Body Break

A real-time image processing game designed for the PYNQ Board using FPGA High-Level Synthesis design

Nathan Amar, na6518@ic.ac.uk

Raphaël Bijaoui, rdb4017@ic.ac.uk

Daryl Lim, dyl17@ic.ac.uk

May 2019

Department of Electrical and Electronic Engineering

Imperial College London

Dr. C. Bouganis, Ms. E. Perea

Contents

1- Abstract

This report investigates the application of an interactive visual game applied to a Field Programmable Gate Array (FPGA). The report details the development process of the game starting off as an abstract concept and turning into q fully functioning image processing system. Project management was carefully executed through delegation of tasks, specialization and workflow protocols. Potential solutions for the implementation of marker detection, visual feedback generation, and PYNQ board interfacing were thoroughly considered and explored with the system specifications and user experience in mind. HSV color detection was implemented in Vivado HLS and marker-centered ring filters were visualized in OpenCV in the Jupyter Notebook environment. The fully functioning game is demonstrated and its hardware optimizations are analysed, noting fundamental limitations.

2- Introduction

Body Break is a game that draws inspiration from blindfolded, hide-and-seek and code-breaking type games throughout history. The core design philosophy of the project was to develop an interactive game that encourages motion through rotation, contraction and extension of the joints and limbs, with the body transitioning between static and dynamic states as the player navigates towards the objective. Over the course of its development, Body Break has adhered to its underlying notion that as the user gets closer to the objective, the system informs the player that it is getting "warmer", and "colder" otherwise. The game objective and gameplay has not changed from the initial design, which is for the players use their bodies to crack a code in the form of five different points on the screen, i.e. to guess a solution pose.

The final product has fulfilled the objectives laid out previously, that were to achieve working color detection and real-time image processing. The current iteration of the system is able to detect and identify different colors that correspond to unique body parts, which for Body Break correspond to the head, wrists, and ankles around which a colored band (*player marker*) would be worn, as such achieving a functional positional tracking system for the player's body. The system also can perform real-time image processing based on the positional values of the markers interpreted on an x-y coordinate system. These coordinates are translated to the heatmap, which is real-time visual feedback in the forms of a full screen filter and marker-centered ring filter.

3- Progress made since the management report

2.1-Updated high level description of the system (C code)

The current iteration of the system functions identically to the system described in the management report apart from the implementation of the functions. Every single change was carefully considered as a team with cost-benefit analysis done in regard to system performance and timing constraints.

Design changes

The largest change was shifting from an object-oriented to a procedural programming paradigm in which the team was more confident and competent. The team did not want to use a programming paradigm that would require additional time to learn and familiarize at the risk of suboptimal resource utilization and running short on time. The concept of having classes and methods that were subsidiaries of larger components in the system was appealing as an idea, however it would have required additional logic blocks to be configured and integrated into the pre-existing overlay, that which the team decided should be minimized to prevent any hold ups and delays in rolling out the final product. As a result, the player, tracker and game classes were removed. The corresponding attributes for player and tracker were managed in real-time during the image stream processing, and the game was full implemented inside the Jupyter notebook.

The filter gradient was originally intended to span from blue to red so that stronger blue meant the player was straying from the solution and stronger red meant that the player was approaching the solution. The team felt that simply having the gradient change from normal image colors to red was more intuitive for the player and reduced the workload on the gradient mapping. The medium difficulty of the game was also removed, leaving hard mode with no changes made and easy mode now displaying a marker-centered ring filter around each player marker in real-time. Individual rings would change their intensity proportionally to the distance from their corresponding solution coordinate (hit zone) and also designed have a fading effect that blends the red gradient seamlessly into the background. The concept of a proximity index was also scrapped and replaced by the raw values returned from distance calculations. Additionally, the team decided that certain player markers had less contribution to the pose than the few key ones that were the head, wrists, and ankles, thus the rest were removed from the game.

Functionality changes

The modifications to the functions were done largely for the sake of optimization. The team decided that the programmable logic (PL) should only contain functions related to image processing and filter display, and anything else should be placed in the Jupyter notebook. This included difficulty selection and the list of solution poses, both of which would have been passed to the function parameters by reading or writing to the IO at the s_axilite ports. The parameter would have been read at evey loop and the state of the game modified accordingly. The latest design now has the PL return the real-time coordinates of each marker to the ports which can then be accessed and processed by Jupyter. The solution coordinates were also not passed into the PL and kept solely in the Jupyter notebook. This allowed the system to receive variable input from the Juypter notebook interface, which saved a lot of time for testing.

Certain functionality was also removed from the game with justification. The team felt that having two ways to change the difficulty of the game was redundant, therefore functionality for the user to adjust the tolerance factor was removed and the tolerance was implemented directly in the code. User body calibration was an attractive feature to include, though the team felt that it was unnecessary as it did not contribute much to the image processing aspect of the project. The player could simply compensate by moving closer or further away from the camera, which conveniently added another dimension of gameplay as a side effect and was incorporated into the game design. The removal of player calibration and the extraneous player markers left the auxiliary user measurements such as upper and lower limits, and wingspan of the user unaccounted for, thereby getting removed from the game. Function design wise, all subsidiary functions were not declared as a separate function and were placed sequentially within the scope of the top function in order to minimize the number of additional logic blocks that would have to be integrated into the PL.

2.2 Schematic

2.2.1 Flowchart

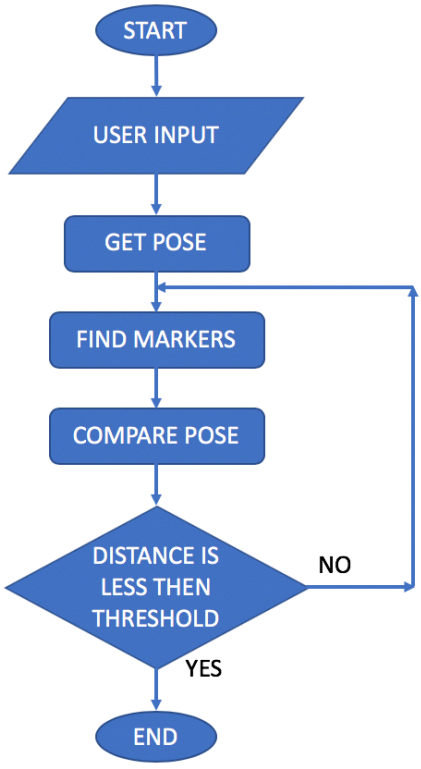


Figure 1. Updated flowchart.

2.2.2 Main Data Flow Diagram

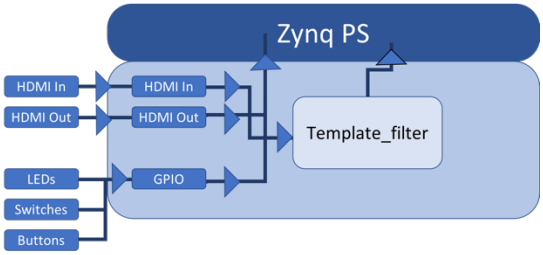


Figure 2. Main Dataflow Diagram.

2.3 Project Planning and management

2.3.1 Team Specializations

The team's strategy towards the project has remained the same over the course of the games development. In the early stages of development, the team decided to reshuffle the specializations in order to better suit its members. It was decided that for primary roles, two members would specialize at the software implementation and the last member at hardware implementation and optimization. For secondary roles, one member from the software side will collaborate with the member from hardware side to work on optimization and system performance while the other member from software side focuses on interfacing the Jupyter notebook with the PL. This delegation of roles and tasks is summarized in the table in figure 3:

	Daryl	Nathan	Raphael
Primary	Image processing and visual feedback generation	Image processing and color detection (RGB and HSV)	Optimization - pipelining, loop unrolling, custom precision
Secondary	Jupyter notebook interfacing and GPIO	Optimization, testing and implementation	Bitstream, TCL generation and testing

Figure 3. Mind map generated during ring filter ideation.

Internal deadlines and milestones were set in order to break down the project into smaller portions. This helped the team keep track of the overall progress of the project and better visualize team performance. Due to changes in the task delegation and shift in responsibilities, the project timeline had to be changed. These changes are reflected in the updated Gantt chart in figure 4.

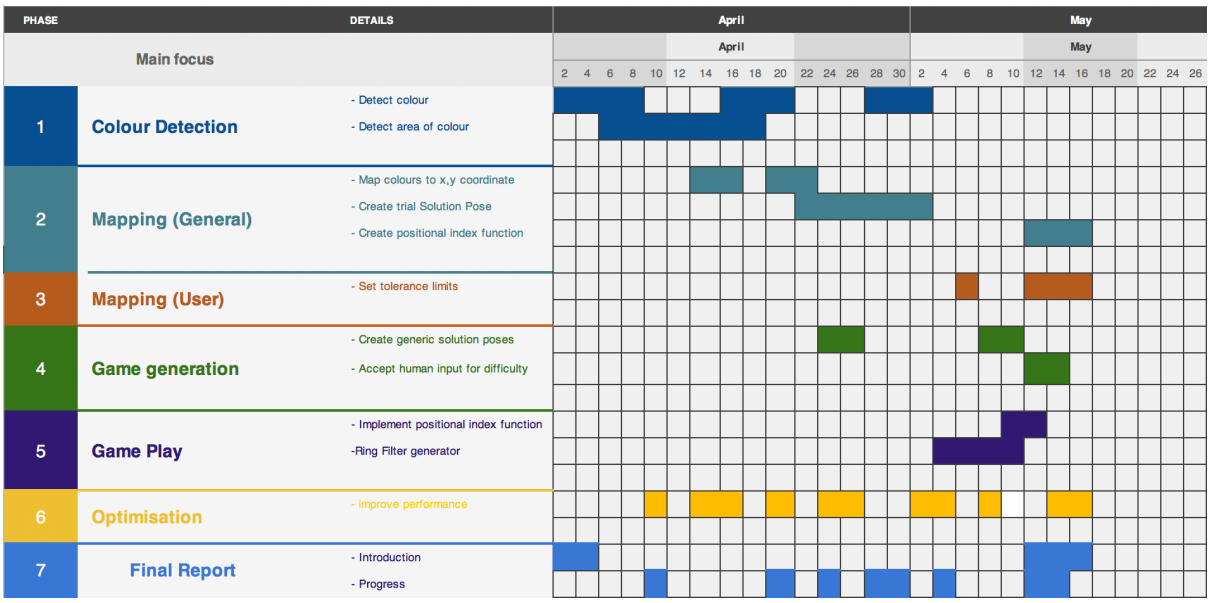


Figure 4. Updated Gantt chart.

