Imperial College London

Department of Electrical and Electronic Engineering

First Year EIE FPGA Project

Kaichun Hu

Aufar Laksana

Shreyus Bagga

Word Count: 2579

Contents

***How Blurs and Filters Work - Computerfile**

1. FPGA Experiments

a. Project Introduction

During our lab sessions, our group was exposed to the programming language Catapult C and learnt of its uses by progressing through the multiple exercises. Reading through the code examples enlightened us to the uses of matrices of RGB values, and the process of sequentially applying 'masks' over pixel arrays in order to obtain the desired image effect (such as grayscale or a Sobel filter). Learning about the advantages and disadvantages of pipelining and unrolled code helped us to understand how a certain software implementation choice would affect factors such as the area of hardware used in the motherboard (and power consumption as a result), and the number of clock cycles required in a process, displayed by the Gantt charts obtained.

b. Cross Product

The vector cross product experiment served as a simple example of how to set up the Catapult C software to be compatible with the DE0 board and Cyclone III chip. However, more importantly, it also clearly introduced Gantt charts while demonstrating the effect of pipelining versus unrolling, with respect to the execution time and the circuit area used up on the board.
The program worked by using the first 5 switches on the DE0 as input for the first Vector A, and the second 5 switches as Vector B. Due to the nature of the implementation, whatever binary value that was selected using the switches was applied to all 5 elements of that vector. E.g. if the decimal value 4 was inputted using the first 5 switches, then Vector A would be (4, 4, 4, 4, 4).

The definition of a dot product of two 1x5 vectors **a** and **b** is:

$$a.b = \sum_{i=0}^{4} a_i b_i$$

The result is a scalar value. The program used a for loop to read every single element of each vector and multiply the element with the corresponding other vector's element, and accumulating the result. The result was then written out.

As can be seen in Figure 1, pipelining the cross product program results in only two clock cycles till completion. This is because only the read and write operations use up a clock cycle (as indicated by the Gantt chart io_read inputs and the io_write output which starts at the second clock cycle) while the operations between the read/write can be thought of as combinational logic, with no timing involved.

This is in direct contrast to the scheduling of the unrolled version:

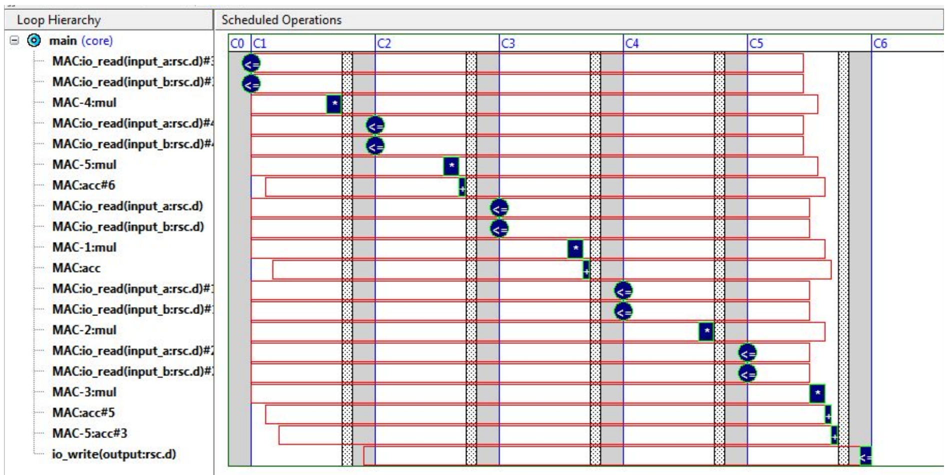

Loop Hierarchy
Scheduled Operations
main (core)
MAC:io_read(input_a:rsc.d)#
MAC:io_read(input_b:rsc.d)#
MAC-4:mul
MAC:io_read(input_a:rsc.d)#
MAC:io_read(input_b:rsc.d)#
MAC-5:mul
MAC:acc#6
MAC:io_read(input_a:rsc.d)
MAC:io_read(input_b:rsc.d)
MAC-1:mul
MAC:acc
MAC:io_read(input_a:rsc.d)#
MAC:io_read(input_b:rsc.d)#
MAC-2:mul
MAC:io_read(input_a:rsc.d)#
MAC:io_read(input_b:rsc.d)#
MAC-3:mul
MAC:acc#5
MAC-5:acc#3
io_write(output:rsc.d)
C0
C1
C2
C3
C4
C5
C6

