Colour-based Object Detector: Code Challenge

See the overview slide for a description of the task.

The scaffold provided here includes:

- bitmap.c/bitmap.h: functions for image handling, conversions to HSV and computing the difference and midpoint between two HSV values
- "images" folder: contains example images (don't delete)
- "test_calibrations": contains example calibration files (don't delete)

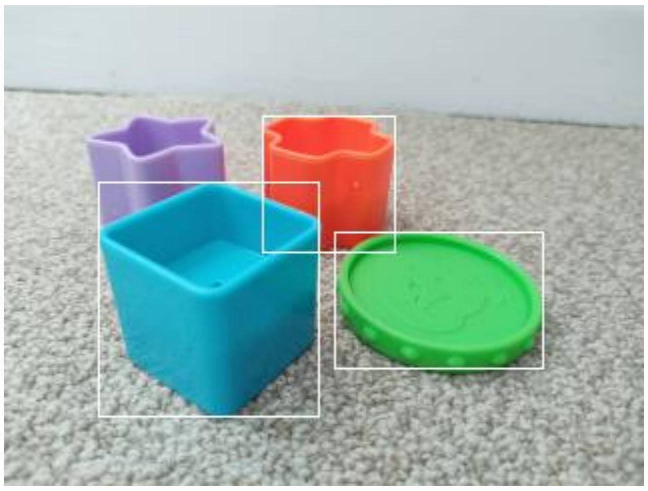
Building, Running and Marking

Your program is expected to be built, without errors, using a **Makefile** and should build an executable called **cam_detect.** During testing, your program will be built using:

make clean && make

Some of the example images and calibration files in "images" and "test_calibrations" are used during testing, so make sure that you don't edit or delete these files.

Overview



In this assignment you will develop an image-based object detector that locates various different types of objects in images based on their colour. Your program will return the bounding box coordinates of each detected object in the image, based on pre-calibrated colour profiles. Your program will also provide the ability to calibrate the colour profile of objects for detection from example images

Background: Images and Colour

Standard colour images contain a two-dimensional array of pixel data, where the colour of each pixel is represented by the component brightness of Red (R), Green (G) and Blue

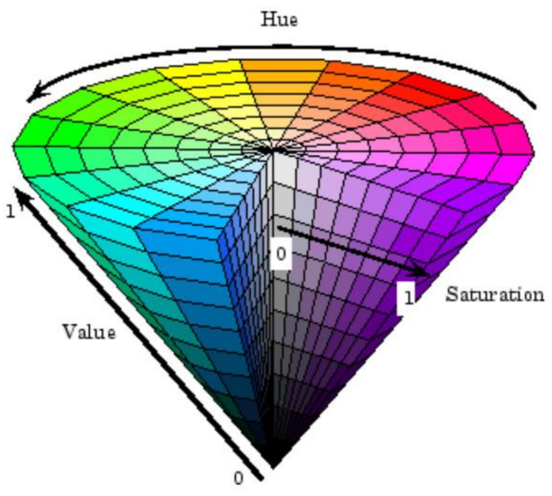
(B) light (similar to the tri-chromatic receptiveness to light by the human eye). In many image formats, each of the Red-Green-Blue values is stored as a number between 0-255, and the combinations of the strength of these different RGB values gives the gamut of colours visible by the human eye, for example:

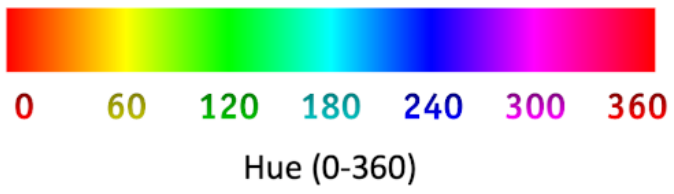
- (RGB) = (255,0,0) indicates pure red
- (RGB) = (0,255,0) indicates pure green
- (RGB) = (0,0,255) indicates pure blue
- (RGB) = (255,255,0) indicates yellow (i.e. a mix of green and red)
- (RGB) = (255,0,255) indicate purple/magenta (i.e. a mix of red and blue)
- (RGB) = (255,128,0) indicate orange (mostly red, with a bit of green)
- (RGB) = (0,0,0) is black and (RGB) = (255,255,255) is white

The combination of (RGB) gives 256 x 256 x 256 = 16,777,216 different possible colours.

