Word count: 6630

Coin Detection Using Catapult C on an FPGA

Mikkel Jordahn MJ2214@IC.AC.UK CID 00924570

Marco Lavalle ML7715@IC.AC.UK CID 01046115

Thomas Nugent TN1115@IC.AC.UK CID 01090230

CONTENTS

1 BACKGROUND

The original idea for this project was to use the FPGA camera feed to provide an image, which would be analysed, and have coins identified, thus functioning as a coin detector. However, it became apparent that this did not fulfil the requirements of the assignment, as the project had to work in real-time.

A new idea sprung to life, which included using a moving conveyor belt, the camera and two laser pointers, to add up all the coins present on the belt, once they had all been detected. The idea proved incredibly difficult to implement. Therefore, it was decided to use the mouse as a secondary input to create a real-time aspect, in which the user would hover over a certain section of coins. When doing so, the FPGA would then show the value of all of these coins in this region on the 7-segment display.

Thus, through the processing of the video feed, and through the use of various algorithms, including, but not limited to, the "grass-fire" algorithm (1), the project should be able to detect coins of the currency Sterling Pounds and determine the total value of all the coins present in the cursor area.

2 HIGH-LEVEL SYSTEM DESCRIPTION

Several changes have been made to the project since the Management Report. First of all, the idea of having two different "modes" from the Management report has been removed. Instead, it has been decided that the mouse will be used as an additional input at all times, and the only adjustable thing will be which part of the image the user wants analysed. This is mainly due to the resource limitations on the FPGA. In addition, it was also determined that constant conditions could not be relied on, so the camera exposure had to be changeable, rather than having to change the exterior light conditions. This meant learning how to change the threshold on the camera, such that when it comes to running demos, the project will work under all conditions.

Another way to keep the conditions for experimenting more or less constant, was to 3D print a stand. This could hold the camera at anywhere between 10-100 cm from the table top. After some measurements, it was originally decided that 50 cm was a good height from the table, as it would allow detection of a fair amount of coins, whilst ensuring that each different type of coin would have a discernible difference in size. This should allow the FPGA to detect different coins, regardless of any noise on the camera feed. The parts have been designed and printed such that the height of the stand is adjustable, should this be necessary.

After the first meeting with the project leader, some time was spent determining the new specifications, and how the project was to incorporate real-time constraints. It was found that median filtering or average filtering was unnecessary for what the project was attempting to fulfil and would have just wasted FPGA resources. Although median or average filtering may have removed some noise from the video feed, it was determined that in reality this was not required. Rather than this, each different coin would just be coded as a pixel range for its size rather than a fixed value. This way, although potential noise could alter the amount of white pixels in a blob slightly, in the end this should not affect the type of coin that the FPGA identifies.

In order to be able to identify objects, and in this case coins, a blob detection algorithm was required, but the original approach was changed. Instead the grass-fire algorithm was implemented as described

by What-When-How (1), as this allowed a sequential code design, suitable for implementation in Catapult C. This also meant that the complementing code in the original report was unnecessary for the project to function. The grass-fire algorithm is performed on a matrix of the same size as the mouse window, which contains an image based on the video camera feed and the cursor selection. This way the program only does blob detection on the coins within the mouse window.

Thus by the use of greyscale conversion, binary image conversion and application of the "grass-fire" algorithm and a user controlled mouse, this project should be able to detect coins of different value in the area of the cursor and in the end sum them together and display the total value on the 7-segment display of the FPGA.

3 System Schematic

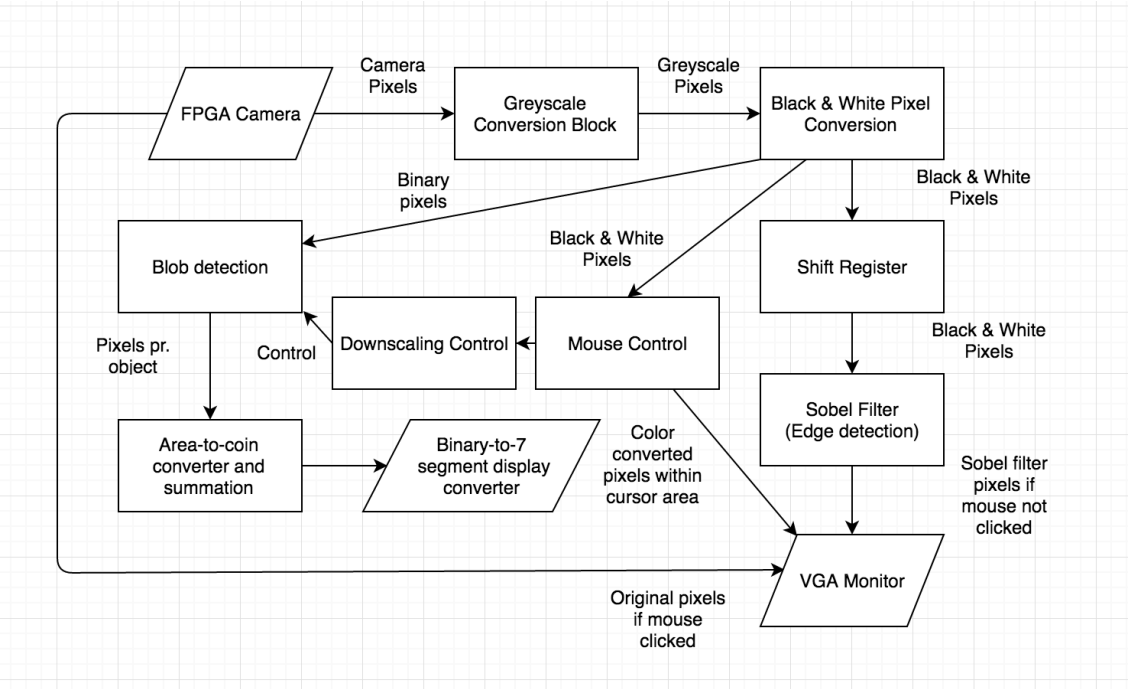


Figure 1 - System schematic

The FPGA camera feed is directed to a greyscale conversion block and then to a black and white conversion block, which uses a threshold value to change the image into one consisting of just black and white. This is then sent through a shift register to ensure that the delay between blocks is consistent. This is needed so that the display is synchronised, and shows the correct image. Finally, a Sobel filter is applied to the image and then it is sent to the VGA monitor to be displayed as the background. The mouse window is controlled by the mouse control block, which sends the corresponding colour change to the VGA display, and also controls what information is sent to the blob detection block. The output of the blob detection and area-to-coin converter is sent to a final converter which translates the coin value into digital signals to drive the 7 segment display. Finally, the FPGA camera is also sent to the VGA display to be used when the mouse is clicked. (See Figure 1)

4 PROJECT PLANNING & MANAGEMENT

Please see attached Gantt Chart (Appendix A).

Almost everything ran according to the Gantt chart, except for one major issue. It had been expected that implementing the grass-fire algorithm was going to be the hardest part of the project. However, once the algorithm had been shown to work, it took three days to be able to maintain stable inputs and outputs, due to the different timings between blocks. This was not something that had been taken in account when making the Gantt chart at first, and as a result was a larger time consumer than expected.

5 DESIGN SPECIFICATION

The FPGA is to read the image feed that is provided from the camera and identify the coins.

The following specifications have been set for the project:

- Will only be able to identify a maximum of three different types of coins.
- Will only be able to analyse a maximum of ten coins at the same time.
- Is in real-time since mouse input is provided directly by the user, and processing occurs whenever the mouse is moved.
- Coins may not overlap or touch.
- Will detect coins by area rather than colour.
- A black, flat background is required.
- Light threshold will be set on the camera prior to the program being run.
- The camera will be set 50 cm from the table top, and will be attached to 3D printed stand.
- The mouse setting will be able to identify in a square of 128x128 pixels.
- No other shapes or objects may be present in the camera range with similar areas to the coins.
- Only Sterling Pound coins are used.
- The total value will be shown on the 7-segment display.
- The VGA monitor will be used to display the image and show the user where the cursor is.

These design specification points are based on the groups understanding of the tools, the limited resources on the FPGA and the time constraints on the project.