Figure 2: Unrolled Gantt chart

As can be seen in Figure 2, an unrolled version an extra clock cycle is used every time a new element is read from A and B in the same clock cycle, whereas the pipelined version . This results in 6 clock cycles being used. The trade-off with pipelining is that, on the DE0 board, more circuit area is used since multiple inputs were read in the same clock cycle, while unrolled uses only one circuit area of the board.

c. <u>Sobel Filter</u>

In this exercise a kernel convolution matrix was created. It is a small matrix which we can apply cross the whole image.

The input image got totally transformed by using kernel convolution. Since the kernel matrix was used across the entire image.

The kernel matrix was applied at the top left corner of the initial image and move by one column per time. The appearance of the new image depends on the value inside the kernel matrix. By changing the values inside, we can achieve different effect.

At first the 3*3 matrix was placed on the left hand corner and the value for each pixel was multiplied with the corresponding value in the kernel matrix. For example, top left multiplied with top left in the kernel, bottom right was multiplied with the bottom right in the kernel and so on.

The process of the multiplication is shown below in figure 3.

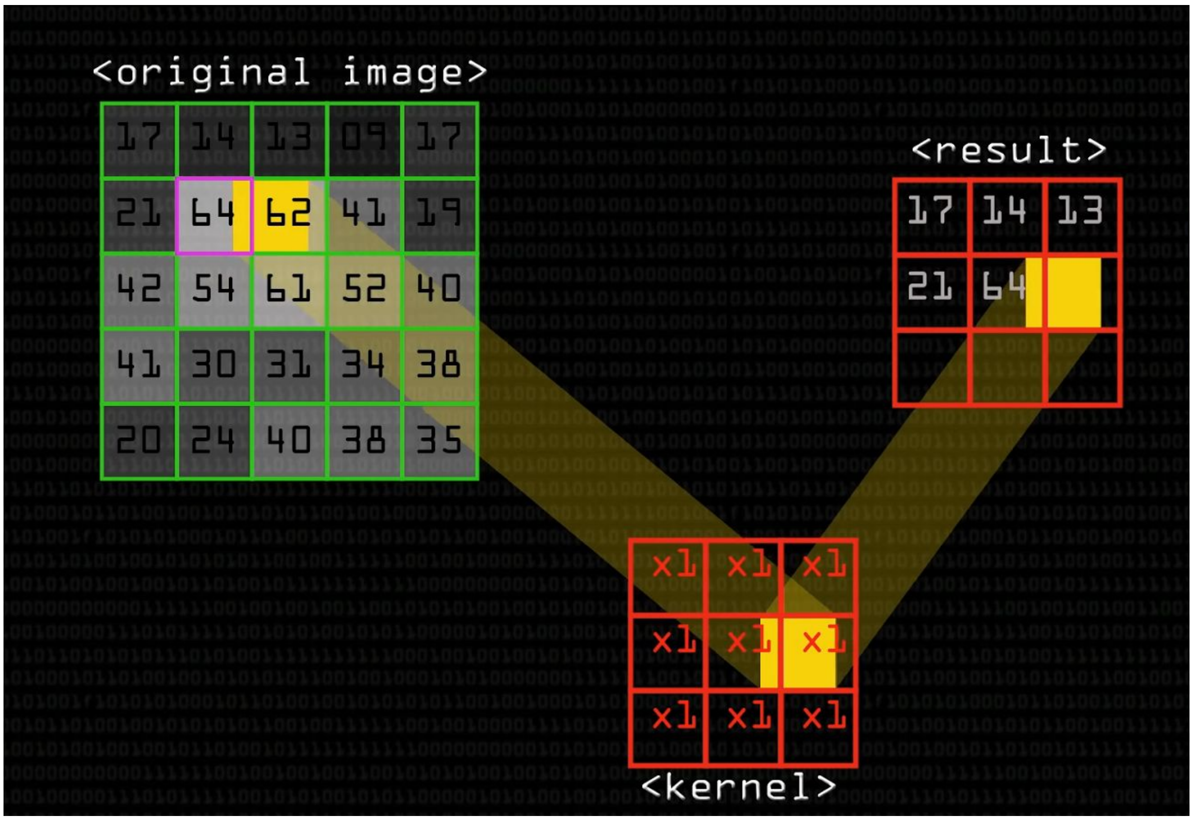


Figure 3 *

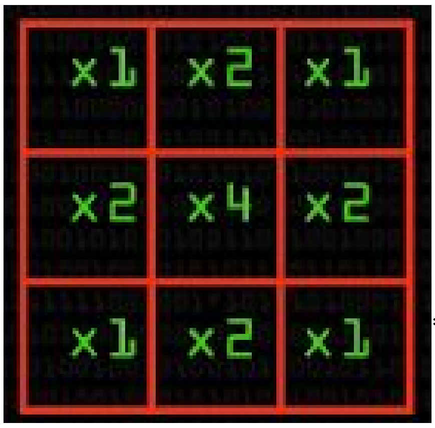