2.3.2 Ideation Process

The system design was chosen through collaborative concept generation and feedback sharing. The team organized a meeting at the very beginning of the project solely to discuss and align our team's objectives and to brainstorm potential directions. All ideas were openly exchanged across the board, allowing team members to evaluate and return feedback regarding the feasibility of the proposed idea. To better visualise these abstract ideas, the team made full use of mind maps to both facilitate the brainstorming process and retain it for future reference.

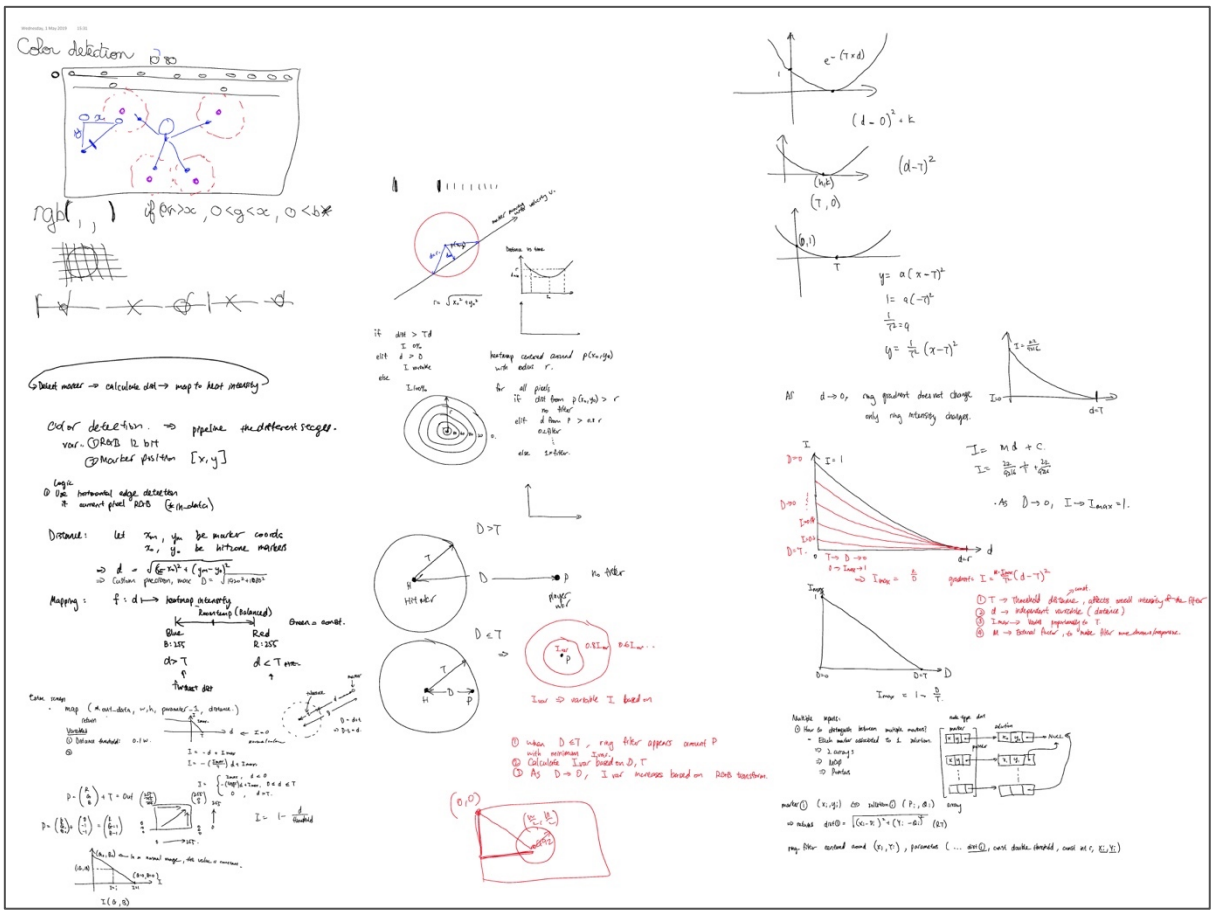


Figure 5. Mind map generated during ring filter ideation

Throughout the process, the key points of consideration revolved around the feasibility and difficulty of the concept in regard to the team's average technical skills level. In considering this, the team briefly evaluated how they would expect to approach its development and made rough estimates on how the workload would vary with time. The ideas that were evaluated as unfeasible were rejected whereas the few that were approved were shortlisted and further evaluation and consideration was done in order to select the actual design.

2.3.3 Implementation, Testing and troubleshooting

The protocol for the workflow was structured in a way that allowed the team to work in parallel. Feature implementation was done by having the team discuss required specifications of the feature, then having the member in charge of that feature design and implement it completely under their control. Concurrently, other team members would be working on their specializations. Afterwards, the whole team would meet to review and amend the feature to be integrated into the rest of the system. For example, if there were two parts of the system that required interaction with each other, the team would discuss how the interaction takes place, e.g. an independent component returns a value that a dependent component requires as input, then part ways to design each component with their interactions in mind.

This enabled both components to be tested simultaneously and reduced bottlenecking of the workflow. If there were setbacks in completing the independent component, the dependent component would have to be tested using dummy inputs until the issues were overcome.

If there were urgent issues that had to be addressed, the team members would pause their own tasks to help troubleshoot the issue. If someone felt that a certain feature should undergo a major overhaul or some feature should be removed from or added to the system, a meeting would be setup for that purpose and the opinions of each team member would be taken into deep consideration before a decision was made unanimously. Otherwise in the case of small implementation changes that did not affect the overall project outcome, the team members agreed to trust each other's judgements to ensuring that that feature still works as intended.

For the testing of the image processing algorithm, the team assessed multiple aspects using different approaches. Comparisons of images with markers and a realistic background and single colored images and images with cropped out sections were used to test the functionality of the color detection and their calculated averages output on the image. The robustness of the color detection was also verified using images with the markers in different lighting conditions so that the color detection ranges could be adjusted. The test bench (see Appendix C) in the C simulation facilitated the testing and was often modified to further test specific functionality, such as ensuring that multiple iterations of the filter could be superimposed onto the same output image. C synthesis was also carried out to measure system performance by removing the variable latency caused by variable image and width inputs and setting those parameters to constants. This ensured that latency and throughput would not appear as undefined values in the synthesis report.

4- Design Process

3.1 Specifications

Throughout the implementation, testing, and debugging process in the creation of *Body Break*, the team constantly held the confines of the project in mind. These specifications ensured all efforts remained consistent with one another and were established based on the following 4 core axioms:

For Body Break to be successful, it must:

1. *Relay a real-time image of the user*
2. *Provide warm feedback of user position to solution pose*
3. *Display the user image clearly, such that all markers are clearly visible and users can recognize themselves.*
4. *Have an overall hardware utilization suitable for the PYNQ board.*

Figure 6. The 4 Axioms of Body Break.

Using these broad axioms, the team proceeded to narrow them down into key metrics for accurate assessment of the game:

Axiom 1: Real-time image

The team defined a real-time image to be within the scope of human vision, which processes regular motion between 25 to 60 Frames Per Second (FPS) (Sudhakaran, S.). However, when considering regular human-screen interactions, this range is far narrower, moving from 24 FPS for movies to about 30 FPS for most single-player games (Stewart, S.). Below 12 FPS, the brain can start detecting individual images, making a game quickly lose a sense of continuous motion (Read, P. and Meyer, M.). With these considerations in mind, it is necessary for us to achieve gameplay within the bounds of 12 - 60 FPS, with the main target band to be within 25-30 FPS. This is so as the team wanted the user experience to feel like a natural extension to any other human-computer interaction.

Axiom 2: warm feedback

The fundamental idea is that as the user gets closer to the solution pose, there is an on-screen indication of this improvement- and vice-versa. For the user to intuitively understand whether a movement was advantageous or not, this feedback needs to therefore be both real-time and prominent on the screen- such that there is no ambiguity towards the change in position. Without these two factors in place, the gameplay would be significantly hampered, as the user loses a crucial sense of coordination.

Axiom 3: clear user image

The team defined clarity of image within the scope of this game according to two key factors. Firstly, users must be able to recognise their own images - such that if there were two similarly dressed and shaped users in the screen, they could clearly differentiate their respective image from their facial features. Secondly, the resolution must be strong enough to catch the distinct pixels of the marker from all confines in a room. After a preliminary round of testing, it was realised that a resolution of 480p (640x480 pixels) was more than sufficient for these two criteria. Nonetheless, to give the user a sleeker

experience, the group aspired for no less than 720p HD Ready (1280x720), as there was a visible difference between the images when images were tested on the target monitors (23" HQ Compaq LA2306x). However, as the resolution was increased beyond this, it did not necessarily translate to a clear improvement in the aforementioned visual factors. With the trade-off of Frames Per Second against visibility in mind, the team unanimously agreed that 720p was the ideal resolution for *Body Break*.

Axiom 4: hardware utilization suitable for the PYNQ board.
Body Break has its own set of physical limitations. Most notably, the Programmable Logic consists of 630KB of BRAM, 220 DSP Slices, 106,400 Flip-Flops, and 53,200 6-input LUTs. However, not all these resources can be used by our image processing IP. This is because our IP must be inserted into the PYNQ Base Overlay, which itself utilises a range of hardware resources as it provides functionality to other useful modules such as GPIO and HDMI in/out.

