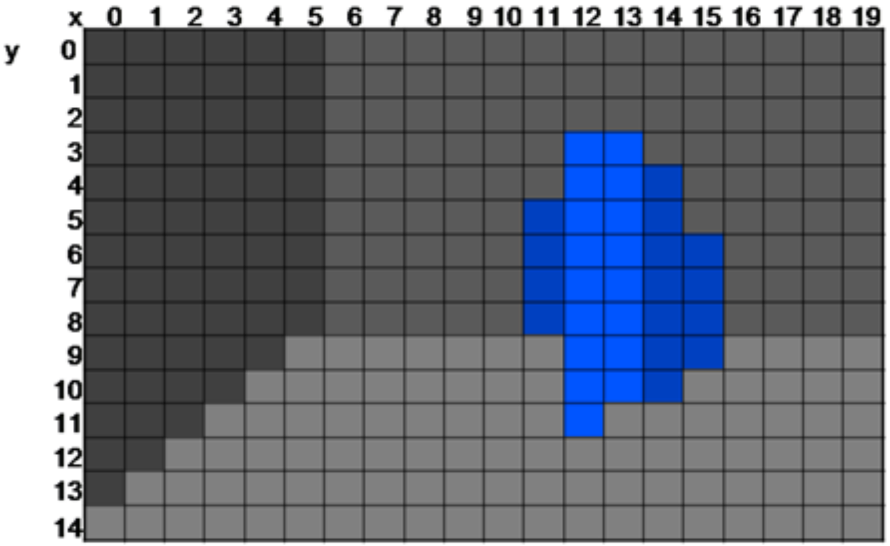


Image Processing

- 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 lets make another array.

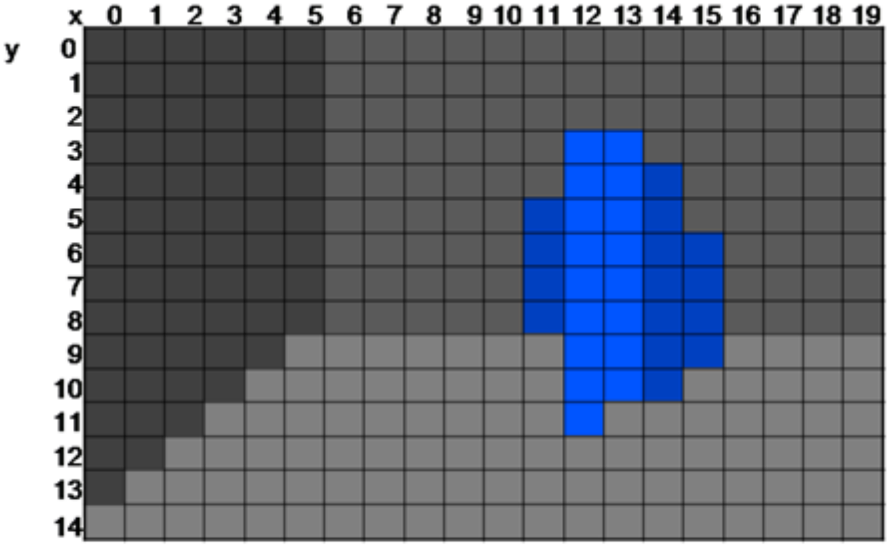
```
…
int isBlue[15*20] = {0};
int i,j;
for (i = 0, i < 15,i++) {
  for (j = 0, j < 20, j++) {
      if (image[(i*20+j)*3] < 100 &&
            image[(i*20+j)*3+1] < 100 &&
            image[(i*20+j)*3+2] > 200) {
                  isBlue[i*20+j] = 1;
      }
  }
}
```