The "colour" corresponding to an (RGB) mapping can also be represented in alternative ways using difference colour "spaces". For example, *Hue-Saturation-Value* (HSV) is an alternative set of three numbers which can also be used to represent colours observable by the human eye:

- **Hue** (the "pure" colour i.e. red to yellow to green): represented by a number from 0 to 360 that maps the angle on a colour wheel
- **Saturation** (the "vibrancy" of the colour. The opposite to how much the colour is diluted by white light): measured from 0 to 100, where 0 is gray (totally diluted) and 100 is a vibrant colour
- **Value** (overall brightness/darkness of the colour along a line from black to white): measured between 0 to 100, where 0 is black and 100 is a bright colour





Many computer vision algorithms based on colour use a HSV representation of image pixels to threshold on colour. For example, one way to find "yellow" objects is to convert pixels from (RGB) to (HSV) and then look for pixels that have a Hue (H) value somewhere close to 60 (the Hue for yellow). The advantage of using Hue, as opposed to (RGB) values directly, is that I can can find both dark and bright yellow objects, which should have the same Hue (although differing Saturation and Value).

RGB to HSV Conversion

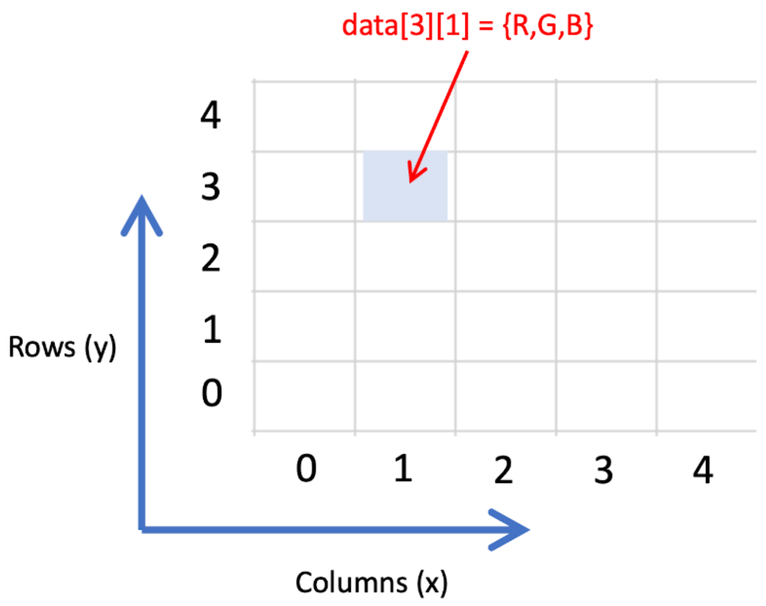
In this assignment, you will be provided with scaffold code that can be used to convert between (RGB) and (HSV) formats (see Section "Pre-provided code in the scaffold" below). Functions are provided that can be used to convert from an (RGB) set of values to (HSV).

Additionally, functions are provided that return the distance between two Hue values and the midpoint of two Hue values. Note from the figure above, that Hue is an angular measurement such that Hue = 0 is the same as Hue = 360 (i.e. Red). So, for example the distance (or difference) between Hue = 300 (Magenta) and Hue = 60 (Yellow) is 120 (NOT 300-30=270) and the mid-point of these two Hues is Hue = 0 (or 360), Red, according to the closest direction of travel around a colour wheel.

Use of these functions is described further in the below sub-sections (see "Bitmap.h", "Bitmap.c" in the scaffold code).

Background: Image Coordinates

Image data is fundamentally a two-dimensional array of pixels. Image coordinates (x, y) represent the index in both the horizontal and vertical directions of the image of a corresponding pixel, and are measured from the bottom left-hand corner of the image:



For example the coordinate (0,0) is the bottom left pixel of an image, and one pixel up from this in the vertical direction is the coordinate (0,1). Pixel data (in this assignment) is stored such that the rows (y) are in the first dimension of the data and the columns (x) are in the second dimension. A third index of the array represents the colour channel (i.e. Red: 0, Green: 1 and Blue: 2).

Background: the BMP File Format

In this assignment, you will work with images stored in the BMP (Bitmap) format. You will be provided with scaffold code that can be used to read in a Bitmap image file and also save image data to a Bitmap file (see "Bitmap.h", "Bitmap.c" in the scaffold code, and further details of the provided functions in the below sub-section "Pre-provided code in the scaffold").