To better understand how many resources the team was available to work with in the custom IP, the utilisation of the Base Overlay with a generic template filter was inspected. The results were as follows:

Resource	Utilization	Available	Utilization %
LUT	36263	53200	68.16
LUTRAM	2292	17400	13.17
FF	52265	106400	49.12
BRAM	64	140	45.71
DSP	31	220	14.09
IO	118	125	94.40
BUFG	5	32	15.63
MMCM	2	4	50.00

Figure 7. Utilisation of Base Overlay prior to implementation of custom IP.

This analysis shows that approximately 32% LUT, 51% FF, 55% BRAM, and 86% DSPs can be worked with using the team's custom IP. In reality we expect these figures to differ as the custom IP (template filter) will be upgraded, which in effect removes the utilisation of the generic template filter with that of the team's custom IP. Nonetheless, it provides a good reference point as the utilisation of the custom IP is first considered in Vivado HLS.

Now, taking all these specifications as a framework, the team revisited the ideation process – seeking to match ambition with reality.

3.2 Marker Detection

3.2.1 RGB Color Detection

For Body Break to run effectively, robust marker detection is the core process to which the rest of the game derives from. In order to get a fundamental prototype up and running as soon as possible, the team first proposed a naïve approach to colour detection – RGB.

RGB Color Detection

The fundamental approach to RGB colour detection lies in reading characteristic pixels and using this information to define a range of colour values for the markers. The team tested the validity of this naïve solution by taking a number of photos of the markers in different places, and studying their RGB values. During this approach, we realized that a large flaw in RGB was that due to shadows and light conditions, the RGB range could vary so greatly that many colours could be confused with each other at the same time.

As a result, in order to get a basic, fundamentally working prototype, the team took these photos and honed into the parts that were most characteristic to the colour.

Using these characteristic images, the team then used an RGB color detector program, and selected the pixels that best approximated to the color seen by the naked eye when looking at the marker. The table below is a snippet of the data obtained for the pink marker.

PINK			
R		**G**	**B**
247		68	168
253		66	171
255		104	188
238		57	154
246		51	155
249		51	164
246		70	174
241		44	149
243		68	163
238		57	153
249		90	174
212		61	130
180		36	110
201		68	125
MAX	**255**	**104**	**188**
MIN	**180**	**36**	**110**

Figure 8. Table of characteristic pink coordinates, used to find color range of marker.

These values were then used to create if conditions on each pixel being sifting through in the project. A snippet of this condition is attached here as follows.

```
//// PINK
if ( r  < 255 && r > 185 && g  < 104 && g > 35 && b  < 188 && b > 109 ){
     out_b = 0xFF;
     out_g = 0xFF;
     out_r = 0xFF;
}
```

Figure 9. If condition for naïve pink detection.

To ensure the condition correctly identified the bands in different locations, the code was tested against different images, with adjustments to the RGB range until a reasonable amount of pixels was detected for each image.

Figure 10. RGB pink detection in different locations.

This solution was implemented fully for the 5 markers – yellow, green, blue, orange, and pink. However, with more testing, it became clear that there were limitations to this solution. Notably- under low light conditions, the reliability of the solution greatly reduced. This is because RGB does not account for different light conditions, only the combination of the three RGB components in a given setting. Hence, if comparing a low-light pink to bright pink, the range of values could be enormous, reducing the RGB colour detection to redundancy.

Moreover, given the game's reliance on an accurate mapping system for markers, the RGB detection performs poorly, as can be seen with the low percentage of pink pixels picked up in *Figure X*. For these reasons, a new detection system became crucial for *Body Break*, and new solutions were brought to the drawing board.

3.2.2 RGB Ratios

As observed, the RGB colour range is very sensitive to light. Another approach for this problem is to use ratios. Our team analysed the pixel data from the markers as shown in the table below:

	Pure RGB		Ratio RGB	
	Min	Max	Min	Max
	132	222	34	37
	130	235	36	39
	91	163	25	27

Figure 11. Table containing pure RGB and ratio RGB values.

Within the same marker, the red component's value can vary between 132 and 222 out of 255 depending on lighting. However, the ratio of the red component's value over the sum of all the component's value matches a narrower range in percentage values. This method was applied to the following yellow marker:

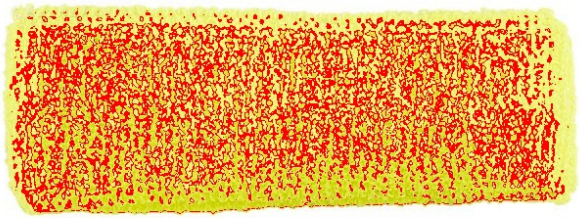


Figure 12. RGB ratio color detection on a yellow marker.

Most of the pixels are picked up by the detection but it is noticeable that on the same marker different shades can appear. Considering this picture is a marker in ideal condition this detection didn't solve the problem faced.

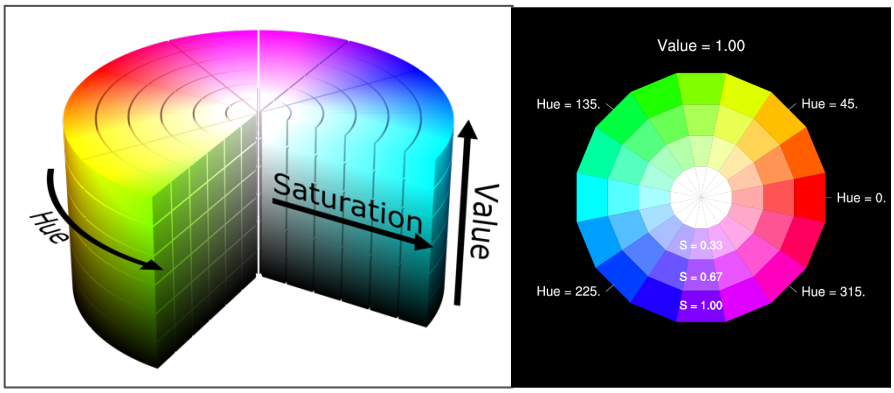
3.2.3 HSV

HSV (Hue Saturation Value) is an alternative to the RGB colour model. This range uses three parameters that are closely related to how human perceive and characterise colours. The three components of each colour represent cylindrical coordinates on the following images:

In the HSV cylinder model (figure 13), the H coordinate can be read on the circle and has values between 0 and 360°. The S coordinate is read on the circle radius and has values between 0 and 1. The V coordinate is read on the vertical coordinate and is between 0 and 1. If the V value is fixed, the picture above shows a slice of the cylinder cut into small sections. Hence, each region of the circle can be associated with a colour. For *Body Break*'s colour detection, the markers' H values are each located within a small region of the circle which is very useful for different light condition. For instance, the pictures below sho

Figure 13. HSV cylinder model and circle with V value fixed.

H value stays at 0. Which means that every red pixel- regardless of the light condition- can be picked by the condition "H=0".



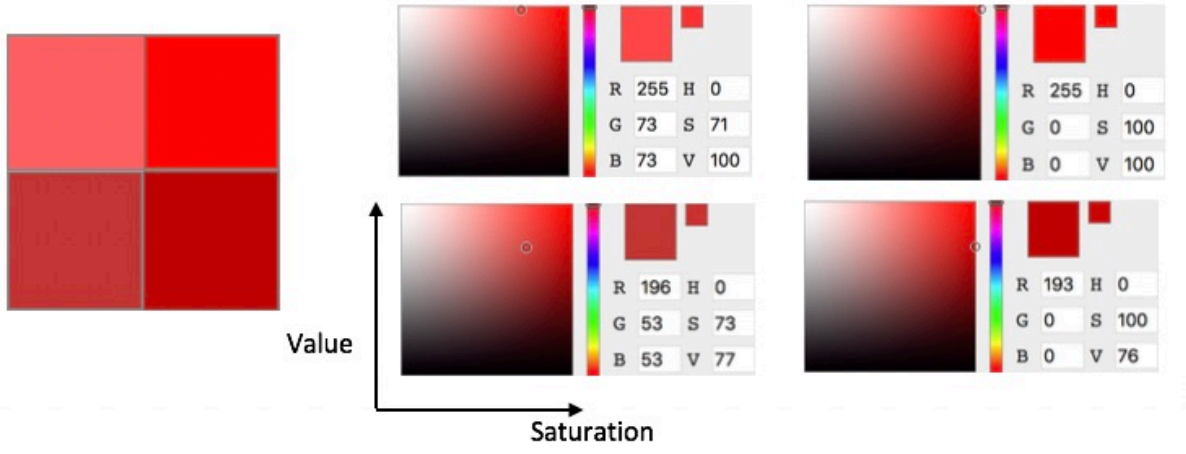


Figure 14. Demonstration of red maintaining same H value for different RGB values.

However, if we look at the RGB pickers above, for H equal zero, the colours can range from white to black with all the shades of red. If H was the only condition, *Body Break* would categorize most of the pixel as false positive. This is the reason the condition on Saturation and Value are added to filter out all the false positives. For each marker, the team identified the correct range to get as many pixels as possible only on the marker. This resulted in the following regions for fixed values of the Hue parameter.

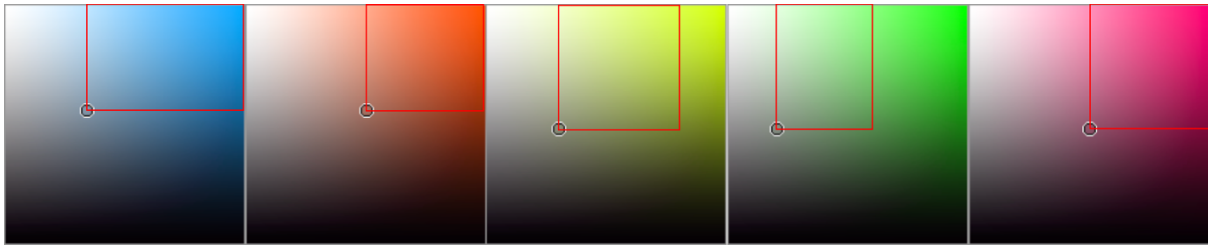


Figure 15. Selected regions for H indicated by the red rectangle.