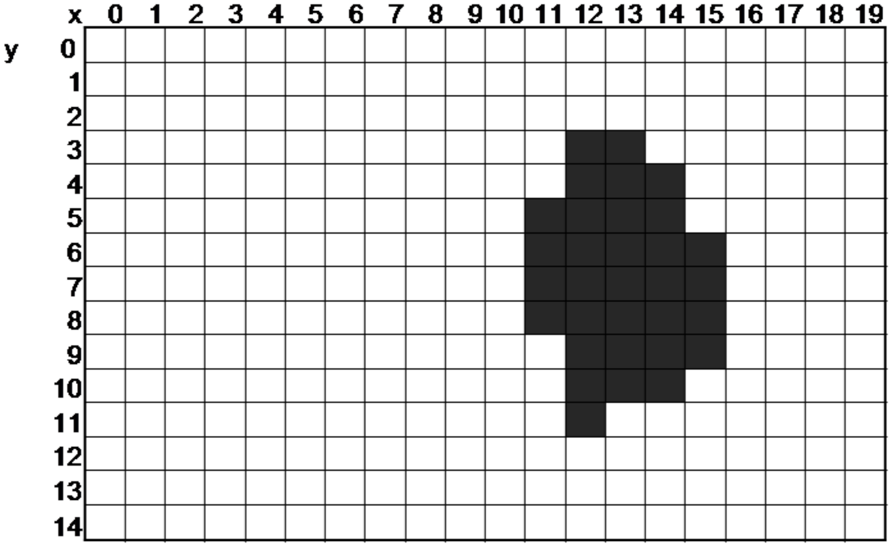
Image Processing

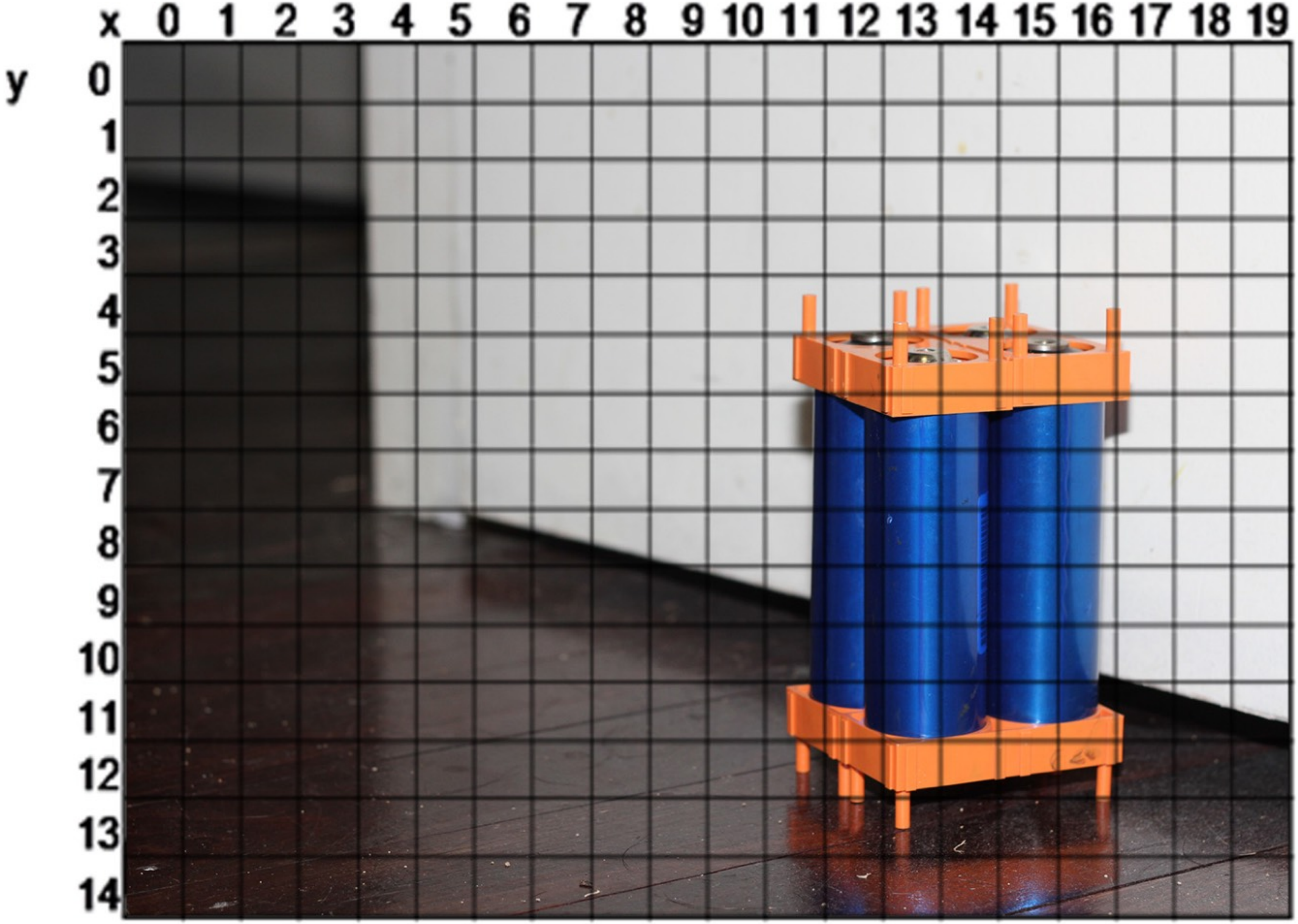
- Pro tip!
  - Do not use the values from the previous slide as your thresholds for matching a colour.
  - You will need to determine your own values based on camera/lighting/etc.