6 DESIGN PROCESS

For the majority of the project Catapult C synthesis has been used, and the Verilog files produced from this were then connected in Quartus. It was debated whether using Verilog would be more useful, but in the end Catapult C was chosen, as the group as a whole already knew how to program in C. In addition to this, it was determined it would be easier to configure some of the already existing blocks, such as the mouse control, to serve the purpose required for the project. By using Catapult C, it was easier to modify the existing code, rather than having to translate it into Verilog. By keeping the design consistent, i.e. by only using Catapult C and not using Verilog at all, it was also determined that this probably would increase expertise and a baseline understanding of the technology and design of the project

For the sake of modularity and ease of debugging, subcomponents were created in Catapult C and then connected in Quartus. This, rather than having one large program and block, would allow each separate block in Quartus to be debugged separately. In the end, separate blocks were created for greyscale pixel conversion, binary image conversion, mouse control and setting, blob detection, area calculation, and finally binary-to-7-segment-display decoder.

The only issue with the approach of creating subcomponents in Catapult C and then connecting them in Quartus, was that all of the blocks had to be synchronized. This meant that precautions had to be taken, such as ensuring that when copying the current frame from the video feed, the copying should only start when the first pixel in the frame was reached. If this had not been done, a frame could be copied incorrectly, for example such that the top half of the frame would be at the bottom of the matrix, and the bottom half of the frame at the top. To fix this issue, a counter was created that would count from 0 to 307,200, the total number of pixels on the display. When this was reached, the counter would reset to 0, and output a 1, indicating the start of a new frame.

An exception to the Catapult C approach was when programming the grass-fire algorithm. Because of the complexity of this algorithm, it was determined that it was good to program the algorithm in C++ first, because this would allow easy testing of the algorithm and the code. In C++, the compiling time is much smaller than the compiling time in Catapult C, followed by the compiling time in Quartus. This way, it was also easy to show the transformed matrix to the user, in order to see that the algorithm had performed the action it was supposed to. Once it was shown to be working, the program was then "translated" into Catapult C, at which point the group knew that the algorithm was working.

7 PROPOSED SOLUTIONS

Initially, the idea of blob detection was to be implemented through the process found in the paper by Kiran et la (2). With this approach, it is necessary to go through a process of greyscale conversion, median filtering, binary image conversion, and in the end using complimenting to perform the blob detection. However, several of these ideas were quickly put aside for others. The median filtering was determined unnecessary, and the grass-fire algorithm found on What-When-How's website (1) was easier to understand and implement, so was therefore chosen to be superior to the complimenting approach.

For the system as a whole, a modular approach had been taken originally. This meant creating a block for each separate function or process to be performed on the video feed. This approach was originally taken, because it meant that the debugging of each separate Quartus block, and sub-system would be easier, as each one could be tested by itself. However, in terms of synchronization in the whole system, this provided another issue. If several Quartus blocks were to be used, this would require many of the Quartus blocks to be synchronised in order to ensure that processing would occur in the correct order, and performed on the correct data. To ensure this was the case throughout the project, some shift registers had to be included in locations, in order to 'delay' the data by one or two clock cycles. Luckily this was sorted out relatively early on in the process, because otherwise we could have had major issues later on.

When converting the original pixel values into greyscale, the simplest approach is taken. For each separate pixel in the frame, each colour component is added to a variable, normalized and then sent to

output. Taking 3 segments of red, green and blue, each are represented by 10 bits. These are then added together and divided by three. The outputs for each colour (RGB) are this averaged value, so that each colour is the same intensity. As greyscale had been used before by the group members, it was a simple task to implement this again. See appendix E for greyscale code.

For the further conversion of the greyscale, another block has been created which converts the greyscale image into a binary image. This block simply works by the use of a threshold, which in this case has been set to 800. The program checks to see if the pixel currently being processed is above or below the threshold. If it is above the threshold, the pixel will be given a value of 1023, which is completely black, and if it is below, it will be given a value of 0, which is white. This block is connected serially with the greyscale block, because otherwise it is not possible to compare the pixels to the threshold value. The threshold value was chosen to be taken as an input, rather than hard coded internally, in order to ensure enough flexibility in the future, to change this easily should the need arise. It turns out the value chosen of 800 was optimal for our solution and this didn't have to be altered. See appendix F for black and white conversion code.

One of the biggest obstacles encountered throughout the project was being able to understand and to implement the grass-fire algorithm sequentially. Had it been possible to do it recursively, then the function is simple enough, but because of the hardware limitations, it had to be programmed sequentially. The function takes a matrix of 0s and 1s as the input and then processes it. The algorithm checks each tile in the matrix sequentially until it finds a 1 (*See Figure 2)*. Once it finds a 1, the tile is set to zero ("burned"), and then goes on to check the neighbours of this tile. Neighbours are defined as the left, right, above and below. In addition, an array keeps track of how many tiles belong to an object, so once an object is found, the current index of the array is incremented by 1. If any neighbours also contain a value of 1, then they are burned, and added to a list to keep track of the next tile to be checked. The algorithm has two other variables. One keeps track of the total number of tiles on the list, and another keeps track of how much of the list has already been processed. This ensures that the same tile does not have its neighbours checked twice because the algorithm keeps on processing all the tiles in this list until the two variables are equal. Once this has happened, there are no more tiles that belong to this object, and the function then returns to check the rest of the image sequentially. Should another tile of value 1 be found, the variable that keeps track of the object number is incremented, signifying the start of a new object and the algorithm then continues accordingly.

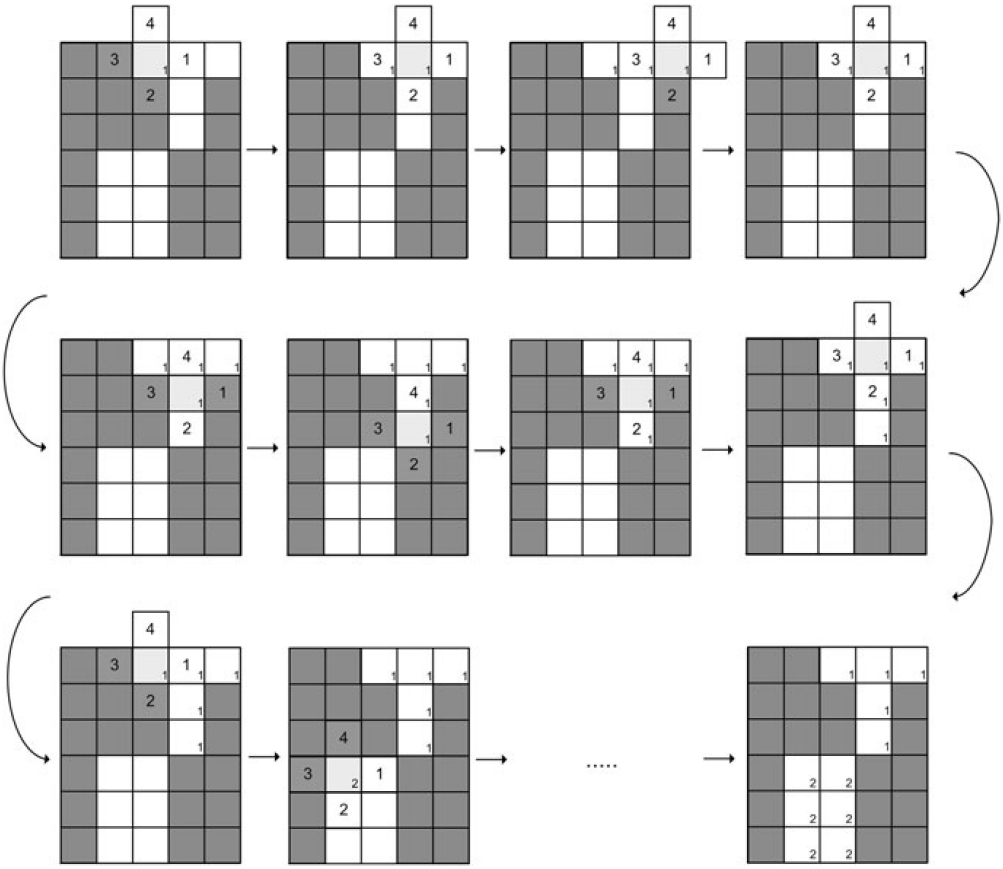


Figure 2 - Blob detection algorithm

For the purpose of programming in C++, the algorithm was programmed in a modular way in order to be able to debug the program easily. Once it had been asserted that the algorithm functioned, it was then recomposed so that only one function call had to occur once a new object had been found. This way, the operations used on branching could be reduced, thus increasing the efficiency of the algorithm. Once it had been determined that this was functioning as well, the code was translated into Catapult C.