However, if the Hue parameter is fixed, the detection is not very flexible. This is why the H parameter's condition is also a range. Using this method, the results for the marker detection were very accurate: most of the markers' pixels were picked up and almost no false positives were detected. This filtering is done using the following algorithm:

- Identify which components are the maximum and minimum in RGB

$$Value = Max$$

  - If maximum != minimum,
    - If R == maximum:

$$Hue = \frac{G-B}{Max-Min}$$

    - If G == maximum:

$$Hue = 60 \times \left(\frac{B-R}{Max-Min} + 2\right)$$

    - If B is the maximum:

$$Hue = 60 \times \left(\frac{R-G}{Max-Min} + 4\right)$$

  - If maximum != 0,

$$Saturation = \frac{Max-Min}{Max}$$

  - If maximum == minimum,

$$Hue = 0$$

$$Saturation = \frac{Max-Min}{Max}$$

The HSL range is very similar to that of HSV. Calculating its values however requires more processing because the formulae use more operation. For optimisation, our team used HSV which fitted perfectly the image processing required. Using this technique most of the pixel of the marker are detected no matter the different shades:

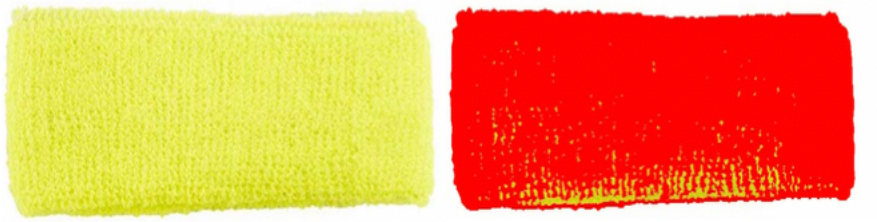


Figure 16. HSV color detection demonstration on yellow marker.

3.3 Coordinate Calculation

Following the marker detection, *Body Break* requires accurate coordinates for each marker. This is done by calculating an average position of the marker with all pixels detected. For instance, for the blue marker, the following code would be executed:

```
if (h_val < 16 && h_val > 4 && s_val > 0.5 && s_val < 1 && v_val > 150 && v_val < 255){
    out_r = 0x00;
    out_g = 0xFF;
    out_b = 0x00;
    orangexcurrent += j;
    orangeycurrent += i;
    orangecounter++;
}
```

Figure 17. Conditions checking for HSV ranges of orange and frequency calculation.

This code changes the colour of the pixel to display visual feedback. This allows the user to see how his marker is being processed or if any other pixel in the background is interfering with the game. The coordinate calculation is explained by the following pseudo code, the condition is simplified to blue on the principles explained earlier:

```
If pixel is blue :
        Blue x coordinate += pixel's x coordinate
        Blue y coordinate += pixel's y coordinate
        Total Blue pixels += 1
```

Figure 18. Pseudocode to calculate frequency, sum of x and y coordinates.

All that is left to do is repeat the process for every marker. Once the program is done looping through the entire image, then the total x and y coordinate value of each marker is divided by their respective total frequency. This returns the average position of each marker represented in 2D coordinates. Using

this technique facilitates smooth tracking which is crucial for user immersion and a pleasant gameplay experience. The average positions are then output as arguments using pointers as shown below:

```
void template_filter(volatile uint32_t* in_data, volatile uint32_t* out_data, int w, int h, int* bluexpose, int* blueypose, int* orangexpose, int* orangeypose,
                                                 int* pinkxpose, int* pinkypose, int* greenxpose, int* greenypose, int* yellowxpose, int* yellowypose );


                                    *bluexpose = bluexcurrent/bluecounter;
                                    *blueypose = blueycurrent/bluecounter;

                                    *orangexpose = orangexcurrent/orangecounter;
                                    *orangeypose = orangeycurrent/orangecounter;
                                    *pinkxpose = pinkxcurrent/pinkcounter;
                                    *pinkypose = pinkycurrent/pinkcounter;

                                    *greenxpose = greenxcurrent/greencounter;
                                    *greenypose = greenycurrent/greencounter;

                                    *yellowxpose = yellowxcurrent/yellowcounter;
                                    *yellowypose = yellowycurrent/yellowcounter;
```

Figure 19. Assignment of coordinate values to be accessed in processing system.

The pointers and their corresponding MMIO addresses can then be accessed from the Jupyter Notebook interface. In the Python code, the memory addresses are first written to and then read from in order to obtain the coordinates of each marker and calculate their individual distances. This allows the program to allocate memory for the average values to be written. The full implementation can be seen in Appendix A.

3.4 Visual Feedback Generation

The following section details the various forms of visual feedback with which the team experimented in chronological order. The visual feedback of the full screen filter and the marker-centered ring filter was implemented by calculating the intensity (dependent variable y) for each pixel in the frame as a function of distance (independent variable x) upon the first loop through the image data stream, then on the second loop backwards the filter would be applied back onto each pixel. The function used to model intensity had several criteria: it must have an x-intercept at the threshold distance (T) beyond which the filter would not be displayed. Within the bounds [0, T] on which the gradient function is defined, the function must have a negative gradient such that the intensity equals zero at x = T and so that the y-intercept is 1 for the maximum intensity (I). An additional variable M was included as an arbitrary valued multiplier that is used to scale the peak of the intensity function to enhance the visibility of the

filter. The team initially modelled intensity linearly against distance, which for T=1.5 behaved according the function in figure 20.

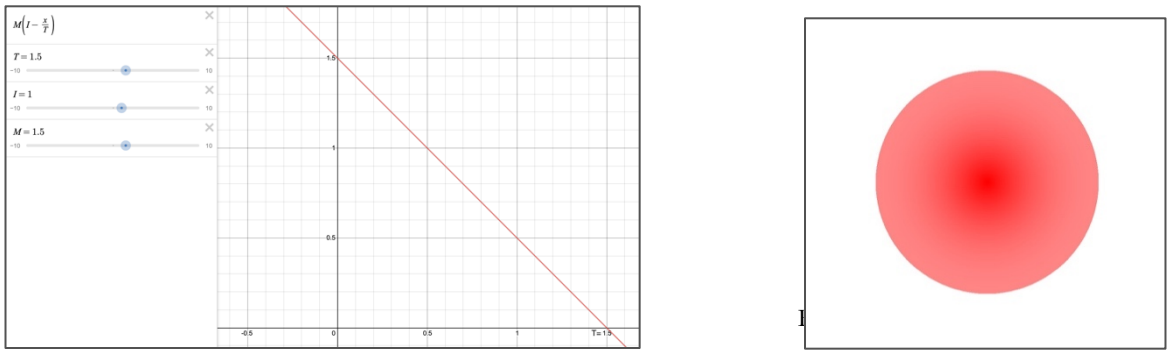


Figure 20. Linear function of distance against intensity.

The issue with the linear model was that at the x=T there was a sudden change in intensity which interrupted the gameplay immersion. This behavior was very obvious in the marker-centered ring filter (figure 21). A better solution was a piecewise quadratic function defined on [0,T] (figure 22) that returned the maximum intensity for x < 0 and 0 for all x > T.

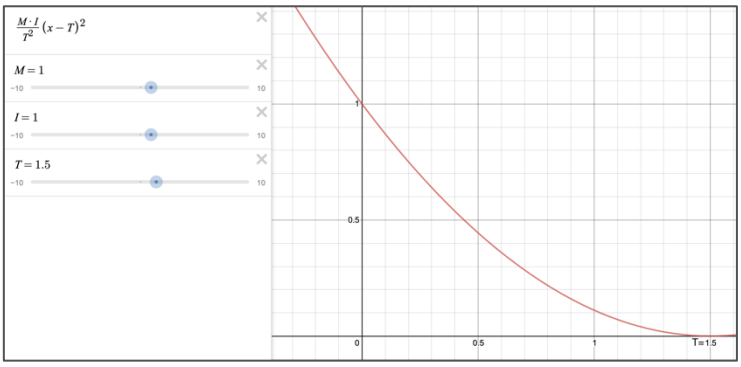


Figure 22. Quadratic function of distance against intensity.

For the full screen filter, the function receives the aggregate distance of all the markers and generates a single output that spans the entire image as such:

Figure 23. Original full filter applied on test image.

In figure 23 above, the sections of the image from left to right demonstrate the filter at zero intensity, medium intensity and maximum intensity, but in reality the filter alters the entire image. The redness of the image was achieved by scaling the green and blue values of the pixels inversely to the intensity function so that the strongest red would be observed at the smallest distance from the solution pose. For the marker-centered ring filter, the function receives the individual distances of each marker from its corresponding solution coordinate and generates an intensity gradient on screen around the calculated position of each marker as seen in figure 24.

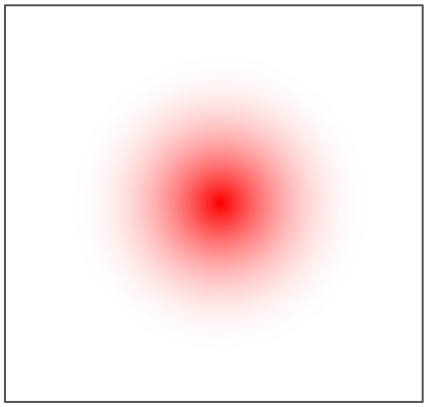


Figure 24. Second-generation ring filter.