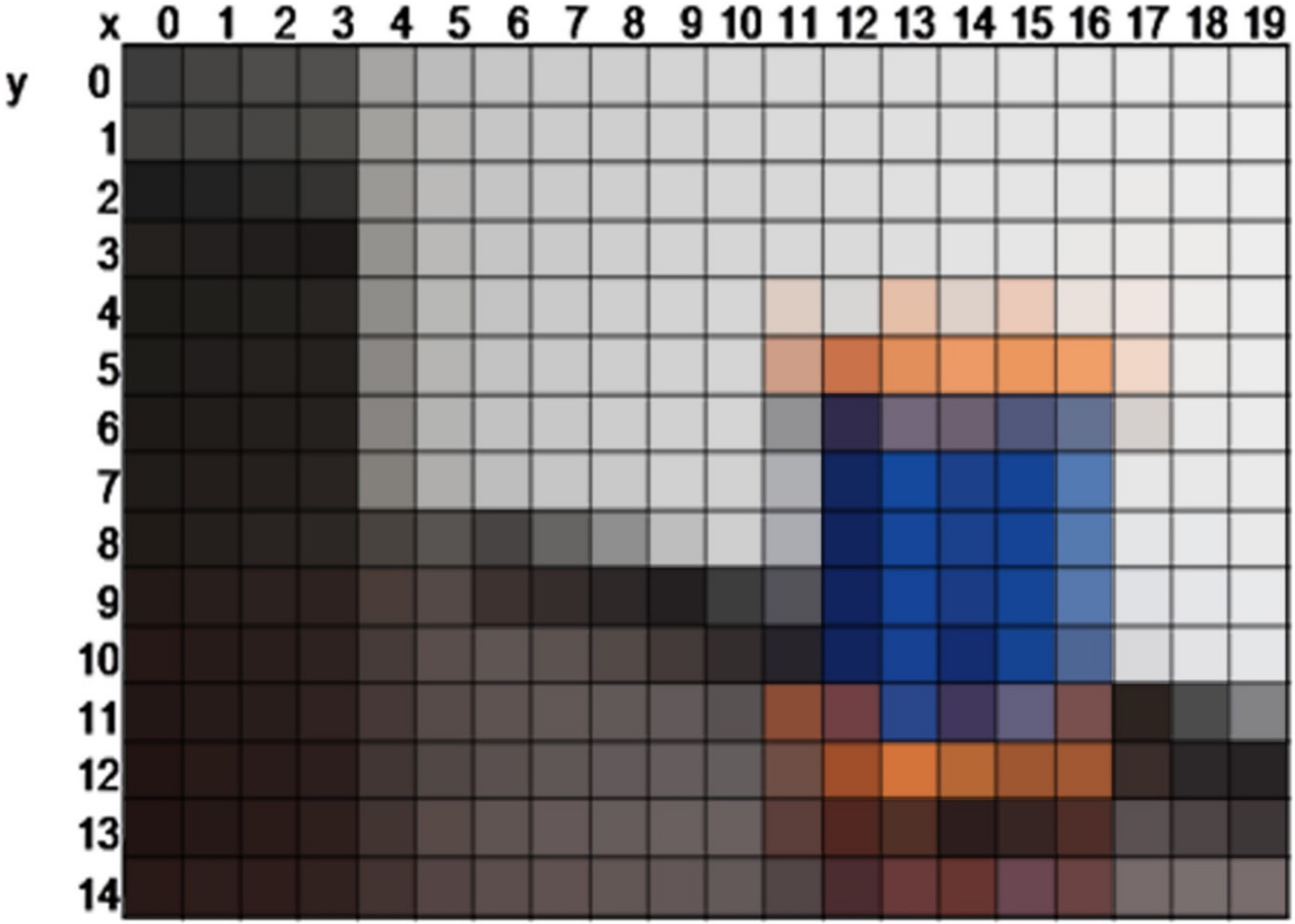
Image Processing

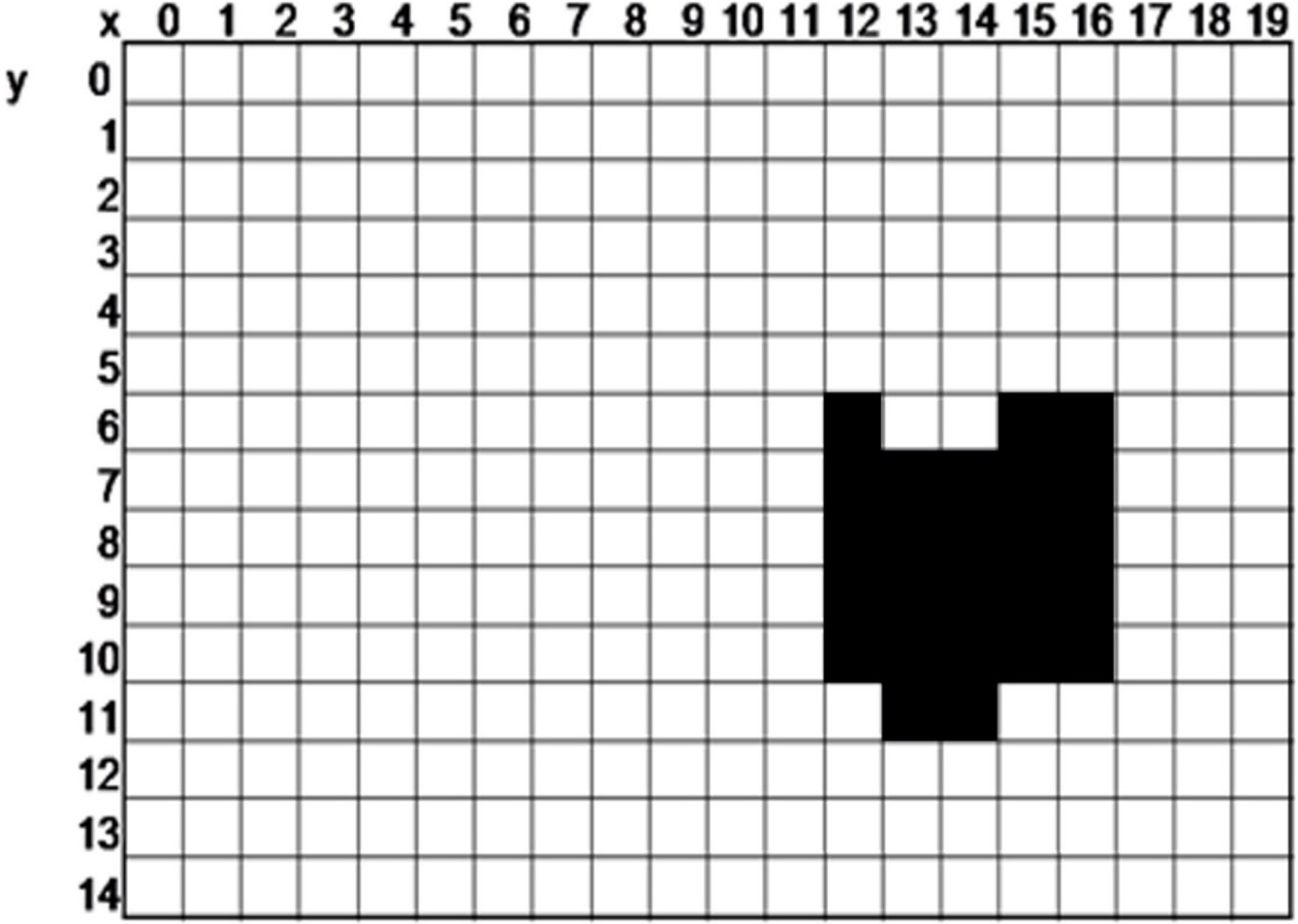
- We now have an array of array of a integers where pixels that are 'blue' are now '1's and those that aren't are '0's.
- If we take the left image as input and were to display the output of this code as a binary image we'd have the image on the right (where '1's are displayed as black and '0's as