Once this had been done, it was realized that several things could be optimized. First of all, the blob detection block was storing 10 bits per tile in the input matrix. This was completely unnecessary and an incredible waste of memory, as the matrix only needed 1 bit per tile to store the value of each tile (1 or 0). Therefore, the two-dimensional array holding the matrix was made to only consist of 1-bit values. In addition, the function originally utilized a structure, which consisted of 4 values representing the x and y coordinates in the matrix, the value of the tile, and the object the specific tile belonged to. This was removed as it was deemed unnecessary and the algorithm was shown to work regardless. Another issue was that during compiling, Catapult C would give an error saying that the allocated memory was too small compared to the requested memory. This was shown to be because some of the arrays exceeded the memory size of the 9K block-RAM on the FPGA. Therefore, the size of the arrays in the program were decreased so no single array would exceed the 9K bit size and the RAM single port libraries were included in Catapult C. This proved to resolve the memory issue that had been encountered. Before this was found, it was attempted to increase the register threshold in Catapult C, which also resolved the problem for compiling in Catapult C, however once brought into Quartus, and attempted compiled, the

compiling would never terminate, even if it had been running for many hours. These kind of challenges were unexpected when the project first was started, and this is something discussed later in the outcomes section of this report. In addition to reducing the bit size of the image matrix, the array size for the list holding the positions to be visited next was changed. Instead two separate arrays were created: one holding the x coordinate and the other holding the y coordinate. Each index in the x array just corresponds to the coordinate of the same index in the y array, and thus a tile is represented. The algorithm was tested with a hardcoded 64x64 matrix and was shown to work.

When programming the matrix to be copied from the cursor area, a variable was created which would only be one when the pixel was inside the cursor area. A variable called “enable copy” was created for this. This would allow the mouse control block to go through the entire frame, and only pass on the pixels that are within the cursor area, thus only copying those into the matrix, which the blob detection was to be performed on. Here, several problems were encountered, as it was not realized at first that the input would come in in a streaming fashion. With this in mind, it was necessary to change the architecture constraints, specifically the initiation interval to 1, in Catapult C. This way, the two inputs, pixel value and enable copy would be updated with every clock cycle, thus copying the matrix correctly from the cursor. See Appendix C for Blob Detection block code.

For the output of the final result it was debated whether to use the LEDs or the 7-segment display. It would have been easier to use the LEDs and it would have saved resources on the FPGA compared to the 7-segment display, as this requires a block in Quartus that converts the binary value into the correct numbers on the display. However, in the end it was deemed that the 7-segment display would make for a clearer output despite the extra resources it required. The reason behind this was because of the ease of reading the 7-segment display for a user compared to binary LEDs.

Another output that is used when the program is running is the VGA monitor. There are two different modes for the output; the first one is when the user does not click the mouse. In this case the user is shown a black and red image on which a sobel filter is applied. This shows the user the outlines of any coins that may be in the image, so that if the user wishes to analyse them, they can hover the cursor area over the coins (see Figure 3 left). The mouse and cursor are represented by the green square. Wherever the mouse is placed, a green and blue filter is applied to the binary image, which shows all black pixels as green and all white pixels as blue. When the mouse is clicked, the other output is shown. In this state raw camera feed pixels are shown, thus giving the user an overview of what they are actually looking at, rather than just showing the edge detection image (see Figure 3 right). The cursor remains the same so that the user is able to see that this indeed is the same image feed they are analysing. It can also clearly be seen here that the background is black as mentioned in the design specifications. It was chosen to use the Sobel filter as the default background image, because this simplified the display by only showing edges in the image. Sometimes, the light conditions can wash out the image or obscure details, making it hard to see what is going on. The Sobel filter just allows the user to see the coin outlines, which is much simpler. See Appendix G for the mouse code.

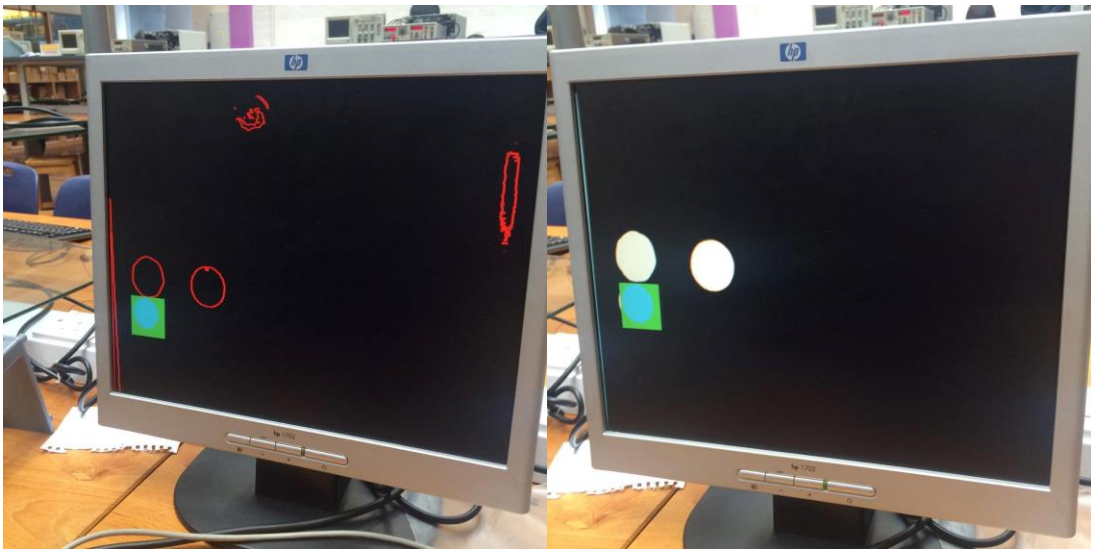


Figure 3 - Sobel filter (left) and raw camera image (right) user interfaces

In designing the stand, several things had to be taken into account. First and foremost, the structure had to be strong enough, such that it could support the weight of the FPGA and camera. In addition, the height had to be adjustable, in case specifications were going to change later in the project. Therefore, a modular design approach was taken. The stand would be created of three different components: A headpiece, which would hold the FPGA; A main structure piece which would be 10 cm tall, of which more could be added or removed in case the height had to be adjusted; and a base piece, which would have to be clamped to a table, in order to secure that the stand would not fall over.

All pieces of the stand were designed in 123D Design, by Autodesk, and printed on a Prusa i3 machine using grey PLA filament. Each piece took approximately 5 hours to print, using 0.3mm layer height, and 15% infill to ensure they could withstand the load. The base component was designed to be flat to sit on the table and be clamped using a G-clamp. It then had some pegs to mount the middle section of the stand and some supports to try and keep the stand stable. The middle section was designed to be very simple and modular. It consisted of two straight pieces with holes on the bottom and pegs on the top, so that it could fit into existing pieces and support further pieces on top. The middle had an X across it for structural support. Finally, the top piece consisted of a simple flat plane with mounting holes for the FPGA at the correct distances. This also contained some support to keep the FPGA camera from falling or the plane from bending under the stress. This also had the same mounting system so that it could be easily attached. See Appendix B for screenshots of the final 3D models.

When the structure first was assembled, there was an obvious issue with the headpiece design. Because the headpiece had not been designed to extend far enough out from the main structure, the camera was actually capturing the structure in the main feed as well. Therefore, the headpiece had to have an extra piece designed, which would ensure that the camera was extended even further away from the body of the structure. This was simple to design, and consisted of just a flat piece with the appropriate mounting positions. Luckily, due to the nature of 3D printing, this was able to be quickly designed, printed and tested.

When the structure was then reassembled, it bent under the heavy weight of the FPGA. It was noticed that because of the modularity of the structure and the fairly low tolerances of the 3D printer, each joint created a weak point, which had to be dealt with. The issue with this was not only that the structure was insecure, but also that the camera would then be capturing uneven frames of the table. This was because the headpiece would bend such that the camera was at an angle. This affected the pixel values of each coin, depending on where in the image they were. It was therefore decided to glue together every joint with a hot glue gun, in order to strengthen the structure. This quick fix helped immensely, and the structure would no longer bend. There was however one issue, as it was noticed that if the FPGA was left too long on the headpiece, the glue would wear out because it is not particularly strong, essentially making the structure bend more and more as time passed. It was therefore decided to have the FPGA only sit on the main structure, when the program was being tested. If necessary some extra supports could have been added in across joints, and this would be considered for a more long-term solution.

8 FINAL DESIGN SELECTION

During the development of the system a few changes were made to the original design in order to improve its functionality.