Bitmap files (*.bmp) have a relatively simple file format. A bitmap file consists of a 14-byte general header followed immediately by a DIB (Device-Independant Bitmap) header containing metadata about the pixel format and image size. The pixel data follows immediately after the DIB header. The size of the pixel data can be calculated from the information in the header. Several example images are provided in the scaffold.

Running the hex dump utility
xxd <file_name.bmp>
in the terminal window is a useful way of displaying the encoded BMP information. For more information see https://en.wikipedia.org/wiki/Hex_dump

Bitmap File Header

A bitmap file always begins with a 14-byte header that describes the contents of the file. The bitmap file header always starts with the **2 bytes** 0x42 0x4d (i.e. BM). The next **4**

bytes give the size of the file in bytes. The next **4 bytes** are reserved. The next **4 bytes** indicate the offset from the start of the file to where the pixel data start.

DIB Header

The bitmap file header is always immediately followed by the DIB header. The first **4 bytes** of the DIB header indicate the size of the DIB header. The next **4 bytes** indicate the number of pixels in the horizontal direction. The next **4 bytes** indicate the number of pixels in the vertical direction. The remainder of the DIB header contains additional information about the image, such as colour data and compression, that is not relevant to this assignment.

If you are interested in learning more about the BMP file format see https://en.wikipedia.org/wiki/BMP_file_format

Image Pixel Data

Each pixel in the image is represented by **3 bytes** that give the values of the red, green and blue (R, G, B) content of the pixel. Each of the 3 bytes can have a value between 0 and 255 that indicates the amount of R, G or B in a pixel.

Overview of the Task

The aim of this assignment is to develop a program (called "cam_detect") that can detect objects with specific pre-calibrated colour profiles (using the Hue-Saturation-Value (HSV) colour space) in Bitmap images. Your program will have two main types of functionality:

1. **Object Detection:** Your program will read in an image file and an ASCII Calibration file (which contains colour thresholds for a pre-calibrated object) and detect instances of this object in an image based on colour. Your program will threshold which pixels are within the calibrated colour profile, determine "connected" regions of these pixels that are above a minimum size, and return the set of bounding box coordinates in the image space ((x,y) of the lower-left corner of the box and the width and height) for each region.

2. **Colour-profile Calibration:** You program will read in a "calibration" image of a proposed target (i.e. a normal image where the object is centered in the image) and measure the range of Hue values present in the middle 50-by-50 pixels of the image to determine it's object colour profile. This profile will then be printed out to the screen, and can be piped to a file which can be used for detection of this object in other images.

This functionality is similar to the Pixy camera (https://pixycam.com/), a popular high-speed machine-vision camera system.

Program Input Format

The program will be run from the command line in the following format:

```
./cam_detect <mode (s|d|c)> <calibration filepath or name> <image filepath>
```

The first argument is a single character in {s,d,c} which indicates the mode of operation:

- **Mode 's' (show calibration):** in this mode, the program just displays back to the screen the contents of a provided calibration file. The last command line argument is not required, e.g. ./cam_detect s calibration001.txt
- **Mode 'd' (detect):** in this mode, the camera detects objects in the image that fit one of the profiles in the provided calibration file (the argument <calibration filepath or name>) e.g. ./cam_detect d calibration001.txt myimage.bmp
- **Mode 'c' (calibration):** in this mode, the program outputs calibration data based on measuring the object at the center of the provided image and gives it the label provided in the second argument <calibration filepath or name>, e.g. ./cam_detect c redapple redapple_image.bmp outputs the line of calibration data with the object label redapple

If the user provides an invalid input mode or an incorrect number of arguments for the corresponding mode, the program should return the message:

```
Incorrect input
```

and then exit. If a calibration file is missing or cannot be opened, your program should return the message:

```
Could not open calibration file
```

and exit.