The ring filter reuses the gradient function for its dynamic internal gradient effect by calculating the distance of each pixel away from the center pixel. It then adjusts the pixels green and blue values according to the gradient function with maximum intensity determined by the distance from the solution coordinate. For example, setting I=0.5 i.e. the marker is further away from the solution coordinate, would result in an intensity gradient as seen in figure 25 below:

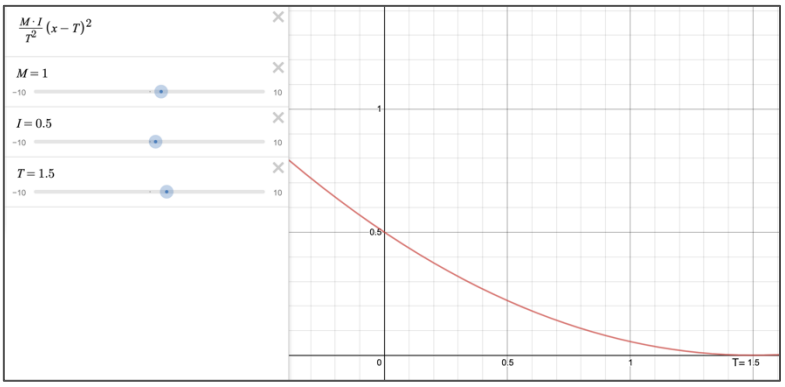


Figure 25. Quadratic gradient function with lowered peak.

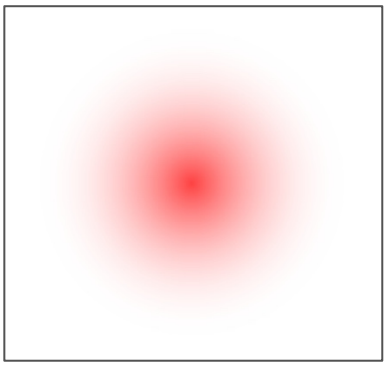


Figure 26. Second-generation ring filter with lowered peak.

This function now has a lowered peak intensity which is shown in the generated ring in figure 26, with a smaller "nucleus" compared to the previously generated ring in figure 24. The ring filter would be displayed around each player marker and its intensity updated in real time. The HLS testbench output of the second-generation ring filter renders the visual in figure 27.

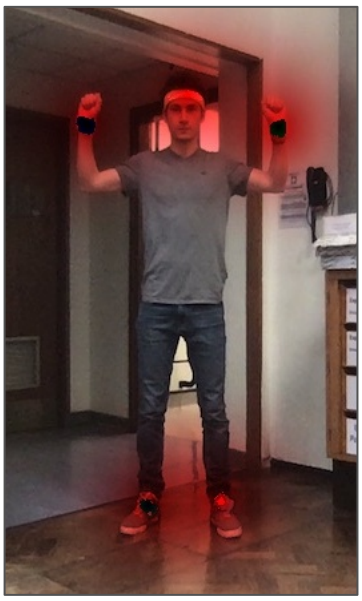


Figure 27. Second-generation ring filter testbench output.

At the later stages of development, the team realized that generation of the filters inside the PL was very costly on performance, as the code had to loop through the image stream twice in order to analyze the image and then apply the filters all within one function call. Trial runs with the full screen filter were giving very low frame rates, which was far from the required performance of the system. It was decided unanimously that to alleviate computational load on the FPGA, calculations would be done in Jupyter and visual feedback would be rendered using the OpenCV (cv2) library. This change benefitted system performance and reduced the complexity of the PL synthesis and implementation. In terms of managing IO, the initial approach was to use the default parameter_1 to control the state of the game from Jupyter and also pass the solution pose coordinates into the PL to perform distance calculations, though after discussion and performance cost evaluation, the team decided it would be better to pass the real-time coordinates from the PL to Jupyter through additional IO parameters. The cost of doing so was requiring 9 additional parameters to accommodate the x-y values of each player marker, though since these ports were only used once at the end of the image data stream it would not significantly compromise the systems performance. This cost was heavily outweighed by the gain in system performance and improvement resource utilization, which was crucial to the playability of the game.

Visual feedback was also rather simple to implement in the form of shape overlays. For the full screen filter, the cv2.rectangle method generates an opaque rectangle on the image, which was modified to span the entire screen. It then had to be overlaid and blended with another layer (alpha layer) using cv2.addWeighted to achieve translucency to replicate the original filter design. The variable alpha with range [0,1] determines the opacity of the blended image and as a result required another mapping of alpha and G and B values to the intensity. In this case, alpha is directly proportional to intensity and is

also inversely proportional to change in the G and B values of the blended layer. The full implementation of the full screen filter in Jupyter can be seen in Appendix A from line X to Y.

On the other hand, having multiple copies of the complete ring filter with dynamic internal gradients rendered in real-time inside the PL would also contribute to lower system performance and increased resource utilization. After deliberation, the team decided to replace the second-generation ring filter with a more intuitive and simpler visual cue that is a marker-centered ring (figure 28). Each ring varied in radius instead of intensity and matched their respective marker color. The function maps the ring radius proportionally to the distance between the player marker and its corresponding solution coordinate such that as the player marker approaches the solution coordinate the ring radius approaches zero. This change was beneficial to the system performance and ultimately achieved the same effect of the system guiding the player towards the objective, however it removed the visuals of "hot and cold" from the gameplay. The full implementation of the third-generation ring filter can be seen in Appendix A.

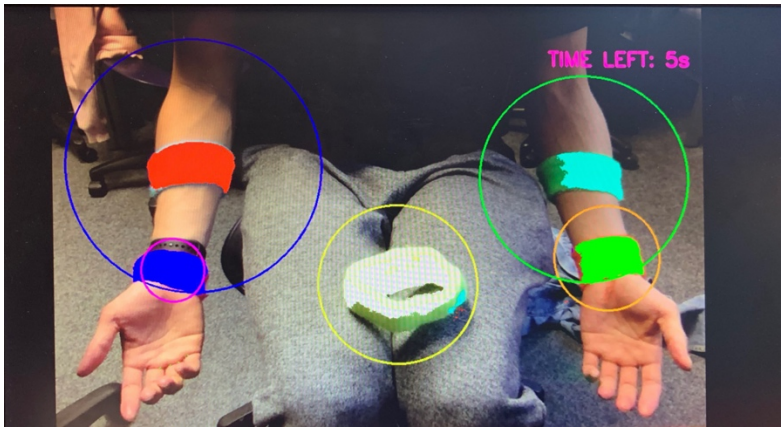


Figure 28. Third-generation ring filter displayed around each marker using OpenCV in real time.

5- Final Design Selection

Having investigated and developed the aforementioned sub-system tools for *Body Break*, it is important to now consider their implementation in the overarching system.

4.1 Rationale

Given that the aforementioned components of *Body Break* are not of equal importance, the final algorithm design decisions were approached sequentially with the overarching system requirements in mind. Starting with the most crucial component (colour detection), and moving towards the lesser importance (i.e. extra photo effects), the team sorted the decisions as shown in the following table.

Higher Importance	Marker detection
	Visual feedback
	Different pose selection
	Button interfacing
Lower Importance	Final pose capture

Figure 29. Ranking of Body Break features in decreasing order of importance.

4.1.1 Marker Detection

Seeing that colour detection held such great importance in the design of *Body Break*, the team decided it best to carefully consider the next step.

Marker Detection Decision Matrix

To help the team make this multi-criteria decision analysis (**reference Perea's lecture**), a decision matrix was used, where each column of the matrix represents a possible solution from the propositions in part *3.31 – Color Detection*, and each row represents a criterion derived from **Part X – System Requirements.** Each criterion is given a rating from 1 to 5, with 5 indicating high **importance** and a 1 low **importance,** as judged from the solution outlines provided in Part 3.2 – marker detection. The values of each column were then summed up to find the design that is most suited to the scope of the project.

	Solution for Final Design			
Criteria	Naïve RGB	Ratio RGB	HSV	HSL
Probability of successful implementation	5	5	4	4
Time to implement	4	4	3	3
Benefit to gameplay	2	2	4	4
Accuracy	1	2	5	5
Estimated hardware utilisation	5	3	3	2
Total	17	16	**19**	18

Figure 30. Decision matrix for marker detection algorithm.

With help from the decision matrix, it is thus clear to see that HSV colour detection was the solution best suited to meet the needs of the project. Compared to RGB detection (naïve and ratio), HSV is exceptionally more performant than the previously explored RGB colour detection, however it has a lower probability of success, and uses a greater amount of resources. The difference between HSV and HSL is minute, and boils down to the greater hardware utilisation required from HSL.

Having considered these factors, the team felt confident to proceed with HSV colour detection as the heart of the *Body Break* mapping system- instigating a chain of decisions for all components.

4.1.2 Visual Feedback

The key design decision regarding the implementation of the visual feedback revolved around whether to use the full filter, or ring filter in the final product.

4.2 Final Design

The outcome of the project was a pose-guessing game that was responsive and intuitive. The final product uses precise HSV color detection to identify colored player markers worn on a player's body, and the corresponding visual feedback would be displayed on-screen to guide the player towards the objective.

After much consideration, the team decided to completely remove the full screen filter from the final iteration of the game as upon testing it was too difficult to find five specific coordinates on an x-y plane with only one form of visual indication. This left the final outcome with the third-generation ring filter that used colored rings around each marker instead of a red gradient. Upon further testing, the team realized that the linear function was responsive and intuitive enough to not require a quadratic function to map distance to ring radius (previously intensity).

When the game is launched, the start menu is loaded and the player is prompted to select the difficulty using the PYNQ board's GPIO functionality.

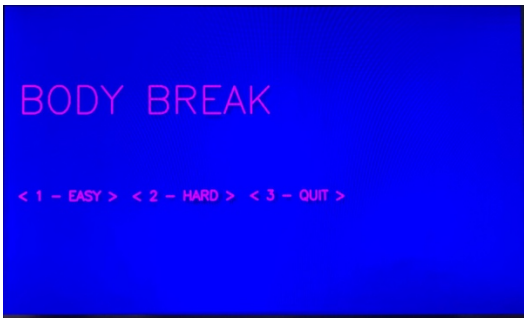


Figure 31. Start menu.