One of the main issues which arose at an early stage during testing, regarded the size of the mouse window. As can be seen in Figure 3 the area that the user can select on the screen using the mouse was limited to 64x64 pixels and did not allow the detection of more than one coin at the same time. This posed a problem as the mouse window had to be increased, affecting the rest of the system and especially the blob detection algorithm, designed to work on a 64x64 matrix.

Since the size of the blob detection algorithm matrix could not be increased due to memory constraints (problems increasing the amount of RAM used on the FPGA, discussed in the outcomes section), downscaling was proposed as a solution.

The original mouse block was modified to include a region of 128x128 pixels as can be seen in Figure 4, and a downscaling control block was implemented. The blob detection block was originally controlled by the mouse block, which provided a high signal whenever the pixel currently being processed was in the mouse area and had to be stored in memory. In order to perform a downscaling, the control block was designed to provide a high signal when the pixel processed is both in the mouse area and has even coordinates: the odd rows and columns of the 128x128 pixel region are therefore skipped so that a quarter of the original pixels are stored in the blob detection matrix to be analysed and counted. The downscaling control block with its inputs and outputs can be seen in Figure 5. The block uses the coordinates of the mouse cursor, the VGA X and Y as well as the size of the mouse cursor to calculate whether to provide a high or a low input to the blob detection block (code included in Appendix H).

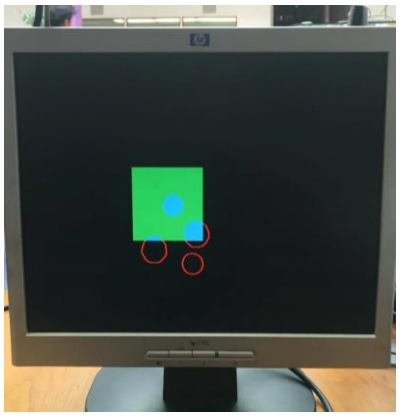


Figure 4 - 128x128 mouse region

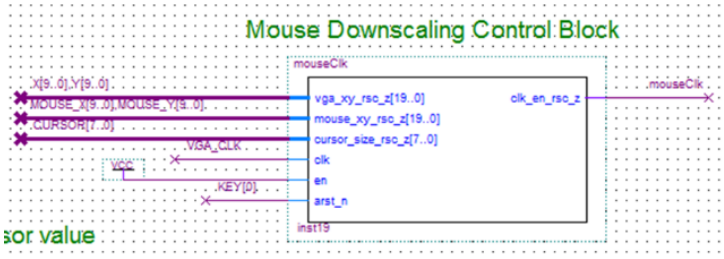


Figure 5 - Downscaling block

Another change made to the system involved modifying the path each pixel goes through before reaching the VGA monitor. Originally the pixels were stored in a shift register before going through the mouse and the blob detection block. This is because one of the early implementations of the blob detection algorithm utilised a shift class to store the pixels in registers to be elaborated. When RAM was used instead of registers, pixels were fed individually and not in groups as before. The use of a shift register was therefore not required and was eliminated from the system. A smaller implementation of the shift register (3 pixels wide) was kept in order to drive the Sobel filter in the user interface. The substitution of the shift register with a direct flow of pixels greatly reduced the latency of the system, which was able to perform calculations without waiting for each pixel to be stored.

During the final implementation of the project all the blocks described in the previous sections were developed and added to the digital design. The implementation of some of these blocks, which was not described before, will now be explained to make their functionality clear.

As can be seen in Figure 6 the user can select the region of the screen of interest using the mouse and can see the total value displayed on the 7-segment display embedded on the FPGA. The blob detection block can just count the areas of a maximum of ten blobs in the mouse area, therefore the area values have to be matched to their corresponding coin amount and summed together. To perform this operation an area calculation block, which can be seen in Figure 7 was defined. This block gets in input each of the ten area values and returns the total amount of coins in the mouse region in binary.

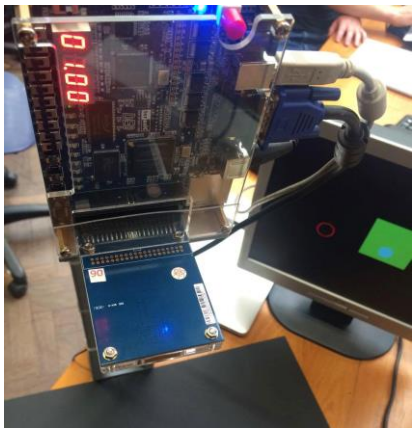


Figure 6 - Selection screen and 7-segment displays

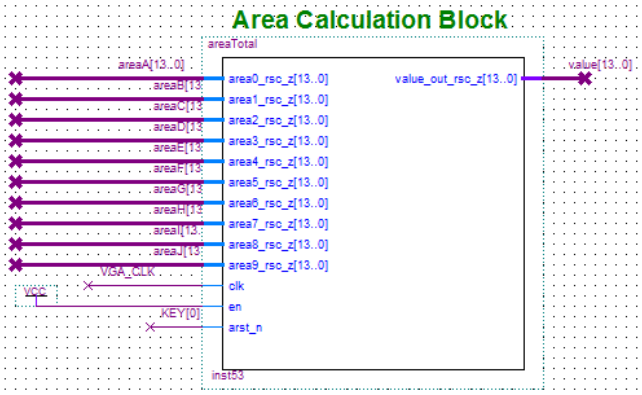


Figure 7 - Area detection block

The area calculation block was developed in Catapult C using ranges that were measured individually, as can be seen in the grid of Figure 8. Ten measurements were taken of each coin in exactly the same conditions, and we took note of the lowest, highest and average value. This then determined which coins were to be implemented, because the ranges had to be far enough apart and not overlapping in any case. £2 was not used, as it was almost the exact same size as the 2p and 50p coin. The threshold values were hardcoded in the area calculation program; the program compares the values of each of the ten areas with the ones recorded to identify the corresponding coin value and once each comparison has been completed the total value is copied to the output. See appendix D for Area Calculation code.

The output provided by the area calculation block is in binary and therefore has to be converted to binary coded decimal to be able to be shown on the 7 segment display and be intelligible to the user. To do this a binary to BCD (binary coded decimal) block was written in Verilog to perform the required operation. We sourced the code for this block online in order to save time, as this was a simple block and we needed to keep time for the more complicated aspects of the design. The code used was only for a number showing hundreds, tens, and units, but the 7 segment display has four digits, so we replicated the pattern of the program to expand the code for the thousand-digit output.

Furthermore, the BCD value has to be decoded to be displayed on the seven-segment display. To do so four BCD to 7-Segment Decoders were used that were already in the base Quartus project provided. The logic of these blocks was written in Verilog.

Coin Value (pence)	Area Value(pixels)												
	1	2	3	4	5	6	7	8	9	10	Lowest Value	Highest Value	Average
1	232	221	210	222	207	250	234	255	226	240	207	255	229.7
2	376	353	329	336	326	359	362	380	364	371	326	380	355.6
5	190	182	193	174	170	188	184	194	169	186	169	194	183
10	347	330	300	310	310	340	334	357	323	347	300	357	329.8
20	270	246	235	257	243	262	254	273	250	273	235	273	256.3
50	475	436	384	400	383	428	455	460	420	410	383	475	425.1
100	305	284	250	279	276	291	290	308	282	307	250	308	287.2
200											0	0	#DIV/0!

Figure 8 - Excel grid

9 Outcomes

The finished design functioned correctly as a coin detector, showing the coins on the screen using a Sobel filter. The mouse window shows up in green and objects within this region are shown as blue. The user can click the mouse to change the background from the Sobel filter to the raw camera feed to get a better idea of what is being looked at, or to check the light conditions as seen by the camera. Up to 10 coins within the mouse region can be added up, and the total value is displayed on the 7-segment display. The switches on the FPGA can be used to change the exposure of the camera to help troubleshoot the system. One of the switches also changes the display from showing the total value of the coins, to the area of the first object, which can be used to help calibrate the system. This solution is similar to the original idea, but with some key differences, mainly due to the limited FPGA resources and the strict time constraints of the project.