Colour Profile Calibration file:

A colour calibration file is a simple text file, where each line of the file contains information about the name of a specific object type and details of the HSV profile which can be used to detect it. An example calibration file (used with the coloured blocks shown in the image at the top of this page) looks like:

```
blueblock 191 4 50 30

coin 132 7 50 30

orangeblock 10 10 50 30
```

where for each line is in the format:

<object_name> <Middle Hue> <Max. Hue Difference> <Minimum Saturation> <Minimum Value>

<Middle Hue> represents the nominal Hue of the corresponding object (between 0-360), and <Max. Hue Difference> represents maximum deviation from this Hue to still be considered as part of the object. <Minimum Saturation> and <Minimum Value> represent the minimum S-V (between 0 to 100) to be considered as part of this object (i.e. Saturation and Value of a pixel also have to be above or equal to these values to be considered as part of an object). All values are stored in integer format.

A calibration file can contain profiles for one or any number of different objects.

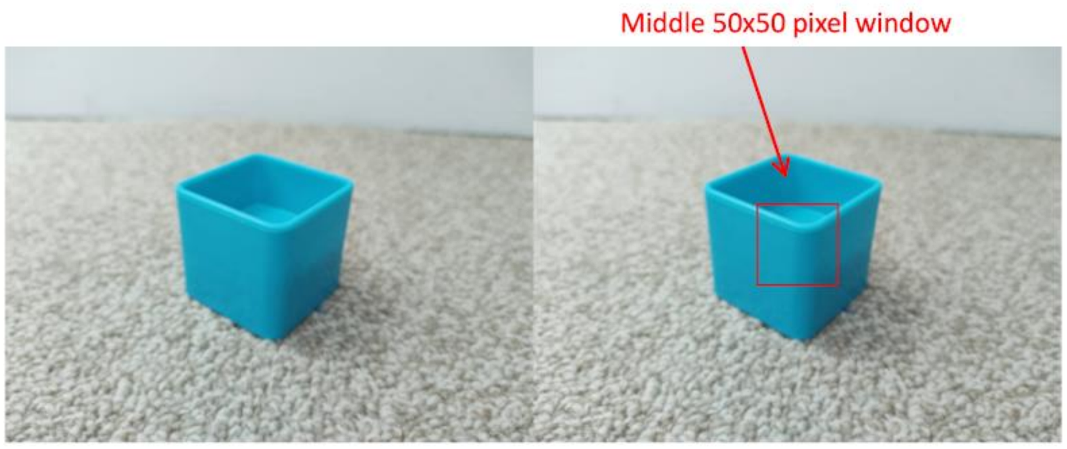
Displaying a calibration file (Mode 's'):

When printing a calibration file to screen using the 's' mode option described above, the program should print the contents to the screen in the following format:

```
Calibrated Objects:

blueblock: Hue: 191 (Max. Diff: 4), Min. SV: 50 30

coin: Hue: 132 (Max. Diff: 7), Min. SV: 50 30

orangeblock: Hue: 10 (Max. Diff: 10), Min. SV: 50 30
```

using for example the specific objects in the colour calibration file shown in the sub-section above.

Calibrating a Colour Profile (Mode 'c'):



During calibration, your program should load the provided image and read the pixel values in the central 50x50 pixel window of the image (all images provided by test cases will have an even number of pixels in the horizontal and vertical direction for simplicity). Your program should only consider pixels in this window that have a Saturation of greater or equal to 50 and a Value of greater or equal to 30. You should record the maximum and minimum Hue values, and determine their mid-point (using the function hue_midpoint() provided in the scaffold code). You should also determine the distance/difference between the minimum/maximum Hues (using the function hue_difference() in the provided scaffold code).

You code should then print out a line of text corresponding to the calibration parameters of a single object entry (with name string provided in the command line arguments) in the same format as in each line of the calibration file described above, i.e.:

<object_name> <Middle Hue> <Max. Hue Difference> <Minimum Saturation> <Minimum Value>

The <Middle Hue> should be the computed Hue mid-point and the <Max. Hue Difference> should be half of the Hue distance/difference, as calculated above. The minimum Saturation and Value should always be printed out at their default values of 50 and 30 respectively.

Note: When using the program, you can choose to "pipe" this output when you run your program and append it to the end of a calibration file (i.e. see Week 8 Monday lecture), for example:

```
./cam_detect c redapple redapple.bmp >> calibration.txt
```

This appends the line of calibration data provided to your program to the end of a calibration file, which can be used for object detection in future images.