Pressing BTN1 enters the easy difficulty, BTN2 enters the hard difficulty, and BTN3 quits the game by terminating the program. The game randomly selects a solution pose from two different lists of poses, one containing easy poses and the other containing hard ones.

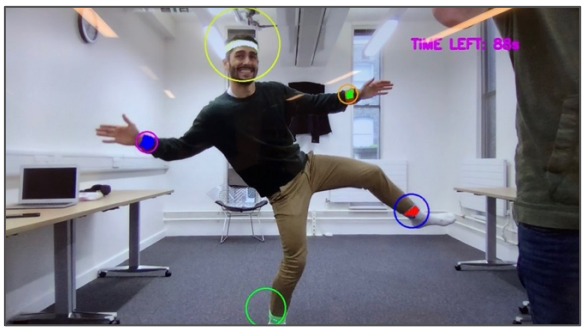


Figure 32. Gameplay.

The game begins rendering the marker-centered rings and simultaneously checks for the winning condition, that is when all individual markers are within certain range of their solutions.

Figure 33. Demonstration of the time out feature.

A timer counts down from a specified time limit and if the solution pose has not been found within that limit, the game displays "TIME OUT" across the screen with the player's final pose as the background.

Figure 34. Demonstration of winning the game.

When the game is won, the game displays a congratulatory message "BODY BROKEN" with the final frame containing the player in the winning pose. In either of these cases the game will proceed to display instructions to return to the start menu or quit the game. The visuals of the game interface are aided by OpenCV to display and overlay images onto the screen. The game also allows the user to quit and reset the game while the game is running, using BTN3 and BTN0 respectively.

6- Hardware Decisions

In order to achieve a high frame per second and meet the aforementioned system requirements, it was essential for the team to make calculated decisions regarding the overarching hardware decisions.

5.1- Loops

5.1.1 Dataflow

It was realized deep into the design of the system that the initial goal of having two large loops in the template filter was greatly problematic to latency. The initial idea was to loop through each pixel sequentially in the frame, analyze the image, and then using that analysis to make a pixel by pixel color transformation to the output. However, this not only greatly increased the latency (and hence frames per second) of the program, but since 5 markers had to be accounted for in two loops- it created a huge resource utilization to the point that the exported IP could not be implemented with the base overlay since it exceeded the hardware resources available on the PYNQ board. To solve this, we focused the FPGA's resources solely on the image analysis, and shifted the visual feedback to the much quicker frame operations in the Processing System using OpenCV.

5.1.2 Pipelining

Having reduced the loops in the template filter to a single image analyser, the focus then shifted on how best to optimise it. The first priority here was to pipeline the inner loop. The core concept underlying pipelining is that one can activate parallelism across the execution of a program. The team realised that since the filter was processing the image row by row, it would be possible to effectuate this parallelism since the output of each row pixel is independent of the other. As a result, this greatly reduced the initiation interval (throughput) of the program, as each similar row operation worked concurrently.

5.2 Variables

A key resource that the team was seeking to bring down concerned the LUTs. At the function's worst state, it was using 64% of the available LUTs on PYNQ, when in reality the target was nearer to 32% as stated in System Requirements.

We managed to reduce them to our target firstly by deeply analysing the conditions within the HSV detection. It was realised that some conditions were unnecessary in our design, such as holding an if condition that v must range from a lower bound of 0.6 and a higher bound of 1. However, the largest possible value of v, by default, was 1, which meant that this condition could be removed, reducing the number of truth tables required for the implemented logic, and hence reducing the number of LUTs.

This process of reduction was applied throughout the program's if conditions to reduce the branching in our FPGA, alleviating the strain on the hardware.

This process of reduction was equally applied to all variables, which were removed where possible.

5.3 Divisions

Divisions were very computationally expensive, as was noticed across the board while investigating the inner works of the function with the analysis perspective in Vivado HLS. This is mainly due to the fact that in long division, a test for overflow is required after each subtraction in the ALU. Therefore, throughout the program, divisions were avoided where possible, and reduced to bare minimum of bits where not possible.

For example, during the initial HSV calculations, the team initially used double division, which is the division of two 64 bit registers. It was noticed to be extremely computationally expensive in the analysis report of Vivado HLS, occupying 30 cycles within each loop iteration- and thus greatly inflating our latency estimation. To solve this, custom precision was applied, reducing the division from 64-bit division to 8-bit division – leading to a huge improvement in the program's utilization.

5.4 Custom precision

Realizing this gain from double division, the same concept of shrinking registers to their bare necessities was applied throughout program- not only for division, but also for multiplication, addition and subtraction. This assisted the group considerably in reducing the total utilization of our project across the board. The custom precisions were defined as follows.

```
typedef ap_uint<8> pixel_type;
typedef ap_int<8> pixel_type_s;
typedef ap_uint<96> u96b;
typedef ap_uint<20> coord_type;
typedef ap_uint<32> word_32;
typedef ap_ufixed<8,0, AP_RND, AP_SAT> comp_type;
typedef ap_fixed<10,2, AP_RND, AP_SAT> coeff_type;
typedef ap_ufixed<10,5> val;
typedef ap_fixed<12,11> htype;
typedef ap_ufixed<10,1> stype;
typedef ap_uint<8> vtype;
typedef ap_ufixed<15,8> convtype;
typedef int disttype;
typedef ap_ufixed<8,1> intensitytype;
typedef ap_ufixed<20,16> dtype;
```

Figure 35. Custom precision

Initially cmath was used, but it was creating a huge amount of latency as noticed with in analysis perspective. Instead of using cmath, hls_math.h was used for functions like sqrt. For the Pow function, 30 clock cycles for squaring a number. Instead, simply multiplied number by itself. The team suspects that these problems likely arised from a lack of a floating point core in Vivado HLS. What this optimized hls_math.h library is likely doing is implementing these functions such as sqrt in a bit-approximate manner, which greatly reduces the number of operations needed at the cost of precision. {MIT,2012}

Anything unexpected, limitations

The visuals of the final product were close to what we had envisioned, the only exception was the style of the visual feedback which was compromised due to resource utilization. However, there were definitely challenges involving both the software and hardware implementations of the design.

5.Limitations

The team realized early on that in order to minimize potential setbacks such as unnecessary complications and. troubleshooting, all high-level C code in the HLS should be implemented under a single top function. Without utility functions, the number of additional logic blocks used in the block design is kept at the minimum of 1. Each additional block in the block design must be synchronized for each IO and rewired to the rest of the hardware design, in which doing so could cause unpredictable behavior and interaction with the base overlay. This allowed the team to avoid RTL issues that were significantly more time consuming and unfamiliar to troubleshoot. Another limitation was that there was no alternative to looping through the image stream twice. The system must run in_data through the first loop to calculate average marker positions and distances before any filter could be applied. The team considered skipping pixels with a slight tradeoff on accuracy to improve performance, however out_data still has to iterate through the entire image stream to apply the filter, which was very costly on performance.

7- Conclusion

Body Break began as an abstract concept to be realized on an image processing system. To do so, the team delegated tasks and specialize workflows with the system requirements in mind. Different implementations of game functionality were explored and carefully considered by the group for the final implementation. *Body Break* successfully adheres to its design specs as stated in the initial management report, however along the way as more was discovered about the system, refinements and modifications had to be made to the design. Not only did the team realize an idea, but the journey also unraveled a deep exploration into the hardware optimizations possible when using a FPGA. The team found the development process incredibly enriching. It is hoped that players of *Body Break* will feel equally so.

REFERENCES

Anon, (2019). *Color maps*. [online] Available at: https://www.pyngl.ucar.edu/Graphics/color_maps.shtml#CreatingColorMapHSV [Accessed 17 May 2019].

High-Level Synthesis with Vivado HLS. XILINX, 2012 [Online]. Available: http://home.mit.bme.hu/~szanto/education/vimima15/heterogen_xilinx_hls.pdf. [Accessed: 16-May- 2019]

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug871-vivado-high-level-synthesis-tutorial.pdf. (2017). [ebook] Xilinx. Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug871-vivado-high-level-synthesis-tutorial.pdf [Accessed 17 May 2019].

Read, P. and Meyer, M. (n.d.). *Restoration of Motion Picture Film*.

Stewart, S. (2019). *What Is The Best FPS For Gaming?*. [online] gamingscan. Available at: https://www.gamingscan.com/best-fps-gaming/ [Accessed 17 May 2019].

Sudhakaran, S. (2019). *Notes by Dr. Optoglass: Motion and the Frame Rate of the Human Eye*. [online] wolfcrow. Available at: https://wolfcrow.com/notes-by-dr-optoglass-motion-and-the-frame-rate-of-the-human-eye/ [Accessed 17 May 2019].

Vivado Design Suite User Guide. (2017). [ebook] Xilinx. Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug908-vivado-programming-debugging.pdf [Accessed 17 May 2019].

Wikipedia. (2019). *HSL and HSV*. [online] Available at: https://en.wikipedia.org/wiki/HSL_and_HSV [Accessed 17 May 2019].