Originally the group wanted to be able to use the switches on the FPGA to change the mouse window size, enabling different amounts of coins to be analysed. However, later during the project the limitations of the FPGAs resources were discovered. The DE0 only has 56 M9K embedded RAM blocks, with an additional 8-Mbyte of external SDRAM and 4-Mbyte of external Flash. The group discovered the external SDRAM and Flash was difficult to use, and many other groups had already spent many hours on this, but to no avail. Flash memory was also slower, so with the chance that it may not be suitable for the real time image processing application, it was decided against. This limited the group to using the on board M9K embedded RAM blocks. The problem with this was that in the code, the FPGA could only have arrays up to a size of 9000 bits, because this was the maximum that could be contained within one RAM block. Once arrays exceeded this size, Catapult C wouldn't know how to manage the memory across different memory blocks in order to see it as one array. Without knowing this, the code was left to compile for hours but with no success, and was in the end terminated without knowing if it could have worked or not. It was at this point a new strategy was needed. Up to this point, the group had defined a structure in C to contain the value of each pixel and its x and y co-ordinates. A 64x64 array had then been created to hold these values, representing the mouse window. Each pixel value was represented by a 10-bit number, and each co-ordinate required 6 bits for x and y ($2^6 = 64$), giving a total array size of 90,112 bits, covering 10 RAM blocks. This was obviously a major problem in the design so this needed to be massively simplified. The 10-bit pixel value previously only represented 0 or 1023 for white and black, but this was only to be compatible with the VGA display. The algorithm could work just as easily on 0s and 1s, which would represent white and black respectively. It was also realised that it was unnecessary to store the x and y co-ordinate for each pixel. Instead this could be found by just taking the current iteration of the for-loops within the code. Each time the loop incremented, the algorithm was working on a new pixel, so the loop iteration could therefore represent the co-ordinates of the pixel within the 64x64 array. This hugely simplified the code and the memory space needed, as now only 1 bit was needed for each pixel value, for a total of 4096 bits. This was well within the space of one RAM block. This however limited the size of the mouse window that could be analysed. When it was attempted to use a 128x128 array, giving an array size of 16,384 bits, this was shown to be too large, giving the same compilation issues. The largest square array that could be fit inside one RAM block was 94x94, giving a size of 8,836 bits. However, it was decided to keep the 64x64 as this simplified the implementation, because this was easier to downscale from the 128x128 mouse window size.

Another part of the project that was changed was the number of different coins that could be detected. Originally the group wanted to be able to detect all the different Sterling coins, but later it was realised this was not feasible. When measuring the coin areas to determine thresholds, it became obvious that most of the areas were very similar. The poor quality of the FPGA camera, and the changing light conditions also meant that large threshold ranges had to be kept each coin, which lead to some of these ranges overlapping. It was therefore decided to get the system working perfectly for a few coins, rather than trying to detect all of them with little success. It was therefore chosen to detect 3 different coins in total: 5p, 50p and £1. Given more time and better resources, it might have been possible to implement other techniques to distinguish the coins and allow the FPGA to detect them all.

10 Future Work and Improvements

There are a number of possible improvements that could be made. The coin detector as said before can recognize three different coins (5p, 50p, £1); one of the first improvements that could be made is to extend the coins detected to all Sterling Pound coins. This would certainly require some modifications to the system as the area values of the coins are very similar: a more sensible measurement system would have to be implemented in order to recognize coins similar in size. Another feature to be implemented could be the recognition of notes. In order to add this functionality either the mouse area should be increased or the distance between the platform and the camera should be increased; the notes would not fit in the mouse region otherwise.

In addition, the coin detector could also be extended to other currencies. The switches embedded in the FPGA could be used to change the mode of operation of the system and load the area values of different currencies. The new area values would have to be hardcoded into separate blocks, which are enabled and disabled according to the mode the system is in; this could be achieved by connecting select lines to the switches, which would enable different blocks depending on which switches are switched on.

Another improvement could be further increasing the cursor area. Unfortunately, the group experienced some issues with the development tool, and was therefore unable to have arrays that were larger than 9K bits, as described above. With the tool working correctly however, it should be easy to increase the cursor area in order to detect more coins. With this, the blob detection block would also have to be changed, such that more than ten different objects can be detected. This is however a rather easy process as it only requires the array which stores the object sizes, to be increased.

An improvement that could be made to the project if more time was available would be calculating area of the coins as a function of radius. One issue, which is removed by the specs, is that no other objects with similar areas can be present in the camera range, which is not very likely to happen in a real-life situation. If there was a function that determined the radius, the system would be able to ignore the objects which do not have a circular shape. In addition, another improvement would be to extend the blob detection so that it not only checks the four nearest tiles around a tile that has just been “burned”, but also checks the four corners for a total of 8 tiles around the tile that currently is being checked. This would make the system more precise and resilient to noise.

It would also be possible to create a setting for the camera in order to stabilize light conditions. Making a box, which surrounds the camera thus ensuring that light conditions almost always remain the same, could do this. This would mean that the ranges wouldn’t have to be as large for each coin, and thus making it possible to have even more coins, as the ranges would no longer overlap for each coin.

Furthermore, the blob detection system could be applied to solve different problems which do not necessarily involve the detection of coins: every application, which involves the distinction of objects based on their area value could present an opportunity to use the blob detection system.

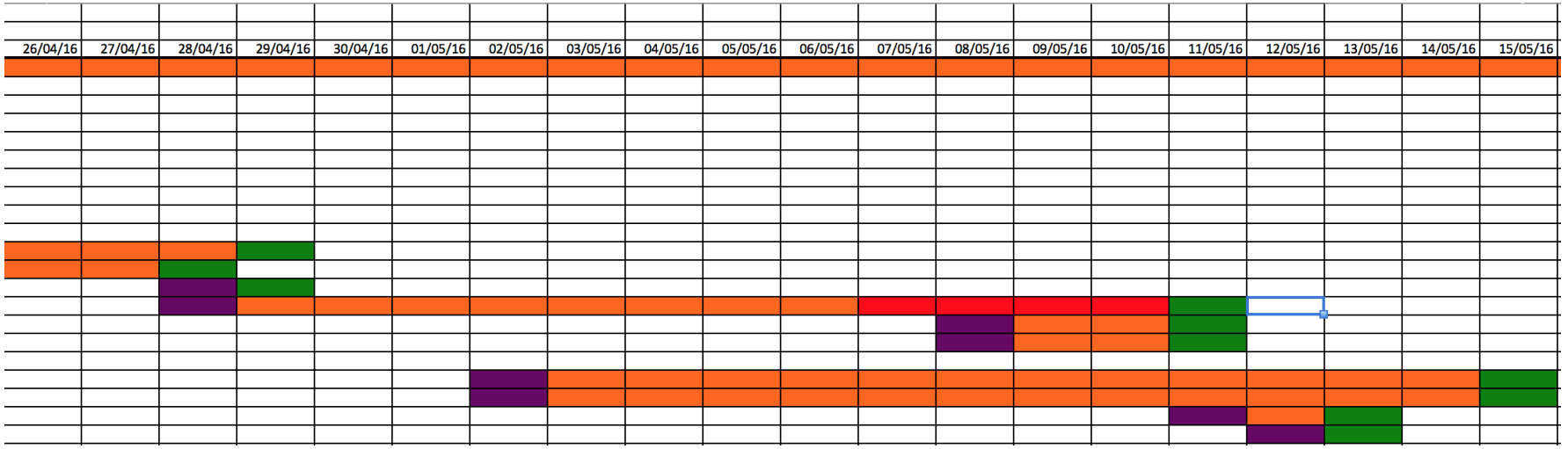
REFERENCES

1. What-when-how. *BLOB Analysis (Introduction to Video and Image Processing) Part 1.* Available from: http://what-when-how.com/introduction-to-video-and-image-processing/blob-analysis-introduction-to-video-and-image-processing-part-1/ [Accessed 6th May 2016].

