# ENGR101 T1 2024

Arthur Roberts

School of Engineering and Computer Science
Victoria University of Wellington

# Project 1 observations

Most people managed OK. Some observations, recommendations and model answers.

It takes too long - reconsider your approach.

Usually submitted code for the Challenge is one big **main** function, up to 100 lines long.

From what I seen testing involves whole cycle:

- make code change
- run the program
- have a look at waveform

If we want to make it faster we have to make it simpler.

- Design your algorithm. Not in all details.
- Do not start typing code until you know what you want to achieve.
- Do not start testing whole program until you are sure of the parts.
- **just trying** is a waste to time, design it and it should work as designed
- Do not change things randomly, chances of finding correct answer this way are zero.
- Break your algorithm into functions.
- Code one function after another.
- Add in small increments. Compile, build, run continuously.
- Combine functions into working program.

# As an example - Challenge of Project 1

General design: what are problems for the challenge?

- read the file
- adjust the volume inside the each note

Can these problems be solved without writing the rest of the program?

# Project 2 - hardware

It runs on Raspberry PI - small microcomputer.
It can not be done on PC (unless you manage cross-compile, which is not easy at all).
It is done to prepare you for big Project 3.

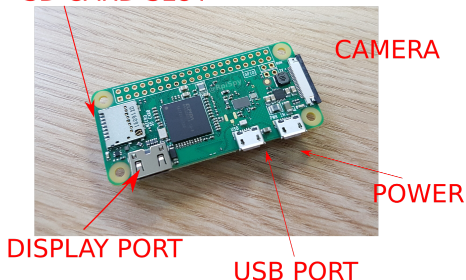



Figure: RPI

- starts once power is applied
- boots in one, two minutes without password
- OS and all files are stored on SD card
- can connect to WiFi

# Project 2 - hardware

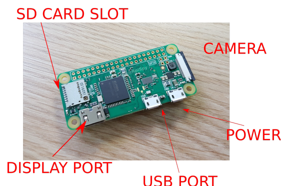



Figure: RPI

- Runs customized Linux
- Starts to text mode. If you want graphical interface - type in **startx**
- C++ compiler and Geany installed
- Custom library **E101.h** to provide custom functions to work with camera and robot hardware
- If you want to work as super-user:
  - Username - **pi**
  - Password - **raspberry**

Code (cpp file) will be stored on RPI. If you want to transfer it to your home folder -

# Project 2 images: What are images in memory?

Project 2 goal is to create video processing software.

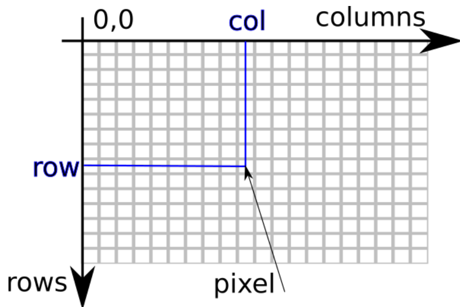



Figure: Grid. Pixels

Raw images are grids of pixels. Each pixel contains three values:

- red
- green
- blue

Sometimes transparency is added.
Usually it is one byte (**char**) for each of the colours.
Makes it 3 bytes for each pixel.
Maximum value of 1 byte is :
$11111111_2 = 255_{10}$

# Raster vs vector

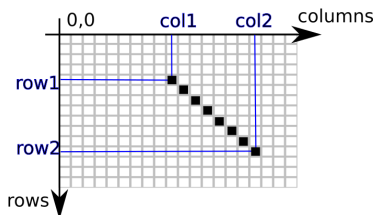



Figure: Vector vs raster

- These pixel arrays are massive. Very small image 320 pixels wide and 240 pixels high will require 320*248*3 = 230400 bytes to store. We are talking about **raster** images here - image is represented as a set of pixels.
- Another way to store the image is to store image components: lines, letters, circles, etc combined with information about their color, size and position. Images stored like that are called **vector** images.

# Working with real images: Converting images

- Going from vector to raster form is easy - just set the pixels following each shape. It is subject of **computer graphics**.
- Another way around is more challenging. It is called **computer vision** and it is hard. In simple case you can find simple shapes (rectangle, ellipse, etc. ) in the image .
  That is not **compression**.
- If you want program to understand what object(car, sheep) is in the image - it is task for **Machine Learning/Artificial Intelligence**. We will cover some of it in ENGR110.

# Helper software

Do not forget to include file **E101.h**. As usual, it should be in the same folder as your program.
If you run your code from command line - use **sudo ./program_name**
There are following functions available:

- **unsigned char get_pixel(int row,int col, int color)** - returns red or green or blue value of the pixel at (row,col) point. If colour is 0 - returns red, if color is 1 - green, if 2 - blue, if 3 - (r+g+b)/3 (luminosity)
- **take_picture()** - makes camera to take a picture and store it into E101.h array. Image is 320 columns by 240 rows.
- **update_screen()** - displays last taken picture in top left corner of the screen. Works in text mode too. Has no window controls.
- **int set_pixel(int row, int col, char red, char green,char blue);** - sets pixel in E101.h memory. Pixel remains modified until next **take_picture()**.

# Helper software - starting code

Listing 1: starting code

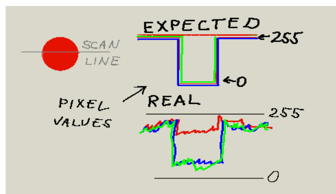
```
int totRed = 0;
int totInt = 0;
double redness = 0.0;
// for all pixels in latest image
for (int row = 0 ; row < 240 ; row++) {
   for (int col = 0; col < 320; col++) {
    totRed = totRed + (int)get_pixel(row, col, 0);
    totInt = totInt + (int)get_pixel(row, col, 3);
    redness = (double)totRed/(3.0*(double)totInt);
   }
}
```

Think about what it really calculates...

# Working with real images

Images coming from the camera are not perfect.
There will be no perfect perfect white (255,255,255) or perfect black (0,0,0) pixels in the camera picture.
There are shadows - bright red can go to dim red as shadow moves and covers red part of the image.
Red does not mean that pixel RGB values are (255,0,0). We perceive (240,90,90) as bright red and (120,30,30) as dim red. It is still red though.
If you want to do the completion, you need to invent some criteria of what red is. And it should work for both bright red and dim red.

# Working with real images




- In ideal case we can expect for white pixels: r=255, g=255, b=255; for red pixels: r=255;g=0;b=0.
- No colour is perfect, pixels colour value will be not 255 and not 0. Only perfect red would be (255,0,0) - no such thing in real images.
- R,G,B values change slightly from one pixel to next: **noise**
- Whole picture can move up and down - camera auto exposure system

# Challenge part

We will try to keep "redness" of the image same.
Only way to detect the "robbery" is to detect movement of red pixels.

Questions?