APPENDIX A

```
# coding: utf-8

# # EIE 1st Year Project - Body Break Version 5 (FINAL) 2019
#
# This notebook demonstrates the interface between PYNQ's Processing Systems
 (PS) and a custom HLS hardware block integrated with PYNQ's base overlay
 for real-time Video Processing. It implements a Vertical-Edge detector
 capable of running in >30fps.
#
# This material can be used for academic purposes only. Any commerical use
 is prohibited.
#
# Contact: Alexandros Kouris (a.kouris16@imperial.ac.uk), Ph.D. Candidate,
 Imperial College London

# In[6]:


from pynq import Overlay, Xlnk
from pynq.overlays.base import BaseOverlay

allocator = Xlnk()

#Load customised version of Base-Overlay,that included the custom hardware
 block, instead of using BaseOverlay("base.bit")
#-------------------------------------------------------------------------
 ----------------------------------------------
#ol = Overlay("eie_v25.bit") #Grayscale, full-frame (33fps)
#ol = Overlay("eie_v25_2.bit") #Grayscale, half-frame (33fps)
ol = BaseOverlay("170519_plswork.bit") #Vertical Edge-Detector, half-frame
 (33fps)  - EIE2019


# In[7]:


from pynq.lib.video import *

hdmi_in = ol.video.hdmi_in
hdmi_out = ol.video.hdmi_out

hdmi_in.configure(PIXEL_RGBA)
hdmi_out.configure(hdmi_in.mode, PIXEL_RGBA)


# In[8]:


hdmi_in.start()
hdmi_out.start()


# In[9]:
```

```
from pynq import MMIO
#filter_reference = MMIO(0x00000000,0x10000)
filter_reference = MMIO(0x83C20000,0x10000)  #MMIO Addresses should always
 be double-checked when exporting a Vivado Design


# In[10]:


import time
import random
import cv2
import numpy as np

font = cv2.FONT_HERSHEY_SIMPLEX
Delay1 = 0.3
Delay2 = 0.1
time_limit = 120
rgbled_position = [4, 5]
frame_w, frame_h = 1280, 720
win_condition = False
time_out = False
tolerance = 200
trial_set = [[640, 160], [440, 360], [840, 360], [740, 160], [540, 160]]
easy_set = [[[853, 561], [825, 213], [387, 283], [565, 547], [776, 84]],  #
 Pose 1
            [[932, 614], [864, 362], [486, 236], [1085, 465], [644, 149]]]
             # Pose 4
hard_set = [[[745, 593], [1114, 212], [517, 206], [564, 443], [872, 125]],
 # Pose 2
            [[876, 620], [1164, 182], [1055, 67], [964, 485], [942, 159]]]
             # Pose 5

# Initialize MMIO ports
filter_reference.write(0x30, 0)
filter_reference.write(0x38, 0)
filter_reference.write(0x40, 0)
filter_reference.write(0x48, 0)
filter_reference.write(0x50, 0)
filter_reference.write(0x58, 0)
filter_reference.write(0x60, 0)
filter_reference.write(0x68, 0)
filter_reference.write(0x70, 0)
filter_reference.write(0x78, 0)

while (ol.buttons[3].read() == 0):
    # Boot LEDs
    color = 0
    for led in ol.leds:
        led.on()
    color = (color+7) % 8
    for led in rgbled_position:
        ol.rgbleds[led].write(color)
        ol.rgbleds[led].write(color)
```

```
    time.sleep(1)
color = 0

# Start menu
in_frame = hdmi_in.readframe()
out_frame = hdmi_out.newframe()
frame_copy = out_frame.copy()
cv2.rectangle(frame_copy, (0, 0),  (frame_w, frame_h), (0, 0, 255), -1)
cv2.addWeighted(frame_copy, 1, out_frame, 0, 0, out_frame)
cv2.putText(out_frame, "BODY BREAK", (100, 260), font, 3, (255, 0, 255),
 4, cv2.LINE_AA)
cv2.putText(out_frame, "< 1 - EASY >  < 2 - HARD >  < 3 - QUIT >",
            (100, 460), font, 1, (255, 0, 255), 4, cv2.LINE_AA)
hdmi_out.writeframe(out_frame)
print("Choose a difficulty: Easy (BTN1) or Hard (BTN2) or Quit (BTN3)")

# Difficulty selection
while ((ol.buttons[1].read() == 0) and (ol.buttons[2].read() == 0) and
 (ol.buttons[3].read() == 0)):
    pass
# Premature quit
if (ol.buttons[3].read() == 1):
    break

if (ol.buttons[1].read() == 1):
    print("Selected difficulty: Easy")
    color = (color+2) % 8
    for led in rgbled_position:
        ol.rgbleds[led].write(color)
        ol.rgbleds[led].write(color)
    solution = easy_set[random.randint(0, len(easy_set)-1)]
    time.sleep(Delay1)

elif (ol.buttons[2].read() == 1):
    print("Selected difficulty: Hard")
    color = (color+4) % 8
    for led in rgbled_position:
        ol.rgbleds[led].write(color)
        ol.rgbleds[led].write(color)
    solution = hard_set[random.randint(0, len(hard_set)-1)]
    time.sleep(Delay1)
time.sleep(Delay1)

# Gameplay
print("Game start")
frames = 0
start = time.time()
while (ol.buttons[0].read() == 0 and ol.buttons[3].read() == 0):
    in_frame = hdmi_in.readframe()
    out_frame = hdmi_out.newframe()
    # Get Pointers to memory
    filter_reference.write(0x10, in_frame.physical_address)  # in_data
    filter_reference.write(0x18, out_frame.physical_address)  # out_data
    filter_reference.write(0x00, 0x01)             # ap_start triggering
    while (filter_reference.read(0) & 0x4) == 0:   # ap_done checking
```

```
    pass
if win_condition or time_out:
    hdmi_out.writeframe(out_frame)
    break

### DISTANCE CALCULATIONS ###
# Yellow
hd_x = filter_reference.read(0x30)
hd_y = filter_reference.read(0x38)
# Orange
rh_x = filter_reference.read(0x40)
rh_y = filter_reference.read(0x48)
# Pink
lh_x = filter_reference.read(0x50)
lh_y = filter_reference.read(0x58)
# Green
rf_x = filter_reference.read(0x60)
rf_y = filter_reference.read(0x68)
# Blue
lf_x = filter_reference.read(0x70)
lf_y = filter_reference.read(0x78)

hd_dist = np.sqrt((hd_x-solution[0][0])*(hd_x-solution[0][0]) +
 (hd_y-solution[0][1])*(hd_y-solution[0][1]))
rh_dist = np.sqrt((rh_x-solution[1][0])*(rh_x-solution[1][0]) +
 (rh_y-solution[1][1])*(rh_y-solution[1][1]))
lh_dist = np.sqrt((lh_x-solution[2][0])*(lh_x-solution[2][0]) +
 (lh_y-solution[2][1])*(lh_y-solution[2][1]))
rf_dist = np.sqrt((rf_x-solution[3][0])*(rf_x-solution[3][0]) +
 (rf_y-solution[3][1])*(rf_y-solution[3][1]))
lf_dist = np.sqrt((lf_x-solution[4][0])*(lf_x-solution[4][0]) +
 (lf_y-solution[4][1])*(lf_y-solution[4][1]))

### VISUAL FEEDBACK OVERLAY ###
# Ring filter
hd_r = int(hd_dist/3)
cv2.circle(out_frame, (hd_x, hd_y), hd_r, (0, 0, 255), 2)  # Head,
 yellow
rh_r = int(rh_dist/3)
cv2.circle(out_frame, (rh_x, rh_y), rh_r, (255, 165, 0), 2)  # Right
 hand, orange
lh_r = int(lh_dist/3)
cv2.circle(out_frame, (lh_x, lh_y), lh_r, (255, 0, 255), 2)  # Left
 hand, pink
rf_r = int(rf_dist/3)
cv2.circle(out_frame, (rf_x, rf_y), rf_r, (0, 255, 0), 2)  # Right
 foot, green
lf_r = int(lf_dist/3)
cv2.circle(out_frame, (lf_x, lf_y), lf_r, (255, 255, 0), 2)  # Left
 foot, blue

# Timer
time_now = time.time()
time_passed = time_now-start
cv2.putText(out_frame, "TIME LEFT: {}s".format(int(
```

```
            time_limit - time_passed)), (900, 100), font, 1, (255, 0, 255),
             3, cv2.LINE_AA)
        if time_passed >= time_limit:
            time_out = True

        ### CHECK WIN CONDITION ###
        if (lh_dist <= tolerance) and (rf_dist <= tolerance)
            and (hd_dist <= tolerance) and (rh_dist <= tolerance) and
             (lf_dist <= tolerance):
            win_condition = True

        ############################
        hdmi_out.writeframe(out_frame)
        frames += 1

    end = time.time()

    # POST GAME CONTROL
    if (ol.buttons[3].read() == 1):
        break
    elif (win_condition):
        cv2.putText(out_frame, "BODY", (200, 150), font,
                    4, (255, 0, 255), 10, cv2.LINE_AA)
        cv2.putText(out_frame, "BROKEN", (150, 600), font,
                    4, (255, 0, 255), 10, cv2.LINE_AA)
        print("Game won")
        for _ in range(10):
            for led in ol.leds:
                led.toggle()
                time.sleep(0.2)
        win_condition = time_out = False
        time_passed = time_now = 0
        cv2.putText(out_frame, "< 0 - MAIN MENU >  < 3 - QUIT >", (300,
         460), font, 1, (255, 0, 255), 4, cv2.LINE_AA)
        while ((ol.buttons[0].read() == 0) and (ol.buttons[3].read() == 0)):
            pass
        if ol.buttons[3].read() == 1:
            break

    elif (time_out):
        cv2.putText(out_frame, "TIME OUT", (100, 360),
                    font, 8, (255, 0, 255), 10, cv2.LINE_AA)
        print("Time out")
        for _ in range(10):
            for led in ol.leds:
                led.toggle()
                time.sleep(0.2)
        win_condition = time_out = False
        time_passed = time_now = 0
        cv2.putText(out_frame, "< 0 - MAIN MENU >  < 3 - QUIT >", (300,
         460), font, 1, (255, 0, 255), 4, cv2.LINE_AA)
        while ((ol.buttons[0].read() == 0) and (ol.buttons[3].read() == 0)):
            pass
        if ol.buttons[3].read():
            break
```