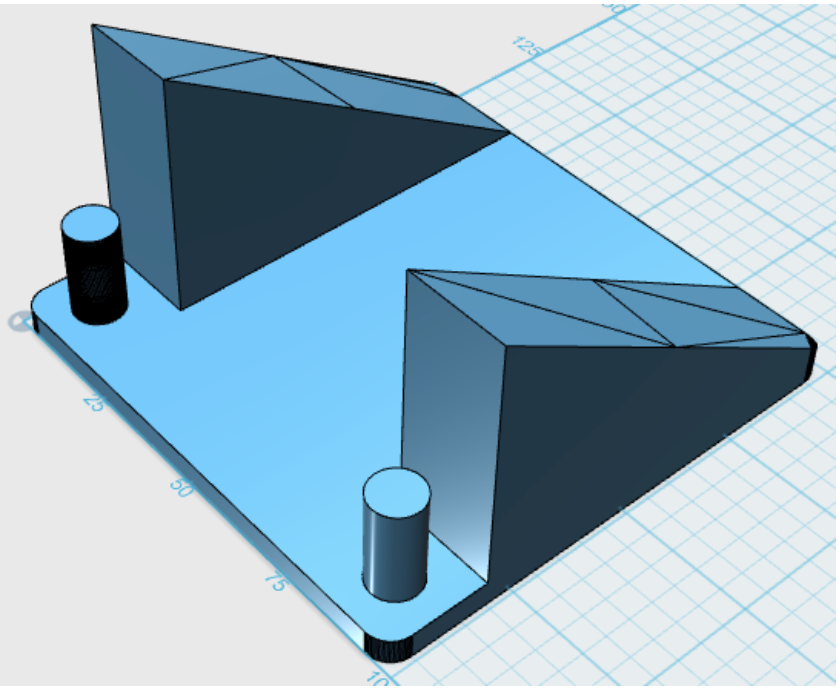
2. Divya Kiran, Abdul Imran Rasheed & Hariharan Ramasangu. *FPGA Implementation of Blob Detection Algorithm for Object Detection in Visual Navigation.* Scientific paper. M.S. Ramaiah School of Advanced Studies; 2013.

3. Engineering Utah. *Binary to BCD Conversion Algorithm.* Available from: http://www.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html/ [Accessed 15th May 2016]

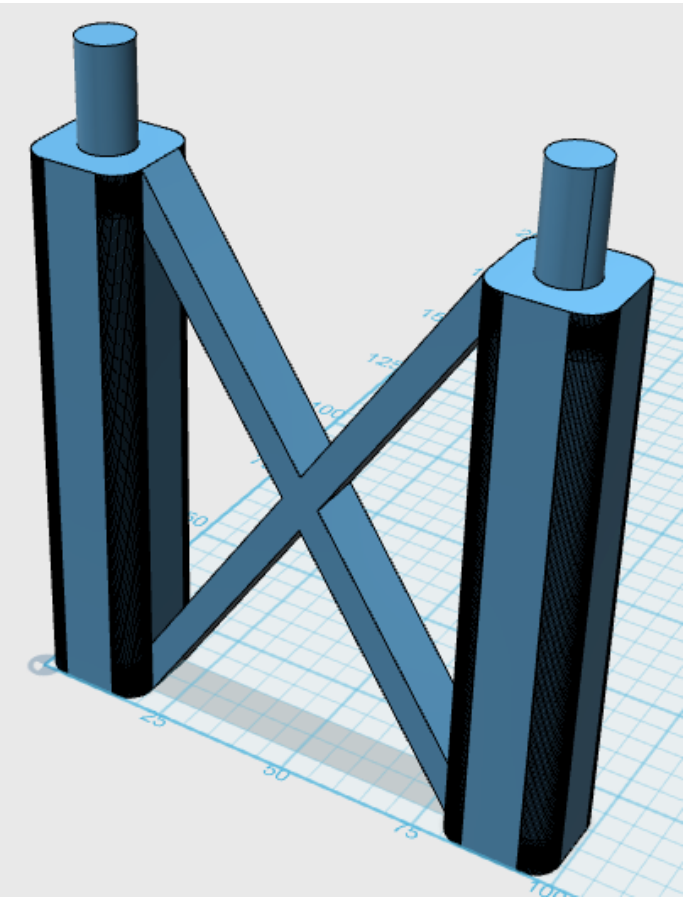
KEY																		
COMPLETE		IN PROGRESS				LATE			RESCHEDULED			UPCOMING						
Name of Task	Duration	Start	Finish	Resources					Group member to fulfill	23/03/16	24/03/16	25/03/16	26/03/16	27/03/16	28/03/16	25/04/16
Gant chart	All of project	23/03/16	16/05/16	Laptop or computer					Mikkel							
Interim report: Project	4 days	23/03/16	27/03/16	Laptop or computer, guidelines, & online resources					Mikkel							
Interim report: Catapult C exercises	4 days	23/03/16	27/03/16	Laptop or computer, lab booklet, & lab notes					Marco & Thomas							
Interim Report: revision and formatting	2 days	27/03/16	28/03/16	Report & guidelines					All members							
Interim Report	5 days	23/03/16	28/03/16	Online submission					All members							
Start of project	1 day	25/04/16	25/04/16	EE Department Lab					All members							
Code for binary image conversion	1 day	25/04/16	25/04/16	EE Department Lab					Marco							
Code for greyscale conversion	1 day	25/04/16	25/04/16	EE Department Lab					Marco							
Determining camera distance	1 day	25/04/16	25/04/16	EE Department Lab					Mikkel & Thomas							
Designing stand for FPGA	1 day	25/04/16	25/04/16	EE Department Lab					Thomas & Mikkel							
Printing 3D stand for Camera	4 days	26/04/16	29/04/16	Thomas' 3D printer					Thomas							
Code for Grass-Fire Algorithm in C++	3 days	26/04/16	28/04/16	Laptop or computer					All members							
Code for Grass-Fire Algorithm converted to Catapult C	1 day	29/04/16	29/04/16	EE Department Lab					All members							
Debugging	5 days	29/04/16	06/05/16	EE Department Lab					Thomas & Marco							
Determining coin area	1 day	07/05/16	07/05/16	EE Department Lab					All members							
Code for coin comparison	1 day	07/05/16	07/05/16	EE Department Lab					Marco & Thomas							
General debugging and administrational time	2 days	11/05/16	12/05/16	EE Department Lab					All members							
Final report: Process (50% of prof Eng)	13 days	04/05/16	16/05/16	Laptop or computer, lab booklet, & lab notes					All members							
Final report: Outcomes	13 days	04/05/16	16/05/16	Laptop or computer, lab booklet, & lab notes					All members							
Presentation (30% of prof Eng)	2 days	12/05/16	13/05/16	Laptop or computer, lab booklet, & lab notes					All members							
Demonstration (Individual oral)	1 day	13/05/16	13/05/16	Laptop or computer, lab booklet, & lab notes					All members							