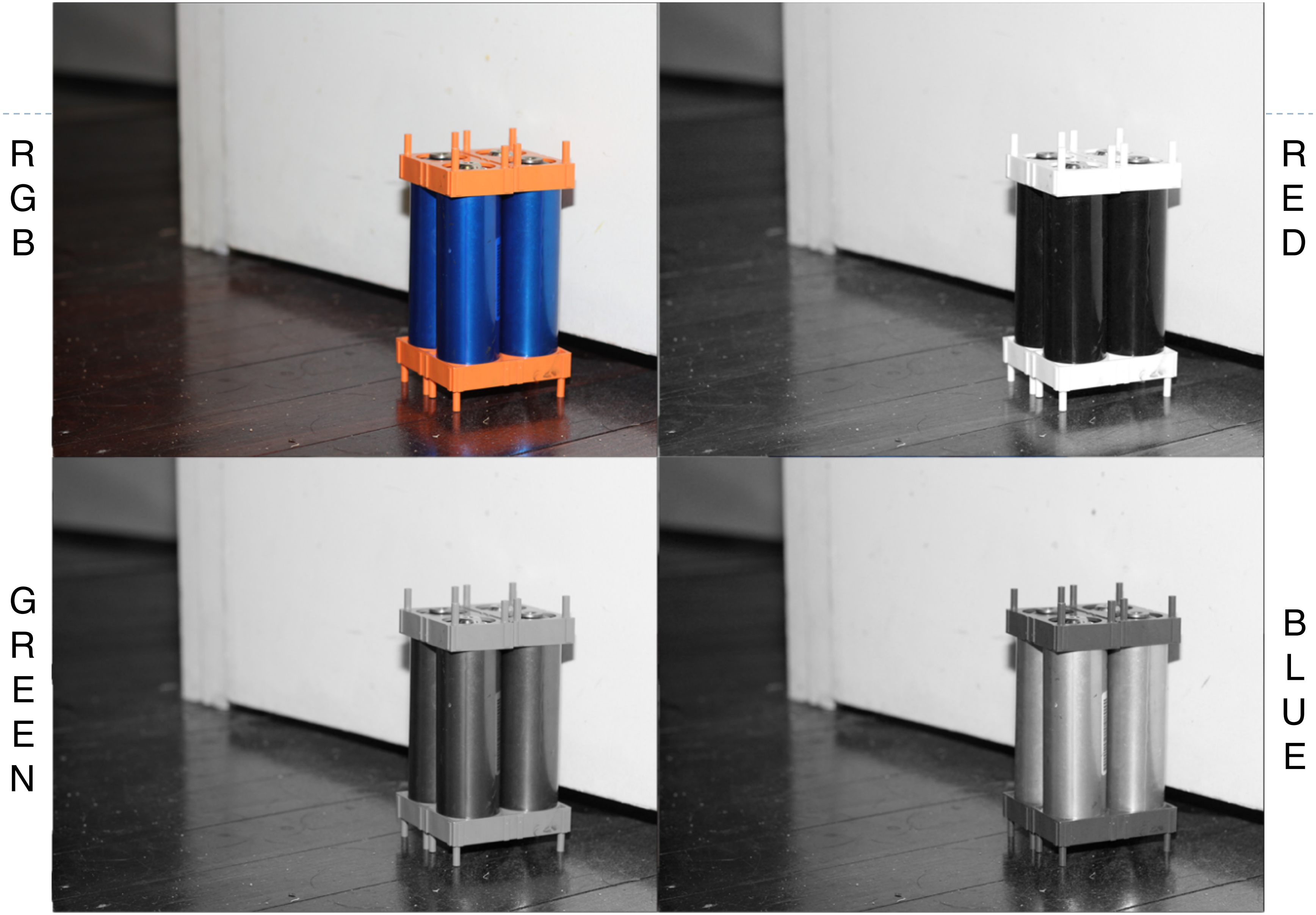


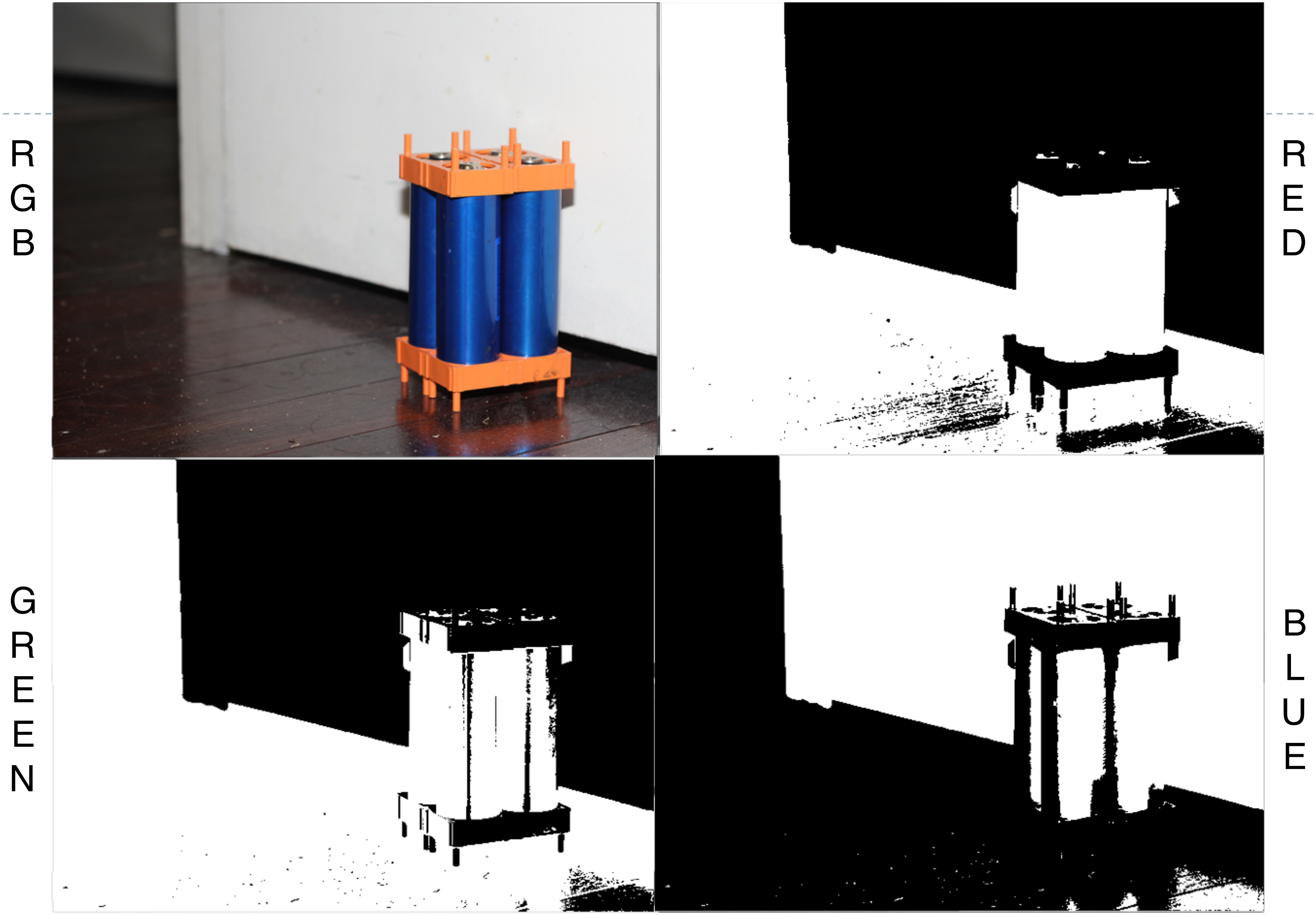



x
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

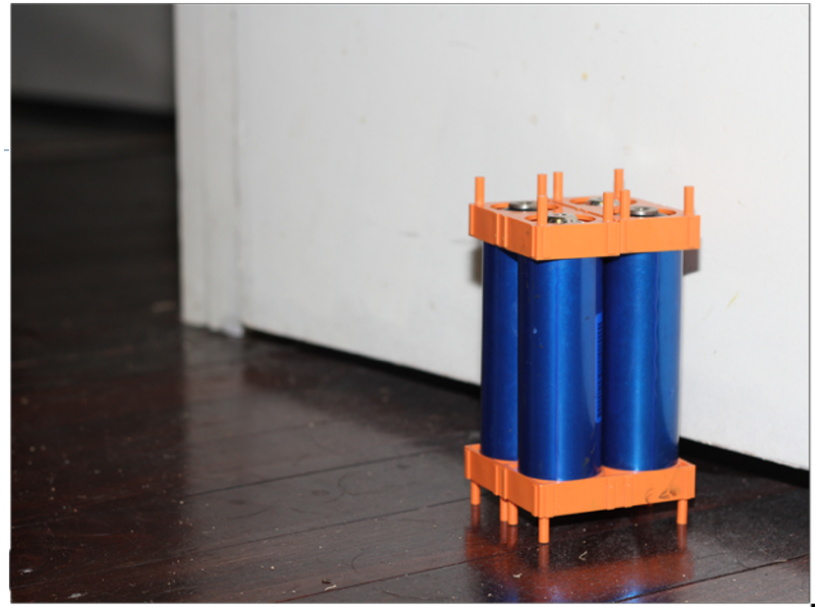

x
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

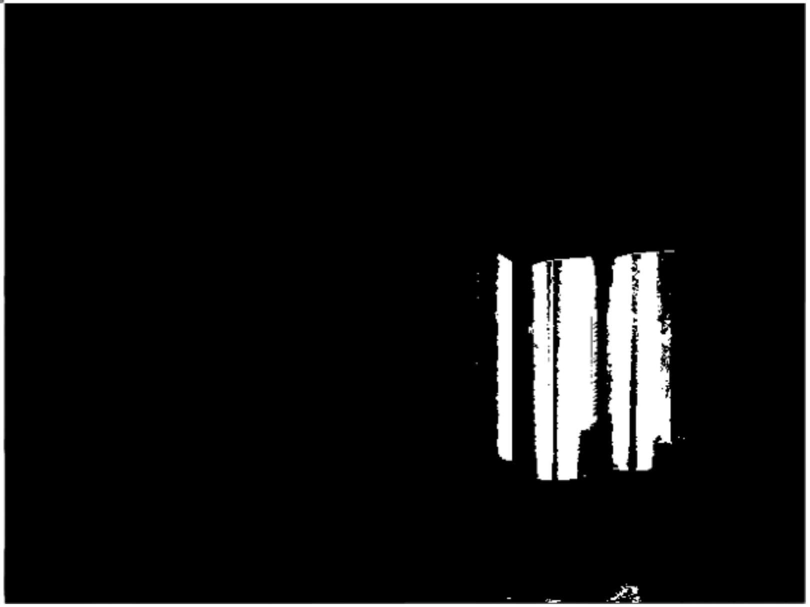

x
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14


RGB
RED
GREEN
BLUE


RGB
RED
GREEN
BLUE

R
G
B

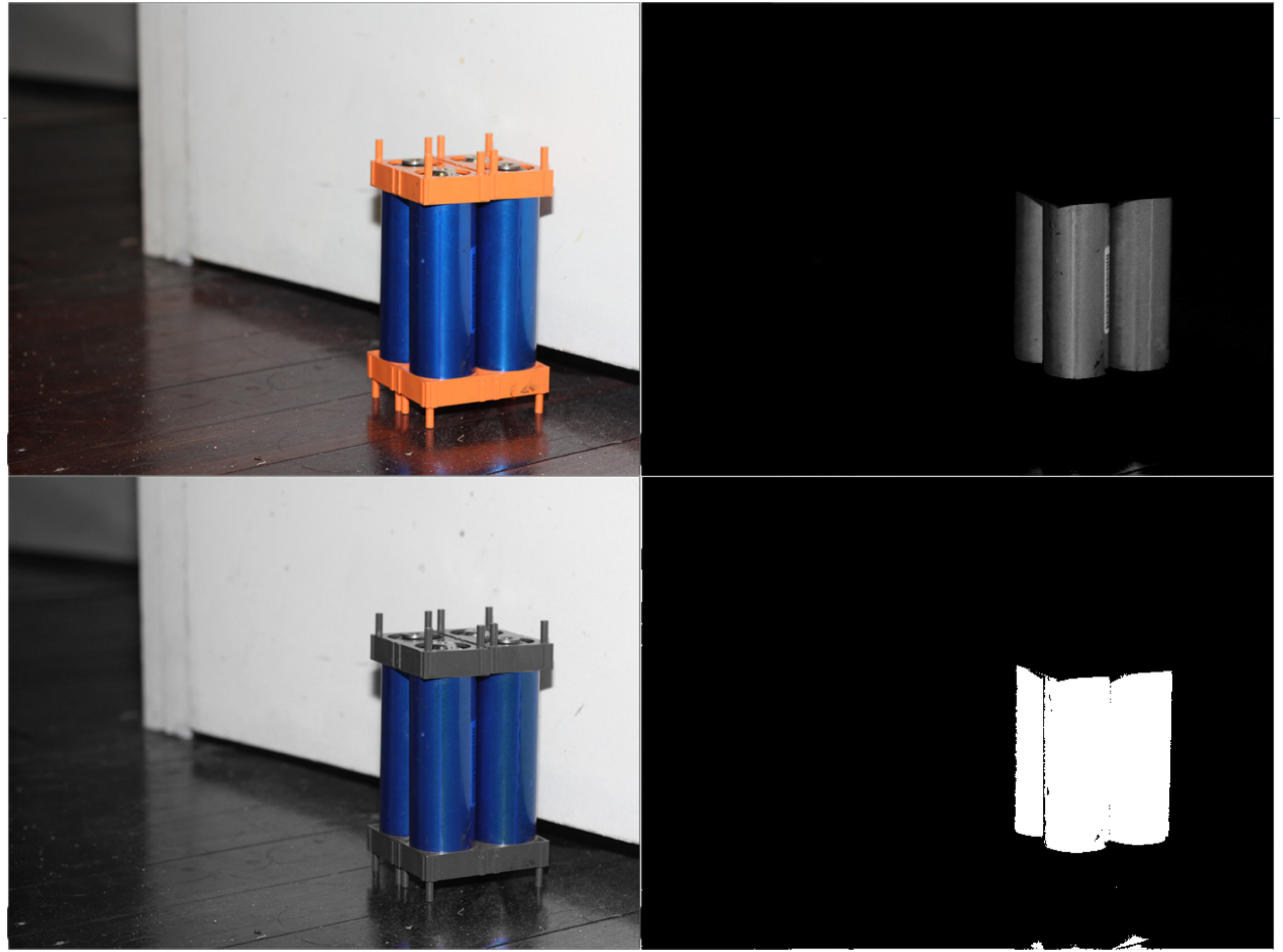




I
S
B
L
U
E

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
  - HSV/HSL
    - Hue, saturation, value/luminance
    - 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.


HSL
ISBLUE

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.