Secondly, after the values has been all collected. They were added up to a single value. And then it was divided by the number of grid there was in the kernel matrix. The blur filter was implemented by setting all of the values inside the matrix to 1. Therefore after summing and dividing, the result that was calculated should match the average value of all 9 pixels. Finally the result was transferred to the output image and it was placed on the centre corresponding to the position of the kernel matrix on the input image. The important part that has been taking care of was to output the data on to a new image since the values on the input image was taken many times before the display process.

During the process of applying kernel matrix across the input image, there was some problem with the edge and there are different method to solve it. One way is to wrap around the image and that can make the edge closer to the desire data. But we have chosen to ignore it since it was only one pixel long and combine it with a whole image the difference is hardly noticeable.

An alternative method was to use Gaussian blur it can make the edges runs smoother and it is more commonly used.

The following diagram shows the kernel matrix for Gaussian blur.

Figure 4 *:



***How Blurs and Filters Work - Computerfile**

The advantage of Gaussian blur is that when an edge get encountered by the 3*3 matrix. Due to the uneven distribution of the values contained in the matrix, result we get after sum and divide will be closer to the value on the lighter side of the edge. Therefore more futures of the edge get preserved.

Now move to edge detection, during scanning through the input image the intensity will change according to the kernel matrix that got applied. A rapid change would indicate an edge and a smooth change shows a slow change of colour.