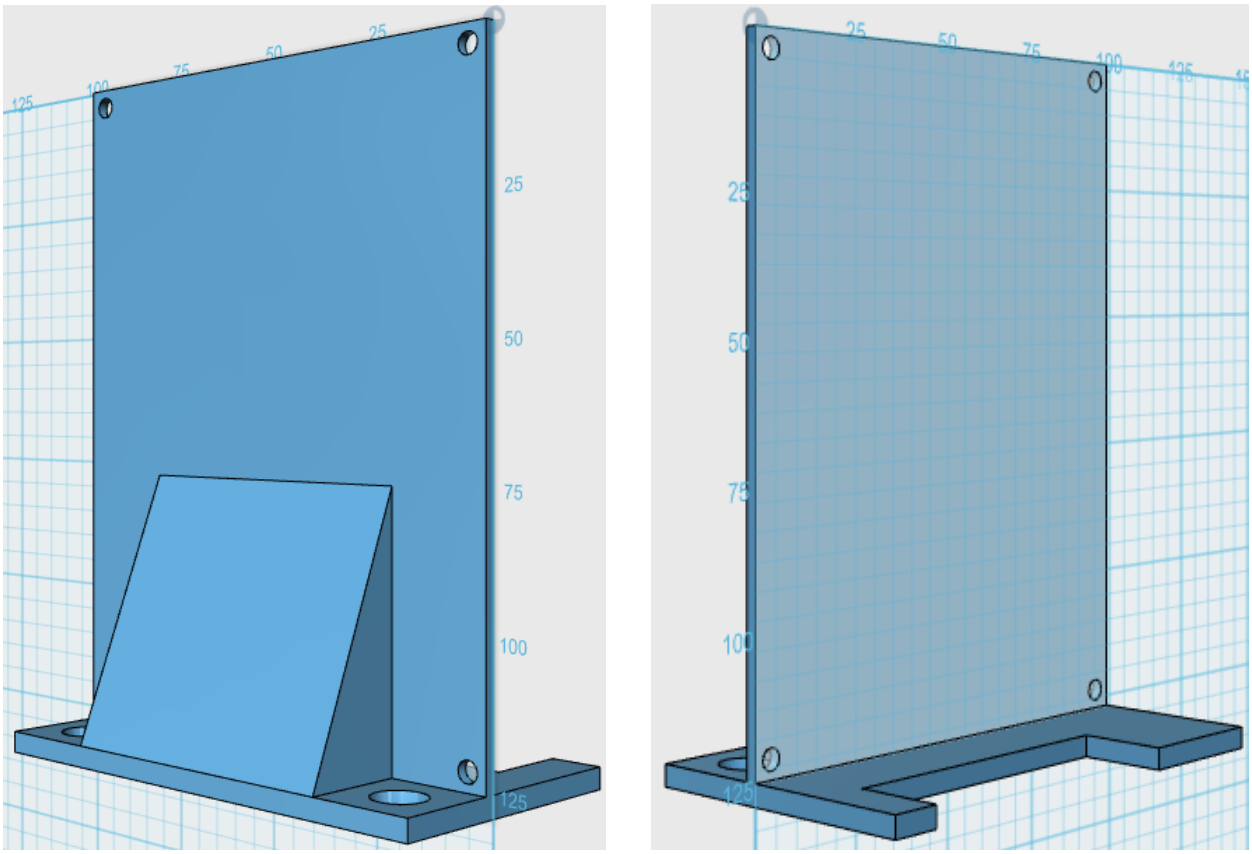
APPENDIX B



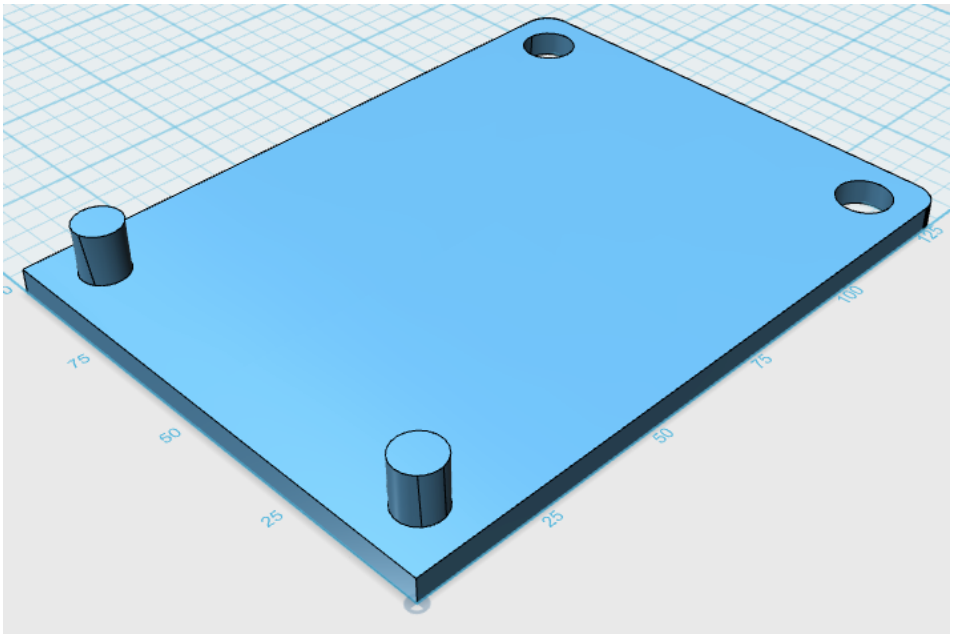
3D Design 1 - Bottom piece of stand



3D Design 2 - Middle piece of stand



3D Design 3 - Top of stand from two different angles



3D Design 4 - Extension piece of stand, designed and printed later

APPENDIX C

```
////////////////////////////////////////////////////////////////////////////////
//  File:           blob_det.cpp
//  Description:    Detects 10 blobs in a 64x64 matrix
//  By:             MTM
////////////////////////////////////////////////////////////////////////////////

#include <ac_fixed.h>
#include "blob_det.h"
#include <iostream>
#define N 4096
#define M 64
#define T 63

void blob_check(ac_int<1, false> value[M][M], int x, int y,int& listSize, ac_int<6, false> xList[N], ac_int<6, false> yList[N])
{
    if(x !=0 && value[x -1][y] == 1){
        xList[listSize] = x;
        yList[listSize] = y;
        listSize++;
        value[x -1][y] = 0;
    }
    if(y !=T && value[x][y +1] == 1){
        xList[listSize] = x;
        yList[listSize] = y+1;
        listSize++;
        value[x][y +1] = 0;
    }
    if(x !=T && value[x +1][y] == 1){
        xList[listSize] = x +1;
        yList[listSize] = y;
        listSize++;
        value[x +1][y] = 0;
    }
    if(x !=0 && value[x][y-1] == 1){
        xList[listSize] = x;
        yList[listSize] = y-1;
        listSize++;
        value[x][y-1] = 0;
    }
}

#pragma hls_design top

void blob_det(ac_int<1,false>& vin, ac_int<1,false>& enable_copy, ac_int<1,false>& startFrame,static ac_int<14,false> *area0,
    static ac_int<14,false> *area1,static ac_int<14,false> *area2,static ac_int<14,false> *area3,static ac_int<14,false> *area4,
    static ac_int<14,false> *area5,static ac_int<14,false> *area6,static ac_int<14,false> *area7,static ac_int<14,false> *area8,
    static ac_int<14,false> *area9)
{


    ac_int<1, false> value[M][M];
    static ac_int<14,false> sums[10];
    ac_int<6, false> xList[N], yList[N];
    ac_int<14, false> total = 0;
    int listSize = 0,counter = 0,sumsIndex = 0, i = 0, j =0, x = 0;
    static ac_int<1,false> g = 0, reset = 0;

#if 1
    if(startFrame){

    FRAME:  for(int p = 0; p < NUM_PIXELS; p++) {
            if(enable_copy){
                value[j][i] = vin;
                total++;
                if(i==T){
                    i = 0;
                    j++;
                }
                else
                    i++;
            }
        }


    RESET: for(int z = 0; z < 10 && total == 4096; z++){
            sums[z] = 0;
            if(z == 9){
                 reset = 1;
                 total = 0;
                 }
            else
                reset = 0;
        }


    BLOB_DET:   for(x = 0; x<M && reset; x++){
            for(int y = 0; y<M; y++){
                if(value[x][y] == 1){
                    value[x][y] = 0;
                    sums[sumsIndex] = 1;
                    blob_check(value, x, y, listSize, xList, yList);
                    sumsIndex++;
                }
                while(counter < listSize){
                    blob_check(value,xList[counter], yList[counter],listSize,xList,yList);
                    counter++;
                    sums[sumsIndex-1]++;
```

```
                }
            }
            if(x == M-1){
                g = 1;
                reset = 0;
                }
            else
                g = 0;
        }

        if (g){
            *area0 = sums[0];
            *area1 = sums[1];
            *area2 = sums[2];
            *area3 = sums[3];
            *area4 = sums[4];
            *area5 = sums[5];
            *area6 = sums[6];
            *area7 = sums[7];
            *area8 = sums[8];
            *area9 = sums[9];
        }

    }
}
```

Code 1 - Blob detection block

APPENDIX D

```
#include "ac_int.h"
#define minB 100
#define maxB 220
#define minC 350
#define maxC 600
#define minD 230
#define maxD 340

#pragma hls_design top
void areaTotal(ac_int<14, false> area0, ac_int<14, false> area1,ac_int<14, false> area2,ac_int<14, false> area3,
    ac_int<14, false> area4,ac_int<14, false> area5,ac_int<14, false> area6,ac_int<14, false> area7,ac_int<14,
    false> area8,ac_int<14, false> area9,static ac_int<14, false> *value_out)
{
    ac_int<14, false> tot_value = 0;
    int i = 0;

    if(minB<area0 && area0<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area0 && area0<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area0 && area0<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////


    if(minB<area1 && area1<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area1 && area1<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area1 && area1<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;
```

```
    ////////////////////////////////////////////////////

    if(minB<area2 && area2<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area2 && area2<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area2 && area2<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area3 && area3<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area3 && area3<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area3 && area3<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area4 && area4<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area4 && area4<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area4 && area4<maxD){
        tot_value += 100;
        i++;
    }
```

```
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area5 && area5<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area5 && area5<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area5 && area5<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area6 && area6<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area6 && area6<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area6 && area6<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area7 && area7<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area7 && area7<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area7 && area7<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area8 && area8<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area8 && area8<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area8 && area8<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(minB<area9 && area9<maxB){
        tot_value += 5;
        i++;
    }
    else if(minC<area9 && area9<maxC){
        tot_value += 50;
        i++;
    }
    else if(minD<area9 && area9<maxD){
        tot_value += 100;
        i++;
    }
    else
        i++;

    ////////////////////////////////////////////////////

    if(i == 10){
        *value_out = tot_value;
        i = 0;
    }
}
```