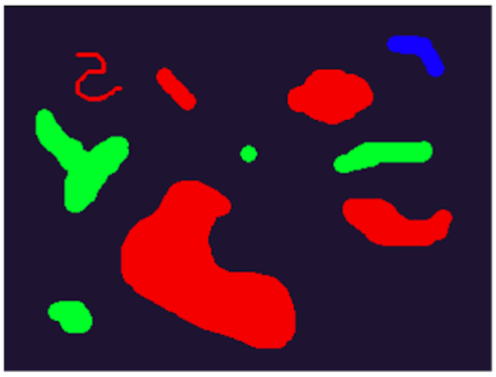
Bounding Box Coordinates

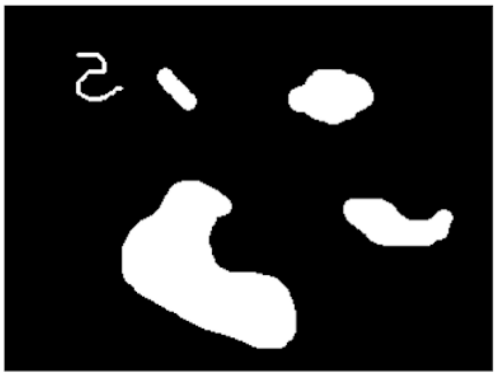
A detected "bounding box" for an object in an image is a set of four coordinates:

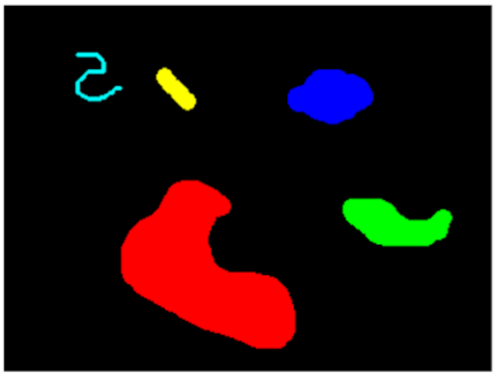
- **x:** pixel index (horizontally, column) of the bottom-left corner of the box
- **y:** pixel index (vertically, row) of the bottom-left corner of the box
- **width:** total width (horizontal direction) of the box in pixels
- **height:** total height (vertical direction) of the box in pixels

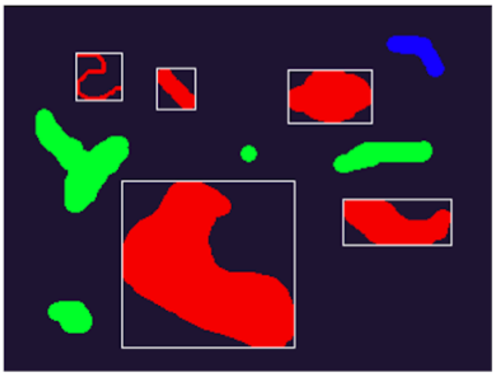
A bounding box fits tightly to the maximum and minimum extents of pixels making up an object: e.g. if the left-most pixel index for an object is 100, then the bounding box (x) coordinate will also be 100. If the width of this box is 10, then correspondingly the right-most pixel of the object would have an index of 109.

Object Detection (Mode 'd'):


(1) Original Image


(2) Thresholded Image

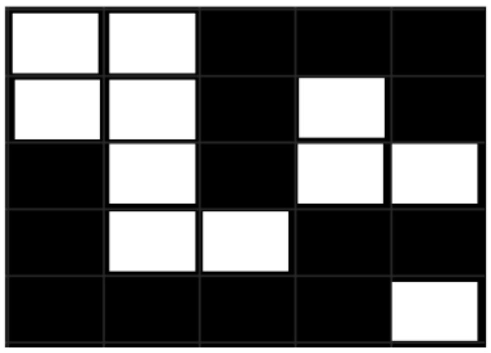

(3) Distinct Image Regions

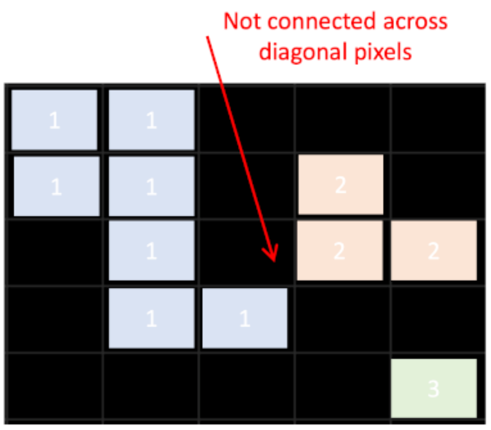

(4) Original Image with corresponding bounding boxes of detected regions

During object detection, your program should load the specified image and calibration file, and for each entry of the calibration file, scan the image for objects following the specified colour profile. Scanning consists of the following steps:

Image thresholding: Calculate the (HSV) of each (RGB) image pixel, and record which pixels meet the (HSV) thresholds based on the calibration values. For Hue, calculate the Hue difference/distance from the target pixel to the <Middle Hue> in the calibration (using hue_difference()) and if this distance is less than or equal to the <Max. Hue Difference>, then the pixel is a potential object pixel. Consider maintaining a separate Bitmap image object with pixel values (0,0,0) for non-object pixels, and (255,255,255) for object pixels that meet the above thresholding process.

Finding Connected Image Regions: From thresholded pixels, your program should then find regions of inter-connected pixels that make up separate distinct objects. Pixels are considered to be a part of a single region if they are "connected to" any other pixel in that region (i.e. they are directly adjacent to, in the vertical or horizontal direction, but not diagonally, see figure below). To do this, consider building an image object where each pixel maintains its corresponding region number (i.e. numbers from 1 to N) and for which pixels that are not part of a region are labelled zero. For each thresholded pixel, find all connected pixels via a depth-first or breadth-first search across the thresholded pixels. Repeat this process until all thresholded pixels have been assigned into a region.


Thresholded Pixels



Coloured by region

Bounding Boxes: For each of the regions identified, the bounding box coordinates are calculated by finding the pixel coordinates that have the maximum and minimum extents in the horizontal and vertical directions. In order to remove outliers and noise, bounding boxes must have a minimum width of 20 pixels and a minimum height of 20 pixels (smaller boxes should be discarded). Once the bounding box coordinates have been calculated they should be printed to the screen in the format:

Detected <object_name>: <x> <y> <width> <height>

For example, the following output would indicate detecting two redblob objects and one greenblob object:

Detected redblob: 77 15 114 110

Detected redblob: 47 177 31 32

Detected greenblob: 29 25 29 21

When printing out bounding boxes, the boxes should be printed in the order that the different objects appear in the calibration file. For multiple boxes detected for the same object, boxes should be ordered based on the way their first pixel would be scanned in the image, where scanning starts in the bottom left hand corner of the image and scans the bottom row first, left to right, followed by the second row from the bottom and so on to the top row.

For example, if scanning through the pixels as described, the first pixel that we arrive at that is part of an object region, the program should list the corresponding bounding box for this region first. Continuing to scan from this point, when we hit a pixel for a new region, the corresponding bounding box would be displayed second, and so on (see also example for "blobs.bmp" below).

Pre-provided Code in the Scaffold

To assist, we have provided a number of useful functions in the scaffold code of the code challenge (see "Bitmap.c" and "Bitmap.h"): read_bmp() can be used to read in a BMP file, and write_bmp() to save out a file. Use copy_bmp() to make copies of the bitmap data in your program.

The functions rgb2hsv(), hue_difference() and hue_midpoint() can be used to convert pixels to HSV, calculate threshold data and test whether pixels meet thresholds when scanning for objects. The function draw_box() can be used to draw a bounding box on an image. For example, to save out a visualisation of the detected bounding boxes for visualisation and testing purposes you could use copy_bmp() to copy the original image data, draw_box() (in a loop) to draw detected coordinates and write_bmp() to save out this image for viewing.

Examples

A number of examples images and calibration files are provided in the code scaffold.

Input:

```
./cam_detect c orangeblock images/orangeblock.bmp
```

Output

```
orangeblock 10 10 50 30
```

Input:

```
./cam_detect c orangeblock images/orangeblock.bmp > calibration.txt
./cam_detect d calibration.txt images/combined002.bmp
```

Output:

```
Detected orangeblock: 129 116 67 69
```

Input:

```
./cam_detect d test_calibrations/blobs.txt images/blobs001.bmp
```

Output:

```
Detected redblob: 77 15 114 110
Detected redblob: 222 82 72 31
Detected redblob: 186 162 56 36
Detected redblob: 100 171 26 28
Detected redblob: 47 177 31 32
Detected greenblob: 29 25 29 21

Detected greenblob: 21 104 61 68

Detected greenblob: 216 130 65 21

Detected blueblob: 251 193 38 27
```