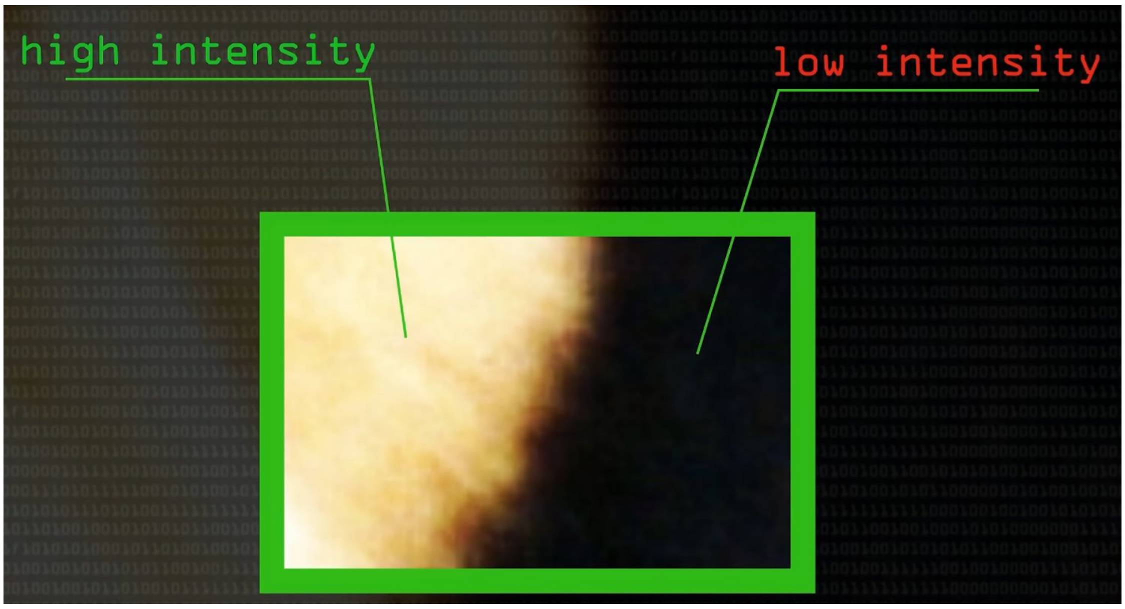


Figure 5 *

The following matrix is a kernel matrix for detecting edges that arise in x direction. The matrix was set up to detect vertical lines by comparing values generated by the leftmost column and the rightmost column. Also since the centre column contain all zeros, the difference will only occur when a vertical line got encountered.

The output result depends on the input data of the image, and if there is no colour transition the output would equal to zero.

If the output value is positive that indicates there is a transition from dark to light. Else if the output values is negative that shows there is a transition from light to dark. And similarly the kernel matrix for y direction only detects horizontal lines. When we apply both kernel matrix on a greyscale image a white grey black image was produced.

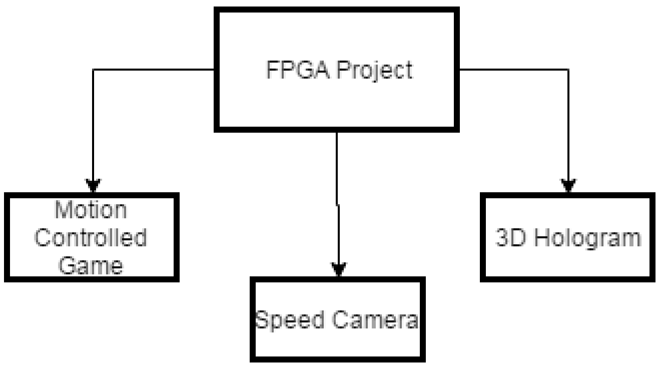
And by applying a^2 + b^2 = c^2 the negative sign disappears. So the result will also be positive. The colour will be consistent if the output value equals to zero.

2. FPGA Project

a. Project Introduction

In our group meetings, we brainstormed several ideas for the project. One of the ideas was a *Speed Camera* where the camera would use motion and object tracking algorithms in order to determine the speed an object was moving at. Another idea was to apply certain filters such as elongation of the captured image in order to be able to output it as a 3D hologram.

However, we finally decided on a *Motion Controlled Game*, since it was an area that interested all three members of the group. Each group member agreed that they had skills which would contribute to the project, for instance, Kaichun's artistic ability would be beneficial in developing graphics for the game.