Code 2 – Area calculation detection block

APPENDIX E

```
////////////////////////////////////////////////////////////////////////////////
//  File:           greyscale.cpp
//  Description:
//  By:             MTM
////////////////////////////////////////////////////////////////////////////////

#include <ac_fixed.h>
#include "blackwhite.h"
#include <iostream>

// shift_class: page 119 HLS Blue Book
#include "shift_class.h"

#pragma hls_design top
void blackwhite(ac_int<PIXEL_WL,false> vin[NUM_PIXELS], ac_int<10, false> threshold, ac_int<PIXEL_WL,false> vout[NUM_PIXELS])
{
    ac_int<16, false> fin;

// #if 1: use filter
// #if 0: copy input to output bypassing filter

    // shifts pixels from KERNEL_WIDTH rows and keeps KERNEL_WIDTH columns (KERNEL_WIDTHxKERNEL_WIDTH pixels stored)
    static shift_class<ac_int<PIXEL_WL*KERNEL_WIDTH,false>, KERNEL_WIDTH> regs;
    int i = 0;

    FRAME: for(int p = 0; p < NUM_PIXELS; p++) {
        // init
        fin = 0;

        // shift input data in the filter fifo
        regs << vin[p]; // advance the pointer address by the pixel number (testbench/simulation only)
        // accumulate
        ACC1:
            // current pixel
            fin += (regs[i].slc<COLOUR_WL>(0));

            if(fin> threshold){
                fin = 1023;
            }
            else{
                fin = 0;
            }

        // group the RGB components into a single signal
        vout[p] = ((((ac_int<PIXEL_WL, false>)fin) << (2*COLOUR_WL)) | (((ac_int<PIXEL_WL, false>)fin) << COLOUR_WL) | (ac_int<PIXEL_WL, false>)fin);

    }
}

// end of file
```

Code 3 – Greyscale conversion block

APPENDIX F

```
////////////////////////////////////////////////////////////////////////////////
//  File:           greyscale.cpp
//  Description:
//  By:             MTM
////////////////////////////////////////////////////////////////////////////////

#include <ac_fixed.h>
#include "blackwhite.h"
#include <iostream>

// shift_class: page 119 HLS Blue Book
#include "shift_class.h"

#pragma hls_design top
void blackwhite(ac_int<PIXEL_WL,false> vin[NUM_PIXELS], ac_int<10, false> threshold, ac_int<PIXEL_WL,false> vout[NUM_PIXELS])
{
    ac_int<16, false> fin;

// #if 1: use filter
// #if 0: copy input to output bypassing filter

    // shifts pixels from KERNEL_WIDTH rows and keeps KERNEL_WIDTH columns (KERNEL_WIDTHxKERNEL_WIDTH pixels stored)
    static shift_class<ac_int<PIXEL_WL*KERNEL_WIDTH,false>, KERNEL_WIDTH> regs;
    int i = 0;

    FRAME: for(int p = 0; p < NUM_PIXELS; p++) {
        // init
        fin = 0;

        // shift input data in the filter fifo
        regs << vin[p]; // advance the pointer address by the pixel number (testbench/simulation only)
        // accumulate
        ACC1:
            // current pixel
            fin += (regs[i].slc<COLOUR_WL>(0));

            if(fin> threshold){
                fin = 1023;
            }
            else{
                fin = 0;
            }

        // group the RGB components into a single signal
        vout[p] = ((((ac_int<PIXEL_WL, false>)fin) << (2*COLOUR_WL)) | (((ac_int<PIXEL_WL, false>)fin) << COLOUR_WL) | (ac_int<PIXEL_WL, false>)fin);

    }
}


// end of file

```

Code 4 – Black & white conversion block

APPENDIX G

```
////////////////////////////////////////////////////////////////////////////////
// Catapult Project options
// Constraint Editor:
//  Frequency: 27 MHz
//  Top design: vga_mouse
//  clk>reset sync: disable; reset async: enable; enable: enable
// Architecture Constraint:
//  core>main: enable pipeline + loop can be merged
////////////////////////////////////////////////////////////////////////////////



#include "stdio.h"
#include "ac_int.h"

#define COLOR_WL          10
#define PIXEL_WL          (3*COLOR_WL)

#define  COORD_WL          10

#pragma hls_design top
void mouse_new(ac_int<(COORD_WL+COORD_WL), false> * vga_xy, ac_int<(COORD_WL+COORD_WL), false> * mouse_xy,
    ac_int<(8), false> cursor_size, ac_int<1, false> * clk_en, ac_int<PIXEL_WL, false> * video_in, ac_int<
    PIXEL_WL, false> * video_out)
{
    ac_int<10, false> i_red, i_green, i_blue; // current pixel
    ac_int<10, false> o_red, o_green, o_blue; // output pixel
    ac_int<10, false> mouse_x, mouse_y, vga_x, vga_y; // mouse and screen coordinates


/*  --extract the 3 color components from the 30 bit signal--
    the 2 blocks are identical - you can shift and mask the desired bits or "slice" the signal
        <length>(location)

    i_red = *video_in >> 20;
    i_green = (*video_in >> 10) & (ac_int<10>)1023;
    i_blue = *video_in & ((ac_int<10>)1023);
*/
    i_red = (*video_in).slc<COLOR_WL>(20);
    i_green = (*video_in).slc<COLOR_WL>(10);
    i_blue = (*video_in).slc<COLOR_WL>(0);

    // extract mouse X-Y coordinates
    mouse_x = (*mouse_xy).slc<COORD_WL>(0);
    mouse_y = -(*mouse_xy).slc<COORD_WL>(10);
    // extract VGA pixel X-Y coordinates
    vga_x = (*vga_xy).slc<COORD_WL>(0);
    vga_y = (*vga_xy).slc<COORD_WL>(10);


    if ((vga_x > mouse_x - cursor_size) && (vga_x <= mouse_x + cursor_size) && (vga_y > mouse_y - cursor_size)
        && (vga_y <= mouse_y + cursor_size)){
        // if it is inside the mouse square
        o_red = 400;
        o_green = 800;
        o_blue = i_blue;
        *clk_en = 1;
    }
    else {
        // if it is outside the mouse square
        o_red = i_red;
        o_green = i_green;
        o_blue = i_blue;
        *clk_en = 0;
    }

    *video_out = ((((ac_int<PIXEL_WL, false>)o_red) << 20) | (((ac_int<PIXEL_WL, false>)o_green) << 10) |
        (ac_int<PIXEL_WL, false>)o_blue);
}

```

Code 5 – VGA mouse control block

APPENDIX H

```
// Catapult Project options
// Constraint Editor:
//  Frequency: 27 MHz
//  Top design: vga_mouse
//  clk>reset sync: disable; reset async: enable; enable: enable
// Architecture Constraint:
//  core>main: enable pipeline + loop can be merged
////////////////////////////////////////////////////////////////////////////////



#include "stdio.h"
#include "ac_int.h"

#define COLOR_WL          10
#define PIXEL_WL          (3*COLOR_WL)

#define  COORD_WL          10

#pragma hls_design top
void mouseClk(ac_int<(COORD_WL+COORD_WL), false> * vga_xy, ac_int<(COORD_WL+COORD_WL), false> * mouse_xy, ac_int
    <(8), false> cursor_size, ac_int<1, false> * clk_en)
{

    ac_int<10, false> mouse_x, mouse_y, vga_x, vga_y; // mouse and screen coordinates

    // extract mouse X-Y coordinates
    mouse_x = (*mouse_xy).slc<COORD_WL>(0);
    mouse_y = -(*mouse_xy).slc<COORD_WL>(10);
    // extract VGA pixel X-Y coordinates
    vga_x = (*vga_xy).slc<COORD_WL>(0);
    vga_y = (*vga_xy).slc<COORD_WL>(10);

    if ((vga_x > mouse_x - cursor_size) && (vga_x <= mouse_x + cursor_size) && (vga_y > mouse_y - cursor_size) &
        & (vga_y <= mouse_y + cursor_size)){
        // if it is inside the mouse square
        if(vga_x%2 == 0 && vga_y%2 == 0){
            *clk_en = 1;
        }
        else{
            *clk_en = 0;
        }
    }
    else {
        // if it is outside the mouse square
        *clk_en = 0;
    }
}
```

Code 6 – Downscaling mouse control block