- 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;    /* the ‘*’ makes it return
                        the value at addr */
  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’*/
```

Pointers

- Common mistakes with pointers
- Missing notation eg

  `x_pnt = x;` instead of `x_pnt = &x;`

- When declaring multiple pointers eg

```
/* the following line of code is valid, however - */
int* x_pnt, y_pnt;     /* only x_pnt is a pointer */
/* make sure you declare each as a pointer */
int *x_pnt, *y_pnt; /* both are now pointers */
```

- Pointer manipulation

Camera Functions
Initialisation and Release

- int **CAMInit** (int size);
  //initialises the camera to the size specified.
  Sizes:
  QQVGA - 160*120,
  QVGA - 320*240,
  VGA - 640 *480,
  CAM1MP - 1296*730,
  CAM5MP - 1920*1080

- int **CAMRelease** (void);
  //releases the camera

Camera Functions Initialisation and Release

```
...
int error = CAMInit(QQVGA);      /* initialise camera */
if (error!=0) LCDPrintf(“Camera initialisation error”);
...
/* get images from camera, do image processing, etc */
...
error = CAMRelease();             /* release camera */
if (error!=0) LCDPrintf(“Camera release error”);
...
```

DON’T FORGET TO RELEASE THE CAMERA AT THE END OF YOUR PROGRAM!

Camera Functions
Retrieving an Image

- Main functions required:
  CAMGet(BYTE *img);
  //captures a colour image (RGB) 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 sam 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);
  //displays a binary image (1 or 0)

  LCDImageSize(size);
  //sets the expected image printing size
  //remember to set this first!!!

  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

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.
  Colors are already predefined eg. RED, BLUE, YELLOW, ORANGE etc.

- **REMEMBER that if you are printing anything to the LCD, you must use either the touchscreen (by placing your binary into the ~/usr/ folder) or through remote desktop.**

- **Function lookup available at:**
  http://robotics.ee.uwa.edu.au/eyebot7/Robios7.html