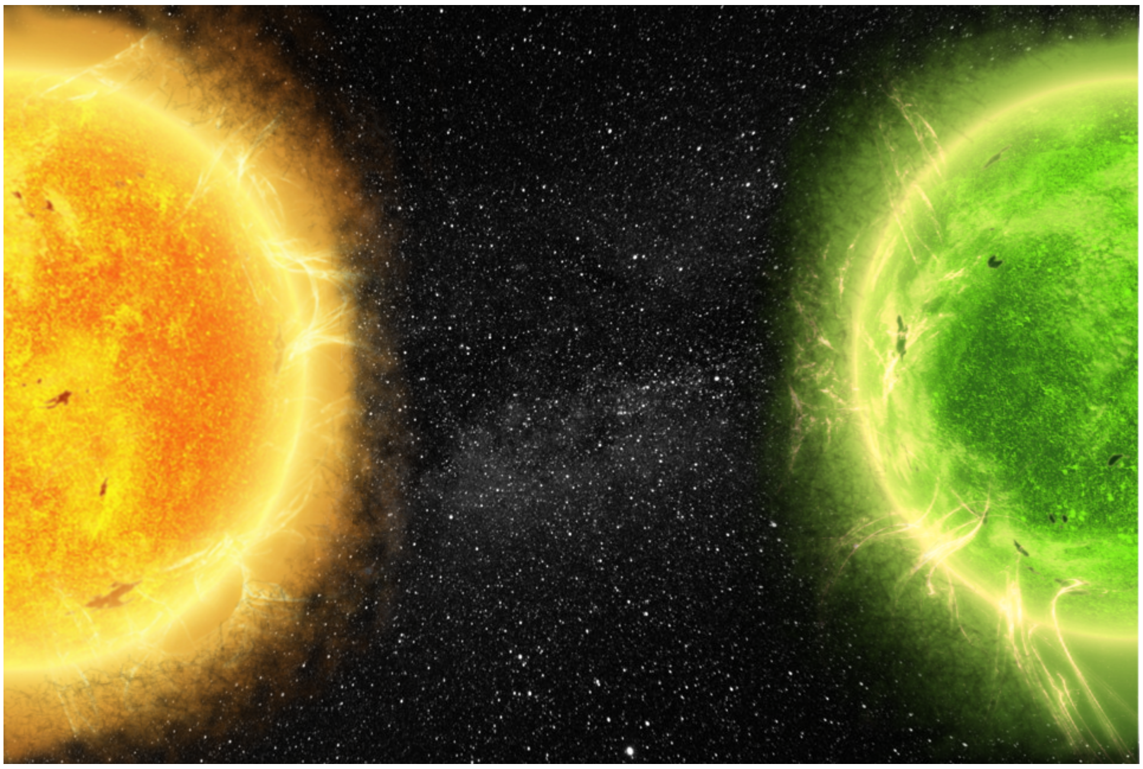


b. High Level Description

i. Gameplay

For this project, we decided to create a new 2 player game, inspired by two classic arcade games; *Pong* and *Space Invaders*.

The screen is divided into two halves, one for each player. Each player has a *spaceship* which they control. Behind the spaceship is the player's *planet*, which is divided into 5 parts. The player must defend their planet from the attacks of the other player. Behind the planet is the *core*, which is initially covered by the planet.



The *spaceships* can fire rockets. These rockets are used to destroy the other player's planet. For instance, if Player A fires a rocket at Player B's planet, and it reaches it, that part of the planet is destroyed.

Once a part of the planet is destroyed, it remains so until a new game session is started. This means that the core is now exposed, and if the core takes a direct hit from another rocket, the player loses one life.

However, players can prevent the rockets from destroying their planets by either firing a rocket of their own at incoming rocket before it reaches the planet, or by blocking the rocket using the spaceship, since the idea is that the spaceship has a force field which can block the damage from the rockets.

To simplify the game, the screen is further divided into 5 sections. Each section contains a part of the planet. The spaceship can be moved between the 5 sections depending on how the player controls it.

Furthermore, each player starts off with 3 lives. The game ends when one player has no more lives remaining, and the winner is the player which still has a life. A player loses a life when their *core* takes a direct hit from a rocket.

ii. Controls

In order to control the position of the spaceships, the 2 players will hold cards of different colours. Player A will hold a green card, and Player B will hold a red card. This is so that we can easily detect the different regions in colour.

Since the screen will be divided into 5 regions, when we take the image in for processing, the position of the cards in the image will correspond to the position of the spaceship within the game.

In order to simplify the game, initially, there will be no separate control to fire a rocket. Instead, the spaceship will fire a rocket automatically every 2 seconds.

iii. Motion/Colour Detection

Once we have read the image from the camera, we store the RGB values in a 3D array (640x480x3). We make 2 copies of the array. By analysing the pixels, we can identify the regions of mostly Red and Green, which will be the square cards the players will be holding.

We initially convert the image to grayscale by averaging the RGB values of each pixel. This image is then converted to a binary image by using the thresholding process. This is done by comparing each pixel value to some preset threshold value, and if the value is greater, we set the pixel to 1, else, we set it to 0. This essentially eliminates the background, so it is easier to process the image.

We then want to detect the boundary of the object in order to ensure that the Red/Green object is the square card the players are holding, and not something else. We take a pixel as a reference and use it to detect other object pixels.

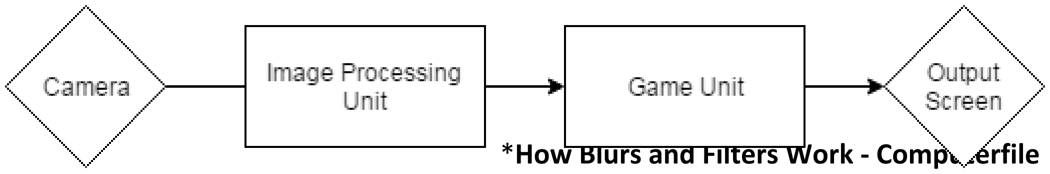
We then calculate the area of the object by summation of the pixels within the boundary of the object. If the area of the object is within some predetermined range, we detect it as the square card held by the players.

