The current algorithm uses two detection stages as in a simple network in order to distinguish the two bracelets of same color. The first stage is a mask to identify whether a pixel is that of a bracelet or not (as currently used in our single player mode). The mask output is a Boolean map of the image with 1 for bracelet pixels and 0 for the others. The second stage identifies independent groups of pixels in the Boolean map. Scan the map, and if a pixel is found to be 1, assign a color, mark as zero, and search all neighbor pixels using the same color until none are found. Carry on in the map until reaching the end of it.

The second stage was implemented over three phases:

1. A recursive implementation: check the current pixel and then check the neighbors by calling the same function with new coordinates (full code on GitHub), based on a depth-first-search
   The problem with this simple, artistic implementation is that it is impossible to synthesis because of the recursive calls (how would you translate recursive calls and unknown, maybe infinite stack storage for previous function calls on hardware)

2. It was clear that a recursive implementation cannot be synthesized. The next step was to implement this function in the main in a while loop, to avoid stacking the variables of long recursive calls. The implementation became breath-first-search using a queue (FIFO protocol: first pixel in, first pixel out) but a similar method could work with a stack DFS implementation. When a pixel is found, it is pushed to the queue, which enters a while-loop. The loop goes on, adding neighbor pixels to the queue and checking them, until the queue is empty (no more valid neighbors to check) (full code on GitHub).
   Another problem showed up: the queue structure couldn't be synthesized. A vector was therefore used to implement the queue. The synthesis would complete without any solution. The explanation is because a vector (and therefore a queue) is a variable-length memory structure: the number of elements cannot be predicted for hardware allocation, and after adding or removing an element, a whole remapping takes place to readjust the indexing of elements

3. The solution was therefore to implement a queue from a static array with a known size. The idea is to use to index-followers: one for the index to read at, and one for the index to write at. These indexes "loop" around the array when they reach the end. The following is an illustration for the following actions on a queue ("read" means *queue.front()* followed by *queue.pop()*):

   Queue actions: START, push(3), push(4), push (2), push(6), read(), read(), push(1), read(), push(9), read(), read(), read(). Array illustration (size 4). Reader index (queue head) in green, writer index (queue tail) in red (yellow: both are on same cell):

Queue Actions		Index 0	Index 1	Index 2	Index 3		Queue head	Queue tail		Output
START		?	?	?	?		0	0		NA
Push(3)		3	?	?	?		0	1		NA
Push(4)		3	4	?	?		0	2		NA
Push(2)		3	4	2	?		0	3		NA
Push(6)		3	4	2	6		0	0		NA
Read()		3	4	2	6		1	0		3
Read()		3	4	2	6		2	0		4
Push(1)		1	4	2	6		2	1		NA
Read()		1	4	2	6		3	1		2
Push(9)		1	9	2	6		3	2		NA
Read()		1	9	2	6		0	2		6
Read()		1	9	2	6		1	2		1
Read()		1	9	2	6		2	2		9

The final question is: how do we track that we the process is done? (in part 2 we checked when the queue was empty). Well, notice the following example for the same process but for an array of 5 entries:

Queue Actions		Index 0	Index 1	Index 2	Index 3	Index 4		i_read	i_write		Output
START		?	?	?	?	?		0	0		NA
Push(3)		3	?	?	?	?		0	1		NA
Push(4)		3	4	?	?	?		0	2		NA
Push(2)		3	4	2	?	?		0	3		NA
Push(6)		3	4	2	6	?		0	4		NA
Read()		3	4	2	6	?		1	4		3
Read()		3	4	2	6	?		2	4		4
Push(1)		3	4	2	6	1		2	0		NA
Read()		3	4	2	6	1		3	0		2
Push(9)		9	4	2	6	1		3	1		NA
Read()		9	4	2	6	1		4	1		6
Read()		9	4	2	6	1		0	1		1
Read()		9	4	2	6	1		1	1		9

Notice how the two indexes meet only when the queue is empty if the array is big enough to avoid overlaying the indexes during the process. The queue can now be implemented using a static array! (Full code on GitHub)

The size of the array is chosen by estimating the size of the detected objects and therefore the maximum number of pixels that will be held in the queue awaiting check. A quick mathematical demonstration shows that for an A (blue) by B (green) canvas to be scanned, the following conclusion applies.

Let $N = \frac{A+B}{2}$, and suppose the non-valid neighbors are not added for checking. The number of maximum elements in the queue will be $S_{odd} = 1 + 3 + 5 + 7 + \cdots + (2N-1)$. To find the sum, let $S_{2N} = 1 + 2 + 3 + \cdots + 2N = (2 + 4 + \cdots + 2N) + \big(1 + 3 + .. + (2N-1)\big) = 2(1 + 2 + 3 + \cdots + N) + S_{odd} \Rightarrow S_{2N} = 2S_N + S_{odd} \Rightarrow S_{odd} = S_{2N} - 2S_N = \frac{2N(2N+1)}{2} - 2\frac{N(N+1)}{2} = N^2$

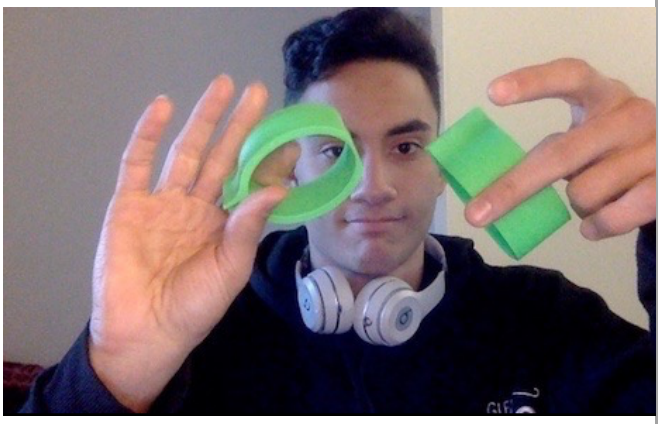
On our screen, the dimensions are 360x640 pixels for the canvas. An object may cover at most $\frac{1}{16}$ of the canvas, or 14 400 pixels. Squaring this value gives an array size of 207,360,000. Quite big… can we do better? Yes. Supposed at some point the array was small enough to allow overwriting pixels that were to be scanned. These will have neighbors that will add them back to the array! After trial and error, the size of the array could be reduced down to 115,200 and still offer great results

4. Further work:

Two other stages are to be added in order to perform a smooth multiplayer mode:

- A stage that checks every detected object's number of pixels: if it too small, it corresponds to noise or misdetection, and is deleted (made black)
- A stage that tracks each object movements vectors, predicting the object's next position environment (in a span of pixels). This will allow removing objects color switching (as when one is detected before the other and then after it)

Stage 0: Image Input	Stage 1: Color detection



Stage 2: Object Distinction

Stage 3: Delete noise (to be done)