APPENDIX B

```
#include <ap_fixed.h>
#include <ap_int.h>
#include <stdint.h>
#include <assert.h>
#include "hls_math.h"

typedef ap_uint<8> pixel_type;
typedef ap_int<8> pixel_type_s;
typedef ap_uint<96> u96b;
typedef int coord_type;
typedef ap_uint<32> word_32;
typedef ap_ufixed<8,0, AP_RND, AP_SAT> comp_type;
typedef ap_fixed<10,2, AP_RND, AP_SAT> coeff_type;
typedef ap_ufixed<10,5> val;
typedef ap_fixed<12,11> htype;
typedef ap_ufixed<10,1> stype;
typedef ap_uint<8> vtype;
typedef ap_ufixed<15,8> convtype;
typedef int disttype;
typedef ap_ufixed<8,1> intensitytype;
typedef ap_ufixed<20,16> dtype;

struct pixel_data {
    pixel_type blue;
    pixel_type green;
    pixel_type red;
};

void template_filter(volatile uint32_t* in_data, volatile uint32_t*
 out_data, int w, int h,int* bluexpose, int* blueypose, int* orangexpose,
 int* orangeypose,  int* pinkxpose, int* pinkypose, int* greenxpose, int*
 greenypose, int* yellowxpose, int* yellowypose){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=w
#pragma HLS INTERFACE s_axilite port=h
#pragma HLS INTERFACE s_axilite port=bluexpose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=blueypose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=orangexpose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=orangeypose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=pinkxpose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=pinkypose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=greenxpose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=greenypose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=yellowxpose // This will NOT work for
 resolutions higher than 1080p
#pragma HLS INTERFACE s_axilite port=yellowypose // This will NOT work for
 resolutions higher than 1080p
```

```
#pragma HLS INTERFACE m_axi depth=2073600 port=in_data offset=slave // This
 will NOT work for resolutions higher than 1080p
#pragma HLS INTERFACE m_axi depth=2073600 port=out_data offset=slave

    w = 1280;
    h = 720;

    coord_type orangecounter = 1;
    coord_type bluecounter = 1;
    coord_type pinkcounter = 1;
    coord_type greencounter = 1;
    coord_type yellowcounter = 1;

    coord_type bluexcurrent = 0;
    coord_type blueycurrent = 0;


    coord_type orangexcurrent = 0;
    coord_type orangeycurrent = 0;

    coord_type pinkxcurrent = 0;
    coord_type pinkycurrent = 0;

    coord_type greenxcurrent = 0;
    coord_type greenycurrent = 0;

    coord_type yellowxcurrent = 0;
    coord_type yellowycurrent = 0;

    float r, g, b;

    convtype max, min, delta;
    htype h_val;

    stype s_val;
    vtype v_val;

    InData_Column_Loop: for (int i = 0; i < h; ++i) {

        InData_Row_Loop: for (int j = 0; j < w; ++j) {
#pragma HLS PIPELINE II=1

            unsigned int current = *in_data++;

            unsigned char in_r = current & 0xFF;
            unsigned char in_g = (current >> 8) & 0xFF;
            unsigned char in_b = (current >> 16) & 0xFF;

            unsigned char out_r = in_r;
            unsigned char out_b = in_b;
            unsigned char out_g = in_g;
            r = in_r;
            g = in_g;
```

```
b = in_b;

float hv, s, v;
float K = 0.f;

if (g < b)
{
    std::swap(g, b);
    K = -1.f;
}

if (r < g)
{
    std::swap(r, g);
    K = -2.f / 6.f - K;
}

float chroma = r - std::min(g, b);
hv = fabs(K + (g - b) / (6.f * chroma + 1e-20f));
s = chroma / (r + 1e-20f);
v = r;
h_val = 360*hv;
v_val = v;
s_val = s;



if (h_val < 28 && h_val > 5 && s_val > 0.7 && v_val > 160){
    out_r = 0x00;
    out_g = 0xFF;
    out_b = 0x00;
    orangexcurrent += j;
    orangeycurrent += i;
    orangecounter++;
} else if (h_val < 90 && h_val > 60 && s_val > 0.3 && s_val <
 0.8 && v_val > 121){
    out_r = 0xFF;
    out_g = 0xFF;
    out_b = 0xFF;
    yellowxcurrent += j;
    yellowycurrent += i;
    yellowcounter++;
} else if (h_val < 156 && h_val > 116 && s_val > 0.3 && v_val >
 128){
    out_r = 0x00;
    out_g = 0xFF;
    out_b = 0xFF;
    greenxcurrent += j;
    greenycurrent += i;
    greencounter++;

} else if (h_val < 210 && h_val > 180 && s_val > 0.3 && v_val >
 80){
```

```
                out_r = 0xFF; // showing the pixel as red.
                out_g = 0x00; // no green component
                out_b = 0x00; // no blue  component
                bluexcurrent += j; // summing all the x coordinates for all
                 pixels detected
                blueycurrent += i; // summing all the y
                bluecounter++; // counting all the pixel detected in this
                 color
            } else if (h_val < 350 && h_val > 310 && s_val > 0.5 && v_val >
             128){
                out_r = 0x00;
                out_g = 0x00;
                out_b = 0xFF;
                pinkxcurrent += j; // summing all the x coordinates for all
                 pixels detected
                pinkycurrent += i; // summing all the y
                pinkcounter++;
            }
            //
            unsigned int output = out_r | (out_g << 8) | (out_b << 16);
            *out_data++ = output;
        }
    }

    *bluexpose = bluexcurrent/bluecounter;
    *blueypose = blueycurrent/bluecounter;

    *orangexpose = orangexcurrent/orangecounter;
    *orangeypose = orangeycurrent/orangecounter;
    *pinkxpose = pinkxcurrent/pinkcounter;
    *pinkypose = pinkycurrent/pinkcounter;

    *greenxpose = greenxcurrent/greencounter;
    *greenypose = greenycurrent/greencounter;

    *yellowxpose = yellowxcurrent/yellowcounter;
    *yellowypose = yellowycurrent/yellowcounter;


}
```

```
    elif (ol.buttons[0].read() == 1):
        print("Restarting game")
        for led in ol.leds:
            led.off()
        time.sleep(0.1)
        for led in ol.leds:
            led.toggle()
            time.sleep(0.1)
        win_condition = time_out = False
        time_passed = time_now = 0

    print(f"{frames} frames took {end-start} seconds at {frames/(end-start)}
     fps")

# QUIT GAME
print("Quitting game")
for led in ol.leds:
    led.off()
for led in rgbled_position:
    ol.rgbleds[led].off()
hdmi_in.close()    # Don't forget to run this to free memory
hdmi_out.close()   # NEVERFORGET NEVERFORGET NEVERFORGET :p
print("HDMI Released")
```

APPENDIX C

```
#include <ap_fixed.h>
#include <ap_int.h>
#include <cassert>
#include <iostream>

#include <hls_opencv.h>

typedef ap_uint<8> pixel_type;
typedef ap_int<8> pixel_type_s;
typedef ap_uint<96> u96b;
typedef ap_uint<32> word_32;
typedef ap_ufixed<8,0, AP_RND, AP_SAT> comp_type;
typedef ap_fixed<10,2, AP_RND, AP_SAT> coeff_type;

struct pixel_data {
    pixel_type blue;
    pixel_type green;
    pixel_type red;
};


void template_filter(volatile uint32_t* in_data, volatile uint32_t*
 out_data, int w, int h, int* bluexpose, int* blueypose, int* orangexpose,
 int* orangeypose, int* pinkxpose, int* pinkypose, int* greenxpose, int*
 greenypose, int* yellowxpose, int* yellowypose );

int main() {

    //Specify Input Image Absolute Path
    cv::Mat src_hls = cv::imread("/home/na6518/fpga/Project/Yellow.jpeg",
     CV_LOAD_IMAGE_UNCHANGED);
    std::cout << "Image type: " << src_hls.type() << ", no. of channels: "
     << src_hls.channels() << std::endl;
    //src_hls.convertTo(src_hls, CV_8UC3);
    //cv::cvtColor(src_hls, src_hls, CV_BGR2RGBA);


    std::cout<<"SIZE  "<<src_hls.size()<<std::endl;
    uchar *data_p = src_hls.data;

    //Make sure that these much input image resolution
    int w = 1280;
    int h = 720;


    int bluexpose = 0;
    int blueypose = 0;
    int orangexpose = 0;
    int orangeypose = 0;
    int pinkxpose = 0;
    int pinkypose = 0;
    int greenxpose = 0;
    int greenypose = 0;
    int yellowxpose = 0;
    int yellowypose = 0;
```

```
    uchar *image = (uchar *)malloc(w*h*4);

    for (int i=0; i<w*h; i++){
        image[4*i + 0] = data_p[3*i + 2]; //R - R
        image[4*i + 1] = data_p[3*i + 1]; // B - B
        image[4*i + 2] = data_p[3*i + 0]; // G - G
        image[4*i + 3] = 0;
    }

     template_filter((volatile uint32_t *)image, (volatile uint32_t *)image,
      w, h, &bluexpose, &blueypose , &orangexpose, &orangeypose, &pinkxpose,
      &pinkypose,  &greenxpose, &greenypose,  &yellowxpose,  &yellowypose);
    //mapping_function((volatile uint32_t *)image, (volatile uint32_t
     *)image, w, h, parameter_1, distance);

    for (int i=0; i<w*h; i++){
        data_p[3*i + 2] = image[4*i + 0];
        data_p[3*i + 1] = image[4*i + 1];
        data_p[3*i + 0] = image[4*i + 2];
    }
//
//  std::cout << "Blue \t \t x \t" << bluexpose << std::endl;
//  std::cout << "Blue \t \t y \t" << blueypose << std::endl;
//  std::cout << "Orange \t x \t" << orangexpose << std::endl;
//  std::cout << "Orange \t y \t" << orangeypose << std::endl;
//  std::cout << "Pink \t \t x \t" << pinkxpose << std::endl;
//  std::cout << "Pink \t \t y \t" << pinkypose << std::endl;
//  std::cout << "Green \t x \t" << greenxpose << std::endl;
//  std::cout << "Green \t y \t" << greenypose << std::endl;
//  std::cout << "Yellow \t x \t" << yellowxpose << std::endl;
//  std::cout << "Yellow \t y \t" << yellowypose << std::endl;

    //Specify an Absolute Path for Storing Output Image
    cv::imwrite("/home/na6518/fpga/Lab_Files/
     Project_Part2/231.jpg",src_hls);

    free(image);

    return 0;

}
```