Once we know the object we are detecting is either the Red/Green card, we store the position of the object pixels. We then return to the original image by accessing the second copy of the array and we check the RGB values of those stored object pixels. For instance, if the Red component of the RGB value is much greater than that of the Green and Blue, we can assume that the object is the Red card. Similarly, if it is mostly Green, then we can assume it is the green card.

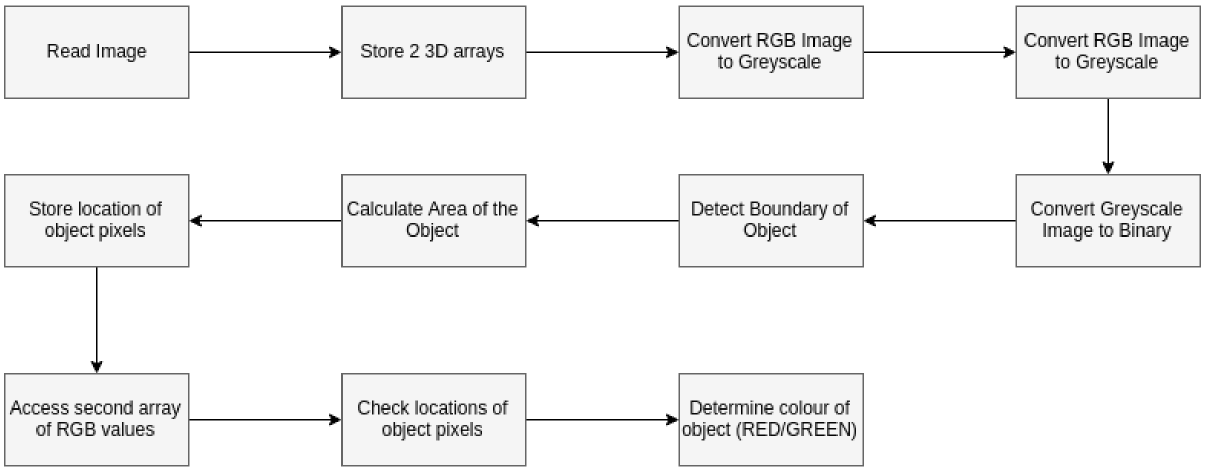
The position of the coloured card will correspond to the position of the spaceship in the game.

c. Block Diagram of System

Using the camera on the FPGA, we capture an RGB image. This image will then be passed onto the *Image Processing Unit*, which will analyse the pixels in order to determine the position of the cards, which will be translated to the positions of the spaceships.



This is done using the following process:



Read Image
Store 2 3D arrays
Convert RGB Image to Greyscale
Convert RGB Image to Greyscale
Convert Greyscale Image to Binary
Detect Boundary of Object
Calculate Area of the Object
Store location of object pixels
Access second array of RGB values
Check locations of object pixels
Determine colour of object (RED/GREEN)

After we determine the colour of the object, its location is passed onto the game unit, which will move the spaceship accordingly. The output of the system is the game itself. The initial captured image is not outputted to the final screen.

d. <u>Expected Outcomes</u>

In order to complete the project in the given amount of time, we will have to make compromises. For instance, the graphics and sprites used in the game will not be 3 dimensional nor of the greatest quality, since the most important aspect of the project would be the colour and motion detection. Also, the spaceship will not fire when the player wants, rather, it fires automatically every 2 seconds.

Goals:

- Working Motion/Colour detection
    - Can distinguish between coloured cards and other objects of similar colours
    - No delay in movement of cards and movement of spaceship in game.
- Working Gameplay
    - Correct game mechanics
    - Simple sprites and graphics
    - No delay

e. <u>Future Developments</u>

Our planned project result will have simple graphics and controls. Therefore, future improvements would involve having a higher quality visuals, animations, backgrounds, and include more refined controls such as being able to control the direction in which to fire a rocket. In other words, a spaceship in one block could shoot in the direction of another block, increasing the overall game difficulty. This could perhaps lead to selecting the difficulty level at the start of the game, easy being our planned current implementation, and the harder level having this extra control and perhaps faster rockets.

Another development would be to eradicate the need for two different coloured paddles by solely relying on hand gestures to control position of each spaceship. Since our planned game will shoot rockets every 2 seconds, using hand gestures could allow for controlling when to fire a rocket by having the player change hand gesture and using edge detection to identify this.

Finally, another improvement could be allowing more than 2 players to participate, with a team-based system. This could either be achieved with a more sophisticated edge-detection implementation, or through the utilisation of the PS/2 inputs of the DE0 motherboard, connecting a keyboard with which to control the game movement.

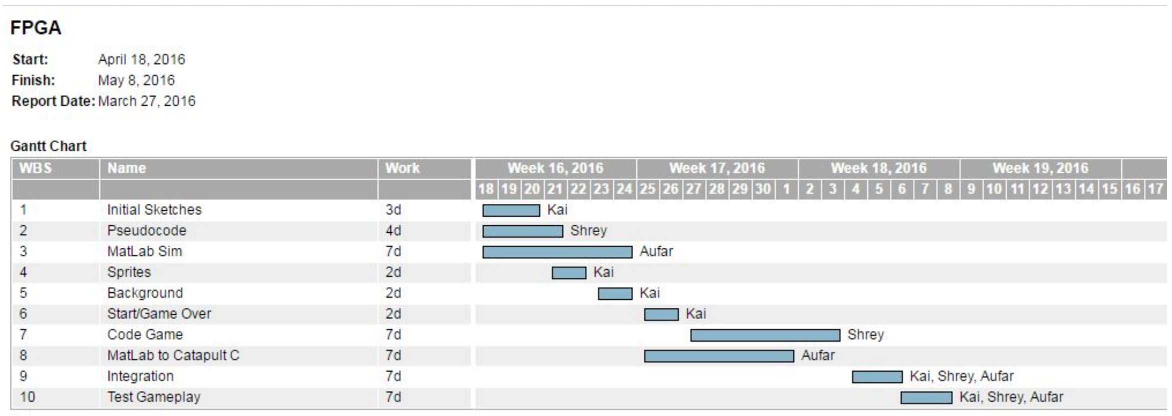
f. Project Management

Each member of the group has been assigned certain tasks in accordance with their strengths. For instance, Shreyus has had some previous experience working with game libraries, and has been tasked with coding the game. Aufar has worked with MatLab before, and has been tasked with creating a MatLab simulation of the colour detection. Kaichun's artistic ability is used to its full potential in the design of sprites and graphics for the game.

If a member of the group has finished their assigned task, they will then help other group members to complete their tasks.

Shrey	**Kai**	**Aufar**
• Write Pseudocode for game • Code test version of game	• Create initial design sketches of game interface • Create game Sprites • Create game Background • Create Start Menu/Game Over screens	• Create MatLab version of Colour Detection • Translate MatLab code into Catapult C code

As a Group
• Integration of Colour Detection block with Game Controls • Test Gameplay



FPGA

Start: April 18, 2016
Finish: May 8, 2016
Report Date: March 27, 2016

Gantt Chart

WBS	Name	Work
1	Initial Sketches	3d
2	Pseudocode	4d
3	MatLab Sim	7d
4	Sprites	2d
5	Background	2d
6	Start/Game Over	2d
7	Code Game	7d
8	MatLab to Catapult C	7d
9	Integration	7d
10	Test Gameplay	7d

4. References

Basic Geometric Shape And Primary COlour Detection Using Image Processing On Matlab
-Chhaya, Khera & Kumar S, Vol 4 Issue 5 May 2015

How Blurs and Filters Work - Computerfile. Available at:
https://www.youtube.com/watch?v=C_zFhWdM4ic&feature=iv&src_vid=uihBwtPIBxM&annotation_id=annotation_4252716785
-Computerphile, Sean Riley, (Accessed: 28.03.15)

Sobel operator (2016) in Wikipedia. Available at: https://en.wikipedia.org/wiki/Sobel_operator (Accessed: 28 March 2016).